Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein

ÚJ ALGORITMUSOK

Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein

ÚJ ALGORITMUSOK

Scolar Kiadó

Budapest, 2003

Az eredeti mű:
T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to Algorithms.*
(Fourth printing (plus corrections up September 16, 2003).
The MIT Press/McGraw Hill, Cambridge/Boston, 2001.
XXI + 1180 oldal, ISBN 0-262-03293-7.

Alkotó szerkesztő: Iványi Antal

Fordítók: Benczúr András Jr. (25. fejezet), Burcsi Péter (17.), Csörnyei Zoltán (18–21.), Fekete István (10.), Gregorics Tibor (23–24.), Hajdú András (8.), Horváth Gyula (13–14., 16.), Ispány Márton (C), Iványi Anna (1–2., 35.), Kása Zoltán (7., 27.), Kovács Attila (31.), Lencse Zsolt (9.), Marx Dániel (3–5.), Nagy Sára (11–12.), Schipp Ferenc (30.), Sike Sándor (22., 32.), Simon Péter (A–B), Szegő László (26.), Szili László (28.), Veszprémi Anna (6., 29.), Vida János (33.), Vizvári Béla (15.), Wiener Gábor (34.)

Lektorok: Benczúr András (20–21.), Csirik János (27., 32–35.), Fábián Csaba (29.), Frank András (22–26.), Kátai Imre (28., 30–31.), Kiss Attila (18–19.), Kormos János (6–9.), Recski András (1–5.), Schipp Ferenc (A–C), Szántai Tamás (15–17.), Varga László (10–14.)

ISBN: 963 9193 90 9

Kiadja a Scolar Kiadó
1114 Budapest, Bartók Béla út 7.
Telefon/fax: 466-76-48
Honlap: http://www.scolar.hu
Elektronikus cím: scolar@scolar.hu

Felelős kiadó: Érsek Nándor
Technikai szerkesztő: Szabó Béla
Borítóterv: Liszi János

Nyomás és kötés: Dürer Nyomda, Gyula

Tartalomjegyzék

Előszó

A szerzők előszava az angol nyelvű kiadáshoz

Ez a könyv átfogó bevezetést nyújt a számítógépes algoritmusok modern felfogású tanulmányozásához. Habár nagyszámú algoritmust mutat be, és jelentős mélységben tárgyalja őket, tervezésük és elemzésük mégis érthető a különböző szintű olvasók széles köre számára. Arra törekedtünk, hogy a tárgyalásmód egyszerű legyen anélkül, hogy feláldoznánk a feldolgozás mélységét vagy a matematikai igényességet.

Minden fejezet bemutat egy algoritmust, tervezési módszert, alkalmazási területet vagy ezekhez kapcsolódó témát. Az algoritmusokat angolul és egy olyan „pszeudokódban" írjuk le, amelyet úgy terveztünk, hogy a kevés programozási tapasztalattal rendelkező olvasó számára is érthető legyen. A könyv több mint 230 ábrát tartalmaz, amelyek bemutatják, hogyan működnek az algoritmusok. Mivel hangsúlyozzuk a *hatékonyságot* mint tervezési célt, a könyv tartalmazza az összes algoritmus futási idejének gondos elemzését is.

Ezt a tankönyvet elsősorban a főiskolai és egyetemi oktatás számára, az algoritmusokról és az adatszerkezetekről szóló előadásokhoz ajánljuk. Mivel az algoritmustervezés technikai részleteit is tárgyalja, alkalmas a műszaki szakemberek önképzéséhez is.

Ebben a második kiadásban felfrissítjük az egész könyvet. A változások széles körűek: nemcsak egyes mondatokat írtunk át vagy fogalmaztunk újra, hanem új fejezetekkel is bővítettük az első kiadást.

Az oktatóhoz

A könyvet úgy terveztük, hogy jól érthető és teljes legyen. Számos tantárgyhoz hasznosnak fogja találni – a főiskolai szintű adatszerkezetektől kezdve az egyetemi szintű algoritmusokig. Mivel jóval több anyagot gyűjtöttünk össze, mint amennyi egy tipikus egy féléves tantárgyba belefér, tekintheti a könyvet „raktárnak" vagy „kincsesládának", amelyből kiválaszthatja azt az anyagot, amely a legjobban támogatja a tanítandó tantárgyat.

A tananyagot könnyen összeállíthatja a szükséges fejezetek segítségével. Könnyűnek fogja találni, hogy tantárgyát a szükséges fejezet segítségével megszervezze. A fejezetek viszonylag önállóak, ezért nem kell félnie egyik fejezetnek a másiktól való váratlan és szükségtelen függésétől. Mindegyik fejezet előbb a könnyebb, azután a nehezebb anyagrészeket mutatja be. Az egyes alfejezetek vége természetes megállási hely. Egy főiskolai szintű elő-

adásban alkalmazhatja csak a könnyebb alfejezeteket, míg az egyetemen feldolgozhatja az egész fejezetet.

920-nál több gyakorlat és 140-nél több feladat van a könyvben. Az alfejezetek gyakorlatokkal, a fejezetek feladatokkal végződnek. A gyakorlatok általában rövid választ igénylő kérdések, amelyek az anyag alapvető részének elsajátítását ellenőrzik. Egy részük az önellenőrzést segíti, másik részük viszont házi feladatnak alkalmas. A feladatok viszont jobban kidolgozott esettanulmányok, amelyek gyakran új anyagot is tartalmaznak, és rendszerint több kérdésből állnak, amelyek a hallgatót lépésről lépésre elvezetik a kívánt megoldáshoz.

Megcsillagoztuk (⋆) azokat az alfejezeteket és gyakorlatokat, amelyeket inkább csak az egyetemi képzéshez ajánlunk. Egy csillagos alfejezet nem szükségképpen bonyolultabb a csillag nélkülinél, de lehet, hogy mélyebb matematikai ismereteket igényel. Hasonlóképpen a csillagos gyakorlatok magasabb szintű kiegészítő ismereteket vagy több kreativitást igényelhetnek.

A hallgatóhoz

Azt reméljük, hogy ez a tankönyv élvezhető bevezetést nyújt az algoritmusok világába. Igyekeztünk, hogy minden algoritmus használható és érdekes legyen. Minden algoritmust több lépésben ismertetünk, hogy segítsünk az ismeretlen vagy bonyolult algoritmusok megértésében. Részletesen tárgyaljuk az algoritmusok elemzésének megértéséhez szükséges matematikai ismereteket. Ezeknek a fejezeteknek a felépítése olyan, hogy aki már rendelkezik az adott területen bizonyos jártassággal, a bevezető alfejezeteket átugorhatja, és gyorsan átismételheti a nehezebb részeket.

A könyv terjedelmes, és évfolyama valószínűleg csak az anyag egy részét fogja feldolgozni. Arra törekedtünk, hogy a könyv most hasznos legyen egy-egy tantárgy tankönyveként, későbbi pályafutása során pedig matematikai kézikönyvként vagy technikai segédeszközként.

Milyen előismeretekre van szükség a könyv olvasásához?

- Bizonyos programozási tapasztalat kívánatos. Többek között értenie kell a rekurzív eljárásokat és olyan egyszerű adatszerkezeteket, mint a tömbök és a listák.
- Rendelkeznie kell bizonyos jártassággal a matematikai indukcióval történő bizonyításokban. A könyv egy része felhasználja a matematikai analízis elemeit. Mindenesetre a könyv első és nyolcadik része megtanítja mindarra, amire matematikából szüksége lesz.

A szakemberhez

A tárgyalt területek széles köre alkalmassá teszi ezt a tankönyvet arra, hogy kitűnő, az algoritmusokkal foglalkozó kézikönyv legyen. Mivel minden fejezet viszonylag önálló, Ön arra a területre koncentrálhat, amelyik a legjobban érdekli.

A legtöbb tárgyalt algoritmus a gyakorlatban jól használható, ezért foglalkozunk a megvalósítással kapcsolatos kérdésekkel és más technikai jellegű részletekkel is. Az elsősorban elméleti érdekességű algoritmusokhoz rendszerint gyakorlati alternatívákat is említünk.

Ha bármelyik algoritmust meg akarja valósítani, a pszeudokódunkról egyszerűen lefordíthatja kedvenc programozási nyelvére. A pszeudokódot olyanra terveztük, hogy min-

den algoritmust világosan és tömören mutasson be. Következésképpen nem foglalkozunk hibakezeléssel és más szoftvergyártási problémákkal, melyek a programozási környezetre vonatkozó speciális feltételezéseket igényelnek. Arra törekszünk, hogy minden algoritmust egyszerűen és közvetlenül mutassunk be anélkül, hogy egy-egy programozási nyelv speciális tulajdonságai eltakarnák a lényeget.

Kollégáinkhoz

A könyv széles körű irodalomjegyzéket tartalmaz és hivatkozásokat a legfrissebb művekre. Minden fejezet „megjegyzésekkel" végződik, amelyek történeti részleteket és hivatkozásokat tartalmaznak. Ezek a megjegyzések azonban nem adnak teljes képet az algoritmusok egész területéről. Bár lehet, hogy nehezen hihető egy ilyen nagy terjedelmű könyv esetében, mégis igaz: helyhiány miatt sok érdekes algoritmus nem került be a könyvbe.

Bár nagyon sok hallgató kérte, hogy a könyv tartalmazza a gyakorlatok és feladatok megoldását, tudatosan elkerültük a megoldások közreadását. Ezzel megszabadítottuk a hallgatókat attól a kísértéstől, hogy a feladatok megoldása helyett a könnyebb utat válasszák.

Változások az első kiadáshoz képest

Mi a különbség az első és a második kiadás között? Attól függően, hogyan nézzük, vagy kevés, vagy meglehetősen sok. A tartalomjegyzékek gyors összehasonlítása azt mutatja, hogy az első kiadás fejezeteinek és alfejezeteinek többsége a második kiadásban is szerepel. Két teljes fejezetet és több alfejezetet elhagytunk, ugyanakkor három teljesen új fejezet és az új fejezetek alfejezetei mellett négy új alfejezet van a második kiadásban. Tehát ha valaki a változások mértékét a tartalomjegyzék alapján határozza meg, azt mondhatja, hogy a változások szerény mértékűek.

Az átdolgozás azonban messze túlmegy azon, amit a tartalomjegyzék tükröz. Különösebb rendszerezés nélkül felsoroljuk a második kiadás legfontosabb új jellemzőit.

- Cliff Stein a könyv társszerzője lett.
- Kijavítottuk az ismert hibákat. Hányat? Mondjuk azt, hogy jó néhányat.
- Az első kiadás 3 új fejezettel bővült:
  - a 2. fejezet az algoritmusoknak a számításokban betöltött szerepét elemzi;
  - az 5. a valószínűségi elemzést és a véletlenített algoritmusokat tárgyalja. Éppúgy, mint az első kiadásban, ez a két téma gyakran szerepel a könyvben.
  - a 29. fejezet a lineáris programozással foglalkozik.
- Az első kiadásban is szereplő fejezeteket a következő alfejezetekkel bővítettük:
  - tökéletes hasítás (11.5. alfejezet),
  - a dinamikus programozás két alkalmazása (15.1. és 15.5. alfejezetek),
  - véletlenítést és lineáris programozást alkalmazó közelítő algoritmusok (35.4. alfejezet).
- Annak érdekében, hogy az algoritmusok előbbre kerüljenek a könyvben, három – a matematikai alapokhoz tartozó – fejezet az I. részből a VIII. részbe került át.
- 40-nél több új feladat és 185-nél több új gyakorlat van.

- A helyességbizonyításokban explicit módon alkalmazzuk a ciklusinvariánsokat. Az első ciklusinvariáns a második fejezetben fordul elő, azután pedig a könyvben több tucatszor alkalmazzuk.
- Több valószínűségi elemzést átírtunk. Tucatszor alkalmazzuk az „indikátor valószínűségi változókat", ami egyszerűsíti a valószínűségi elemzéseket – különösen akkor, ha a véletlen változók összefüggnek.
- Kiterjesztettük és felfrissítettük a fejezetek végén lévő megjegyzéseket és az irodalomjegyzéket. Az irodalomjegyzék több mint 50%-kal bővült, és számos olyan új algoritmikus eredményt megemlítünk, amelyek az első kiadás megjelenése óta születtek.

A következő változásokkal találkozhat még a könyvben.

- A rekurziók megoldásáról szóló fejezetből kimaradt az iterációs módszer ismertetése. Helyette viszont a rekurziós fák önálló alfejezetté „léptek elő" (4.2. alfejezet). Azt tapasztaltuk, hogy a rekurziós fák rajzolása kevesebb hibával történik, mint a rekurziók iterálása. Megjegyezzük, hogy a rekurziós fákat főleg olyan állítások megsejtésére érdemes felhasználni, amelyeket azután a helyettesítő módszerrel bizonyítunk.
- A gyorsrendezésnél (7.1. alfejezet) és az átlagosan lineáris futási idejű rendstatisztikai algoritmusoknál (9.2. alfejezet) alkalmazott felbontás különböző. Most a Lomuto által kifejlesztett módszert alkalmazzuk, amely – az indikátor valószínűségi változókkal együtt – egyszerűbb elemzést tesz lehetővé. Az első kiadásban felhasznált, Hoaretól származó módszer most a hetedik fejezetben szerepel feladatként.
- Úgy módosítottuk az univerzális hasítás tárgyalását a 13.3. alfejezetben, hogy a tökéletes hasítás bemutatását is magában foglalja.
- A 12.4. alfejezetben a véletlen építésű bináris keresőfák magasságára a korábbinál lényegesen egyszerűbb elemzés található.
- A dinamikus programozás (13.3. alfejezet) és a mohó algoritmusok (16.2. alfejezet) elemeiről szóló elemzéseket lényegesen kibővítettük. Az aktivitás-kiválasztási probléma – amellyel a mohó algoritmusokról szóló fejezet kezdődik – segít abban, hogy a dinamikus programozás és a mohó algoritmusok közötti összefüggések világosabbá váljanak.
- A diszjunkt halmazok unióját tartalmazó adatszerkezet futási idejének bizonyítását a 21.4. alfejezetben helyettesítettük egy olyan bizonyítással, amely a potenciál-módszert alkalmazza és pontos korlátot eredményez.
- A 22.5. alfejezetben az erősen összefüggő komponenseket meghatározó algoritmus helyességbizonyítása egyszerűbb, világosabb és közvetlenebb.
- Az egy csúcsból induló legrövidebb utakkal foglalkozó 24. fejezetet úgy alakítottuk át, hogy a lényeges tulajdonságok bizonyítása külön alfejezetbe került. Az új szerkezet lehetővé teszi, hogy először az algoritmusra figyeljünk.
- A 34.5. alfejezet a korábbinál alaposabb áttekintést tartalmaz az NP-teljességről, valamint új NP-teljességi bizonyításokat a Hamilton-kör és a részletösszeg problémákra.

Végül lényegében minden alfejezetet átdolgoztunk annak érdekében, hogy a magyarázatok és bizonyítások pontosabbak, egyszerűbbek és érthetőbbek legyenek.

Honlap

További változás az első kiadáshoz viszonyítva, hogy a könyvnek önálló honlapja van, melynek címe `http://mitpress.mit.edu/algorithms/`. Ezt a címet felhasználhatja a hibák jelzésére, kérheti az ismert hibák listáját vagy megjegyzéseket tehet; várjuk észrevételeit.

Nagyon sajnáljuk, hogy nem tudunk személyesen válaszolni minden levélre.

Köszönetnyilvánítás az első kiadásért

Sok barátunk és kollégánk járult hozzá jelentősen a könyv minőségének javításához. Mindannyiuknak köszönjük a segítséget és a konstruktív kritikát.

Az MIT Számítástudományi Laboratóriuma ideális munkakörülményeket biztosított. Kollégáink a laboratórium Számításelméleti Csoportjában különösen sokat segítettek és türelmesen teljesítették a fejezetek ismételt átolvasására vonatkozó kéréseinket. Különösen hálásak vagyunk Baruch Averbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Schmoys és Tardos Éva kollégáknak. Köszönjük William Angnak, Sally Bemusnak, Ray Hirschfeldnek és Mark Reinholdnak, hogy DEC Microvax, Apple Macintosh és Sun Sparcstation típusú gépeinket karbantartották és újragenerálták a TEX fordítóprogramot, ha túlléptük a fordítási időt. A Thinking Machines Corporation részleges támogatást nyújtott Charles Leisersonnak a munkához az MIT-ből való távolléte idején.

Sok kolléga használta ennek a könyvnek a kéziratát az előadásaihoz más egyetemeken is. Ők is számos hibát észrevettek, és módosításokat javasoltak. Különösen köszönjük Richard Beigel, Andrew Goldberg, Joan Lucas, Mark Overmars, Alan Sherman és Diane Souvaine segítségét.

Gyakorlatvezetőink jelentősen hozzájárultak az anyag javításához. Külön köszönjük Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Philipps, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel és Margaret Tuttle segítségét.

További értékes technikai segítséget kaptunk számos más személytől is. Denise Sergent sok órát töltött az MIT könyvtáraiban az irodalmi hivatkozások összegyűjtésével. Olvasótermünk könyvtárosa, Maria Sensale ugyancsak kedves és segítőkész volt. Albert Meyer személyes könyvtárának használata sok munkaóránkat takarította meg a fejezetekhez fűzött megjegyzések megírása során. Shlomo Kipnis, Bill Niehaus és David Wilson ellenőrizték a régi gyakorlatokat, újakat írtak, és jegyzeteket készítettek a megoldásokhoz. Marios Papaefthymiou és Gregory Troxel segítettek a tárgymutató elkészítésében. Az évek során titkárnőink – Inna Radzihovsky, Denise Sergent, Gayle Sherman és különösen Be Blackburn – állandóan segítettek ebben a munkánkban, amiért nagyon hálásak vagyunk nekik.

A korai változatokban sok hibát találtak a hallgatók. Külön köszönjük Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio és Margaret Tuttle lelkiismeretes olvasását.

Számos kolléga vállalta az egyes fejezetek kritikai átnézését, vagy informált bennünket speciális algoritmusokról, amit köszönünk. Különösen hálásak vagyunk Bill Aiello, Alok Aggarwal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Håstad, Alex Ishi, David Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Hers-

hel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan és Paul Wang segítségéért. Számos kolléga bőkezűen támogatott bennünket problémákkal – különösen Andrew Goldberg, Danny Sleator és Umesh Vazirani.

A könyv elkészítése során nagyon kellemes volt az MIT Press és a McGraw-Hill munkatársaival való együttműködés. Külön köszönjük Frank Satlow, Terry Ehling, Larry Cohen, Lorrie Lejeune (MIT Press) és David Shapiro (McGraw-Hill) segítőkészségét, támogatását és figyelmét. Különösen hálásak vagyunk Larry Cohennek a kitűnő korrektori munkáért.

Köszönetnyilvánítás a második kiadásért

Amikor megkértük Julie Sussmannt, hogy legyen a második kiadás technikai szerkesztője, nem is tudtuk, hogy így milyen nagy segítséghez jutottunk. A technikai szerkesztés mellett Julie önzetlenül javította a szöveget. Szinte szégyelljük, milyen sok hibát talált korai kéziratainkban. Ha figyelembe vesszük, mennyi hibát talált az első kiadásban (sajnos, már a megjelenés után), akkor ez nem is olyan meglepő. Továbbá háttérbe szorította saját ügyeit, hogy a miénket előbbre vigye. Még a Virgin-szigeteken tett utazásra is elvitt magával néhány fejezetet. Julie, nem tudjuk eléggé megköszönni azt a hatalmas munkát, amit végeztél.

Míg a második kiadást írták, a könyv szerzői a Dartmouth College Számítástudományi Tanszékén, illetve az MIT Számítástudományi Laboratóriumában dolgoztak. Mindkét hely ösztönzően hatott munkánkra, köszönjük kollégáink támogatását.

Barátok és kollégák a világ minden részéből elláttak bennünket javaslatokkal, az írást segítő észrevételekkel. Köszönjük Sanjeev Arora, Javed Aslam, Guy Blelloch, Avrim Blum, Scot Drysdale, Hany Farid, Hal Gabow, Andrew Goldberg, David Johnson, Yanlin Liu, Nicolas Schabanel, Alexander Schrijver, Sasha Shen, David Shmoys, Dan Spielman, Gerald Jay Sussman, Bob Tarjan, Mikkel Thorup és Vijay Vazirani segítségét.

Nagyon sokat tanultunk az algoritmusokkal kapcsolatban tanárainktól és kollégáinktól. Különösen az alábbi tanárainknak vagyunk hálásak: Jon L. Bentley, Bob Floyd, Don Knuth, Harold Kuhn, H. T. Kung, Richard Lipton, Arnold Ross, Larry Snyder, Michael I. Shamos, David Shmoys, Ken Steiglitz, Tom Szymanski, Tardos Éva, Bob Tarjan és Jeffrey Ullman.

Köszönjük azok munkáját, akik az algoritmusok tantárgyból az MIT-n és Dartmouthban gyakorlatot vezettek: Joseph Adler, Craig Barrack, Bobby Blumofe, Roberto De Prisco, Matteo Frigo, Igal Galperin, David Gupta, Raj D. Iyer, Nabil Kahale, Sarfraz Khurshid, Stavros Kolliopoulos, Alain Leblanc, Yuan Ma, Maria Minkoff, Dimitris Mitsouras, Alin Popescu, Harald Prokop, Sudipta Sengupta,Donna Slonim, Joshua A. Tauber, Sivan Toledo, Elisheva Werner-Reiss, Lea Wittie, Qiang Wu és Michael Zhang.

A számítógépes segítséget William Ang, Scott Blomquist és Greg Shomo (MIT), valamint Wayne Cripps, John Konkle, Tim Tregubov (Dartmouth) biztosította. Köszönjük Be Blackburn, Don Dailey, Leigh Deacon, Irene Sebeda és Cheryl Patton Wu (MIT), valamint Phyllis Bellmore, Kelly Clark, Delia Mauceli, Sammie Travis, Deb Whiting és Beth Young (Dartmouth) segítségét az adminisztratív ügyek intézésében. Alkalmanként Michael Fromberger, Brian Campbell, Amanda Eubanks, Sung Hoon Kim és Neha Narula is segített Dartmouthban.

Sokan vettek észre hibákat az első kiadásban. Ezért köszönetet mondunk azoknak, akik elsőként jelezték az első kiadás valamelyik hibáját: Len Adleman, Selim Akl, Richard Anderson, Juan Andrade-Cetto, Gregory Bachelis, David Barrington, Paul Beame, Richard Beigel, Margrit Betke, Alex Blakemore, Bobby Blumofe, Alexander Brown, Xavier Cazin,

Jack Chan, Richard Chang, Chienhua Chen, Ien Cheng, Hoon Choi, Drue Coles, Christian Collberg, George Collins, Eric Conrad, Peter Csaszar, Paul Dietz, Martin Dietzfelbinger, Scot Drysdale, Patricia Ealy, Yaakov Eisenberg, Michael Ernst, Michael Formann, Nedim Fresko, Hal Gabow, Marek Galecki, Igal Galperin, Luisa Gargano, John Gately, Rosario Genario, Mihaly Gereb, Ronald Greenberg, Jerry Grossman, Stephen Guattery, Alexander Hartemik, Anthony Hill, Thomas Hofmeister, Mathew Hostetter, Yih-Chun Hu, Dick Johnsonbaugh, Marcin Jurdzinki, Nabil Kahale, Fumiaki Kamiya, Anand Kanagala, Mark Kantrowitz, Scott Karlin, Dean Kelley, Sanjay Khanna, Haluk Konuk, Dina Kravets, Jon Kroger, Bradley Kuszmaul, Tim Lambert, Hang Lau, Thomas Lengauer, George Madrid, Bruce Maggs, Victor Miller, Joseph Muskat, Tung Nguyen, Michael Orlov, James Park, Seongbin Park, Ioannis Paschalidis, Boaz Patt-Shamir, Leonid Peshkin, Patricio Poblete, Ira Pohl, Stephen Ponzio, Kjell Post, Todd Poynor, Colin Prepscius, Sholom Rosen, Dale Russell, Hershel Safer, Karen Seidel, Joel Seiferas, Erik Seligman, Stanley Selkow, Jeffrey Shallit, Greg Shannon, Micha Sharir, Sasha Shen, Norman Shulman, Andrew Singer, Daniel Sleator, Bob Sloan, Michael Sofka, Volker Strumpen, Lon Sunshine, Julie Sussman, Asterio Tanaka, Clark Thomborson, Nils Thommesen, Homer Tilton, Martin Tompa, Andrei Toom, Felzer Torsten,Hirendu Vaishnav, M. Veldhorst, Luca Venuti, Jian Wang, Michael Wellman, Gerry Wiener, Ronald Williams, David Wolfe, Jeff Wong, Richard Woundy, Neal Young, Huaiyuan Yu, Tian Yuxing, Joe Zachary, Steve Zhang, Florian Zschoke és Uri Zwick.

Számos kollégánk tartalmas véleményt mondott, vagy hosszú kérdőívet töltött ki. Köszönjük Nancy Amato, Jim Aspnes, Kevin Compton, William Evans, Gács Péter, Michael Goldwasser, Andrzej Proskurowski, Vijaya Ramachandran és John Reif véleményét. A következőknek köszönjük, hogy kitöltve visszaküldték a kérdőívet: James Abello, Josh Benaloh, Bryan Beresford-Smith, Kenneth Blaha, Hans Bodlaender, Richard Borie, Ted Brown, Domenico Cantone, M. Chen, Robert Cimikowski, William Clocksin, Paul Cull,Rick Decker, Matthew Dickerson, Robert Douglas, Margaret Fleck, Michael Goodrich, Susanne Hambrusch, Dean Hendrix, Richard Johnsonbaugh, Kyriakos Kalorkoti, Srinivas Kankanahalli, Hikyoo Koh, Steven Lindell, Errol Lloyd, Andy Lopez, Dian Rae Lopez, George Lucker, David Maier, Charles Martel, Xiannong Meng, David Mount, Alberto Policriti, Andrzej Proskurowski, Kirk Pruhs, Yves Robert, Guna Seetharaman, Stanley Selkow, Robert Sloan, Charles Steele, Gerard Tel, Murali Varanasi, Bernd Walter és Alden Wright. Bár minden javaslatukat meg tudtuk volna valósítani. Az egyetlen probléma az, hogy akkor a második kiadás körülbelül 3000 oldalas lett volna!

A második kiadást LaTeX 2_ϵ segítségével szerkesztettük. Michael Downes alakította át a „klasszikus” LaTeX makrókat a LaTeX 2_ϵ makróivá, és ő alakította át a szövegfájlokat is úgy, hogy az új makrókat használhassuk. Az ábrákat a MacDraw II program segítségével rajzoltuk, Apple Macintosh számítógépen; köszönet illeti Joanna Terryt (Claris Corporation) és Michael Mahoneyt (Advanced Computer Graphics) a sok időt igénylő támogatásért. A tárgymutató Windex – a szerzők által írt C nyelvű program – alkalmazásával készült. Az irodalomjegyzéket BibTeX segítségével készítettük. Ayarkor Mills-Tettey és Rob Leathern segítettek az ábrákat a MacDraw Pro számára átalakítani, és Ayarkor irodalomjegyzékünket is javította.

Az első kiadáshoz hasonlóan nagyon kellemes volt a The MIT Press és a McGraw-Hill munkatársaival együtt dolgozni. Szerkesztőink, Bob Prior (The MIT Press) és Betsy Jones (McGraw-Hill) elviselték a furcsa ötleteinket, mézesmadzaggal és ostorral ösztönöztek a munkára.

Végezetül köszönjük feleségeink – Nicole Cormen, Gail Rivest és Rebecca Ivry –, gyermekeink – Ricky, William és Debby Leiserson; Alex és Christopher Rivest; Molly, Noah és Benjamin Stein –, valamint szüleink – Renee és Perry Cormen, Jean és Mark Leiserson, Sirley és Lloyd Rivest, Irene és Ira Stein – szeretetét és támogatását a könyv írása alatt. Családjaink szeretete és gondoskodása tette lehetővé ennek a tervnek a megvalósítását. Szeretettel ajánljuk nekik ezt a könyvet.

2001. május

A szerzők előszava a magyar nyelvű kiadáshoz

TISZTELT OLVASÓ! Az *Introduction to Algorithms* második kiadásának kiváló fordítását tartja a kezében. Bár mi, a könyv szerzői, nem tudunk magyarul, meg vagyunk arról győződve, hogy a fordítók Iványi Antal vezetésével nagyszerű munkát végeztek. Miért vagyunk erről meggyőződve? Azért, mert a fordítók elárasztottak bennünket „hibajelzésekkel". Bár az üzenet egy része apró tipográfiai hibákról szólt, legtöbbjük lényeges volt és bizonyította a fordítók hozzáértését.

Gratulálunk Benczúr András Jr., Burcsi Péter, Csörnyei Zoltán, Fekete István, Gregorics Tibor, Hajdú András, Horváth Gyula, Ispány Márton, Iványi Anna, Kása Zoltán, Kovács Attila, Lencse Zsolt, Marx Dániel, Nagy Sára, Schipp Ferenc, Sike Sándor, Simon Péter, Szeg ő László, Szili László, Veszprémi Anna, Vida János, Vizvári Béla és Wiener Gábor fordítóknak, valamint Benczúr András, Csirik János, Fábián Csaba, Frank András, Kátai Imre, Kiss Attila, Kormos János, Recski András, Schipp Ferenc, Szántai Tamás és Varga László lektoroknak a kiváló fordítás elkészítéséhez, és köszönjük az eredeti könyv jobbításához nyújtott segítségüket.

Ugyancsak gratulálunk a magyar Olvasóknak az algoritmusok iránti érdekl ődésükhöz. Magyarország története sok kiváló matematikusról is szól, különösen kombinatorikusokról. Mivel az algoritmusok hidat alkotnak a kombinatorika és a számítástudomány között, ezért nem meglepő, hogy egyre több magyar részesül nemzetközi elismerésben számítástudományi kutatásaiért. Amikor Ön, tisztelt Olvasó, átmegy ezen a hídon, remélhet ően ösztönző társnak fogja találni könyvünk fordítását, amely megmozgatja a gondolatait és finomítja mérnöki szemléletét. Azt reméljük, élvezni fogja a könyv olvasását, amint mi is élveztük a megírását.

Thomas H. Cormen	*Hanover, New Hampshire*
Charles E. Leiserson	*Cambridge, Massachusetts*
Ronald L. Rivest	*Cambridge, Massachusetts*
Clifford Stein	*New York, New York*

2003. augusztus

Előszó a magyar nyelvű kiadáshoz

KEDVES OLVASÓ! 1990-ben jelent meg Cormen, Leiserson és Rivest *Introduction to Algorithms* című könyvének első kiadása, amelyet 2001-ig további 23 változatlan utánnyomásban kiadtak. A könyv magyar fordítását 1997-ben készítettük el *Algoritmusok* címmel, amelyet két további kiadás követett.

Lényegesen kibővített és átdolgozott formában jelentette meg a *The MIT Press* és a *McGraw-Hill Company* 2001-ben az *Introduction to Algorithms* második kiadását, amelynek változatlan népszerűségét mutatja az azóta megjelent három javított kiadás is.

A tartalom nagymértékű változását azzal is hangsúlyozni kívánjuk, hogy ezt a könyvet *Új algoritmusok* címmel adjuk közre.

Munkánk során hatékony segítséget kaptunk a szerzőktől és a *The MIT Press* munkatársaitól. Thomas Cormen professzortól még februárban megkaptunk a javított angol szöveget LaTeX és PS formában. A kéziratot HILaTeX segítségével készítettük, melyet Belényesi Viktorral és Locher Kornéllal fejlesztettünk ki.

A magyar változat megjelenéséért köszönet illeti azokat – elsősorban a *Babeş-Bolyai Egyetem*, a *Budapesti Műszaki és Gazdaságtudományi Egyetem*, a *Debreceni Egyetem*, az *Eötvös Loránd Tudományegyetem* és a *Szegedi Tudományegyetem* tanárait – akik sok időt áldoztak a könyv fordítására és lektorálására.

A fordítás gyors megjelenéséhez és igényes kiadásához elengedhetetlen volt az a megkülönböztetett figyelem, amellyel a Scolar Kiadó kezelte a könyvet.

A magyar kiadásban az eredeti könyv honlapján 2003. szeptember 16-ig közreadott javításokat tudtuk figyelembe venni. A könyvet kísérő erős nemzetközi érdeklődést jellemzi, hogy a negyedik utánnyomás májusi megjelenése óta a szerzők 39 javító észrevételt fogadtak el (ezek közül 18 a magyar alkotóktól származik).

Arra számítunk, hogy rövidesen újabb magyar nyelvű kiadásra lesz igény. Ebben szeretnénk az első kiadás hibáit kijavítani. Ezért kérjük az Olvasókat, hogy javaslataikat, észrevételeiket (lehetőleg pontosan megjelölve a hiba előfordulási helyét, és megadva a javasolt új szöveget) küldjék el a `tony@inf.elte.hu` elektronikus címre.

A fordítás során a legtöbb fejezethez kiegészítést írtunk, névmutatót készítettünk, az irodalomjegyzéket kiegészítettük magyar és friss idegen nyelvű hivatkozásokkal. Ezek és az ismert hibák listája letölthető a `http://people.inf.elte.hu/tony/books/ujalg` címről.

Iványi Antal *Budapest*

2003. szeptember 16.

I. ALAPOK

Bevezetés

Ez a rész elindítja az Olvasót az algoritmusok tervezéséről és elemzéséről való gondolkodás útján. Az a célunk vele, hogy barátságos bevezetést adjon ahhoz, hogyan kell egy algoritmust megadni, néhány, a könyv további részeiben használt tervezési stratégiához és több, az algoritmusok elemzésében használt alapvető gondolathoz. A könyv későbbi részei ezekre a gondolatokra épülnek.

Az 1. fejezet áttekintést ad az algoritmusokról és a modern számítógépes rendszerekben elfoglalt helyükről. Ebben a fejezetben definiáljuk az algoritmus fogalmát, és bemutatunk néhány példát. Bemutatjuk, hogy az algoritmusok éppúgy technológiák, mint a gyors hardver, a grafikus felhasználói felületek, az objektumelvű rendszerek és a hálózatok.

A 2. fejezetben találkozunk az első algoritmusokkal, amelyek megoldják egy n elemből álló sorozat rendezésének feladatát. Ezek az algoritmusok olyan pszeudokódban vannak megadva, amely közvetlenül ugyan nem fordítható le egyetlen programozási nyelvre sem, elég világosan tükrözi azonban az algoritmusok szerkezetét ahhoz, hogy egy hozzáért ő programozó a kedvére választott nyelven meg tudja valósítani. Egyrészt a beszúró rendezést vizsgáljuk, amely növekményes megközelítést alkalmaz, másrészt az összefésülő rendezést, amely az „oszd-meg-és-uralkodj” néven ismert rekurzív módszert alkalmazza. Bár mindkét algoritmus futási ideje nő a bemenet n méretének növelésekor, a növekedési sebesség a két algoritmusra nézve különböző. A 2. fejezetben meghatározzuk ezeket a futási időket, és hasznos jelölést vezetünk be ahhoz, hogy a futási időket kifejezzük.

A 3. fejezetben pontosan értelmezzük ezt a jelölést, amelyet aszimptotikus jelölésnek nevezünk. A fejezet néhány aszimptotikus jelölés definiálásával kezdődik, amelyeket arra használunk, hogy az algoritmusok futási idejére felső és alsó korlátokat adjunk. A 3. fejezet további része elsősorban a matematikai jelölések bemutatására szolgál. Ennek a résznek a célja sokkal inkább az, hogy az Olvasó által használt jelölések megegyezzenek a könyv jelöléseivel, semmint az, hogy új matematikai fogalmakat tanítson.

A 4. fejezetben rekurzív feladatok megoldási módjait ismertetjük, amelyeket például az 1. fejezetben az összefésülő rendezés elemzésénél használtunk, és amelyek majd még sokszor előkerülnek. Az oszd-meg-és-uralkodj típusú algoritmusokból eredő rekurzív feladatok megoldására nagyon hatékony a „mestermódszer”. A fejezet nagy része a mester módszer helyességének bizonyítása, de ezt a bizonyítást gond nélkül el is lehet hagyni.

Az 5. fejezet bevezetés a valószínűségi elemzésekhez és véletlenített algoritmusokhoz. A valószínűségi elemzést rendszerint arra használjuk, hogy az algoritmusok futási idejét olyan esetekben meghatározzuk, amikor a problémához tartozó valószínűségeloszlás miatt

a futási idő azonos méretű, de különböző bemenetekre különböző lehet. Bizonyos esetekben feltesszük, hogy a bemenetek ismert valószínűségeloszlást követnek, és a futási időt az összes lehetséges bemenetre nézve átlagoljuk. Más esetekben a valószínűségeloszlás nem a bemenetekkel, hanem az algoritmus működése során hozott véletlen döntésekkel kapcsolatos. Azt az algoritmust, melynek működését nem csak a bemenet, hanem egy véletlenszám generátor által előállított értékek is befolyásolnak, véletlenített algoritmusnak nevezzük. A véletlenített algoritmusokat arra használjuk, hogy kikényszerítsék a bemenetek bizonyos eloszlását – és ezáltal biztosítsák, hogy egyetlen bemenet se okozzon mindig rossz hatékonyságot, vagy azt, hogy korlátozzák az olyan algoritmusok hibázási arányát, amelyek bizonyos esetekben hibás eredményt is adhatnak.

Az A–C függelékek további matematikai anyagot tartalmaznak, amelyet hasznosnak fog találni a könyv olvasása során. Ezen függelékek anyagának a nagy részét már valószínűleg látta korábban is (bár az általunk alkalmazott jelölési megállapodások különbözhetnek azoktól, amelyeket korábban látott). Másrészt viszont az I. részben tárgyalt anyag nagyobb részét korábban még nem látta. Mind az I. rész, mind pedig a Függelék fejezeteit módszertani céllal írtuk.

1. Az algoritmusok szerepe a számításokban

Mi az az algoritmus? Miért érdemes az algoritmusokat tanulmányozni? Mi az algoritmusok szerepe a számítógépekben alkalmazott más technológiákhoz viszonyítva? Ebben a fejezetben válaszolunk ezekre a kérdésekre.

1.1. Algoritmusok

Informálisan ***algoritmusnak*** nevezünk bármilyen jól definiált számítási eljárást, amely ***bemenetként*** bizonyos értéket vagy értékeket kap és ***kimenetként*** bizonyos értéket vagy értékeket állít elő. Eszerint az algoritmus olyan számítási lépések sorozata, amelyek a bemenetet átalakítják kimenetté.

Az algoritmusokat tekinthetjük olyan eszköznek is, amelynek segítségével pontosan meghatározott ***számítási feladatokat*** oldunk meg. Ezeknek a feladatoknak a megfogalmazása általában a bemenet és a kimenet közötti kívánt kapcsolat leírása.

Például szükségünk lehet egy számsorozat növekvő sorrendbe való rendezésére. Ez a feladat a gyakorlatban sűrűn előfordul, és termékeny alapul szolgál számos tervezési módszer bevezetéséhez. A ***rendezési feladatot*** formálisan a következőképpen definiáljuk.

Bemenet: n számot tartalmazó $\langle a_1, a_2, \ldots, a_n \rangle$ sorozat.

Kimenet: a bemenet sorozat olyan $\langle a'_1, a'_2, \ldots, a'_n \rangle$ permutációja (újrarendezése), hogy $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Ha például a bemenet $\langle 31, 41, 59, 26, 41, 58 \rangle$, akkor egy rendező algoritmus kimenetként a $\langle 26, 31, 41, 41, 58, 59 \rangle$ sorozatot adja meg. Egy ilyen bemenet a rendezési feladat egy ***esete.*** Általában egy ***feladat (probléma) egy esete*** az a – a feladat megfogalmazásában szereplő feltételeknek eleget tevő – bemenet, amely a feladat megoldásának kiszámításához szükséges.

A rendezés az informatikában alapvető művelet (számos program alkalmazza közbeeső lépésként), ezért nagyszámú jó rendező algoritmust dolgoztak ki. Az, hogy adott alkalmazás esetén melyik rendező algoritmus a legjobb – más tényezők mellett – függ a rendezendő elemek számától, rendezettségétől, a rájuk vonatkozó lehetséges korlátoktól és a felhasználandó tároló fajtájától (központi memória, lemezek, szalagok).

Egy algoritmust ***helyesnek*** mondunk, ha minden bemenetre megáll és helyes eredményt ad. Azt mondjuk, hogy a helyes algoritmus megoldja az adott számítási feladatot. Egy nem helyes algoritmus esetén előfordulhat, hogy nem minden bemenetre áll meg, vagy bizonyos bemenetekre nem a kívánt választ adja. Várakozásunkkal ellentétben egy nemhelyes algoritmus is lehet hasznos, ha hibázási aránya elfogadható. Ilyen algoritmusra példát látunk majd a 31. fejezetben, amikor a nagy prímszámokat előállító algoritmusokat tanulmányozzuk.

Egy algoritmus megadható magyar nyelven, számítógépes programként vagy akár hardver segítségével. Az egyetlen követelmény az, hogy a leírás pontosan adja meg a követendő számítási eljárást.

Milyen feladatokat oldunk meg algoritmusokkal?

Természetesen nem a rendezés az egyetlen olyan feladat, melynek megoldására algoritmusokat dolgoztak ki. (Ezt az Olvasó valószínűleg sejtette, amikor meglátta ennek a könyvnek a terjedelmét.) Az algoritmusok gyakorlati alkalmazásai mindenütt előfordulnak – például a következő területeken.

- A Human Genom Project célja, hogy azonosítsák az emberi DNS-ben lévő 100 000 gént, és meghatározzák az emberi DNS-t alkotó 3 milliárd kémiai bázispárt, ezt az információt adatbázisban tárolják és eszközöket fejlesszenek ki az adatok elemzésére. Ezen lépések mindegyike bonyolult algoritmusokat igényel. Ezeknek a feladatoknak a megoldása túlnő könyvünk keretein, ugyanakkor számos fejezetének gondolatait alkalmazzák ezeknek a biológiai feladatoknak a megoldása során, és így ezek a gondolatok segítenek a tudósoknak abban, hogy úgy oldjanak meg feladatokat, hogy közben hatékonyan használják az erőforrásokat.
- Az internet világszerte lehetővé teszi, hogy az emberek gyorsan elérjenek nagy mennyiségű információt, és abban keresni tudjanak. Ennek érdekében okos algoritmusokat alkalmaznak ezen nagy mennyiségű információ kezelésére és átalakítására. Két példa a megoldandó feladatokra: a jó utak megtalálása az adatok továbbításához (ilyen feladatok megoldására szolgáló módszereket ismertetünk a 24. fejezetben), és egy gyors módszer azoknak az oldalaknak a megkereséséhez, amelyeken a számunkra fontos információ található (ilyen módszerek szerepelnek a 11. és a 32. fejezetben).
- Az elektronikus kereskedelem lehetővé teszi, hogy anyagi javakat cseréljünk, szolgáltatásokat vegyünk igénybe elektronikusan. Az a képesség, hogy olyan információt, mint a hitelkártyánk száma, jelszavak és banki utasítások, a nyilvánosság elől elzártan kezelhessünk, lényeges, ha az elektronikus kereskedelmet széles körben akarjuk alkalmazni. A nyilvános kulcsú kriptográfia és a digitális aláírás (melyekkel a 31. fejezet foglalkozik) a felhasznált alapvető eszközökhöz tartozik és a numerikus algoritmusokon, valamint a számelméleten alapul.
- A termelésben és más kereskedelmi tevékenység során gyakran lényeges, hogy a szűkös erőforrásokat a legelőnyösebben használjuk fel. Egy olajtársaság számára például fontos lehet, hol állítsa fel kútjait, hogy a várható profitja a lehető legnagyobb legyen. Aki az Egyesült Államok elnöke szeretne lenni, tudni szeretné, hogyan fordítsa pénzét kampánytanácsadókra ahhoz, hogy a legnagyobb valószínűséggel megválasszák. Egy légitársaság számára lényeges lehet, hogyan rendeljen személyzetet a járatokhoz a lehető legolcsóbban úgy, hogy minden járatnak legyen személyzete, és a személyzet

beosztására vonatkozó állami előírásokat betartsa. Ezek mind olyan feladatok, amelyek a 29. fejezetben tárgyalt lineáris programozás segítségével megoldhatók.

Bár ezeknek a feladatoknak bizonyos részletei túlmutatnak könyvünk keretein, tárgyalunk olyan módszereket, amelyek ezen feladatok, illetve feladatosztályok megoldására alkalmazhatók. A könyvben sok konkrét feladat megoldását is bemutatjuk. Ezek közé tartoznak az alábbi feladatok:

- Adott egy autóstérkép, amelyen az utak bármely két szomszédos metszéspontjának távolsága fel van tüntetve, és célunk az, hogy meghatározzuk egy adott metszéspontból egy másikba vezető legrövidebb utat. A lehetséges utak száma még akkor is hatalmas lehet, ha nem engedjük meg az önmagukat keresztező utakat. Hogyan válasszuk ki a lehetséges utak közül a legrövidebbet? Ebben az esetben az autóstérképet (amelyik önmagában is a valódi utak egy modellje) egy gráffal modellezzük (amivel majd a 10. fejezetben és a B függelékben fogunk találkozni), és a gráfban fogjuk az egyik csúcsból a másik csúcsba vezető legrövidebb utat keresni. A 24. fejezetben látni fogjuk, hogyan oldható meg ez a feladat hatékonyan.
- Adott egy n mátrixból álló $\langle A_1, A_2, \ldots, A_n \rangle$ sorozat, és meg akarjuk határozni a mátrixok $A_1 A_2 \cdots A_n$ szorzatát. Mivel a mátrixok szorzása asszociatív művelet, több szorzási sorrend lehetséges. Ha például $n = 4$, akkor a mátrixszorzást az alábbi zárójelezések bármelyike szerint elvégezhetjük: $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$ vagy $(((A_1A_2)A_3)A_4)$. Ha a mátrixok négyzetesek (és ebből adódóan azonos méretűek), a szorzás sorrendje nem befolyásolja a szorzás elvégzésének idejét. Ha azonban a mátrixok mérete különböző (bár a méretük lehetővé teszi a szorzást), akkor a szorzás sorrendje nagy különbséget okozhat. A lehetséges szorzási sorrendek száma n-nek exponenciális függvénye, ezért minden sorrend kipróbálása nagyon sokáig tartana. A 15. fejezetben látni fogjuk, hogyan használjuk a dinamikus programozásnak nevezett általános módszert arra, hogy ezt a feladatot sokkal hatékonyabban oldjuk meg.
- Adott az $a \equiv b \pmod{n}$ egyenlet, ahol a, b és n egész számok, és az összes olyan x számot meg akarjuk határozni modulo n, amely kielégíti az egyenletet. Lehet, hogy nulla, egy vagy több ilyen megoldás van. Egyszerűen rendre kipróbálhatjuk az $x = 0, 1, \ldots, n - 1$ számokat, de a 31. fejezetben bemutatunk egy ennél hatékonyabb módszert.
- Adott a síkban n pont, és meg akarjuk határozni ezeknek a pontoknak a konvex burkát. A konvex burok a legkisebb konvex sokszög, amely tartalmazza a pontokat. Intuitíven azt képzelhetjük, hogy minden pontot egy táblából kiálló szög helyettesít. A konvex burkot pedig egy szoros – a szögeket körülvevő – gumiszalag szemlélteti. Minden olyan szög, amelynél változik a szalag iránya, a konvex burok egy csúcsa. (Példaként lásd a 33.6. ábrát.) A pontoknak mind a 2^n részhalmaza lehet a konvex burok csúcsainak halmaza. Annak ismerete, hogy mely csúcsok tartoznak a konvex burokhoz, nem elég, mivel még azt is tudnunk kell, hogy a csúcsok milyen sorrendben következnek a konvex burokban. Ezért a konvex burok csúcsainak megválasztására sok lehetőség van. A 33. fejezet két módszert is tartalmaz a konvex burok meghatározására.

Ez a felsorolás távolról sem teljes (amint azt a könyv súlya alapján valószínűleg gyanította), de már ezekből is kitűnik két közös tulajdonság, amelyek számos érdekes algoritmusra jellemzők.

1. Sok megoldás van, de a legtöbbjük nem az, ami nekünk kell. Egy olyat megtalálni, amilyenre szükségünk van, nagyon nehéz lehet.

2. Vannak gyakorlati alkalmazások. A fenti példák közül a legrövidebb út megtalálása a legegyszerűbb példa. Egy vasúti vagy kamionos szállítással foglalkozó cégnek pénzügyi érdeke a legrövidebb utak megtalálása adott út- vagy vasúthálózatban, mivel a legrövidebb utak kevesebb munkaidőt és kevesebb üzemanyagot igényelnek. Vagy egy irányítópont az interneten azért keresi a legrövidebb utat a hálózatban, hogy egy üzenet gyorsan célba érjen.

Adatszerkezetek

Ez a könyv többféle adatszerkezettel is foglalkozik. Az ***adatszerkezet*** adatok tárolására és szervezésére szolgáló módszer, amely lehetővé teszi a hozzáférést és módosításokat. Egyetlen adatszerkezet sem megoldás minden célra, és ezért fontos ismernünk több adatszerkezetet és azok korlátait.

Technika

Bár használhatja ezt a könyvet az algoritmusok „szakácskönyveként”, egy napon találkozhat olyan problémával, amelynek megoldására még nem publikáltak algoritmust (ilyen például ennek a könyvnek számos gyakorlata és feladata!). Ez a könyv megtanítja az Olvasót algoritmusok tervezésének és elemzésének olyan módszereire, melyekkel algoritmusokat dolgozhat ki saját használatra, megmutathatja, hogy az algoritmusok helyes megoldást adnak és jellemezheti hatékonyságukat.

Nehéz feladatok

Ennek a könyvnek a nagy része hatékony algoritmusokról szól. Hatékonysági mértékként rendszerint a sebességet használjuk, azaz azt az időt, amennyit az algoritmus felhasznál az eredmény előállítására. Vannak azonban olyan feladatok, amelyeknek a megoldására nem ismerünk hatékony algoritmust. A 34. fejezet ezeknek a feladatoknak egy érdekes csoportjával foglalkozik – az NP-teljes feladatokkal.

Miért érdekesek az NP-teljes feladatok? Először azért, mert bár NP-teljes feladat megoldására még soha senki nem talált hatékony algoritmust, soha senki nem bizonyította be, hogy ilyen algoritmus nem létezik. Más szavakkal, nem tudjuk, vajon léteznek-e hatékony algoritmusok az NP-teljes feladatok megoldására. Másodszor azért, mert az NP-teljes feladatoknak megvan az az érdekes tulajdonsága, hogy ha bármelyikük megoldására létezik hatékony algoritmus, akkor mindegyik megoldására létezik ilyen algoritmus. Az NP-teljes feladatok közötti kapcsolat a hatékony megoldás hiányát még kínzóbbá teszi. Harmadszor, több NP-teljes probléma hasonló – bár nem azonos velük – olyan feladatokhoz, amelyek megoldására ismerünk hatékony algoritmusokat. A feladat megfogalmazásának kis változtatása hatalmas változást okozhat a legjobb ismert algoritmus hatékonyságában.

Értékes az NP-teljes feladatok ismerete, mivel meglepően sűrűn előfordulnak a gyakorlati alkalmazásokban. Ha önt megkérik, hogy keressen egy NP-teljes feladat megoldására hatékony algoritmust, valószínűleg jelentős időt tölt a sikertelen kereséssel. Ha meg tudja

mutatni, hogy a feladat NP-teljes, akkor az előbbiek helyett arra fordíthatja az idejét, hogy olyan hatékony algoritmust tervezzen, amely jó – bár nem a lehető legjobb – eredményt ad.

Konkrét példaként tekintsünk egy teherszállító céget, amelynek van egy központi telephelye. Mindennap megrakják a teherautót és elküldik, hogy különböző helyekre szállítsa ki az árut. A nap végére a teherautónak vissza kell térnie a telephelyre, hogy készen álljon a következő napi rakodáshoz. A költségek csökkentése érdekében a cég úgy akarja megválasztani a szállítási megállóhelyeket, hogy az a teherautó számára a legkisebb megtett utat tegye lehetővé. Ez a feladat a jól ismert „utazóügynök probléma", amely NP-teljes. Nem ismerünk hatékony algoritmust a megoldására. Bizonyos feltételek esetén azonban vannak olyan hatékony algoritmusok, amelyek olyan megoldást szolgáltatnak, amely nincs messze az optimálistól. A 35. fejezetben ilyen „közelítő algoritmusokat" elemzünk.

Gyakorlatok

1.1-1. Adjunk meg egy-egy olyan példát mindennapi életünkből, amelyben a következő számítási feladatok előfordulnak: rendezés, a legjobb sorrend meghatározása mátrixok szorzására és a konvex burok meghatározása.

1.1-2. A sebességen kívül milyen más hatékonysági mértékek alkalmazhatók egy valódi számítógépben?

1.1-3. Válasszunk egy ismerős adatszerkezetet, elemezzük előnyeit és korlátait.

1.1-4. Miben hasonlítanak egymásra a fentebb ismertetett, a legrövidebb utakról, ill. az utazóügynökről szóló feladatok? Miben különböznek egymástól?

1.1-5. Mondjunk egy olyan problémát a hétköznapi életből, amelyre csak az optimális megoldás az elfogadható. Mondjunk egy olyan problémát is, ahol egy „közelítő" megoldás is közel olyan jó, mint az optimális.

1.2. Algoritmusok mint technológia

Tegyük fel, hogy a számítógépek sebessége végtelen és a memória ingyenes. Van értelme ekkor az algoritmusokat tanulmányozni? A válasz igen; ha másért nem is, például azért, hogy megmutassuk, hogy algoritmusunk futása befejeződik, és helyes eredményt ad.

Ha a számítógépek végtelen gyorsak lennének, akkor bármely helyes megoldási módszer megfelelő lenne. Valószínűleg azt kívánnánk, hogy a megvalósítás megfeleljen a jó szoftvermérnöki gyakorlatnak (azaz jól tervezett és dokumentált legyen), és azt a módszert alkalmaznánk, amelyet a legkönnyebb megvalósítani.

Természetesen a számítógépek lehetnek gyorsak, de nem végtelenül gyorsak. A memória pedig lehet olcsó, de nem ingyenes. A számítási idő ezért korlátos erőforrás, és ugyanez vonatkozik a tárolási kapacitásra is. Ezeket az erőforrásokat okosan kell felhasználni, és a futási időt, valamint memóriafelhasználást tekintve hatékony algoritmusok segítenek ebben.

Hatékonyság

Az ugyanannak a feladatnak a megoldására tervezett algoritmusok gyakran drámai módon különböző hatékonyságot mutatnak. Ezek a különbségek jóval nagyobbak lehetnek, mint ami a hardver és a szoftver különbözőségéből adódhat.

A 2. fejezetben példaként két rendezési algoritmust mutatunk be. Az első ***beszúró rendezésként*** ismert, és n elem rendezéséhez körülbelül $c_1 n^2$ időt használ fel, ahol c_1 olyan állandó, amely nem függ n-től. Azaz n^2-tel arányos időt használ fel. A második az ***összefésülő rendezés,*** amelynek körülbelül $c_2 n \lg n$ időre van szüksége, ahol $\lg n$ a $\log_2 n$ függvényt jelenti, és c_2 egy másik állandó, amely ugyancsak független n-től. A beszúró rendezés állandó tényezője rendszerint kisebb, mint az összefésülő rendezésé, ezért $c_1 < c_2$. Látni fogjuk, hogy a futási időben szereplő állandó tényezők sokkal kevésbé fontosak, mint a bemenet n méretétől való függés. Míg az összefésülő rendezés futási idejében egy $\lg n$ tényező van, a beszúró rendezés tényezője n, amely sokkal nagyobb. Bár kisméretű bemenetek esetén a beszúró rendezés gyorsabb, mint az összefésülő rendezés, ha a bemenet n mérete elég nagy, akkor az összefésülő rendezés előnye, azaz $\lg n$ az n-nel szemben bőven kompenzálja az állandó tényezőkben rejlő különbséget. Mindegy, hányszor kisebb c_1, mint c_2, mindig lesz egy váltási érték, amely felett az összefésülő rendezés gyorsabb.

Konkrét példaként állítsunk szembe egy gyorsabb (A) számítógépet, amelyen a beszúró rendezés fut, egy lassabb (B) számítógéppel, amelyen az összefésülő rendezés fut. Mindegyiknek egy egymillió elemet tartalmazó tömböt kell rendeznie. Tegyük fel, hogy az A számítógép másodpercenként 1 milliárd műveletet hajt végre, a B számítógép pedig másodpercenként 10 millió műveletet, azaz az A számítógép nyers erőben százszor gyorsabb, mint a B számítógép. Azért, hogy a különbség még drámaibb legyen, tegyük fel, hogy a világ legjobb programozója kódolja a beszúró rendezést gépi kódban az A számítógépre, és az így kapott kódnak $2n^2$ utasításra van szüksége ahhoz, hogy n számot rendezzen. (Azaz itt $c_1 = 2$.) Az összefésülő rendezést a B számítógépre egy átlagos programozó kódolta magas szintű nyelven, a fordítóprogram nem volt hatékony, így a kapott kód $50n \lg n$ utasítást igényel n szám rendezéséhez. Egymillió szám rendezéséhez az A számítógép

$$\frac{2 \cdot \left(10^6\right)^2 \text{ művelet}}{10^9 \text{ művelet/s}} = 2000\text{s}, \tag{1.1}$$

míg a B számítógép

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ művelet}}{10^7 \text{ művelet/s}} \approx 100\text{s} \tag{1.2}$$

időt használ fel. Egy olyan algoritmus felhasználásával, amelynek a futási ideje lassabban nő, gyenge fordítóprogrammal, a B számítógép hússzor rövidebb idő alatt végez, mint az A számítógép! Az összefésülő rendezés előnye még jobban kidomborodik, ha 10 millió számot kell rendezni: ekkor a beszúró rendezés körülbelül 2,3 napig fut, míg az összefésülő rendezés mindössze 20 percig. Általában, ahogy a feladat mérete nő, úgy nő az összefésülő rendezés relatív előnye.

Algoritmusok és más technológiák

Az előző példák azt mutatják, hogy az algoritmusok – a számítógépek hardveréhez hasonlóan – ***technológiák.*** Egy rendszer teljesítménye éppúgy függ a hatékony algoritmusok kiválasztásától, mint a gyors hardver alkalmazásától. Amilyen gyors a haladás a többi számítógépes technológiában, éppoly gyors a fejlődés az algoritmusokkal kapcsolatban is.

Az Olvasó csodálkozhat, vajon az algoritmusok valóban olyan fontosak a korszerű számítógépekben, mint a többi fejlett technológia, például

- az időegységenként nagyszámú órajelet, csővezetéket és szuperskalár architektúrákat alkalmazó hardver,
- a könnyen alkalmazható, intuitív grafikus felhasználói felületek,
- az objektumelvű rendszerek,
- a helyi és az osztott hálózatok.

A válasz igen. Bár vannak bizonyos alkalmazások, amelyek alkalmazói szinten nem igényelnek algoritmust (például bizonyos egyszerű web-alapú alkalmazások), a legtöbb alkalmazás igényel bizonyos szintű algoritmikus tartalmat. Például tekintsünk egy olyan web alapú szolgáltatást, amely meghatározza, hogyan utazzunk el egyik helyről a másikra. (A könyv írása idején több ilyen szolgáltatás is létezett.) Megvalósítása gyors hardveren, grafikus felhasználói felületen, távoli hálózaton és esetleg objektumelvűségen alapul. Azonban valószínűleg bizonyos műveletekhez algoritmusokat igényel, mint útvonal megtalálása (valószínűleg legrövidebb út algoritmus), térképek rajzolása és címek interpolálása.

Továbbá, még ha egy alkalmazásnak az alkalmazás szintjén nincs is algoritmikus tartalma, erősen támaszkodhat algoritmusokra. Támaszkodik az alkalmazás gyors hardverre? A hardvertervezés algoritmusokat használt. Támaszkodik az alkalmazás grafikus felhasználói felületre? A csomagirányítás a hálózatokban erősen támaszkodik algoritmusokra. Az alkalmazás a gépi kódtól különböző nyelven íródott? Akkor egy fordítóprogram, interpreter vagy assembler dolgozta fel, amelyek mindegyike intenzíven használ algoritmusokat. Az algoritmusok a modern számítógépekben használt legtöbb technológia lényeges részét alkotják.

Továbbá, az állandóan növekvő kapacitású számítógépeket nagyobb méretű feladatok megoldására használjuk, mint korábban bármikor. Amint azt láttuk a beszúró rendezés és az összefésülő rendezés fenti összehasonlításánál, nagyobb méretű feladatok esetén az algoritmusok hatékonyságának különbsége különösen fontossá válik.

Az algoritmusok és módszerek biztos ismerete olyan jellemző, amely elválasztja a jól képzett programozókat a kezdőktől. A modern számítógépes technológia segítségével megoldhatunk bizonyos feladatokat anélkül, hogy sokat tudnánk az algoritmusokról, de az algoritmusokkal kapcsolatos jó alapok megléte esetén sokkal-sokkal többet tudunk elérni.

Gyakorlatok

1.2-1. Adjunk példát egy olyan alkalmazásra, amelynek az alkalmazás szintjén algoritmikus tartalma van, és elemezzük az alkalmazásban foglalt algoritmust.

1.2-2. Tegyük fel, hogy a beszúró rendezés és az összefésülő rendezés ugyanazon a gépen való megvalósításait hasonlítjuk össze. n méretű bemenetekre a beszúró rendezés $8n^2$ lépést végez, míg az összefésülő rendezés $64n \lg n$ lépést. Milyen n értékekre jobb a beszúró rendezés, mint az összefésülő rendezés?

1.2-3. Határozzuk meg a legkisebb olyan n értéket, amelyre a $100n^2$ futási idejű algoritmus gyorsabb, mint az az algoritmus, melynek ugyanazon a gépen 2^n a futási ideje.

Feladatok

1-1. Futási idők összehasonlítása

A következő táblázat minden $f(n)$ függvényére és t idejére határozzuk meg a probléma legnagyobb n méretét, amely még megoldható t idő alatt, feltételezve, hogy a probléma megoldása az algoritmusnak $f(n)$ mikromásodpercig tart.

	1 másodperc	1 perc	1 óra	1 nap	1 hónap	1 év	1 évszázad
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Megjegyzések a fejezethez

Sok kitűnő anyag áll rendelkezésre az algoritmusokkal kapcsolatban, ezen belül Aho, Hopcroft és Ullman [5, 6], Baase [26], Brassard és Bratley [46, 47], Goodrich és Tamassia [128], Horowitz, Sahni és Rajasekaran [157], Kingston [179], Knuth [182, 183, 185], Kozen [193], Manber [210], Mehlhorn [219, 217, 218], Purdom és Brown [252], Reingold, Nievergelt és Deo [257], Sedgewick [269], Skiena [280] és Wilf [315]. Az algoritmustervezés néhány gyakorlatiasabb szempontját Bentley [39, 40] és Gonnet [126] tárgyalja. Az algoritmusokról szóló összefoglaló található a van Leeuwen [302] és az Atallah [24] által szerkesztett kézikönyvekben. A biológiában használt számítási algoritmusok áttekintése megtalálható Gusfield [136], Pevzner [240], Setubal és Meidanis [272], valamint Waterman [309] könyvében.

2. Elindulunk

Ebben a fejezetben megismerkedünk azokkal az alapvető fogalmakkal és eszközökkel, amelyeket a könyvben az algoritmusok tervezése és elemzése során használni fogunk. A fejezet önmagában is megállja a helyét, de tartalmaz számos hivatkozást olyan anyagokra, amelyeket a 3. és 4. fejezetben fogunk bemutatni. (Több olyan összegformulát is tartalmaz, amelyeknek a bizonyítását majd az A függelékben mutatjuk be.)

Az 1. fejezetben definiált rendezési feladatot megoldó, beszúró rendező algoritmussal kezdjük. Bemutatunk egy „pszeudokódot”, amely ismerős lesz mindazon Olvasóknak, akik már foglalkoztak számítógép-programozással. A kód segítségével megmutatjuk, hogyan fogjuk algoritmusainkat megadni. Miután leírtuk az algoritmust, belátjuk, hogy helyesen rendez, azután elemezzük a futási idejét. Az elemzés során bevezetünk egy jelölést annak kifejezésére, hogyan nő a futási idő a rendezendő elemek számának növekedésével. A beszúró rendezés elemzése után bemutatjuk az algoritmusok elemzésének oszd-meg-és-uralkodj megközelítését, és ezt a megközelítést felhasználva eljutunk az összefésülő rendező algoritmushoz. A fejezetet az összefésülő rendezés futási idejének elemzése zárja.

2.1. Beszúró rendezés

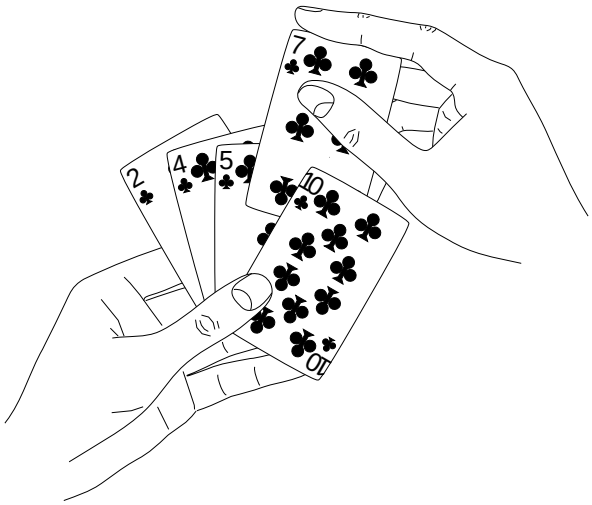
Első algoritmusunk a beszúró rendezés, amely megoldja az 1. fejezetben már bemutatott ***rendezési feladatot:***

Bemenet: n számot tartalmazó $\langle a_1, a_2, \ldots, a_n \rangle$ sorozat.

Kimenet: a bemenő sorozat olyan $\langle a'_1, a'_2, \ldots, a'_n \rangle$ permutációja (újrarendezése), hogy $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

A rendezendő számokat ***kulcsoknak*** is szokás nevezni.

Ebben a könyvben rendszerint egy olyan ***pszeudokódban*** írt programként írjuk le az algoritmusokat, amely nagyon hasonlít a C-hez, a Pascalhoz vagy a Javához. Ha ezek közül bármelyik nyelvet ismeri, nem lesz gondja az algoritmusok olvasása során. Ami az „igazi” kódot megkülönbözteti a pszeudokódtól, az az, hogy pszeudokódban bármilyen kifejező eszközt alkalmazhatunk, amellyel világosan és tömören megadhatunk egy algoritmust. Néha a legjobb eszköz a magyar nyelv, így ne lepődjünk meg, ha magyar kifejezéssel vagy mondattal találkozunk az „igazi” kódba ágyazva. Egy másik különbség az igazi



2.1. ábra. Kézben tartott lapok rendezése beszúró rendezéssel.

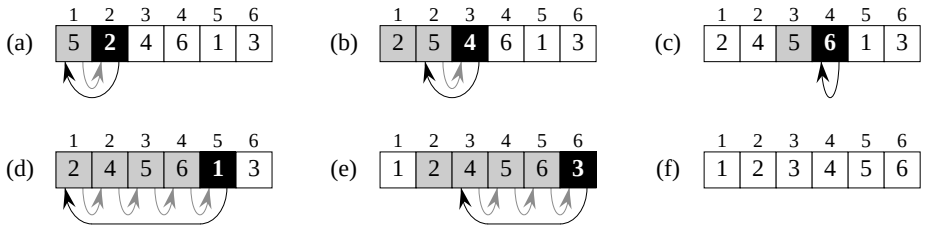
és a pszeudokód között, hogy a pszeudokód rendszerint nem foglalkozik szoftvertervezési kérdésekkel. Adatáltalánosítási, modularitási és hibakezelési feladatokat gyakran figyelmen kívül hagyunk annak érdekében, hogy az algoritmus lényegét tömörebben visszaadhassuk.

A ***beszúró rendezéssel*** kezdjük, amely hatékony algoritmus kisszámú elem rendezésére. A beszúró rendezés úgy dolgozik, ahogy az ember bridzsnél vagy röminél a kezében lév ő lapokat rendezi. Üres bal kézzel kezdünk, a lapok fejjel lefelé az asztalon fekszenek. Egy lapot felveszünk az asztalról, és elhelyezzük a bal kezünkben a megfelel ő helyre. Ahhoz, hogy megtaláljuk a megfelelő helyet, a felvett lapot összehasonlítjuk a már kezünkben lév ő lapokkal, jobbról balra, ahogy a 2.1. ábra mutatja. A kezünkben tartott lapok minden esetben rendezettek, és ők az eredetileg az asztalon fekvők közül a legfelső lapok voltak.

A beszúró rendezés pszeudokódját egy BESZ ÚRÓ-RENDEZÉS nevű eljárásként mutatjuk be, amelynek paramétere egy n hosszúságú, a rendezend ő számok sorozatát tartalmazó $A[1\,..\,n]$ tömb. (A kódban A elemeinek n számát $hossz[A]$-val jelöljük.) A bemen ő elemek ***helyben*** rendeződnek: a számokat az eljárás az A tömbön belül rakja a helyes sorrendbe, bel őlük bármikor legfeljebb csak állandó számú tárolódik a tömbön kívül. Amikor a BESZ ÚRÓ-RENDEZÉS befejeződik, az A tömb tartalmazza a rendezett elemeket.

BESZÚRÓ-RENDEZÉS(A)

```
1  for j ← 2 to hossz[A]
2      do kulcs ← A[j]
3         ▷ A[j] beszúrása az A[1 . . j − 1] rendezett sorozatba.
4         i ← j − 1
5         while i > 0 és A[i] > kulcs
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8         A[i + 1] ← kulcs
```



2.2. ábra. A BESZÚRÓ-RENDEZÉS működése az $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ tömbön. A tömbindexek a téglalapok felett, a tömbelemekben tárolt értékek pedig a téglalapokban láthatók. **(a)–(e)** Az 1–8. sorokban szereplő **for** ciklus iterációi. Mindegyik iterációs lépésben a fekete téglalap tartalmazza az $A[j]$ elemből vett értéket, amely az 5. sor szerint összehasonlítódik a tőle balra lévő szürke tömbelemekkel. A szürke nyilak azokra a tömbelemekre mutatnak, amelyek a 6. sor szerint egy hellyel jobbra tolódtak. A fekete nyilak azt mutatják, hová kerül a kulcs a 8. sorban. **(f)** A végső, rendezett tömb.

A beszúró rendezés ciklusinvariánsai és helyessége

A 2.2. ábra azt mutatja, hogyan működik ez az algoritmus az $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ értékekre. A j index jelöli az éppen a kézbe beszúrandó *aktuális lapot*. A „külső” **for** ciklus – amelyet j-vel indexelünk – minden iterációjának kezdetén az $A[1 \,..\, j-1]$ elemekből álló résztömb alkotja a már rendezett kezet, az $A[j+1 \,..\, n]$ elemek pedig a még az asztalon fekvő laphalomnak felelnek meg. Valójában az $A[1 \,..\, j-1]$ elemek azok, amelyek *eredetileg* is az 1., 2., ..., $(j-1)$. pozíciókban voltak, de most már rendezett formában. Az $A[1 \,..\, j-1]$ résztömb ezen tulajdonságait formálisan ***ciklusinvariánsként*** fogalmazzuk meg:

> Az 1–8. sorokbeli **for** ciklus minden iterációjának kezdetén az A[1 . . j − 1] résztömb azon elemekből áll, amelyek eredetileg is az A[1 . . j − 1] résztömbben voltak, de most már rendezett sorrendben.

A ciklusinvariánsokat arra használjuk, hogy megértsük, miért helyes egy algoritmus. Egy ciklusinvariánsról három dolgot kell megmutatnunk:

Teljesül: Igaz közvetlenül a ciklus első iterációjának megkezdése előtt.

Megmarad: Ha igaz a ciklus egy iterációjának megkezdése előtt, akkor igaz marad a következő iteráció előtt is.

Befejeződik: Amikor a ciklus befejeződik, az invariáns olyan hasznos tulajdonságot ír le, amely segít abban, hogy az algoritmus helyességét bebizonyítsuk.

Ha az első két feltétel teljesül, akkor a ciklusinvariáns a ciklus minden iterációja előtt teljesül. Vegyük észre a teljes indukcióhoz való hasonlóságot, ahol egy tulajdonság teljesülését úgy mutatjuk meg, hogy egy kezdőértékre bebizonyítjuk, és az induktív lépés helyességét belátjuk. Az invariánsnak az első iteráció előtt való teljesülése a kezdőértékhez hasonlít, az invariánsnak lépésről lépésre való teljesülése pedig az induktív lépéshez.

Talán a harmadik tulajdonság a legfontosabb, mivel a ciklusinvariánst a helyesség bizonyítására használjuk. Ez a tulajdonság a teljes indukciótól való eltérésre utal, mivel ott az induktív lépést végtelen sokszor alkalmazzuk; itt megállítjuk az „indukciót”, amikor a ciklus véget ér.

Vizsgáljuk meg, hogyan teljesülnek ezek a tulajdonságok a beszúró rendezés esetén.

Teljesül: Azt látjuk be először, hogy a ciklusinvariáns teljesül az első ciklusiteráció előtt, amikor is $j = 2$.[1] Az $A[1\,..\,j-1]$ résztömb így pontosan az $A[1]$ elemből áll, amely valóban az eredeti $A[1]$ elem. Továbbá, ez a résztömb rendezett (természetesen), azaz a ciklusinvariáns a ciklus első iterációja előtt teljesül.

Megmarad: Ezután a második tulajdonsággal foglalkozunk: megmutatjuk, hogy minden iteráció fenntartja a ciklusinvariánst. Informálisan a külső **for** ciklus magja az $A[j-1]$, $A[j-2]$, $A[j-3]$ stb. elemeket mozgatja jobbra mindaddig, amíg $A[j]$ a helyes pozíciót el nem éri (4–7. sorok), amikor is az $A[j]$ elemet beszúrja (8. sor). A második tulajdonság formálisabb tárgyalása azt igényelné, hogy fogalmazzunk meg és bizonyítsunk egy ciklusinvariánst a „belső" **while** ciklusra. Itt azonban inkább nem bonyolódunk bele ilyen formalizmusba, hanem az informális elemzésre hagyatkozunk, amely szerint a második tulajdonság teljesül a külső ciklusra nézve.

Befejeződik: Végezetül megvizsgáljuk, hogy mi történik a ciklus befejeződésekor. A beszúró rendezés esetén a külső **for** ciklus akkor ér véget, amikor j meghaladja n-et, azaz, amikor $j = n+1$. Ha a ciklusinvariáns megfogalmazásában $(n+1)$-et írunk j helyett, akkor azt kapjuk, hogy az $A[1\,..\,n]$ résztömb azokból az elemekből áll, amelyek eredetileg az $A[1\,..\,n]$ elemek voltak, de már rendezett formában. Az $A[1\,..\,n]$ résztömb azonban az egész tömb! Tehát az egész tömb rendezve van, azaz az algoritmus helyes.

A ciklusinvariánsok ezen módszerét ebben és a későbbi fejezetekben helyességbizonyításra fogjuk felhasználni.

Pszeudokód megállapodások

Pszeudokódunkban a következő megállapodásokat alkalmazzuk.

1. A tagolás a blokkszerkezetet jelzi. Például az 1. sorban kezdődő **for** ciklus magja a 2–8. sorokból áll, az 5. sorban kezdődő **while** ciklus magja a 6–7. sorokat tartalmazza, de a 8.-at nem. Ez a tagolásos szerkezet vonatkozik az **if-then-else** utasításokra is. Tagolást használva a blokkszerkezet olyan hagyományos jelölései helyett, mint a **begin** és az **end** utasítások, csökken a zsúfoltság, és megmarad – sőt javul – az érthetőség.[2]
2. A **while, for** és **repeat** ciklusutasítások, valamint az **if, then** és **else** feltételes szerkezetek értelmezése ugyanolyan, mint a Pascalban.[3] A **for** utasítással kapcsolatban van azonban egy lényeges különbség a Pascalhoz képest, ahol a ciklusból kilépve a ciklusváltozó értéke definiálatlan. Ebben a könyvben azonban a ciklusváltozó megtartja értékét a ciklusból való kilépés után. Így például közvetlenül a **for** ciklusból való kilépés után a ciklusváltozó értéke az, amely először lépte át a ciklusváltozóra megadott korlátot. Ezt a tulajdonságot felhasználtuk a beszúró rendezés helyességének bizonyításakor. A **for** ciklus feje az első sorban **for** $j \leftarrow 2$ **to** $hossz[A]$, és így amikor a ciklus befejeződik, $j = hossz[A]+1$ (vagy, ami ezzel ekvivalens, $j = n+1$, mivel $hossz[A] = n$).

[1] **for** ciklusokban a ciklusinvariánst az első iteráció előtt ellenőrizzük, közvetlenül azután, hogy a ciklusváltozóhoz a kezdeti értéket hozzárendeltük, és közvetlenül azelőtt, hogy a ciklusfejben az első tesztet elvégeznénk. A BESZÚRÓ-RENDEZÉS esetén ez az időpont azután van, hogy a 2 értéket hozzárendeltük a j változóhoz, de annak első vizsgálata előtt, hogy $j \leq hossz[A]$.

[2] Az igazi programnyelvekben nem tanácsos csak tagolást használni a blokkszerkezet jelölésére, mivel a bekezdések mélységét nehéz meghatározni, amikor a kód megtörik az oldalak között.

[3] A legtöbb blokkszerkezetű nyelvben vannak ezzel ekvivalens elemek, bár a pontos szintaxis különbözhet a Pascaltól.

3. A „▷” jel azt mutatja, hogy a sor többi része megjegyzés.
4. Az $i \leftarrow j \leftarrow e$ többszörös értékadás eredményeképp i és j egyaránt felveszi az e kifejezés értékét; ez ekvivalens a $j \leftarrow e$, majd azt követő $i \leftarrow j$ értékadással.
5. A változók (például i, j és *kulcs*) az adott eljárásra lokálisak. Külön jelzés nélkül nem használunk globális változókat.
6. Egy tömb elemeihez a tömb nevének megjelölésével és az index szögletes zárójelbe helyezésével férhetünk hozzá. Például az $A[i]$ A i-edik elemét jelenti. A ”. .” jelzést a tömbön belüli értékek tartományának jelzésére használjuk. Így $A[1\,..\,j]$ az $A[1], A[2], \ldots, A[j]$ elemekből álló A résztömböt jelöli.
7. Az összetett adatokat szokás ***objektumoknak*** is nevezni, amelyeket tulajdonságokkal ***(attribútumokkal*** vagy ***mezőkkel)*** jellemezhetünk. Egy bizonyos mezőhöz úgy férhetünk hozzá, hogy szögletes zárójelbe téve megadjuk az objektum nevét, majd elé írjuk a mező nevét. Például, ha egy tömböt objektumként kezelünk, a *hossz* tulajdonság jelöli, hány elemet tartalmaz. Egy A tömbben az elemek számának meghatározásához *hossz*$[A]$-t írunk. Bár szögletes zárójelet használunk a tömb indexeinél és az objektum tulajdonságainál is, a szövegösszefüggésből általában egyértelműen kiderül, melyik értelmezésre gondolunk.

   Egy tömböt (vagy objektumot) ábrázoló változót a tömböt ábrázoló adat mutatójaként kezelünk. Egy x objektum minden f mezőjére az $y \leftarrow x$ jelölés eredménye $f[y] = f[x]$. Sőt, ezután az $f[x] \leftarrow 3$ utasítás hatására nemcsak $f(x) = 3$, hanem $f(y) = 3$ is teljesül. Más szóval az x és y az $y \leftarrow x$ jelölés után ugyanarra az objektumra mutatnak.

   Néha egy mutató egyáltalán nem utal semmilyen objektumra. Ebben az esetben a NIL különleges értéket adjuk neki.
8. Az eljárások paraméterei ***érték szerint*** adódnak át: a hívott eljárás megkapja a paraméterek másolatát, és ha értéket rendel a paraméterhez, a változást a hívó rutin *nem* látja. Amikor objektum adódik át, az objektumot ábrázoló adat mutatója átadódik, az objektum mezői azonban nem. Például, ha x a hívott eljárás paramétere, a hívott eljáráson belüli $x \leftarrow y$ értékadás nem látható a hívó eljárás számára. Az $f[x] \leftarrow 3$ értékadás azonban látható.
9. Az „és” és „vagy” operátorok ***gyors kiértékelésűek.*** Ez azt jelenti, hogy amikor az „x és y” kifejezést kiértékeljük, először csak x-et értékeljük ki. Ha x értéke HAMIS, akkor az egész kifejezés értéke nem lehet IGAZ, ezért nem értékeljük ki y-t. Másrészt, ha x értéke IGAZ, ki kell értékelnünk y-t, hogy a teljes kifejezés értékét megkapjuk. Hasonlóképpen, az „x vagy y” kifejezésben csak akkor értékeljük ki y-t, ha x értéke HAMIS. A gyors kiértékelésű operátorok lehetővé teszik, hogy anélkül írjunk le olyan logikai kifejezéseket, mint „$x \neq$ NIL és $f[x] = y$”, hogy nyugtalankodnánk amiatt, mi történik akkor, ha megpróbáljuk $f[x]$-et kiértékelni, amikor x értéke NIL.

Gyakorlatok

2.1-1. A 2.2. ábrát modellként használva, ábrázoljuk a BESZÚRÓ-RENDEZÉS működését az $A = \langle 31, 41, 59, 26, 41, 58 \rangle$ tömbön.

2.1-2. Írjuk át a BESZÚRÓ-RENDEZÉS eljárást úgy, hogy nemcsökkenő helyett nemnövekvő sorrendbe rendezzen.

2.1-3. Vizsgáljuk meg a ***keresési feladatot.***

Bemenet: $A = \langle a_1, a_2, \ldots, a_n \rangle$ n számból álló sorozat és v érték.

Kimenet: Olyan i index, hogy $v = A[i]$ vagy a NIL különleges érték (ha v nem jelenik meg A-ban).

Írjunk pszeudokódot a ***lineáris keresésre,*** amely v-t keresve végigpásztázza a sorozatot. Ciklusinvariáns felhasználásával mutassuk meg, hogy algoritmusunk helyes. Gondosan vizsgáljuk meg, teljesíti-e ciklusinvariánsunk mind a három feltételt.

2.1-4. Vizsgáljuk meg azt a feladatot, melyben össze kell adnunk két n bites egész számot, melyeket két n elemű tömbben, A-ban és B-ben tárolunk. A két egész összegét bináris formában kell tárolnunk egy $(n + 1)$ elemű C tömbben. Fogalmazzuk meg a feladatot formálisan, és írjunk pszeudokódot a két egész szám összeadására.

2.2. Algoritmusok elemzése

Egy algoritmus ***elemzése*** azt jelenti, hogy előre megmondjuk, milyen erőforrásokra lesz szüksége az algoritmusnak. Alkalmanként az olyan erőforrások, mint memória, kommunikációs sávszélesség vagy logikai kapuk, elsődleges fontosságúak, de leggyakrabban a számítási idő az, amit mérni akarunk. Általában az adott feladat megoldására szóba jövő algoritmusok elemzésével könnyen meghatározható a leghatékonyabb. Az efféle elemzés adhat egynél több megfelelő jelöltet is, de számos gyenge algoritmus rendszerint kiesik az eljárás során.

Mielőtt elemeznénk egy algoritmust, rendelkeznünk kell a felhasználandó megvalósítási technológia modelljével, beleértve a technológia erőforrásainak modelljét és azok költségeit is. E könyv nagy részében számítási modellként egy általános feldolgozó egységet, az ún. ***közvetlen hozzáférésű gépet (random access machine = RAM)*** alkalmazzuk megvalósítási technológiaként, és algoritmusainkat úgy értelmezzük, hogy számítógépprogramként valósítjuk meg őket. A RAM modellben az utasítások egymás után hajtódnak végre, egyidejű műveletek nélkül. A későbbi fejezetekben pedig arra is lesz alkalmunk, hogy digitális hardverre való modelleket is megvizsgáljunk.

Szigorúan fogalmazva, pontosan definiálnunk kell a RAM modell műveleteit és azok költségét. Ez azonban unalmas lenne és kevés betekintést engedne az algoritmusok tervezésébe és elemzésébe. És nagyon gondosnak kellene lennünk, hogy ne éljünk vissza a RAM modellel. Például mi lenne akkor, ha a RAM modellben szerepelne egy rendező utasítás? Akkor egyetlen utasítással rendezhetnénk. Egy ilyen RAM modell nem lenne életszerű, mivel a valódi számítógépek nem rendelkeznek ilyen utasításokkal. A mi vezérfonalunk ezért az, ahogyan a valódi számítógépek vannak megtervezve. A RAM modell olyan utasításokat tartalmaz, amelyek rendszerint megtalálhatók a valódi számítógépeken: aritmetikai (összeadás, kivonás, szorzás, osztás, maradékképzés, alsó egész rész, felső egész rész), adatmozgató (betöltés, tárolás, másolás) és vezérlésátadó (feltételes és feltétel nélküli elágazás, eljáráshívás és visszatérés). Minden ilyen utasítás konstans hosszúságú időt vesz igénybe.

A RAM modell adattípusai az egész és a lebegőpontos. Bár ebben a könyvben rendszerint nem foglalkozunk a pontossággal, az bizonyos alkalmazásokban döntő fontosságú. Feltételezzük továbbá, hogy minden adatszó mérete korlátos. Például amikor n méretű bemenetekkel dolgozunk, rendszerint feltesszük, hogy az egészeket $c \lg n$ bittel ábrázoljuk,

ahol $c \geq 1$ állandó. Azért használjuk a $c \geq 1$ feltételt, hogy minden szó tartalmazni tudja az n értéket, és így indexelni tudjuk az egyes bemenő elemeket, és azért kötjük ki, hogy c állandó legyen, hogy a szavak mérete ne nőhessen tetszőlegesen. (Ha a szavak mérete tetszőlegesen nőhet, hatalmas mennyiségű adatot tárolhatunk egyetlen szóban és konstans idő alatt végezhetünk ezekkel az adatokkal műveleteket, ami nyilvánvalóan nem reális.)

A valódi számítógépek olyan utasításokat is tartalmaznak, amelyek a fenti listában nem szerepelnek. Ezek az utasítások fehér foltot képeznek a RAM modellekkel kapcsolatban. Például a hatványozás konstans idejű művelet? Általában nem: ha x és y valós számok, akkor x^y kiszámítása több műveletet igényel. Bizonyos helyzetekben azonban a hatványozás állandó idejű művelet. Sok számítógép rendelkezik „eltolás balra” művelettel, amely egy egész szám bitjeit állandó idő alatt balra tolja k bittel. Egy egész szám bitjeinek balra tolása egy hellyel a legtöbb számítógépben egyenértékű a 2-vel való szorzással. A bitek k hellyel való balra tolása egyenértékű a 2^k-val való szorzással. Ezért az ilyen számítógépek a 2^k-t konstans idő alatt ki tudják számítani úgy, hogy az 1 szám bitjeit k hellyel balra tolják – amennyiben k nem nagyobb, mint a számítógép egy szavában lévő bitek száma. Igyekszünk elkerülni ezeket a RAM modellel kapcsolatos fehér foltokat, de a 2^k kiszámítását állandó idejű műveletnek fogjuk tekinteni, amennyiben k elég kicsi pozitív egész.

A RAM modellben nem teszünk kísérletet a korszerű számítógépekben szokásos hierarchikus memóriafelépítés leírására. Azaz nem modellezzük a gyorsítótárat és a virtuális memóriát (amelyet gyakran igény szerinti lapozással valósítanak meg). Több olyan számítási modell ismert, amely igyekszik figyelembe venni a hierarchikus memória felépítését, ami esetenként a valódi programokban és számítógépekben nagyon lényeges. Ebben a könyvben több feladatban vizsgáljuk a hierarchikus memória hatásait, de a könyv nagyobb részében az elemzések nem veszik figyelembe ezeket a hatásokat. A hierarchikus memóriát leíró modellek lényegesen bonyolultabbak, mint a RAM modell, ezért lényegesen nehezebb velük dolgozni. Továbbá a RAM modell segítségével végzett elemzések rendszerint nagyon jól jelzik előre a valódi gépeken mutatott teljesítményt.

Még egy RAM modellben megadott egyszerű algoritmus elemzése is lehet kihívás. A szükséges matematikai eszközök között szerepelhet kombinatorika, elemi valószínűségszámítás, algebrai ügyesség és az a képesség, hogy azonosítani tudjuk egy képlet legfontosabb tagjait. Mivel az algoritmusok viselkedése bemenetenként különböző lehet, szükségünk van olyan eszközre, amely összefoglalja ezt a viselkedést egyszerű, könnyen érthető képletekben.

Noha rendszerint csak egy számítási modellt választunk ki egy adott algoritmus elemzésére, még mindig több lehetőségünk van annak eldöntésére, hogyan fejezzük ki elemzésünk eredményét. Közvetlen célunk, hogy olyan kifejezési eszközt találjunk, amelyik könnyen írható és kezelhető, megmutatja az algoritmus erőforrás-szükségletének fontos jellemzőit, és figyelmen kívül hagyja az unalmas részleteket.

A beszúró rendezés elemzése

A Beszúró-rendezés eljárás által felhasznált idő a bemenettől függ: ezer szám rendezése tovább tart, mint három számé. Sőt, a Beszúró-rendezés akár két ugyanolyan méretű bemenő sorozatot is rendezhet különböző idő alatt attól függően, hogy azok mennyire vannak már eleve rendezve. Általában egy algoritmus által felhasznált idő a bemenet méretével nő, így hagyományosan egy program futási idejét bemenete méretének függvényével

írjuk le. Ehhez pontosabban kell definiálnunk a „futási idő” és a „bemenet mérete” kifejezéseket.

A ***bemenet méretének*** legjobb mértéke a vizsgált feladattól függ. Sok feladatra, mint például a rendezés vagy a diszkrét Fourier-transzformáltak kiszámítása, a legtermészetesebb mérték a *bemenő elemek száma.* Például rendezésnél a rendezendő tömb n elemszáma lehet a mérték. Sok egyéb feladatra, mint például két egész szám szorzása, a bemenet méretére a legjobb mérték a bemenet közönséges bináris jelölésben való ábrázolásához szükséges *bitek teljes száma*. Néha megfelelőbb a bemenet méretét egy szám helyett inkább kettővel leírni. Például, ha az algoritmus bemenete egy gráf, a bemenet méretét leírhatjuk a gráf éleinek és csúcsainak számával. Minden vizsgált feladatnál megadjuk, hogy a bemenet méretének melyik mértékét használjuk.

Az algoritmusok ***futási ideje*** egy bizonyos bemenetre a végrehajtott alapműveletek vagy „lépések” száma. Kényelmes úgy definiálni a lépést, hogy minél inkább gépfüggetlen legyen. Egyelőre fogadjuk el a következőt: pszeudokódunk mindegyik sorának végrehajtásához állandó mennyiségű idő szükséges. Lehet, hogy az egyik sor tovább tart, mint a másik, de feltesszük, hogy az i-edik sor minden végrehajtása c_i ideig tart, ahol c_i állandó. Ez a nézőpont megfelel a RAM modellnek, és tükrözi azt is, ahogyan a pszeudokódot végrehajtaná a legtöbb jelenlegi számítógép.[4]

A következő elemzésben a BESZÚRÓ-RENDEZÉS futási idejét a – minden egyes utasítás c_i végrehajtási idejét figyelembe vevő – bonyolult képletből egyszerűbb, tömörebb és könnyebben kezelhető alakra hozzuk. Ez az egyszerűbb alak annak eldöntését is megkönnyíti, hogy egy algoritmus hatékonyabb-e egy másiknál.

A BESZÚRÓ-RENDEZÉS eljárás bemutatását az utasítások „időköltségével” és az egyes utasítások végrehajtásainak számával kezdjük. Minden $j = 2, 3, \ldots n$-re, ahol $n = hossz[A]$, t_j legyen az a szám, ahányszor a **while** ciklus 5. sorban lévő tesztje végrehajtódik az adott j értékre. Ha egy **for** vagy **while** ciklusból a szokásos módon lépünk ki (azaz a ciklusfejben lévő teszt eredményeképpen), akkor a teszt eggyel többször hajtódik végre, mint a ciklusmag. Feltesszük, hogy a megjegyzések nem végrehajtható utasítások, így nem vesznek igénybe időt.

BESZÚRÓ-RENDEZÉS(A)	*költség*	*végrehajtási szám*
1 **for** $j \leftarrow 2$ **to** $hossz[A]$	c_1	n
2 **do** $kulcs \leftarrow A[j]$	c_2	$n-1$
3 ▷$A[j]$ beszúrása az $A[1\,..\,j-1]$ rendezett sorozatba.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ *és* $A[i] > kulcs$	c_5	$\sum_{n}^{j=2} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{n}^{j=2} t_{j-1}$
7 $i \leftarrow i-1$	c_7	$\sum_{n}^{j=2} t_{j-1}$
8 $A[i+1] \leftarrow kulcs$	c_8	$n-1$

[4]Itt tegyünk néhány észrevételt. Bizonyos számítási lépések, amelyeket magyarul adunk meg, gyakran egy olyan eljárás változatai, amelynek végrehajtása különböző ideig tart. Például ebben a könyvben később azt mondhatnánk, hogy „rendezzük a pontokat x koordinátájuk szerint”, ami, ahogy látni fogjuk, állandó időnél tovább tart. Jegyezzük meg azt is, hogy egy szubrutint hívó utasítás végrehajtása állandó ideig tart, de a már egyszer hívott szubrutin tovább tarthat. Azaz megkülönböztetjük a szubrutin ***hívását*** – a paraméterek átadását stb. – a szubrutin ***végrehajtásának*** folyamatától.

Az algoritmus futási ideje a végrehajtott utasítások végrehajtási időinek összege; egy utasítás, amelynek végrehajtásához c_i lépés szükséges, és n-szer hajtódik végre, a teljes futási időhöz $c_i n$-nel járul hozzá.[5] A BESZÚRÓ-RENDEZÉS $T(n)$ futási ideje tehát az egyes sorokhoz tartozó *költségek* és *végrehajtási számok* szorzatösszege:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1). \end{aligned}$$

Még adott méretű bemeneteknél is függhet az algoritmus futási ideje attól, hogy az algoritmus az adott méretű bemenetek közül *melyiket* kapja. Például a BESZÚRÓ-RENDEZÉS esetén a legjobb eset akkor fordul elő, amikor a tömb már eleve rendezett. Ekkor minden $j = 2, 3, \ldots, n$-re azt találjuk, hogy $A[i] \leq kulcs$ az 5. sorban, amikor i kezdeti értéke $j - 1$. Így $t_j = 1$ (ha $j = 2, 3, \ldots, n$) és a futási idő a legjobb esetben

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ezt a futási időt kifejezhetjük úgy is, hogy $an + b$, ahol az a és b állandók az utasítások c_i költségeitől függnek; így ez n ***lineáris függvénye.***

Ha a tömb fordított sorrendben rendezett – azaz csökkenő sorrendben –, akkor fordul elő a legrosszabb eset. Minden $A[j]$ elemet össze kell hasonlítanunk az egész $A[1 \ldots j-1]$ rendezett résztömb minden elemével, és így $t_j = j$ (ha $j = 2, 3, \ldots, n$). Felhasználva, hogy

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

és

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2},$$

(lásd az A függeléket, hogy miként lehet ezeket az összegzéseket elvégezni), akkor azt találjuk, hogy BESZÚRÓ-RENDEZÉS futási ideje legrosszabb esetben

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ezt a legrosszabb futási időt ki lehet fejezni $(an^2 + bn + c)$ alakban a, b és c állandókkal, amelyek ismét az utasítások c_i költségeitől függnek; így ez n ***négyzetes függvénye.***

[5] Ez nem feltétlenül igaz egy olyan erőforrásra, mint a memória. Nem biztos, hogy egy utasítás, amely a memória m szavára hivatkozik és n-szer hajtódik végre, a memóriából összesen mn szót fogyaszt.

Rendszerint, ahogyan a beszúró rendezésnél is, egy algoritmus futási ideje egy adott bemenetre rögzített, bár későbbi fejezetekben látni fogunk néhány érdekes „véletlenített" algoritmust, amelyek viselkedése még rögzített bemenetnél is változhat.

A legrosszabb eset és az átlagos eset elemzése

A beszúró rendezés elemzésében megnéztük mind a legjobb esetet, amelyben a bemenő tömb már eleve rendezve volt, mind a legrosszabb esetet, amelyben a bemenő tömb fordított sorrendben volt rendezve. A könyv hátralévő részében azonban általában a ***legrosszabb futási idő*** keresésére összpontosítunk, azaz a leghosszabb futási időre tetszőleges n méretű bemenetre. Erre három indokot adunk.

- Az algoritmus legrosszabb futási ideje bármilyen bemenetre a futási idő felső korlátja. Garanciát jelent arra, hogy az algoritmus soha nem fog tovább tartani. Nem szükséges találgatásokba bocsátkoznunk a futási időről, azután reménykedni, hogy soha nem lesz ennél rosszabb.
- Bizonyos algoritmusokra gyakran fordul elő a legrosszabb eset. Például, ha egy bizonyos információt keresünk egy adatbázisban, a kereső-algoritmus legrosszabb esete gyakran előfordul, ha az információ nincs az adatbázisban. Néhány kereső alkalmazásban hiányzó információ keresése gyakori lehet.
- Az „átlagos eset" gyakran nagyjából ugyanolyan rossz, mint a legrosszabb eset. Tegyük fel, hogy véletlenszerűen kiválasztunk n számot, és beszúró rendezést alkalmazunk. Mennyi ideig tart annak meghatározása, hogy az $A[1\,..\,j-1]$ résztömbben hova kerüljön az $A[j]$ elem? Átlagosan $A[1\,..\,j-1]$ elemeinek fele kisebb, mint $A[j]$, és fele nagyobb. Ezért átlagosan az $A[1\,..\,j-1]$ résztömb felét nézzük át, így t_j körülbelül $j/2$. Az ebből adódó átlagos futási idő a bemenet méretének négyzetes függvénye.

Bizonyos esetekben az algoritmus ***átlagos*** vagy ***várható*** futási idejére van szükségünk; az 5. fejezetben megismerkedünk a ***valószínűségi elemzés*** módszerével, amellyel meghatározható a várható futási idő. Az átlagos eset elemzésének vizsgálata során probléma lehet, hogy nem feltétlenül nyilvánvaló, mi az „átlagos" bemenet. Gyakran feltételezzük, hogy minden adott méretű bemenet ugyanolyan valószínű, de ez a gyakorlatban nem mindig jogos. A valószínűségi elemzésre olyankor is szükség van, ha ***véletlenített algoritmust*** alkalmazunk.

Növekedési rend

Használtunk néhány egyszerűsítő általánosítást, hogy megkönnyítsük a BESZÚRÓ-RENDEZÉS eljárásának elemzését. Először is figyelmen kívül hagytuk az utasítások tényleges költségét, és jelölésükre a c_i állandókat használtuk. Ezután észrevettük, hogy még ezek az állandók is a szükségesnél több részletet adnak: a legrosszabb futási idő $an^2 + bn + c$ bizonyos a, b és c állandókra, amelyek az utasítások c_i költségeitől függnek. Így figyelmen kívül hagytuk nemcsak a tényleges utasítási költségeket, hanem az absztrakt c_i költségeket is.

Bevezettünk egy további egyszerűsítést. Bennünket igazán a futási idő ***növekedési sebessége*** vagy ***növekedési rendje*** érdekel, ezért a képletnek csak a fő tagját vesszük figyelembe (pl. an^2), mivel az alacsonyabb rendű tagok nagy n-re kevésbé lényegesek. Szintén figyelmen kívül hagyjuk a fő tag állandó együtthatóját, mivel a nagy bemenetekre jellemző

számítási hatékonyság meghatározásában az állandó tényezők kevésbé fontosak, mint a növekedés sebessége. Így azt írjuk, hogy a beszúró rendezés legrosszabb futási ideje $\Theta(n^2)$ („théta n négyzet”-nek ejtjük). A Θ-jelölést kötetlenül használjuk ebben a fejezetben; a 3. fejezetben fogjuk pontosan definiálni.

Általában akkor tartunk egy algoritmust hatékonyabbnak egy másiknál, ha legrosszabb futási idejének alacsonyabb a növekedési rendje. Az állandó tényezők és alacsonyabb rendű tagok miatt ez az értékelés hibás lehet kis bemenetre. Elég nagy bemenetekre azonban például egy $\Theta(n^2)$ algoritmus gyorsabban fut a legrosszabb esetben, mint egy $\Theta(n^3)$ algoritmus.

Gyakorlatok

2.2-1. Fejezzük ki a $(n^3/1000 - 100n^2 - 100n + 3)$ függvényt a Θ-jelölés segítségével.

2.2-2. Vizsgáljuk meg az A tömbben tárolt n szám rendezését, ha először A legkisebb elemét keressük meg, és azt kicseréljük az $A[1]$ elemmel. Azután megkeressük A második legkisebb elemét, és azt kicseréljük $A[2]$-vel. Folytassuk így A első $n - 1$ elemére. Írjunk pszeudokódot erre az algoritmusra, amely ***kiválasztásos rendezésként*** ismert. A Θ-jelölés alkalmazásával adjuk meg a kiválasztásos rendezés legjobb és legrosszabb futási idejét.

2.2-3. Vizsgáljuk meg a lineáris keresést újra (lásd 2.1-3. gyakorlat). Átlagosan a bemenő sorozat hány elemét kell ellenőrizni, feltételezve, hogy a keresett elem ugyanolyan valószínűséggel lehet a tömb bármely eleme? Mi a helyzet a legrosszabb esetben? A Θ-jelölés alkalmazásával adjuk meg a lineáris keresés futási idejét átlagos és legrosszabb esetben. Igazoljuk a választ.

2.2-4. Hogyan módosíthatjuk szinte bármelyik algoritmust, hogy futási ideje a legjobb esetben jó legyen?

2.3. Algoritmusok tervezése

Sokféleképpen lehet algoritmust tervezni. A beszúró rendezés ***növekményes*** megközelítést használ: miután rendeztük az $A[1 \mathinner{..} j-1]$ résztömböt, beszúrjuk az egyetlen $A[j]$ elemet a megfelelő helyre, így az $A[1 \mathinner{..} j]$ rendezett résztömböt kapjuk.

Ebben az alfejezetben egy másik lehetséges tervezési megközelítést vizsgálunk meg, amely oszd-meg-és-uralkodj elvként ismert. Az oszd-meg-és-uralkodj módszerrel tervezünk egy rendezési algoritmust, amelynek futási ideje a legrosszabb esetben sokkal kisebb, mint a beszúró rendezésé. Az oszd-meg-és-uralkodj típusú algoritmusok egyik előnye az, hogy futási idejük gyakran könnyen meghatározható olyan módszerek használatával, amelyeket a 4. fejezetben mutatunk be.

2.3.1. Az oszd-meg-és-uralkodj megközelítés

Sok hasznos algoritmus szerkezetében ***rekurzív:*** egy adott feladat megoldásához rekurzívan hívják magukat egyszer vagy többször, egymáshoz szorosan kötődő részfeladatok kezelésére. Ezek az algoritmusok általában az ***oszd-meg-és-uralkodj*** megközelítést követik: a feladatot több részfeladatra osztják, amelyek hasonlóak az eredeti feladathoz, de méretük kisebb, rekurzív módon megoldják a részfeladatokat, majd összevonják ezeket a megoldásokat, hogy az eredeti feladatra megoldást adjanak.

Az oszd-meg-és-uralkodj paradigma a rekurzió minden szintjén három lépésből áll.

Felosztja a feladatot több részfeladatra.

Uralkodik a részfeladatokon rekurzív módon való megoldásukkal. Ha a részfeladatok mérete elég kicsi, akkor közvetlenül megoldja a részfeladatokat.

Összevonja a részfeladatok megoldásait az eredeti feladat megoldásává.

Az ***összefésülő rendezés*** algoritmus szorosan követi az oszd-meg-és-uralkodj paradigmát. Lényegét tekintve a következőképpen működik.

Felosztás: Az n elemű rendezendő sorozatot felosztja két $n/2$ elemű részsorozatra.

Uralkodás: A két részsorozatot összefésülő rendezéssel rekurzív módon rendezi.

Összevonás: Összefésüli a két részsorozatot, létrehozva a rendezett választ.

A rekurzió akkor „áll le”, amikor a rendezendő sorozat hossza 1: ekkor nincs mit csinálnia, mivel minden 1 hosszúságú sorozat már eleve sorba van rendezve.

Az összefésülő rendezés kulcsművelete a két rendezett sorozat összefésülése az „összevonás” lépésben. Az összefésülés végrehajtásához az ÖSSZEFÉSÜL(A, p, q, r) kisegítő eljárást használjuk, ahol A egy tömb és p, q és r a tömb elemeinek indexei úgy, hogy $p \leq q < r$. Az eljárás feltételezi, hogy az $A[p\,..\,q]$ és $A[q+1\,..\,r]$ résztömbök rendezettek. ***Összefésüli*** azokat egyetlen rendezett tömbbé, amely helyettesíti az addigi $A[p\,..\,r]$ résztömböt.

ÖSSZEFÉSÜL eljárásunk $\Theta(n)$ idejű, ahol $n = r - p + 1$ az összefésülendő elemek száma, és a következőképpen működik. Visszatérve a kártyás hasonlatra, tegyük fel, hogy két halom kártyánk van az asztalon fejjel felfelé. Mindkét halom rendezett, a legkisebb lappal a tetején. A két halmot szeretnénk egyetlen rendezett kimeneti halomba fésülni úgy, hogy a halom lapjai fejjel lefelé legyenek az asztalon. Az alaplépés az, hogy kiválasztjuk a két halom tetejéről a kisebb lapot, eltávolítjuk a halomból (amelyben így új felső lap lesz látható), és fejjel lefelé ráhelyezzük a kimeneti halomra. Ezt a lépést addig ismételjük, amíg az egyik bemenő halom üres nem lesz; ekkor a nemüres bemeneti halmot egyszerűen rátesszük fejjel lefelé a kimeneti halomra. A számításigényt tekintve minden alaplépés állandó ideig tart, mivel mindig csak a két felső lapot ellenőrizzük. Mivel legfeljebb n alaplépést hajtunk végre, az összefésülés $\Theta(n)$ ideig tart.

A következő pszeudokód a fenti gondolatot valósítja meg, azzal a kiegészítéssel, hogy nem kell minden alaplépésben megvizsgálni, vajon üres-e valamelyik halom. Ez azon az ötleten alapul, hogy mindegyik halom aljára teszünk egy ***őrszem*** lapot, amely különleges értéket tartalmaz, és ezzel egyszerűsítjük a kódot. Itt a ∞ értéket alkalmazzuk őrszemként, és ha ∞ értékű lapot látunk, az csak akkor lehet a kisebbik lap, ha már mindkét őrszem lapot látjuk. Ez azonban csak úgy fordulhat elő, ha a nem őrszem lapokat már mind a kimeneti halomba tettük. Mivel előre tudjuk, hogy pontosan $r-p+1$ lapot fogunk a kimeneti halomba tenni, megállhatunk, amikor ennyi alapműveletet elvégeztünk.

Az ÖSSZEFÉSÜL eljárás részletesen a következőképp működik. Az 1. sorban kiszámítjuk az $A[p\,..\,q]$ résztömb n_1 hosszát, a 2. sorban pedig az $A[q+1\,..\,r]$ tömb n_2 hosszát. A 3. sorban létrehozzuk az $n_1 + 1$ hosszúságú L („bal”) és az $n_2 + 1$ hosszúságú R („jobb”) résztömböket. A 4–5. sorokban lévő **for** ciklus az $A[p\,..\,q]$ résztömböt $L[1\,..\,n_1]$-be, a 6–7. sorokban lévő **for** ciklus pedig az $A[q+1\,..\,r]$ résztömböt $R[1\,..\,n_2]$-be másolja. A 8–9. sorokban az L, illetve R tömbök végére másoljuk az őrszemeket. A 10–17. sorokban – amint azt a 2.3. ábra mutatja – végrehajtjuk az $r - p + 1$ alaplépést, miközben fenntartjuk a ciklusinvariánst.

ÖSSZEFÉSÜL(A, p, q, r)

```
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  az L[1 . . n1 + 1] és R[1 . . n2 + 1] tömbök létrehozása
 4  for i ← 1 to n1
 5      do  L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7      do  R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13      do if L[i] ≤ R[j]
14            then  A[k] ← L[i]
15                  i ← i + 1
16            else  A[k] ← R[j]
17                  j ← j + 1
```

A ciklusinvariáns a következő:

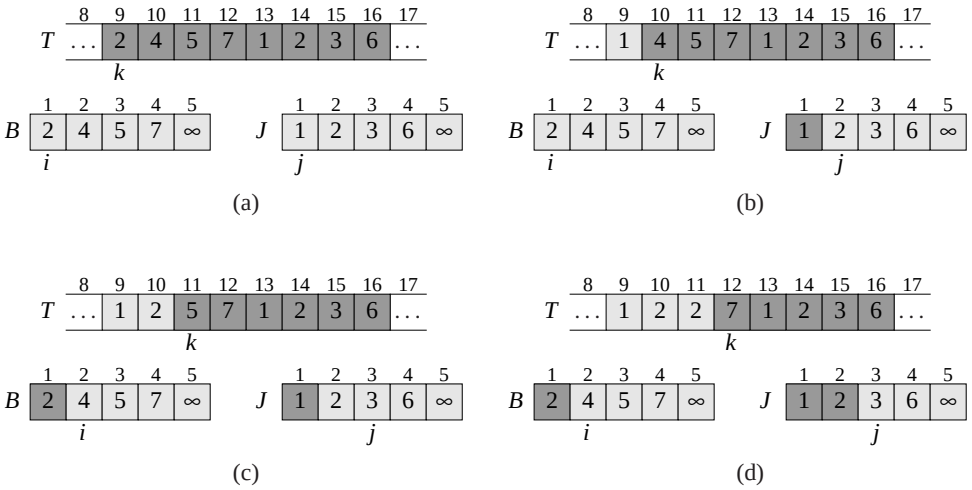
A 12–17. sorokbeli **for** ciklus minden iterációjának kezdetén az $A[p\,..\,k-1]$ résztömb az $L[1\,..\,n_1+1]$ és $R[1\,..\,n_2+1]$ tömbök $k-p$ darab legkisebb elemét tartalmazzák, rendezetten. Továbbá $L[i]$ és $R[j]$ a megfelelő tömbök legkisebb olyan elemei, amelyek még nincsenek visszamásolva A-ba.

Meg kell mutatnunk, hogy ez a ciklusinvariáns teljesül a 12–17. sorokban szereplő **for** ciklus első iterációja előtt, hogy a ciklus minden iterációja fenntartja az invariánst, és hogy amikor a ciklus véget ér, az invariáns hasznos tulajdonságot ír le a helyesség belátásához.

Teljesül: A ciklus első iterációja előtt $k = p$, így az $A[p\,..\,k-1]$ résztömb üres. Ez az üres résztömb tartalmazza az L és R résztömbök $k - p = 0$ darab legkisebb elemét, és mivel $i = j = 1$, $L[i]$ és $R[j]$ saját tömbjük legkisebb olyan elemei, amelyek még nincsenek visszamásolva A-ba.

Megmarad: Annak belátásához, hogy minden iteráció fenntartja a ciklusinvariánst, el őször tegyük fel, hogy $L[i] \le R[j]$. Így $L[i]$ a legkisebb olyan elem, amelyet még nem másoltunk vissza A-ba. Mivel $A[p\,..\,k-1]$ a $k-p$ legkisebb elemet tartalmazza, miután a 14. sorban $L[i]$-t $A[k]$-ba másoltuk, ezért az $A[p\,..\,k]$ résztömb a $k-p+1$ legkisebb elemet fogja tartalmazni. k növelése a **for** ciklus fejében és i növelése a 15. sorban helyreállítja a ciklusinvariánst a következő iterációhoz. Ha viszont $L[i] > R[j]$, akkor a 16–17. sorokban állítjuk helyre a ciklusinvariánst.

Befejeződik: Befejezéskor $k = r + 1$. A ciklusinvariáns szerint az $A[p\,..\,k-1]$ résztömb, amely egyúttal $A[p\,..\,r]$ résztömb is, az $L[1\,..\,n_1+1]$ és $R[1\,..\,n_2+1]$ résztömbök $k-p = r-p+1$ darab legkisebb elemét tartalmazza, rendezetten. L és R együtt összesen $n_1 + n_2 + 2 = r - p + 3$ elemet tartalmaznak. A két legnagyobb kivételével az elemek már visszamásolódtak A-ba, és ez a két kivételes elem a két őrszem.



2.3. ábra. A 10–17. sorok működése az ÖSSZEFÉSÜL$(A, 9, 12, 16)$ hívásakor, amikor az $A[9\,..\,16]$ résztömb a $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$ sorozatot tartalmazza. Az őrszemek másolása és beszúrása után az L tömb a $\langle 2, 4, 5, 7, \infty\rangle$ elemeket, az R tömb pedig az $\langle 1, 2, 3, 6, \infty\rangle$ elemeket tartalmazza. Az A tömb világosszürke pozíciói a végső értékeket, L és R világosszürke pozíciói pedig az A-ba még vissza nem másolt értékeket tartalmazzák. A világosszürke pozíciók együtt az eredetileg $A[9\,..\,16]$-ban tárolt értékeket tartalmazzák, a kétőrszemmel együtt. A sötétszürke pozíciói a felülírandó, L és R sötétszürke pozíciói a már A-ba visszamásolt értékeket tartalmazzák. **(a)–(h)** Az A, L és R tömbök, valamint a megfelelő k, i és j indexek a 12–17. sorokban szereplő ciklus egyes iterációi előtt. **(i)** A tömbök és az indexek befejezéskor. Ekkor az $A[9\,..\,16]$ résztömb rendezett, és L, valamint R elemei közül csak a két őrszem az, amelyet nem másoltunk át A-ba.

Annak belátásához, hogy ÖSSZEFÉSÜL $\Theta(n)$ ideig fut, ahol $n = r - p + 1$, figyeljük meg, hogy az 1–3. és a 8–11. sorok állandó időt vesznek igénybe, a 4–7. sorokban szereplő **for** ciklusok $\Theta(n_1+n_2) = \Theta(n)$ időt,[6] a 12–17. sorokban szereplő **for** ciklusnak pedig n iterációja van, és mindegyik állandó ideig tart.

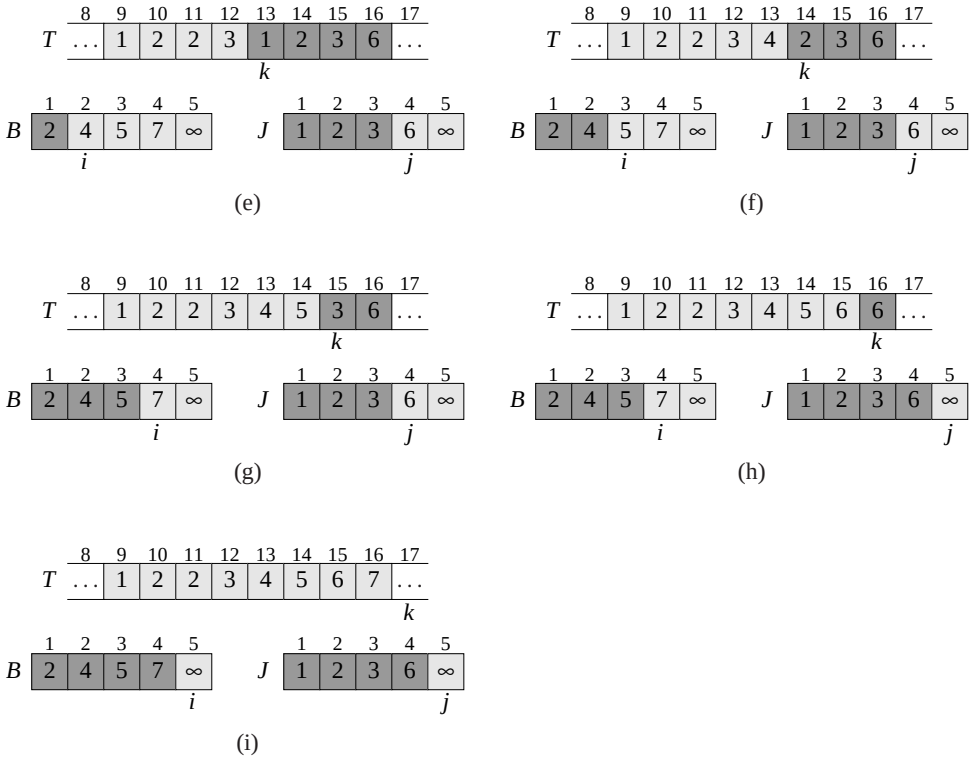
Most már használhatjuk szubrutinként az ÖSSZEFÉSÜL eljárást az ÖSSZEFÉSÜLŐ-RENDEZÉSben. Az ÖSSZEFÉSÜLŐ-RENDEZÉS(A, p, r) rendezi az $A[p\,..\,r]$ résztömb elemeit. Ha $p \geq r$, a résztömb legfeljebb egy elemet tartalmaz, tehát már eleve rendezett. Egyébként a felosztás lépés egyszerűen meghatároz egy q indexet, amely $A[p\,..\,r]$-t két résztömbre osztja: az $\lceil n/2\rceil$ elemet tartalmazó $A[p\,..\,q]$-ra, és az $\lfloor n/2\rfloor$ elemet tartalmazó $A[q+1\,..\,r]$-re.[7]

ÖSSZEFÉSÜLŐ-RENDEZÉS(A, p, r)

```
1  if  p < r
2     then q ← ⌊(p + r)/2⌋
3          ÖSSZEFÉSÜLŐ-RENDEZÉS(A, p, q)
4          ÖSSZEFÉSÜLŐ-RENDEZÉS(A, q + 1, r)
5          ÖSSZEFÉSÜL(A, p, q, r)
```

[6] A 3. fejezetben látni fogjuk, hogyan értelmezzük formálisan a Θ-jelölést tartalmazó egyenlőségeket.

[7] Az $\lceil x\rceil$ kifejezés azt a legkisebb egész számot jelenti, amely nagyobb vagy egyenlő x-szel, és $\lfloor x\rfloor$ jelöli azt a legnagyobb egész számot, amely kisebb vagy egyenlő x-szel. Ezeket a jelöléseket a 3. fejezetben definiáljuk. A legegyszerűbben úgy mutathatjuk meg, hogy q-nak $\lfloor(p + r/2)\rfloor$-vel való helyettesítése az $\lceil n/2\rceil$ és $\lfloor n/2\rfloor$ méretű $A[p\,..\,q]$ és $A[q+1\,..\,r]$ méretű résztömböket eredményezi, hogy megvizsgáljuk azt a négy esetet, amelyek attól függően fordulnak elő, hogy p, illetve q páratlan vagy páros.

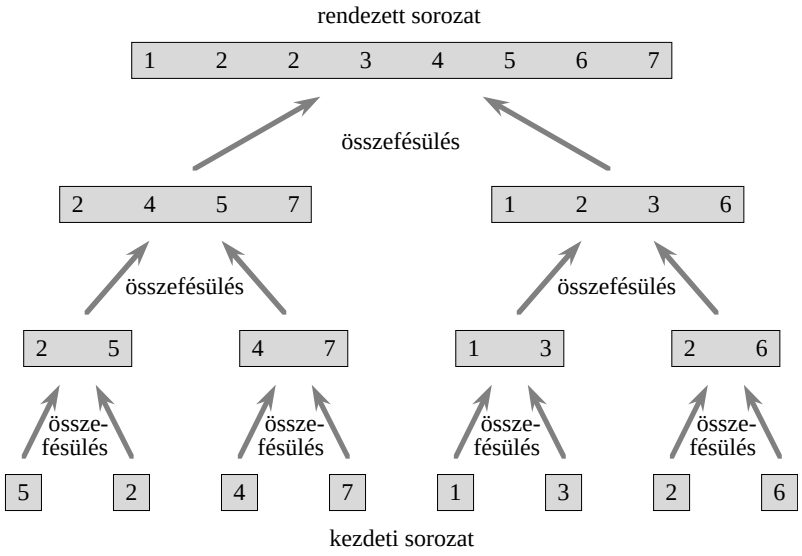


Az egész $A = \langle A[1], A[2], \ldots, A[n] \rangle$ sorozat rendezéséhez Összefésülő-rendezés$(A, 1, hossz[A])$-t hívjuk, ahol ismét $hossz[A] = n$. A 2.4. ábra alulról felfelé mutatja az eljárás működését abban az esetben, amikor n kettőhatvány. Az algoritmus egyelemű sorozatpárok kételemű rendezett sorozattá fésüléséből, kételemű sorozatok négyelemű rendezett sorozatokká fésüléséből stb. áll, végül pedig két $n/2$ hosszúságú sorozatot fésül össze, hogy létrehozza a végső, rendezett n hosszúságú sorozatot.

2.3.2. Oszd-meg-és-uralkodj algoritmusok elemzése

Amikor egy algoritmus rekurzívan hívja magát, futási idejét gyakran ***rekurzív egyenlőséggel*** vagy ***rekurzióval*** írhatjuk le, amely az n méretű feladat megoldásához szükséges teljes futási időt a kisebb bemenetekhez tartozó futási időkkel fejezi ki. A rekurzió megoldására matematikai eszközöket használunk, melyekkel az algoritmusra teljesítménykorlátokat tudunk adni.

Az oszd-meg-és-uralkodj algoritmusok futási idejére a rekurzió az alapparadigma három lépésén alapszik. Ahogy korábban, legyen $T(n)$ a futási idő, ha a feladat mérete n. Ha a feladat mérete elég kicsi, mondjuk $n \leq c$ egy bizonyos c állandóra, a közvetlen megoldás állandó ideig tart, amit $\Theta(1)$-nek írunk. Tegyük fel, hogy a feladatot a darab részfeladatra osztjuk, melyek mindegyikének mérete az eredeti méretének $(1/b)$-szerese. Ha $D(n)$ ideig tart a feladat részfeladatokra osztása, és $C(n)$ ideig a részfeladatok megoldásainak összevonása az eredeti feladat megoldásává, akkor a következő rekurzív egyenlőséget kapjuk:



2.4. ábra. Az összefésülő rendezés működése az $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ tömbön. Az összefésülendő rendezett sorozatok hossza növekszik, ahogy az algoritmus halad lentről felfelé.

$$T(n) = \begin{cases} \Theta(1), & \text{ha } n \leq c, \\ aT(n/b) + D(n) + C(n), & \text{egyébként.} \end{cases}$$

A 4. fejezetben látni fogjuk, hogyan lehet ilyen jellegű általános rekurziókat megoldani.

Az összefésülő rendezés elemzése

Noha az Összefésülő-rendezés pszeudokódja helyesen dolgozik, amikor az elemek száma nem páros, rekurzió alapú elemzésünk egyszerűbb, ha feltesszük, hogy az eredeti feladat mérete kettőhatvány. Így minden felosztó lépés két, pontosan $n/2$ méretű részsorozatot ad. A 4. fejezetben látni fogjuk, hogy ez a feltételezés nincs hatással a rekurzió megoldásának növekedési rendjére.

A következőképp érvelünk, hogy $T(n)$-re, az összefésülő rendezés – n szám rendezéséhez szükséges – legrosszabb futási idejére felírjuk a rekurziót. A összefésülő rendezés egy elemre állandó ideig tart. Amikor $n > 1$ elemünk van, a következőképpen bontjuk fel a futási időt.

Felosztás: A felosztási lépés csak a résztömb közepét számolja ki, ami állandó ideig tart. Így $D(n) = \Theta(1)$.

Uralkodás: Rekurzív módon megoldjuk a két részfeladatot, melyeknek mérete $n/2$, ami $2T(n/2)$-vel járul hozzá a futási időhöz.

Összevonás: Már megjegyeztük, hogy az Összefésül eljárás egy n-elemű résztömbön $\Theta(n)$ ideig tart, így $C(n) = \Theta(n)$.

Amikor összeadjuk a $D(n)$ és $C(n)$ függvényeket az összefésülő rendezés elemzésekor, egy $\Theta(n)$ és egy $\Theta(1)$ függvényt összegzünk. Ez az összeg n lineáris függvénye, azaz $\Theta(n)$. Ezt

hozzáadva az „uralkodás” részből származó $2T(n/2)$ kifejezéshez, az összefésülő rendezés legrosszabb futási idejére a

$$T(n) = \begin{cases} \Theta(1), & \text{ha } n = 1, \\ 2T(n/2) + \Theta(n), & \text{ha } n > 1 \end{cases} \tag{2.1}$$

rekurziót kapjuk. A 4. fejezetben tárgyaljuk a „mester tételt”, melynek felhasználásával megmutatjuk, hogy $T(n) = \Theta(n \lg n)$, ahol lg a kettes alapú logaritmusfüggvényt jelenti. Mivel a logaritmusfüggvény minden lineáris függvénynél lassabban nő, az összefésülő rendezés, $\Theta(n \lg n)$ futási idejével, elég nagy n-re jobb a beszúró rendezésnél, amelynek legrosszabb esetben a futási ideje $\Theta(n^2)$.

Nincs szükségünk a mester tételre ahhoz, hogy intuitívan belássuk, miért megoldása a (2.1) rekurziónak a $T(n) = \Theta(n \lg n)$ függvény. Írjuk át a (2.1) rekurziót a következő alakra:

$$T(n) = \begin{cases} c, & \text{ha } n = 1, \\ 2T(n/2) + cn, & \text{ha } n > 1, \end{cases} \tag{2.2}$$

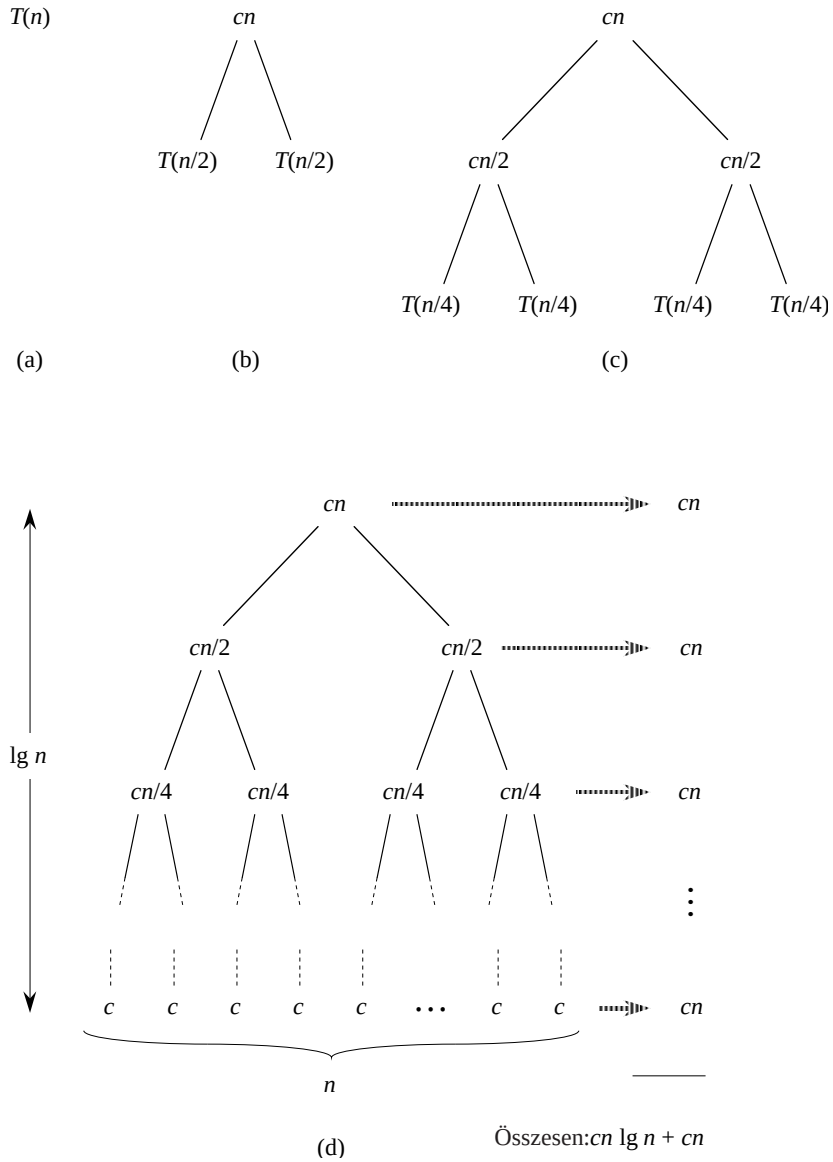
ahol a c állandó azt az időt jelenti, amelyre az 1 méretű feladat megoldásához szükség van, valamint azt az időt, amelyre a felosztás és összevonás műveletekhez tömbelemenként szükségünk van.[8]

A 2.5. ábra mutatja a (2.2) rekurzió megoldását. Az egyszerűség kedvéért feltesszük, hogy n kettőhatványa. Az ábra (a) része $T(n)$-t mutatja, amelyet a (b) részben a rekurziót ábrázoló ekvivalens fává terjesztünk ki. A cn tag a gyökér (a költség a rekurzió legfelső szintjén), a gyökérhez tartozó két részfa pedig a két kisebb $T(n/2)$ tagot ábrázolja. A (c) részben ezt az eljárást egy lépéssel tovább folytatjuk, kiterjesztve $T(n/2)$-t. A rekurzió második szintjén lévő mindkét csúcsra a költség $cn/2$. A fa csúcsainak kiterjesztését azzal folytatjuk, hogy minden csúcsot – a rekurzió által meghatározott módon – részeire bontunk mindaddig, amíg a feladat mérete le nem csökken 1-re, amikor is minden feladat megoldási költsége c lesz. Az ábra (d) része mutatja az eredményül kapott fát.

Ezután összeadjuk a fa egyes szintjeihez tartozó költségeket. A legfelső szinten a költség összesen cn, az alatta lévő szinten $c(n/2) + c(n/2) = cn$, az ezalatt lévő szinten a költség $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ és így tovább. Általában a felső szinttől számított i-edik szinten a csúcsok száma 2^i, és mindegyik $c(n/2^i)$-vel járul hozzá a költséghez, így az i-edik szint összes költsége $2^i c(n/2^i) = cn$. A legalsó szinten n csúcs van, és mindegyik c-vel járul az összköltséghez, ezért a legalsó szint hozzájárulása is cn.

A 2.5. „rekurziós fa” szintjeinek száma $\lg n + 1$. Ez informális indukcióval könnyen belátható. Az alapeset akkor fordul elő, amikor $n = 1$ – ekkor csak egy szint van. Mivel $\lg 1 = 0$, azt kapjuk, hogy $\lg n + 1$ megadja a szintek számát. Most indukciós feltételként tegyük fel, hogy a 2^i csúcsú rekurziós fa szintjeinek száma $\lg 2^i + 1 = i + 1$ (mivel bármely i-re fennáll $\lg 2^i = i$.) Mivel feltettük, hogy az eredeti bemenet mérete kettőhatvány, a bemenet méretének következő vizsgálandó értéke 2^{i+1}. A 2^{i+1} csúcsú fának eggyel több szintje van, mint a 2^i csúcsú fának, ezért szintjeinek száma összesen $(i + 1) + 1 = \lg 2^{i+1} + 1$.

[8] Nem valószínű, hogy pontosan ugyanannyi időre van szükség az 1 méretű feladat megoldásához és tömbelemenként a felosztás és összevonás lépésekhez. Elkerülhetjük ezt a problémát, ha ezen állandó idők közül a nagyobbikat választjuk c-nek és elfogadjuk, hogy ekkor a rekurziónk felső korlátot ad a futási időre, vagy a kisebbik állandót választjuk c-nek és elfogadjuk, hogy ebben az esetben a rekurzió alsó korlátot ad a futási időre. Mindkét korlát $n \lg n$ nagyságrendű lesz, így együtt $\Theta(n \lg n)$ futási időt eredményeznek.



2.5. ábra. Rekurziós fa építése a $T(n) = 2T(n/2) + cn$ rekurzióhoz. Az **(a)** rész $T(n)$-t mutatja, amelyet fokozatosan terjesztünk ki a **(b)–(d)** részekben rekurziós fává. A teljesen kiterjesztett fának a (d) részben $\lg n + 1$ szintje van (azaz a magassága $\lg n$, amint azt az ábra mutatja), és minden szint cn-nel járul a teljes költséghez. Ezért az összes költség $cn \lg n + cn$, azaz $\Theta(n \lg n)$.

A (2.2) rekurzióhoz tartozó költség kiszámításához egyszerűen összeadjuk az egyes szintekhez tartozó költségeket. $\lg n + 1$ szint van, ezért az összes költség $cn(\lg n + 1) = cn \lg n + cn$. Az alacsonyabb rendű tagokat és a c állandót elhanyagolva a kívánt $\Theta(n \lg n)$ eredményt kapjuk.

Gyakorlatok

2.3-1. A 2.4. ábrát alapul véve mutassuk be az összefésülő rendezés működését az $A = \langle 3, 41, 52, 26, 38, 57, 9, 49\rangle$ tömbre.

2.3-2. Írjunk pszeudokódot az Összefésül(A, p, q, r) eljárásra.

2.3-3. Használjunk teljes indukciót annak megmutatására, hogy amikor n kettőhatvány, a

$$T(n) = \begin{cases} 2, & \text{ha } n = 2, \\ 2T(n/2) + n, & \text{ha } n = 2^k,\ k > 1 \end{cases}$$

rekurzió megoldása $T(n) = n \lg n$.

2.3-4. A beszúró rendezést a következőképpen adhatjuk meg rekurzív eljárásként. Az $A[1\,..\,n]$ tömb rekurzív módon való rendezéséhez rendezzük $A[1\,..\,n-1]$-et, és beszúrjuk $A[n]$-t az $A[1\,..\,n-1]$ rendezett tömbbe. Írjuk fel a rekurziót a beszúró rendezés ezen rekurzív változatának futási idejére.

2.3-5. Visszautalva a keresési feladatra (lásd 2.1-3. gyakorlat), vizsgáljuk meg, hogy ha az A sorozat rendezve van, akkor összevethetjük a sorozat középső elemét v értékével, és kivonhatjuk a sorozat felét a további vizsgálódás alól. A ***bináris keresés*** egy olyan algoritmus, amely ezt az eljárást ismétli, minden alkalommal megfelezve a sorozat megmaradó részét. Bizonyítsuk, hogy a bináris keresés futási ideje a legrosszabb esetben $\Theta(\lg n)$.

2.3-6. Vegyük észre, hogy a 2.1. alfejezetben a Beszúró-rendezés eljárás **while** ciklusa az 5–7. sorokban egy lineáris keresést használ az $A[1\,..\,j-1]$ rendezett résztömb pásztázására (visszafelé). Használhatunk-e ehelyett bináris keresést (lásd 2.3-5. gyakorlat), hogy a beszúró rendezés legrosszabb futási idejét $\Theta(n \lg n)$-ra javítsuk?

2.3-7.⋆ Írjunk le egy olyan $\Theta(n \lg n)$ futási idejű algoritmust, amely egy n egész számból álló S halmazra és egy x egész számra meghatározza, hogy van-e S-nek két olyan eleme, amelyek összege pontosan x.

Feladatok

2-1. Beszúró rendezés az összefésülő rendezés kis tömbjeire

Bár az összefésülő rendezés a legrosszabb esetben $\Theta(n \lg n)$ idő alatt fut le, és a beszúró rendezés $\Theta(n^2)$ idő alatt, a beszúró rendezést az állandó tényezők kis n-re gyorsabbá teszik. Így van értelme a beszúró rendezést használni az összefésülő rendezésen belül, amikor a részfeladatok elég kicsik. Vizsgáljuk meg az összefésülő rendezésnek egy olyan módosítását, amelyben n/k darab k hosszúságú részlistát beszúró rendezéssel rendezünk, és azután a szabályos összefésülési eljárással összefésülünk, ahol k adott érték.

a. Mutassuk meg, hogy n/k darab k hosszúságú részlistát a beszúró rendezéssel a legrosszabb esetben is lehet $\Theta(nk)$ idő alatt rendezni.

b. Mutassuk meg, hogy a részlistákat legrosszabb esetben $\Theta\,(n \lg(n/k))$ idő alatt lehet összefésülni.

c. Ha a módosított algoritmus legrosszabb esetben $\Theta(nk + n \lg(n/k))$ idő alatt fut, melyik az a legnagyobb aszimptotikus (Θ-jelölés) k érték n függvényeként, amelyre a módosított algoritmusnak ugyanaz az aszimptotikus futási ideje, mint a szabályos összefésülő rendezésnek?

d. Hogyan válasszuk ki k-t a gyakorlatban?

2-2. A buborékrendezés helyessége

A buborékrendezés népszerű rendező algoritmus. Úgy dolgozik, hogy egymás után felcseréli azokat a szomszédos elemeket, amelyek rossz sorrendben vannak.

BUBORÉK-RENDEZÉS(A)

```
1  for i ← 1 to hossz[A]
2  do for j ← hossz[A] downto i + 1
3        do if A[j] < A[j − 1]
4              then hajtsuk végre az A[j] ↔ A[j − 1] cserét
```

a. Jelöljük A'-vel a BUBORÉK-RENDEZÉS kimenetét. A BUBORÉK-RENDEZÉS helyességének bebizonyításához meg kell mutatnunk, hogy befejeződik, és hogy

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n], \tag{2.3}$$

ahol $n = hossz[A]$. Mit kell még belátnunk ahhoz, hogy bebizonyítsuk, a BUBORÉK-RENDEZÉS valóban rendez?

A következő két rész bizonyítja a (2.3) egyenlőtlenséget.

b. Fogalmazzuk meg pontosan a ciklusinvariánst a 2–4. sorokban lévő **for** ciklusra, és bizonyítsuk be, hogy ez a ciklusinvariáns teljesül. A bizonyítás használja fel a ciklusinvariáns bizonyításának ebben a fejezetben bemutatott szerkezetét.

c. A ciklusinvariánsnak a (b) részben bizonyított befejezési feltételét felhasználva fogalmazzunk meg az 1–4. sorokban lévő **for** ciklusra egy olyan ciklusinvariánst, amely lehetővé teszi, hogy bebizonyítsuk a (2.3) egyenlőtlenséget. A bizonyítás használja fel a ciklusinvariáns bizonyításának ebben a fejezetben bemutatott szerkezetét.

d. Mennyi a buborékrendezés futási ideje a legrosszabb esetben? Hasonlítsuk ezt össze a beszúró rendezés futási idejével.

2-3. A Horner-séma helyessége

Az alábbi kódrészlet a polinomok helyettesítési értékének – adott $a_0, a_1, \ldots, a_n$ együtthatók és adott x érték esetén való – meghatározására szolgáló

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n)\cdots)) \end{aligned}$$

Horner-sémát valósítja meg.

```
1  y ← 0
2  i ← n
3  while i ≥ 0
4        do y → a_i + x · y
5           i ← i − 1
```

a. Mi a Horner-séma fenti kódrészletének az aszimptotikus futási ideje?

b. Polinomok helyettesítési értéke olyan egyszerű algoritmussal is meghatározható, amely a polinom minden tagjának értékét külön kiszámítja, majd ezeket az értékeket összeadja. Mi ennek az algoritmusnak a futási ideje? Hasonlítsuk össze ezt a Horner-séma futási idejével.

c. Bizonyítsuk be, hogy az alábbi tulajdonság a 3–5. sorokban lévő **while** ciklus egy ciklusinvariánsa.

A 3–5. sorokban lévő **while** ciklus minden iterációjának kezdetén teljesül

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$$

A tag nélküli összeg értéke legyen nulla. A bizonyításban használjuk fel a ciklusinvariáns bizonyításának ebben a fejezetben bemutatott szerkezetét és mutassuk meg, hogy befejezéskor $y = \sum_{k=0}^{n} a_k x^k$.

d. Végezetül lássuk be, hogy az adott kódrészlet helyesen értékeli az $a_0, a_1, \ldots, a_n$ együtthatókkal megadott polinomot.

2-4. Inverziók

Tegyük fel, hogy $A[1..n]$ n különböző számot tartalmazó tömb. Ha $i < j$ és $A[i] > A[j]$, akkor az (i, j) párt A egy ***inverziójának*** nevezzük.

a. Soroljuk fel a $\langle 2, 3, 8, 6, 1 \rangle$ tömb öt inverzióját.

b. Melyik – az $\{1, 2, \ldots, n\}$ tömb elemeiből álló – tömbben van a legtöbb inverzió? Mennyi?

c. Mi a kapcsolat a beszúró rendezés futási ideje és a bemenő tömb inverzióinak száma között? Igazoljuk a választ.

d. Adjunk egy algoritmust, amely legrosszabb esetben $\Theta(n \lg n)$ idő alatt meghatározza n elem inverzióinak számát bármilyen permutációban. (*Útmutatás.* Módosítsuk az összefésülő rendezést.)

Megjegyzések a fejezethez

Knuth 1968-ban publikálta *A számítógép-programozás művészete* [182, 183, 185] című háromkötetes művét, mely a számítógéptudomány alapvető művévé vált. Az első kötet bevonult az számítógép-algoritmusok modern tudományába a futási idő elemzésével, és az egész sorozat is megnyerő és hitelt érdemlő hivatkozás az itt tárgyalt témák egy részére. A könyv bevezetőjéből (is) megtudhatjuk, hogy az algoritmus szó a IX. századi perzsa matematikus, al-Khwârizmî nevéből származik.

Aho, Hopcroft és Ullman [5] az algoritmusok aszimptotikus elemzését úgy tekintették, mint a relatív teljesítmények összehasonlításának eszközét. Népszerűsítették a rekurziók használatát is a rekurzív algoritmusok futási idejének leírására.

Knuth [185] sok rendező algoritmusról ad részletes leírást. Az ő összehasonlításai a rendező algoritmusokról olyan pontos lépésszámelemzéseket tartalmaznak, mint amilyet a beszúró rendezésre bemutattunk. Knuth leírása a beszúró rendezésről az algoritmus számos

változatát felöleli. Ezek közül a legfontosabb a Shell-rendezés, amit D. L. Shell mutatott be, és amely a beszúró rendezést használja a bemenet periodikus részsorozatainak rendezésére, hogy gyorsabb rendezési algoritmust hozzon létre.

Knuth leírja az összefésülő rendezést is. Megemlíti, hogy 1938-ban konstruáltak egy olyan mechanikus összehasonlítót, amely képes két halom lyukkártyát összefésülni. Neumann János, az informatika egyik úttörője 1945-ben írt egy programot az összefésülő rendezésre az EDVAC számítógépen.

A programhelyesség bizonyítása történetének kezdetét Gries [133] írta le, aki P. Naurnak tulajdonítja az első cikket ezen a területen. Gries szerint a ciklusinvariánst R. W. Floyd javasolta. Mitchell tankönyve [222] beszámol a helyességbizonyítás történetének későbbi fejezeteiről.

3. Függvények növekedése

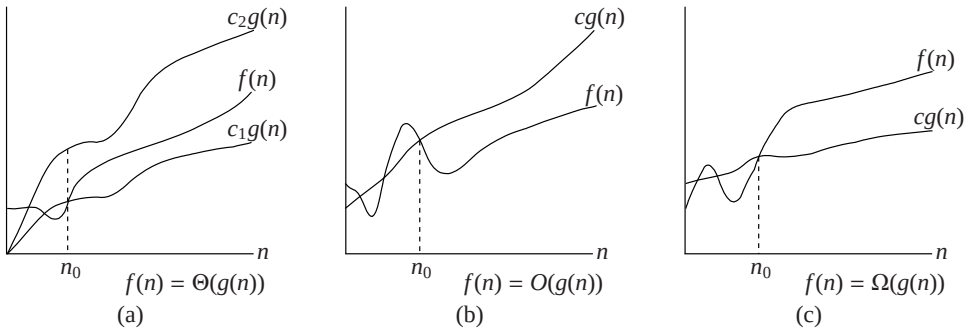
Az algoritmus futási idejének növekedési rendje, melyet a 2. fejezetben definiáltunk, az algoritmus hatékonyságának egyszerű leírását adja, valamint lehetővé teszi a szóba jövő algoritmusok relatív teljesítményének összehasonlítását is. Ha az n bemenő méret elég nagy, akkor az összefésülő rendezés, amelynek a futási ideje a legrosszabb esetben $\Theta(n \lg n)$, jobb, mint a beszúró rendezés, amelynek futási ideje a legrosszabb esetben $\Theta(n^2)$. Bár néha pontosan meg tudjuk határozni az algoritmus futási idejét, ahogy ezt a 2. fejezetben is tettük a beszúró rendezés esetében, általában nem éri meg az erőfeszítést ez a különleges pontosság. Elég nagy bemenő adatokra a futási idő multiplikatív állandóinak és alacsonyabb rendű tagjainak a hatása eltörpül a futási idő nagyságrendjéhez képest.

Ha a bemenet mérete elég nagy, akkor az algoritmus futási idejének csak a nagyságrendje lényeges, ezért az algoritmusnak az ***aszimptotikus*** hatékonyságát vizsgáljuk. Ekkor csak azzal foglalkozunk, hogy a bemenet növekedésével miként növekszik az algoritmus futási ideje *határértékben*, ha a bemenet mérete minden határon túl nő. Általában az aszimptotikusan hatékonyabb algoritmus lesz a legjobb választás, kivéve a kis bemenetek esetét.

Ebben a fejezetben bemutatjuk az algoritmusok aszimptotikus elemzését megkönnyítő általános módszereket. A következő rész néhány *aszimptotikus jelölés* definíciójával kezdődik, amelyek közül eggyel, a Θ-jelöléssel, már találkoztunk. Ezután a könyvben használatos egyezményes jelöléseket mutatjuk be, és végül összefoglaljuk azoknak a függvényeknek a viselkedését, amelyek gyakran előkerülnek az algoritmusok elemzése során.

3.1. Aszimptotikus jelölések

Egy algoritmus aszimptotikus futási idejét leíró jelöléseket olyan függvényekként definiáljuk, melyeknek értelmezési tartománya a természetes számok $\mathbf{N} = \{0, 1, 2, \ldots\}$ halmaza. Ilyen jelölésekkel kényelmesen leírható a futási idő a legrosszabb esetben, azaz a $T(n)$ függvény, amely általában csak egész számú bemenő adattól függ. Mindemellett néha kényelmes az aszimptotikus jelölések többféle *pontatlan* használata. Például a jelölés értelmezési tartománya könnyedén kiterjeszthető a valós számokra, vagy leszűkíthető a természetes számok egy részhalmazára. Mindazonáltal fontos, hogy megértsük a jelölés pontos jelentését, hogy amikor pontatlanul használjuk, akkor ne használjuk tévesen. Ez a rész az alapvető aszimptotikus jelöléseket definiálja, valamint bemutat néhány gyakori pontatlan jelölést is.



3.1. ábra. Grafikus példák a Θ, O és Ω jelölésekre. Mindhárom esetben n_0 a szóba jöhető legkisebb érték, és bármely ennél nagyobb számra is igaz az állítás. (**a**) A Θ-jelöléssel olyan korlátot adunk a függvényre, amelyben a konstans tényezőktől eltekintünk. Akkor mondjuk, hogy $f(n) = \Theta(g(n))$, ha léteznek n_0, c_1 és c_2 pozitív állandók úgy, hogy az n_0-nál nagyobb értékekre az $f(n)$ függvényértékek mindig $c_1g(n)$ és $c_2g(n)$ között vannak (az egyenlőség megengedett). (**b**) Az O-jelöléssel olyan felső korlátot adunk a függvényre, amelyben a konstans tényezőktől eltekintünk. Akkor mondjuk, hogy $f(n) = O(g(n))$, ha léteznek n_0 és c pozitív állandók úgy, hogy minden n_0-nál nagyobb értékre az $f(n)$ függvényérték kisebb vagy egyenlő $cg(n)$-nél. (**c**) Az Ω-jelöléssel olyan alsó korlátot adunk a függvényre, amelyben a konstans tényezőktől eltekintünk. Akkor mondjuk, hogy $f(n) = \Omega(g(n))$, ha léteznek n_0 és c pozitív állandók úgy, hogy minden n_0-tól jobbra lévő $f(n)$ érték nagyobb vagy egyenlő $cg(n)$-nel.

Θ-jelölés

A 2. fejezetben megállapítottuk, hogy a beszúró rendezés futási ideje a legrosszabb esetben $T(n) = \Theta(n^2)$. Most pontosan definiáljuk, hogy mit jelent ez a jelölés. Egy adott $g(n)$ függvény esetén $\Theta(g(n))$-nel jelöljük a függvényeknek azt a halmazát, amelyre

$$\Theta(g(n)) = \{f(n) : \text{ létezik } c_1, c_2 \text{ és } n_0 \text{ pozitív állandó úgy, hogy } 0 \le c_1g(n) \le f(n) \le c_2g(n) \text{ teljesül minden } n \ge n_0 \text{ esetén}\}.$$[1]

Más szóval az $f(n)$ függvény hozzátartozik a $\Theta(g(n))$ halmazhoz, ha léteznek c_1 és c_2 pozitív állandók úgy, hogy $f(n)$ – elég nagy n-re – „beszorítható” $c_1g(n)$ és $c_2g(n)$ közé. Bár $\Theta(g(n))$ egy halmaz, mégis úgy írjuk, hogy „$f(n) = \Theta(g(n))$”, ha azt akarjuk jelezni, hogy $f(n)$ eleme $\Theta(g(n))$-nek, azaz „$f(n) \in \Theta(g(n))$”. Ez a pontatlanság az elem jelölésére először talán megtévesztőnek tűnik, de a későbbiek során látni fogjuk ennek előnyeit is.

A 3.1(a) ábra intuitív képet ad az $f(n)$ és $g(n)$ függvényekről, ahol $f(n) = \Theta(g(n))$.

Minden n_0-tól jobbra lévő n értékre $f(n)$ értéke $c_1g(n)$-nel egyenlő vagy annál nagyobb és $c_2g(n)$-nel egyenlő vagy annál kisebb. Más szóval, minden $n \ge n_0$ esetén az $f(n)$ függvény – egy állandó szorzótényezőtől eltekintve – egyenlő $g(n)$-nel. Ekkor azt mondjuk, hogy $g(n)$ ***aszimptotikusan éles korlátja*** $f(n)$-nek.

$\Theta(g(n))$ definíciója megkívánja, hogy minden $f(n) \in \Theta(g(n))$ ***aszimptotikusan nemnegatív*** legyen, azaz $f(n)$ nemnegatív legyen, ha n elég nagy. (Egy ***aszimptotikusan pozitív*** függvény szigorúan pozitív minden elég nagy n-re.) Következésképpen a $g(n)$ függvénynek aszimptotikusan nemnegatívnak kell lennie, vagy $\Theta(g(n))$ üres. Ezért ezután azt tételezzük fel, hogy minden Θ-jelölésen belül használt függvény aszimptotikusan nemnegatív. Ez a feltételezés érvényes az e fejezetben definiált további aszimptotikus jelölésekre is.

A 2. fejezetben informálisan vezettük be a Θ-jelölést. Lényegében azt mondtuk, hogy el kell dobni az alacsonyabb rendű tagokat, és figyelmen kívül kell hagyni a legmagasabb

[1] Halmazok megadásánál a kettőspont jelentése „melyre teljesül, hogy”.

rendű tag főegyüttható ját. Bizonyítsuk röviden ezt az intuitív elképzelést úgy, hogy a formális definíció felhasználásával megmutatjuk, hogy $n^2/2 - 3n = \Theta(n^2)$. Hogy ezt megtegyük, meg kell határoznunk a c_1, c_2 és n_0 pozitív állandókat, hogy

$$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$$

teljesüljön minden $n \geq n_0$ esetén. Elosztva n^2-tel kapjuk, hogy

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

A jobb oldali egyenlőtlenség igaz minden $n \geq 1$ esetén a $c_2 \geq 1/2$ választásra. Hasonlóan, a bal oldali egyenlőtlenség minden $n \geq 7$ értékre teljesül, ha $c_1 \leq 1/14$. Így $c_1 = 1/14$, $c_2 = 1/2$ és $n_0 = 7$ választással bizonyíthatjuk, hogy $n^2/2 - 3n = \Theta(n^2)$. Természetesen más lehetőségek is vannak az állandók megválasztására, de a lényeg az, hogy léteznek ilyenek. Vegyük észre, hogy ezek az állandók függnek az $n^2/2 - 3n$ függvénytől, és egy – a $\Theta(n^2)$-hez tartozó – másik függvény általában más állandók választását kívánja meg.

Ugyancsak a formális definíció segítségével bizonyíthatjuk, hogy $6n^3 \neq \Theta(n^2)$. Indirekt bizonyítást használva tegyük fel, hogy létezik c_2 és n_0 úgy, hogy $6n^3 \leq c_2 n^2$ minden $n \geq n_0$ esetén. Ekkor azonban $n \leq c_2/6$, ami lehetetlen tetszőleges nagy n esetén, hiszen c_2 állandó.

Szemléletesen, egy aszimptotikusan pozitív függvény alacsonyabb rendű tagjai az aszimptotikusan éles korlát meghatározásánál figyelmen kívül hagyhatók, mert nagy n-re elhanyagolhatók. A legmagasabb rendű tagnak még egy törtrésze is nagyobb lesz az alacsonyabb rendű tagoknál. Így ha c_1 értékét kicsit kisebbre, c_2 értékét pedig kicsit nagyobbra választjuk, mint a főegyüttható, akkor a Θ-jelölés definíciójában lévő egyenlőtlenségeket kielégítettük. Hasonló módon a főegyüttható is figyelmen kívül hagyható, mert ez csak egy konstans szorzóval változtathatja meg c_1 és c_2 értékét.

Tekintsük például az $f(n) = an^2 + bn + c$ függvényt, ahol a, b és c állandók és $a > 0$. Eldobva az alacsonyabb rendű tagokat és a főegyütthatót $f(n) = \Theta(n^2)$ adódik. Hogy ezt formálisan is megmutassuk, legyenek az állandók $c_1 = a/4$, $c_2 = 7a/4$ és $n_0 = 2 \cdot \max((|b|/a), \sqrt{(|c|/a)})$. Az olvasó bizonyíthatja, hogy $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ minden $n \geq n_0$ esetén. Általában, tetszőleges $p(n) = \sum_{i=0}^{d} a_i n^i$ polinomra, ahol minden i-re a_i állandó és $a_d > 0$, fennáll a $p(n) = \Theta(n^d)$ becslés (lásd a 3-1. feladatot).

Mivel bármely állandó egy nulladfokú polinom, ezért bármelyik konstans függvényre fennáll a $\Theta(n^0)$ vagy $\Theta(1)$ becslés. Az utóbbi jelölés kissé pontatlan, hiszen nem nyilvánvaló, hogy ekkor mely változó tart a végtelenhez.[2] Gyakran fogjuk használni a $\Theta(1)$ jelölést akár állandóra, akár egy adott változóra nézve konstans függvényre is.

O-jelölés

A Θ-jelölés aszimptotikus alsó és felső korlátot ad a függvényre. Amikor csak az ***aszimptotikus felső korlát*** jön szóba, akkor használjuk az O-jelölést. Egy adott $g(n)$ függvény esetén $O(g(n))$-nel (olvasd: „ordó $g(n)$” vagy „nagy ordó $g(n)$”) jelöljük a függvényeknek azt a halmazát, amelyre

[2] Az igazi probléma az, hogy a mi általános függvényjelöléseink nem különböztetik meg a függvényt a függvényértékektől. A λ-kalkulusban a függvényhez tartozó paraméterek egyértelműen adottak: az $\hat{n}^2$ függvény úgy írható, mint $\lambda n.n^2$ vagy akár $\lambda r.r^2$. Azonban egy szigorúbb jelölés elfogadása körülményessé tenné az algebrai műveleteket, ezért inkább elviseljük ezt a pontatlanságot.

$$O(g(n)) = \{f(n) : \text{létezik } c \text{ és } n_0 \text{ pozitív állandó úgy, hogy } 0 \le f(n) \le cg(n) \text{ teljesül minden } n \ge n_0 \text{ esetén}\}.$$

Az O-jelölést arra használjuk, hogy egy állandó tényezőtől eltekintve felső korlátot adjunk a függvényre. A 3.1(b) ábra szemléletesen mutatja az O-jelölést, az n_0-tól jobbra lévő minden n értékre az $f(n)$ függvényérték $cg(n)$-nel egyenlő vagy ennél kisebb.

Ha $f(n)$ eleme $O(g(n))$-nek, akkor azt írjuk, hogy $f(n) = O(g(n))$. Figyeljük meg, hogy $f(n) = \Theta(g(n))$ maga után vonja $f(n) = O(g(n))$ teljesülését. A halmazelméletben szokásos jelöléseket használva $\Theta(g(n)) \subseteq O(g(n))$. Abból, hogy bármely másodfokú $an^2 + bn + c$, $a > 0$ függvény benne van $\Theta(n^2)$-ben, következik, hogy az ilyen másodfokú függvények benne vannak $O(n^2)$-ben. Meglepő lehet, hogy bármely *elsőfokú* $an + b$ alakú függvény is benne van $O(n^2)$-ben, ami könnyen bizonyítható a $c = a + |b|$ és $n_0 = 1$ választással.

Néhány olvasó, aki már korábban is találkozott az O-jelöléssel, talán furcsának találja, hogy pl. azt kell írnunk: $n = O(n^2)$. Egyes szerzők (informálisan) az O-jelölést aszimptotikusan éles korlátok leírására használják úgy, ahogy mi a Θ-jelölést definiáltuk. Azonban ha ebben a könyvben azt írjuk, hogy $f(n) = O(g(n))$, akkor csupán azt kívánjuk meg, hogy $f(n)$-nek aszimptotikus felső korlátja legyen $g(n)$ egy állandószorosa, a felső korlát élességéről semmit sem állítunk. Manapság az algoritmusok irodalmában már általánossá vált az aszimptotikus felső korlát és az aszimptotikusan éles felső korlát megkülönböztetése.

Az O-jelölés használatával gyakran úgy is meg tudjuk határozni az algoritmus futási idejét, hogy pusztán az algoritmus egészének a szerkezetét vizsgáljuk. Pl. a 2. fejezetben tárgyalt beszúró rendező algoritmus egymásba ágyazott ciklusokat tartalmazó szerkezetéből közvetlenül adódik az $O(n^2)$ felső korlát a legrosszabb futási időre; a belső ciklus költségét az $O(1)$ konstanssal becsülhetjük, az i és j indexek mindegyike legfeljebb n, és a belső ciklust legfeljebb egyszer hajtjuk végre az n^2 db i, j pár mindegyikére.

Mivel az O-jelölés egy felső korlátot határoz meg, ezért amikor egy algoritmus legrosszabb esetbeli futási idejére használjuk, akkor ezzel minden bemenetre felső korlátot adunk az algoritmus futási idejére. Így a beszúró rendezés legrosszabb futási idejére vonatkozó $O(n^2)$ korlát az algoritmus bármely bemenetéhez tartozó futási időre is korlát. A $\Theta(n^2)$ korlát a beszúró rendezés legrosszabb futási idejére, de ebből nem következik, hogy $\Theta(n^2)$ korlátja a beszúró rendezés futási idejének *minden* bemenetre. Pl. a 2. fejezetben láttuk, hogy amikor a bemenet már rendezett, akkor a beszúró rendezés $\Theta(n)$ idő alatt fut le.

Az a kijelentés, hogy a beszúró rendezés futási ideje $O(n^2)$, pontatlan, mivel egy adott n-re a tényleges futási idő nemcsak n-től, hanem a konkrét bemenettől is függ. Így valójában a futási idő n-nek nem függvénye. Amikor azt mondjuk, hogy a „futási idő $O(n^2)$”, akkor ez azt jelenti, hogy a legrosszabb futási idő (ami függvénye n-nek) $O(n^2)$, vagy másképpen, nem számít az, hogy melyik n méretű bemenetet választjuk, a futási idő a bemenetek azon halmazán $O(n^2)$ lesz.

Ω-jelölés

Ahogy az O-jelölés aszimptotikus *felső* korlátot, úgy az Ω-jelölés ***aszimptotikus alsó korlátot*** ad a függvényre. Egy adott $g(n)$ függvény esetén $\Omega(g(n))$-nel (kiejtve „ómega $g(n)$” vagy „nagy ómega $g(n)$”) jelöljük a függvényeknek azt a halmazát, amelyre

$$\Omega(g(n)) = \{f(n) : \text{létezik } c \text{ és } n_0 \text{ pozitív konstans úgy, hogy } 0 \le cg(n) \le f(n) \text{ teljesül minden } n \ge n_0 \text{ esetén}\}.$$

Az Ω-jelölést szemlélteti a 3.1(c) ábra. Minden az n_0-tól jobbra lévő n értékre az $f(n)$ függvényérték $cg(n)$-nel egyenlő vagy annál nagyobb.

Az eddig látott aszimptotikus jelölések definícióját felhasználva könnyen bizonyítható az alábbi fontos tétel (lásd a 3.1-5. gyakorlatot).

3.1. tétel. *Bármely két $f(n)$ és $g(n)$ függvény esetén $f(n) = \Theta(g(n))$ akkor és csak akkor, ha $f(n) = O(g(n))$ és $f(n) = \Omega(g(n))$.*

E tétel alkalmazása például a következő. Az előbb beláttuk, hogy $an^2 + bn + c = \Theta(n^2)$, ahol b, c tetszőleges állandók, $a > 0$. A tételből azonnal következik, hogy $an^2 + bn + c = \Omega(n^2)$ és $an^2 + bn + c = O(n^2)$. A gyakorlatban a 3.1. tételt általában nem arra használjuk, hogy aszimptotikusan éles korlát létezéséből az aszimptotikus felső és alsó korlát létezését mutassuk meg, hanem az aszimptotikusan éles korlát létét mutatjuk meg az aszimptotikus alsó és felső korlátok felhasználásával.

Mivel az Ω-jelöléssel alsó korlátot adunk meg, ezért amikor egy algoritmus legjobb esetbeli futási idejének becslésére használjuk, akkor egyúttal tetszőleges bemenet esetére is korlátot adunk az algoritmus futási idejére. Például a beszúró rendezés legjobb futási ideje $\Omega(n)$, amiből következik, hogy a beszúró rendezés futási ideje $\Omega(n)$ minden bemeneten.

Így a beszúró rendezés futási ideje $\Omega(n)$ és $O(n^2)$ közé esik, mivel n egy elsőfokú és n egy másodfokú függvénye közé bárhova eshet. Ráadásul ezek a korlátok aszimptotikusan a lehető legélesebbek: pl. a beszúró rendezés futási ideje nem $\Omega(n^2)$, mivel a beszúró rendezés $\Theta(n)$ idő alatt lefut, ha a bemenet már rendezve van. Ennek ellenére nem ellentmondás azt állítani, hogy a beszúró rendezés futási ideje a *legrosszabb esetben* $\Omega(n^2)$, mivel létezik olyan bemenet, amikor az algoritmusnak $\Omega(n^2)$ időre van szüksége. Amikor azt mondjuk, hogy az algoritmus *futási ideje* (jelző nélkül) $\Omega(g(n))$, akkor ezt úgy értjük, *nem számít, hogy melyik n méretű bemeneteket választottuk*, elég nagy n esetén a bemenetek azon halmazán a futási idő legalább $g(n)$ egy konstansszorosa.

Aszimptotikus jelölések képletekben

Már láttuk, hogyan használhatunk aszimptotikus jelöléseket matematikai képletekben; pl. mikor bevezettük az O-jelölést, azt írtuk, hogy „$n = O(n^2 l)$.” Esetleg azt is írhatnánk, hogy $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Hogyan értelmezzünk egy ilyen képletet?

Ha az aszimptotikus jelölés egyedül van az egyenlet jobb oldalán, mint $n = O(n^2)$ esetén, akkor úgy definiáltuk az egyenlőségjelet, mint a halmazhoz tartozás jelét: $n \in O(n^2)$. Általában azonban, egy képletben előforduló aszimptotikus jelölést úgy tekintünk, mint egy olyan ismeretlen függvény jelölését, amit nem is akarunk megnevezni. Pl. a $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ képlet azt jelenti, hogy $2n^2 + 3n + 1 = 2n^2 + f(n)$, ahol $f(n)$ egy $\Theta(n)$ halmazbeli függvény. Ebben az esetben $f(n) = 3n + 1$, ami tényleg benne van $\Theta(n)$-ben.

Ekképpen használva, az aszimptotikus jelölés segít a képletből eltávolítani a lényegtelen részleteket. Pl. a 2. fejezetben rekurzívan kifejeztük az összefésülő rendezés legrosszabb futási idejét:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Ha csak $T(n)$ aszimptotikus viselkedése érdekel minket, akkor nincs értelme minden alacsonyabb rendű tagot pontosan megadni; mindet beleértjük abba a névtelen függvénybe, amit a $\Theta(n)$ taggal jelölünk.

Megállapodás szerint egy kifejezésben a névtelen függvények száma egyenlő az előforduló aszimptotikus jelek számával. Pl. a

$$\sum_{i=1}^{n} O(i)$$

kifejezésben csak egy ismeretlen függvény van (i-nek egy függvénye). Ez a kifejezés *nem* ugyanaz, mint $O(1) + O(2) + \cdots + O(n)$, aminek nincs igazán világos jelentése.

Bizonyos esetekben aszimptotikus jelölés előfordulhat az egyenlet bal oldalán is, mint például

$$2n^2 + \Theta(n) = \Theta(n^2).$$

Ezeket az egyenleteket a következő szabály szerint értelmezzük: *Mindegy, hogy miképpen választjuk meg az egyenlet bal oldalán lévő ismeretlen függvényt, van mód arra, hogy a jobb oldali ismeretlen függvényt úgy válasszuk meg, hogy az egyenlőség igaz legyen.* Így példánk jelentése az, hogy *tetszőleges* $f(n) \in \Theta(n)$ függvényhez *létezik* $g(n) \in \Theta(n^2)$ úgy, hogy $2n^2 + f(n) = g(n)$ minden n-re. Más szóval az egyenlet jobb oldala durvábban, kevésbé pontosan írja le a függvényt, mint az egyenlet bal oldala.

Több ilyen összefüggés is összekapcsolható, mint pl.

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

Mindegyik egyenletet külön-külön értelmezhetjük a fenti szabály segítségével. Az első egyenlet azt állítja, hogy *létezik* $f(n) \in \Theta(n)$ úgy, hogy $2n^2 + 3n + 1 = 2n^2 + f(n)$ teljesül minden n-re. A második egyenlet pedig azt mondja, hogy *tetszőleges* $g(n) \in \Theta(n)$-hez (mint például az említett $f(n)$ függvényhez) *létezik* $h(n) \in \Theta(n^2)$ úgy, hogy $2n^2 + g(n) = h(n)$ minden n-re. Vegyük észre, hogy ebből az értelmezésből következik, hogy $2n^2 + 3n + 1 = \Theta(n^2)$, ami az egyenletek összekapcsolásával szemléletből is adódik.

o-jelölés

Az O-jelölés által adott felső korlát aszimptotikusan vagy éles, vagy nem. A $2n^2 = O(n^2)$ aszimptotikusan éles korlát, míg a $2n = O(n^2)$ nem az. A o-jelölést aszimptotikusan nem éles felső korlát jelölésére használjuk. $o(g(n))$ (olvasd: „kis ordó $g(n)$”) formális definícióját az alábbi halmaz adja:

$$o(g(n)) = \{f(n) : \text{ tetszőleges pozitív } c > 0 \text{ konstansra létezik olyan } n_0 > 0 \text{ konstans, melyre } 0 \le f(n) < cg(n) \text{ minden } n \ge n_0 \text{ esetén}\}.$$

Pl. $2n = o(n^2)$, de $2n^2 \neq o(n^2)$.

Az O- és a o-jelölés definíciója hasonló. A fő különbség az, hogy míg az $f(n) = O(g(n))$ esetén *létezik* olyan $c > 0$ állandó, mellyel a $0 \le f(n) \le cg(n)$ egyenlőtlenség teljesül, addig az $f(n) = o(g(n))$ esetén a $0 \le f(n) \le cg(n)$ egyenlőtlenség *minden* $c > 0$ állandóra igaz. Szemléletesen, a o-jelölésben az $f(n)$ függvény jelentéktelenné válik a $g(n)$ függvényhez képest, ha n a végtelenhez tart, azaz

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0. \tag{3.1}$$

Többen a *o*-jelölés definíciójaként használják ezt a határértéket; ebben a könyvben azt is kikötjük, hogy az ismeretlen függvények aszimptotikusan nemnegatívok legyenek.

ω-jelölés

Analóg módon a ω-jelölés úgy viszonyul az Ω jelöléshez, mint a *o*-jelölés az *O*-jelöléshez. A ω-jelöléssel aszimptotikusan nem éles alsó korlátot jelzünk. Egy lehetséges definíciója pl. a következő:

$$f(n) \in \omega(g(n)) \text{ akkor és csak akkor, ha } g(n) \in o(f(n)).$$

Formálisan azonban az alábbi halmazzal definiáljuk $\omega(g(n))$-t (olvasd: „kis ómega $g(n)$”):

$$\omega(g(n)) = \{f(n) : \text{ tetszőleges pozitív } c > 0 \text{ konstansra létezik olyan } n_0 > 0 \text{ konstans, melyre } 0 \le cg(n) < f(n) \text{ minden } n \ge n_0 \text{ esetén}\}.$$

Pl. $n^2/2 = \omega(n)$, de $n^2/2 \neq \omega(n^2)$. Az $f(n) = \omega(g(n))$ egyenlőségből következik, hogy

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

ha ez a határérték létezik. Azaz $f(n)$ tetszőlegesen nagy lesz $g(n)$-hez képest, ha n a végtelenhez tart.

Függvények összehasonlítása

Számos – valós számokra vonatkozó – összefüggés alkalmazható aszimptotikus összehasonlításokra is. A továbbiakban tegyük fel, hogy $f(n)$ és $g(n)$ aszimptotikusan pozitív.

Tranzitivitás:

$$\begin{array}{llll}
\text{ha } f(n) = \Theta(g(n)) \text{ és } g(n) = \Theta(h(n)), & \text{akkor} & f(n) = \Theta(h(n)), \\
\text{ha } f(n) = O(g(n)) \text{ és } g(n) = O(h(n)), & \text{akkor} & f(n) = O(h(n)), \\
\text{ha } f(n) = \Omega(g(n)) \text{ és } g(n) = \Omega(h(n)), & \text{akkor} & f(n) = \Omega(h(n)), \\
\text{ha } f(n) = o(g(n)) \text{ és } g(n) = o(h(n)), & \text{akkor} & f(n) = o(h(n)), \\
\text{ha } f(n) = \omega(g(n)) \text{ és } g(n) = \omega(h(n)), & \text{akkor} & f(n) = \omega(h(n)).
\end{array}$$

Reflexivitás:

$$\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}$$

Szimmetria:

$$f(n) = \Theta(g(n)) \text{ akkor és csak akkor, ha } g(n) = \Theta(f(n)).$$

Felcserélt szimmetria:

$$\begin{aligned}
f(n) &= O(g(n)) \text{ akkor és csak akkor, ha } g(n) = \Omega(f(n)), \\
f(n) &= o(g(n)) \text{ akkor és csak akkor, ha } g(n) = \omega(f(n)).
\end{aligned}$$

Mivel ezek a tulajdonságok érvényesek az aszimptotikus jelölésekre, ezért párhuzamot vonhatunk két függvény, f és g, valamint két valós szám, a és b összehasonlítása között:

$$\begin{aligned} f(n) = O(g(n)) &\approx a \leq b, \\ f(n) = \Omega(g(n)) &\approx a \geq b, \\ f(n) = \Theta(g(n)) &\approx a = b, \\ f(n) = o(g(n)) &\approx a < b, \\ f(n) = \omega(g(n)) &\approx a > b. \end{aligned}$$

Az $f(n)$ függvény ***aszimptotikusan kisebb,*** mint $g(n)$, ha $f(n) = o(g(n))$, illetve $f(n)$ ***aszimptotikusan nagyobb,*** mint $g(n)$, ha $f(n) = \omega(g(n))$. Az $f(n)$ függvény ***aszimptotikusan egyenlő*** a $g(n)$ függvénnyel, ha $f(n) = \Theta(g(n))$.

A valós számok következő tulajdonsága azonban nem vihető át aszimptotikus jelölésekre.

Trichotómia: Bármely két valós a és b szám esetén az alábbi tulajdonságok közül pontosan egy teljesül: $a < b$, $a = b$ vagy $a > b$.

Annak ellenére, hogy bármely két valós szám összehasonlítható, bármely két függvény nem hasonlítható össze aszimptotikusan. Azaz előfordulhat, hogy két függvényre, $f(n)$-re és $g(n)$-re, sem az nem teljesül, hogy $f(n) = O(g(n))$, sem pedig az, hogy $f(n) = \Omega(g(n))$. Pl. az n és az $n^{1+\sin n}$ függvény aszimptotikus jelölésekkel nem hasonlítható össze, mivel az $n^{1+\sin n}$ kitevője 0 és 2 között minden értéket felvéve oszcillál.

Gyakorlatok

3.1-1. Legyen $f(n)$ és $g(n)$ aszimptotikusan nemnegatív függvény. Bizonyítsuk be a Θ-jelölés alapdefiníciójával, hogy $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1-2. Mutassuk meg, hogy tetszőleges valós a, b ($b > 0$) konstansokra

$$(n + a)^b = \Theta(n^b). \tag{3.2}$$

3.1-3. Magyarázzuk meg, hogy miért semmitmondó az alábbi állítás: „Az A algoritmus futási ideje legalább $O(n^2)$”.

3.1-4. Igaz-e, hogy: a) $2^{n+1} = O(2^n)$ b) $2^{2n} = O(2^n)$?

3.1-5. Bizonyítsuk be a 3.1. tételt.

3.1-6. Bizonyítsuk be, hogy egy algoritmus futási ideje akkor és csak akkor $\Theta(g(n))$, ha legrosszabb futási ideje $O(g(n))$, a legjobb futási ideje pedig $\Omega(g(n))$.

3.1-7. Bizonyítsuk be, hogy $o(g(n)) \cap \omega(g(n)) = \emptyset$.

3.1-8. Jelöléseinket kétváltozós esetre is kiterjeszthetjük, mikor az m, n paraméterek egymástól függetlenül, különböző sebességgel tartanak a végtelenhez. Egy adott $g(n, m)$ függvény esetén $O(g(n, m))$-mel jelöljük a következő függvényhalmazt:

$$\begin{aligned} O(g(n, m)) = \{ f(n, m) : \quad & \text{létezik } c, n_0 \text{ és } m_0 \text{ pozitív konstans úgy, hogy} \\ & 0 \leq f(n, m) \leq cg(n, m) \text{ teljesül} \\ & \text{minden } n \geq n_0 \text{ és } m \geq m_0 \text{ esetén}\}. \end{aligned}$$

Adjuk meg $\Omega(g(n, m))$ és $\Theta(g(n, m))$ megfelelő definícióját.

3.2. Szokásos jelölések és alapfüggvények

Ebben az alfejezetben áttekintünk néhány alapfüggvényt és szokásos jelölést, valamint a köztük lévő kapcsolatokat vizsgáljuk, és az aszimptotikus jelölések használatát is szemléltetjük.

Monotonitás

Az $f(n)$ függvény ***monoton növekedő,*** ha minden $m \leq n$ esetén $f(m) \leq f(n)$. Hasonlóan, ***monoton csökkenő,*** ha minden $m \leq n$ esetén $f(m) \geq f(n)$. Az $f(n)$ függvény ***szigorúan monoton növekedő,*** ha minden $m < n$ esetén $f(m) < f(n)$, és ***szigorúan monoton csökkenő,*** ha minden $m < n$ esetén $f(m) > f(n)$.

Alsó egészrész és felső egészrész

Minden x valós szám esetén $\lfloor x \rfloor$-szel (olvasd: x alsó egészrésze) jelöljük azt a legnagyobb egész számot, ami kisebb vagy egyenlő x-szel. Azt a legkisebb egész számot pedig, ami nagyobb vagy egyenlő x-szel, $\lceil x \rceil$-szel (olvasd: x felső egészrésze) jelöljük. Minden valós x esetén

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1. \tag{3.3}$$

Minden n egész számra

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n,$$

és minden valós $n \geq 0$ értékre és $a, b > 0$ egészre

$$\left\lceil \frac{\lceil \frac{n}{a} \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil, \tag{3.4}$$

$$\left\lfloor \frac{\lfloor \frac{n}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor, \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b}, \tag{3.6}$$

$$\left\lceil \frac{a}{b} \right\rceil \geq \frac{a - (b-1)}{b}. \tag{3.7}$$

Az alsó és felső egészrész függvények monoton növekedők.

Osztási maradékok

Tetszőleges egész a és pozitív egész n egész esetén jelölje $a \bmod n$ az a szám n-nel vett ***osztási maradékát,*** vagyis

$$a \bmod n = a - \left\lfloor \frac{a}{n} \right\rfloor n. \tag{3.8}$$

Az a egész n-nel vett osztási maradékára adott jelölés mellett hasznos lesz, ha arra is definiálunk egy külön jelölést, hogy az a és b egészek n-nel vett osztási maradéka megegyezik. Ha $(a \bmod n) = (b \bmod n)$, akkor ezt úgy jelöljük, hogy $a \equiv b \pmod{n}$, és azt

mondjuk, hogy a és b ***kongruens*** modulo n. Másként fogalmazva, $a \equiv b \pmod{n}$, ha a és b ugyanazt az osztási maradékot adja n-nel osztva. Ekvivalens módon: $a \equiv b \pmod{n}$ akkor és csak akkor, ha n osztja a $b - a$ különbséget. Ha a és b nem ekvivalens modulo n, akkor erre az $a \not\equiv b \pmod{n}$ jelölést használjuk.

Polinomok

Adott d pozitív egész számra az ***n változó d-edfokú polinomján*** egy

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

alakú függvényt értünk, ahol az $a_0, a_1, \ldots, a_d$ állandók a polinom ***együtthatói,*** és $a_d \neq 0$. Egy polinom aszimptotikusan pozitív akkor és csak akkor, ha $a_d > 0$. Az aszimptotikusan pozitív d-edfokú $p(n)$ polinomra teljesül, hogy $p(n) = \Theta(n^d)$. Minden valós $a \geq 0$ állandó esetén n^a monoton növekedő, minden valós $a \leq 0$ állandó esetén n^a monoton csökkenő. Az $f(n)$ függvény ***polinomiálisan korlátos,*** ha $f(n) = n^{O(1)}$, ami ekvivalens azzal, hogy $f(n) = O(n^k)$ valamely k állandóra.

Hatványok

A következő azonosságok teljesülnek minden $a \neq 0$, m, n valós számra:

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

Ha $a \geq 1$, akkor az n változótól függő a^n függvény monoton nő. Mikor úgy kényelmesebb, akkor $0^0 = 1$ feltételezéssel élünk.

A polinomok és hatványok növekedése a következőképpen hasonlítható össze. Minden $a > 1$ és b valós állandó esetén

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0, \tag{3.9}$$

amiből

$$n^b = o(a^n).$$

Azaz minden exponenciális függvény, amelynek alapja nagyobb 1-nél, gyorsabban növekszik, mint bármely polinom függvény.

A természetes logaritmus alapját $e = 2{,}71828\ldots$ jelöli. Minden valós x-re

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{3.10}$$

teljesül, ahol a „ ! ” faktoriálist jelent, amit ennek az alfejezetnek egy későbbi részében fogunk definiálni. Minden valós x-re teljesül az

$$e^x \geq 1 + x \tag{3.11}$$

egyenlőtlenség, ahol az egyenlőség csak akkor áll fenn, ha $x = 0$. Az $|x| \leq 1$ esetben teljesül az alábbi közelítés:

$$1 + x \leq e^x \leq 1 + x + x^2. \tag{3.12}$$

Ha $x \to 0$, akkor az e^x függvény $(1 + x)$-szel való közelítése elég jó:

$$e^x = 1 + x + \Theta(x^2).$$

(Ebben a képletben az aszimptotikus jelölést az $x \to \infty$ helyett az $x \to 0$ határértékbeli viselkedés leírására használjuk.) Minden x-re teljesül, hogy

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x. \tag{3.13}$$

Logaritmusok

A következő jelöléseket fogjuk használni:

$$\begin{array}{rcll} \lg n & = & \log_2 n & \text{(kettes alapú logaritmus),} \\ \ln n & = & \log_e n & \text{(természetes logaritmus),} \\ \lg^k n & = & (\lg n)^k & \text{(hatványozás),} \\ \lg\lg n & = & \lg(\lg n) & \text{(kompozíció).} \end{array}$$

Egy lényeges jelölésbeli szokás az, amit mi is elfogadunk, hogy *a logaritmusfüggvény csak a képletben közvetlenül mellette álló tagra vonatkozik*, azaz $\lg n+k$ azt jelenti, hogy $(\lg n)+k$, nem pedig azt, hogy $\lg(n+k)$. Bármely $b > 1$ konstans mellett $n > 0$ esetén a $\log_b n$ függvény szigorúan monoton nő.

Minden valós $a > 0$, $b > 0$, $c > 0$ és n számra igaz, hogy

$$\begin{aligned} a &= b^{\log_b a}, \\ \log_c(ab) &= \log_c a + \log_c b, \\ \log_b a^n &= n \log_b a, \\ \log_b a &= \frac{\log_c a}{\log_c b}, \qquad (3.14) \\ \log_b \frac{1}{a} &= -\log_b a, \\ \log_b a &= \frac{1}{\log_a b}, \\ a^{\log_b c} &= c^{\log_b a}, \qquad (3.15) \end{aligned}$$

ahol a fenti egyenlőségek egyikében sem 1 a logaritmus alapja.

A (3.14) egyenlőség miatt a logaritmus alapjának egyik állandóról egy másikra való megváltoztatása csak egy állandó tényezővel változtatja a logaritmus értékét, ezért gyakran

fogjuk az $\lg n$ jelölést használni, mikor nem lényeges számunkra ez az állandó tényező, mint pl. az O-jelölésben. A számítástechnikai szakemberek a 2-t tartják a legtermészetesebb alapnak, mivel nagyon sok algoritmus és adatszerkezet tartalmazza a feladat két részre való felbontását.

Az alábbi egyszerű végtelen sor megadja az $\ln(1 + x)$-et, ha $|x| < 1$:

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .$$

Az $x > -1$ esetre teljesül a következő egyenlőtlenség:

$$\frac{x}{1 + x} \leq \ln(1 + x) \leq x, \tag{3.16}$$

ahol az egyenlőség csak az $x = 0$ esetben áll fenn.

Azt mondjuk, hogy az $f(n)$ függvény ***polilogaritmikusan korlátos,*** ha $f(n) = O(\lg^k n)$ valamilyen k konstansra. Összehasonlíthatjuk a polinomok és polilogaritmusok növekedését, ha a (3.9) képletben n helyére $\lg n$-et, a helyére pedig 2^a-t helyettesítünk, ekkor

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0.$$

Ebből a határértékből azt kapjuk, hogy bármely $a > 0$ állandóra

$$\lg^b n = o(n^a).$$

Azaz minden pozitív polinomfüggvény gyorsabban nő, mint bármelyik polilogaritmikus függvény.

Faktoriálisok

Az $n!$ számot (olvasd: „n faktoriális”) $n \geq 0$ egészekre definiáljuk a következőképpen:

$$n! = \begin{cases} 1, & \text{ha } n = 0, \\ n \cdot (n-1)!, & \text{ha } n > 0. \end{cases}$$

Azaz $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

A függvény egyik durva felső korlátja $n! \leq n^n$, mivel a faktoriálisban az n tényezős szorzat minden tényezője legfeljebb n. A ***Stirling-formula***

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right), \tag{3.17}$$

ahol e a természetes logaritmus alapja. Ez élesebb felső korlát, és egyben alsó korlátot is ad. A Stirling-formula segítségével bizonyíthatjuk (3.2-3. gyakorlat), hogy

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n). \end{aligned} \tag{3.18}$$

Az alábbi korlát szintén minden $n \geq 1$-re teljesül:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n}, \tag{3.19}$$

ahol

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}\,. \tag{3.20}$$

Iterált függvények

Az $f(n)$ függvény i-edik iteráltján azt az $f^{(i)}(n)$ függvényt értjük, amelynek az értékét úgy kapjuk, hogy a kiindulási n értéken az $f(n)$ függvényt i-szer egymás után alkalmazzuk. Formálisan, legyen $f(n)$ egy a valós számokon értelmezett függvény. Nemnegatív i egészek esetén a következő rekurziós képletet adhatjuk:

$$f^{(i)}(n) = \begin{cases} n, & \text{ha } i = 0, \\ f(f^{(i-1)}(n)), & \text{ha } i > 0. \end{cases}$$

Például $f(n) = 2n$ esetén $f^{(i)}(n) = 2^i n$.

Az iterált logaritmusfüggvény

Az iterált logaritmusfüggvényt $\lg^* n$-nel fogjuk jelölni (olvasd: „logaritmus csillag n”), és a következőképpen definiáljuk. Legyen $\lg^{(i)} n$ a fenti módon definiálva, ahol most $f(n) = \lg n$. Mivel egy nempozitív szám logaritmusa nem értelmezett, ezért $\lg^{(i)} n$ csak akkor van értelmezve, ha $\lg^{(i-1)} n > 0$. Fontos, hogy megkülönböztessük a $\lg^{(i)} n$-t (az n argumentumra egymás után i-szer alkalmazzuk a logaritmusfüggvényt) a $\lg^i n$-től (logaritmus n az i-edik hatványon). Az iterált logaritmusfüggvény definíciója a következő:

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}.$$

Az iterált logaritmusfüggvény *nagyon* lassan növekszik:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

Mivel a megfigyelhető világegyetemben lévő atomok számát 10^{80}-ra becsülik, ami sokkal kevesebb, mint 2^{65536}, ezért ritkán fordul elő olyan n méretű bemenet, amelyre $\lg^* n > 5$.

Fibonacci-számok

A ***Fibonacci-számokat*** az alábbi rekurzióval definiáljuk:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2}, \qquad \text{ha } i \geq 2. \end{aligned} \tag{3.21}$$

Minden Fibonacci-szám a megelőző kettőnek az összege, tehát a szóban forgó sorozat

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots\ .$$

A Fibonacci-számok kapcsolatban vannak az ***aranymetszés*** ϕ arányával és annak $\widehat{\phi}$ konjugáltjával, amelyek az alábbi képletekkel adhatók meg:

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1{,}61803\ldots, \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -0{,}61803\ldots. \end{aligned} \tag{3.22}$$

Pontosabban az teljesül, hogy

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}, \tag{3.23}$$

amit teljes indukcióval bizonyíthatunk (3.2-6. gyakorlat). Mivel $|\widehat{\phi}| < 1$, ezért $|\widehat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, vagyis az i-edik Fibonacci-szám, F_i, egyenlő $\phi^i/\sqrt{5}$ legközelebbi egészre kerekített értékével. Azaz a Fibonacci-számok exponenciálisan nőnek.

Gyakorlatok

3.2-1. Mutassuk meg, hogy amennyiben $f(n)$ és $g(n)$ monoton növekedő függvények, akkor $f(n) + g(n)$ és $f(g(n))$ is azok, és ha ezen kívül $f(n)$ és $g(n)$ nemnegatívak, akkor $f(n) \cdot g(n)$ is monoton nő.

3.2-2. Bizonyítsuk be a (3.15) képletet.

3.2-3. Bizonyítsuk be a (3.18) képletet. Bizonyítsuk be azt is, hogy $n! = \omega(2^n)$ és $n! = o(n^n)$.

3.2-4.⋆ Polinomiálisan korlátos-e a $\lceil \lg n \rceil!$ függvény? Polinomiálisan korlátos-e a $\lceil \lg \lg n \rceil!$ függvény?

3.2-5.⋆ Melyik nagyobb aszimptotikusan: $\lg(\lg^* n)$ vagy $\lg^*(\lg n)$?

3.2-6. Bizonyítsuk be teljes indukcióval, hogy az i-edik Fibonacci-szám kielégíti az

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}$$

egyenletet, ahol ϕ az aranymetszés arányszáma, $\widehat{\phi}$ pedig annak konjugáltja.

3.2-7. Bizonyítsuk be, hogy ha $i \geq 0$, akkor az $(i + 2)$-dik Fibonacci-szám kielégíti az $F_{i+2} \geq \phi^i$ egyenlőtlenséget.

Feladatok

3-1. Polinomok aszimptotikus viselkedése

Legyen

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

az n változó d-edfokú polinomja, ahol $a_d > 0$, és k legyen egy állandó. Az aszimptotikus jelölések definíciójának segítségével bizonyítsuk be a következő tulajdonságokat.

a. Ha $k \geq d$, akkor $p(n) = O(n^k)$.

b. Ha $k \leq d$, akkor $p(n) = \Omega(n^k)$.

c. Ha $k = d$, akkor $p(n) = \Theta(n^k)$.

d. Ha $k > d$, akkor $p(n) = o(n^k)$.

e. Ha $k < d$, akkor $p(n) = \omega(n^k)$.

3-2. Relatív aszimptotikus növekedések

Írjuk be az alábbi táblázatba minden (A, B) pár esetén, hogy az $A = O(B), A = o(B), A = \Omega(B), A = \omega(B)$ és $A = \Theta(B)$ közül melyik teljesül. Tegyük fel, hogy $k \geq 1$, $\epsilon > 0$ és $c > 1$ állandók. A válasz a táblázat minden cellájában „igen” vagy „nem” legyen.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3. Rendezés az aszimptotikus növekedési rend alapján

a. Állítsuk sorrendbe növekedési rendjük szerint az alábbi függvényeket; azaz keressük a függvényeknek egy olyan $g_1, g_2, \ldots, g_{30}$ elrendezését, amely kielégíti a $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$ feltételeket. Listánkat bontsuk ekvivalencia osztályokra úgy, hogy $f(n)$ és $g(n)$ akkor és csak akkor tartozzon ugyanabba az osztályba, ha $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Adjunk meg egy olyan nemnegatív $f(n)$ függvényt, hogy egyetlen, az (a) részben el őforduló, $g_i(n)$ függvényre se teljesüljön, hogy $f(n) = O(g_i(n))$ vagy $f(n) = \Omega(g_i(n))$.

3-4. Az aszimptotikus jelölések tulajdonságai

Legyen $f(n)$ és $g(n)$ aszimptotikusan pozitív függvény. Igazoljuk, vagy cáfoljuk a következő állításokat.

a. $f(n) = O(g(n)) \Rightarrow g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n)) \Rightarrow \lg(f(n)) = O(\lg(g(n)))$, ahol $\lg(g(n)) > 0$ és $f(n) \geq 1$ minden elég nagy n-re.

d. $f(n) = O(g(n)) \Rightarrow 2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5. Variációk O-ra és Ω-ra

Néhány szerző az Ω-t kicsit másképp definiálja; mi erre a változatra az $\overset{\infty}{\Omega}$ (olvasd: „ómega végtelen") jelölést használjuk. Azt mondjuk, hogy $f(n) = \overset{\infty}{\Omega}(g(n))$, ha létezik c pozitív állandó úgy, hogy $f(n) \geq cg(n) \geq 0$ teljesül végtelen sok egész n-re.

a. Mutassuk meg, hogy bármely két aszimptotikusan nemnegatív $f(n)$ és $g(n)$ függvény esetén vagy $f(n) = O(g(n))$, vagy $f(n) = \overset{\infty}{\Omega}(g(n))$, vagy mindkettő teljesül, ha azonban $\overset{\infty}{\Omega}$ helyett Ω-t használunk, akkor nem igaz az állítás.

b. Mik a lehetséges előnyei és hátrányai annak, ha a programok futási idejének jellemzésére $\overset{\infty}{\Omega}$-t használunk Ω helyett?

Néhány szerző az O-t is kissé másképp definiálja; mi erre az alternatív definícióra az O' jelölést használjuk. Azt mondjuk, hogy $f(n) = O'(g(n))$ akkor és csak akkor, ha $|f(n)| = O(g(n))$.

c. Mi történik a 3.1. tétel „akkor és csak akkor" állításával az egyes irányokban, ha ezt az új definíciót használjuk?

Néhány szerző a $\widetilde{O}$-n (olvasd: „gyenge ordó") szimbólumon a logaritmikus tényezők elhanyagolásával kapott O-jelölést érti.

$$\widetilde{O}(g(n)) = \{f(n) : \text{ létezik } c, k \text{ és } n_0 \text{ pozitív konstans úgy, hogy } 0 \leq f(n) \leq cg(n)\lg^k(n) \text{ teljesül minden } n \geq n_0 \text{ esetén}\}.$$

d. Definiáljuk hasonló módon $\widetilde{\Omega}$-t és $\widetilde{\Theta}$-t. Bizonyítsuk be a 3.1. tétel ennek megfelelő változatát.

3-6. Iterált függvények

Az $\lg^*$ függvénynél használt „*" iterációs operátor alkalmazható a valós számokon értelmezett, monoton növekedő függvényekre is. Adott $c \in \mathbf{R}$ állandó esetén az f_c^* iterált függvényt a

$$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\}$$

képlettel definiáljuk, de ennek nem kell minden esetben jól definiáltnak lennie. Más szóval az $f_c^*(n)$ mennyiség az f függvényre alkalmazott iterációk száma, ami ahhoz szükséges, hogy az argumentumát c-re vagy annál kisebbre csökkentsük.

Adjuk meg $f_c^*(n)$ lehető legélesebb korlátját az alábbi $f(n)$ függvényekre és c állandókra.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Megjegyzések a fejezethez

Knuth [182] az *O*-jelölés eredetét P. Bachmann 1892-ben írt számelméleti cikkére vezeti vissza. A *o*-jelölést 1909-ben E. Landau használta egy, a prímszámok eloszlásáról szóló eszmefuttatásához. Az Ω és a Θ jelöléseket Knuth [186] javasolta a népszerű, de alakilag pontatlan gyakorlat helyett, hogy az *O*-jelölést az alsó és felső korlát jelölésére is használták. Sokan továbbra is az *O*-jelölést használják, pedig a Θ-jelölés pontosabb. Knuth [182, 186], valamint Brassard és Bratley [46] további leírást nyújt az aszimptotikus jelölések történetéről és fejlődéséről.

Az aszimptotikus jelöléseket nem minden szerző definiálja ugyanúgy, de a különféle definíciók a leggyakrabban előforduló esetekben egybeesnek. Az alternatív definíciók némelyike magában foglal olyan függvényeket is, amelyek ugyan aszimptotikusan nem nemnegatívak, de az abszolút értékük megfelelőképpen korlátos.

A (3.19) egyenlőség Robbinstól [260] származik. Az alapfüggvények egyéb tulajdonságairól bármely jó matematikai szakkönyvben olvashatunk, mint pl. Abramowitz és Stegun [1], vagy Zwillinger [320], vagy analízis könyvekben, mint pl. Apostol [18], vagy Thomas és Finney [296]. Knuth [182], illetve Graham, Knuth és Patashnik [132] bőséges anyagot tartalmaz a számítástudományban használatos diszkrét matematikáról.

4. Függvények rekurzív megadása

Amint azt a 2.3.2. pontban már megemlítettük, ha egy algoritmus önmagát hívja egymás után, akkor futási ideje gyakran jellemezhető ***rekurzióval***, azaz egy egyenlőséggel (vagy egyenlőtlenséggel) és egy kezdeti értékkel. A rekurzióban szereplő ***rekurzív képlet*** olyan egyenlőség vagy egyenlőtlenség, amely egy függvény értékeit a kisebb helyeken felvett értékekkel fejezi ki. Például a 2.3.2. pontban láttuk, hogy az ÖSSZEFÉSÜLŐ-RENDEZÉS $T(n)$ legrosszabb futási ideje a

$$T(n) = \begin{cases} \Theta(1), & \text{ha } n = 1, \\ 2T(n/2) + \Theta(n), & \text{ha } n > 1 \end{cases} \tag{4.1}$$

rekurzióval írható le. Erről azt állítottuk, hogy a megoldásra $T(n) = \Theta(n \lg n)$ teljesül.

Ez a fejezet három módszert mutat a rekurzió megoldására (vagy feloldására) – azaz arra, hogyan kaphatunk aszimptotikus Θ vagy O korlátokat a megoldásra. A ***helyettesítő módszernél*** megsejtünk egy korlátot, majd teljes indukcióval igazoljuk sejtésünk helyességét. A ***rekurziós fa módszerben*** a rekurzív képlet alapján felépítünk egy fát, melyben egy adott szinten lévő csúcsok a rekurzió adott mélységében fellépő költségeknek felelnek meg; a rekurziót ezután az összegek becslésére szolgáló módszerekkel oldjuk meg. A ***mester módszer*** a

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

alakú rekurzív egyenlet megoldására ad korlátot, ahol $a \geq 1$, $b > 1$ és $f(n)$ egy adott függvény. Ehhez a módszerhez három esetet kell megjegyezni, de ha ezt egyszer megtesszük, utána már könnyedén tudunk aszimptotikus korlátot megadni sok egyszerű rekurzív képletre vonatkozóan.

Technikai részletek

A gyakorlatban a rekurzív problémák megfogalmazása és megoldása során bizonyos technikai részleteket elhanyagolunk. Jó példa ilyen elhallgatott részletre, hogy a függvények argumentumáról azt tesszük fel, hogy azok egészek. Rendes körülmények között, az algoritmus $T(n)$ futási idejét csak egész n-ekre definiáljuk, mivel a legtöbb algoritmus esetén a bemenet mérete mindig egész szám. Például, az ÖSSZEFÉSÜLŐ-RENDEZÉS legrosszabb futási idejét leíró rekurzió valójában

$$T(n) = \begin{cases} \Theta(1), & \text{ha } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & \text{ha } n > 1. \end{cases} \tag{4.2}$$

A kezdeti feltételek a részleteknek egy másik olyan jellegzetes csoportját alkotják, amelyet általában elhanyagolunk. Mivel az algoritmus futási ideje állandó méretű bemenet esetén állandó, ezért azokra a $T(n)$ függvényekre, amelyek algoritmusok futási idejére vonatkoznak, elég kicsi n esetén általában teljesül, hogy $T(n) = \Theta(1)$. Ezért az egyszerűség kedvéért általában elhagyjuk a rekurzióból a kezdeti feltételre vonatkozó részt, azzal a feltételezéssel, hogy kis n értékekre $T(n)$ állandó. Például, a (4.1) rekurziót úgy adjuk meg, hogy

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n), \tag{4.3}$$

anélkül, hogy explicit értékeket adnánk kis n-ekre. Ennek oka az, hogy bár $T(1)$ értékének változása megváltoztatja a rekurzió megoldását is, de az általában legfeljebb csak egy állandó szorzótényezővel változik, így a növekedés rendje változatlan marad.

Amikor megfogalmazunk és megoldunk egy rekurziót, gyakran elhagyjuk a felső és alsó egészrészt, valamint a kezdeti feltételt. E részletek nélkül haladunk előre, és a végén eldöntjük, hogy fontosak-e vagy sem. Általában nem fontosak, de azt azért tudnunk kell, hogy mikor azok. Ebben segít a gyakorlat, valamint néhány tétel, amely kimondja, hogy az algoritmusok elemzése során használatos rekurziók aszimptotikus korlátját nem befolyásolják ezek a részletek (lásd a 4.1. tételt). Ebben a fejezetben azonban mégis rámutatunk majd néhány ilyen részletre, hogy lássuk a rekurziók megoldási módszereinek finomságait is.

4.1. A helyettesítő módszer

A rekurziók helyettesítő módszerrel való megoldása két lépésből áll:

1. Sejtsük meg a megoldást.
2. Teljes indukcióval határozzuk meg az állandókat és igazoljuk a megoldás helyességét.

Az elnevezés onnan ered, hogy a helyesnek vélt megoldást be kell helyettesíteni a függvénybe, miközben az indukciós feltevést kisebb értékekre alkalmazzuk. Ez hatékony módszer, de nyilvánvalóan csak akkor alkalmazható, ha a helyes válasz könnyen megsejthető.

A helyettesítő módszer rekurzióval megadott függvény felső vagy alsó korlátjának a meghatározására is használható. Példaként határozzuk meg a (4.2) és a (4.3) képlethez hasonló

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \tag{4.4}$$

rekurzív képlettel jellemzett függvény egy felső korlátját. Sejtésünk az, hogy a megoldás a $T(n) = O(n \lg n)$. Módszerünk az, hogy bebizonyítjuk, megfelelően választott $c > 0$ állandóra $T(n) \leq cn \lg n$. Először is feltesszük, hogy ezzel a korláttal $\lfloor n/2 \rfloor$-re érvényes az egyenlőtlenség, azaz $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Ezt a rekurzív egyenletbe helyettesítve kapjuk, hogy

$$\begin{aligned} T(n) &\leq 2\left(c\left\lfloor\frac{n}{2}\right\rfloor \lg\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\right) + n \\ &\leq cn\lg\left(\frac{n}{2}\right) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\leq cn\lg n, \end{aligned}$$

ahol az utolsó lépés akkor igaz, ha $c \geq 1$.

Most a teljes indukció megkívánja, hogy megmutassuk, megoldásunk a kezdeti feltételekre is igaz. Azaz, meg kell mutatnunk, hogy meg tudjuk választani a c állandót úgy, hogy az elég nagy legyen ahhoz, hogy a $T(n) \leq cn\lg n$ korlát a kezdeti feltételekre is teljesüljön. Ez a követelmény néha problémát okoz. Tegyük fel, hogy $T(1) = 1$ a rekurzív egyenlet egyetlen kezdeti feltétele. Sajnos, ekkor nem tudjuk c értékét elég nagynak választani, mert $T(1) > c \cdot 1 \lg 1 = 0$. Így tehát az indukciós feltevés nem teljesül a kezdeti esetre, az indukciós bizonyítás nem tud elindulni.

Indukciós feltevések bizonyítása során a kezdeti feltételeknél felmerülő nehézségek könnyedén leküzdhetők. A (4.4) rekurziós képletnél azt a tényt használjuk ki, hogy az aszimptotikus jelölés csak azt kívánja meg, hogy a $T(n) \leq cn\lg n$ egyenlőtlenséget az $n \geq n_0$ esetre bizonyítsuk, ahol n_0 egy általunk választott állandó. Az ötlet az, hogy a problémát okozó $T(1) = 1$ kezdeti eset helyett az $n = 2$, ill. $n = 3$ kezdeti esetekről indítjuk az indukciós bizonyítást. Azért vehetjük az indukciós bizonyításban $T(2)$-t, ill. $T(3)$-at kezdeti esetnek, mert $n > 3$ esetén a rekurzív képlet nem függ közvetlenül $T(1)$-től. A bizonyítást úgy folytatjuk, hogy most az indukciós bizonyítás kezdeti esete ($n = 2$ és $n = 3$) különbözik a rekurziós képlet kezdeti feltételétől ($n = 1$). A rekurzióból levezethetjük, hogy $T(2) = 4$ és $T(3) = 5$. Indukciós bizonyításunkat – melyben azt kívánjuk igazolni, hogy $T(n) \leq cn\lg n$ valamely $c \geq 1$ állandóra – most már be tudjuk fejezni, meg tudjuk választani a c állandót úgy, hogy a $T(2) \leq c \cdot 2\lg 2$ és a $T(3) \leq c \cdot 3\lg 3$ legyen. Most bármely $c \geq 2$ választás megfelelő. A legtöbb általunk vizsgálandó rekurzív egyenlet esetében, mindjárt úgy terjesztjük ki a kezdeti feltételeket, hogy az indukciós feltevés kis n-ekre is igaz legyen.

A jó eredmény megsejtése

Sajnos, nincs általános szabály a pontos végeredmény megsejtésére. A jó sejtés kitalálásához gyakorlatra, és néha találékonyságra van szükség. Szerencsére azonban van néhány heurisztikus módszer, amely segít kitalálni a jó sejtést. Jó sejtéseket kaphatunk a rekurziós fa módszer (4.2. alfejezet) használatával is.

Ha egy rekurzív képlet hasonló egy olyanhoz, amit már korábban is láttunk, akkor ésszerű hasonló megoldásra gyanakodni. Példaként tekintsük a

$$T(n) = 2T\left(\left\lfloor\frac{n}{2}\right\rfloor + 17\right) + n$$

rekurzív képletet, amely bonyolultnak tűnik, mert a jobb oldalon T argumentumához 17-et hozzáadtunk. Úgy érezzük azonban, hogy ez a tag nem befolyásolhatja alapvetően a rekurzív egyenlet megoldását. Ha n értéke nagy, akkor nincs nagy különbség $T(\lfloor n/2 \rfloor)$ és $T(\lfloor n/2 \rfloor + 17)$ között: mindkettő nagyjából felezi n-et. Következésképpen sejtésünk az, hogy $T(n) = O(n\lg n)$, amit a helyettesítő módszerrel igazolhatunk is (lásd a 4.1-5. gyakorlatot).

Egy másik módszer a helyes eredmény megsejtésére az, hogy megkeressük a rekurzió egy laza alsó és felső korlátját, és azt élesítjük. Például, a (4.4) rekurzív képlet esetében kiindulhatunk a $T(n) = \Omega(n)$ alsó korlátból, mivel az n tag szerepel a képletben, és választhatjuk $T(n) = O(n^2)$-et kezdeti felső korlátnak. Ezután fokozatosan csökkenthetjük a felső, és növelhetjük az alsó korlátot, amíg az eredmény rá nem húzódik a $T(n) = \Theta(n \lg n)$ aszimptotikusan éles megoldásra.

Finomságok

Néha ugyan helyesen sejtjük meg a rekurzív egyenlet megoldásának aszimptotikus korlátját, de az indukció mégsem akar működni. Általában az a probléma, hogy az indukciós feltevés nem elég erős ahhoz, hogy bizonyítsuk a pontos korlátot. Ha ilyen váratlan akadállyal találjuk szembe magunkat, akkor célszerű a sejtést felülvizsgálni, és kivonni bel őle egy alacsonyabb rendű tagot. Ez általában lehet ővé teszi, hogy megkapjuk a helyes eredményt.

Tekintsük a következő rekurzív egyenletet:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

Azt sejtjük, hogy $O(n)$ a megoldás, és megpróbáljuk bebizonyítani, hogy $T(n) \leq cn$, ha megfelelően választjuk meg a c állandót. Sejtésünket a rekurzív képletbe helyettesítve azt kapjuk, hogy

$$\begin{aligned} T(n) &\leq c\left\lceil \frac{n}{2} \right\rceil + c\left\lfloor \frac{n}{2} \right\rfloor + 1 \\ &= cn + 1, \end{aligned}$$

amiből nem következik, hogy $T(n) \leq cn$ bármely c választással. Csábító, hogy inkább egy nagyobb, például a $T(n) = O(n^2)$ feltételezéssel éljünk, ami lehetne jó, de most ki fog derülni, hogy a $T(n) = O(n)$ sejtés helyes. Ahhoz azonban, hogy ezt bebizonyítsuk, er ősebb indukciós feltevésre van szükségünk.

Szemléletesen, sejtésünk majdnem jó, csak az alacsonyabb rendű 1 állandóban különbözik. Mindazonáltal a teljes indukció csak akkor működik, ha az indukciós feltevést pontosan bizonyítjuk. Úgy győzzük le ezt a nehézséget, hogy *kivonunk* egy alacsonyabb rendű tagot az eredeti sejtésünkből. Így az új feltevésünk az, hogy $T(n) \leq cn - b$, ahol $b \geq 0$ állandó. Ekkor azt kapjuk, hogy

$$\begin{aligned} T(n) &\leq \left(c\left\lfloor \frac{n}{2} \right\rfloor - b\right) + \left(c\left\lceil \frac{n}{2} \right\rceil - b\right) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b, \end{aligned}$$

ha $b \geq 1$. Akárcsak az előbb, a c állandót olyan nagynak kell választanunk, hogy teljesüljenek a kezdeti feltételek.

Legtöbben az alacsonyabb rendű tag kivonását a józan ésszel ellentétesnek érzik. Végül is nem növelni kellene a sejtésünket, ha nem működik az eredeti? Ennél a lépésnél a megértés kulcsa az, hogy emlékezetünkbe idézzük, teljes indukciót használunk: úgy tudunk erősebb állítást bebizonyítani egy adott értékre, ha a kisebb értékekre vonatkozó feltevésünk is erősebb.

Buktatók elkerülése

Aszimptotikus jelölések használatánál könnyű hibázni. Például, a (4.4) rekurzív képlet esetében a $T(n) \leq cn$ sejtésből arra a téves következtetésre juthatunk, hogy $T(n) = O(n)$, ha a következő gondolatmenetet követjük:

$$\begin{aligned} T(n) &\leq 2\left(c\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq cn + n \\ &= O(n), \qquad \Longleftarrow \textit{hiba!!} \end{aligned}$$

mivel c konstans. A hiba az, hogy ezzel nem bizonyítottuk az indukciós feltevés pontos alakját, hogy ti. $T(n) \leq cn$.

Új változó bevezetése

Néha elég egy kis algebrai átalakítás, hogy egy ismeretlen rekurzív képletet hasonlóvá tegyünk egy korábbihoz. Példaként tekintsük a bonyolultnak tűn ő

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

rekurzív egyenletet. Új változó bevezetésével egyszerűsíthetjük a rekurzív egyenletet. Az egyszerűség kedvéért most nem foglalkozunk a $\sqrt{n}$ típusú értékek egészekre kerekítésével. Az $m = \lg n$ bevezetésével kapjuk, hogy

$$T(2^m) = 2T(2^{m/2}) + m.$$

Ezután bevezethetjük az $S(m) = T(2^m)$ változót, és így a (4.4) rekurzív egyenlethez nagyon hasonló

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

egyenletet kapjuk, melynek ugyanaz a megoldása: $S(m) = O(m \lg m)$. Visszahelyettesítve $T(n)$-re azt kapjuk, hogy $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Gyakorlatok

4.1-1. Mutassuk meg, hogy $T(n) = T(\lceil n/2 \rceil) + 1$ megoldása $O(\lg n)$ nagyságrendű.
4.1-2. Láttuk, hogy a $T(n) = T(\lfloor n/2 \rfloor) + n$ megoldása $O(n \log n)$ nagyságrendű. Mutassuk meg azt is, hogy a megoldás $\Omega(n \lg n)$ nagyságrendű, és igazoljuk a $T(n) = \Theta(n \lg n)$ állítást.
4.1-3. Mutassuk meg, hogy a (4.4) rekurzív egyenlet esetén, egy másik indukciós feltevéssel áthidalhatjuk a $T(1) = 1$ kezdeti feltételből adódó nehézséget, anélkül, hogy a teljes indukció kezdeti esetét módosítanánk.
4.1-4. Mutassuk meg, hogy az összefésülő rendezésre vonatkozó (4.2) rekurzív egyenlet *pontos* alakjának a megoldása $\Theta(n \lg n)$ nagyságrendű.
4.1-5. Mutassuk meg, hogy $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ megoldása $O(n \lg n)$ nagyságrendű.
4.1-6. Új változó bevezetésével oldjuk meg aszimptotikusan pontosan a $T(n) = 2T(\sqrt{n}) + 1$ rekurzív egyenletet. Nem fontos, hogy az értékek egészek legyenek.

4.2. A rekurziós fa módszer

A helyettesítő módszerrel tömören be lehet bizonyítani, hogy a rekurziós képletre adott megoldásunk helyes, de néha nagyon nehéz megsejteni a jó megoldást. A rekurziós fa felrajzolása viszont segíthet a helyes megoldás megsejtésében. A 2.3.2. pontban is a rekurziós fáról olvastuk le az összefésülő rendezés futási idejét. A ***rekurziós fa*** minden egyes csúcsa egy részfeladatnak felel meg: a függvény kiértékelésekor végrehajtódó minden rekurziós híváshoz tartozik egy csúcs. Szintenként összegezzük a csúcsok költségét, majd az így kapott szintenkénti költségeket összegezzük, hogy megkapjuk a teljes költséget. Oszd-meg-és-uralkodj elvet használó algoritmusok elemzésénél a rekurziós fa nagyon kényelmes eszköznek bizonyul.

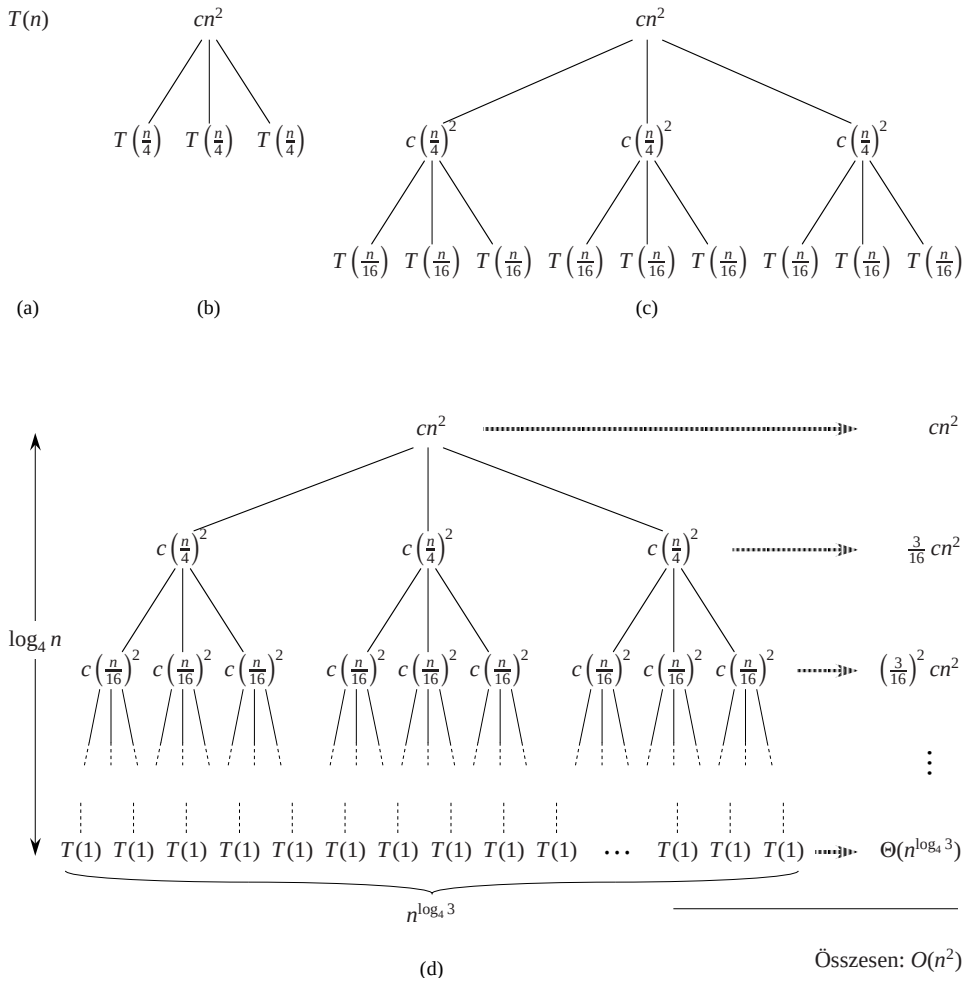
A rekurziós fa módszert leginkább egy jó sejtés megtalálására érdemes használni, amit aztán a helyettesítő módszerrel ellenőrzünk. Ha a rekurziós fa csak arra kell, hogy egy jó sejtést kapjunk, akkor elegendő a számolást „nagyvonalúan” végezni, hiszen a megoldást úgyis precízen ellenőrizzük a helyettesítő módszerrel. De ha a fa felrajzolását és az összegzéseket gondosan végezzük, akkor a rekurziós fa módszer közvetlenül is bizonyítja a kapott megoldás helyességét. Ebben az alfejezetben azt mutatjuk meg, hogy hogyan lehet a rekurziós fákból jó sejtéseket kapni, míg a 4.4. alfejezetben a rekurziós fa módszerrel közvetlenül bizonyítjuk a mester módszer alapjául szolgáló tételt.

Példaként sejtsük meg a megoldást a $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ rekurzív egyenletre a rekurziós fa módszerrel. Kezdetben próbáljunk egy felső korlátot kapni a megoldásra. Mivel tudjuk, hogy az egészrész nem változtatja meg jelentősen a megoldást (most, mint ahogy említettük, számolhatunk nagyvonalúan), a rekurziós fát a $T(n) = 3T(n/4) + cn^2$ egyenletre építjük fel, ahol $c > 0$ a Θ jelölés definíciójából származó konstans.

A $T(n) = 3T(n/4) + cn^2$ egyenlethez tartozó rekurziós fát a 4.1. ábra mutatja. Az egyszerűség kedvéért feltehetjük, hogy n a négynek egy egész kitevős hatványa (megint nagyvonalúan számolunk, de ez nem fogja befolyásolni a végeredményt). Az ábra (a) része magát a $T(n)$ függvényt ábrázolja, a rekurzió behelyettesítésével kapjuk az ábra (b) részét. A gyökérben lévő cn^2 tag a rekurzió csúcsán fellépő költségnek felel meg, míg a gyökér három részfája a három $n/4$ méretű részprobléma költségét jelképezi. Az ábra (c) részében tovább folytatjuk a fa építését: a $T(n/4)$ költségű csúcsokba is behelyettesítjük a rekurziós egyenletet. A gyökér három gyerekének a költsége egyenként $c(n/2)^2$. A fa építését folytatjuk tovább, minden csúcsot tovább bontunk a rekurziós képlet alapján.

Mivel a részproblémák mérete egyre csökken, ahogy egyre távolabb kerülünk a gyökértől, előbb-utóbb a részproblémák mérete olyan kicsi lesz, hogy rájuk nem a rekurziós képlet vonatkozik, hanem a kezdeti feltétel. Milyen messze leszünk ekkor a gyökértől? Az i-edik szinten lévő részproblémák mérete $n/4^i$. Vagyis a részproblémák mérete akkor lesz az $n = 1$ kezdeti feltétel által meghatározott, ha $n/4^i = 1$ vagy ezzel ekvivalens módon, ha $i = \log_4 n$. Így tehát a fának $\log_4 n + 1$ szintje lesz, nullától egészen $\log_4 n$-ig.

Most meghatározzuk a fa mindegyik szintjének a költségét. Minden szinten háromszor annyi csúcs van, mint egy szinttel feljebb, így az i-edik szinten 3^i csúcs van. Mivel a részproblémák mérete negyedére csökken minden szinten, ezért az i-edik szinten, $i = 0, 1, 2, \ldots, \log_4 n - 1$ esetén, minden csúcs költsége $c(n/4^i)^2$. Összeszorozva azt kapjuk, hogy az i-edik szinten lévő csúcsok összköltsége, $i = 0, 1, 2, \ldots, \log_4 n - 1$ esetén, pontosan $3^i c(n/4^i)^2 = (3/16)^i cn^2$. Az utolsó $\log_4 n$-edik szinten $3^{\log_4 n} = n^{\log_4 3}$ csúcs van, mindegyiknek a költsége $T(1)$, így az utolsó szint teljes költsége $n^{\log_4 3} T(1)$, ami $\Theta(n^{\log_4 3})$.



4.1. ábra. A $T(n) = 3T(n/4) + cn^2$ rekurzív egyenlet rekurziós fájának előállítása. Az **(a)** rész $T(n)$-et mutatja, amit a **(b)–(d)** részben fokozatosan rekurziós fává bővítünk. A **(d)** részben a teljesen kibontott fa magassága $\log_4 n$ (mivel $\log_4 n + 1$ szintje van).

Ezután összegezzük a szintek költségét, hogy megkapjuk a teljes fa költségét:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\frac{3}{16} - 1} cn^2 + \Theta(n^{\log_4 3}). \end{aligned}$$

Az utolsó képlet elég bonyolultnak tűnik, de az első tag – újabb nagyvonalúsággal – felülről

becsülhető egy végtelen (csökkenő) mértani sor összegével. Egy egyenlőséggel visszalépve, és az (A.6) egyenlőséget alkalmazva kapjuk, hogy

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-\frac{3}{16}} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2). \end{aligned}$$

Vagyis a $T(n) = O(n^2)$ sejtés adódik a $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ rekurzív képletre. Ebben a példában a cn^2 együtthatói csökkenő mértani sorozatot alkotnak, így (A.6) alapján ezen együtthatók összegét felülről becsülhetjük a $16/13$ konstanssal. Mivel a gyökér költsége cn^2, ezért a gyökér költsége konstans hányadát adja a teljes költségnek. Vagyis a teljes költség legmeghatározóbb része a gyökér költsége.

Ha $O(n^2)$ valóban felső korlát a rekurzióra (ezt mindjárt ellenőrizni fogjuk), akkor valójában éles felső korlát is. Ez azért igaz, mert az első rekurzív hívás költsége már rögtön $\Theta(n^2)$, így tehát $\Omega(n^2)$ alsó korlát a teljes költségre.

A helyettesítő módszerrel ellenőrizhetjük, hogy jó volt a sejtésünk, vagyis a $T(n) = O(n^2)$ felső becslés tényleg igaz a $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ rekurzív képletre. Meg kell mutatnunk, hogy $T(n) \leq dn^2$ teljesül valamilyen $d > 0$ konstansra. Legyen c ugyanaz a konstans, mint a korábbiakban, ekkor

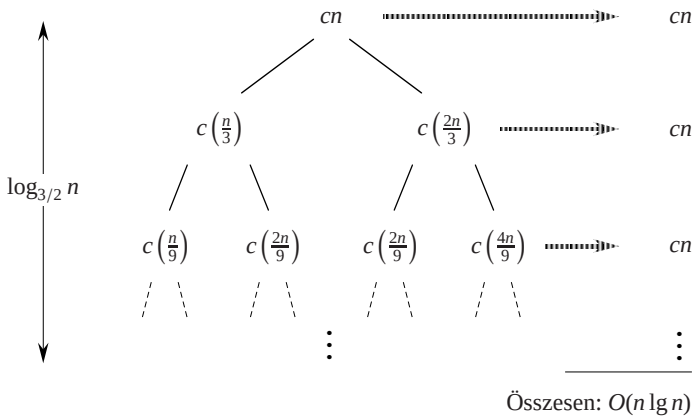
$$\begin{aligned} T(n) &\leq 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + cn^2 \\ &\leq 3d\left\lfloor \frac{n}{4} \right\rfloor^2 + cn^2 \\ &\leq 3d\left(\frac{n}{4}\right)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\leq dn^2, \end{aligned}$$

ahol az utolsó egyenlőtlenség $d \geq (16/13)c$ választás esetén teljesül.

A 4.2. ábra egy másik, bonyolultabb példa, a

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n),$$

rekurziós fáját mutatja. (Az egyszerűség kedvéért ismét elhagyjuk az alsó és felső egészrész függvényeket.) A korábbiakhoz hasonlóan, legyen c az O jelöléshez tartozó konstans. Ha összeadjuk az egyes szinteken lévő értékeket, akkor minden szinten cn-et kapunk. Az $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$ a leghosszabb út a gyökértől egy levélig. Mivel $(2/3)^k n = 1$, ha $k = \log_{3/2} n$, ezért a fa magassága $\log_{3/2} n$.



4.2. ábra. A $T(n) = T(n/3) + T(2n/3) + cn$ rekurzív egyenlethez tartozó rekurziós fa.

Azt várjuk, hogy a rekurzív egyenlet megoldása legfeljebb a szintek száma szorozva a szintek egyenkénti költségével, vagyis $O(cn \log_{3/2} n) = O(n \lg n)$. A teljes költség egyenletesen van elosztva a szintek között. Van viszont egy kis komplikáció: figyelembe kell még vennünk a levelek költségét is. Ha ez a rekurziós fa egy $\log_{3/2} n$ szintű teljes bináris fa lenne, akkor $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ levele lenne. Mivel egy levél költsége konstans, ezért ez azt jelentené, hogy a levelek teljes költsége $\Theta(n^{\log_{3/2} 2})$ lenne, ami $\omega(n \lg n)$. Ez a rekurziós fa viszont nem egy teljes fa, így kevesebb mint $n^{\log_{3/2} 2}$ levele van. Továbbá, ahogy megyünk egyre lejjebb, egyre több belső csúcs hiányzik. Vagyis nem mindegyik szint költsége pontosan cn, a lejjebb lévő szintek költsége kisebb. Pontosan kiszámolhatnánk a teljes költséget, de ne felejtsük el, hogy nekünk csak egy jó sejtésre van szükségünk, amit aztán a helyettesítő módszerrel fogunk ellenőrizni. Ezért nagyvonalúan hanyagoljuk el ezeket a kérdéseket, és próbáljuk meg bebizonyítani, hogy az $O(n \lg n)$ sejtés tényleg helyes.

A helyettesítő módszer tényleg igazolja, hogy az $O(n \lg n)$ tényleg felső korlát a rekurzív képlet megoldására. Megmutatjuk, hogy $T(n) \leq dn \lg n$, ahol d egy alkalmasan választott konstans:

$$\begin{aligned}
T(n) &\leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \\
&\leq d\frac{n}{3}\lg\left(\frac{n}{3}\right) + d\frac{2n}{3}\lg\left(\frac{2n}{3}\right) + cn \\
&= \left(d\frac{n}{3}\lg n - d\frac{n}{3}\lg 3\right) + \left(d\frac{2n}{3}\lg n - d\frac{2n}{3}\lg\left(\frac{3}{2}\right)\right) + cn \\
&= dn\lg n - d\left(\frac{n}{3}\lg 3 + \frac{2n}{3}\lg\left(\frac{3}{2}\right)\right) + cn \\
&= dn\lg n - d\left(\frac{n}{3}\lg 3 + \frac{2n}{3}\lg 3 - \frac{2n}{3}\lg 2\right) + cn \\
&= dn\lg n - dn\left(\lg 3 - \frac{2}{3}\right) + cn \\
&\leq dn\lg n,
\end{aligned}$$

ha $d \geq c/(\lg 3 - \frac{2}{3})$) teljesül. Vagyis a sejtés jónak bizonyult, annak ellenére, hogy a rekurziós fában csak nagyvonalúan határoztuk meg a költségeket.

Gyakorlatok

4.2-1. Rekurziós fa segítségével határozzuk meg a $T(n) = 3T(\lfloor n/2 \rfloor) + n$ rekurzív egyenlet megoldásának egy jó aszimptotikus felső korlátját. Ellenőrizzük válaszunkat a helyettesítő módszerrel.
4.2-2. Rekurziós fa segítségével igazoljuk, hogy a $T(n) = T(n/3) + T(2n/3) + cn$ rekurzív egyenlet megoldása $\Omega(n \lg n)$. Ellenőrizzük válaszunkat a helyettesítő módszerrel.
4.2-3. Rajzoljuk fel a $T(n) = 4T(\lfloor n/2 \rfloor) + cn$ rekurzív egyenlet rekurziós fáját, és adjunk éles aszimptotikus korlátot a megoldásra. A helyettesítő módszerrel ellenőrizzük a megoldást.
4.2-4. Rekurziós fa segítségével adjunk éles aszimptotikus korlátot a $T(n) = T(n - a) + T(a) + cn$ rekurzív egyenlet megoldására, ahol $a \geq 1$ és $c > 0$ konstans.
4.2-5. Rekurziós fa segítségével adjunk éles aszimptotikus korlátot a $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$ rekurzív egyenlet megoldására, ahol az α és c állandókra teljesül, hogy $0 < \alpha < 1$ és $c > 0$.

4.3. A mester módszer

A mester módszer receptet ad a

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{4.5}$$

alakú rekurzív egyenletek megoldására, ahol $a \geq 1$ és $b > 1$ állandók, és az $f(n)$ aszimptotikusan pozitív függvény. A mester módszer alkalmazásához három esetet kell megjegyezni, ezután sok rekurzív egyenlet megoldása könnyen, sok esetben papír és ceruza nélkül, meghatározható.

A (4.5) rekurzív egyenlet egy olyan algoritmus futási idejét írja le, amely az n méretű feladatot, a – egyenként n/b méretű – alfeladatra bontja, ahol a és b pozitív állandók. Az a számú alfeladatot – egyenként $T(n/b)$ idő alatt – rekurzívan oldjuk meg. A feladat felbontásának és az alfeladatok eredményének egyesítési költségét az $f(n)$ függvény írja le. (A 2.3.2. alpont jelölése szerint $f(n) = D(n) + C(n)$.) Például, az Összefésülő-rendezés-ből származó rekurzív egyenletnél $a = 2$, $b = 2$ és $f(n) = \Theta(n)$.

Ami a technikai korrektséget illeti, a rekurzív egyenlet nincs pontosan definiálva, mivel előfordulhat, hogy n/b nem egész. Azonban, ha mind az a darab $T(n/b)$ tagot $T(\lfloor n/b \rfloor)$-re vagy $T(\lceil n/b \rceil)$-re cseréljük, ez nem változtatja meg a rekurzív egyenlet megoldásának aszimptotikus viselkedését. (Ezt a következő fejezetben fogjuk bizonyítani.) Így általában kényelmesnek találjuk, hogy elhagyjuk az alsó és felső egészrész függvényeket, ha oszd-meg-és-uralkodj elvű algoritmusokat írunk ilyen formában.

Mester tétel

A mester módszer az alábbi tételen alapszik.

4.1. tétel (mester tétel). *Legyenek $a \geq 1$, $b > 1$ állandók, $f(n)$ egy függvény, $T(n)$ pedig a nemnegatív egészeken a*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

rekurzív egyenlettel definiált függvény, ahol n/b jelentheti akár az $\lfloor n/b \rfloor$ egészrészt, akár az $\lceil n/b \rceil$ egészrészt. Ekkor $T(n)$-re a következő aszimptotikus korlátok adhatók meg.

1. *Ha $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ valamely $\epsilon > 0$ állandóra, akkor $T(n) = \Theta(n^{\log_b a})$.*
2. *Ha $f(n) = \Theta\left(n^{\log_b a}\right)$, akkor $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.*
3. *Ha $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ valamely $\epsilon > 0$ állandóra, és $af(n/b) \leq cf(n)$ valamely $c < 1$ állandóra és elég nagy n-re, akkor $T(n) = \Theta(f(n))$.*

Mielőtt néhány feladatra alkalmaznánk a mester tételt, szánjunk rá egy kis időt, hogy megértsük, miről is szól. Mindhárom esetben az $f(n)$ és az $n^{\log_b a}$ függvényt hasonlítjuk össze. Szemléletesen: a rekurzív egyenlet megoldását a nagyobb függvény határozza meg. Ha, mint az 1. esetben, az $n^{\log_b a}$ függvény a nagyobb, akkor $T(n) = \Theta(n^{\log_b a})$ a megoldás. Ha, mint a 3. esetben, az $f(n)$ függvény a nagyobb, akkor $T(n) = \Theta(f(n))$ a megoldás. Ha, mint a 2. esetben, a két függvény azonos nagyságrendű, akkor egy logaritmikus tényez ővel szorzunk, és a $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$ lesz a megoldás.

A szemléletes jelentés mögött vannak bizonyos technikai részletek, amelyeket fontos megérteni. Az első esetben az $f(n)$-nek nemcsak kisebbnek, hanem *polinomiálisan* kisebbnek kell lennie, mint $n^{\log_b a}$. Azaz $f(n)$-nek egy n^{ϵ} szorzótényezővel aszimptotikusan kisebbnek kell lennie, ahol $\epsilon > 0$ állandó. A harmadik esetben $f(n)$-nek nemcsak nagyobbnak, de polinomiálisan nagyobbnak kell lennie, mint $n^{\log_a b}$, ráadásul ki kell elégítenie azt a *regularitási* feltételt, hogy $af(n/b) \leq cf(n)$. Ezt a feltételt a legtöbb – eddig el őforduló – polinomiálisan korlátos függvény kielégíti.

Fontos, hogy észrevegyük, a három eset nem fedi le az $f(n)$-re vonatkozó összes lehetőséget. Az 1. és a 2. eset között van egy lyuk, mikor $f(n)$ ugyan kisebb, mint $n^{\log_a b}$, de nem polinomiálisan kisebb. Hasonlóan, a 2. és 3. eset között is lyuk van, mikor $f(n)$ ugyan nagyobb, mint $n^{\log_a b}$, de nem polinomiálisan nagyobb. Ha az $f(n)$ függvény ezekbe a hézagokba esik, vagy a 3. esetben a regularitási feltétel nem teljesül, akkor nem lehet a mester módszerrel megoldani a rekurziót.

A mester módszer használata

A mester módszer használatánál egyszerűen meghatározzuk, hogy a mester tétel melyik esetéről van szó (ha egyáltalán szó van valamelyikről), és ezzel már meg is van a válasz.

Első példaként tekintsük a

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

egyenletet. Ehhez a rekurzív egyenlethez $a = 9$, $b = 3$, $f(n) = n$ tartozik, és így $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Mivel $f(n) = O(n^{\log_3(9-\epsilon)})$, ahol $\epsilon = 1$, ezért a mester tétel 1. esetét alkalmazzuk, és a $T(n) = \Theta(n^2)$ megoldásra jutunk.

Most tekintsük a

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

egyenletet, ahol $a = 1$, $b = 3/2$, $f(n) = 1$ és $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Itt a 2. eset érvényes, mivel $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, és így a megoldás a $T(n) = \Theta(\lg n)$.

A

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

rekurzív egyenlet esetében $a = 3$, $b = 4$, $f(n) = n \lg n$, és $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$. Mivel $f(n) = \Omega\left(n^{\log_4 3+\epsilon}\right)$, ahol $\epsilon \approx 0{,}2$, ezért a 3. esetet alkalmazhatjuk, ha meg tudjuk mutatni, hogy $f(n)$-re érvényes a regularitási feltétel. Elég nagy n-re, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$, ha $c = 3/4$. Következésképpen, a 3. eset alapján, a rekurzív egyenlet megoldása $T(n) = \Theta(n \lg n)$.

A mester tétel nem alkalmazható a

$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$$

rekurzív egyenletre, annak ellenére, hogy az megfelelő alakú: $a = 2$, $b = 2$, $f(n) = n \lg n$, és $n^{\log_b a} = n$. Úgy néz ki, mintha a 3. esetet kellene alkalmazni, mivel az $f(n) = n \lg n$ a szimptotikusan nagyobb, mint $n^{\log_b a} = n$, de nem *polinomiálisan* nagyobb. Az $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ hányados aszimptotikusan kisebb, mint n^{ϵ}, bármely ϵ pozitív állandó esetén. Következésképpen, ez a rekurzív egyenlet éppen a 2. és a 3. eset közé esik. (A megoldást lásd a 4.4-2. gyakorlatnál.)

Gyakorlatok

4.3-1. A mester módszerrel adjunk éles aszimptotikus korlátot az alábbi rekurzív problémákra:

a. $T(n) = 4T(n/2) + n$.

b. $T(n) = 4T(n/2) + n^2$.

c. $T(n) = 4T(n/2) + n^3$.

4.3-2. Az A algoritmus futási idejét a $T(n) = 7T(n/2) + n^2$ rekurzív egyenlet írja le. Egy másik A' algoritmus futási ideje $T'(n) = aT'(n/4) + n^2$. Melyik az a legnagyobb a érték, amelyre A' aszimptotikusan gyorsabb, mint A?

4.3-3. Bizonyítsuk be a mester módszer segítségével, hogy a bináris keresés (lásd a 2.3-5. gyakorlatot) $T(n) = T(n/2) + \Theta(1)$ rekurzív egyenletének megoldása $T(n) = \Theta(\lg n)$.

4.3-4. Lehet-e a mester módszert alkalmazni a $T(n) = 4T(n/2) + n^2 \log n$ rekurzív egyenletre? Adjunk aszimptotikus felső korlátot a rekurzióra.

4.3-5.$\star$ Tekintsük a mester tétel 3. esetéhez tartozó $af(n/b) \leq cf(n)$ ún. regularitási feltételt valamely $c < 1$ állandóval. Adjunk meg olyan egyszerű $f(n)$ függvényt, valamint olyan $a \geq 1$ és $b > 1$ konstansokat, hogy a regularitási feltétel kivételével a mester tétel 3. esetének valamennyi feltétele teljesüljön.

★ 4.4. A mester tétel bizonyítása

Ez az alfejezet a mester tétel (4.1. tétel) egy bizonyítását tartalmazza haladók részére. A bizonyítás megértésére nincs szükség a tétel alkalmazásához.

A bizonyítás két részből áll. Az első rész a (4.5) „mester” egyenletet elemzi azzal az egyszerűsítő feltételezéssel, hogy $T(n)$ csak $b > 1$ egész kitevős hatványain ($n = 1, b, b^2, \ldots$) van értelmezve. Ez a rész a szükséges összes szemléletes lépést tartalmazza, ami a mester tétel igaz voltának megértéséhez kell. A második rész azt mutatja meg, hogyan lehet minden egész n-re kiterjeszteni az elemzést, itt már csak az alsó és felső egészrészek kezelésére szolgáló technikai eszközöket kell kidolgoznunk.

Ebben az alfejezetben aszimptotikus jelöléseinket kissé pontatlanul fogjuk használni olyan függvények leírására, amelyek csak b egész kitevős hatványain vannak értelmezve. Emlékezzünk vissza, hogy az aszimptotikus jelölések definíciója megkívánja, hogy a korlátokat minden elég nagy számra bizonyítsuk (nem csak b hatványaira). Mivel be lehetne vezetni olyan aszimptotikus jelöléseket is, amelyek a nemnegatív egészek helyett csak a $\{b^i : i = 0, 1, \ldots\}$ halmazra alkalmazhatók, ezért ez csak apró pontatlanság.

Mindazonáltal mindig elővigyázatosnak kell lennünk, ha csak egy leszűkített értelmezési tartományra használjuk az aszimptotikus jelöléseket, nehogy téves következtetésre jussunk. Például, ha $T(n) = O(n)$ teljesül arra az esetre, ha n kettőhatvány, ez nem bizonyítja, hogy $T(n) = O(n)$. A $T(n)$ függvényt definiálhatjuk úgy, hogy

$$T(n) = \begin{cases} n, & \text{ha } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{egyébként,} \end{cases}$$

és ekkor a megadható legjobb felső korlát a $T(n) = O(n^2)$. Az ilyen típusú drasztikus következmények miatt soha nem fogunk aszimptotikus jelöléseket leszűkített értelmezési tartományon használni anélkül, hogy azt külön ki ne emelnénk.

4.4.1. Bizonyítás egész kitevős hatványokra

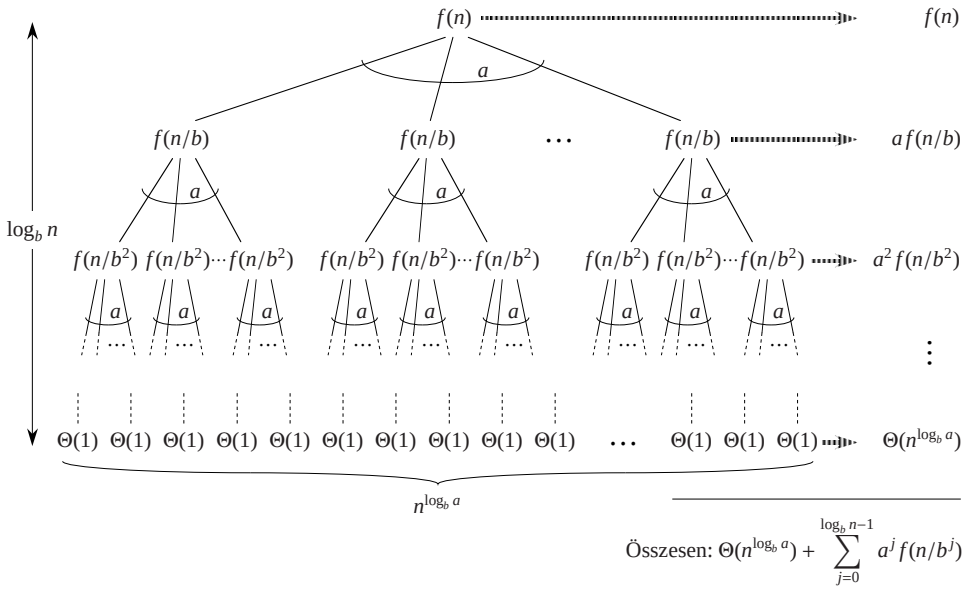
A mester tétel bizonyításának első része a (4.5)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

mester egyenlet elemzése, azzal a feltételezéssel, hogy n egész kitevős hatványa b-nek, de $b > 1$ nem szükségképpen egész. Az elemzés három lemmából tevődik össze. Az első a mester egyenlet megoldását egy összegzést tartalmazó kifejezés kiszámítására vezeti vissza. A második erre az összegzésre ad meg korlátokat. A harmadik lemma egyesíti az első kettőt, és ezzel bizonyítja a tétel azon változatát, ahol n egész kitevős hatványa b-nek.

4.2. lemma. *Legyenek $a \geq 1$ és $b > 1$ pozitív állandók, $f(n)$ pedig legyen b egész kitevős hatványain értelmezett nemnegatív függvény. A $T(n)$ függvényt definiáljuk b egész kitevős hatványain a következő rekurzív egyenlettel:*

$$T(n) = \begin{cases} \Theta(1), & \text{ha } n = 1, \\ aT(n/b) + f(n), & \text{ha } n = b^i, \end{cases}$$



4.3. ábra. A $T(n) = aT(n/b) + f(n)$ rekurzív egyenlet rekurziós fája. A fa egy $\log_b n$ magasságú a-ad rendű teljes fa, $n^{\log_b a}$ levéllel. Az ábra jobb oldalán az egyes szintek költsége van feltüntetve, a teljes költséget a (4.6) egyenlet adja.

ahol i pozitív egész. Ekkor

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right). \tag{4.6}$$

Bizonyítás. A 4.3. ábrán látható rekurziós fát használjuk a bizonyítás során. A fa gyökerének $f(n)$ a költsége és a gyereke van, egyenként $f(n/b)$ költséggel. (Ha a rekurziós fát vizsgáljuk, akkor kényelmes úgy tekinteni, hogy a egész, de a számolások során ezt nem használjuk ki.) A gyerekek mindegyikének van egyenként a gyereke $f(n/b^2)$ költséggel, így pontosan a^2 csúcs van 2 távolságra a gyökértől. Általánosan, pontosan a^j csúcs van j távolságra a gyökértől, és mindegyiknek $f(n/b^j)$ a költsége. A leveleknek egyenként $T(1) = \Theta(1)$ a költsége, és mindegyik levél a $\log_b n$ szinten található (mivel $n/b^{\log_b n} = 1$). A fának összesen $a^{\log_b n} = n^{\log_b a}$ levele van.

Ha az ábrán látható módon összegezzük a szintek költségét, akkor pont a (4.6) egyenlethez jutunk. A j-edik szinten lévő belső csúcsok költsége $a^j f(n/b^j)$, és így a belső csúcsok teljes költsége

$$\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right).$$

A rekurzióhoz tartozó oszd-meg-és-uralkodj elvű algoritmusban a fenti összeg a részproblémákra bontás és a részmegoldások újraegyesítéséből eredő teljes költség. A levelek összes költsége $\Theta(n^{\log_b a})$ nagyságrendű (ennyi darab 1 méretű részproblémánk volt). ■

Rekurziós fával kifejezve, a mester tétel három esete megfelel annak, amikor a fa teljes költségében (1) túlsúlyban van a levelek költsége, (2) egyenletesen oszlik el a költség az egyes szinteken, (3) túlsúlyban van a gyökér költsége.

A (4.6) egyenletben az összegzés a vizsgált oszd-meg-és-uralkodj elvű algoritmus szétosztási és az újraegyesítési lépéseinek költségét írja le. A következő lemma aszimptotikus korlátokat ad az összegek növekedésére.

4.3. lemma. *Legyenek $a \geq 1$ és $b > 1$ állandók, $f(n)$ pedig legyen b egész kitevős hatványain értelmezett nemnegatív függvény. A $g(n)$ függvényt definiáljuk b egész kitevős hatványain a következő egyenlettel:*

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right). \tag{4.7}$$

Erre a függvényre b egész kitevői esetén a következő korlát adható

1. *Ha $f(n) = O(n^{\log_b a-\epsilon})$ valamely $\epsilon > 0$ állandóra, akkor $g(n) = O(n^{\log_b a})$.*
2. *Ha $f(n) = \Theta(n^{\log_b a})$, akkor $g(n) = \Theta(n^{\log_b a} \lg n)$.*
3. *Ha $af(n/b) \leq cf(n)$ valamely $c < 1$ állandóra és minden $n \geq b$ esetén, akkor $g(n) = \Theta(f(n))$.*

Bizonyítás. Az 1. esetben $f(n) = O(n^{\log_b a-\epsilon})$. Ebből következik, hogy $f(n/b^j) = O((n/b^j)^{\log_b a-\epsilon})$. A (4.7) egyenletbe helyettesítve azt kapjuk, hogy

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right). \tag{4.8}$$

Az O-jelölésen belüli összegre úgy adunk korlátot, hogy bizonyos tagokat kiemelünk, ill. egyszerűsítünk, így végül egy növekvő mértani sorhoz jutunk:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right).
\end{aligned}$$

Mivel b és ϵ állandó, ezért ez utóbbi kifejezés egyszerűsíthető úgy, hogy $n^{\log_b a-\epsilon} O(n^\epsilon) = O(n^{\log_b a})$. Ezt a kifejezést a (4.8) összegébe helyettesítve kapjuk, hogy

$$g(n) = O(n^{\log_b a}),$$

és ezzel az 1. esetet bebizonyítottuk.

A 2. esetben az $f(n) = \Theta(n^{\log_b a})$ feltétel mellett azt kapjuk, hogy $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. A (4.7) egyenletbe helyettesítve arra jutunk, hogy

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right). \tag{4.9}$$

A Θ-n belüli összegre az 1. esethez hasonlóan adunk korlátot, de ebben az esetben nem kapunk mértani sort. Ehelyett azt vesszük észre, hogy minden tag ugyanaz:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a} \log_b n.\end{aligned}$$

Ezt a kifejezést a (4.9) egyenletben szereplő összegbe helyettesítve kapjuk, hogy

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a} \log_b n) \\ &= \Theta(n^{\log_b a} \lg n),\end{aligned}$$

és ezzel a 2. esetet is bebizonyítottuk.

A 3. esetet hasonlóan igazoljuk. Mivel $f(n)$ előfordul $g(n)$ (4.7) definíciójában, és $g(n)$ minden tagja nemnegatív, arra a következtetésre jutunk, hogy $g(n) = \Omega(f(n))$ teljesül b minden egész kitevős hatványára. Feltettük, hogy valamely $c < 1$ állandó esetén minden $n \geq b$-re fennáll az $af(n/b) \leq cf(n)$ egyenlőtlenség, amiből átrendezve $f(n/b) \leq (c/a)f(n)$ következik. Az egyenlőtlenséget j-szer alkalmazva kapjuk, hogy $f(n/b^j) \leq (c/a)^j f(n)$, illetve ezt átrendezve $a^j f(n/b) \leq c^j f(n)$ adódik. A (4.7) egyenletbe helyettesítve és egyszerűsítve egy mértani sort kapunk, de az 1. esettől eltérően ennek tagjai csökkenőek:

$$\begin{aligned}g(n) &= \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \\ &\leq f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n)\left(\frac{1}{1-c}\right) \\ &= O(f(n)),\end{aligned}$$

mivel c állandó. Így arra a következtetésre jutunk, hogy b egész kitevős hatványaira $g(n) = \Theta(f(n))$. A 3. esetet is bebizonyítottuk, és ezzel a lemmát beláttuk. ■

Most bebizonyíthatjuk a mester tételnek azt a változatát, amelyben n egész kitevős hatványa b-nek.

4.4. lemma. *Legyenek $a \geq 1$ és $b > 1$ pozitív állandók, $f(n)$ pedig legyen b egész kitevős hatványain értelmezett nemnegatív függvény. A $T(n)$ függvényt definiáljuk b egész kitevős hatványain a következő rekurzív képlettel:*

$$T(n) = \begin{cases} \Theta(1), & \text{ha } n = 1, \\ aT(n/b) + f(n), & \text{ha } n = b^i, \end{cases}$$

ahol i pozitív egész. Ekkor $T(n)$ nagyságrendjéről a következőket mondhatjuk b egész kitevős hatványain:

1. *Ha $f(n) = O(n^{\log_b a - \epsilon})$ valamely $\epsilon > 0$ állandóval, akkor $T(n) = \Theta(n^{\log_b a})$.*
2. *Ha $f(n) = \Theta(n^{\log_b a})$, akkor $T(n) = \Theta(n^{\log_b a} \lg n)$.*
3. *Ha $f(n) = \Omega(n^{\log_b a + \epsilon})$ valamely $\epsilon > 0$ állandóval, és $af(n/b) \leq cf(n)$ valamely $c < 1$ állandóra és elég nagy n-re, akkor $T(n) = \Theta(f(n))$.*

Bizonyítás. A 4.2. lemma (4.6) összegének kiszámításához a 4.3. lemma korlátját használjuk. Az 1. esetben

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}), \end{aligned}$$

a 2. esetben

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n). \end{aligned}$$

A 3. esetben

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)), \end{aligned}$$

mivel $f(n) = \Omega(n^{\log_b a + \epsilon})$. ■

4.4.2. Alsó és felső egészrészek

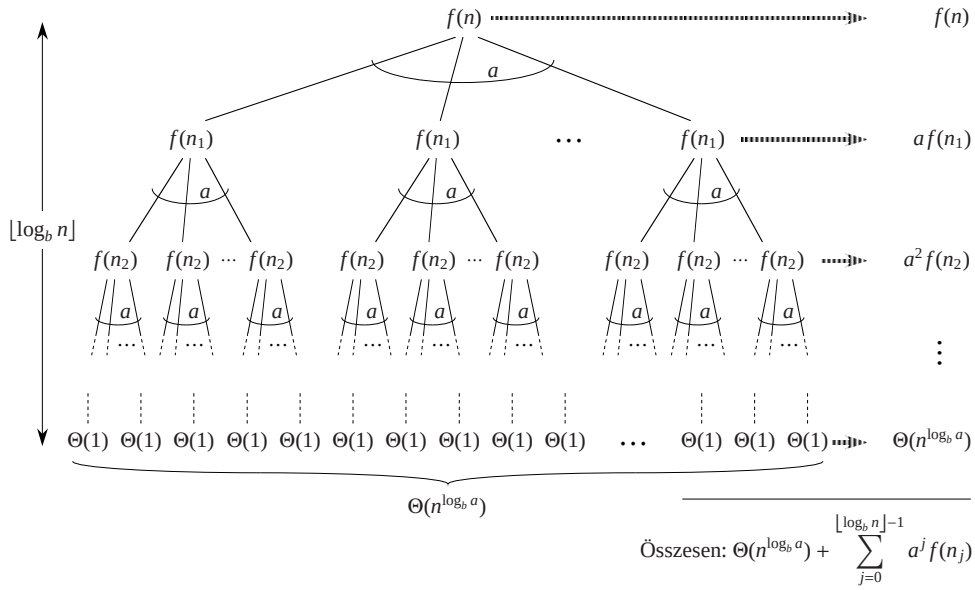
Ahhoz, hogy teljessé tegyük a mester tétel bizonyítását, elemzésünket ki kell terjesztenünk azokra az esetekre is, amikor a rekurzív egyenletben alsó és felső egészrészek is vannak, azért, hogy a rekurzív egyenletet minden egész számra definiáljuk, ne csak azokra, amelyek b-nek egész kitevős hatványai. A szokásos módon tudunk alsó korlátot adni a

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n) \tag{4.10}$$

egyenlet megoldására, és felső korlátot adni a

$$T(n) = aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n) \tag{4.11}$$

egyenlet megoldására, mivel az 1. esetben felhasználva az $\lceil n/b \rceil \geq n/b$ egyenlőtlenséget, megkapjuk a kívánt eredményt, az $\lfloor n/b \rfloor \leq n/b$ egyenlőtlenséget pedig a 2. eset vizsgálatánál használjuk. Mivel alsó korlátot adni a (4.11) egyenlet megoldására szinte ugyanúgy



4.4. ábra. A $T(n) = aT(\lceil n/b \rceil) + f(n)$ rekurziós egyenlethez tartozó rekurziós fa. A rekurzív hívások argumentumában szereplő n_j értéket a (4.12) egyenlet definiálja.

kell, mint felső korlátot adni a (4.10) egyenlet megoldására, ezért csak ez utóbbit fogjuk megmutatni.

Némileg módosítjuk a 4.3. ábra rekurziós fáját, a 4.4. ábra mutatja az új fát. Ahogy ereszkedünk lefelé a rekurziós fán, a rekurziós hívások argumentuma a következ ő sorozatot adja:

$$\begin{aligned} &n, \\ &\lceil n/b \rceil, \\ &\lceil \lceil n/b \rceil / b \rceil, \\ &\lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil, \\ &\quad \vdots \end{aligned}$$

Jelölje n_j a fenti sorozat j-edik elemét, vagyis

$$n_j = \begin{cases} n, & \text{ha } j = 0, \\ \left\lceil (n_{j-1})/b \right\rceil, & \text{ha } j > 0. \end{cases} \tag{4.12}$$

Először is meghatározunk egy olyan k szintet, ahol az n_k értéke már csak egy konstans. Az $\lceil x \rceil \leq x + 1$ egyenlőtlenséget felhasználva adódik, hogy

$$\begin{aligned} n_0 &\leq n, \\ n_1 &\leq \frac{n}{b} + 1, \end{aligned}$$

$$\begin{aligned} n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1, \\ n_3 &\leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1, \\ &\vdots \end{aligned}$$

Általánosan,

$$\begin{aligned} n_j &\leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1}\,. \end{aligned}$$

A $j = \lfloor \log_b n \rfloor$ választással adódik, hogy

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \\ &\leq \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\ &= \frac{n}{\frac{n}{b}} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \\ &= O(1), \end{aligned}$$

vagyis a $\lfloor \log_b n \rfloor$ szinten a részprobléma mérete konstans.

A 4.4. ábrából leolvasható, hogy

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j), \tag{4.13}$$

és ez majdnem ugyanolyan alakú, mint a (4.6) egyenlet, de itt n tetsz őleges egész lehet, és nem szorítkozunk b egész kitev ős hatványaira.

Most a 4.3. lemma bizonyításához hasonlóan a (4.13)-ból kiszámíthatjuk a

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.14}$$

összeget. A 3. esettel kezdve, legyen $af(\lceil n/b \rceil) \leq cf(n)$, ha $n > b + b/(b-1)$, ahol $c < 1$ állandó. Ekkor $a^j f(n_j) \leq c^j f(n)$. Így a (4.14) egyenletben lév ő összeg ugyanúgy számítható ki, mint a 4.3. lemmában. A 2. esetben $f(n) = \Theta(n^{\log_b a})$ teljesül. Ha meg tudjuk mutatni, hogy $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, akkor a 4.3. lemma 2. esetre vonatkozó bizonyítása alkalmazható. Figyeljük meg, ha $j \leq \lfloor \log_b n \rfloor$, akkor $b^j/n \leq 1$. Az $f(n) = O(n^{\log_b a})$ korlát maga után vonja olyan $c > 0$ állandó létezését, hogy elég nagy n_j-re

$$\begin{aligned} f(n_j) &\leq c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\ &= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &\leq c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\ &= O\left(\frac{n^{\log_b a}}{a^j}\right), \end{aligned}$$

mivel $c(1 + b/(b-1))^{\log_b a}$ állandó. Ezzel a 2. esetet bebizonyítottuk. Az 1. eset bizonyítása ehhez hasonló. A bizonyítás kulcsa az, hogy egy bonyolultabb számítást igénylő, de a 2. eset megfelelő bizonyításához hasonló módon igazoljuk az $f(n_j) = O(n^{\log_b a - \epsilon})$ becslést.

Ezzel a mester tételben szereplő felső korlátokat valamennyi egész n-re igazoltuk. Az alsó korlátokra vonatkozó bizonyítás hasonlóan elvégezhető.

Gyakorlatok

4.4-1.⋆ Fejezzük ki egyszerűen és pontosan a (4.12) egyenletben szereplő n_j-t abban az esetben, ha b nem egy tetszőleges valós szám, hanem pozitív egész.

4.4-2.⋆ Mutassuk meg, hogy amennyiben $f(n) = \Theta(n^{\log_b a} \lg^k n)$ és $k \geq 0$, akkor a mester egyenlet megoldása $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Az egyszerűség kedvéért szorítkozzunk b egész kitevős hatványaira.

4.4-3.⋆ Mutassuk meg, hogy a mester tétel 3. esete túlhatározott abban az értelemben, hogy az $af(n/b) \leq cf(n)$ regularitási feltétel valamely $c < 1$ állandóra maga után vonja olyan $\epsilon > 0$ állandó létezését, amelyre $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Feladatok

4-1. Példák rekurzív egyenletekre

Adjunk aszimptotikus alsó és felső korlátot az alábbi rekurzív egyenletek $T(n)$ megoldására. Tételezzük fel, hogy $T(n)$ állandó, ha $n \leq 2$. Adjuk meg a lehető legpontosabb korlátokat és igazoljuk állításainkat.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

4-2. A hiányzó egész meghatározása

Egy $A[1\,..\,n]$ tömb egy kivételével minden egész számot tartalmaz 0-tól n-ig. Egyszerű volna a hiányzó egész számot $O(n)$ idő alatt meghatározni, ha a $B[0\,..\,n]$ segédtömbbel regisztrálnánk az A-ban előforduló számokat. Ebben a feladatban azonban nem érhetjük el A egy teljes egész számát egyetlen művelettel. A elemei binárisan vannak megjelenítve, és az egyetlen művelet, amivel elérhetjük őket, az, hogy „menj és hozd ide az $A[i]$ szám j-edik bitjét", ami konstans ideig tart.

Mutassuk meg, hogy ezzel az egyetlen művelettel is meg tudjuk határozni a hiányzó egész számot $O(n)$ idő alatt.

4-3. Paraméterátadási költségek

A könyv egészében azt tételezzük fel, hogy az eljárások hívása során a paraméterátadás konstans időt vesz igénybe még akkor is, ha egy N-elemű tömböt adunk át. Ez a feltételezés az esetek többségében megállja a helyét, mert egy mutatót adunk át, nem pedig magát a tömböt. Ez a feladat három paraméterátadási stratégiát vizsgál:

1. Mutatóval adjuk át a tömböt. Idő = $\Theta(1)$.
2. Másolással adjuk át a tömböt. Idő = $\Theta(N)$, ahol N a tömb mérete.
3. A tömböt úgy adjuk át, hogy csak azt a résztömböt másoljuk át, amelyikre a hívott eljárásnak szüksége van. Idő = $\Theta(p - q + 1)$, ha egy $A[p..q]$ résztömb ment át.

a. Tekintsük azt az esetet, amikor rekurzív bináris kereső algoritmussal keresünk egy számot egy rendezett tömbben (lásd a 2.3-5. gyakorlatot). Adjunk rekurzív képleteket a bináris keresés legrosszabb futási idejére, ha a tömböket a fenti három módszerrel adjuk át, és adjunk éles felső korlátot a rekurzív egyenletek megoldásaira. Az eredeti feladat mérete legyen N, az alfeladaté pedig n.

b. Oldjuk meg az (*a*) feladatot a 2.3.1. pontban ismertetett Összefésülő-rendezés-re.

4-4. További példák rekurzív egyenletekre

Adjunk aszimptotikus alsó és felső korlátot $T(n)$-re az alábbi rekurzív egyenletekben. Tételezzük fel, hogy kellően kicsi n értékre $T(n)$ állandó. A lehető legpontosabb korlátokat adjuk meg, és igazoljuk állításainkat.

a. $T(n) = 3T(n/2) + n \lg n$.

b. $T(n) = 5T(n/5) + n/\lg n$.

c. $T(n) = 4T(n/2) + n^2\sqrt{n}$.

d. $T(n) = 3T(n/3 + 5) + n/2$.

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n - 1) + 1/n$.

h. $T(n) = T(n - 1) + \lg n$.

i. $T(n) = T(n - 2) + 2\lg n$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5. Fibonacci-számok

Ez a feladat a (3.21) rekurzióval definiált Fibonacci-számok néhány tulajdonságára világít rá. A generátorfüggvény-módszert fogjuk használni a Fibonacci-rekurzió megoldásához. Definiáljuk az $\mathcal{F}$ ***generátorfüggvényt*** (vagy ***formális hatványsort***) úgy, hogy

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots .\end{aligned}$$

a. Mutassuk meg, hogy $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Mutassuk meg, hogy

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right),\end{aligned}$$

ahol

$$\phi = \frac{1 + \sqrt{5}}{2} = 1{,}61803\ldots$$

és

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} = -0{,}61803\ldots .$$

c. Mutassuk meg, hogy

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) z^i .$$

d. Bizonyítsuk be, hogy $i > 0$-ra F_i egyenlő a $(\phi^i / \sqrt{5})$-höz legközelebbi egész számmal. (*Útmutatás.* $|\hat{\phi}| < 1$.)

e. Bizonyítsuk be, hogy $F_{i+2} \geq \phi^i$, ha $i \geq 0$.

4-6. VLSI lapka ellenőrzés

Diogenész professzornak n darab, látszólag egyforma VLSI[1] lapkája (angolul chip) van, amelyek elvileg képesek egymás tesztelésére. A professzor tesztelő szerkezetébe egyszerre két lapkát lehet elhelyezni. Ha a szerkezetet készenlétbe helyeztük, akkor mindegyik lapka teszteli a másikat és jelenti az eredményt. A jó lapka mindig helyesen számol be arról, hogy a másik lapka jó-e vagy sem, de egy rossz lapka válaszában nem lehet megbízni. Így a tesztnek az alábbi négy lehetséges kimenetele van:

[1] A VLSI a *very-large-scale integration*, azaz a *nagyon nagy mértékű integráció* rövidítése. Manapság a mikroprocesszorok többségét ezzel az integrált-áramkör lapkagyártási technológiával gyártják.

A lapka állítása	*B* lapka állítása	Következtetés
B jó	*A* jó	mindkettő jó, vagy mindkettő rossz
B jó	*A* rossz	legalább az egyik rossz
B rossz	*A* jó	legalább az egyik rossz
B rossz	*A* rossz	legalább az egyik rossz

a. Mutassuk meg, hogy amennyiben több mint $n/2$ lapka rossz, akkor a professzor nem feltétlenül tudja meghatározni, hogy mely lapkák jók – bármilyen, ezen a páros teszten alapuló stratégiát is használ. Tegyük fel, hogy a rossz lapkák összejátszanak, hogy megtévesszék a professzort.

b. Tekintsük azt a feladatot, hogy n darab lapkából egyetlen jó lapkát kell kiválasztanunk, feltéve, hogy több mint $n/2$ jó lapka van. Mutassuk meg, hogy $\lfloor n/2 \rfloor$ páros teszt elég ahhoz, hogy csaknem felére csökkentsük a feladat nagyságát.

c. Mutassuk meg, hogy $\Theta(n)$ páros teszttel azonosíthatók a jó lapkák, feltéve, hogy több mint $n/2$ lapka jó. Adjuk meg, és oldjuk meg a tesztek számát leíró rekurzív egyenletet.

4-7. Monge-tömbök

Egy $m \times n$ méretű valós elemekből álló A tömböt ***Monge-tömbnek*** nevezünk, ha minden i, j, k és l egészekre $1 \le i < k \le m$ és $1 \le j < l \le n$ esetén teljesül, hogy

$$A[i, j] + A[k, l] \le A[i, l] + A[k, j].$$

Másként fogalmazva, ha kiválasztunk két sort és két oszlopot egy Monge-tömbben, és tekintjük a sorok és oszlopok metszetében lévő négy elemet, akkor a bal felső és a jobb alsó elem összege kisebb vagy egyenlő, mint a bal alsó és a jobb felső elem összege. A következő tömb például rendelkezik ezzel a tulajdonsággal:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Bizonyítsuk be, hogy egy tömb akkor és csak akkor Monge-tömb, ha minden $i = 1, 2, \ldots, m - 1$ és $j = 1, 2, \ldots, n - 1$ esetén teljesül az

$$A[i, j] + A[i + 1, j + 1] \le A[i, j + 1] + A[i + 1, j]$$

egyenlőtlenség. (*Útmutatás.* Az akkor rész bizonyításához használjunk teljes indukciót külön a sorokra és az oszlopokra.)

b. A következő tömb nem Monge-tömb. Változtassunk meg egy elemet úgy, hogy Monge-tömb legyen. (*Útmutatás.* Használjuk a feladat (a) részét.)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Jelölje $f(i)$ az i-edik sor balról a legelső minimális elemének az oszlopindexét. Bizonyítsuk be, hogy $f(1) \leq f(2) \leq \cdots \leq f(m)$ teljesül minden $m \times n$ méretű Monge-tömbben.

d. A következő oszd-meg-és-uralkodj elvű algoritmus megkeresi egy A Monge-tömb minden egyes sorában a balról az első minimális elemet:

> Az A' részmátrix álljon A páros sorszámú soraiból. Rekurzívan határozzuk meg A' minden sorában a balról az első minimális elemet, majd keressük meg A páratlan soraiban is a balról az első minimális elemet.

Mutassuk meg, hogy a páratlan sorokban $O(m + n)$ időben meg tudjuk keresni a balról az első minimális elemet (ha a páros sorszámú sorokra ez már ismert).

e. Adjunk rekurzív egyenletet a feladat (d) részében leírt algoritmus futási idejére. Mutassuk meg, hogy ennek megoldása $O(m + n \log m)$.

Megjegyzések a fejezethez

A rekurzív egyenleteket Fibonacci (róla nevezték el a Fibonacci-számokat) már 1202-ben tanulmányozta. A. De Moivre vezette be a rekurzív egyenletek megoldására a generátorfüggvények módszerét (lásd a 4-5. feladatot). A mester módszert Bentley, Haken és Saxe művéből [41] vettük át, ahol az a bővített változat szerepel, amelyet a 4.4-2. gyakorlatban igazoltunk. Knuth [182] és Liu [205] azt mutatja meg, hogyan lehet lineáris rekurzív egyenleteket megoldani generátorfüggvény módszerrel. Purdom és Brown [252], illetve Graham, Knuth és Patashnik [132] bővebben tárgyalja a rekurzív egyenletek megoldását.

Oszd-meg-és-uralkodj típusú rekurziók megoldására számos olyan módszert adtak, amelyek olyan esetekben is használhatók, amikor a mester módszer nem: lásd Akra és Bazzi [13], Roura [262], illetve Verma [306] munkáit. Akra és Bazzi módszere a következő alakú rekurziós egyenletekre használható:

$$T(n) = \sum_{i=1}^{k} a_i T\left(\left\lfloor \frac{n}{b_i} \right\rfloor\right) + f(n), \tag{4.15}$$

ahol $k \geq 1$; az a_i együtthatók pozitívak és összegük legalább 1; mindegyik b_i nagyobb mint 1; az $f(n)$ függvény deriváltja polinomiálisan korlátos, pozitív és monoton növekedő; és végül minden $c > 1$ konstansra léteznek olyan n_0, $d > 0$ konstansok, hogy $f(n/c) \geq d f(n)$ teljesül minden $n \geq n_0$ esetén. Ez a módszer használható például a $T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$ rekurzióra is, amelyre a mester módszer már nem alkalmazható. A (4.15) rekurzió

megoldásához először meghatározzuk azt a p értéket, amelyre $\sum_{i=1}^{k} a_i b_i^{-p} = 1$. (Ilyen p érték mindig egyértelműen létezik, és mindig pozitív.) Ekkor a rekurzió megoldása

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^{n} \frac{f(x)}{x^{p+1}}\, dx\right),$$

ahol n' egy kellően nagy konstans. Az Akra–Bazzi-módszer használata némileg körülményes, de jól alkalmazható olyan rekurziók esetén, amelyek nagyon különböző méretű részproblémákra bontást modelleznek. A mester módszert egyszerűbb használni, de csak olyankor alkalmazható, amikor a részproblémák mérete azonos.

5. Valószínűségi elemzés

Ez a fejezet a valószínűségi elemzés és a véletlenített algoritmusok témakörébe nyújt bevezetést. A valószínűségszámításban nem járatos olvasó számára hasznos lehet elolvasni a C. függeléket, amely áttekinti a szükséges előismereteket. Valószínűségi elemzéssel és véletlenített algoritmusokkal a könyvben számos helyen fogunk találkozni.

5.1. A munkatársfelvétel probléma

Tegyük fel, hogy egy új munkatársra van szükségünk. Kísérleteink az állás betöltésére kudarcot vallottak, így egy állásközvetítő céghez fordulunk. Az ügynökség minden nap küld egy új jelentkezőt az állásra. A jelölttel folytatott felvételi beszélgetés után eldöntjük, hogy felvesszük-e az állásra, vagy sem. Minden egyes érkező jelölt után egy csekély kezelési költséget kell fizetni az ügynökségnek. A jelölt felvétele már sokkal költségesebb, ugyanis ekkor el kell bocsátani az előző alkalmazottat és nagyobb jutalékot kell fizetni az ügynökségnek. Mindenképpen azt szeretnénk, hogy mindig a lehető legalkalmasabb munkatárssal dolgozzunk együtt. Ezért úgy döntünk, hogy ha a felvételi beszélgetés során kiderül, hogy a jelölt alkalmasabb, mint az asszisztensünk, akkor elbocsátjuk az asszisztenst és helyére felvesszük a jelöltet. Készek vagyunk kifizetni ennek a módszernek a költségeit, de azért szeretnénk megbecsülni, hogy mekkora összeg kifizetésére kell felkészülnünk.

A MUNKATÁRSFELVÉTEL eljárás ezt a stratégiát írja le formálisan. Feltételezzük, hogy az állásra jelentkező jelöltek 1-től n-ig terjedő sorszámokat kaptak. Továbbá feltesszük, hogy az i-edik jelölttel folytatott felvételi beszélgetés során el tudjuk dönteni, hogy ő volt-e az eddigi legalkalmasabb jelölt. Az eljárás kezdetben felvesz egy fiktív nulladik jelöltet, aki minden más jelöltnél rosszabb.

```
MUNKATÁRSFELVÉTEL(n)
1  legjobb ← 0        ▷ a nulladik jelölt egy mindenkinél rosszabb fiktív jelölt
2  for i ← 1 to n
3      do felvételi beszélgetés az i-edik jelölttel
4         if i-edik jelölt jobb, mint a legjobb sorszámú jelölt
5            then legjobb ← i
6                 i-edik jelölt felvétele
```

Az eljárás költsége most mást jelent, mint a 2. fejezetben vizsgált modellben. Most ugyanis nem a MUNKATÁRSFELVÉTEL eljárás futási ideje érdekel minket, hanem a felvételi beszélgetések és a munkatársak felvétele miatt fizetendő összeg. Első pillantásra úgy tűnhet, hogy ennek a költségnek a vizsgálata teljesen más jellegű feladat, mint mondjuk az összefésülő rendezés futási idejének vizsgálata. Ennek ellenére teljesen azonos módszereket alkalmazhatunk a költség és a futási idő vizsgálatához, hiszen mindkét esetben azt kell meghatároznunk, hogy az egyes elemi műveletek hányszor kerülnek végrehajtásra.

A felvételi beszélgetés költsége alacsony, legyen ez c_b, míg egy új alkalmazott felvétele igencsak költséges, jelölje ezt c_f. Ha a jelöltek közül összesen m embert vettünk fel, akkor az algoritmus teljes költsége $O(nc_b + mc_f)$. Függetlenül attól, hogy végül hány embert alkalmaztunk, mindenképpen lefolytattunk n felvételi beszélgetést az n jelölttel, így a felvételi beszélgetések összköltsége mindig nc_b. Ezért a továbbiakban az alkalmazottak felvételének a költségét, az mc_f tagot fogjuk vizsgálni, ez az érték az algoritmus minden futtatásánál különböző lehet.

A fent leírt szituáció egy gyakran előforduló számítási feladatot modellez. Sokszor előfordul az a feladat, hogy egy sorozat maximális vagy minimális elemét úgy kell meghatároznunk, hogy végigolvassuk a sorozatot és közben nyilvántartjuk az aktuális *bajnokot*. A munkatársfelvétel probléma azt modellezi, hogy milyen sűrűn kell változtatni az eddig legjobbnak tartott elemet.

A legrosszabb eset vizsgálata

Az számít a legrosszabb esetnek, ha végül az összes jelentkezőt alkalmazzuk. Ez akkor fordulhat elő, ha a jelöltek alkalmasság szerint növekvő sorrendben érkeznek. Ebben az esetben mind az n jelöltet felvesszük, így a teljes felvételi költség $O(nc_f)$.

Általában persze azt várjuk, hogy a jelöltek nem pont alkalmasság szerint növekvő sorrendben érkeznek. Valójában fogalmunk sincs arról, hogy milyen sorrendben érkeznek, és semmilyen módon sem tudjuk befolyásolni ezt a sorrendet. Ezért természetesen adódó kérdés azt vizsgálni, hogy mire számíthatunk átlagos, tipikus esetekben.

Valószínűségi elemzés

Valószínűségi elemzésről akkor beszélünk, amikor valószínűségszámítási módszereket alkalmazunk egy probléma vizsgálatára. Legtöbbször az algoritmusok futási idejének meghatározására fogjuk használni a valószínűségi elemzést. Néha viszont valamilyen más mennyiséget kívánunk meghatározni, mint például a felvételek költségét a MUNKATÁRSFELVÉTEL eljárásban. Valószínűségi elemzést csak akkor lehet alkalmazni, ha ismerjük a bemeneti adatok eloszlását, vagy legalábbis valamilyen feltételezéseink vannak a bemeneti adatok eloszlásáról. Ekkor meghatározhatjuk az algoritmusunk várható futási idejét. A várható érték a bemeneti adatok eloszlása felett értendő, vagyis lényegében a futási időt átlagoljuk az összes lehetséges bemenetre.

Nagyon körültekintő módon kell eljárnunk a bemeneti eloszlás megválasztásakor. Bizonyos problémáknál természetesen adódó, ésszerű feltételezésekkel élhetünk a bemeneti eloszlással kapcsolatban. Ilyenkor alkalmazhatjuk a valószínűségi elemzést új, hatékony algoritmusok tervezésére, illetve a probléma jobb megértésére. Más problémáknál viszont nem tudjuk a bemeneti eloszlást meghatározni, ebben az esetben a valószínűségi elemzés nem alkalmazható.

A munkatársfelvétel problémánál feltehetjük, hogy a jelöltek véletlenszerű sorrendben érkeznek. Mit jelent ez a mi esetünkben? Feltehetjük, hogy bármely két jelöltet össze tudunk hasonlítani, és el tudjuk dönteni melyik az alkalmasabb, vagyis adott egy teljes rendezés a jelöltek halmazán. (A teljes rendezés definícióját lásd a B. függelékben.) Így a jelölteket rangsor szerint sorba lehet rendezni. Minden jelölt kap egy 1 és n közti egyedi számot, $rang(i)$ jelzi az i-edik jelölt rangsor szerinti sorrendjét, ahol a nagyobb szám az alkalmasabb jelöltet jelenti. Ha a rangokat egy rendezett listában felsoroljuk, akkor az így kapott $\langle rang(1), rang(2), \ldots, rang(n)\rangle$ lista egy permutációja lesz az $\langle 1, 2, \ldots, n\rangle$ listának. Feltételeztük, hogy a jelöltek véletlenszerű sorrendben érkeznek, ez azzal ekvivalens, hogy rangok listája azonos valószínűséggel lehet az $1, 2, \ldots, n$ számok $n!$ különböző permutációjának egyike. Másként fogalmazva: a rangok listája ***egyenletes eloszlású véletlen permutáció,*** vagyis az összes lehetséges $n!$ permutáció azonos valószínűséggel fordulhat elő.

A munkatársfelvétel probléma valószínűségi elemzését az 5.2. alfejezetben tárgyaljuk.

Véletlenített algoritmusok

Ahhoz, hogy a valószínűségi elemzést alkalmazni lehessen, valamilyen ismerettel mindenképpen rendelkeznünk kell a bemenő adatok eloszlásáról. Gyakran nagyon keveset tudunk erről az eloszlásról. Előfordul, hogy tudunk ugyan valamit az eloszlásról, de ezt a tudást mégsem tudjuk olyan formában megfogalmazni, ami algoritmustervezésnél hasznos lehet. Gyakran viszont magát a véletlenszerűséget tudjuk felhasználni algoritmusok tervezésére és elemzésére azáltal, hogy egy algoritmus bizonyos részeinek működését véletlenszerűvé tesszük.

A munkatársfelvétel problémában úgy tűnhet, hogy a jelöltek véletlen sorrendben érkeznek, de valójában nem tudjuk megállapítani, hogy ez tényleg teljesül-e. Véletlenített algoritmust szeretnénk adni a problémára, de ehhez valamilyen módon tudnunk kell befolyásolni az érkező jelöltek sorrendjét. Ezért némileg változtatunk modellünkön. Feltesszük, hogy az ügynökségnek n jelöltje van, és előre elküldik nekünk az összes jelölt listáját. Erről a listáról minden nap véletlenszerűen választunk egy jelöltet, akit elhívunk felvételi beszélgetésre. Noha semmit sem tudunk a jelöltekről (a nevükön kívül), a változás mégis jelentős. Mivel az érkezési sorrendet mi határozzuk meg, nem kell pusztán arra a feltételezésünkre hagyatkoznunk, hogy a jelöltek véletlenszerű sorrendben érkeznek, a véletlenszerű választás biztosítja a véletlenszerű érkezési sorrendet.

Általánosabban, egy algoritmust ***véletlenített algoritmusnak*** nevezünk, ha a viselkedését, lépéseit nemcsak a bemenet határozza meg, hanem a ***véletlenszám generátor*** által előállított értékek is. Feltételezzük, hogy rendelkezésünkre áll egy véletlenszámokat generáló VÉLETLEN eljárás. Az eljárás úgy működik, hogy a VÉLETLEN(a, b) hívásra egy a és b közötti (a határokat is beleértve) egész számot ad vissza, minden számnak azonos a valószínűsége. Például a VÉLETLEN$(0, 1)$ hívás eredménye $1/2$ valószínűséggel 0 és $1/2$ valószínűséggel 1. A VÉLETLEN$(3, 7)$ hívás 3, 4, 5, 6 vagy 7 egyikét adja vissza, mindegyiknek $1/5$ a valószínűsége. A VÉLETLEN által visszaadott értékek függetlenek a korábbi hívások eredményétől. Úgy is gondolhatunk a VÉLETLEN eljárás meghívására, mint egy $(b - a + 1)$ oldalú kocka feldobására. (Gyakorlatban a legtöbb programozási környezetben van ***pszeudo-véletlenszám generátor,*** ami egy statisztikusan véletlennek tűnő értékeket visszaadó determinisztikus algoritmus.)

Gyakorlatok

5.1-1. Mutassuk meg, hogy a MUNKATÁRSFELVÉTEL eljárás 4. sorának azon feltételezéséből, hogy el tudjuk dönteni, melyik jelölt a legjobb, következik az, hogy ismerjük a jelöltek rangsor szerinti teljes rendezését.

5.1-2.⋆ Írjunk egy olyan RANDOM(a, b) eljárást, ami csak RANDOM$(0, 1)$ hívásokat használ. Mennyi lesz az eljárás várható futási ideje a és b függvényében?

5.1-3.⋆ Tegyük fel, hogy 0-t vagy 1-et szeretnénk kiírni, mindkettőt $1/2$ valószínűséggel. Rendelkezésünkre áll egy HAMIS-VÉLETLEN eljárás, amely 0-t vagy 1-et ad vissza. A 0 valószínűsége p, az 1 valószínűsége $1 - p$, ahol $0 < p < 1$, de p pontos értékét nem ismerjük. Írjunk egy olyan algoritmust, ami a HAMIS-VÉLETLEN eljárást használja, de $1/2$-$1/2$ valószínűséggel ad vissza 0-t vagy 1-et. Mennyi lesz az algoritmus várható futási ideje p függvényében?

5.2. Indikátor valószínűségi változók

Indikátor valószínűségi változókat számos esetben használunk algoritmusok valószínűségi elemzésére, az 5.2. alfejezetben is ezzel a módszerrel fogjuk vizsgálni a munkatársfelvétel problémát. Indikátor valószínűségi változók segítségével kényelmes módon tudunk kapcsolatot teremteni a valószínűségek és a várható értékek között. Tekintsünk egy A eseményt az S eseménytérben. Ekkor az A eseményhez tartozó $\mathrm{I}\{A\}$ ***indikátor valószínűségi változó*** definíciója a következő:

$$\mathrm{I}\{A\} = \begin{cases} 1, & \text{ha az } A \text{ esemény bekövetkezik,} \\ 0, & \text{ha az } A \text{ esemény nem következik be.} \end{cases} \tag{5.1}$$

Példaként határozzuk meg a kapott fejek számának várható értékét, ha feldobunk egy (szabályos) érmét. Az eseménytér $S = \{F, I\}$, a valószínűségek $\Pr\{F\} = \Pr\{I\} = 1/2$. Tekintsük az F eseményhez tartozó X_F indikátor valószínűségi változót. Ennek a változónak az értéke a dobott fejek számát adja meg: ha fejet dobunk, akkor 1, egyébként nulladik Formálisan megfogalmazva:

$$X_F = \mathrm{I}\{F\} = \begin{cases} 1, & \text{ha az } F \text{ esemény következik be,} \\ 0, & \text{ha az } I \text{ esemény következik be.} \end{cases}$$

A dobás során kapott fejek várható száma megegyezik az X_F indikátor valószínűségi változó várható értékével:

$$\begin{aligned} \mathrm{E}[X_F] &= \mathrm{E}[\mathrm{I}\{F\}] \\ &= 1 \cdot \Pr\{F\} + 0 \cdot \Pr\{I\} \\ &= 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{2} \\ &= \frac{1}{2} . \end{aligned}$$

Vagyis a dobás során kapott fejek számának várható értéke $1/2$. A következő lemma általánosan is kimondja, hogy az A eseményhez tartozó indikátor valószínűségi változó várható értéke megegyezik az A esemény bekövetkezésének valószínűségével.

5.1. lemma. *Legyen A egy esemény az S eseménytérben, és legyen $X_A = \mathrm{I}\{A\}$. Ekkor* $\mathrm{E}[X_A] = \Pr\{A\}$.

Bizonyítás. Az indikátor valószínűségi változó (5.1) definíciójából és a várható érték definíciójából következően

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\left\{\overline{A}\right\} \\ &= \Pr\{A\}, \end{aligned}$$

ahol $\overline{A}$ jelöli az $S - A$ eseményt, az A esemény komplementerét. ■

Kissé nehézkes módszernek tűnhet az egy dobás során kapott fejek várható számának meghatározására indikátor valószínűségi változót használni. Viszont nagyon hasznos lehet ez a módszer olyan esetekben, amikor több ismételt kísérletet végzünk. Például az indikátor valószínűségi változók segítségével könnyen levezethetjük a (C.36) egyenlőséget. Itt az n dobás során kapott fejek számának várható értékét úgy határozzuk meg, hogy külön-külön kiszámoljuk annak a valószínűségét, hogy 0 fejet dobunk, 1 fejet dobunk, 2 fejet dobunk stb. A (C.37) egyenlet viszont egy egyszerűbb módszert mutat, amely valójában rejtett módon indikátor valószínűségi változókat használ. Hogy pontosabban lássuk mir ől is van szó, jelöljük X_i-vel azt az indikátor valószínűségi változót, ami ahhoz az eseményhez tartozik, hogy az i-edik dobás fej: $X_i = \mathrm{I}\{\text{az } i\text{-edik dobásnál az } F \text{ esemény következik be}\}$. Az X valószínűségi változó jelölje az n érmedobás során kapott fejek számát, vagyis

$$X = \sum_{i=1}^{n} X_i.$$

A dobott fejek számának várható értékét szeretnénk meghatározni, ezért tekintsük a fenti egyenletben mindkét oldal várható értékét:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right].$$

Az egyenlet jobb oldalán valószínűségi változók összegének várható értéke áll. Az 5.1. lemma révén mindegyik valószínűségi változónak ismerjük a várható értékét. A (C.20) egyenlőség (a várható érték linearitása) alapján viszont az összeg várható értéke is könnyen meghatározható: egyszerűen csak az n valószínűségi változó várható értékének az összegét kell venni. A várható érték linearitása az indikátor valószínűségi változók módszerét nagyon hatékony eszközzé teszi, olyan esetekben is lehet használni, amikor a valószínűségi változók nem függetlenek. A fejek számának várható értékét a következ őképpen tudjuk meghatározni:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] \end{aligned}$$

$$= \sum_{i=1}^{n} \frac{1}{2}$$
$$= \frac{n}{2}.$$

Az indikátor valószínűségi változók alkalmazásával sikerült a számolást nagyban leegyszerűsíteni a (C.36) egyenlethez képest. A könyvben még számos helyen fogunk használni indikátor valószínűségi változókat.

A munkatársfelvétel probléma elemzése indikátor valószínűségi változókkal

Visszatérve a munkatársfelvétel problémára, szeretnénk meghatározni, hogy várhatóan hányszor fogunk új asszisztenst felvenni. Ahhoz, hogy a valószínűségi elemzést lehessen alkalmazni, az előző alfejezetben tárgyalt módon feltesszük, hogy a jelöltek véletlenszerű sorrendben érkeznek. (Az 5.3. alfejezetben majd látni fogjuk, hogy hogyan szabadulhatunk meg ettől a feltételezéstől.) Legyen X az a valószínűségi változó, amelynek értéke azt adja meg, hogy hányszor veszünk fel új asszisztenst. X várható értékének meghatározására alkalmazhatnánk a várható érték definícióját, de a (C.19) egyenletből kapott

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\}$$

kiszámolása igencsak fáradságos lenne. Ehelyett inkább az indikátor valószínűségi változók módszerét fogjuk használni, ez nagyban leegyszerűsíti a számolást.

Az indikátor valószínűségi változók módszerét a következőképpen fogjuk alkalmazni: $\mathrm{E}[X]$ értékét nem úgy határozzuk meg, hogy definiálunk egyetlen, a felvett alkalmazottak számát megadó valószínűségi változót, hanem definiálunk n változót, mindegyik változó egy adott jelöltről mondja meg, hogy felvettük-e vagy sem. Az X_i indikátor valószínűségi változó tartozzon ahhoz az eseményhez, hogy felvettük az i-edik jelöltet. Ekkor

$$X_i = \mathrm{I}\{\text{az } i\text{-edik jelöltet felvettük}\} = \begin{cases} 1, & \text{ha az } i\text{-edik jelöltet felvettük,} \\ 0, & \text{ha az } i\text{-edik jelöltet nem vettük fel,} \end{cases} \tag{5.2}$$

és

$$X = X_1 + X_2 + \cdots + X_n. \tag{5.3}$$

Az 5.1. lemma miatt

$$\mathrm{E}[X_i] = \Pr\{\text{az } i\text{-edik jelöltet felvettük}\},$$

így annak a valószínűségét kell meghatároznunk, hogy az 5–6. sorok végrehajtódnak a MUNKATÁRSFELVÉTEL eljárás során.

Az 5. sorban az i-edik jelöltet pontosan akkor vesszük fel, ha az i-edik jelölt jobb az 1., 2., . . ., $(i-1)$-edik jelöltnél. Feltételezésünk szerint a jelöltek véletlen sorrendben érkeznek, így az első i jelölt sorrendje is véletlen. Az első i jelölt közül bármelyik azonos valószínűséggel lehet az eddigi legjobb. Így tehát $1/i$ annak a valószínűsége, hogy az i-edik jelölt jobb az előtte lévő $i-1$ jelölt mindegyikénél, vagyis $1/i$ a valószínűsége annak, hogy felvesszük az i-edik jelöltet. Az 5.1. lemma alapján így

$$\mathrm{E}[X_i] = \frac{1}{i} \tag{5.4}$$

írható. Ekkor $\mathrm{E}[X]$ a következő módon határozható meg:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{((5.3) egyenlet)} && (5.5)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(várható érték linearitása)} &&\\
&= \sum_{i=1}^{n} \frac{1}{i} && \text{((5.4) egyenlet)} &&\\
&= \ln n + O(1) && \text{((A.7) egyenlet).} && (5.6)
\end{aligned}$$

Annak ellenére, hogy n jelölt érkezett, átlagosan csak nagyjából $\ln n$ jelöltet vettünk fel. Eddigi eredményeinket a következő lemma foglalja össze.

5.2. lemma. *Ha a jelöltek véletlen sorrendben érkeznek, akkor a* Munkatársfelvétel *eljárásban a jelöltek felvételének teljes költsége* $O(c_f \ln n)$.

Bizonyítás. A korlát közvetlen következménye a költség definíciójának és az (5.6) egyenletnek. ■

A jelöltek felvételének várható költsége jelentősen kisebb, mint a legrosszabb esetre adott $O(nc_f)$ felső korlát.

Gyakorlatok

5.2-1. Feltéve, hogy a jelöltek véletlenszerű sorrendben érkeznek, mi a valószínűsége annak, hogy a Munkatársfelvétel eljárásban pontosan egy jelöltet veszünk fel? Mi a valószínűsége, hogy pontosan n jelöltet veszünk fel?

5.2-2. Feltéve, hogy a jelöltek véletlenszerű sorrendben érkeznek, mi a valószínűsége annak, hogy a Munkatársfelvétel eljárásban pontosan két jelöltet veszünk fel?

5.2-3. Indikátor valószínűségi változók felhasználásával határozzuk meg n kockadobás összegének várható értékét.

5.2-4. Indikátor valószínűségi változók segítségével oldjuk meg a ***kalap problémát.*** Egy étteremben n vendég leadja a kalapját a ruhatárosnak. A ruhatáros a kalapokat véletlenszerű sorrendben adja vissza. Várhatóan hány vendég fogja visszakapni saját kalapját?

5.2-5. Az $A[1\,..\,n]$ tömb tartalmazzon n darab különböző számot. Ha $i < j$ esetén $A[i] > A[j]$, akkor az (i, j) párt A egy ***inverziójának*** nevezzük. (Inverziókkal kapcsolatban lásd még a 2-4. feladatot.) Tegyük fel, hogy az A tömb az $\langle 1, 2, \ldots, n\rangle$ elemek egy véletlen permutációja. Indikátor valószínűségi változók segítségével határozzuk meg az inverziók számának várható értékét.

5.3. Véletlenített algoritmusok

Az előző alfejezetben a bemenő adatok eloszlásának ismeretében vizsgáltuk az algoritmusok átlagos viselkedését. Sok esetben ez nem tehető meg, mivel nem ismerjük a bemenő adatok eloszlását. Véletlenített algoritmusok viszont ilyen esetekben is használhatók.

Tekintsük a munkatársfelvétel problémát, vagy egy más olyan feladatot, ahol hasznos lenne, ha feltehetnénk, hogy minden bemenő permutáció azonos valószínűségű. Ekkor a valószínűségi elemzés rögtön egy véletlenített algoritmust is sugall. Nem teszünk semmilyen feltevést a bemenő adatok eloszlására, hanem mi magunk biztosítjuk a megfelelő eloszlást. Az algoritmus futása előtt véletlenszerűen permutáljuk a jelölteket, ez garantálja, hogy minden permutáció azonos valószínűségű legyen. Várakozásaink szerint ekkor durván számolva $\ln n$ jelöltet fogunk felvenni. Viszont ez az érték most nem függ a bemenő adatoktól, *minden* bemenetre ennyi lesz a várható érték, nem csak akkor, ha a bemenő adatokat valamilyen speciális eloszlás szerint állítjuk elő.

Vegyük észre a különbséget a valószínűségi elemzés és a véletlenített algoritmusok között. Az 5.2. alfejezetben azt mutattuk meg, hogy ha a jelöltek sorrendje véletlenszerű, akkor várhatóan $\ln n$ új munkatársat fogunk felvenni. Az algoritmus determinisztikus volt, egy adott bemenet esetén minden futtatásnál ugyanazokat a jelölteket vesszük fel. Különböző bemenő adatokra a felvett jelöltek száma különbözhet, ez az érkező jelöltek rangsor szerinti sorrendjétől függ. Mivel a felvett jelöltek száma csak a jelöltek rangsorolásától függ, ezért egy adott bemenetet teljesen leírunk azzal, ha megadjuk sorban az érkező jelöltek rangsor szerinti sorrendjét, vagyis a $\langle rang(1), rang(2), \ldots, rang(n)\rangle$ listát. Például az $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$ lista esetén az algoritmus mindig felveszi mind a 10 jelöltet. Mivel mindegyik jelölt jobb az összes előzőnél, ezért az 5–6. sorok minden egyes iterációban végrehajtásra kerülnek. Az $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$ rangsor lista esetén viszont csak az első jelöltet vesszük fel. Az $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$ lista esetén háromszor fogunk új munkatársat felvenni: az ötödik, a nyolcadik és a tizedik jelöltet alkalmazzuk. Mivel az algoritmus költsége a felvett új alkalmazottak számától függött, ezért a lehetséges bemenetek egy része nagyon költséges (mint például az A_1), egy része nagyon alacsony költségű (például A_2), egy része pedig csak közepesen költséges (például A_3).

Tekintsük viszont a véletlenített algoritmusunkat, amely először véletlenszerű sorrendbe rendezi a jelölteket, és csak utána választja ki közülük a legjobbat. Most a véletlenszerűség magában az algoritmusban van, nem a bemenő adatok eloszlásában. Ha veszünk egy tetszőleges bemenetet, például a fenti A_3-t, akkor nem lehet előre megmondani, hogy hányszor fog az algoritmus új maximumot találni, ugyanis ez futásról futásra változhat. Lehetséges, hogy egy futtatásnál az algoritmus az A_1 permutációt állítja elő, és ezért tíz alkalommal kell módosítani az aktuális legjobb jelöltet. Ugyanakkor az is lehetséges, hogy a következő futtatásnál viszont az A_2 permutáció áll elő, és így csak egy alkalommal kell az aktuális maximumot módosítani. Harmadszorra futtatva megint egy új permutációt kaphatunk, a módosítások száma megint különböző lehet. Minden egyes futtatásnál az algoritmus végrehajtása a véletlenszerű választásoktól is függ, így a pontos működése alkalmasint különbözni fog az előző futtatásoktól. Hasonlóan sok más véletlenített algoritmushoz, ennél az algoritmusnál *nincs olyan speciális bemenő adat, amelyre az algoritmus a legrosszabb esetbeli viselkedését mutatná*. Még egy esküdt ellenségünk sem tudna olyan bemenő adatot adni, amelyre az algoritmus rosszul viselkedne, mivel a véletlenszerű permutálás miatt a bemenő adatok sorrendje nem befolyásolja az algoritmus működését. A véletlenített algoritmusnak a viselkedése csak akkor lesz kedvezőtlen, ha a véletlenszám generátor egy *szerencsétlen* permutációt állít elő.

A munkatársfelvétel problémára adott algoritmus véletlenített változatához csak egy módosításra van szükség: az első felvételi beszélgetés előtt véletlenszerű sorrendbe kell rendezni a jelölteket.

VÉLETLENÍTETT-MUNKATÁRSFELVÉTEL(n)

```
1  a jelöltek listájának véletlenszerű permutálása
2  legjobb ← 0        ▷ a nulladik jelölt egy mindenkinél rosszabb fiktív jelölt
3  for i ← 1 to n
4      do felvételi beszélgetés az i-edik jelölttel
5         if i-edik jelölt jobb, mint a legjobb sorszámú jelölt
6           then legjobb ← i
7                i-edik jelölt felvétele
```

Ezzel az egyszerű módosítással olyan véletlenített algoritmust kaptunk, amely minden további feltételezés nélkül pontosan olyan jól működik, mint amikor az eredeti algoritmust használtuk azzal a feltételezéssel, hogy a jelöltek sorrendje véletlenszerű.

5.3. lemma. *A* VÉLETLENÍTETT-MUNKATÁRSFELVÉTEL *eljárás futása során a felvételi költség várható értéke* $O(c_f \ln n)$.

Bizonyítás. A bemenő adatok permutálása után a helyzet pontosan megegyezik a MUNKATÁRSFELVÉTEL valószínűségi elemzésében vizsgálttal. ■

Jól látszik a különbség a valószínűségi elemzés és a véletlenített algoritmusok között, ha összehasonlítjuk az 5.2. és az 5.3. lemmát. Az 5.2. lemmában valamilyen feltételezéssel éltünk a bemenő adatokkal kapcsolatban. Az 5.3. lemmában nem volt szükségünk ilyen feltételezésekre, de a bemenet véletlenítése némileg növelte a futási időt. Az alfejezet hátralevő részében a bemenet véletlen permutálásával kapcsolatos további tudnivalókat tárgyaljuk.

Tömbök véletlen permutációi

Sok esetben a véletlenített algoritmus úgy biztosítja a bemenet véletlenszerűségét, hogy a bemenetként kapott adatokat véletlenszerű sorrendbe rendezi. (Persze sok más módon is használhat egy algoritmus véletlenszerű lépéseket.) Egy tömb véletlen permutációjának előállítására két módszert mutatunk. Az általánosság megszorítása nélkül feltehetjük, hogy a tömb az 1, 2, . . ., n elemeket tartalmazza, ennek keressük egy véletlen permutációját.

Egy lehetséges módszer a feladatra a következő: rendeljünk minden $A[i]$ elemhez egy véletlenszerű $P[i]$ prioritást, majd rendezzük A elemeit a prioritás szerint növekvő sorrendbe. Például ha a bemeneti tömb $A = \langle 1, 2, 3, 4\rangle$ és a véletlenszerű választás során $P = \langle 36, 3, 97, 19\rangle$ lesz a prioritások értéke, akkor végeredményül a $B = \langle 2, 4, 1, 3\rangle$ sorrend adódik, mivel a második prioritás a legkisebb, és utána sorrendben a negyedik, az első és a harmadik prioritás következik. A PERMUTÁLÁS-RENDEZÉSSEL eljárás ezt a módszert használja:

PERMUTÁLÁS-RENDEZÉSSEL(A)

```
1  n ← hossz[A]
2  for i ← 1 to n
3       do P[i] = VÉLETLEN(1, n^3)
4  rendezzük az A tömböt P szerint
5  return A
```

A 3. sorban 1 és n^3 közötti véletlen számot választunk. Azért választjuk a számokat az 1 és n^3 közötti tartományból, hogy nagy valószínűséggel csupa különböző számot kapjunk. (Az 5.3-5. gyakorlat annak bizonyítását kéri, hogy legalább $1 - 1/n$ valószínűséggel csupa különböző számokat kapunk. Az 5.3-6. gyakorlatban úgy kell módosítani az algoritmust, hogy akkor is működjön, ha a prioritások között vannak azonosak is.) A továbbiakban feltételezzük, hogy a prioritások különböznek.

Az algoritmus legidőigényesebb része a rendezés a 4. sorban. A 8. fejezetben látni fogjuk, hogy összehasonlítás alapú rendezést használva a rendezés futási ideje $\Omega(n \lg n)$. Ezt az alsó korlátot az összefésülő rendezés eléri, láttuk, hogy a futási ideje $\Theta(n \lg n)$. (A II. részben további $\Theta(n \lg n)$ idejű összehasonlító rendezésekkel fogunk találkozni.) Ha a $P[i]$ a j-edik legkisebb prioritás, akkor a rendezés után az $A[i]$ elem a j-edik pozíción fog állni. A kimenet tehát egy permutációt ad meg. Azt kell még belátnunk, hogy az eljárás ***egyenletes eloszlású véletlen permutációt*** ad, vagyis az 1, 2, . . ., n számok minden permutációja azonos valószínűséggel áll elő.

5.4. lemma. *A* Permutálás-rendezéssel *eljárás egyenletes eloszlású véletlen permutációt ad, ha a prioritások mind különböznek.*

Bizonyítás. Először tekintsük azt a permutációt, amelyben minden $A[i]$ elem az i-edik legkisebb prioritást kapja. Megmutatjuk, hogy ez a permutáció $1/n!$ valószínűséggel áll elő. Jelöljük X_i-vel azt az eseményt, hogy az $A[i]$ elem kapja az i-edik legkisebb prioritást ($i = 1, 2, \ldots, n$). Szeretnénk meghatározni annak a valószínűségét, hogy az összes X_i esemény egyszerre bekövetkezik:

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} .$$

A C.2-6. gyakorlat alapján ez a valószínűség úgy is írható, hogy

$$\begin{aligned}&\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\} \\ &\quad \cdots \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\} .\end{aligned}$$

Világos, hogy $\Pr\{X_1\} = 1/n$: ez annak a valószínűsége, hogy az n véletlen prioritásból pont az első lesz a legkisebb. Továbbá $\Pr\{X_2 \mid X_1\} = 1/(n-1)$, ugyanis ha $A[i]$ prioritása a legkisebb, akkor a maradék $n - 1$ elem közül bármelyik azonos valószínűséggel lehet a második legkisebb. Általánosan azt mondhatjuk, hogy $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n - i + 1)$ minden $i = 2, 3, \ldots, n$ értékre, hiszen ha az első $i - 1$ elem kapta a legkisebb $i-1$ prioritást, akkor a maradék $(n-i+1)$ elem bármelyike azonos valószínűséggel kaphatja az i-edik legkisebb prioritást. Ebből következően

$$\begin{aligned}\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ &= \frac{1}{n!} ,\end{aligned}$$

és ezzel megmutattuk, hogy az eljárás $1/n!$ valószínűséggel állítja elő az identitás permutációt.

Kis módosítással a fenti bizonyítás tetszőleges permutációra is alkalmazható. Legyen $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$ az $\{1, 2, \ldots, n\}$ halmaz egy rögzített permutációja. Jelöljük r_i-vel

az $A[i]$ elemhez rendelt prioritás rangsor szerinti sorszámát az összes prioritás között, vagyis ha $A[i]$ prioritása az összes prioritás közül a j-edik legkisebb, akkor legyen r_i értéke j. Ha X_i-vel jelöljük azt az eseményt, hogy az $A[i]$ elem éppen a $\sigma(i)$-edik legkisebb prioritást kapja (vagyis $r_i = \sigma(i)$), akkor a bizonyítás módosítás nélkül végigvihető. Vagyis ha ki akarjuk számolni annak a valószínűségét, hogy egy adott σ permutációt kapunk, akkor pontosan a fenti számolást kell megismételni, így ennek a permutációnak a valószínűségére is $1/n!$ adódik. ■

Azt gondolhatnánk, hogy annak igazolására, hogy minden permutáció azonos valószínűséggel áll elő, elegendő csak azt belátni, hogy minden $A[i]$ elem pontosan $1/n$ valószínűséggel kerülhet egy tetszőleges j helyre. Az 5.3-4. gyakorlatban megmutatjuk, hogy ez a gyengébb feltétel valójában nem elégséges.

Jobb módszert kapunk egy véletlen permutáció előállítására, ha a tömb elemeit véletlenszerűen cserélgetjük. A PERMUTÁCIÓ-CSERÉKKEL eljárás egy véletlen permutációt állít elő $O(n)$ időben úgy, hogy csak cseréket végez, a bemeneti tömbön kívül nem igényel több memóriát. A ciklus i-edik iterációjában az $A[i]$ elemet véletlenszerűen kicseréljük az $A[i]$, $A[i+1]$, ..., $A[n]$ elemek egyikére. A további iterációkban az $A[i]$ elemhez többször már nem nyúlunk.

```
PERMUTÁCIÓ-CSERÉKKEL(A)
1  n ← hossz[A]
2  for i ← 1 to n
3        do cseréljük meg az A[i] és az A[VÉLETLEN(i, n)] elemeket
```

Egy ciklusinvariáns feltétel igazolásával fogjuk bizonyítani, hogy a PERMUTÁCIÓ-CSERÉKKEL eljárás egyenletes eloszlású véletlen permutációt állít elő. Adott n elem k-ad osztályú permutációja alatt az elemeknek egy k hosszú ismétlődés nélküli sorozatát értjük, tudjuk, hogy n elemnek $n!/(n-k)!$ különböző k-ad osztályú permutációja van (lásd a C. függeléket).

5.5. lemma. *A* PERMUTÁCIÓ-CSERÉKKEL *eljárás egy egyenletes eloszlású véletlen permutációt állít elő.*

Bizonyítás. A következő ciklusinvariáns feltételt fogjuk használni:

> A 2–3. sorokban lévő **for** ciklus i-edik iterációja előtt minden lehetséges $(i-1)$-ed osztályú permutáció pontosan $(n-i+1)!/n!$ valószínűséggel áll elő az $A[1\,..\,i-1]$ résztömbben.

Először ellenőriznünk kell, hogy az invariáns feltétel teljesül a ciklus első iterációja előtt, majd igazolnunk kell, hogy a ciklus minden iterációja után továbbra is érvényes marad. Végül meg kell mutatnunk, hogy a ciklus befejeződésekor az invariáns feltétel segítségével igazolhatjuk az algoritmus helyességét.

Teljesül: Tekintsük az első iteráció előtti helyzetet, ekkor i értéke 1. A ciklusinvariáns feltétel azt állítja, hogy minden lehetséges 0-ad osztályú permutáció $(n-i+1)!/n! = n!/n! = 1$ valószínűséggel áll elő az $A[1\,..\,0]$ résztömbben. Az $A[1\,..\,0]$ résztömb egy üres tömb és a 0-ad osztályú permutáció nem tartalmaz elemeket. Vagyis az $A[1\,..\,0]$ résztömb

minden 0-ad osztályú permutációt 1 valószínűséggel tartalmaz, tehát a ciklusinvariáns feltétel teljesül az első iteráció előtt.

Megmarad: Tegyük fel, hogy a ciklus i-edik iterációja előtt minden lehetséges $(i-1)$-ed osztályú permutáció pontosan $(n-i+1)!/n!$ valószínűséggel áll elő az $A[1\,..\,i-1]$ résztömbben. Meg kell mutatnunk, hogy a ciklus i-edik iterációja után minden lehetséges i-ed osztályú permutáció pontosan $(n-i)!/n!$ valószínűséggel áll elő az $A[1\,..\,i]$ résztömbben. Ha ez teljesül, akkor i értékének növelése után a következő iterációban is teljesülni fog az invariáns feltétel.

Tekintsük az i-edik iterációt. Rögzítsünk egy tetszőleges i-edrendű permutációt, az elemeit jelölje $\langle x_1, x_2, \ldots, x_i\rangle$. A permutációt úgy is tekinthetjük, mint az $(i-1)$-ed osztályú $\langle x_1, \ldots, x_{i-1}\rangle$ permutációt, amit még egy elem, x_i, követ. Legyen E_1 az az esemény, hogy az első $(i-1)$ iteráció után az $A[1\,..\,i-1]$ résztömbben pont az $\langle x_1, \ldots, x_{i-1}\rangle$ $(i-1)$-edrendű permutáció állt elő. A ciklusinvariáns feltétel miatt $\Pr\{E_1\} = (n-i+1)!/n!$ teljesül. Legyen E_2 az az esemény, hogy az i-edik iteráció az x_i elemet helyezi az $A[i]$ helyre. Az $A[1\,..\,n]$ résztömbben pontosan akkor áll elő az $\langle x_1, x_2, \ldots, x_i\rangle$ i-ed osztályú permutáció, ha E_1 és E_2 mindkettő bekövetkezik, vagyis $\Pr\{E_2 \cap E_1\}$ értékét szeretnénk meghatározni. A (C.14) egyenlőség alapján

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\}\Pr\{E_1\}\,.$$

A $\Pr\{E_2 \mid E_1\}$ valószínűség értéke $1/(n-i+1)$, mivel az algoritmus 3. sora az $A[i\,..\,n]$ résztömb $n-i+1$ eleme közül véletlenszerűen választja ki az $A[i]$ helyre kerülőt. Ezért

$$\begin{aligned}\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\}\Pr\{E_1\}\\ &= \frac{1}{n-i+1}\cdot\frac{(n-i+1)!}{n!}\\ &= \frac{(n-i)!}{n!}\,.\end{aligned}$$

Befejeződik: Az eljárás befejeződésekor $i = n+1$, így a ciklusinvariáns feltétel alapján minden n-ed osztályú permutáció pontosan $(n-n)!/n! = 1/n!$ valószínűséggel áll elő az $A[1\,..\,n]$ tömbben.

Ezzel bizonyítottuk, hogy a PERMUTÁLÁS-CSERÉKKEL eljárás egyenletes eloszlású véletlen permutációt állít elő. ■

Gyakran egy véletlenített algoritmus adja a feladatra a legegyszerűbb és leghatékonyabb megoldást. A könyvben több helyen fogunk még véletlenített algoritmusokkal találkozni.

Gyakorlatok

5.3-1. Marceau professzornak kifogásai vannak az 5.5. lemmában használt ciklusinvariáns feltétellel kapcsolatban. Szerinte a feltétel nem teljesül az első iteráció előtt. Azt mondja ugyanis, hogy az üres résztömböt akár úgy is tekinthetjük, hogy nem tartalmaz egyetlen 0-ad osztályú permutációt sem. Ekkor viszont az üres tömb 0 valószínűséggel tartalmaz minden 0-ad osztályú permutációt, vagyis a ciklusinvariáns feltétel nem igaz az első iteráció előtt. Írjuk át a PERMUTÁLÁS-CSERÉKKEL eljárást úgy, hogy az új ciklusinvariáns feltétel az első

iteráció előtt ne egy üres résztömbre vonatkozzon. Módosítsuk az 5.5. lemma bizonyítását az új eljárásnak megfelelően.

5.3-2. Kelp professzor olyan eljárást szeretne írni, ami az identitás permutáció kivételével bármelyik permutációt elő tudja állítani véletlenszerűen. A következő eljárást javasolja:

NEM-IDENTITÁS-PERMUTÁCIÓ(A)

```
1  n ← hossz[A]
2  for i ← 1 to n − 1
3       do cseréljük meg az A[i] és az A[VÉLETLEN(i + 1, n)] elemeket
```

Tényleg azt teszi ez az eljárás, amit Kelp professzor szeretne?

5.3-3. Módosítsuk úgy az algoritmust, hogy az $A[i]$ elemet nem az $A[i\,..\,n]$ résztömb egy véletlen elemével cseréljük meg, hanem a teljes tömbből választunk véletlenül:

PERMUTÁCIÓ-MINDENKIVEL-CSERÉL(A)

```
1  n ← hossz[A]
2  for i ← 1 to n
3       do cseréljük meg az A[i] és az A[VÉLETLEN(1, n)] elemeket
```

Igaz-e, hogy ez az eljárás egyenletes eloszlású véletlen permutációt ad? Bizonyítsuk be, vagy cáfoljuk meg az állítást.

5.3-4. Amstrong professzor a következő eljárást javasolja egyenletes eloszlású véletlen permutációk előállítására:

PERMUTÁCIÓ-FORGATÁSSAL(A)

```
1  n ← hossz[A]
2  eltolás ← VÉLETLEN(1, n)
3  for i ← 1 to n
4       do cél ← i + eltolás
5          if cél > n
6             then cél ← cél − n
7          B[cél] ← A[i]
8  return B
```

Igazoljuk, hogy egy adott $A[i]$ elem pontosan $1/n$ valószínűséggel kerülhet a B tömb bármely $B[j]$ pozíciójába. Mutassuk meg, hogy Armstrong professzor téved, az el őállított permutációk nem egyenletes eloszlásúak.

5.3-5.$\star$ Bizonyítsuk be, hogy a PERMUTÁCIÓ-RENDEZÉSSEL eljárásban legalább $1 - 1/n$ annak a valószínűsége, hogy a P tömb csupa különböző elemből áll.

5.3-6. Hogyanlehet úgy megvalósítani a PERMUTÁCIÓ-RENDEZÉSSEL eljárást, hogy abban az esetben is működjön, ha van több azonos prioritás? Az algoritmusnak akkor is egyenletes eloszlású véletlen permutációt kell adnia, ha a prioritások között vannak azonosak.

⋆ 5.4. További példák valószínűségi elemzésre és az indikátor valószínűségi változók használatára

Ebben az alfejezetben a valószínűségi elemzést fogjuk négy példán keresztül szemléltetni. Az elsőben annak a valószínűségét határozzuk meg, hogy egy szobában lévő k ember között található olyan pár, akiknek megegyezik a születésnapjuk. A második példában golyóknak urnákban való véletlenszerű elrendezéseit fogjuk vizsgálni. A harmadikban pedig egymás után következő fejek *futamait* fogjuk tanulmányozni az érmedobás során. Végül a negyedikben a munkatársfelvétel problémának egy olyan változatát vizsgáljuk, ahol a legjobb jelöltet úgy próbáljuk kiválasztani, hogy valójában nem is ismerjük az összes jelentkezőt.

5.4.1. A születésnap-paradoxon

A klasszikus ***születésnap-paradoxon*** egy jó példa a valószínűségi érvelés szemléltetésére. Hány embernek kell lennie egy szobában ahhoz, hogy legalább 50% eséllyel legyen két olyan közöttük, akik az évnek ugyanazon a napján születtek? A válasz: meglepően kevésnek. A paradoxon az, hogy valójában jóval kevesebb, mint ahány napja az évnek van, ahogy egyébként gondolnánk, sőt ennek felénél is jóval kevesebb.

Hogy a kérdésre választ adjunk, indexeljük a szobában lévő embereket az $1, 2, \ldots, k$ egészekkel, ahol k a szobában lévő emberek száma. A szökőéveket figyelmen kívül hagyva feltesszük, hogy minden év $n = 365$ napból áll. Legyen $i = 1, 2, \ldots, k$ esetén b_i ($1 \le b_i \le 365$) az a napja az évnek, amelyre az i-edik ember születésnapja esik. Szintén feltételezzük, hogy a születésnapok egyenletesen oszlanak el az év n napján, így $\Pr\{b_i = r\} = 1/n$ minden $i = 1, 2, \ldots, k$ és $r = 1, 2, \ldots, n$ esetén.

Annak a valószínűsége, hogy két embernek, i-nek és j-nek, megegyezik a születésnapja, függ attól, hogy a születésnapok véletlenszerű megválasztása vajon függetlennek tekinthető-e. Ha a születésnapok függetlenek, akkor annak a valószínűsége, hogy i-nek és j-nek a születésnapja egyaránt az r-edik napra esik

$$\begin{aligned}\Pr\left\{b_i = r \text{ és } b_j = r\right\} &= \Pr\{b_i = r\}\Pr\left\{b_j = r\right\} \\ &= \frac{1}{n^2}\,.\end{aligned}$$

Így annak a valószínűsége, hogy mindkettő ugyanarra a napra esik,

$$\begin{aligned}\Pr\left\{b_i = b_j\right\} &= \sum_{r=1}^{n} \Pr\left\{b_i = r \text{ és } b_j = r\right\} \\ &= \sum_{r=1}^{n} \frac{1}{n^2} \\ &= \frac{1}{n}\,. \qquad (5.7)\end{aligned}$$

Szemléletesen, miután b_i-t már megválasztottuk, $1/n$ a valószínűsége annak, hogy b_j-t is ugyanannak választjuk. Így annak a valószínűsége, hogy i-nek és j-nek ugyanaz a születésnapja, megegyezik azzal, hogy egyikük születésnapja egy adott napra esik. Megjegyezzük azonban, hogy ez az egybeesés azon a feltételen múlik, hogy a születésnapok függetlenek.

A komplementer eseményt tekintve megvizsgálhatjuk annak a valószínűségét, hogy k ember közül legalább kettőnek megegyezik a születésnapja. Annak a valószínűsége, hogy legalább két születésnap megegyezik, egyenlő 1 mínusz annak a valószínűsége, hogy az összes születésnap különböző. Ha B_k az az esemény, hogy k embernek különbözik a születésnapja, akkor

$$B_k = \bigcap_{i=1}^{k} A_i,$$

ahol A_i az az esemény, hogy az i-edik személynek más a születésnapja mint a j-edik személynek minden $j < i$ esetén. Mivel $B_k = A_k \cap B_{k-1}$, ezért a (C.16) egyenletből a

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \tag{5.8}$$

rekurziót kapjuk, ahol kezdeti feltételnek a $\Pr\{B_1\} = \Pr\{A_1\} = 1$-et választjuk. Más szóval annak a valószínűsége, hogy $b_1, b_2, \ldots, b_k$ különböző születésnapok, megegyezik annak a valószínűségével, hogy $b_1, b_2, \ldots, b_{k-1}$ különböző születésnapok, szorozva annak a valószínűségével, hogy $b_k \neq b_i$ minden $i = 1, 2, \ldots, k-1$ esetén feltéve, hogy $b_1, b_2, \ldots, b_{k-1}$ különbözők.

Ha $b_1, b_2, \ldots, b_{k-1}$ különbözők, akkor annak a valószínűsége, hogy $b_k \neq b_i$ minden $i = 1, 2, \ldots, k-1$ esetén, egyenlő $(n-k+1)/n$-nel, ugyanis az n nap közül még $n-(k-1)$-et nem választottunk ki. Az (5.8) rekurzív összefüggést ismételve azt kapjuk, hogy

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\} \Pr\{A_{k-1} \mid B_{k-2}\} \Pr\{A_k \mid B_{k-1}\} \\
&\vdots \\
&= \Pr\{B_1\} \Pr\{A_2 \mid B_1\} \Pr\{A_3 \mid B_2\} \cdots \Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right) \cdots \left(1-\frac{k-1}{n}\right).
\end{aligned}$$

A (3.11) egyenlőtlenség, $1 + x \leq e^x$ alapján azt kapjuk, hogy

$$\begin{aligned}
\Pr\{B_k\} &\leq e^{-\frac{1}{n}} e^{-\frac{2}{n}} \cdots e^{-\frac{(k-1)}{n}} \\
&= e^{-\sum_{i=1}^{k-1} \frac{i}{n}} \\
&= e^{-\frac{k(k-1)}{2n}} \\
&\leq \frac{1}{2},
\end{aligned}$$

ahol $-k(k-1)/2n \leq \ln 1/2$. Legfeljebb $1/2$ a valószínűsége annak, hogy az összes k születésnap különböző, ha $k(k-1) \geq 2n \ln 2$, vagy megoldva ezt a másodfokú egyenletet, ha $k \geq (1 + \sqrt{1 + (8 \ln 2)n})/2$. Ez alapján $n = 365$ esetén $k \geq 23$-at kell választanunk. Így ha legalább 23 ember van egy szobában, akkor legalább $1/2$ a valószínűsége annak, hogy legalább két embernek ugyanaz a születésnapja. A Marson, mivel egy év 669 marsi napból áll, 31 marslakó esetén kapjuk ugyanezt az eredményt.

Elemzés indikátor valószínűségi változók használatával

Az indikátor valószínűségi változók módszerével egy egyszerűbb, de csak közelítő analízisét adhatjuk a születésnap-paradoxonnak. A szobában lévő k számú ember minden (i, j) párja esetén $(1 \le i < j \le k)$ definiáljuk az X_{ij} indikátor valószínűségi változót:

$$\begin{aligned} X_{ij} &= \mathrm{I}\{\text{az } i\text{-edik és a } j\text{-edik személynek megegyezik a születésnapja}\} \\ &= \begin{cases} 1, & \text{ha az } i\text{-edik és a } j\text{-edik személynek megegyezik a születésnapja,} \\ 0 & \text{egyébként.} \end{cases} \end{aligned}$$

Az (5.7) egyenletben láttuk, hogy $1/n$ a valószínűsége annak, hogy két embernek megegyezik a születésnapja, így az 5.1. lemma alapján

$$\begin{aligned} \mathrm{E}\left[X_{ij}\right] &= \Pr\{\text{az } i\text{-edik és a } j\text{-edik személynek megegyezik a születésnapja}\} \\ &= \frac{1}{n}\,. \end{aligned}$$

Legyen az X valószínűségi változó értéke azon párok száma, akiknek ugyanaz a születésnapja, ekkor

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}.$$

Mindkét oldal várható értékét véve és kihasználva a várható érték linearitását:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] \\ &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathrm{E}\left[X_{ij}\right] \\ &= \binom{k}{2}\frac{1}{n} \\ &= \frac{k(k-1)}{2n}\,. \end{aligned}$$

Ezért, ha $k(k-1) \ge 2n$, akkor az olyan párok várható száma, akiknek a születésnapjuk megegyezik, legalább 1. Így, ha legalább $\sqrt{2n}+1$ személy van egy szobában, akkor számíthatunk arra, hogy legalább kettőnek közülük közös a születésnapja. Esetünkben $n = 365$, ezért ha $k = 28$, akkor az olyan párok várható száma, akiknek megegyezik a születésnapjuk, $(28 \cdot 27)/(2 \cdot 365) \approx 1{,}0356$. Így legalább 28 embernél már számíthatunk arra, hogy találunk legalább egy olyan párt, akiknek megegyezik a születésnapjuk. A Marson, ahol egy év 669 marsi napból áll, ehhez legalább 38 marslakó szükséges.

Az első esetben az ahhoz szükséges emberek számát határoztuk meg, hogy legalább $1/2$ legyen a valószínűsége annak, hogy két egyező születésnap létezzen, módszerünk a valószínűségek pontos meghatározása volt. A második esetben pedig indikátor valószínűségi változók használatával az ahhoz szükséges számot adtuk meg, amelynél a közös születésnappárok várható száma 1. Bár az emberek száma a két esetben eltér egymástól, a viselkedésük azonban aszimptotikusan ugyanaz: $\Theta(\sqrt{n})$.

5.4.2. Golyók és urnák

Tekintsük azt a kísérletsorozatot, amely során véletlenszerűen dobunk ugyanolyan golyókat b számú urnába, amelyek az $1, 2, \ldots, b$ számokkal vannak megjelölve. A dobások függetlenek egymástól, és minden egyes dobásnál a golyó ugyanolyan valószínűséggel kerül bármelyik urnába. Annak a valószínűsége, hogy egy eldobott golyó egy adott urnába esik, $1/b$. Így a golyódobás kísérletsorozata Bernoulli-kísérleteknek (C.4. függelék) egy olyan sorozata, ahol a jó eset valószínűsége $1/b$, ahol a jó eset azt jelenti, hogy a golyó a megadott urnába esik. Ez a modell nagyon jól használható például hasító táblák teljesítményének vizsgálatára (11. fejezet). Számos érdekes kérdést tehetünk fel a golyódobás kísérletsorozatával kapcsolatban (a C-1. feladatban további lehetséges kérdéseket vizsgálunk):

Hány golyó esik egy adott urnába? Azoknak a golyóknak a száma, amelyek egy adott urnába esnek, $b(k; n, 1/b)$ binomiális eloszlást követ. Ha n darab golyóval dobunk, akkor a (C.36) egyenlet alapján n/b azoknak a golyóknak a várható száma, amelyek az adott urnába esnek.

Átlagosan mennyi golyóval kell dobni ahhoz, hogy egy adott urna tartalmazzon egy golyót? Az ahhoz szükséges dobások száma, hogy az adott urnába egy golyó kerüljön, $1/b$ paraméterű geometriai eloszlást követ, és így a (C.31) egyenlet alapján a dobások várható száma, amíg a jó eset be nem következik, $1/(1/b) = b$.

Hány golyóval kell dobni ahhoz, hogy minden urnába kerüljön legalább egy golyó? Nevezzünk *találatnak* egy olyan dobást, amely során a golyó egy üres urnába esik. Szeretnénk meghatározni a b számú találat eléréséhez szükséges dobások n átlagos számát.

A találatokat felhasználhatjuk arra, hogy az n dobást blokkokra bontsuk. Az i-edik blokk az $(i-1)$-edik találat utáni dobásokból áll egészen az i-edik találattal bezárólag. Az első blokk az első dobásból áll, hiszen amikor még minden urna üres, akkor biztosak lehetünk abban, hogy találatot kapunk. Az i-edik blokk minden egyes dobásánál $i-1$ számú urna tartalmaz golyót, és $b-i+1$ számú urna üres. Így az i-edik blokk minden dobásánál $(b-i+1)/b$ a valószínűsége annak, hogy találatot kapunk.

Jelölje n_i a dobások számát az i-edik blokkban. Az ahhoz szükséges dobások száma, hogy b találatunk legyen, így $n = \sum_{i=1}^{b} n_i$. Mindegyik n_i valószínűségi változó olyan geometriai eloszlás, ahol a jó eset valószínűsége $(b-i+1)/b$, ezért a (C.31) egyenlet alapján

$$\mathrm{E}[n_i] = \frac{b}{b-i+1} .$$

A várható érték linearitása alapján tehát

$$\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b\sum_{i=1}^{b} \frac{1}{i} \\
&= b(\ln b + O(1)).
\end{aligned}$$

Az utolsó sor a harmonikus sorra adott (A.7) becslésből következik. Ezért közelítőleg $b \ln b$ számú dobás után várhatjuk, hogy minden urna tartalmaz golyót. Ez a probléma ***jutalomszelvény-gyűjtési probléma*** néven is ismert: ha a b fajta szelvény mindegyikét be akarjuk gyűjteni, akkor ehhez körülbelül $b \log b$ véletlenszerűen választott szelvényt kell szereznünk.

5.4.3. Futamok

Dobjunk fel egy szabályos érmét n-szer. Hány egymás után következő fejből áll az a leghosszabb futam, amelyet várhatóan látni fogunk? A válasz, amint azt az alábbi analízis mutatja, $\Theta(\lg n)$.

Először bebizonyítjuk, hogy a fejekből álló leghosszabb futam hossza $O(\lg n)$. Minden dobás $1/2$ valószínűséggel lesz fej. Legyen A_{ik} az az esemény, hogy egy legalább k hosszú, fejekből álló futam kezdődik az i-edik dobásnál, vagy pontosabban, az az esemény, hogy az i-edik, $(i+1)$-edik, $\ldots$, $(i+k-1)$-edik egymás után következő k darab dobás mindegyike fej lesz, ahol $1 \le k \le n$ és $1 \le i \le n-k+1$. Mivel a dobások függetlenek egymástól, ezért tetszőleges A_{ik} eseménynél annak a valószínűsége, hogy mind a k dobás fej lesz

$$\Pr\{A_{ik}\} = \frac{1}{2^k}. \tag{5.9}$$

Így tehát $k = 2\lceil \lg n \rceil$ esetén

$$\begin{aligned} \Pr\left\{A_{i,2\lceil \lg n \rceil}\right\} &= \frac{1}{2^{2\lceil \lg n \rceil}} \\ &\le \frac{1}{2^{2 \lg n}} \\ &= \frac{1}{n^2}, \end{aligned}$$

vagyis egészen kicsi a valószínűsége annak, hogy egy legalább $2\lceil \lg n \rceil$ hosszú, fejekből álló futam kezdődik az i-edik dobásnál. Egy ilyen hosszúságú futam $n - 2\lceil \lg n \rceil + 1$ különböző dobás valamelyikénél kezdődhet. Ezért annak a valószínűsége, hogy egy legalább $2\lceil \lg n \rceil$ hosszú, fejekből álló futam kezdődik valahol

$$\begin{aligned} \Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \frac{1}{n^2} \\ &< \sum_{i=1}^{n} \frac{1}{n^2} \\ &= \frac{1}{n}, \end{aligned} \tag{5.10}$$

ugyanis a (C.18) Boole-egyenlőtlenség alapján események uniójának a valószínűsége nem lehet nagyobb az események egyenkénti valószínűségeinek az összegénél. (Megjegyezzük, hogy a Boole-egyenlőtlenség olyan eseményekre is fennáll, amelyek, akár a fentiek, nem függetlenek.)

Az (5.10) egyenlőtlenség segítségével korlátot adunk a leghosszabb futam hosszára. Legyen L_j az az esemény, hogy a fejekből álló leghosszabb futam pontosan j hosszúságú ($j = 0, 1, 2, \ldots, n$), és legyen L a leghosszabb futam hossza. A várható érték definíciója alapján

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\left\{L_j\right\}. \tag{5.11}$$

A fenti összeg minden $\Pr\left\{L_j\right\}$ tagját felülről becsülhetjük az (5.10) egyenlőtlenség segítségével, így az összegre is kaphatunk egy felső becslést. Sajnos ezzel a módszerrel túl gyenge korlátot kapunk. A fenti gondolatmenetből mégis ki lehet hozni egy jó felső korlátot, a következőképpen. Informálisan fogalmazva azt mondhatjuk, hogy az (5.11) egyenletben az összegzés semelyik tagjában sem lehet j és $\Pr\left\{L_j\right\}$ egyszerre nagy. Miért? Ha $j \geq 2\lceil \lg n \rceil$, akkor $\Pr\left\{L_j\right\}$ nagyon kicsi, ha $j < 2\lceil \lg n \rceil$, akkor j nem túl nagy. Formálisan, vegyük észre, hogy az L_j események minden $j = 0, 1, \ldots, n$ esetén diszjunktak, így $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\left\{L_j\right\}$ annak a valószínűsége, hogy egy legalább $2\lceil \lg n \rceil$ fejből álló sorozat kezdődik valahol. Az (5.10) egyenlőtlenség alapján $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\left\{L_j\right\} < 1/n$. Továbbá vegyük észre, hogy $\sum_{j=0}^{n} \Pr\left\{L_j\right\} = 1$ miatt $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\left\{L_j\right\} \leq 1$ következik. Így tehát azt kapjuk, hogy

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\left\{L_j\right\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\left\{L_j\right\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\left\{L_j\right\} \\
&< \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \lg n \rceil) \Pr\left\{L_j\right\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\left\{L_j\right\} \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\left\{L_j\right\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\left\{L_j\right\} \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot \frac{1}{n} \\
&= O(\lg n).
\end{aligned}$$

Annak az esélye, hogy egy fejekből álló futam túllépi az $r\lceil \lg n \rceil$ dobásszámot, gyorsan csökken r függvényében. Tetszőleges $r \geq 1$ esetén annak a valószínűsége, hogy egy $r\lceil \lg n \rceil$ számú fejből álló futam kezdődik az i-edik dobásnál,

$$\begin{aligned}
\Pr\left\{A_{i, r\lceil \lg n \rceil}\right\} &= \frac{1}{2^{r\lceil \lg n \rceil}} \\
&\leq \frac{1}{n^r}.
\end{aligned}$$

Így legfeljebb $n/n^r = 1/n^{r-1}$ a valószínűsége annak, hogy a leghosszabb futam legalább $r\lceil \lg n \rceil$ hosszú, vagy ezzel ekvivalens módon, $1 - 1/n^{r-1}$ a valószínűsége annak, hogy a leghosszabb futam hossza kisebb mint $r\lceil \lg n \rceil$.

Például $n = 1000$-szer feldobva az érmét legfeljebb $1/n = 1/1000$ a valószínűsége annak, hogy egy legalább $2\lceil \lg n \rceil = 20$ fejből álló futamot kapunk. Legfeljebb $1/n^2 = 1/1\,000\,000$ az esélye annak, hogy egy $3\lceil \lg n \rceil = 30$ fejnél hosszabb futamot kapunk.

Ezután az alsó korlátot bizonyítjuk: egy érmének n-szeri feldobása esetén a fejekből álló leghosszabb futam várható hossza $\Omega(\lg n)$. Ennek a becslésnek a bizonyításához s hosszúságú futamokat fogunk keresni úgy, hogy az n dobást nagyjából n/s darab s hosszú csoportra osztjuk. Megmutatjuk, hogy az $s = \lfloor (\lg n)/2 \rfloor$ választás esetén nagy valószínűséggel az egyik ilyen csoport csupa fejet tartalmaz, vagyis nagy valószínűséggel a leghosszabb futam hossza legalább $s = \Omega(\lg n)$. Ennek segítségével bizonyíthatjuk, hogy a leghosszabb sorozat várható hosszúsága $\Omega(\lg n)$.

Az n darab dobást felbonthatjuk $\lfloor (\lg n)/2 \rfloor$ egymás utáni dobás legalább $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ számú csoportjára, megbecsüljük annak a valószínűségét, hogy egyik csoport sem áll csupa fejből. Az (5.9) egyenlet alapján annak a valószínűsége, hogy az i-edik dobással kezdődő csoportban csupa fej van, pontosan

$$\begin{aligned} \Pr\left\{A_{i,\lfloor (\lg n)/2 \rfloor}\right\} &= \frac{1}{2^{\lfloor (\lg n)/2 \rfloor}} \\ &\geq \frac{1}{\sqrt{n}} . \end{aligned}$$

Ezért legfeljebb $1 - 1/\sqrt{n}$ a valószínűsége annak, hogy az i-edik dobásnál nem kezdődik olyan fejekből álló futam, amelynek hossza legalább $\lfloor \lg n \rfloor / 2$. Mivel az $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ csoportok független dobásokból állnak, ezért annak a valószínűsége, hogy ezen csoportok egyike sem $\lfloor \lg n \rfloor / 2$ hosszúságú futam, legfeljebb

$$\begin{aligned} \left(1 - \frac{1}{\sqrt{n}}\right)^{\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor} &\leq \left(1 - \frac{1}{\sqrt{n}}\right)^{n/\lfloor (\lg n)/2 \rfloor - 1} \\ &\leq \left(1 - \frac{1}{\sqrt{n}}\right)^{2n/\lg n - 1} \\ &\leq e^{-(2n/\lg n - 1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O\left(\frac{1}{n}\right). \end{aligned}$$

Érvelésünk során felhasználtuk az $1 + x \leq e^x$ (3.11) egyenlőtlenséget, és azt a könnyen igazolható tényt, hogy $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$, ha n elég nagy.

Így annak a valószínűsége, hogy a leghosszabb futam meghaladja a $\lfloor (\lg n)/2 \rfloor$ hosszúságot legalább

$$\sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \Pr\left\{L_j\right\} \geq 1 - O\left(\frac{1}{n}\right). \tag{5.12}$$

Szeretnénk alsó korlátot adni a leghosszabb futam hosszának várható értékére. A számolás hasonlóan történik, mint ahogy az a felső korlát esetén történt, az (5.11) egyenlőségből indulunk ki:

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j\Pr\left\{L_j\right\} \\
&= \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} j\Pr\left\{L_j\right\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} j\Pr\left\{L_j\right\} \\
&\geq \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} 0\cdot\Pr\left\{L_j\right\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \left\lfloor\frac{\lg n}{2}\right\rfloor\Pr\left\{L_j\right\} \\
&= 0\cdot\sum_{j=0}^{\lfloor(\lg n)/2\rfloor}\Pr\left\{L_j\right\} + \left\lfloor\frac{\lg n}{2}\right\rfloor\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n}\Pr\left\{L_j\right\} \\
&\geq 0 + \left\lfloor\frac{\lg n}{2}\right\rfloor\left(1 - O\left(\frac{1}{n}\right)\right) \qquad \text{((5.12) egyenlőtlenség)} \\
&= \Omega(\lg n).
\end{aligned}$$

A születésnap paradoxonhoz hasonlóan az indikátor valószínűségi változók segítségével itt is kaphatunk egy egyszerűbb, de csak közelítő eredményt. Az $X_{ik} = \mathrm{I}\{A_{ik}\}$ indikátor valószínűségi változó tartozzon ahhoz az eseményhez, hogy egy fejekből álló legalább k hosszú futam kezdődik az i-edik dobásnál. Az ilyen hosszúságú futamok számát az X valószínűségi változó adja meg:

$$X = \sum_{i=1}^{n-k+1} X_{ik}.$$

A várható érték linearitását felhasználva azt kapjuk, hogy

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1}\mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{n-k+1}\Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1}\frac{1}{2^k} \\
&= \frac{n-k+1}{2^k}\,.
\end{aligned}$$

Ebből a képletből behelyettesítéssel tetszőleges k értékre meghatározhatjuk a k hosszú futamok várható számát. Ha a kapott szám nagy (jóval nagyobb mint 1), akkor sok k hosszú futam várható, és nagy a valószínűsége annak, hogy legalább egy lesz. Ha a kapott szám kicsi (jóval kisebb mint 1), akkor kevés k hosszú futam várható, és így kicsi a valószínűsége annak, hogy lesz egyáltalán ilyen futam. Ha $k = c\lg n$ valamilyen pozitív c értékre, akkor

$$\begin{aligned} \mathrm{E}[X] &= \frac{n - c\lg n + 1}{2^{c\lg n}} \\ &= \frac{n - c\lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c\lg n - 1)/n}{n^{c-1}} \\ &= \Theta\left(\frac{1}{n^{c-1}}\right). \end{aligned}$$

Ha c nagy, akkor a $c\lg n$ hosszú futamok száma nagyon kicsi, és nagyon valószínűtlen, hogy akár egy is lesz. Másrészt viszont ha $c < 1/2$, akkor $\mathrm{E}[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, és így azt várjuk, hogy sok ilyen hosszúságú futamot találunk. Ekkor persze nagyon valószínű, hogy lesz legalább egy ilyen futam. Ezekből a durva becslésekből már bizonyíthatjuk, hogy a leghosszabb futam várható hossza $\Theta(\lg n)$.

5.4.4. A munkatársfelvétel probléma on-line változata

Befejező példaként tekintsük a munkatársfelvétel probléma következő változatát. Most úgy próbáljuk megtalálni a legalkalmasabb jelöltet, hogy valójában nem is találkozunk mindegyikükkel. További különbség, hogy csak egyetlen jelöltet veszünk fel, nem azt tesszük, mint korábban, hogy az érkező egyre jobb munkatársakat mind felvesszük, elbocsátva a korábbi legjobbat. Csak egy jelöltet vehetünk fel, de most azzal is megelégszünk, ha nem a legjobbat alkalmazzuk, hanem egy olyat, aki majdnem a legjobb. Cégünknél az a szokás, hogy a felvételi beszélgetés után vagy rögtön fel kell vennünk a jelöltet, vagy meg kell mondanunk neki, hogy nem fogjuk alkalmazni. Egyrészt szeretnénk minél jobb jelöltet találni, másrészt viszont szeretnénk minél kevesebb jelöltet behívni. Azt fogjuk vizsgálni, hogy hogyan tudunk olyan módszert találni, ami mindkét feltételnek valamilyen mértékben eleget tesz.

A problémát a következőképpen modellezhetjük. A felvételi beszélgetés után a jelöltnek adunk egy pontszámot, ami azt fejezi ki, hogy mennyire alkalmas a feladatra. Legyen *pontszám*(i) az i-edik jelölt pontszáma, feltesszük, hogy semelyik két jelölt sem kapta ugyanazt a pontszámot. Az első j jelölttel való találkozás után meg tudjuk mondani, hogy ebből a j jelentkezőből ki kapja a legmagasabb pontszámot, de azt nem tudhatjuk, hogy lesz-e a hátralevő $n-j$ jelölt között olyan, aki ennél magasabb pontszámot kap. A következő módszert fogjuk alkalmazni: kiválasztunk egy $k < n$ pozitív egész számot, elbeszélgetünk az első k jelentkezővel, mindet elutasítjuk, majd felvesszük az első olyan jelöltet, aki magasabb pontot kap, mint az összes őt megelőző. Ha kiderül, hogy a legjobb jelentkező az első k jelölt között volt, akkor az utolsó, n-edik jelöltet fogjuk felvenni. Ezt a stratégiát az ON-LINE-MAXIMUM(k, n) eljárás írja le formálisan. Az eljárás annak a jelöltnek a sorszámát adja vissza, akit fel akarunk venni.

```
ON-LINE-MAXIMUM(k, n)

1  legjobb ← −∞
2  for i ← 1 to k
3       if pontszám(i) > legjobb
4            then legjobb ← pontszám(i)
```

```
5  for i← k + 1 to n
6       if pontszám(i) > legjobb
7          then return i
8  return n
```

Minden lehetséges k értékre meg szeretnénk határozni annak a valószínűségét, hogy a fenti módszer a jelöltet választja. Ezután a módszert azzal a k értékkel fogjuk használni, amely a legnagyobb valószínűséget adja. Tegyük fel először, hogy k értéke rögzítve van. Legyen $M(j) = \max_{1 \le i \le j}\{pontszám(i)\}$ az első j jelölt közül a legjobbnak a pontszáma. Legyen S az az esemény, hogy sikerült a legjobb jelöltet felvenni, továbbá legyen S_i az az esemény, hogy sikerült felvenni a legjobb jelöltet, aki pont az i-edik jelölt volt. Mivel az S_i események diszjunktak, ezért $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Ha a legjobb jelentkező az első k között volt, akkor biztos, hogy módszerünkkel őt nem vesszük fel, így $\Pr\{S_i\} = 0$, ha $i = 1, 2, \ldots, k$. Ebből következően

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} . \tag{5.13}$$

Most meghatározzuk $\Pr\{S_i\}$ értékét. Két dolognak kell teljesülnie ahhoz, hogy az i-edik helyen álló legjobb jelöltet vegyük fel. Egyrészt a legjobb jelöltnek az i-edik helyen kell lennie, jelöljük ezt az eseményt B_i-vel. Másrészt az algoritmusnak nem szabad felvenni egyik jelöltet sem a $(k + 1)$-ediktől az $(i - 1)$-edikig. Ez pontosan akkor lesz igaz, ha a 6. sorban $pontszám(j) < legjobb$ teljesül minden $k + 1 \le j \le i - 1$ feltételt kielégítő j esetén. (Mivel a pontszámok különböznek, ezért a $pontszám(j) = legjobb$ esettel nem kell foglalkoznunk.) Vagyis annak kell teljesülnie, hogy a $pontszám(k+1)$-től a $pontszám(i-1)$-ig mindegyik érték kisebb legyen $M(k)$-nál: ha ugyanis valamelyik nagyobb lenne, akkor az algoritmus azt a jelöltet választaná. Jelöljük O_i-vel azt az eseményt, hogy a $(k + 1)$-ediktől az $(i - 1)$-edikig egyik jelöltet sem vesszük fel. Szerencsés módon a B_i és az O_i események függetlenek. Az O_i esemény ugyanis csak az első $i - 1$ érték egymás közötti sorrendjétől függ, míg B_i csak attól függ, hogy az i-edik érték nagyobb-e az összes korábbinál. Az első $i - 1$ érték sorrendje nem befolyásolja azt, hogy az i-edik érték-e a legnagyobb, és az i-edik helyen álló érték nem befolyásolja az első $i - 1$ elem sorrendjét. Így a (C.15) egyenlőséget alkalmazva

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\} \Pr\{O_i\} .$$

A $\Pr\{B_i\}$ valószínűség értéke nyilván $1/n$, hiszen a maximális érték azonos valószínűséggel lehet az n pozíció bármelyikén. Az O_i esemény bekövetkezéséhez az szükséges, hogy az első $i - 1$ érték közül a legnagyobb az első k pozíció valamelyikén legyen. Mivel ez a legnagyobb érték az első $i - 1$ pozíció bármelyikén azonos valószínűséggel lehet, így $\Pr\{O_i\} = k/(i - 1)$ és $\Pr\{S_i\} = k/(n(i - 1))$ adódik. Az (5.13) egyenletet felhasználva

$$\begin{aligned} \Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\ &= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \end{aligned}$$

$$= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1}$$
$$= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} .$$

Az összeget integrálással becsülhetjük alulról és felülről. Felhasználva az (A.12) egyenlőtlenséget:

$$\int_k^n \frac{1}{x}\,dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}\,dx.$$

Kiszámítva a határozott integrálok értékét

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)),$$

adódik, ami elég pontos becslés a $\Pr\{S\}$ valószínűségre. Szeretnénk maximalizálni a legjobb jelölt kiválasztásának a valószínűségét, ezért meghatározzuk azt a k értéket, amelynél a fenti egyenlőtlenségből a legnagyobb alsó korlát adódik a $\Pr\{S\}$ valószínűségre. (Jelen esetben az alsó korlátot könnyebb maximalizálni, mint a felső korlátot.) Ha a $(k/n)(\ln n - \ln k)$ kifejezést k szerint deriváljuk, akkor

$$\frac{1}{n}(\ln n - \ln k - 1)$$

adódik. A derivált értéke $\ln k = \ln n - 1 = \ln(n/e)$ esetén lesz nulla, tehát az alsó korlát $k = n/e$ esetén lesz maximális. Vagyis ha módszerünket a $k = n/e$ választással alkalmazzuk, akkor legalább $1/e \approx 0{,}368$ valószínűséggel a legjobb jelöltet fogjuk felvenni.

Gyakorlatok

5.4-1. Hány embernek kell lennie egy szobában ahhoz, legalább $1/2$ legyen annak a valószínűsége, hogy valakinek a születésnapja megegyezzen az Olvasó születésnapjával? Hány embernek kell lennie egy szobában ahhoz, hogy legalább $1/2$ valószínűséggel legyen két ember, akinek március 15-ére esik a születésnapja?

5.4-2. Golyókat dobunk n darab urnába, ahol a dobások függetlenek egymástól, és a golyók egyenlő valószínűséggel esnek bármelyik urnába. Várhatóan hány dobás kell ahhoz, hogy legyen egy olyan urna, amelybe legalább két golyót dobtunk?

5.4-3.⋆ Lényeges-e a születésnap paradoxon vizsgálatánál, hogy a születésnapok teljesen függetlenek, vagy a páronkénti függetlenség már elegendő? Indokoljuk meg válaszunkat.

5.4-4.⋆ Hány embert kell meghívni egy partira ahhoz, hogy jó eséllyel legyen közöttük három olyan, akiknek megegyezik a születésnapja?

5.4-5.⋆ Mennyi a valószínűsége annak, hogy egy n elemű halmaz feletti k hosszú string egy k-ad osztályú permutáció? Hogyan kapcsolódik ez a kérdés a születésnap paradoxonhoz?

5.4-6.⋆ Tegyük fel, hogy n golyót dobunk n darab urnába, ahol a dobások függetlenek egymástól, és a golyók egyenlő valószínűséggel esnek bármelyik urnába. Mennyi az üres urnák várható száma? Mennyi azoknak az urnáknak a várható száma, amelyek pontosan egy golyót tartalmaznak?

5.4-7.⋆ Élesítsük a futamok hosszára adott alsó becslést megmutatva, hogy egy szabályos érme n-szeri feldobása során kevesebb mint $1/n$ a valószínűsége annak, hogy nem fordul elő egymás utáni fejeknek $\lg n - 2\lg\lg n$-nél hosszabb futama.

Feladatok

5-1. Valószínűségi leszámlálás

Egy b bites számlálóval csak $2^b - 1$-ig tudunk elszámolni a szokásos módon. R. Morris ***valószínűségi leszámlálás***ával nagyobb értékig is el tudunk számolni azon az áron, hogy a pontosságból veszítünk.

Reprezentálja a számláló i értéke $i = 0, 1, \ldots, 2^b - 1$ esetén az n_i összeget, ahol az n_i-k nemnegatív számoknak egy növekvő sorozatát alkotják. Feltesszük, hogy a számláló kezdeti értéke 0, amely az $n_0 = 0$ összeget reprezentálja. Az i értéket tartalmazó számlálón a NÖVEL művelet véletlen módon fog hatni. Ha $i = 2^b - 1$, akkor egy túlcsordulást jelző üzenetet kapunk. A többi esetben pedig a számlálót megnöveljük 1-gyel $1/(n_{i+1} - n_i)$ valószínűséggel, és változatlanul hagyjuk $1 - 1/(n_{i+1} - n_i)$ valószínűséggel.

Ha minden $i \geq 0$ esetén az $n_i = i$ választással élünk, akkor a számláló megegyezik a hagyományossal. Érdekesebb helyzet adódik, ha mondjuk azt választjuk, hogy minden $i > 0$ esetén $n_i = 2^{i-1}$ vagy $n_i = F_i$ (az i-edik Fibonacci-szám – lásd 3.2. alfejezet).

A feladatban feltételezzük, hogy n_{2^b-1} elég nagy ahhoz, hogy a túlcsordulási hiba valószínűsége elhanyagolható legyen.

a. Mutassuk meg, hogy n végrehajtott NÖVEL művelet után a számláló által reprezentált összeg várható értéke pontosan n.

b. A számláló által reprezentált mennyiség szórása az n_i sorozattól függ. Tekintsünk egy egyszerű esetet: legyen $n_i = 100i$ minden $i > 0$ esetén. Becsüljük meg a számláló által reprezentált összeg szórását n végrehajtott NÖVEL művelet után.

5-2. Keresés rendezetlen tömbben

Ebben a feladatban három olyan algoritmust vizsgálunk, amely egy n elemű rendezetlen A tömbben keres egy adott x elemet.

Tekintsük a következő véletlenített módszert. Válasszunk véletlenszerűen egy i indexet, amely az A tömb valamelyik elemére mutat. Ha $A[i] = x$, akkor készen vagyunk, egyébként válasszunk véletlenszerűen egy újabb tömbindexet. Folytassuk az indexek véletlen választását addig, amíg nem találunk egy olyan j-t, amelyre $A[j] = x$, vagy amíg az A tömb összes elemét meg nem vizsgáltuk. Minden újabb választásnál az összes lehetséges index közül választunk, vagyis előfordulhat, hogy a tömb egy elemét többször is megvizsgáljuk.

a. Adjunk pszeudokódot a fent leírt VÉLETLEN-KERESÉS eljárásra. Ügyeljünk arra, hogy az eljárás befejeződjön, ha az A tömb összes indexét már kiválasztottuk legalább egyszer.

b. Tegyük fel, hogy pontosan egy olyan i index van, amelyre $A[i] = x$. A VÉLETLEN-KERESÉS eljárás várhatóan hány tömbindexet fog végigpróbálni, amíg megtalálja az x elemet?

c. Általánosítsuk a (b) kérdést: tegyük fel, hogy pontosan $k \geq 1$ darab olyan index van, amelyre $A[i] = x$. A VÉLETLEN-KERESÉS eljárás várhatóan hány tömbindexet fog végigpróbálni, amíg megtalálja az egyik x elemet? A választ n és k függvényében adjuk meg.

d. Tegyük fel, hogy nincs olyan i index, amelyre $A[i] = x$ teljesülne. A VÉLETLEN-KERESÉS eljárás várhatóan hány tömbindexet fog végigpróbálni, amíg az összes elemet legalább egyszer megvizsgálja, és így megáll?

A következő módszer amit vizsgálunk, az a kézenfekvő determinisztikus lineáris keresés. A DETERMINISZTIKUS-KERESÉS eljárás úgy keresi meg az x elemet az A tömbben, hogy sorban

végignézi az $A[1], A[2], A[3], \ldots, A[n]$ elemeket, amíg vagy talál egy $A[i] = x$ tömbindexet, vagy eléri a tömb végét. Tegyük fel, hogy a bemeneti A tömb minden permutációja azonos valószínűségű.

e. Tegyük fel, hogy pontosan egy olyan i index létezik, amelyre $A[i] = x$. Mi a DETERMINISZTIKUS-KERESÉS eljárás várható futási ideje? Mi a DETERMINISZTIKUS-KERESÉS futási ideje a legrosszabb esetben?

f. Általánosítsuk az (e) kérdést: tegyük fel, hogy pontosan $k \geq 1$ darab olyan index van, amelyre $A[i] = x$. Mi a DETERMINISZTIKUS-KERESÉS eljárás várható futási ideje? Mi a DETERMINISZTIKUS-KERESÉS futási ideje a legrosszabb esetben? A választ n és k függvényében adjuk meg.

g. Tegyük fel, hogy nem létezik olyan i index, amelyre $A[i] = x$ teljesülne. Mi a DETERMINISZTIKUS-KERESÉS eljárás várható futási ideje? Mi a DETERMINISZTIKUS-KERESÉS futási ideje a legrosszabb esetben?

Végül tekintsük a KEVERŐ-KERESÉS eljárást, amely először véletlenszerűen permutálja a bemeneti tömb elemeit, majd a megkevert tömbben determinisztikus lineáris keresést használ a keresett elem megtalálására.

h. Tegyük fel, hogy pontosan k darab olyan index van, amelyre $A[i] = x$. Határozzuk meg a KEVERŐ-KERESÉS eljárás futási idejét átlagos, illetve legrosszabb esetben, ha $k = 0$ vagy $k = 1$. Általánosítsuk a megoldást a $k > 1$ esetre.

i. A három kereső algoritmus közül melyiket érdemes használni? Indokoljuk a választ.

Megjegyzések a fejezethez

Bollobás [44], Hofri [151] és Spencer [283] gazdag anyagot tartalmaz magasabb szintű valószínűségi módszerekről. Karp [174] és Rabin [253] részletesen bemutatja a véletlenített algoritmusok használatának előnyeit. Motwani és Raghavan könyve [228] mélyrehatóan tárgyalja a témát.

A munkatársfelvétel problémának számos változatát vizsgálták. Az ilyen típusú problémákat az angol nyelvű irodalom „secretary problems”, (titkár(nő) probléma) néven tárgyalja. Ajtai, Meggido és Waarts cikke [12] is ezzel a kérdéskörrel foglalkozik.

II. RENDEZÉSEK ÉS RENDEZETT MINTÁK

Bevezetés

Ez a rész rendezési módszereket mutat be. A ***rendezési feladat*** a következő.

Bemenet: egy n számból álló $\langle a_1, a_2, \ldots, a_n \rangle$ sorozat.

Kimenet: a bemenő sorozat elemeinek olyan $\langle a'_1, a'_2, \ldots, a'_n \rangle$ permutációja (átrendezése), amelyre $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

A bemenő sorozat rendszerint egy n elemű tömb, bár másképp is ábrázolható, például láncolt lista adatszerkezettel.

Az adatok szerkezete

A rendezendő adatok a legritkább esetben különálló számok, általában egy összetett adategyüttesnek, ***rekordnak*** a részei. Minden rekordban van egy ***kulcs,*** ez a rendezendő érték, a rekord többi része ***kísérő adat,*** amelyet többnyire a kulccsal együtt tárolnak. A gyakorlatban, ha a rendezés során két kulcsot fel kell cserélni, a hozzájuk tartozó kísérő adatokat is felcserélik, azonban ha ezek az adatok túlságosan terjedelmesek, csak a rekordokra mutató pointereket cserélik, hogy minimalizálják az adatmozgatást.

Valójában ezek már megvalósítási részletkérdések, amelyek megkülönböztetik az algoritmust a ténylegesen futó programtól. A rendezési *módszer* szempontjából lényegtelen, hogy hosszú rekordokat vagy egyes különálló számokat rendezünk. Így, a rendezési problémára összpontosítva, feltesszük, hogy a bemenő adatok egyszerűen számok. Bár egy számokat rendező algoritmusból rekordokat rendező programot készíteni eléggé kézenfekvő, mégis egy adott környezetben felléphetnek olyan technikai részletkérdések, melyek igazi kihívássá teszik ezt a programozási feladatot.

Miért fontos a rendezés?

Számos informatikai szakember az algoritmusokkal kapcsolatos tanulmányok legfontosabb problémájának tekinti a rendezési feladatot.

- Gyakori, hogy az információ rendezése elengedhetetlen egy alkalmazásban. Például az ügyfelek számlakivonatának elkészítéséhez a bankoknak a csekkeket csekkszám szerint rendezniük kell.
- Az algoritmusok sokszor kulcsfontosságú alprogramként használják a rendezést. Például annak a programnak, mely egymást fedő grafikai objektumokat jelenít meg, az

objektumokat rendeznie kell egy „föléhelyez” reláció szerint, hogy alulról felfelé egymást fedő rétegekbe megrajzolhassa őket. Számos algoritmussal fogunk találkozni a könyvben, melyek alprogramként rendezést használnak.

- A rendezési algoritmusok választéka igen széles, és nagyon gazdag a technikai eszközkészletük. Valójában, az évek során állandó fejlődésben lévő rendezési algoritmusok sok fontos, az algoritmustervezésben széles körben használatos technikát mutatnak be. Ezért a rendezési feladat történeti érdekességgel is rendelkezik.
- A rendezés olyan probléma, melyre egy nem triviális alsó korlátot adhatunk (ahogy ezt a 8. fejezetben meg is tesszük). A legjobb felső korlátjaink aszimptotikusan megegyeznek az alsó korláttal, így elmondhatjuk, hogy rendezési algoritmusaink aszimptotikusan optimálisak. Ezenkívül a rendezési feladat alsó korlátját felhasználhatjuk számos más probléma alsó korlátjának bizonyításában.
- Jó néhány mérnöki megfontolás kerül előtérbe a rendezési algoritmusok megvalósítása során. A leggyorsabb rendezési algoritmus egy adott helyzetben sok tényezőtől függhet, úgymint a kulcsokról és a kapcsolódó adatokról szerzett előzetes információktól, a gazda számítógép memória- (gyorsmemória és virtuális memória) szerkezetétől és a szoftverkörnyezettől. Ezek a fontos kérdések legjobban az algoritmus szintjén kezelhetők, és nem programozási „trükkökkel”.

Rendezési algoritmusok

A 2. fejezetben bemutattunk két rendezési algoritmust n valós szám rendezésére. Tudjuk, hogy a beszúró rendezés legrosszabb esetben $\Theta(n^2)$ futási idejű, de a belső ciklusai szorosak (a kódolásuk kevés lépéssel, hatékonyan megvalósítható), így kevés számú adat esetén ez egy gyors helyben rendező algoritmus. (Emlékeztetőül: egy algoritmusról azt mondjuk, ***helyben rendező,*** ha a bemenő adatok tömbjén kívül csak állandó számú változóra van szüksége a rendezéshez.) Az összefésülő rendezés aszimptotikus futási ideje jobb, $\Theta(n \lg n)$, de az ÖSSZEFÉSÜL eljárás, amit felhasznál, nem helyben működik.

Ebben a részben két másik algoritmust tárgyalunk, melyekkel tetszés szerinti valós számokat rendezhetünk. A 6. fejezetben bemutatott kupacrendezés helyben rendez n számot $O(n \lg n)$ idő alatt. Egy fontos – kupacnak nevezett – adatszerkezetet használ, mellyel az elsőbbségi sort is megvalósíthatjuk.

A 7. fejezetben ismertetett gyorsrendezés szintén helyben rendez, de ennek n elem esetén a legrosszabb futási ideje $\Theta(n^2)$. Átlagosan azonban $\Theta(n \lg n)$ futási idejű, és a gyakorlatban rendszerint felülmúlja a kupacrendezést. A beszúró rendezéshez hasonlóan szoros a kódja, így futási idejének állandó tényezője kicsi. Népszerű algoritmus nagy adathalmazok rendezéséhez.

A beszúró rendezés, az összefésülő rendezés, a kupacrendezés és a gyorsrendezés egyaránt összehasonlító rendezés: az elemek sorrendjét azok összehasonlításával állapítja meg. A 8. fejezet elején döntési fa segítségével elemezzük az összehasonlító rendezések hatékonyságának korlátait. Megmutatjuk, hogy bármely, összehasonlításokon alapuló rendezési módszer futási idejének alsó korlátja a legrosszabb esetet tekintve $\Omega(n \lg n)$, ezzel belátjuk, hogy a kupacrendezés és az összefésülő rendezés aszimptotikusan optimális összehasonlító rendezés.

Ezután a 8. fejezetben megmutatjuk, hogyan faraghatunk le ebből az $\Omega(n \lg n)$ alsó korlátból, ha nem összehasonlítunk, hanem más eszközökkel gyűjtünk információt az elemek rendezett sorozatbeli helyéről. A leszámláló rendezés például felteszi, hogy a bemenő számok az $\{1, 2, \ldots, k\}$ halmaz elemei. Az elemek relatív sorrendjének meghatározásához tömb indexelést használva n számot $\Theta(k+n)$ idő alatt rendez. Tehát amikor $k = O(n)$, a leszámláló rendezés futási ideje lineáris függvénye a bemenő adatok számának. Egy rokon algoritmus, a számjegyes rendezés, a leszámláló algoritmust terjeszti ki, tágítva annak alkalmazhatósági tartományát. Ha rendeznünk kell n darab d számjegyű egész számot, melyek számjegyei az $\{1, 2, \ldots, k\}$ halmaz elemei, a számjegyes rendezéssel $\Theta(d(n + k))$ idő alatt oldhatjuk meg a feladatot. Ez abban az esetben, ha k nagysága $O(n)$ és d konstans, lineáris futási időt jelent. Egy harmadik algoritmus, az edényrendezés a bemenő adatok valószínűségi eloszlásának ismeretét igényli. Ezzel átlagosan $O(n)$ idő alatt rendezhetünk n olyan valós számot, melyek eloszlása egyenletes a [0,1) félig nyílt intervallumon.

Rendezett minták

Egy n elemből álló rendezett minta i-edik eleme az i-edik legkisebb elemet jelenti. Ezt természetesen kiválaszthatjuk úgy, hogy rendezzük az adatokat, majd az eredmény i-edik elemét tekintjük. Ismeretlen eloszlású bemenő adatok esetén, ez $\Omega(n \lg n)$ idejű, a 8. fejezetben bizonyított alsó korlát szerint.

A 9. fejezetben megmutatjuk, hogy az i-edik legkisebb elem $O(n)$ idő alatt megtalálható, még akkor is, ha a bemenő adatok tetszőleges valós számok. Adunk egy egyszerűbb, rövid algoritmust, mely a legrosszabb esetben $\Theta(n^2)$, de átlagosan lineáris futási idejű, majd egy bonyolultabb algoritmust, mely viszont a legrosszabb esetben is $O(n)$ idejű.

Matematikai háttér

Bár a fejezet nagy része nem támaszkodik bonyolult matematikai alapokra, néhány alfejezethez szükség van matematikai ismeretekre. Nevezetesen a gyorsrendezés, az edényrendezés és a rendezett minta algoritmusok átlagos esetben való viselkedésének vizsgálata használja egyrészt a C függelékben áttekintett valószínűségszámítási alapismereteket, másrészt az 5. fejezet valószínűségi elemzéssel és a véletlenített algoritmusokkal foglalkozó anyagát. A legrosszabb esetek vizsgálatai közül a rendezett minta legrosszabb esetben is lineáris idejű algoritmusának elemzéséhez szükségesek mélyebb matematikai ismeretek.

6. Kupacrendezés

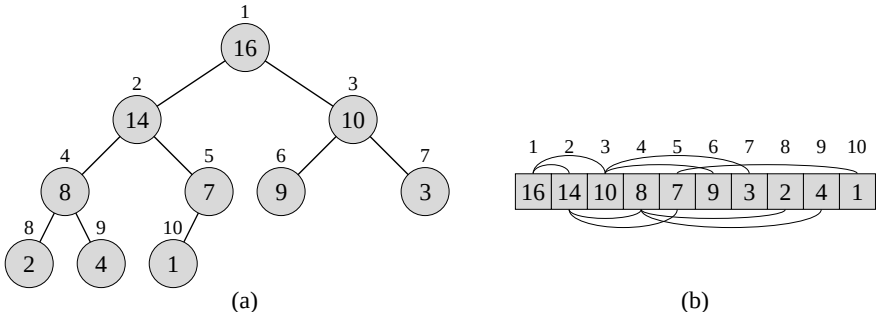
Ebben a fejezetben egy újabb rendezési algoritmust mutatunk be. Korábban két rendezési módszert is tárgyaltunk, a beszúró rendezést, melynek előnye, hogy helyben rendez, viszont a futási ideje nem túl kedvező, és az összefésülő rendezést, melynek futási ideje n elemre $O(n \lg n)$, de nem helyben rendez. A kupacrendezés egyesíti e két rendezési módszer jó tulajdonságait, ugyanis helyben rendez, a bemenő adatok tömbjén kívül csak néhány, állandó számú változóra lesz szükség, és futási ideje $O(n \lg n)$.

A kupacrendezés egy új algoritmustervezési technikát is bemutat, az algoritmus egy adatszerkezet felhasználásán alapul, jelen esetben ez a „kupac". A kupac adatszerkezet nemcsak a kupacrendezés miatt érdemel említést, hanem mint látni fogjuk, hatékonyan alkalmazható elsőbbségi sor kezeléséhez is. A későbbi fejezetek algoritmusaiban is találkozunk még a kupac adatszerkezettel.

Megemlítjük, hogy bár a „kupac" (heap) elnevezést elsőként a kupacrendezéssel kapcsolatban vezették be, de használják memóriafoglalással, a „hulladékgyűjtéses-memóriakezelés"-sel (garbage-collected storage) kapcsolatban is, például a Lisp és Java nyelvekben. Az általunk használt kupac adatszerkezet *nem* a programozási nyelvekben előforduló „heap", s valahányszor ebben a könyvben megemlítjük a kupac fogalmat, az itt definiált szerkezetre gondolunk.

6.1. Kupac

A (***bináris***) ***kupac*** adatszerkezet úgyis szemlélhető, mint egy majdnem teljes bináris fa (lásd B.5.3. pont) egy tömbben ábrázolva, ahogy a 6.1. ábra mutatja. A fa minden csúcsa megfelel a tömb egy elemének, mely a csúcs értékét tárolja. A fa minden szintjén teljesen kitöltött, kivéve a legalacsonyabb szintet, ahol balról jobbra haladva csak egy adott csúcsig vannak elemek. Az A tömb, mely egy kupacot alkot, két tulajdonsággal rendelkezik: *hossz*[A], mely a tömb elemeinek száma, és *kupac-méret*[A], mely az A tömbben tárolt kupac elemeinek száma. Így, jóllehet $A[1 \,..\, \textit{hossz}[A]]$ minden eleme tartalmazhat érvényes számot, $A[\textit{kupac-méret}[A]]$ utáni értékek, ahol $\textit{kupac-méret}[A] \leq \textit{hossz}[A]$, már nem tartoznak a kupachoz. A fa gyökere $A[1]$, és ha i a fa egy adott csúcsának tömbbeli indexe, akkor az ősének Szülő(i), bal oldali gyerekének Bal(i), és jobb oldali gyerekének Jobb(i) indexe egyszerűen kiszámítható:



6.1. ábra. A maximum-kupac mint bináris fa **(a)** és mint tömb **(b)**. A körök belsejébe írt szám a fa csúcsában tárolt érték, a kör fölé írt szám a csúcsnak megfelelő tömbelem indexe. A tömb alatti és feletti ívek mutatják a szülő-gyerek viszonyokat; a szülő mindig balra található a gyerekeihez képest. A fa magassága három; a 4 indexű csúcs (melynek 8 az értéke) egy magasságú.

SZÜLŐ(i)

1 **return** $\lfloor i/2 \rfloor$

BAL(i)

1 **return** $2i$

JOBB(i)

1 **return** $2i + 1$

A legtöbb számítógépen a BAL eljárás megvalósításakor a $2i$ értéket egy művelettel kiszámíthatjuk, az i binárisan ábrázolt értékét egyszerűen balra léptetve eggyel. Hasonlóan a JOBB eljárás $2i + 1$ értéke könnyen kiszámítható a binárisan ábrázolt i eggyel balra léptetésével, úgy hogy a belépő legalacsonyabb helyi értékű bit 1 legyen. A SZÜLŐ eljárásban az $\lfloor i/2 \rfloor$-t i eggyel jobbra léptetésével számíthatjuk ki. A kupacrendezés hatékony megvalósításában ezt a három eljárást gyakran „makróként” vagy „in-line” eljárásként készítik el.

A bináris kupacnak két fajtája van: a maximum-kupac és a minimum-kupac. Mindkét fajta kupacban a csúcsok értékei kielégítik a ***kupactulajdonságot,*** melynek jellemző vonása függ a kupac fajtájától. ***Maximum-kupac*** esetén a kupac minden gyökértől különböző i eleme ***maximum-kupactulajdonságú:***

$$A[\text{SZÜLŐ}(i)] \geq A[i],$$

azaz az elem értéke legfeljebb akkora, mint a szülőjének értéke. Így a maximum-kupac legnagyobb eleme a gyökér, és egy adott csúcs alatti részfa minden elemének értéke nem nagyobb, mint az adott csúcsban lévő elem értéke. A ***minimum-kupac*** épp ellentétes szerkezetű; a kupac minden gyökértől különböző i eleme ***minimum-kupactulajdonságú:***

$$A[\text{SZÜLŐ}(i)] \leq A[i].$$

A minimum-kupacban a legkisebb elem van a gyökérben.

A kupacrendezés algoritmusban maximum-kupacot használunk. A minimum-kupacot általában az elsőbbségi soroknál alkalmazzák, amit a 6.5. alfejezetben fogunk tárgyalni. Mindig pontosan megadjuk, hogy egy bizonyos alkalmazásban maximum- vagy minimum-kupacra van szükségünk, ha az adott feladathoz mindkettő megfelelő, egyszerűen csak a „kupac” megnevezést használjuk.

A kupacot egy faként vizsgálva, a kupac egy adott elemének ***magasságán*** azon leghosszabb egyszerű út éleinek a számát értjük, mely az adott csúcsból egy levélhez vezet. A kupac magasságának a gyökérelem magasságát tekintjük. Minthogy egy n elemű kupac egy teljes bináris fán alapszik, a magassága $\Theta(\lg n)$ (lásd a 6.1-2. gyakorlatot). Látni fogjuk, hogy a kupacon végzett alapműveletek időköltsége a fa magasságával arányos, azaz $O(\lg n)$ idejű. A továbbiakban bemutatunk néhány alapvető eljárást, és azok alkalmazását a rendezési algoritmusban és az elsőbbségi sor adatszerkezetben.

- A MAXIMUM-KUPACOL eljárás $O(\lg n)$ futási idejű, kulcsfontosságú a maximum-kupactulajdonság fenntartásához.
- A MAXIMUM-KUPACOT-ÉPÍT eljárás lineáris futási idejű, tetszőleges adatokat tartalmazó tömböt maximum-kupaccá alakít.
- A KUPACRENDEZÉS eljárás $O(n \lg n)$ futási idejű, helyben rendez egy tömböt.
- A MAXIMUM-KUPACBA-BESZÚR, KUPACBÓL-KIVESZ-MAXIMUM, KUPACBAN-KULCSOT-NÖVEL és KUPAC-MAXIMUMA eljárások $O(\lg n)$ futási idejűek. Lehetővé teszik a kupac adatszerkezet elsőbbségi sorként való felhasználását.

Gyakorlatok

6.1-1. Egy h magasságú kupacnak legalább (ill. legfeljebb) hány eleme van?

6.1-2. Mutassuk meg, hogy egy n elemű kupac magassága $\lfloor \lg n \rfloor$.

6.1-3. Mutassuk meg, hogy egy maximum-kupac bármely részfájának legnagyobb eleme a részfa gyökerében van.

6.1-4. Hol helyezkedhet el a legkisebb elem a maximum-kupacban, ha minden elem különböző?

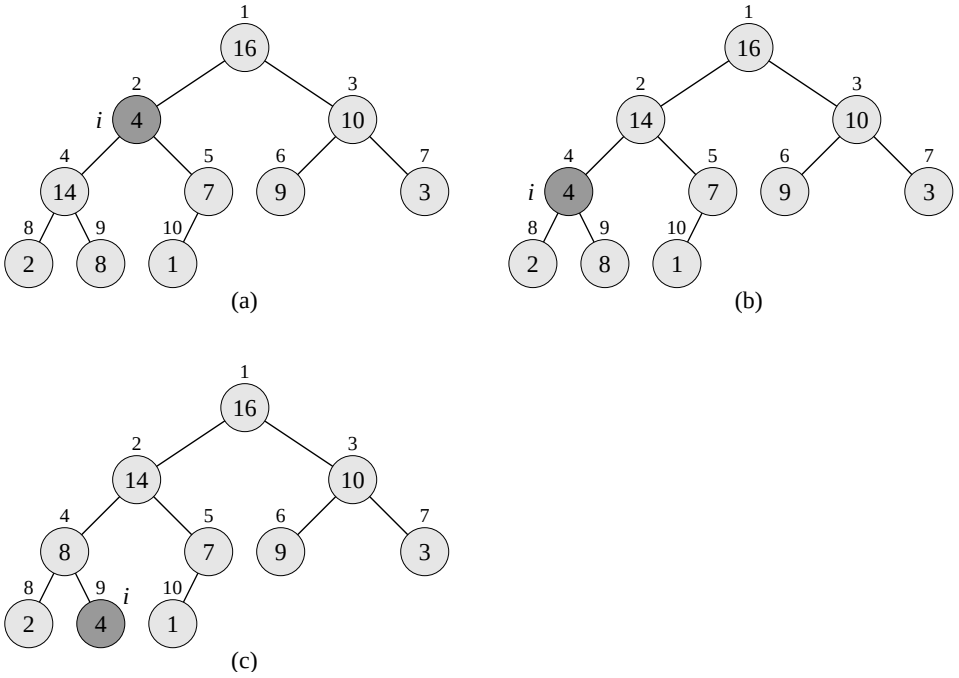
6.1-5. Minimum-kupac-e a nagyság szerint rendezett értékeket tartalmazó tömb?

6.1-6. Maximum-kupac-e a $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ sorozat?

6.1-7. Mutassuk meg, hogy egy n elemű kupacban tömbös ábrázolás esetén a levelek indexei rendre $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2. A kupactulajdonság fenntartása

A MAXIMUM-KUPACOL eljárás fontos a maximum-kupacok kezelésénél. Bemenő adatai az A tömb, és annak egy i indexe. MAXIMUM-KUPACOL meghívásakor feltesszük, hogy a BAL(i) és JOBB(i) gyökerű részfák maximum-kupac szerkezetűek, de $A[i]$ kisebb lehet a gyerekeinél, így megsértheti a maximum-kupactulajdonságot. MAXIMUM-KUPACOL feladata, hogy $A[i]$ értéket „lefelé görgesse” a maximum kupacban úgy, hogy az i gyökerű részfa maximum-kupaccá alakuljon.



6.2. ábra. A MAXIMUM-KUPACOL($A, 2$) működése, ahol *kupac-méret*[A] = 10. **(a)** Kiinduláskor $A[2]$ az i = 2-es csúcsnál sérti a maximum-kupactulajdonságot, mivel nem nagyobb a gyerekeinél. A maximum-kupactulajdonságot helyreállítjuk a 2-es csúcsra nézve azzal, hogy $A[2]$ és $A[4]$ elemeket felcseréljük, de elrontjuk a 4-es csúcsra nézve **(b)**. A MAXIMUM-KUPACOL($A, 4$) rekurzív hívása i-t 4-re állítja. $A[4]$ és $A[9]$ felcserélése után, ahogy azt **(c)** mutatja, a 4 értékű elem helyére kerül, és a MAXIMUM-KUPACOL($A, 9$) rekurzív hívás már nem talál változtatni valót az adatszerkezeten.

MAXIMUM-KUPACOL(A, i)

```
1  l ← BAL(i)
2  r ← JOBB(i)
3  if l ≤ kupac-méret[A] és A[l] > A[i]
4     then legnagyobb ← l
5     else legnagyobb ← i
6  if r ≤ kupac-méret[A] és A[r] > A[legnagyobb]
7     then legnagyobb ← r
8  if legnagyobb ≠ i
9     then A[i] ↔ A[legnagyobb] csere
10          MAXIMUM-KUPACOL(A, legnagyobb)
```

A 6.2. ábra a MAXIMUM-KUPACOL eljárás működését mutatja be. Minden lépésnél meghatározzuk $A[i]$, $A[\text{BAL}(i)]$ és $A[\text{JOBB}(i)]$ közül a legnagyobbat, és indexét a *legnagyobb* nevű változóba tesszük. Ha $A[i]$ a legnagyobb, akkor az i gyökerű részfa maximum-kupac és az eljárásnak vége. Egyébként a két gyerek valamelyike a legnagyobb elem, ezért $A[i]$-t felcseréljük $A[$*legnagyobb*$]$-bal, így az i csúcs és gyerekei kielégítik a maximum-kupactulajdonságot. A *legnagyobb* indexű csúcs felveszi az eredeti $A[i]$ értéket, ezért most

a *legnagyobb* gyökerű részfa sértheti a maximum-kupactulajdonságot. Következésképpen a MAXIMUM-KUPACOL eljárást rekurzívan meg kell hívni erre a részfára.

A MAXIMUM-KUPACOL eljárás futási ideje egy n méretű i gyökerű részfát tekintve $\Theta(1)$ – mely alatt az $A[i]$, $A[\text{BAL}(i)]$ és $A[\text{JOBB}(i)]$ közötti viszonyok helyreállíthatók –, plusz az az idő, mely alatt a MAXIMUM-KUPACOL végrehajtódik az i csúcs egyik gyerekéből származó részfára. Ezen részfák mindegyike legfeljebb $2n/3$ nagyságú – a legkedvezőtlenebb az az eset, amikor a fa legalsó szintje pontosan félig van kitöltve –, így a MAXIMUM-KUPACOL eljárás futási ideje a következő rekurzív egyenlőtlenséggel adható meg:

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1).$$

A rekurzió megoldása a mester tétel (4.1. tétel) második esete alapján $T(n) = O(\lg n)$. Másképpen felírva a MAXIMUM-KUPACOL futási ideje h magasságú csúcsra $O(h)$-val jellemezhető.

Gyakorlatok

6.2-1. A 6.2. ábrát mintául véve mutassuk meg, hogyan működik a MAXIMUM-KUPACOL$(A, 3)$ az $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$ tömbön.

6.2-2. Készítsük el MINIMUM-KUPACOL(A, i) absztrakt kódját, mely MAXIMUM-KUPACOL-hoz hasonló műveletet végez egy minimum-kupacon. Hasonlítsuk össze MINIMUM-KUPACOL és MAXIMUM-KUPACOL futási idejét.

6.2-3. Mi történik MAXIMUM-KUPACOL(A, i) hívásakor abban az esetben, ha $A[i]$ nagyobb a gyerekeinél?

6.2-4. Mi történik MAXIMUM-KUPACOL(A, i) hívásakor, ha $i >$ *kupac-méret*$[A]/2$?

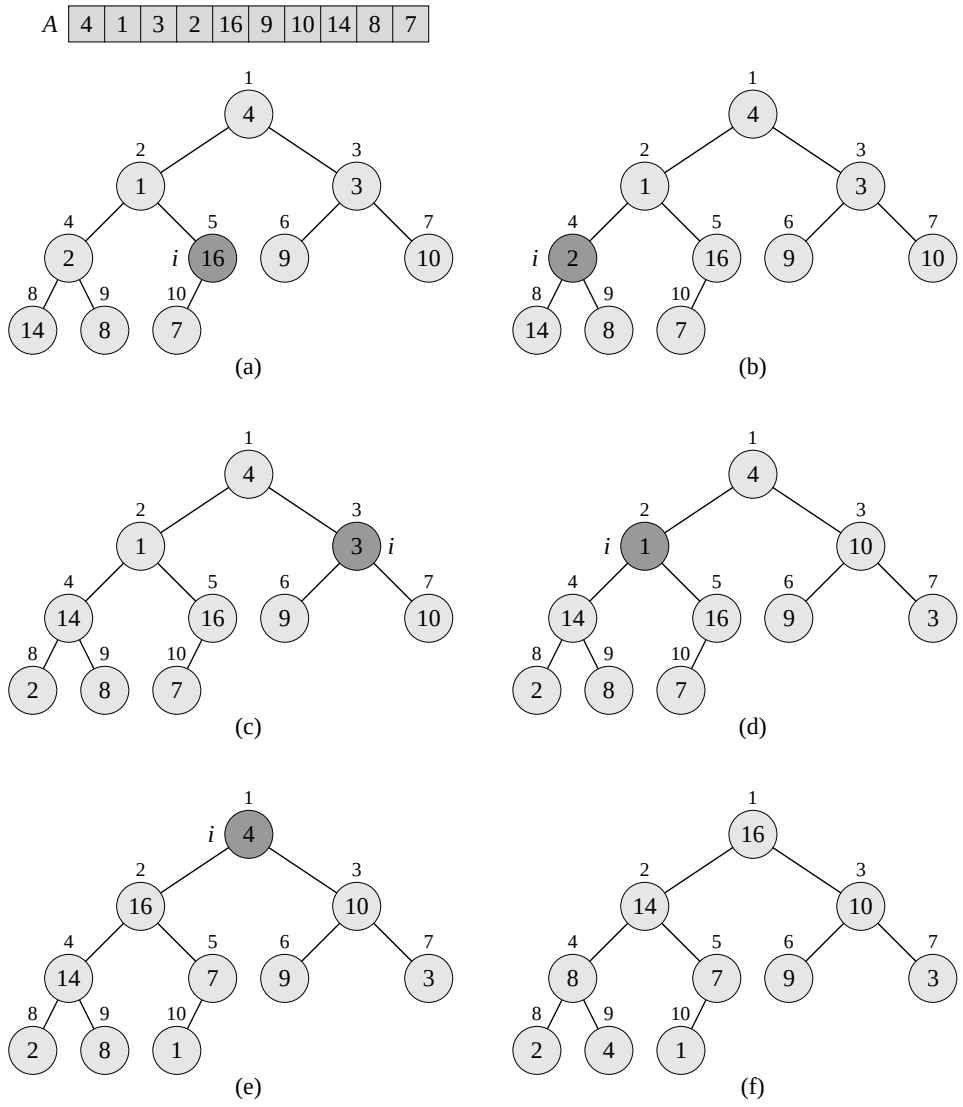
6.2-5. A MAXIMUM-KUPACOL eljárás az állandó végrehajtási idejű lépéseket tekintve nagyon hatékonyan kódolható, míg a 10. sorban lévő rekurzív hívásnál előfordulhat, hogy a fordító által előállított kód nem elég hatékony. A rekurzió helyett ciklust használva írjunk hatékony kódot.

6.2-6. Mutassuk meg, hogy a MAXIMUM-KUPACOL futási ideje n méretű kupac esetén legrosszabb esetben $\Omega(\lg n)$. (*Útmutatás.* Egy n elemből álló kupac esetén adjuk meg a csúcspontok értékét úgy, hogy a MAXIMUM-KUPACOL eljárás a gyökértől a levélig vezető út minden elemére rekurzívan meghívódjon.)

6.3. A kupac építése

A MAXIMUM-KUPACOL eljárást alulról felfelé haladva felhasználhatjuk, hogy az $A[1\,..\,n]$ tömböt, ahol $n =$ *hossz*$[A]$, maximum-kupaccá alakítsuk. A 6.1-7. gyakorlat szerint az A tömb $A[(\lfloor n/2\rfloor + 1)\,..\,n]$ elemei levelek, melyek egyelemű kupacnak tekinthetők. A MAXIMUM-KUPACOT-ÉPÍT eljárásnak tehát csak a többi csúcson kell végighaladnia, és minden egyes csúcsra lefuttatni a MAXIMUM-KUPACOL eljárást.

```
MAXIMUM-KUPACOT-ÉPÍT(A)
1  kupac-méret[A] ← hossz[A]
2  for i ← ⌊hossz[A]/2⌋ downto 1
3      do MAXIMUM-KUPACOL(A, i)
```



6.3. ábra. A MAXIMUM-KUPACOT-ÉPÍT működése. Az ábrákon rendre a MAXIMUM-KUPACOL meghívása előtt (MAXIMUM-KUPACOT-ÉPÍT 3. sora) láthatjuk az adatszerkezetet. **(a)** A bemenő 10 elemű A tömb, és a bináris fa, melyet a tömb ábrázol. Ahogy az ábra mutatja, az i ciklusváltozó az 5. csúcsra mutat a MAXIMUM-KUPACOL(A, i) hívásakor. **(b)** Az eredményt ábrázolja, ekkor az i ciklusváltozó a következő iterációnak megfelelően már a 4. csúcsra mutat. **(c)–(e)** A MAXIMUM-KUPACOT-ÉPÍT eljárás **for** ciklusának egymást követő iterációit szemlélteti. Figyeljük meg, hogy a MAXIMUM-KUPACOL adott csúcsra való meghívásakor a csúcs alatti két részfa már kupac. **(f)** A MAXIMUM-KUPACOT-ÉPÍT tevékenységének végeredményét mutatja.

A 6.3. ábra a MAXIMUM-KUPACOT-ÉPÍT eljárás működését mutatja be.

MAXIMUM-KUPACOT-ÉPÍT eljárás helyességének bizonyításához az alábbi ciklusinvariáns feltételt használjuk:

A 2–3. sorokban levő **for** ciklus minden iterációjának kezdetén fennáll, hogy az $i + 1, i + 2, \ldots, n$ csúcsok egy-egy maximum-kupac gyökerei.

Először ellenőriznünk kell, hogy az invariáns feltétel teljesül a ciklus első iterációja előtt, majd igazolnunk kell, hogy a ciklus minden iterációja után továbbra is érvényes marad. Végül meg kell mutatnunk, hogy a ciklus befejeződésekor az invariáns feltétel segítségével igazolhatjuk az algoritmus helyességét.

Teljesül: Tekintsük az első iteráció előtti helyzetet, ekkor $i = \lfloor n/2 \rfloor$. Az $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ csúcsok mindegyike levél, azaz egy-egy triviális maximum-kupac gyökere.

Megmarad: Belátjuk, hogy a ciklusinvariáns minden iteráció után érvényben marad, ehhez vegyük észre, hogy az i csúcs gyerekeinek indexe nagyobb, mint i, így a ciklusinvariáns szerint ezek egy-egy maximumkupacnak a gyökerei. Pontosan ezeknek a feltételeknek kell teljesülniük ahhoz, hogy a MAXIMUM-KUPACOL(A, i) hívás az i csúcsot egy maximum-kupac gyökerévé alakítsa. Továbbá tudjuk, hogy a MAXIMUM-KUPACOL hívás megtartja az $i + 1, i + 2, \ldots, n$ csúcsoknak azt a tulajdonságát, hogy maximum-kupac gyökerek. A **for** ciklus végén i értéke eggyel csökken, ezzel helyreáll a ciklusinvariáns feltétel a következő iterációhoz.

Befejeződik: Az eljárás befejeződésekor $i = 0$. A ciklusinvariáns szerint az $1, 2, \ldots, n$ csúcsok mindegyikére igaz, hogy maximum-kupac gyökere, tehát az 1 indexű csúcs is az.

A MAXIMUM-KUPACOT-ÉPÍT eljárás futási idejének egy egyszerű felső korlátját a következőképpen kaphatjuk meg: MAXIMUM-KUPACOL minden egyes meghívása $O(\lg n)$ idejű, és $O(n)$ ilyen hívás adódik, így a futási idő legfeljebb $O(n \lg n)$. Ez a felső becslés bár korrekt, de aszimptotikusan nem szoros.

Élesebb korlátot kapunk, ha megfigyeljük, hogy a MAXIMUM-KUPACOL futási ideje függ a csúcs magasságától a fában, ami a csúcsok többségére kicsi. Erősebb becslésünk azon a tulajdonságon alapszik, hogy egy n elemű kupac magassága $\lfloor \lg n \rfloor$ (lásd a 6.1-2. gyakorlatot), és bármely h magasság esetén, a h magasságú csúcsok száma legfeljebb $\lceil n/2^{h+1} \rceil$ (lásd a 6.3-3. gyakorlatot).

MAXIMUM-KUPACOL eljárás futási ideje egy h magasságú csúcsra $O(h)$, így a fenti korlátokat figyelembe véve a MAXIMUM-KUPACOT-ÉPÍT eljárás teljes futási ideje

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

A jobb oldali összeget úgy számíthatjuk ki, hogy az (A.8) képletbe $x = 1/2$-et helyettesítünk, azaz

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1 - 1/2)^2} \\ &= 2. \end{aligned}$$

Innen a MAXIMUM-KUPACOT-ÉPÍT futási idejére a következő felső korlát adódik:

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n). \end{aligned}$$

Tehát egy rendezetlen tömb lineáris idő alatt kupaccá alakítható.

A MAXIMUM-KUPACOT-ÉPÍT eljáráshoz hasonló MINIMUM-KUPACOT-ÉPÍT eljárással minimum-kupacot építhetünk. A 3. sorban a MAXIMUM-KUPACOL hívást helyettesítjük egy MINIMUM-KUPACOL hívásra (lásd a 6.2-2. gyakorlatot). MINIMUM-KUPACOT-ÉPÍT egy rendezetlen tömböt lineáris időben minimum-kupaccá alakít.

Gyakorlatok

6.3-1. A 6.3. ábrát mintául véve mutassuk meg, hogyan működik a MAXIMUM-KUPACOT- ÉPÍT az $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$ tömbön.

6.3-2. Miért csökkentjük a ciklusváltozót $\lfloor hossz[A]/2\rfloor$-től 1-ig a MAXIMUM-KUPACOT-ÉPÍT 2. sorában, ahelyett hogy növelnénk 1-től $\lfloor hossz[A]/2\rfloor$-ig?

6.3-3. Mutassuk meg, hogy egy n elemű kupacban a h magasságú csúcsok száma legfeljebb $\lceil n/2^{h+1}\rceil$.

6.4. A kupacrendezés algoritmus

Az algoritmus a MAXIMUM-KUPACOT-ÉPÍT meghívásával kezdődik, mely maximum-kupaccá alakítja az $A[1\,..\,n]$ bemenő tömböt, ahol $n = hossz[A]$. Mivel a maximum-kupacban a legnagyobb elem a gyökérben, azaz az $A[1]$ elemben található, tehát ha felcseréljük az $A[1]$ és az $A[n]$ elemeket, a legnagyobb elem a rendezés szerinti helyére kerül. Majd az n-edik elemet „kizárva” a kupacból (eggyel csökkentve *kupac-méret*$[A]$-t), a maradék $A[1\,..\,(n-1)]$ elemet ismét könnyen maximum-kupaccá alakíthatjuk, ugyanis a gyökér gyerekei maximum-kupacok maradtak, csak az új gyökérelem sértheti a maximum-kupactulajdonságot. Ezért a MAXIMUM-KUPACOL$(A, 1)$ hívásával helyreállíthatjuk a kupactulajdonságot az $A[1\,..\,(n-1)]$ tömbön. Ezt ismételjük a kupac méretét csökkentve $(n-1)$-től 2-ig. (A ciklusinvariáns pontos megadását lásd a 6.4-2. gyakorlatnál.)

KUPACRENDEZÉS(A)

```
1  MAXIMUM-KUPACOT-ÉPÍT(A)
2  for i ← hossz[A] downto 2
3      do A[1] ↔ A[i] csere
4         kupac-méret[A] ← kupac-méret[A] – 1
5         MAXIMUM-KUPACOL(A, 1)
```

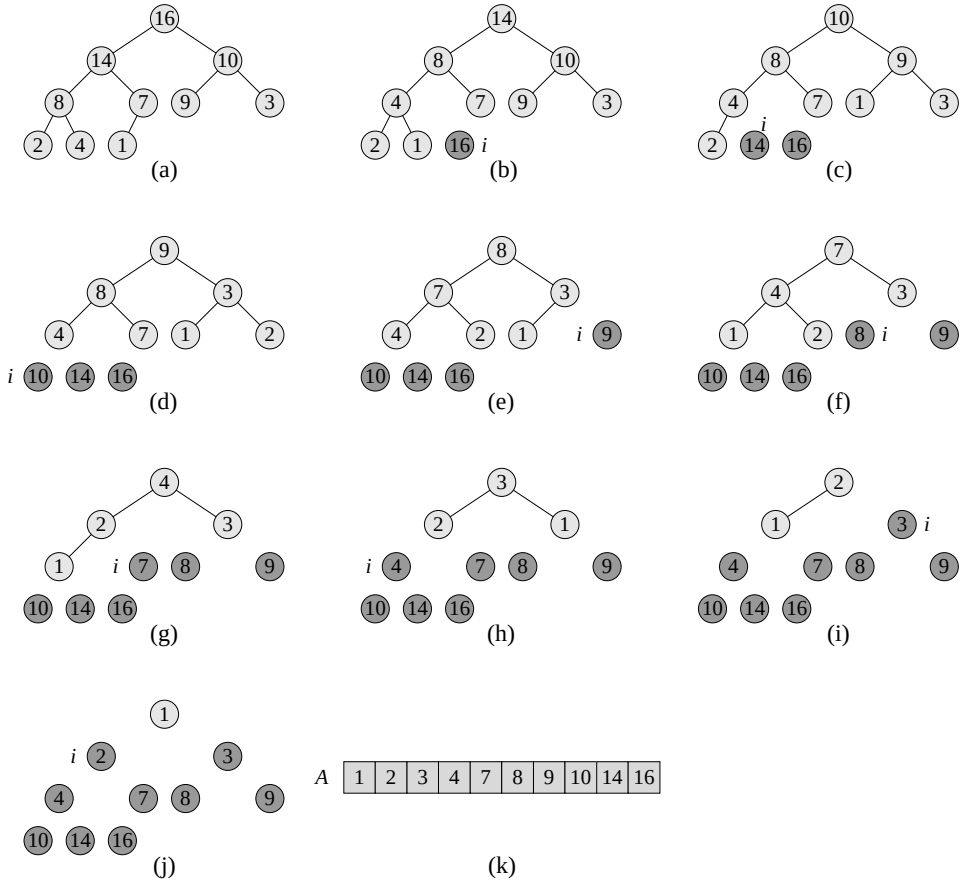
A 6.4. ábra egy példán bemutatja KUPACRENDEZÉS működését, onnan indulva, hogy az adatokból a kezdeti maximum-kupac elkészült. Az egyes maximum-kupacok a 2–5. sorbeli **for** ciklus adott iterációs lépésének kezdetekor fennálló állapotot mutatják.

KUPACRENDEZÉS futási ideje $O(n \lg n)$, minthogy MAXIMUM-KUPACOT-ÉPÍT $O(n)$ idő alatt fut, és MAXIMUM-KUPACOL minden egyes $(n-1)$-szer történő lefutása $O(\lg n)$ idejű.

Gyakorlatok

6.4-1. A 6.4. ábrát mintául véve mutassuk meg, hogyan működik a KUPACRENDEZ ÉS eljárás az $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$ tömbön.

6.4-2. Bizonyítsuk be a KUPACRENDEZÉS algoritmus helyességét a következő ciklusinvariánst használva:



6.4. ábra. A KUPACRENDEZÉS működése. **(a)** A MAXIMUM-KUPACOT-ÉPÍT eljárás által felépített maximum-kupac adatszerkezet. **(b)–(j)** Az adatszerkezet a MAXIMUM-KUPACOL egy-egy hívása után (algoritmus 5. sora). Az ábra mutatja az i változó pillanatnyi értékét. A kupacban csak a világosabb színnel jelzett csúcsok maradtak. **(k)** Az eredményül kapott rendezett A tömb.

A 2–5. sorokban lévő **for** ciklus minden iterációjának kezdetén fennáll, hogy az $A[1\,..\,i]$ résztömb egy maximum-kupac, amely az $A[1\,..\,n]$ tömb i darab legkisebb elemét tartalmazza, továbbá az $A[i+1\,..\,n]$ résztömb az $A[1\,..\,n]$ tömb $n-i$ darab legnagyobb elemét tartalmazza növekvő leg rendezve.

6.4-3. Mennyi a futási ideje a kupacrendezésnek n hosszúságú növekvően rendezett tömbön? Mit mondhatunk csökkenő sorrend esetén?

6.4-4. Mutassuk meg, hogy a kupacrendezés legrosszabb futási ideje $\Omega(n \lg n)$.

6.4-5. ⋆ Mutassuk meg, hogy abban az esetben, amikor minden elem különböző, a kupacrendezés legjobb futási ideje $\Omega(n \lg n)$.

6.5. Elsőbbségi sorok

A kupacrendezés kiváló algoritmus, de egy jól megvalósított gyorsrendezés (a 7. fejezetben ismertetjük) a gyakorlatban általában hatékonyabb. Azonban maga a kupac adatszerkezet igen sokoldalúan használható. Ebben az alfejezetben a kupac egyik leggyakoribb alkalmazási területét mutatjuk be: hogyan használhatjuk elsőbbségi sorok hatékony kezelésére. A kupachoz hasonlóan kétféle elsőbbségi sorról beszélhetünk: maximum- és minimum-elsőbbségi sor. Vizsgálatunk tárgya a maximum-elsőbbségi sor maximum-kupaccal történő megvalósítása lesz; ez alapján elkészíthetők a minimum-elsőbbségi sor algoritmusai, lásd a 6.5-3. gyakorlatot.

Elsőbbségi soron egy S halmazt értünk, melynek minden eleméhez egy ***kulcs*** értéket rendelünk. A ***maximum-elsőbbségi sort*** kezelő műveletek a következők:

BESZÚR(S, x) egy x elemet hozzáad az S halmazhoz. Ezt az $S \leftarrow S \cup \{x\}$ módon írhatjuk fel.

MAXIMUM(S) megadja S legnagyobb kulcsú elemét.

KIVESZ-MAXIMUM(S) megadja és törli S legnagyobb kulcsú elemét.

KULCSOT-NÖVEL(S, x, k) megnöveli az x elem kulcsát, az új értéke k lesz, amiről feltesszük, hogy legalább akkora, mint az x elem kulcsának pillanatnyi értéke.

Maximum-elsőbbségi sort használnak például az osztott működésű számítógépeken a munkák ütemezéséhez. Az elvégzendő feladatokat és relatív prioritásukat egy maximum-elsőbbségi sorban tárolják. Ha egy feladat elkészült, vagy megszakítás következett be, a várakozók közül a legnagyobb prioritású munkát a KIVESZ-MAXIMUM eljárással választhatjuk ki. A BESZÚR segítségével pedig új munkákat vehetünk fel a sorba.

Ennek mintájára a ***minimum-elsőbbségi sort*** a BESZÚR, a MINIMUM, a KIVESZ-MINIMUM és a KULCSOT-CSÖKKENT műveletekkel kezelhetjük. A minimum-elsőbbségi sort használhatjuk például az esemény-vezérelt szimulációhoz. A sorban lévő elemek a szimulálandó események, amelyekhez hozzárendeljük az esemény bekövetkezésének időpontját mint kulcsot. Az eseményeket a bekövetkezési idejüknek megfelelő sorrendben kell szimulálni, mivel egy esemény bekövetkezése egy másik esemény jövőbeli bekövetkezését okozhatja. A szimulációs program KIVESZ-MINIMUM segítségével határozza meg a következő szimulálandó eseményt. Ha új esemény jelentkezik, annak felvételét a mimimum-elsőbbségi sorba a BESZÚR művelet végzi el. A 23. és 24. fejezetekben fogunk még példákat látni a mimimum-elsőbbségi sor alkalmazására, ahol fontos szerepet kap majd a KULCSOT-CSÖKKENT művelet.

Természetesen adódik, hogy a kupac adatszerkezet segítségével megvalósítható az elsőbbségi sor. Egy adott alkalmazásban, mint például a feladatütemezés vagy az esemény-vezérelt szimuláció, az elsőbbségi sor elemei az alkalmazás objektumainak felelnek meg. Valami módon egyértelművé kell tennünk, hogy egy adott objektum az elsőbbségi sor melyik eleméhez tartozik, és fordítva. Kupac használata esetén ez azt jelenti, hogy gyakran a kupac elemeiben egy ***nyél*** tárolására is szükség van, mely azonosítja az elemhez tartozó objektumot. Ennek megvalósítása az alkalmazástól függ, lehet egy mutató vagy egy egész szám. Hasonlóan az objektumokban is tárolni kell egy nyelet, mellyel azok összekapcsolhatók a hozzájuk tartozó kupacbeli elemmel. Itt például nyélként egy tömbindexet használhatunk. Mivel a kupacban az elemek a műveletek során megváltoztathatják tömbbeli elhelyezkedésüket, a megvalósításban szükség van az objektumokban nyélként tárolt tömbindexek megfelelő módosítására is. Az objektumok elérésével kapcsolatos részletek

erősen függnek az alkalmazástól, és annak tényleges megvalósításától, ezért itt most ezzel nem foglalkozunk, de megjegyezzük, hogy szükség van az azonosítást biztosító nyelek helyes karbantartására.

Nézzük a maximum-elsőbbségi sor műveleteinek megvalósítását. A KUPAC-MAXIMUMA eljárás $\Theta(1)$ idő alatt megvalósítja a MAXIMUM műveletet.

```
KUPAC-MAXIMUMA(A)
1  return A[1]
```

A KUPACBÓL-KIVESZ-MAXIMUM eljárás valósítja meg a KIVESZ-MAXIMUM műveletet. Ez a KUPACRENDEZÉS **for** ciklusának magjához (3–5. sorok) hasonló.

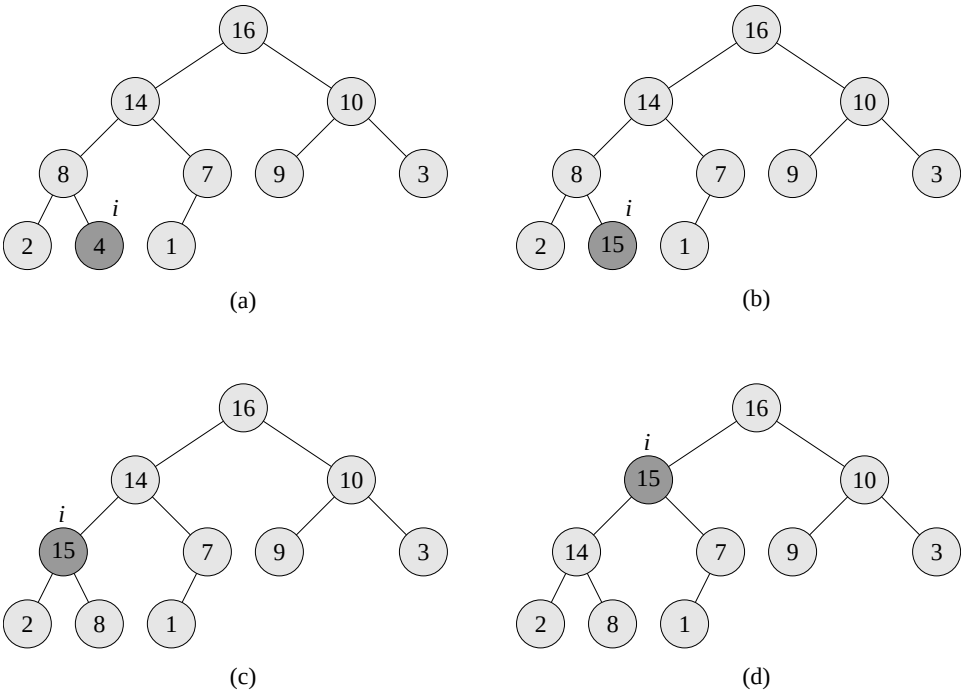
```
KUPACBÓL-KIVESZ-MAXIMUM(A)
1  if kupac-méret[A] < 1
2     then error „kupacméret alulcsordulás”
3  max ← A[1]
4  A[1] ← A[kupac-méret[A]]
5  kupac-méret[A] ← kupac-méret[A] – 1
6  MAXIMUM-KUPACOL(A, 1)
7  return max
```

A KUPACBÓL-KIVESZ-MAXIMUM futási ideje $O(\lg n)$, mivel csak néhány konstans végrehajtási idejű lépést tartalmaz a $O(\lg n)$ futási idejű MAXIMUM-KUPACOL eljárás meghívása előtt.

A KUPACBAN-KULCSOT-NÖVEL eljárás valósítja meg a KULCSOT-NÖVEL műveletet. Az i index azonosítja a tömbbeli helyét az elsőbbségi sor azon elemének, amelynek kulcsát megváltoztatjuk. Elsőként az $A[i]$ kulcsát frissíti az új értékre. Az $A[i]$ kulcsának megnövelése elronthatja a maximum-kupactulajdonságot, ezért az eljárás ezek után a 2.1. alfejezetben található BESZÚRÓ-RENDEZÉS algoritmus beszúró ciklusára (5–7. sorok) emlékeztető módon végigszalad az adott elemtől a gyökérhez vezető úton, hogy megkeresse az újonnan módosított kulcs megfelelő helyét. A művelet során újra és újra összehasonlítja az elemet az ősével, majd ha szükséges – azaz a módosított kulcs nagyobb –, felcseréli őket és tovább folytatja mindaddig, amíg a módosított elem kulcsa kisebb nem lesz az ősénél, vagy felértünk a kupac tetejére, ami azt jelenti, hogy a maximum-kupactulajdonság ismét fennáll. (A ciklusinvariáns pontos megadását a 6.5-5. gyakorlatnál találjuk.)

```
KUPACBAN-KULCSOT-NÖVEL(A, i, kulcs)
1  if kulcs < A[i]
2     then error „az új kulcs kisebb, mint az eredeti”
3  A[i] ← kulcs
4  while i > 1 és A[SZÜLŐ(i)] < A[i]
5        do A[i] ↔ A[SZÜLŐ(i)] csere
6           i ← SZÜLŐ(i)
```

A 6.5. ábrán egy példát látunk a KUPACBAN-KULCSOT-NÖVEL eljárás működésére. A KUPACBAN-KULCSOT-NÖVEL futási ideje egy n elemű kupacra $O(\lg n)$, mivel az eljárás 3. sorában módosított csúcstól a gyökérig vezető út hossza $O(\lg n)$.



6.5. ábra. A KUPACBAN-KULCSOT-NÖVEL működése. **(a)** A 6.4(a) ábrán látható maximum-kupac, sötét szürke szín jelzi az *i* indexű csúcsot. **(b)** Az adott csúcs kulcsát 15-re növeljük. **(c)** A 4–6. sorok iterációjának egy menete után a vizsgált csúcs és az ősének kulcsa felcserélődött, az *i* index feljebb lépett, az ősre mutat. **(d)** A maximum kupac a **while** ciklus egy újabb menete után. Ekkor $A[\text{SZÜLŐ}(i)] \geq A[i]$. A maximum-kupac tulajdonság helyreállt, az eljárás befejeződik.

A MAXIMUM-KUPACBA-BESZÚR eljárás valósítja meg a BESZÚR műveletet. A maximum-kupacba beszúrandó új elem kulcsát bemenetként kapja meg az algoritmus. El őször kibővíti a maximum-kupacot, hozzáadva egy olyan új levelet a fához, amelynek kulcsa $-\infty$. Majd meghívja a KUPACBAN-KULCSOT-NÖVEL eljárást, mely beállítja az új elem kulcsát, és helyreállítja a maximum-kupactulajdonságot.

MAXIMUM-KUPACBA-BESZÚR(A, *kulcs*)

1 *kupac-méret*$[A] \leftarrow$ *kupac-méret*$[A] + 1$
2 $A[$*kupac-méret*$[A]] \leftarrow -\infty$
3 KUPACBAN-KULCSOT-NÖVEL(A,*kupac-méret*$[A]$, *kulcs*)

A MAXIMUM-KUPACBA-BESZÚR futási ideje egy n elemű kupacra $O(\lg n)$.

Összegezve, elmondhatjuk, hogy a kupac adatszerkezetet használva az els őbbségi sor bármelyik művelete n elemű halmazra $O(\lg n)$ id ő alatt elvégezhető.

Gyakorlatok

6.5-1. Ábrázoljuk az $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$ kupacon a KUPACB ÓL-KIVESZ-MAXIMUM eljárás működését.

6.5-2. Ábrázoljuk, hogyan működik a MAXIMUM-KUPACBA-BESZÚR$(A, 10)$ az $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$ kupacon. A 6.5. ábrát mintául véve mutassuk be, mi történik a KUPACBAN-KULCSOT-NÖVEL hívásakor.

6.5-3. Készítsük el a minimum-elsőbbségi sor megvalósításához szükséges KUPAC-MINIMUMA, KUPACBÓL-KIVESZ-MINIMUM, KUPACBAN-KULCSOT-CSÖKKENT és MINIMUM-KUPACBA-BESZÚR eljárások algoritmusait.

6.5-4. Miért fáradozunk az új elem kulcsának $-\infty$ értékre történő állításával a MAXIMUM-KUPACBA-BESZÚR eljárás 2. sorában, amikor a következő lépésben megváltoztatjuk a kívánt értékre?

6.5-5. Bizonyítsuk be KUPACBAN-KULCSOT-NÖVEL eljárás helyességét az alábbi ciklusinvariánst használva:

> A 4–6. sorokban levő **while** ciklus minden iterációjának kezdetén fennáll, hogy az $A[1\,..\,\textit{kupac-méret}[A]]$ tömb kielégíti a maximum-kupactulajdonságot, egy helyet kivéve: $A[i]$ értéke nagyobb lehet az $A[\text{SZÜLŐ}(i)]$ értékénél.

6.5-6. Mutassuk meg, hogyan implementálható az első-be, első-ki sor (FIFO) elsőbbségi sorral. És a verem? (A sor és a verem definícióját a 10.1. alfejezetben találjuk.)

6.5-7. A KUPACBÓL-TÖRÖL(A, i) törli az A kupacból az i-edik pozíción lévő elemet. Adjunk egy olyan algoritmust a KUPACBÓL-TÖRÖL műveletre, mely n elemű maximum-kupacon $O(\lg n)$ futási idejű.

6.5-8. Adjunk egy $O(n \lg k)$ idejű algoritmust, mely összefésül k db rendezett listát, ahol n az összes lista összes elemének együttes száma. (*Útmutatás.* Használjunk minimum-kupacot a k-asával való összefésülésre!)

Feladatok

6-1. Kupacépítés beszúrással

A 6.3. alfejezetben megadott MAXIMUM-KUPACOT-ÉPÍT eljárást megvalósíthatjuk beszúrásokkal. A MAXIMUM-KUPACBA-BESZÚR eljárás ismételt meghívásával beszúrjuk az elemeket a kupacba. Nézzük az alábbi megvalósítást:

```
MAXIMUM-KUPACOT-ÉPÍT′(A)
1  kupac-méret[A] ← 1
2  for i ← 2 to hossz[A]
3        do KUPACBA-BESZÚR(A, A[i])
```

a. Vajon MAXIMUM-KUPACOT-ÉPÍT és MAXIMUM-KUPACOT-ÉPÍT′ mindenkor azonos kupacot építenek, ha ugyanazon bemenő tömbön futtatjuk őket? Bizonyítsuk be, hogy igen, vagy mutassunk ellenpéldát.

b. Mutassuk meg, hogy a legrosszabb esetben a MAXIMUM-KUPACOT-ÉPÍT′ $\Theta(n \lg n)$ idő alatt épít fel egy n elemű kupacot.

6-2. d-rendű kupacok elemzése

Egy ***d-rendű kupac*** a bináris kupachoz hasonló, azzal a különbséggel, hogy a belső, nemlevél csúcsoknak (egy lehetséges kivétellel) nem kettő, hanem d gyereke van.

a. Hogyan ábrázolható egy d-rendű kupac egy tömbbel?

b. Adjuk meg egy n elemű, d-rendű kupacnak a magasságát d és n függvényeként.

c. Készítsünk hatékony megvalósítást a KIVESZ-MAXIMUM eljárásra d-rendű maximum-kupac esetében. Elemezzük a futási időt d és n függvényeként.

d. Készítsünk hatékony megvalósítást a BESZÚR eljárásra d-rendű maximum-kupac esetében. Elemezzük a futási időt d és n függvényeként.

e. Készítsünk hatékony megvalósítást a KUPACBAN-KULCSOT-NÖVEL(A, i, k) eljárásra, mely elsőként az $A[i] \leftarrow max(A[i], k)$ értékadást hajtja végre, majd megfelelő módon frissíti a d-rendű maximum-kupac adatszerkezetet. Elemezzük a futási időt d és n függvényeként.

6-3. Young-táblák

Egy $(m \times n)$-es ***Young-tábla*** egy olyan $(m \times n)$-es mátrix, melynek soraiban az értékek balról jobbra, az oszlopokban pedig felülről lefelé nagyság szerint rendezve vannak. A Young-táblában lehetnek ∞ értékek, ami a hiányzó elemeket jelzi. Innen adódik, hogy a Young-tábla $r \leq mn$ véges számot tartalmazhat.

a. Rajzoljunk egy (4×4)-es Young-táblát, mely a $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$ értékeket tartalmazza.

b. Igazoljuk, hogy egy Y $(m \times n)$-es Young-tábla üres, ha $Y[1, 1] = \infty$. Igazoljuk, hogy Y tele van (mn darab elemet tartalmaz), ha $Y[m, n] < \infty$.

c. Készítsük el a nem üres, $(m \times n)$-es Young-táblán működő KIVESZ-MINIMUM eljárás algoritmusát, úgy hogy futási ideje $O(m + n)$ legyen. Használjunk a megoldáshoz egy rekurzív alprogramot, mely az $(m \times n)$-es feladat megoldását rekurzív hívással visszavezeti egy $((m - 1) \times n)$-es vagy egy $(m \times (n - 1))$-es részfeladatra. (*Útmutatás.* Gondoljunk a MAXIMUM-KUPACOL eljárásra.) Definiáljuk a $T(p)$ függvényt $(p = m + n)$ úgy, hogy legyen a KIVESZ-MINIMUM futási idejének maximuma egy tetszőleges $(m \times n)$-es Young-táblán. Írjuk fel és oldjuk meg a $T(p)$ függvény rekurzív egyenletét oly módon, hogy eredményül az $O(m + n)$ időkorlátot kapjuk.

d. Mutassunk módszert, hogyan szúrható be egy még nem teli Young-táblába egy új elem $O(m + n)$ időben.

e. Mutassuk meg, hogyan rendezhető n^2 szám Young-tábla segítségével $O(n^3)$ időben anélkül, hogy bármely egyéb rendező alprogramot használnánk.

f. Adjunk $O(m + n)$ idejű algoritmust annak meghatározására, hogy egy megadott szám benne van-e egy adott $(m \times n)$-es Young-táblában.

Megjegyzések a fejezethez

A kupacrendezés algoritmus Williams [316] nevéhez fűződik, csakúgy mint a kupac felhasználása elsőbbségi sorok kezeléséhez. A MAXIMUM-KUPACOT-ÉPÍT algoritmust Floyd [90] javasolta.

Mimimum-kupacot használunk a 16., 23. és 24. fejezetekben a minimum-elsőbbségi sor megvalósításához. A 19. és 20. fejezetekben tovább javítjuk az időkorlátokat bizonyos műveletek megvalósításánál.

Az egész értékeket tartalmazó elsőbbségi sorokra gyorsabb megvalósítás adható. A P. van Emde Boas [301] által kifejlesztett adatszerkezetet alkalmazva a MINIMUM, a MAXIMUM, a BESZÚR, a TÖRÖL, a KERES, a KIVESZ-MINIMUM, a KIVESZ-MAXIMUM, az ELŐZŐ és a RÁKÖVETKEZŐ eljárások legrosszabb futási ideje $O(\lg\lg C)$, feltéve, hogy a kulcs-tartomány az $\{1, 2, \ldots, C\}$ halmaz. Fredman és Willard [99] megmutatták, hogyan valósítható meg a MINIMUM $O(1)$, a BESZÚR és a KIVESZ-MINIMUM $O(\sqrt{\lg n})$ időben feltéve, hogy az adatok b-bites egészek, és a számítógép memóriája is b-bites szavakból áll. Thorup [299] még tovább javította a $O(\sqrt{\lg n})$ korlátot az $O(\lg\lg n)$ futási időre. E korlát eléréséhez azonban a szükséges tárméret nagysága csak n nem korlátos függvényeként adható meg, viszont a megvalósításhoz véletlenített hasítást használva a szükséges tárméret nagysága $O(n)$.

Fontos speciális esete az elsőbbségi soroknak az, amikor a KIVESZ-MINIMUM műveletek által kapott sorozat ***monoton,*** azaz az időben egymást követő KIVESZ-MINIMUM műveletek által visszaadott értékek monoton növekvő sorozatot adnak. Ez számos fontos alkalmazásnál megfigyelhető, például Dijkstrának a „legrövidebb utak egy adott forrásból” problémát megoldó algoritmusánál (a 24. fejezetben tárgyaljuk), vagy a diszkrét-esemény szimulációnál. A Dijkstra-algoritmus megvalósításakor különösen fontos, hogy a KULCSOT-CSÖKKENT művelet hatékony legyen. Arra a monoton esetre, amikor az adatok az $1, 2, \ldots, C$ intervallumba eső egészek, Ahuja, Mehlhorn, Orlin és Tarjan [8] megmutatták, hogyan valósítható meg a KIVESZ-MINIMUM és a BESZÚR $O(\lg C)$ amortizációs költséggel (az amortizációs elemzést a 17. fejezetben tárgyaljuk), valamint a KULCSOT-CSÖKKENT $O(1)$ időben a számjegy kupac adatszerkezetet használva. Az $O(\lg C)$ korlát tovább javítható az $O(\sqrt{\lg C})$ értékre együtt használva a Fibonacci-kupacot (20. fejezet) és a számjegy kupacot. Cherkassky, Goldberg és Silverstein [58] tovább javították ezt az eredményt az $O(\lg^{1/3+\epsilon} C)$ értékre, kombinálva Denardo és Fox [72] többszintű edény adatszerkezetét és a fentebb említett Thorup-féle kupacot. Raman [256] továbbfejlesztésében ez a korlát az $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$ értékre csökken, tetszőleges rögzített $\epsilon > 0$ esetén. Ezen eredmény részletesebb tárgyalását találjuk Raman [256] és Thorup [299] cikkeiben.

7. Gyorsrendezés

A gyorsrendezés olyan rendezési algoritmus, amelynek futási ideje – n elemű bemenő tömbre – legrosszabb esetben $\Theta(n^2)$. Ennek ellenére a gyakorlatban sokszor érdemes a gyorsrendezést választani, mivel átlagos futási ideje nagyon jó: $\Theta(n \lg n)$, és a $\Theta(n \lg n)$ képlet rejtett állandói meglehetősen kicsik. Egy másik előnye, hogy helyben rendez (lásd a 2.1. alfejezetet), és virtuálismemória-környezetben is jól működik.

A 7.1. alfejezetben leírjuk az algoritmust, és megadunk egy fontos eljárást, melyet a gyorsrendezés a tömb felosztására használ. Mivel a gyorsrendezés bonyolult viselkedésű, először a 7.2. alfejezetben intuitív módon tárgyaljuk a teljesítményét, alapos elemzését pedig a fejezet végére hagyjuk. A 7.3. alfejezetben a gyorsrendezés olyan változatát mutatjuk be, amely véletlen mintát használ. Ennek az algoritmusnak jó az átlagos futási ideje, és egyetlen egyedi bemenetre sem éri el a legrosszabb futási időt. A véletlenített változatot a 7.4. alfejezetben tárgyaljuk, ahol megmutatjuk, hogy futási ideje legrosszabb esetben $\Theta(n^2)$, átlagosan pedig – különböző elemeket feltételezve – $O(n \lg n)$.

7.1. A gyorsrendezés leírása

A gyorsrendezés, az összefésülő rendezéshez hasonlóan, a 2.3.1. pontban bevezetett *oszd-meg-és-uralkodj* elven alapszik. Íme a háromlépéses *oszd-meg-és-uralkodj* algoritmus egy tipikus $A[p\,..\,r]$ résztömb rendezésére.

Felosztás: Az $A[p\,..\,r]$ tömböt két (esetleg üres) $A[p\,..\,q-1]$ és $A[q+1\,..\,r]$ résztömbre osztjuk úgy, hogy az $A[p\,..\,q-1]$ minden eleme kisebb vagy egyenlő $A[q]$-nál, ez utóbbi elem viszont kisebb vagy egyenlő $A[q+1\,..\,r]$ minden eleménél. A q index kiszámítása része ennek a felosztó eljárásnak.

Uralkodás: Az $A[p\,..\,q-1]$ és $A[q+1\,..\,r]$ résztömböket a gyorsrendezés rekurzív hívásával rendezzük.

Összevonás: Mivel a két résztömböt helyben rendeztük, nincs szükség egyesítésre: az egész $A[p\,..\,r]$ tömb rendezett.

Az algoritmus leírása a következő:

GYORSRENDEZÉS(A, p, r)

```
1  if p < r
2     then q ← FELOSZT(A, p, r)
3          GYORSRENDEZÉS(A, p, q − 1)
4          GYORSRENDEZÉS(A, q + 1, r)
```

Egy teljes tömb rendezésénél a kezdőhívás GYORSRENDEZÉS(A, 1, *hossz*[A]).

A tömb felosztása

Az algoritmus kulcsa a FELOSZT függvényeljárás, amely helyben átrendezi az $A[p, r]$ résztömböt.

FELOSZT(A, p, r)

```
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                A[i] ↔ A[j] csere
7  A[i + 1] ↔ A[r] csere
8  return i + 1
```

A 7.1. ábra a FELOSZT működését mutatja egy 8 elemű tömbön. A FELOSZT mindig kiválasztja az $x = A[r]$ ***őrszemet,*** amely körül az $A[p\,..\,r]$ tömböt felosztja. Az eljárás a tömböt folyamatosan négy (esetleg üres) tartományra osztja. A **for** ciklus (3–6. sorok) minden iterációjának kezdetén ezek a tömbök bizonyos, ciklusinvariánsoknak tekinthető tulajdonságokkal rendelkeznek.

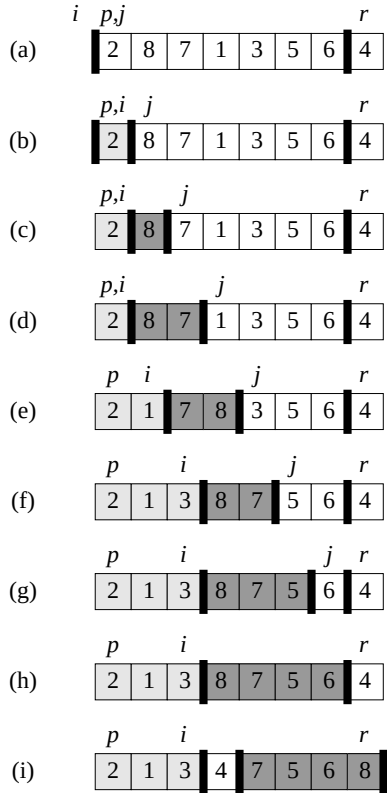
A 3–6. sorokban levő ciklus minden iterációjának kezdetén a k tömbindexre fennáll, hogy

1. Ha $p \le k \le i$, akkor $A[k] \le x$.
2. Ha $i + 1 \le k \le j - 1$, akkor $A[k] > x$.
3. Ha $k = r$, akkor $A[k] = x$.

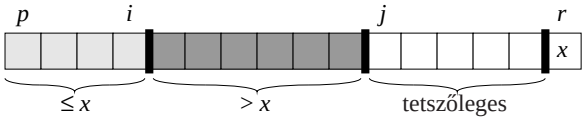
A 7.2. ábra összefoglalja ezt a struktúrát. A j és $r - 1$ közötti indexekre nem vonatkozik a fenti feltételek egyike sem, a nekik megfelelő elemek nincsenek semmilyen kapcsolatban az x őrszemmel.

Meg kell mutatnunk, hogy ez a ciklusinvariáns igaz az első iteráció előtt, és a ciklus minden iterációja megtartja invariánsnak, majd, hogy az invariáns hasznos tulajdonságnak bizonyul a helyesség bizonyítására, amikor a ciklus befejeződik.

Teljesül: A ciklus első iterációja előtt $i = p - 1$ és $j = p$. Mivel egyetlen elem sincs p és i között, sem pedig $i + 1$ és $j + 1$ között, a ciklusinvariáns első két feltétele triviálisan igaz. Az 1. sor értékadása miatt a harmadik feltétel is igaz.

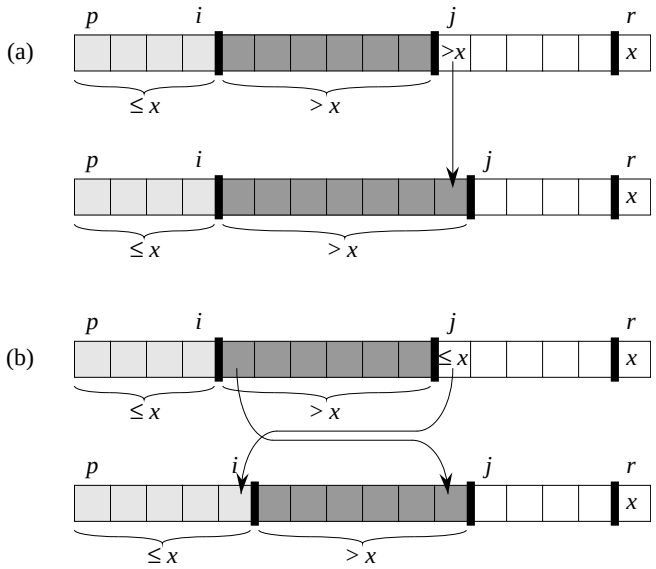


7.1. ábra. A FELOSZT eljárás működése egy példatömbön. A világosszürke elemek, amelyeknek értéke nem nagyobb, mint x, az első résztömbben vannak. A sötétszürkék a második részben vannak, és értékük nagyobb, mint x. A fehér elemek még nem kerültek be egyik részbe sem, a végső fehér elem az őrszem. **(a)** Az eredeti tömb és a változók értékadása. Még egyetlen elem se került a felosztás egyik részébe se. **(b)** A 2-t önmagával felcseréljük és betesszük a felosztás kisebb elemei közé. **(c)–(d)** A 8 és a 7 bekerül a felosztás nagyobb elemei közé. **(e)** Az 1 és 8 felcserélődik, a felosztás első része megnő. **(f)** A 3 és 7 felcserélődik, az első rész megnő. **(g)–(h)** A második, nagyobb elemeket tartalmazó rész megnő az 5 és 6 bekerülésével, és a ciklus befejeződik. **(i)** A 7. sorban az őrszem bekerül a két rész közé.



7.2. ábra. A FELOSZT eljárás által az $A[p\,..\,r]$ résztömbön kezelt négy tartomány. Az $A[p\,..\,i]$ elemei x-nél kisebbek vagy egyenlők vele, az $A[i+1\,..\,j-1]$ elemei mind nagyobbak x-nél, míg $A[r] = x$. Az $A[j\,..\,r-1]$ elemei tetszőlegesek.

Megmarad: Amint azt a 7.3. ábra mutatja, két esetet kell megkülönböztetnünk, mégpedig a 4. sorban végzett összehasonlítás eredményétől függően. A 7.3(a) ábra azt az esetet mutatja, amikor $A[j] > x$, és ekkor csupán a j értékét kell növelni. Amikor j értéke megnő, a második feltétel $A[j-1]$-re igaz, míg a többi elem változatlan marad. A 7.3(b) ábra azt mutatja, hogy amikor $A[j] \le x$, akkor i megnő, $A[i]$ és $A[j]$ felcserélődik, majd



7.3. ábra. A FELOSZT eljárás egy iterációjának két esete. **(a)** Ha $A[j] > x$, akkor csupán a j-t kell növelni, és ez a művelet megőrzi a ciklusinvariánst. **(b)** Ha $A[j] \leq x$, akkor az i értéke megnő, $A[i]$ és $A[j]$ értékeit felcseréljük, aztán j is megnő. A ciklusinvariáns marad.

j megnő. A csere miatt most $A[i] \leq x$, és az 1. feltétel teljesül. Hasonlóan, $A[j-1] > x$, mivel az az elem, amely az $A[j-1]$-be került, a ciklusinvariáns miatt nagyobb, mint x.

Befejeződik: Befejezéskor $j = r$, ezért a tömb minden eleme az invariáns által leírt valamelyik halmazban van. A tömb elemeit három halmazba tettük: az elsőben x-nél kisebbek vagy vele egyenlők vannak, a másodikban x-nél nagyobbak, és a harmadikban csak az x.

A FELOSZT két utolsó sora a helyére teszi az őrszemet, a tömb közepére, felcserélve őt a legbaloldalibb, x-nél nagyobb elemmel. A FELOSZT kimenete kielégíti a *felosztás* lépés specifikációját.

A FELOSZT eljárás futási ideje egy $A[p..r]$ tömb esetén $\Theta(n)$, ahol $n = r - p + 1$ (lásd a 7.1-3. gyakorlatot).

Gyakorlatok

7.1-1. A 7.1. ábrát mintaként használva, mutassuk meg a FELOSZT függvény működését az $A = (13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11)$ tömbön.

7.1-2. Milyen q értéket ad vissza a FELOSZT függvény, ha az $A[p..r]$ tömb minden eleme azonos? Módosítsuk a FELOSZT függvényt úgy, hogy $q = \lfloor (p+r)/2 \rfloor$ legyen, amikor a tömb minden eleme azonos.

7.1-3. Indokoljuk meg röviden, hogy a FELOSZT függvény futási ideje egy n elemű tömbre $\Theta(n)$.

7.1-4. Hogyan kell módosítani a GYORSRENDEZÉS eljárást, hogy csökkenő sorrendbe rendezzen?

7.2. A gyorsrendezés hatékonysága

A gyorsrendezés futási ideje függ attól, hogy a felosztás kiegyensúlyozott-e vagy sem, ez utóbbi pedig attól, hogy milyen elemeket választunk a felosztáshoz. Ha a felosztás kiegyensúlyozott, akkor az algoritmus aszimptotikusan olyan gyors, mint az összefésülő rendezés. Ha viszont a felosztás nem kiegyensúlyozott, akkor aszimptotikusan olyan lassú lehet, mint a beszúró rendezés. Ebben az alfejezetben megvizsgáljuk a gyorsrendezés teljesítményét a kiegyensúlyozott és a nem kiegyensúlyozott felosztás esetén.

Legrosszabb felosztás

A gyorsrendezés legrosszabb esete az, amikor a felosztó eljárás az eredeti tömböt egy $n-1$ és egy 0 elemű tömbre osztja. (Ezt az állítást a 7.4.1. pontban bizonyítjuk.) Tételezzük fel, hogy ez a kiegyensúlyozatlan felosztás az algoritmus minden lépésénél (minden rekurzív hívásnál) bekövetkezik. A felosztási idő $\Theta(n)$. A rekurzív hívás egy 0 nagyságú tömbre nem csinál egyebet, éppen csak visszatér, ezért $T(0) = \Theta(1)$, és a gyorsrendezés futási idejének rekurzív képlete

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n). \end{aligned}$$

Intuitív módon, ha összeadjuk a költségeket a rekurzió minden szintjén, akkor egy számtani sorozat összegét kapjuk (A.2. képlet), amely $\Theta(n^2)$. Valóban, ha alkalmazzuk a helyettesítő módszert, akkor a $T(n) = T(n-1) + \Theta(n)$ rekurziós összefüggés megoldása $T(n) = \Theta(n^2)$ (7.2-1. gyakorlat).

Tehát, ha a felosztás a rekurzív hívás minden lépésekor maximálisan kiegyensúlyozatlan, akkor a futási idő $\Theta(n^2)$. Így a gyorsrendezés futási ideje legrosszabb esetben nem jobb, mint a beszúró rendezésé. Mi több, a futási idő akkor is $\Theta(n^2)$, ha a bemeneti tömb már teljesen rendezett – ebben az esetben pedig a beszúró rendezés futási ideje $O(n)$.

Legjobb felosztás

A leggyakoribb felosztásnál a FELOSZT eljárás két, $n/2$ eleműnél nem nagyobb tömböt hoz létre, mivel az egyik $\lfloor n/2 \rfloor$, a másik pedig $\lceil n/2 \rceil - 1$ elemű. A futási idő rekurzív képlete

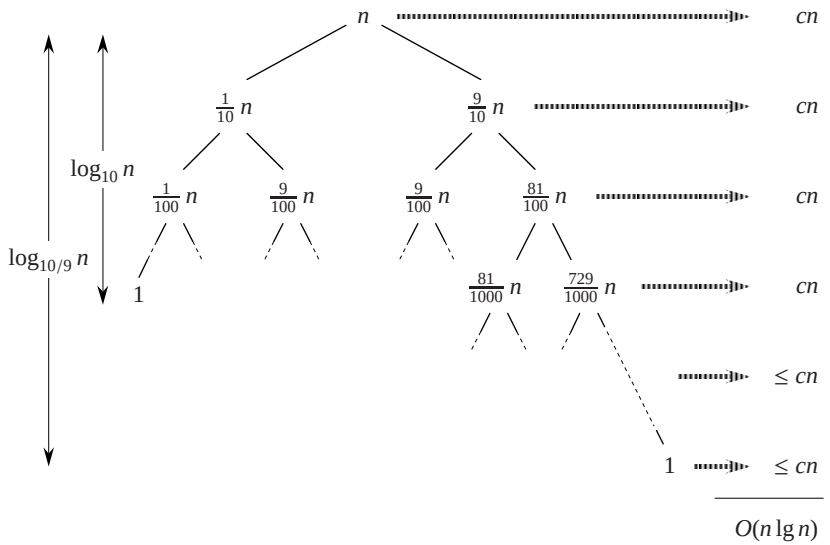
$$T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n),$$

amelynek megoldása a 4.1. tétel második esete alapján $T(n) = \Theta(n \lg n)$, így ez a legjobb felosztás aszimptotikusan gyorsabb algoritmust eredményez.

Kiegyensúlyozott felosztás

A gyorsrendezés átlagos futási ideje sokkal közelebb áll a legjobb, mint a legrosszabb futási időhöz, amint azt a 7.4. alfejezet elemzése bizonyítja majd. Ahhoz, hogy megérthessük, miért van ez így, meg kell vizsgálnunk, hogyan jelenik meg a felosztás kiegyensúlyozottsága a futási időt leíró rekurzív képletben.

Tételezzük fel, hogy a felosztó eljárás mindig 9 az 1-hez felosztást ad, amely kiegyensúlyozatlan felosztásnak tűnik. Ekkor a gyorsrendezés futási idejének rekurzív képlete



7.4. ábra. A GYORSRENDEZÉS rekurziós fája, amikor a FELOSZT eljárás mindig 9:1 arányú felosztást eredményez, és ekkor a futási idő $O(n \lg n)$. A csúcsok a részfeladatok nagyságát mutatják, jobb oldalon a megfelelő szintek költségével. A szintköltség eleve magában foglalja a c állandót a $\Theta(n)$ tagban.

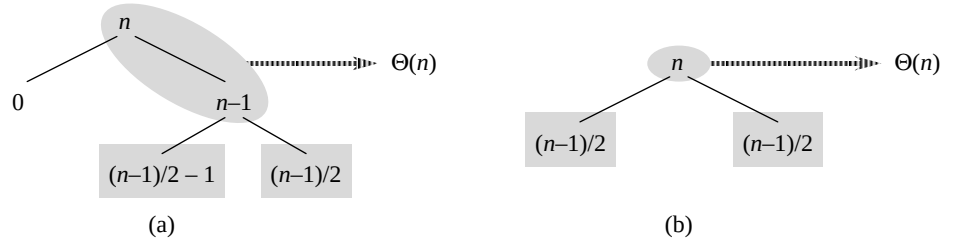
$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn,$$

ahol expliciten kiírtuk a $\Theta(n)$-ben rejlő c állandót. A megfelelő rekurzív fát a 7.4. ábra mutatja. Figyeljük meg, hogy a fa minden szintjének költsége cn, ameddig a $\log_{10} n = \Theta(\lg n)$ mélységben egy kezdeti feltételt el nem érünk. Az ezután következő szintek költsége legfeljebb cn. A rekurzió a $\log_{10/9} n = \Theta(\lg n)$ mélységben ér véget. A gyorsrendezés teljes költsége tehát $O(n \lg n)$. Ha tehát a felosztás a rekurzív hívás minden szintjén 9:1 arányú (amely kiegyensúlyozatlannak tűnik), a gyorsrendezés futási ideje $O(n \lg n)$ – aszimptotikusan azonos az egyenletes kétfelé osztás esetével. Tulajdonképpen a futási idő akkor is $O(n \lg n)$, ha minden felosztás 99:1 arányú. Ez azért van így, mert minden *állandó* arányú felosztáskor a rekurziós fa mélysége $\Theta(\lg n)$, és minden szinten a költség $O(n)$. A futási idő tehát $O(n \lg n)$ bármilyen állandó arányú felosztáskor.

Az átlagos viselkedés megsejtése

Ahhoz, hogy a gyorsrendezés átlagos viselkedését pontosan meghatározhassuk, egy feltevést kell megfogalmaznunk a bemenő tömbök gyakoriságáról. A gyorsrendezés viselkedése nem a bemenő elemektől, hanem azoknak az egymáshoz viszonyított helyétől függ. Akárcsak a munkatársfelvétel valószínűségi elemzésekor az 5.2. alfejezetben, a legnyilvánvalóbb feltételezés itt is az, hogy a bemeneti elemek minden permutációja ugyanolyan eséllyel fordulhat elő.

Amikor a gyorsrendezés véletlen bemenetekre fut, nem valószínű, hogy a felosztás minden szinten egyformán történik, ahogy azt az előbbi fejtegetéseinkben feltételeztük. Várható, hogy bizonyos felosztások kiegyensúlyozottak lesznek, mások pedig nem. Például,



7.5. ábra. (a) A gyorsrendezés rekurziós fájának két szintje. A tömb felosztása a gyökér szintjén n költségű, és *rossz* felosztás: egy 0 és egy $n-1$ elemű résztömböt eredményez. Az $n-1$ elemű résztömb felosztása $n-1$ költségű, és *jó* felosztás: egy $(n-1)/2-1$ és egy $(n-1)/2$ elemű résztömböt eredményez. **(b)** Egy egyszintű rekurziós fa, amely eléggé kiegyensúlyozott. Mindkét esetben a felosztás költsége $\Theta(n)$. A megmaradt részfeladatok, amelyeket szürke téglalapok jelölnek, az (*a*) esetben nem nehezebbek, mint a (*b*) esetben.

a 7.2-6. gyakorlatban azt kell bizonyítani, hogy a FELOSZT eljárás 80% körül 9:1 aránynál kiegyensúlyozottabb, míg 20% körül legfeljebb ennyire kiegyensúlyozott felosztást ad.

Általában a FELOSZT eljárás a „jó” és „rossz” felosztások keverékét adja. A FELOSZT közepes viselkedése esetén a rekurziós fában a jó és rossz felosztások véletlenszerűen jelennek meg. Az egyszerűség kedvéért tételezzük fel, hogy a rekurziós fában a jó és rossz felosztások váltakozva jelennek meg és, hogy a jó felosztások a legjobb, míg a rossz felosztások a legrosszabb felosztásnak felelnek meg. A 7.5(a) ábra a felosztást két egymás utáni szinten mutatja. A fa gyökerénél a felosztás költsége n, a keletkezett tömbök pedig $n-1$ és 0 eleműek: ez a legrosszabb eset. A következő szintnél az $n-1$ méretű tömböt a legjobb esetnek megfelelően egy $(n-1)/2-1$ és egy $(n-1)/2$ méretű résztömbre bontjuk. Tegyük fel, hogy a 0 elemű tömb költsége 1.

Egy rossz és utána egy jó felosztás három résztömböt hoz létre, rendre 0, $(n-1)/2$ és $(n-1)/2-1$ elemmel, összesen $\Theta(n)+\Theta(n-1)=\Theta(n)$ költséggel. Természetesen ez nem rosszabb, mint amit a 7.5(b) ábra mutat, azaz egy olyan egyszintű felosztás, amely két $(n-1)/2$ elemű tömböt hoz létre, összesen $\Theta(n)$ költséggel. Ez utóbbi már kiegyensúlyozott felosztás. Úgy tűnik tehát, hogy a $\Theta(n-1)$ költségű rossz felosztást elnyelheti egy $\Theta(n)$ költségű jó felosztás, és az eredmény jó felosztás. Tehát a gyorsrendezés futási ideje, amikor a jó és rossz felosztások váltakoznak, olyan mintha csak jó felosztások lennének: ugyancsak $O(n \lg n)$ idejű, csak nagyobb állandó szerepel az O-jelölésben. A gyorsrendezés egy véletlenített változata átlagos viselkedésének alapos elemzését a 7.4.2. pontban adjuk meg.

Gyakorlatok

7.2-1. Felhasználva a helyettesítő módszert, bizonyítsuk be, hogy a $T(n)=T(n-1)+\Theta(n)$ rekurzív összefüggés megoldása $T(n)=\Theta(n^2)$, ahogy azt a 7.2. alfejezet elején állítottuk.

7.2-2. Mennyi a GYORSRENDEZÉS futási ideje, ha az A tömb elemei azonosak?

7.2-3. Bizonyítsuk be, hogy ha az A tömb elemei csökkenő sorrendben vannak, a GYORSRENDEZÉS futási ideje $\Theta(n^2)$.

7.2-4. A bankok általában időrendi sorrendben jegyzik egy-egy ügyfél ügyleteit, az ügyfelek azonban szeretik, ha a banki kivonaton a csekkszám szerinti elrendezés szerepel, ők ugyanis a csekkeket sorrendben töltik ki. A kereskedők a begyűjtött csekkeket többnyire egyenlő időközönként küldik el. Az időrendi elrendezés átalakítása csekkszám szerinti

elrendezéssé tehát egy majdnem rendezett sorozat rendezése. Indokoljuk meg, hogy miért jobb ebben az esetben a BESZÚRÓ-RENDEZÉS, mint a GYORSRENDEZÉS.

7.2-5. Feltételezzük, hogy a gyorsrendezés minden szintjén a felosztás aránya $1 - \alpha$ az α-hoz, ahol $0 < \alpha \leq 1/2$ és α állandó. Bizonyítsuk be, hogy a rekurziós fában egy levél minimális mélysége körülbelül $-\lg n / \lg \alpha$, míg maximális mélysége körülbelül $-\lg n / \lg(1 - \alpha)$. (Ne törődjünk a kerekítésekkel!)

7.2-6.$\star$ Indokoljuk meg, hogy tetszőleges $0 < \alpha \leq 1/2$ állandó esetén a FELOSZT eljárás tetszőleges bemenetre $1-2\alpha$ valószínűséggel állít elő $(1-\alpha) : \alpha$ aránynál kiegyensúlyozottabb felosztást.

7.3. A gyorsrendezés egy véletlenített változata

A gyorsrendezés átlagos viselkedésének vizsgálatakor feltételeztük, hogy a bemeneti számok minden permutációjának ugyanaz a valószínűsége. Egy valóságos helyzetben azonban ezt nem várhatjuk el (lásd a 7.2-4. gyakorlatot). Amint azt az 5.3. alfejezetben láttuk, néha véletleníthetjük az algoritmust, hogy jó átlagos viselkedést érjünk el minden bemenetre. Sokan azt tartják, hogy a gyorsrendezés véletlenített változata nagy bemenetekre a lehető legjobb választás.

Az 5.3. alfejezetben úgy véletlenítettük az algoritmusunkat, hogy permutáltuk a bemenetet. Ezt ebben az esetben is megtehetjük, ellenben a véletlenítés egy másik változata, a ***véletlen mintavétel*** nevezetű, egyszerűbb elemzést biztosít. Őrszemnek nem mindig az $A[r]$ elemet tekintjük, hanem helyette az $A[p\,..\,r]$ résztömb egy véletlenszerűen kiválasztott elemét. Ezt úgy oldjuk meg egyszerűen, hogy az $A[r]$ elemet felcseréljük az $A[p\,..\,r]$ résztömb egy véletlenszerűen választott elemével. Ez a módosítás biztosítja, hogy az $x = A[r]$ őrszem ugyanolyan valószínűséggel lehet az $A[p\,..\,r]$ résztömb bármelyik eleme. Mivel az őrszemet véletlenszerűen választjuk ki, az átlagos viselkedésben a felosztás várhatóan jól kiegyensúlyozott lesz.

A FELOSZT és GYORSRENDEZÉS eljárások kevéssé módosulnak. Egyszerűen csak beépítjük két elem cseréjét az új felosztási eljárásba a felosztás előtt:

```
VÉLETLEN-FELOSZT(A, p, r)
1  i ← VÉLETLEN(p, r)
2  A[r] ↔ A[i] csere
3  return FELOSZT (A, p, r)
```

Az új gyorsrendezés a FELOSZT helyett a VÉLETLEN-FELOSZT eljárást használja:

```
VÉLETLEN-GYORSRENDEZÉS(A, p, r)
1  if p < r
2     then q ← VÉLETLEN-FELOSZT(A, p, r)
3          VÉLETLEN-GYORSRENDEZÉS(A, p, q − 1)
4          VÉLETLEN-GYORSRENDEZÉS(A, q + 1, r)
```

Az algoritmus elemzését a következő alfejezetben végezzük el.

Gyakorlatok

7.3-1. Egy véletlen algoritmusnak miért az átlagos viselkedését vizsgáljuk, és nem a legrosszabb esetét?

7.3-2. Legrosszabb esetben hányszor hívja meg futás közben a VÉLETLEN-GYORSRENDEZÉS eljárás a véletlenszám-generátort? Hogyan változik ez a szám, ha a legjobb esetet vizsgáljuk? A választ a Θ-jelölés segítségével adjuk meg.

7.4. A gyorsrendezés elemzése

A 7.2. alfejezetben megvizsgáltuk a gyorsrendezés legrosszabb viselkedésének néhány esetét, és azt, hogy miért feltételezzük, hogy gyors. Ebben az alfejezetben a gyorsrendezés viselkedésének pontosabb elemzését adjuk. A legrosszabb eset elemzésével kezdjük, amely alkalmazható mind a GYORSRENDEZÉS, mind a VÉLETLEN-GYORSRENDEZÉS esetén. Végül a VÉLETLEN-GYORSRENDEZÉS átlagos esetét vizsgáljuk.

7.4.1. A legrosszabb eset elemzése

Láttuk a 7.2. alfejezetben, hogy ha a gyorsrendezés minden szintjén a felosztás a lehető legrosszabb, akkor az algoritmus futási ideje $\Theta(n^2)$, és úgy látszott, hogy ez a legrosszabb eset. Most ezt az állítást be is bizonyítjuk.

A helyettesítő módszert alkalmazva (lásd a 4.1. alfejezetet) bebizonyíthatjuk, hogy a futási idő $O(n^2)$. Legyen $T(n)$ a gyorsrendezés legrosszabb futási ideje egy n elemű bemenet esetén. Ekkor

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n), \tag{7.1}$$

ahol a q paraméter végigfutja a 0 és $n - 1$ közötti értékeket, mivel a FELOSZT két résztömbje összesen $n - 1$ elemű. Feltételezzük, hogy $T(n) \le cn^2$ valamilyen c állandóra. Ezt behelyettesítve a (7.1)-be, a következőket kapjuk:

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) + \Theta(n). \end{aligned}$$

A $q^2 + (n - q - 1)^2$ kifejezés a maximumát a $0 \le q \le n - 1$ értékekre valamelyik végpontban éri el, mivel a q szerinti második deriváltja pozitív (lásd a 7.4-3. gyakorlatot). Ezért $\max_{0 \le q \le n-1}(q^2 + (n - q - 1)^2) \le (n - 1)^2 = n^2 - 2n + 1$. Folytatva $T(n)$ majorálását:

$$\begin{aligned} T(n) &\le cn^2 - c(2n - 1) + \Theta(n) \\ &\le cn^2, \end{aligned}$$

mivel a c állandó tetszőlegesen nagyra választható úgy, hogy a $c(2n - 1)$ meghaladja $\Theta(n)$-et. Tehát $T(n) = O(n^2)$. A 7.2. alfejezetben láttunk egy sajátos esetet, a kiegyensúlyozatlan felosztás esetét, amikor a gyorsrendezés futási ideje $\Omega(n^2)$. A 7.4-1. gyakorlat annak bizonyítását kéri, hogy a (7.1) rekurzív összefüggés megoldása $T(n) = \Omega(n^2)$. Így tehát a gyorsrendezés (legrosszabb) futási ideje $\Theta(n^2)$.

7.4.2. A várható futási idő

Már előzőleg indokoltuk, hogy a VÉLETLEN-GYORSRENDEZÉS átlagos futási ideje miért $O(n \lg n)$: ha a VÉLETLEN-FELOSZT minden lépésben ugyanolyan arányban osztja ketté a tömböt, a rekurziós fa mélysége $\Theta(\lg n)$, és a futási idő minden szinten $O(n)$. Még ha be is iktatunk olyan lépéseket, amelyek a lehető legkiegyensúlyozatlanabb felosztást eredményezik, a teljes futási idő $O(n \lg n)$. Pontosabban elemezhetjük a VÉLETLEN-GYORSRENDEZÉS várható futási idejét, ha előbb megértjük a felosztó eljárás lényegét. Azután ezt felhasználva, levezetjük az $O(n \lg n)$ korlátot a várható futási időre. Ez a felső korlát a várható értékre (feltételezve, hogy az elemek értékei különbözők) és a 7.2. alfejezetben látott $\Theta(n \lg n)$ érték a legjobb esetre a várható futási időre $\Theta(n \lg n)$ értéket ad.

Futási idő és összehasonlítások

A GYORSRENDEZÉS futási idejét nagyban befolyásolja a FELOSZT eljárás futási ideje. A FELOSZT minden hívásakor egy őrszemet jelölünk ki, amely aztán többé nem szerepel a GYORSRENDEZÉS és a FELOSZT egyetlen hívásakor sem. Tehát a gyorsrendezési algoritmus legfeljebb n-szer hívja meg a FELOSZT eljárást. A FELOSZT egy hívásának futási ideje $O(1)$, amelyhez még hozzájön a 3–6. sorokban levő **for** ciklus iterációinak a számával arányos idő. A **for** ciklus minden iterációja egy összehasonlítást végez a 4. sorban, összehasonlítva az őrszemet az A tömb egy másik elemével. Tehát, ha ki tudjuk számítani, hogy hányszor hajtjuk végre a 4. sort, akkor ki tudjuk számítani, hogy mennyi időt igényel a **for** ciklus végrehajtása a GYORSRENDEZÉS egész ideje alatt.

7.1. lemma. *Legyen X a* GYORSRENDEZÉS *egész futási ideje alatt a* FELOSZT 4. *sorában végrehajtott összehasonlítások száma, ha n elemű tömböt rendezünk. Ekkor a* GYORSRENDEZÉS *futási ideje $O(n + X)$.*

Bizonyítás. Az előbbi megjegyzések alapján, a FELOSZT eljárást n-szer hívjuk meg, ennek futási ideje egy állandó, amelyhez hozzáadódik a **for** ciklus végrehajtásából adódó idő. A **for** minden iterációjakor végrehajtjuk a 4. sort is. ■

Célunk, hogy kiszámítsuk X-et, az összehasonlítások számát a FELOSZT összes hívásában. Nem fogjuk megszámolni, hogy a FELOSZT egy-egy híváskor hány összehasonlítást végez, hanem inkább egy felső korlátot adunk az összehasonlítások összértékére. Hogy ezt megtehessük, szükséges tudnunk, hogy az algoritmus mikor hasonlít össze két elemet, és mikor nem. Az elemzés megkönnyítéséért nevezzük át az A tömb elemeit, legyenek ezek $z_1, z_2, \ldots, z_n$, ahol z_i az i-edik legkisebb elem. Ugyancsak értelmezzük a $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ halmazt, amely a z_i és z_j közötti elemeket tartalmazza, ezekkel bezárólag.

Mikor hasonlítja össze az algoritmus a z_i és z_j elemeket? Hogy megválaszolhassuk a kérdést, vegyük észre, hogy bármely két számpárt legfeljebb egyszer hasonlítjuk össze. Miért? Az elemeket csak az őrszemmel hasonlítjuk össze, és a FELOSZT egy hívása után az abban használt őrszemet többé már nem használjuk összehasonlításra.

Elemzésünk indikátor valószínűségi változót használ (lásd az 5.2. alfejezetet). Legyen

$$X_{ij} = \mathrm{I}\left\{z_i \text{ összehasonlítása } z_j\text{-vel}\right\},$$

ahol úgy tekintjük, hogy az összehasonlítás az algoritmus egészére vonatkozik, és nem csupán FELOSZT egyetlen hívására. Mivel bármely két elempárt legfeljebb egyszer hasonlítunk össze, az összehasonlítások számára a következő adódik:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Mindkét oldalnak vesszük a várható értékét, majd felhasználva a várható érték linearitását és az 5.1. lemmát, azt kapjuk, hogy

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}\left[X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\left\{z_i \text{ összehasonlítása } z_j\text{-vel}\right\}. \end{aligned} \tag{7.2}$$

Most már csak a $\Pr\left\{z_i \text{ összehasonlítása } z_j\text{-vel}\right\}$ kiszámítása van hátra. Elemzésünk felteszi, hogy az őrszemeket véletlenül és egymástól függetlenül választjuk.

Hasznos megnézni, hogy két elemet mikor *nem* hasonlítunk össze. Tekintsük a gyorsrendezés egy bemenetét, amely az 1 és 10 közötti számokat tartalmazza tetszőleges sorrendben, és legyen az őrszem 7. Ekkor a FELOSZT első hívása két részre osztja a bemenetet: $\{1, 2, 3, 4, 5, 6\}$ és $\{8, 9, 10\}$. Amikor ezt a felosztást végzi, a 7-et összehasonlítja az összes többi számmal, de az első halmaz egyetlen elemét (pl. 2) sem hasonlítja össze a második halmaz egyetlen elemével (pl. 9) sem.

Általában, mivel feltesszük, hogy az elemek értékei függetlenek. ha az x őrszemet úgy választjuk meg, hogy $z_i < x < z_j$, akkor tudjuk, hogy z_i és z_j többé nem kerülhet összehasonlításra. Ellenben, ha a z_i elemet választjuk őrszemnek a Z_{ij} halmaz bármelyik eleme előtt, akkor z_i-t összehasonlítjuk a Z_{ij} halmaz minden elemével, természetesen önmagán kívül. Példánkban a 7-et és 9-et összehasonlítjuk, mert 7 az első elem $Z_{7,9}$-ből, amelyik őrszem lesz. Ellenben a 2-t és 9-et soha nem hasonlítjuk össze, mert a $Z_{2,9}$-ben az első őrszem a 7. Tehát, z_i és z_j csak abban az esetben kerül összehasonlításra, ha az első őrszem a Z_{ij} halmazból z_i vagy z_j.

Most pedig kiszámítjuk ennek az eseménynek a valószínűségét. Először megjegyezzük, hogy az őrszemválasztás előtt a Z_{ij} halmaz teljes egészében ugyanabban a felosztásban van. Így a Z_{ij} minden eleme ugyanolyan valószínűséggel választható őrszemnek. Mivel a Z_{ij} halmaznak $j - i + 1$ eleme van, és az őrszemeket véletlenül és függetlenül választjuk meg, annak a valószínűsége, hogy a halmaz bármelyik eleme őrszem legyen, egyenlő $1/(j-i+1)$-gyel. Tehát

$$\begin{aligned} \Pr\left\{z_i \text{ összehasonlítása } z_j\text{-vel}\right\} &= \Pr\left\{z_i \text{ vagy } z_j \text{ az első őrszem } Z_{ij}\text{-ből}\right\} \\ &= \Pr\left\{z_i \text{ az első őrszem } Z_{ij}\text{-ből}\right\} \\ &\quad + \Pr\left\{z_j \text{ az első őrszem } Z_{ij}\text{-ből}\right\} \end{aligned}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$
$$= \frac{2}{j-i+1}\,. \tag{7.3}$$

A második és harmadik sort azért írhattuk fel, mert a két esemény kizárja egymást. A (7.2) és (7.3) képletek alapján

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}\,.$$

Ezt az összeget könnyen kiszámíthatjuk, ha változócserét vezetünk be ($k = j - i$), majd felhasználjuk a harmonikus sor felső korlátját (A.7. képlet).

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n). \qquad (7.4)
\end{aligned}$$

A következtetésünk tehát az, hogy VÉLETLEN-GYORSRENDEZÉS várható futási ideje – feltéve, hogy az elemek értékei különbözők – $O(n \lg n)$.

Gyakorlatok

7.4-1. Bizonyítsuk be, hogy a

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

rekurziós összefüggés megoldása $T(n) = \Omega(n^2)$.

7.4-2. Bizonyítsuk be, hogy a gyorsrendezés futási ideje legjobb esetben $\Omega(n \lg n)$.

7.4-3. Bizonyítsuk be, hogy a $q = 0, 1, \ldots, n-1$ értékeken definiált $q^2 + (n-q-1)^2$ a maximumát $q = 0$ vagy $q = n-1$ értékre veszi fel.

7.4-4. Bizonyítsuk be, hogy a VÉLETLEN-GYORSRENDEZÉS várható futási ideje $\Omega(n \lg n)$.

7.4-5. A gyakorlatban a gyorsrendezés futási ideje javítható, ha figyelembe vesszük a beszúró rendezés rövid futási idejét *majdnem* rendezett sorozatokra. Amikor a gyorsrendezés egy k-nál kevesebb elemű részsorozathoz ér, egyszerűen hagyja azt rendezetlenül. Majd amikor a gyorsrendezés befejeződött, futtassuk a beszúró rendezést az egész sorozatra. Indokoljuk meg, hogy ebben az esetben a várható futási idő $O(nk + n \lg(n/k))$. Hogyan kell megválasztani k értékét elméletben és gyakorlatban?

7.4-6. ⋆ Módosítsuk a FELOSZT eljárást úgy, hogy válasszunk ki véletlenszerűen három elemet az A tömbből, és ezek közül nagyságban a középső legyen az őrszem elem. Adjunk közelítő értéket annak a valószínűségére, hogy legrosszabb esetben egy $\alpha : (1-\alpha)$ felosztást kapjunk α függvényében, ahol $0 < \alpha < 1$.

Feladatok

7-1. Hoare felosztó algoritmusának helyessége

Az ebben a fejezetben tárgyalt felosztó algoritmus nem az eredeti változat. Az eredeti felosztó algoritmus C. A. R. Hoare-tól származik:

HOARE-FELOSZT(A, p, r)

```
 1  x ← A[p]
 2  i ← p − 1
 3  j ← r + 1
 4  while IGAZ
 5        do  repeat j ← j − 1
 6              until  A[j] ≤ x
 7            repeat i ← i + 1
 8              until  A[i] ≥ x
 9            if i < j
10              then A[i] ↔ A[j] csere
11              else  return j
```

a. Vizsgáljuk meg a HOARE-FELOSZT algoritmust az $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$ tömbön, megadva a tömb elemeit, valamint a segédértékeket a **while** ciklus (4–11. sorok) minden végrehajtása után.

A következő három kérdés a HOARE-FELOSZT eljárás helyességének bizonyítására vár alapos érvelést. Bizonyítsuk be a következőket.

b. Az i és j indexek olyanok, hogy az algoritmus sohasem hivatkozik az $A[p\,..\,r]$ résztömbön kívüli elemre.

c. Amikor a HOARE-FELOSZT befejeződik, a visszaadott j értékre igaz, hogy $p \le j < r$.

d. Amikor a HOARE-FELOSZT befejeződik, az $A[p\,..\,j]$ minden eleme kisebb vagy egyenlő az $A[j + 1\,..\,r]$ minden eleménél.

A 7.1. alfejezet FELOSZT eljárása az őrszemet (amely eredetileg $A[r]$) nem teszi be egyik keletkező résztömbbe sem. A HOARE-FELOSZT viszont az őrszemet (amely eredetileg $A[p]$) mindig az $A[p\,..\,j]$ vagy $A[j + 1\,..\,r]$ valamelyikébe helyezi. Mivel $p \le j < r$, a felosztás sohasem triviális.

e. Írjuk át GYORSRENDEZÉS algoritmust, ha az a HOARE-FELOSZT eljárást használja.

7-2. A gyorsrendezés egy másik elemzése

A véletlenített gyorsrendezés egy másik elemzése a GYORSRENDEZÉS minden egyes rekurzív hívásának a várható futási idejét vizsgálja az összehasonlítások helyett.

a. Indokoljuk meg, hogy egy n elemű tömb esetén annak a valószínűsége, hogy egy adott elem őrszem legyen, egyenlő $1/n$-nel. Felhasználva ezt, definiáljuk az $X_i = \mathrm{I}\{\text{az őrszem az } i\text{-edik legkisebb elem}\}$ indikátor valószínűségi változókat. Mennyi $\mathrm{E}[X_i]$?

b. Legyen $T(n)$ az a valószínűségi változó, amely egy n elemű tömb gyorsrendezési futási idejét jelöli. Bizonyítsuk be, hogy

$$\mathrm{E}\left[T(n)\right] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right]. \tag{7.5}$$

c. Bizonyítsuk be, hogy a (7.5) képlet átírható a következőképpen.

$$\mathrm{E}\left[T(n)\right] = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}\left[T(q)\right] + \Theta(n). \tag{7.6}$$

d. Mutassuk meg, hogy

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2. \tag{7.7}$$

(*Útmutatás.* Bontsuk két részre az összeget, egyik legyen a $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$ értékekre, a másik pedig a $k = \lceil n/2 \rceil, \ldots, n-1$ értékekre.)

e. A (7.7) korlátot felhasználva, igazoljuk, hogy a (7.6) megoldása $\mathrm{E}\left[T(n)\right] = \Theta(n \lg n)$. (*Útmutatás.* Igazoljuk helyettesítéssel, hogy $\mathrm{E}\left[T(n)\right] \leq an \log n$, ha n elég nagy és a megfelelő pozitív állandó.)

7-3. Cirkáló rendezés

Fred, Jimmy és Fred professzorok egy „elegáns” rendezési algoritmust javasoltak:

CIRKÁLÓ-RENDEZÉS(A, i, j)

```
1  if A[i] > A[j]
2     then A[i] ↔ A[j] csere
3  if i + 1 ≥ j
4     then return
5  k ← ⌊(j − i + 1)/3⌋                ▷ Lekerekítés
6  CIRKÁLÓ-RENDEZÉS(A, i, j − k)      ▷ Első kétharmad
7  CIRKÁLÓ-RENDEZÉS(A, i + k, j)      ▷ Utolsó kétharmad
8  CIRKÁLÓ-RENDEZÉS(A, i, j − k)      ▷ Első kétharmad újra
```

a. Igazoljuk, hogy a CIRKÁLÓ-RENDEZÉS($A, 1, n$) helyesen rendezi az $A[1\,..\,n]$ bemeneti tömböt, ahol $n = hossz[A]$.

b. Adjunk meg egy rekurzív képletet a CIRKÁLÓ-RENDEZÉS legrosszabb futási idejére, majd egy éles aszimptotikus korlátot (Θ-jelölést használva).

c. Hasonlítsuk össze a CIRKÁLÓ-RENDEZÉS legrosszabb futási idejét a beszúró, összefésülő, kupac- és gyorsrendezés legrosszabb esetével. Megérdemlik-e a professzorok az elismerést?

7-4. A gyorsrendezés veremmélysége

A 7.1. alfejezetben bemutatott GYORSRENDEZÉS algoritmus kétszer hívja rekurzívan önmagát. A FELOSZT hívása után előbb a bal oldali, majd a jobb oldali résztömböt rendezi rekurzívan. A második rekurzív hívás nem feltétlenül szükséges, ki lehet küszöbölni egy megfelelő iteratív struktúrával. Ezt a megoldást, amelyet ***jobbrekurziónak*** hívunk, a jó fordítóprogramok automatikusan beépítik a programba. A gyorsrendezés következő változata szimulálja a jobbrekurziót:

GYORSRENDEZÉS′(A, p, r)

```
1  while p < r
2        do          ▷ Felosztás és a bal oldali résztömb rendezése
3           q ← FELOSZT(A, p, r)
4           GYORSRENDEZÉS′(A, p, q − 1)
5           p ← q + 1
```

a. Igazoljuk, hogy GYORSRENDEZÉS′$(A, 1,$ *hossz* $[A])$ helyesen rendezi az A tömböt.

A fordítóprogramok a rekurzív hívásokat általában ***verem*** segítségével valósítják meg, amelyben a szükséges információkat őrzik, beleértve mindegyik rekurzív híváshoz a paraméterértékeket is. A legutóbbi hívás információi a verem tetején, míg a kezdeti hívás információi a verem alján találhatók. Amikor egy eljárást meghívunk, a megfelelő információk bekerülnek a ***verembe,*** amikor pedig az eljárás befejeződik, a neki megfelelő információk kikerülnek a ***veremből.*** Mivel feltételezhetjük, hogy a tömbparamétereket mutatókkal ábrázoljuk, minden eljáráshívásnak $O(1)$ veremhelyre van szüksége az információk tárolására. A ***veremmélység*** az a legnagyobb tárterület, amelyet a verem a feldolgozás során használ.

b. Írjunk le egy olyan helyzetet, amikor n elemű bemenet esetén a GYORSRENDEZÉS′ veremmélysége $\Theta(n)$.

c. Módosítsuk a GYORSRENDEZÉS′ algoritmust úgy, hogy a veremmélység a legrosszabb esetben $\Theta(\lg n)$ legyen. Maradjon meg az $O(n \lg n)$ várható érték az algoritmus futási idejére.

7-5. Háromból-a-középső felosztás

A VÉLETLEN-GYORSRENDEZÉS javításának egyik módja, hogy ahelyett hogy az x őrszemet véletlenszerűen vesszük, körültekintőbben választjuk meg. Egy ilyen egyszerű módszer az ún. ***háromból-a-középső:*** három véletlenszerűen vett elem közül válasszuk nagyságban a középsőt (lásd a 7.4-6. gyakorlatot). Tételezzük fel, hogy a bemeneti $A[1\,..\,n]$ tömb elemei mind különbözők, és $n \geq 3$. Jelöljük $A'[1\,..\,n]$-nel az eredménytömböt. Legyen $p_i = \Pr\{x = A'[i]\}$ annak a valószínűsége, hogy a ***háromból-a-középső*** módszer esetén az x őrszem az eredménytömb i-edik eleme legyen.

a. Adjunk meg egy képletet p_i értékére n és i függvényében, ahol $i = 2, 3, \ldots, n-1$. (Vegyük észre, hogy $p_1 = p_n = 0$.)

b. Mennyivel nő az eredeti algoritmushoz képest annak a valószínűsége, hogy a rendezett sorozat középső eleme (tehát $x = A'[\lfloor (n+1)/2 \rfloor]$) legyen az őrszem? Számítsuk ki ennek a valószínűségnek a határértékét, ha $n \to \infty$.

c. Ha *jó* felosztásnak nevezzük azt, ha az őrszem $x = A'[i]$, ahol $n/3 \leq i \leq 2n/3$, akkor mennyivel nő az eredeti algoritmushoz képest annak a valószínűsége, hogy jó felosztást kapjunk? (*Útmutatás.* Az összeget integrállal közelítsük meg.)

d. Igazoljuk, hogy a ***háromból-a-középső*** módszer a gyorsrendezés $\Omega(n \lg n)$ futási idejének csak a konstans tényezőjét befolyásolja.

7-6. Intervallumok fuzzy-rendezése

Tekintsünk egy olyan rendezési feladatot, amelyben a számokat nem ismerjük pontosan, csak egy-egy intervallumot, amelyhez tartoznak. Tehát, adott n darab $[a_i, b_i]$ zárt intervallum, ahol $a_i \leq b_i$. A célunk, hogy ***fuzzy-rendezéssel*** átrendezzük az intervallumokat,

azaz megadjuk az intervallumok egy $\langle i_1, i_2, \ldots, i_n \rangle$ permutációját úgy, hogy létezzenek a $c_i \in [a_{i_j}, b_{i_j}]$ számok, amelyekre $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Tervezzünk egy algoritmust n intervallum fuzzy-rendezésére. Az algoritmus struktúrájának olyannak kell lennie, hogy gyorsrendezze az intervallum bal oldali végpontjait (az a_i-ket), de ugyanakkor vegye figyelembe az átfedő intervallumokat, hogy javítsa a futási időt. (Ahogy az intervallumok egyre jobban átfedik egymást, a fuzzy-rendezés is egyre egyszerűbb lesz. Az algoritmus vegye figyelembe az ilyen átfedések előnyét.)

b. Igazoljuk, hogy az algoritmusunk várható futási ideje általában $\Theta(n \lg n)$, de lehet akár $O(n)$ is, ha az intervallumok teljesen átfedik egymást, azaz létezik x úgy, hogy $x \in [a_i, b_i]$ minden i-re. Az algoritmus ne ellenőrizze ezt az esetet, de természetes módon javuljon a teljesítménye, ahogy az átfedés egyre nagyobb lesz.

Megjegyzések a fejezethez

A gyorsrendezést Hoare [147] vezette be. Hoare változatát a 7-1. feladatban mutatjuk be. A 7.1. alfejezetbeli FELOSZT eljárás N. Lomutótól származik. A 7.4. alfejezet elemzése Avrim Blum műve. Sedgewick [268] és Bentley [40] alapos betekintést nyújt az algoritmus megvalósítási módozataiba.

McIlroy [216] megmutatta, hogyan lehet „gyilkos ellenfelet” készíteni, amely olyan bemenetet hoz létre, hogy azon a gyorsrendezés bármely változata $\Theta(n^2)$ időt igényel. Ha az algoritmus véletlenített, akkor az ellenfél a bemenetet azután állítja elő, hogy a gyorsrendezés kiválasztotta a véletlen elemet.

8. Rendezés lineáris időben

Az eddigiekben bemutattunk néhány olyan algoritmust, amelyek képesek $O(n \lg n)$ id őben rendezni n elemet. Az összefésüléses rendezés és a kupacrendezés ezt a fels ő határt a legrosszabb esetben éri el, a gyorsrendezésnek ez az átlagos futási ideje. Ráadásul ezeknél az algoritmusoknál megadhatunk olyan n elemű bemenetet, amelyre a futási id ő $\Omega(n \lg n)$.

Ezeknek az algoritmusoknak van egy érdekes, közös tulajdonságuk: *a rendezéshez csak a bemeneti elemek összehasonlítását használják*. Éppen ezért ezeket az algoritmusokat ***összehasonlító rendezéseknek*** nevezzük. Az összes eddig bemutatott algoritmus összehasonlító rendezés.

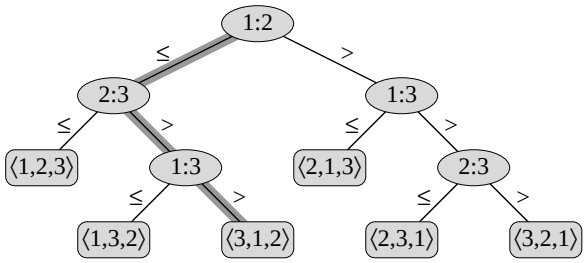
A 8.1. alfejezetben megmutatjuk, hogy bármely összehasonlító algoritmusnak a legrosszabb esetben $\Omega(n \lg n)$ összehasonlításra van szüksége n elem rendezéséhez. Következésképpen az összefésüléses rendezés és a kupacrendezés aszimptotikusan optimális, és minden összehasonlító rendezés legfeljebb egy állandó szorzóval lehet gyorsabb.

A 8.2., 8.3. és 8.4. alfejezetben három lineáris idejű rendez őalgoritmust mutatunk be, nevezetesen a leszámláló rendezést, a számjegyes rendezést és az edényrendezést. Talán mondanunk sem kell, hogy ezek az algoritmusok nem az összehasonlítást használják a rendezéshez, így az $\Omega(n \lg n)$ alsó korlát rájuk nem vonatkozik.

8.1. Alsó korlátok a rendezés időigényére

Összehasonlító rendezésnél ahhoz, hogy információkat szerezzünk az $\langle a_1, \ldots, a_n \rangle$ bemeneti sorozat rendezettségéről, csak elemek közötti összehasonlításokat használunk. Az a_i és a_j elemek esetén az $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ vagy $a_i > a_j$ tesztek egyikét végezzük el, hogy megtudjuk a két elem egymáshoz viszonyított sorrendjét. Az elemek értékét nem vizsgálhatjuk meg, és más módon sem kaphatunk róluk információt.

Ebben az alfejezetben az általánosság megszorítása nélkül feltételezzük, hogy az összes bemenő elem különböző. E feltételezés mellett szükségtelenné válik az $a_i = a_j$ típusú összehasonlítás, ezért feltételezzük, hogy ilyen nem is történik. Továbbá az $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ és $a_i < a_j$ összehasonlítások mind egyenértékűek abban az értelemben, hogy azonos információt szolgáltatnak az a_i és a_j egymáshoz viszonyított sorrendjéről. Éppen ezért feltételezzük, hogy az összes összehasonlítás $a_i \leq a_j$ alakú.



8.1. ábra. Döntési fa három elem beszúrásos rendezéséhez. Az $i{:}j$ párral jelölt belső csúcs az a_i és a_j elem összehasonlítását, a $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ permutációval jelölt levél pedig az $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ rendezést jelenti. Az árnyékolt útvonal az $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ bemeneti sorozat rendezése során meghozott döntéseket jelzi, a levélen lévő $\langle 3, 1, 2 \rangle$ permutáció pedig azt, hogy a sorba rendezés $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. A bemenő elemeknek $3! = 6$ permutációja lehetséges, így a döntési fának legalább 6 levelűnek kell lennie.

A döntésifa-modell

Az összehasonlító rendezéseket tekinthetjük ***döntési fáknak***. A döntési fa egy adott elemű bemenet rendezése során történt összehasonlításokat ábrázoló teljes bináris fa. A vezérléstől, az adatok mozgatásától és az algoritmus egyéb jellemzőitől eltekintünk. A 8.1. ábra a három elemű bemenet 1.1. alfejezetben ismertetett beszúrásos rendezésének döntési fáját mutatja.

A döntési fában minden belső csúcsot egy $i : j$ számpárral jelölünk, ahol $1 \leq i, j \leq n$, n pedig a bemenet elemeinek a száma. Mindegyik levél egy $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ permutációval jelölhető. (A permutációkról bővebben a C.1. alfejezetben olvashatunk.) A rendezési algoritmus végrehajtása megfelel a döntési fában a gyökértől valamely levélig vezető út bejárásának. A belső csúcsoknál elvégezzük az $a_i \leq a_j$ összehasonlítást. A csúcs bal oldali részfája az $a_i \leq a_j$ esetben szükséges későbbi összehasonlításokat mutatja, a jobb oldali részfa pedig az $a_i > a_j$ esetén teendőket. Amikor elérkezünk egy levélhez, az $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ sorrend a rendezési algoritmus által előállított rendezést jelöli. Mivel minden helyes rendezési algoritmusnak előt kell tudni állítania a bemeneti elemek összes permutációját, ezért a helyes rendezés szükséges feltétele, hogy az n elem összes $n!$ permutációjának meg kell jelennie a döntési fa levelei között, és ezen levelek mindegyikének elérhetőnek kell lennie a gyökérből egy összehasonlító rendezéshez tartozó úttal. (Ezeket a leveleket „elérhetőnek” nevezzük.) Vagyis csak olyan döntési fákat tekintünk, ahol minden permutáció megjelenik egy elérhető levélként.

Alsó korlát a legrosszabb esetre

A döntési fa gyökerét és legtávolabbi levelét összekötő út hossza adja meg azoknak az összehasonlításoknak a számát, melyeket a rendezőalgoritmus a legrosszabb esetben végez. Következésképpen, egy rendezőalgoritmus legrosszabb esetben elvégzett összehasonlításainak a száma megegyezik a hozzá tartozó döntési fa magasságával. Ezért az összes permutációt elérhető levélként tartalmazó döntési fák magasságára adott alsó korlát egyben az összehasonlító rendezőalgoritmusok futási idejének is alsó korlátja. A következő tétel egy ilyen alsó korlátot ad meg.

8.1. tétel. *Bármely összehasonlító rendezőalgoritmus a legrosszabb esetben* $\Omega(n \lg n)$ *összehasonlítást végez.*

Bizonyítás. A fentiek alapján elegendő egy olyan döntési fa magasságát meghatározni, amely az összes permutációt elérhető levélként tartalmazza. Vegyünk egy n elemet rendező döntési fát, amelynek magassága h, elérhető leveleinek száma pedig l. Mivel a bemenet összes $n!$ permutációja megjelenik levélként, ezért $n! \leq l$. Mivel egy h mélységű bináris fa leveleinek a száma nem lehet nagyobb, mint 2^h, ezért

$$n! \leq l \leq 2^h,$$

ahonnan mindkét oldal logaritmusát véve

$$\begin{aligned} h &\geq \lg(n!) && \text{(mivel a logaritmusfüggvény monoton növekvő)} \\ &= \Omega(n \lg n) && \text{((3.18) alapján).} \end{aligned}$$

■

8.2. következmény. *A kupacrendezés és az összefésüléses rendezés aszimptotikusan optimális összehasonlító rendezés.*

Bizonyítás. A kupacrendezés és az összefésüléses rendezés futási idejének $O(n \lg n)$ felső korlátja megegyezik a 8.1. tételben a legrosszabb esetre megadott $\Omega(n \lg n)$ alsó korláttal. ■

Gyakorlatok

8.1-1. Egy rendezőalgoritmushoz tartozó döntési fában mennyi egy levél legkisebb lehetséges mélysége?

8.1-2. Adjunk meg aszimptotikusan éles korlátot a $\lg n!$ kifejezésre a Stirling-formula használata nélkül. Helyette számítsuk ki a $\sum_{k=1}^{n} \lg k$ összeget, felhasználva az A.2. alfejezetben bemutatott módszereket.

8.1-3. Mutassuk meg, hogy nem létezik olyan összehasonlító rendezés, amelynek a futási ideje lineáris az $n!$ darab n hosszúságú bemenet legalább felére. Mit mondhatunk, ha az n elem $1/n$ részéről van szó? És ha az $1/2^n$ részéről?

8.1-4. Adott egy n elemű sorozat, amit rendezni kell. Tudjuk, hogy a bemeneti sorozat n/k részsorozatból áll, mindegyikben k elem van. Egy adott részsorozatban az elemek kisebbek, mint a következő részsorozat elemei és nagyobbak, mint az előző részsorozat elemei. Ezért ahhoz, hogy rendezzük az egész n hosszúságú sorozatot, elegendő rendezni az n/k darab részsorozatban levő k elemet. Mutassuk meg, hogy az ilyen típusú rendezési feladat megoldásához szükséges összehasonlítások számának $\Omega(n \lg k)$ alsó korlátja. (*Útmutatás.* Ez nem feltétlenül a résztömbök alsó korlátainak egyszerű összekombinálása.)

8.2. Leszámláló rendezés

A ***leszámláló rendezésnél*** feltételezzük, hogy az n bemeneti elem mindegyike 0 és k közötti egész szám, ahol k egy egész. Ha $k = O(n)$, a rendezés futási ideje $\Theta(n)$.

A leszámláló rendezés alapötlete az, hogy minden egyes x bemeneti elemre meghatározza azoknak az elemeknek a számát, amelyek kisebbek, mint az x. Ezzel az információval az x elemet közvetlenül a saját pozíciójába tudjuk helyezni a kimeneti tömbben. Ha például 17 olyan elem van, amelyik kisebb, mint az x, akkor az x elem a 18. helyen lesz a kimeneti tömbben. Ez a séma valamelyest megváltozik abban az esetben, amikor néhány elem egyenlő, hiszen nem akarjuk mindet ugyanabba a pozícióba helyezni.

A leszámláló rendezés kódjában feltételezzük, hogy a bemenet egy $A[1..n]$ tömb, melyre hossz$[A] = n$. Szükségünk van még két tömbre: a $B[1..n]$ tömb a rendezett kimeneti tömböt tartalmazza, a $C[0..k]$ tömb pedig átmeneti munkaterület.

LESZÁMLÁLÓ-RENDEZÉS(A, B, k)

```
 1 for i ← 0 to k
 2     do C[i] ← 0
 3 for j← 1 to hossz[A]
 4     do C[A[j]] ← C[A[j]] + 1
 5 ▷ C[i] most azoknak az elemeknek a számát tartalmazza, amelyek értéke i.
 6 for i ← 1 to k
 7     do C[i] ← C[i] + C[i − 1]
 8 ▷ C[i] most azoknak az elemeknek a számát tartalmazza,
     amelyek értéke kisebb vagy egyenlő, mint i.
 9 for j ← hossz[A] downto 1
10     do B[C[A[j]]] ← A[j]
11        C[A[j]] ← C[A[j]] − 1
```

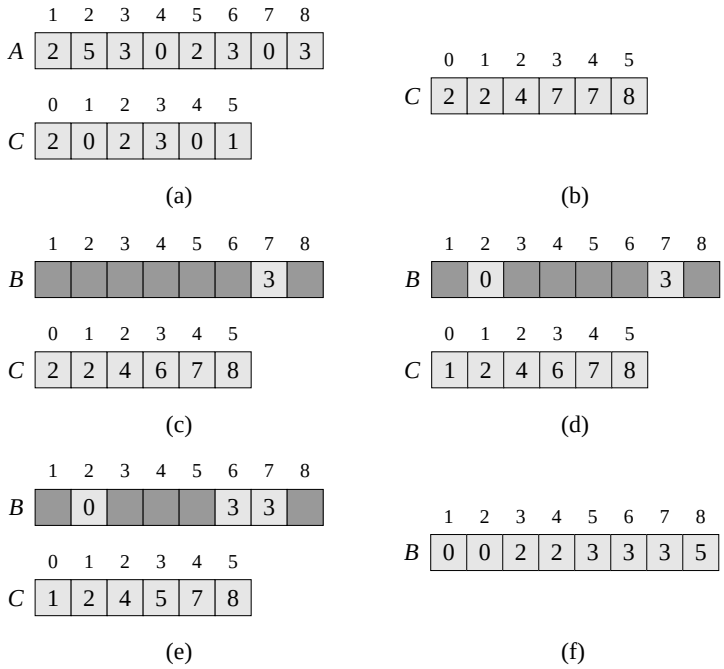
A leszámláló rendezést a 8.2. ábra mutatja. A kezdeti értékek 1–2. sorbeli **for** ciklusban történő beállítása után megvizsgáljuk az összes bemeneti elemet a 3–4. sorbeli **for** ciklusban. Ha egy elem értéke i, akkor megnöveljük a $C[i]$ értékét. Így a 4. sor után a $C[i]$ az i értékű elemeknek a számát tartalmazza, minden egyes $i = 0, 1, \ldots, k$ értékre. A 6–7. sorokban a C tömbön végigszaladó összegzéssel minden egyes $i = 0, 1, \ldots, k$ értékre meghatározzuk, hogy hány olyan bemeneti elem van, amelynek értéke kisebb vagy egyenlő, mint i.

Végül a 9–11. sorbeli **for** ciklusban minden egyes $A[j]$ elemet elhelyezünk a B kimeneti tömbbe, a megfelelő helyesen rendezett pozícióba. Ha mind az n elem különböző, akkor a 9. sor első érintésekor az összes $A[j]$ elemre $C[A[j]]$ lesz az $A[j]$ elem helyes végső pozíciója a kimeneti tömbben, mivel $C[A[j]]$ darab elem van, amelyik kisebb vagy egyenlő, mint $A[j]$. Mivel azonban az elemeknek nem kell eltérniük, ezért valahányszor beteszünk egy $A[j]$ értéket a B tömbbe, csökkentjük a $C[A[j]]$ értékét. A $C[A[j]]$ csökkentésével a következő $A[j]$-vel egyenlő elem (ha létezik ilyen) közvetlenül az $A[j]$ előtti pozícióba kerül a kimeneti tömbben.

Mennyi a leszámláló rendezés futási ideje? Az 1–2. sorokban levő **for** ciklus időigénye $\Theta(k)$, a 3–4. sorokban található **for** ciklus időigénye $\Theta(n)$, a 6–7. sorokban levő **for** ciklus időigénye $\Theta(k)$, a 9–11. sorokban levő **for** ciklus időigénye pedig $\Theta(n)$. Így, a teljes időigény $\Theta(k+n)$. A gyakorlatban akkor használunk leszámláló rendezést, ha $k = O(n)$, mely esetben a futási idő $\Theta(n)$.

A leszámláló rendezés futási ideje azért kisebb, mint a 8.1. alfejezetben bizonyított $\Omega(n \lg n)$ alsó korlát, mert ez nem összehasonlító rendezés. Valóban, a kódban sehol nem találunk a bemeneti tömb elemei közötti összehasonlítást. Ehelyett a leszámláló algoritmus az elemek értékét egy másik tömb indexeként használja fel. A rendezésre adott $\Omega(n \lg n)$ alsó korlát nem érvényes, ha eltérünk az összehasonlító rendezés modelljétől.

A leszámláló rendezés egy fontos tulajdonsága az, hogy ***stabil:*** az azonos értékű elemek ugyanabban a sorrendben jelennek meg a kimeneti tömbben, mint ahogyan a bemeneti tömbben szerepeltek. Azaz két szám között a kapcsolat megmarad azon szabály szerint, hogy amelyik szám először jelenik meg a bemeneti tömbben, az jelenik meg először a ki-



8.2. ábra. A LESZÁMLÁLÓ-RENDEZÉS működése egy olyan $A[1\,..\,8]$ bemeneti tömbön, amelynek elemei $k = 5$-nél nem nagyobb nemnegatív egészek. **(a)** Az A tömb és a C segédtömb a 4. sor után. **(b)** A C tömb a 7. sor után. **(c)–(e)** A B kimeneti tömb és a C segédtömb a 9–11. sorokban levő ciklus első, második, illetve harmadik iterációs lépése után. A B tömbnek csak a világos elemei kaptak értéket. **(f)** A végső, rendezett B kimeneti tömb.

meneti tömbben is. Általában a stabilitás csak akkor fontos tulajdonság, amikor a rendezendő elemek mellett kísérő adatok is vannak. A leszámláló rendezés stabilitása egy másik szempontból is fontos: a leszámláló rendezést gyakran felhasználjuk a számjegyes rendezés eljárásaként. Ahogyan ez a következő alfejezetben kiderül, a leszámláló rendezés stabilitása életbevágó a számjegyes rendezés helyes működéséhez.

Gyakorlatok

8.2-1. A 8.2. ábra alapján mutassuk be a LESZÁMLÁLÓ-RENDEZÉS működését az $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2\rangle$ tömbön.

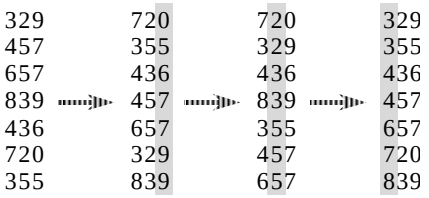
8.2-2. Bizonyítsuk be, hogy a LESZÁMLÁLÓ-RENDEZÉS stabil algoritmus.

8.2-3. Tegyük fel, hogy a LESZÁMLÁLÓ-RENDEZÉS eljárás 9. sorában a **for** ciklust a következőképpen átírtuk:

9 **for** $j \leftarrow 1$ **to** *hossz*$[A]$

Mutassuk meg, hogy az algoritmus így is megfelelően működik. Stabil-e a módosított algoritmus?

8.2-4. Adjunk meg olyan algoritmust, amelyik az adott, 0 és k közötti n darab egész számot előzetesen feldolgozza, majd $O(1)$ időben eldönti, hogy az n egész számból hány esik az $[a\,..\,b]$ intervallumba. Az algoritmus $\Theta(n + k)$ előfeldolgozási időt használhat.



8.3. ábra. A számjegyes rendezés működése 7 darab 3 számjegyű számot tartalmazó lista esetén. A bal szélső oszlop a bemenet. A többi oszlop a listát mutatja, az egyre fontosabb helyértékek alapján történt rendezések után. Az árnyékolás azt a számjegypozíciót mutatja, amelyet rendezve kaptuk meg az új listát az előzőből.

8.3. Számjegyes rendezés

A ***számjegyes rendezést*** a manapság már csak múzeumokban található lyukkártyarendező berendezéseknél használták. A kártyáknak 80 oszlopa van és minden oszlopban a 12 pozíció közül egy ki van lyukasztva. A rendező-berendezés mechanikusan „programozható” oly módon, hogy megvizsgálja a kártyacsomag minden lapjának egy adott oszlopát és szétosztja a lapokat 12 dobozba, aszerint, hogy melyik pozíció volt kilyukasztva. Ezután egy kezelő dobozról dobozra összegyűjtheti a lapokat úgy, hogy az első pozícióban kilyukasztott lapok a második pozícióban kilyukasztott lapok felett legyenek és így tovább.

Decimális számjegyek esetén csak 10 pozíció szükséges minden oszlopban. (A másik két pozíció nem numerikus karakterek kódolására használatos.) Így egy d számjegyű szám esetén d oszlopra van szükség. Mivel a kártyarendező csak egy oszlopot vizsgál egyszerre, ezért n darab d számjegyű számot tartalmazó lap rendezéséhez rendezőalgoritmusra van szükség.

Intuitív módon, valaki rendezheti a számokat a *legfontosabb* számjegy alapján, majd a keletkezett dobozokat rekurzív módon ugyanígy, a kártyacsomagokat a végén sorrendben egymás mellé téve. Sajnos, a 10 dobozból 9 doboz lapjait félre kell tenni ahhoz, hogy egy doboz tartalmát rendezni tudjuk, ezért ez az eljárás sok olyan átmeneti laprakást eredményez, amelyet később nyomon kell követni. (Lásd 8.3-5. gyakorlat.)

A számjegyes rendezés ezt a feladatot épp ellenkező módon oldja meg, mivel először a *legkevésbé fontos* számjegy alapján rendez. A lapokat egy csomagba rendezzük oly módon, hogy a 0. dobozból kivett kártyák megelőzik az 1. dobozból kivett kártyákat, amelyek megelőzik a 2. dobozból kivett kártyákat és így tovább. Ezután az egész csomagot újra rendezzük a második legkevésbé fontos számjegy alapján, és a kártyákat az előbbi módon összegyűjtjük. Ezt mindaddig végezzük, ameddig a kártyákat mind a d számjegy szerint nem rendeztük. Figyelemre méltó, hogy ezen a ponton a kártyákon levő d számjegyű számok teljesen rendezettek. Így csak d-szer kell átnézni a rendezéshez a kártyacsomagot. A 8.3. ábrán látható, hogyan működik a számjegyes rendezés olyan „csomagra”, amelyik 7 darab, 3 számjegyű számból áll.

Lényeges, hogy a számjegyek rendezése stabil legyen ebben az algoritmusban. A kártyarendező által végrehajtott rendezés stabil, de a kezelőnek vigyáznia kell, nehogy felcserélje a dobozból kivett kártyák sorrendjét, még akkor sem, ha a doboz összes kártyáján ugyanaz a számjegy szerepel az adott oszlopon.

Egy általános számítógépen, amelyik egy soros véletlen hozzáférésű gép, néha a számjegyes rendezést használják olyan információrekordok rendezésére, amelyeknek kulcsa több

mezőből áll. Tegyük fel például, hogy szeretnénk rendezni a dátumokat három kulcs szerint: év, hónap, nap. Futtathatunk olyan algoritmust, amelyik összehasonlításon alapul, azaz adott két dátum esetén összehasonlítja az éveket, amelyek ha egyenlőek, akkor összehasonlítja a hónapokat, és ha ezek is egyenlőek, akkor összehasonlítja a napokat. Egy másik módszerként egy stabil rendezőalgoritmussal háromszor rendezhetjük az információkat: először a napok szerint, aztán a hónapok szerint, végül az évek szerint.

A számjegyes rendezés kódja értelemszerű. A következő eljárás feltételezi, hogy az n elemű A tömb minden egyes eleme d jegyű, ahol az első számjegy a legalacsonyabb helyértékű számjegy és a d-edik számjegy a legmagasabb helyértékű számjegy.

SZÁMJEGYES-RENDEZÉS(A, d)

1 **for** i$\leftarrow$ 1 **to** d
2 **do** stabil algoritmussal rendezzük az A tömböt az i-edik számjegy szerint.

8.3. lemma. *Legyen adott n darab d jegyből álló szám, ahol a számjegyek legfeljebb k értéket vehetnek fel. Ekkor a* SZÁMJEGYES-RENDEZÉS $\Theta(d(n+k))$ *időben rendezi helyesen ezeket a számokat.*

Bizonyítás. A számjegyes rendezés helyessége a rendezendő oszlopon végzett indukcióból következik (lásd 8.3-3. gyakorlat). A futási idő függ a közbenső rendezőalgoritmusként alkalmazott stabil rendezőalgoritmus megválasztásától. Ha minden számjegy 0 és $k-1$ között van (azaz k lehetséges értéket vehet fel), és k nem túl nagy, a leszámláló rendezés a kézenfekvő választás. Így n darab d számjegyű szám esetén egy lépés $\Theta(n+k)$ időt igényel. Mivel d lépésünk van, a számjegyes rendezés teljes időigénye $\Theta(d(n+k))$. ■

Ha d állandó és $k = O(n)$, a számjegyes rendezés lineáris idejű. Általánosabban fogalmazva, a kulcsokat bizonyos rugalmassággal bonthatjuk számjegyekre.

8.4. lemma. *Legyen adott n darab b bites szám és egy tetszőleges $r \leq b$ pozitív egész. Ekkor a* SZÁMJEGYES-RENDEZÉS $\Theta((b/r)(n+2^r))$ *időben rendezi helyesen ezeket a számokat.*

Bizonyítás. Egy $r \leq b$ értékre minden kulcs $d = \lceil b/r \rceil$ darab egyenként r bites számjegyként tekinthető. Minden számjegy egy 0 és $2^r - 1$ közötti egész, ezért alkalmazhatjuk a leszámláló rendezést a $k = 2^r - 1$ paraméterrel. (Egy 32 bites szót tekinthetünk például 4 darab 8 bites számjegynek, azaz $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ és $d = b/r = 4$.) A leszámláló rendezés minden lépése $\Theta(n+k) = \Theta(n+2^r)$ ideig tart, így a d lépés megtételéhez szükséges teljes idő $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$. ■

Rögzített n és b számokhoz úgy szeretnénk megválasztani az r értékét ($r \leq b$), hogy az minimalizálja a $(b/r)(n+2^r)$ kifejezést. Ha $b < \lfloor \lg n \rfloor$, akkor minden $r \leq b$ esetén $(n+2^r) = \Theta(n)$. Így az $r = b$ választás $(b/b)(n+2^b) = \Theta(n)$ futási időt eredményez, ami aszimptotikusan optimális. Ha $b \geq \lfloor \lg n \rfloor$, akkor az $r = \lfloor \lg n \rfloor$ adja a legjobb időt egy állandó szorzótól eltekintve, ahogy ezt az alábbiakban látni fogjuk. Az $r = \lfloor \lg n \rfloor$ választás $\Theta(bn/\lg n)$ futási időt ad. Ha az r értéke nagyobb, mint $\lfloor \lg n \rfloor$, akkor a nevezőben szereplő 2^r érték gyorsabban nő mint a számlálóban az r, így ha az r értékét $\lfloor \lg n \rfloor$ fölé visszük, a futási idő $\Omega(bn/\lg n)$ lesz. Ha az r értékét $\lfloor \lg n \rfloor$ alá csökkentjük, akkor a b/r érték nő, míg az $n + 2^r$ értéke $\Theta(n)$ marad.

Kedvezőbb-e a számjegyes rendezés az összehasonlító rendező algoritmusoknál, így például a gyorsrendezésénél? Ha $b = O(\lg n)$, ami gyakran teljesül, és $r \approx \lg n$, akkor a számjegyes rendezés futási ideje $\Theta(n)$, ami jobbnak tűnhet, mint a gyorsrendezés $\Theta(n \lg n)$ átlagos futási ideje. A Θ függvényben található állandó szorzó azonban különbözik. Habár a számjegyes rendezés esetleg kevesebb lépést tesz az n kulcson, mint a gyorsrendezés, az egyes lépések jelentősen tovább tarthatnak számjegyes rendezés esetében. Hogy melyik algoritmus a jobb választás, függ a megvalósítástól, a futtatáshoz használt géptől (a gyorsrendezés például általában hatékonyabban használja ki a hardveres gyorsítótárat, mint a számjegyes rendezés) és a bemeneti adatoktól. Továbbá a leszámláló rendezést stabil közbenső rendezésként használó számjegyes rendezés nem helyben rendez, szemben sok $\Theta(n \lg n)$ idejű összehasonlító rendezéssel. Vagyis ha kevesebb elsődleges memória áll rendelkezésre, célszerűbb helyben rendező algoritmust (például gyorsrendezést) használni.

Gyakorlatok

8.3-1. A 8.3. ábra alapján mutassuk meg a SZÁMJEGYES-RENDEZÉS működését a következő szavak listáján: ÓRA, LAP, TEA, GÉP, ING, SZÓ, FÜL, ORR, LÁB, SOR, KÉS, ÁGY, VÍZ, RÁK, CÉL, FIÚ.

8.3-2. A következő algoritmusok közül melyek stabilak: beszúró rendezés, összefésüléses rendezés, kupac- és gyorsrendezés? Adjunk egy egyszerű módszert, amellyel minden rendezőalgoritmus stabillá tehető. Mennyi további idő- és tárigényt von ez maga után?

8.3-3. Indukcióval bizonyítsuk be, hogy a számjegyes rendezés helyesen működik. Hol szükséges a bizonyításban az a feltevés, hogy a közbenső rendezés stabil?

8.3-4. Hogyan rendezhető n darab, 0 és $n^2 - 1$ közötti egész szám $O(n)$ időben?

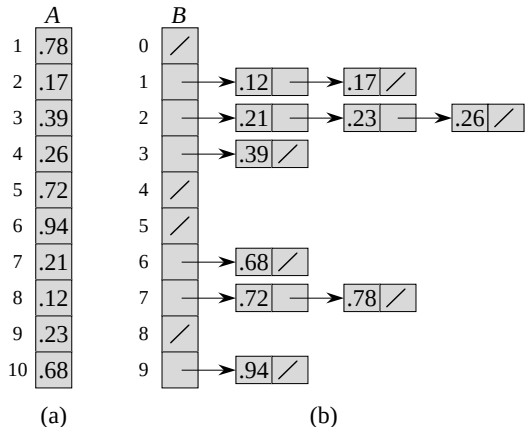
8.3-5.$\star$ Az alfejezet elején bemutatott, kártyalapokat rendező algoritmusnál a legrosszabb esetben pontosan hány rendező lépés szükséges d számjegyű decimális számok rendezéséhez? A kezelőnek hány darab laprakást kell nyomon követnie a legrosszabb esetben?

8.4. Edényrendezés

Az ***edényrendezés*** várható futási ideje lineáris, amennyiben a bemenet egyenletes eloszlásból származik. Éppúgy, mint a leszámláló rendezés, az edényrendezés is azért gyors, mert feltesz valamit a bemenetről. Míg a leszámláló rendezés azt feltételezi, hogy a bemenet olyan egészekből áll, amelyek egy kis intervallumba tartoznak, addig az edényrendezés azt, hogy a bemenetet egy olyan véletlen folyamat generálja, amelyik egyenletesen osztja el az elemeket a $[0, 1)$ intervallumon. (Az egyenletes eloszlás definícióját lásd a C.2. alfejezetben.)

Az edényrendezés alapötlete, hogy felosztja a $[0, 1)$ intervallumot n egyenlő részintervallumra, úgynevezett ***edényekre,*** és az n bemenő számot szétosztja az edényekbe. Mivel a bemenet egyenletes eloszlású a $[0, 1)$ intervallumon, ezért várhatóan nem fog egyetlen edénybe sem túl sok elem kerülni. Ahhoz, hogy a kimenetet előállítsuk, egyszerűen rendezzük az edényekben levő számokat, ezt követően végigmegyünk sorban az edényeken és kiírjuk minden elemüket.

Az edényrendezés általunk megadott kódja feltételezi, hogy a bemenet egy n elemű A tömb, és a tömb minden egyes $A[i]$ elemére teljesül, hogy $0 \leq A[i] < 1$. A kódban



8.4. ábra. Az EDÉNY-RENDEZÉS működése. **(a)** Az $A[1\,..\,10]$ bemeneti tömb. **(b)** A rendezett listák (edények) $B[0\,..\,9]$ tömbje az algoritmus 5. sora után. Az i-edik edény az $[i/10, (i+1)/10)$ félig nyílt intervallumhoz tartozó értékeket tartalmazza. A rendezett kimenet a $B[0], B[1], \ldots, B[9]$ listák sorban történő összefűzéséből áll elő.

szükség van továbbá egy, a láncolt listák (edények) $B[0\,..\,n-1]$ segédtömbjére, és feltesszük, hogy az ilyen listák kezeléséhez szükséges eljárások is rendelkezésünkre állnak. (A 10.2 alfejezetben megadjuk a láncolt listák alapműveleteinek megvalósítását.)

EDÉNY-RENDEZÉS(A)

1 $n \leftarrow hossz[A]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** szúrjuk be az $A[i]$ elemet a $B[\lfloor nA[i] \rfloor]$ listába.
4 **for** $i \leftarrow 0$ **to** $n-1$
5 **do** rendezzük a $B[i]$ listát beszúrásos rendezéssel.
6 sorban összefűzzük a $B[0], B[1], \ldots, B[n-1]$ listákat.

A 8.4. ábra egy 10 számból álló bemeneti tömbön mutatja be az edényrendezés működését.

Ahhoz, hogy lássuk az algoritmus helyes működését, tekintsünk két elemet, $A[i]$-t és $A[j]$-t. Tegyük fel az általánosság megszorítása nélkül, hogy $A[i] \leq A[j]$. Mivel $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, az $A[i]$ elem vagy ugyanabba az edénybe kerül, mint az $A[j]$, vagy egy kisebb indexű edénybe. Ha $A[i]$ és $A[j]$ ugyanabba az edénybe került, akkor a 4–5. sorban található **for** ciklus a helyes sorrendbe állítja őket. Ha $A[i]$ és $A[j]$ különböző edényekbe kerültek, akkor a 6. sor állítja a helyes sorrendbe őket. Tehát az edényrendezés helyesen működik.

A futási időt vizsgálva észrevehetjük, hogy az 5. sort kivéve az összes sor időigénye legrosszabb esetben $O(n)$. Marad még az 5. sorban levő n elem beszúrásos rendezése időigényének az elemzése.

A beszúrásos rendezés időigényének a megvizsgálásához legyen n_i az a valószínűségi változó, amelyik a $B[i]$ edénybe kerülő elemek számát jelöli. Mivel a beszúrásos rendezés futási ideje négyzetes (lásd 2.2. alfejezet), ezért az edényrendezés futási ideje

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Mindkét oldal várható értékét véve, és kihasználva a várható érték linearitását

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] && \text{(a várható érték linearitása miatt)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}\left[n_i^2\right]) && \text{(a (C.21) egyenlet alapján).} \qquad (8.1)
\end{aligned}$$

Azt állítjuk, hogy $i = 0, 1, \ldots, n-1$ esetén

$$\mathrm{E}\left[n_i^2\right] = 2 - \frac{1}{n}\,. \qquad (8.2)$$

Nem meglepő, hogy minden i edényhez ugyanaz az $\mathrm{E}\left[n_i^2\right]$ érték tartozik, mivel az A bemeneti tömb minden eleme ugyanakkora valószínűséggel eshet bármely edénybe. A (8.2) egyenlet bizonyításához az $i = 0, 1, \ldots, n-1$ és $j = 1, 2, \ldots, n$ esetekre

$$X_{ij} = \mathrm{I}\{A[j] \text{ az } i \text{ edénybe esik}\}$$

indikátor valószínűségi változókat definiálunk. Így

$$n_i = \sum_{j=1}^{n} X_{ij}.$$

Az $\mathrm{E}\left[n_i^2\right]$ kiszámításához elvégezzük a négyzetre emelést, és átcsoportosítjuk a tagokat:

$$\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} \mathrm{E}\left[X_{ij}X_{ik}\right], \qquad (8.3)
\end{aligned}$$

ahol az utolsó sor a várható érték linearitásából következik. A két összegzést külön értékeljük ki. Az X_{ij} indikátor valószínűségi változó 1 valószínűséggel $1/n$ értékű és 0 egyébként, ezért

$$\begin{aligned}\mathrm{E}\left[X_{ij}^2\right] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} \,.\end{aligned}$$

Ha $k \neq j$, akkor az X_{ij} és X_{ik} változók függetlenek, így

$$\begin{aligned}\mathrm{E}\left[X_{ij}X_{ik}\right] &= \mathrm{E}\left[X_{ij}\right]\mathrm{E}\left[X_{ik}\right] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} \,.\end{aligned}$$

Behelyettesítve a fenti két várható értéket a (8.3) egyenletbe,

$$\begin{aligned}\mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \neq j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1)\frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n}\end{aligned}$$

adódik, ami bizonyítja a (8.2) egyenletet.

A (8.1) egyenletben szereplő várható értéket használva megállapíthatjuk, hogy az edényrendezés várható ideje $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$. Vagyis a teljes edényrendezés lineáris várható időben fut le.

Még ha a bemenet nem is egyenletes eloszlásból származik, az edényrendezés futhat lineáris időben. Amíg a bemenet rendelkezik azzal a tulajdonsággal, hogy az edényméretek négyzeteinek összege lineáris a teljes elemszámban, addig a (8.1) egyenlet értelmében az edényrendezés futási ideje lineáris lesz.

Gyakorlatok

8.4-1. A 8.4. ábra alapján mutassuk meg az edényrendezés működését az $A = \langle 0{,}79; 0{,}13; 0{,}16; 0{,}64; 0{,}39; 0{,}20; 0{,}89; 0{,}53; 0{,}71; 0{,}42\rangle$ tömbön.

8.4-2. Mennyi az edényrendezés futási ideje legrosszabb esetben? Milyen egyszerű változtatással őrizhető meg a várható lineáris idő, és lesz $O(n \lg n)$ a legrosszabb futási idő?

8.4-3. Jelölje az X valószínűségi változó a fejek számát egy szabályos pénzérme kétszeri feldobása után. Mennyi az $\mathrm{E}\left[X^2\right]$, illetve az $\mathrm{E}^2[X]$ értéke?

8.4-4.⋆ Adott n pont az egységkörben, $p_i = (x_i, y_i)$ úgy, hogy $0 < x_i^2 + y_i^2 \le 1$, ahol $i = 1, 2, \ldots, n$. Feltételezzük, hogy a pontok egyenletes eloszlásúak, azaz annak a valószínűsége, hogy a kör valamelyik részén találjunk egy pontot, az adott rész területével arányos. Tervezzünk olyan $\Theta(n)$ várható idejű algoritmust, amelyik a n pontot az origótól mért

$d_i = \sqrt{x_i^2 + y_i^2}$ távolsága szerint rendezi. (*Útmutatás.* Az EDÉNY-RENDEZÉS algoritmusban az edények méretét tervezzük úgy, hogy tükrözzék a pontok egyenletes eloszlását a körben.)

8.4-5.⋆ Az X valószínűségi változó $P(x)$ ***eloszlásfüggvényét*** a következő módon definiáljuk: $P(x) = \Pr(X \leq x)$. Tegyük fel, hogy az n darab $X_1, X_2, \ldots, X_n$ valószínűségi változó folytonos P eloszlásfüggvénye $O(1)$ időben kiszámítható. Mutassuk meg, hogyan rendezhetők ezek a számok lineáris várható időben.

Feladatok

8-1. Összehasonlító rendezés átlagos futási idejére adott alsó korlátok

Ebben a feladatban bebizonyítunk egy $\Omega(n \lg n)$ alsó korlátot az n különböző bemeneti elemet rendező, determinisztikus és véletlenített összehasonlító rendezések várható futási idejére. Egy A determinisztikus összehasonlító rendezés vizsgálatával kezdjük, amelynek döntési fája F_A. Feltételezzük, hogy az A algoritmus bemeneti elemeinek összes permutációja egyformán valószínű.

a. Tegyük fel, hogy az F_A minden levelét megcímkéztük azzal a valószínűséggel, amellyel ez a levél egy adott bemenet esetén elérhető. Bizonyítsuk be, hogy pontosan $n!$ levél címkéje $1/n!$, a többi levél címkéje pedig 0.

b. Jelölje $D(F)$ az F döntési fa külső úthosszát, azaz $D(F)$ az F összes levele mélységének az összege. Legyen $k > 1$ az F döntési fa leveleinek a száma, és jelölje BF és JF az F fa bal, illetve jobb részfáját. Mutassuk meg, hogy $D(F) = D(BF) + D(JF) + k$.

c. Legyen $d(k)$ a $D(F)$ minimális értéke az összes $k > 1$ levelű F döntési fára nézve. Mutassuk meg, hogy $d(k) = \min_{1 \leq i \leq k-1}\{d(i) + d(k-i) + k\}$. (*Útmutatás.* Vegyünk egy k levelű F döntési fát, amelyik felveszi ezt a minimumot. Legyen i_0 a BF részfa leveleinek a száma, $k - i_0$ pedig a JF részfa leveleinek a száma.)

d. Bizonyítsuk be, hogy adott k és i értékekre, ahol $k > 1$ és $1 \leq i \leq k-1$, az $i \lg i + (k-i)\lg(k-i)$ függvény az $i = k/2$ értékre veszi fel a minimumát. Mutassuk meg, hogy $d(k) = \Omega(k \lg k)$.

e. Bizonyítsuk be, hogy $D(F_A) = \Omega(n! \lg(n!))$ és vezessük le, hogy n elem rendezésének várható ideje $\Omega(n \lg n)$.

Most pedig legyen B egy *véletlenített* összehasonlító rendezés. Kiterjeszthetjük a döntésifa-modellt a véletlenszerűség kezeléséhez úgy, hogy két típusú csúcsot engedünk meg: hagyományos összehasonlító csúcsot és „véletlen” csúcsot. Egy véletlen csúcs a B algoritmus véletlen, VÉLETLEN$(1, r)$ alakú választását modellezi; a csúcsnak r gyereke van, és az algoritmus egyforma valószínűséggel választ a csúcsok közül.

f. Mutassuk meg, hogy bármely B véletlenített összehasonlító rendezéshez létezik olyan A determinisztikus összehasonlító rendezés, amelyik átlagosan nem végez több összehasonlítást, mint a B algoritmus.

8-2. Lineáris idejű helyben rendezés

Tegyük fel, hogy egy n elemű adatrekordokból álló tömböt kell rendeznünk és mindegyik rekord kulcsa 0 vagy 1 értékű. Egy ilyen rekordhalmazt rendező algoritmus rendelkezhet az alábbi három hasznos tulajdonság némelyikével:

1. Az algoritmus futási ideje $O(n)$.
2. Az algoritmus stabil.
3. Az algoritmus helyben rendez, azaz az eredeti tömb tárolása mellett csak egy állandó méretű tárhelyre van szükség.

a. Adjunk meg olyan algoritmust, amely rendelkezik az 1. és 2. tulajdonsággal.

b. Adjunk meg olyan algoritmust, amely rendelkezik az 1. és 3. tulajdonsággal.

c. Adjunk meg olyan algoritmust, amely rendelkezik a 2. és 3. tulajdonsággal.

d. Lehet-e az (a)–(c) pontokban megadott rendezést $O(bn)$ időben n darab olyan rekord számjegyes rendezésére használni, melyek kulcsai b bit hosszúak? Indokoljuk meg, hogyan, vagy miért nem.

e. Tegyük fel, hogy az n darab rekord kulcsa 1 és k között van. Mutassuk meg, hogyan lehet a leszámláló rendezést úgy módosítani, hogy a rekordokat helyben rendezzük $O(n + k)$ időben. A bemeneti tömbön kívül $O(k)$ tárkapacitást használhatunk. Stabil ez az algoritmus? (*Útmutatás.* Hogyan csinálnánk ezt meg $k = 3$ esetén?)

8-3. Változó hosszúságú elemek rendezése

a. Adott egy egészeket tartalmazó tömb, ahol a különböző egészek különböző számú számjegyből állhatnak, de a tömbben található *összes* egészet leíró számjegyek száma n. Mutassuk meg, hogyan lehet rendezni a tömböt $O(n)$ időben.

b. Adott egy karakterláncokat tartalmazó tömb, ahol a különböző karakterláncok különböző számú karakterből állhatnak, de a tömbben található összes karakterláncot leíró karakterek száma n. Hogyan lehet rendezni a karakterláncokat $O(n)$ időben?

(Figyeljünk arra, hogy itt a kívánt sorrend az ábécé szerint növekvő sorrend; például, a < ab < b.)

8-4. Vizeskancsók

Legyen adva n darab piros és n darab kék, különböző formájú és méretű vizeskancsó. Minden piros és minden kék kancsóba különböző mennyiségű víz fér. Továbbá, minden piros kancsóhoz található egy ugyanakkora térfogatú kék kancsó, és fordítva.

A feladat az ugyanakkora térfogatú piros és kék kancsók párokba rendezése. Ehhez használhatjuk a következő eljárást: tekintsünk egy piros és kék kancsóból álló párt, töltsük meg vízzel a piros kancsót, majd öntsük azt át a kék kancsóba. Ezzel a módszerrel eldönthetjük, hogy a piros vagy kék kancsóba fér-e több víz, illetve hogy a térfogatuk azonos. Tegyük fel, hogy egy ilyen összehasonlító lépés egységnyi ideig tart. A cél olyan algoritmus keresése, amely minimális számú összehasonlítást végez az összetartozó párok meghatározásához. Emlékezzünk rá, hogy két piros vagy két kék kancsó közvetlenül nem hasonlítható össze.

a. Adjunk meg egy olyan determinisztikus algoritmust, amelyik $\Theta(n^2)$ összehasonlítással párokba rendezi a kancsókat.

b. Bizonyítsuk be, hogy egy, a problémát megoldó algoritmus esetén $\Omega(n \lg n)$ az alsó határa a szükséges összehasonlítások számának.

c. Adjunk meg egy olyan véletlenített algoritmust, amelyre az összehasonlítások várható száma $O(n \lg n)$, majd mutassuk meg, hogy ez a korlát helyes. Mennyi az összehasonlítások száma a legrosszabb esetben erre az algoritmusra?

8-5. Átlagos rendezés

Tegyük fel, hogy egy tömb rendezése helyett csak azt követeljük meg, hogy az elemek átlaga növekvő legyen. Pontosabban fogalmazva, egy n elemű A tömböt k-***rendezettnek*** nevezünk, ha minden $i = 1, 2, \ldots, n-k$ esetén

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. Mit jelent az, ha egy tömb 1-rendezett?

b. Adjuk meg az $1, 2, \ldots, 10$ elemek egy 2-rendezett, de nem rendezett permutációját.

c. Mutassuk meg, hogy egy n elemű tömb pontosan akkor k-rendezett, ha $A[i] \leq A[i+k]$ minden $i = 1, 2, \ldots, n-k$ esetén.

d. Adjunk meg olyan algoritmust, amely egy n elemű tömböt $O(n \lg(n/k))$ idő alatt k-rendez.

Egy k állandó esetén megadható alsó korlát egy tömb k-rendezésének idejére.

e. Mutassuk meg, hogy egy n-hosszú k-rendezett tömb $O(n \lg k)$ időben rendezhető. (*Útmutatás.* Használjuk fel a 6.5-8. gyakorlat megoldását.)

f. Mutassuk meg, hogy egy k állandó esetén $\Omega(n \lg n)$ időre van szükség egy n elemű tömb k-rendezésére. (*Útmutatás.* Használjuk fel az előző rész megoldását és a leszámláló rendezésekre vonatkozó alsó korlátot.)

8-6. Rendezett listák összefésülésének alsó korlátja

Gyakran felmerül két rendezett lista összefésülésének a problémája. A két rendezett listát összefésülő, az ÖSSZEFÉSÜLÉSES-RENDEZÉS alprogramjaként használt ÖSSZEFÉSÜLÉS eljárást a 2.3.1. pontban írtuk le. Ennél a problémánál belátjuk, hogy két n elemet tartalmazó rendezett lista összefésülése esetén $2n-1$ az alsó korlátja a szükséges összehasonlítások számának a legrosszabb esetben.

Egy döntési fa segítségével az összehasonlítások számára először egy $2n - o(n)$ alsó korlátot mutatunk meg.

a. Bizonyítsuk be, hogy $2n$ adott számot $\binom{2n}{n}$ lehetséges módon sorolhatunk be két n elemű rendezett listába.

b. Egy döntési fa segítségével mutassuk meg, hogy két rendezett listát helyesen összefésülő algoritmus legalább $2n - o(n)$ összehasonlítást végez.

Most a kicsit élesebb $2n-1$ korlátot mutatjuk meg.

c. Bizonyítsuk be, hogy ha két eltérő listában található elem egymást követi a rendezés szerint, akkor ezeket össze kell hasonlítani.

d. Az előző részre adott válasz alapján mutassuk meg, hogy a $2n-1$ összehasonlítás alsó korlát a két rendezett lista összefésülésére.

Megjegyzések a fejezethez

Az összehasonlító rendezésekre a döntésifa-modellt Ford és Johnson [94] vezette be. Knuth rendezésről szóló átfogó tanulmánya [185] a rendezési probléma több változatát lefedi, és

a rendezőalgoritmusok bonyolultságának itt megadott információelméleti alsó korlátját is tartalmazza. A döntési fa általánosítását használó rendezések alsó korlátját Ben-Or [36] tárgyalja átfogóan.

Knuth szerint H. H. Seward találta ki a leszámláló rendezést 1954-ben, és a számjegyes rendezés leszámláló rendezéssel való kombinálásának ötletét. A legkevésbé fontos számjegy szerint történő számjegyes rendezés egy széles körben elterjedt módszer volt a mechanikus lyukkártyarendezők kezelői körében. Knuth szerint erre a módszerre az első publikált hivatkozás 1929-ben jelent meg L. J. Comrie kártyalyukasztókról szóló leírásában. Az edényrendezést 1956 óta használjuk E. J. Isaac és R. C. Singleton ötlete alapján.

Munro és Raman [229] olyan stabil rendezőalgoritmust adtak meg, amely a legrosszabb esetben $O(n^{1+\epsilon})$ összehasonlítást végez, ahol $0 < \epsilon \leq 1$ tetszőleges rögzített állandó. Habár a $O(n \lg n)$ idejű algoritmusok mindegyike kevesebb összehasonlítást végez, Munro és Raman algoritmusa csak $O(n)$-szer mozgat adatot és helyben rendez.

Sok kutató foglalkozott az n darab b bites egész $o(n \lg n)$ idejű rendezésével. Számos pozitív eredmény született, amelyek a számítási modellre vonatkozó feltételezésekben és az algoritmusra megadott korlátozásokban különböztek valamelyest. Minden eredmény feltételezi, hogy a számítógépes memória fel van osztva b bites címezhető szavakra. Fredman és Willard [99] bevezette az egyesítési fa adatstruktúrát, és azt használva rendezte az n darab egészet $O(n \lg n / \lg \lg n)$ idő alatt. Ezt a korlátot később Andersson [16] $O(n \sqrt{\lg n})$-re javította. Ezek az algoritmusok szorzások használatát és bizonyos állandók előre történő kiszámítását követelik meg. Andersson, Hagerup, Nilsson és Raman [17] megmutatta, hogyan lehet szorzás nélkül n darab egészet rendezni $O(n \lg \lg n)$ idő alatt, ám az eljárásuk n-ben nem korlátos tárhely használatát feltételezheti. Multiplikatív elhelyező eljárást használva a tárhely mérete $O(n)$-re csökkenthető, viszont ekkor a legrosszabb esetre vonatkozó $O(n \lg \lg n)$ időkorlát a várható időt fogja jelenteni. Az Andersson [16] által bevezetett exponenciális keresőfák általánosításával Thorup [297] megadott egy $O(n(\lg \lg n)^2)$ idejű, lineáris helyigényű, szorzást nem használó, nem véletlenített rendezőalgoritmust. Ezeket a technikákat új elképzelésekkel kiegészítve a rendezés időkorlátját Han [137] $O(n \lg \lg n \lg \lg \lg n)$-re javította. Habár ezek az algoritmusok komoly elméleti jelentőséggel bírnak, meglehetősen bonyolult voltuk miatt nem tűnnek versenyképesnek a gyakorlatban jelenleg használt rendezőalgoritmusokkal szemben.

9. Mediánok és rendezett minták

Egy n elemű halmaz esetén a ***rendezett minta*** i-edik eleme a halmaz i-edik legkisebb eleme. Például, egy halmaz ***minimuma*** a halmaz rendezett mintájának első eleme ($i = 1$), míg a ***maximuma*** a rendezett minta n-edik eleme ($i = n$). A ***medián***, általánosan fogalmazva tulajdonképpen a halmaz „középpontja”. Ha n páratlan, akkor a medián egyedi, és az $i = (n + 1)/2$ pozícióban jelenik meg. Ha n páros, akkor két medián létezik, amelyek az $i = n/2$ és az $i = n/2 + 1$ pozíciókban jelennek meg. Tehát, az n paritásától függetlenül mediánok az $i = \lfloor (n + 1)/2 \rfloor$ pozícióban ***(alsó medián)*** és az $i = \lceil (n + 1)/2 \rceil$ pozícióban ***(felső medián)*** jelennek meg. Az egyszerűség kedvéért a továbbiakban következetesen a medián kifejezést használjuk majd és ez az alsó mediánra utal.

Ez a fejezet azzal foglalkozik, hogy miképpen választható ki az n különböző számot tartalmazó halmazból a rendezett minta i-edik eleme. Kényelmi okokból feltételezzük, hogy a halmaz elemei különböző számok, habár gyakorlatilag minden, amit teszünk, kiterjeszthető olyan esetre is, amikor egy halmaz ismétlődő értékeket is tartalmaz. A ***kiválasztási feladat*** formálisan a következőképpen fogalmazható meg:

Bemenet: Az n (különböző) számból álló A halmaz és egy i szám, amelyre fennáll, hogy $1 \leq i \leq n$.

Kimenet: Az $x \in A$ elem, amelyik nagyobb, mint az A pontosan $i - 1$ másik eleme.

A kiválasztási feladat $O(n \lg n)$ időben megoldható, hiszen rendezhetjük az elemeket kupacrendezéssel vagy összefésüléses rendezéssel, és ezt követően a kimeneti tömbben egyszerűen rámutatunk az i-edik elemre. Léteznek azonban ennél gyorsabb algoritmusok is.

A 9.1. alfejezetben egy halmaz minimumának és maximumának kiválasztási problémáját tanulmányozzuk. Ennél azonban sokkal érdekesebb az általános kiválasztási feladat, amelyet a következő két alfejezetben vizsgálunk meg. A 9.2. alfejezet egy olyan gyakorlati algoritmust elemez, amelynek az átlagos futási ideje – különböző elemeket feltételezve – $O(n)$. A 9.3. alfejezet egy inkább elméleti szempontból érdekes algoritmust tartalmaz, amelynek $O(n)$ a legrosszabb futási ideje.

9.1. Minimális és maximális elem

Hány összehasonlítás szükséges egy n elemű halmaz minimumának meghatározásához? Könnyedén megkaphatunk egy $n - 1$ összehasonlításból álló felső korlátot: sorban megvizs-

gáljuk a halmaz összes elemét, és mindig megjegyezzük az addig megvizsgált elemekből a legkisebbet. A következő eljárásban feltételezzük, hogy a halmazt az A tömbbel ábrázoljuk, ahol $hossz[A] = n$.

MINIMUM(A)

```
1  min ← A[1]
2  for i ← 2 to hossz[A]
3      do if min > A[i]
4           then min ← A[i]
5  return min
```

Természetesen a maximum meghatározása is ugyanígy elvégezhető $n-1$ összehasonlítással.

Vajon ez a legjobb, amit tehetünk? Igen, hiszen a minimum-meghatározásnak megkapható egy alsó korlátja, ami $n-1$ összehasonlítás. Képzeljünk el valamennyi minimumkereső algoritmust úgy, mint egy versenyt az elemek között. Mindegyik összehasonlítás a verseny egy olyan mérkőzése, ahol a két elem közül a kisebb a győztes. A legfontosabb észrevétel az, hogy a győztes elemet kivéve mindegyik elemnek veszíteni kell legalább egy mérkőzésen. Ennélfogva $n-1$ összehasonlítás szükséges a minimum meghatározásához, és a MINIMUM algoritmus optimális az elvégzett összehasonlítások számára nézve.

Minimum és maximum egyidejű meghatározása

Néhány alkalmazásban egy n elemű halmaz minimumát és maximumát is meg kell határozni. Például, egy grafikus program esetén szükség lehet arra, hogy átalakítsuk (x, y) értékek egy halmazát azért, hogy ráillesszük egy téglalap alakú képernyőre vagy más grafikus kimeneti eszközre. Ehhez először meg kell határozni mindegyik koordináta minimumát és maximumát.

Nem túl nehéz kigondolni egy olyan algoritmust, amely egy n elemű halmaz minimumát és maximumát is meghatározza, és aszimptotikusan optimális $\Theta(n)$ számú összehasonlítást használ. Egyszerűen keressük meg a minimumot és maximumot egymástól függetlenül, mindegyikhez $n-1$ összehasonlítást használva. Így összesen $2n-2$ összehasonlítást végzünk.

A minimum és maximum egyidejű meghatározásához valójában elég mindössze $3\lfloor n/2 \rfloor$ darab összehasonlítás. Ehhez őrizzük meg az addig megvizsgált elemek minimumát és maximumát. Ahelyett, hogy a bemenet mindegyik elemét összehasonlítanánk az aktuális minimum- és maximumértékekkel, ami két összehasonlítást jelentene elemenként, inkább elempárokat dolgozzunk fel. A bemeneti elempárokat először *egymással* hasonlítjuk össze, aztán a kisebbiket az aktuális minimum értékkel hasonlítjuk össze, a nagyobbikat pedig az aktuális maximum értékével, ami költségben három összehasonlítást jelent két elemre nézve.

Az aktuális minimum és maximum kezdőértékek meghatározása attól függ, hogy n páratlan vagy páros. Ha n páratlan, a minimális és maximális értéknek az első elem értékét vesszük, és páronként dolgozzuk fel a maradék elemeket. Ha n páros, az első két elemen egy összehasonlítást végzünk a minimum és a maximum kezdőértékének meghatározása céljából, majd mint a páratlan n esetén, párokban dolgozzuk fel a maradék elemeket.

Vizsgáljuk meg az összehasonlítások számát. Ha n páratlan, $3\lfloor n/2 \rfloor$ összehasonlítást végzünk. Ha n páros, 1 kezdeti összehasonlítás után $3(n-2)/2$ összehasonlítást végzünk, ami összesen $3n/2-2$. Így mindkét esetben az összehasonlítások száma legfeljebb $3\lfloor n/2 \rfloor$.

Gyakorlatok

9.1-1. Mutassuk meg, hogy n elem második legkisebb eleme legfeljebb $n+\lceil \lg n \rceil - 2$ összehasonlítással meghatározható . (*Útmutatás.* Határozzuk meg a legkisebb elemet is.)

9.1-2.* Mutassuk meg, hogy legrosszabb esetben $3\lceil n/2 \rceil - 2$ összehasonlítás szükséges n elem minimumának és maximumának egyidejű meghatározásához. (*Útmutatás.* Figyeljük meg, hány olyan szám van, amelyik lehetséges minimum vagy maximum, és vizsgáljuk meg, hogyan befolyásolja egy összehasonlítás ezek számát.)

9.2. Kiválasztás átlagosan lineáris időben

Az általános kiválasztási feladat sokkal bonyolultabbnak tűnik, mint az egyszerű minimumkereső feladat és mégis, meglepő módon mindkét feladat esetén az aszimptotikus futási idő $\Theta(n)$ feltételezve, hogy az elemek mind különbözők. Ebben az alfejezetben a kiválasztási feladat egy *oszd-meg-és-uralkodj* elvű algoritmusát mutatjuk be. A VÉLETLEN-KIVÁLASZT eljárás a 7. fejezetben bemutatott gyorsrendezés mintájára épül. Épp úgy, mint a gyorsrendezésben, az elv az, hogy a bemeneti tömböt rekurzív módon felosztjuk. Csakhogy amíg gyorsrendezés a felosztás mindkét felét rekurzív módon feldolgozza, addig a VÉLETLEN-KIVÁLASZT a felosztásnak csak az egyik felével dolgozik tovább. A különbség az elemzésben mutatkozik meg: amíg a gyorsrendezés várható futási ideje $\Theta(n \lg n)$, addig a VÉLETLEN-KIVÁLASZT várható ideje $\Theta(n)$.

A VÉLETLEN-KIVÁLASZT a 7.3. alfejezetben bemutatott VÉLETLEN-FELOSZTÁS eljárást használja. Ez, éppúgy mint a VÉLETLEN-GYORSRENDEZÉS, egy véletlenített algoritmus, hiszen viselkedését részben egy véletlenszám-generátor kimenete befolyásolja. A VÉLETLEN-KIVÁLASZT most következő kódja az $A[p\,..\,r]$ tömb i-edik legkisebb elemét adja vissza.

```
VÉLETLEN-KIVÁLASZT(A, p, r, i)

1  if p = r
2     then return A[p]
3  q ← VÉLETLEN-FELOSZTÁS(A, p, r)
4  k ← q − p + 1
5  if i = k                          ▷ az őrszem elem a válasz
6     then return A[q]
7  elseif i < k
8        then return VÉLETLEN-KIVÁLASZT(A, p, q − 1, i)
9  else return VÉLETLEN-KIVÁLASZT(A, q + 1, r, i − k)
```

Az algoritmus 3. sorában végrehajtott VÉLETLEN-FELOSZTÁS után az $A[p\,..\,r]$ tömb az $A[p\,..\,q-1]$ és $A[q+1\,..\,r]$ valószínűleg üres altömbökre bomlik szét úgy, hogy az $A[p\,..\,q-1]$ mindegyik eleme kisebb vagy egyenlő lesz, mint $A[q]$, ami kisebb vagy egyenlő lesz, mint a $A[q+1\,..\,r]$ összes eleme. Mint a GYORSRENDEZÉS esetén, $A[q]$-t itt is ***őrszem*** elemnek nevezzük. Az algoritmus 4. sora kiszámolja az $A[p\,..\,q]$ altömb elemeinek k számát,

ami a felosztás alsó része elemszáma és – az őrszem elem miatt – 1 összegeként adódik. Az 5. sor ezután ellenőrzi, hogy $A[q]$ az i-edik legkisebb elem-e. Ha az, a visszaadott érték $A[q]$ lesz. Ha az előbbi nem áll fenn, akkor az algoritmus meghatározza, hogy az $A[p\,..\,q-1]$ és $A[q+1\,..\,r]$ altömbök közül melyikben található az i-edik legkisebb elem. Ha $i < k$, akkor a keresett elem a felosztás alsó részében található és a 8. sorban rekurzívan választódik ki a megfelelő altömbből. Ha $i > k$, akkor viszont a keresett elem a felosztás felső részében található. Mivel már k darab értékről tudjuk, hogy kisebbek, mint az $A[p\,..\,r]$ tömb i-edik legkisebb eleme – nevezetesen az $A[p\,..\,q]$ tömb elemei ilyenek –, ezért a keresett elem az $A[q+1\,..\,r]$ tömb $(i-k)$-adik legkisebb eleme, amelyet a 9. sorban rekurzívan határozunk meg. A kód alapján úgy tűnik, mintha 0 hosszúságú altömbökre is lenne rekurzív hívás, de ez a helyzet nem állhat elő. A 9.2-1. feladatban épp ezt kell bebizonyítani.

A Véletlen-kiválaszt futási ideje legrosszabb esetben $\Theta(n^2)$, még a minimum kiválasztásának esetében is. Ez azért lehetséges, mert előfordulhat, hogy annyira szélsőségesen nincs szerencsénk, hogy a felosztás minduntalan a legnagyobb megmaradó értékek körül történik, és a felosztás $\Theta(n)$ időt vesz igénybe. Az algoritmus átlagos esetben jól teljesít, sőt, mivel véletlenített, nem létezik olyan sajátos bemenet, amelyik a legrosszabb viselkedésű esetet idézné elő.

Az n elemű $A[p\,..\,r]$ tömbön végrehajtott Véletlen-kiválaszt szükséges futási ideje egy $T(n)$ valószínűségi változó. Ennek $\mathrm{E}[T(n)]$ felső korlátját a következő módon kaphatjuk meg: Bármely elem esetén annak a valószínűsége, hogy a Véletlen-felosztás kimenetén őrszemként jelenik meg, ugyanaz. Így minden k esetén, melyre $1 \le k \le n$, annak valószínűsége, hogy a $A[p\,..\,q]$ k elemű (minden elem kisebb vagy egyenlő, mint az őrszem), $1/n$. Definiáljuk $k = 1, 2, \ldots n$ esetén az X_k indikátor valószínűségi változókat a következőképpen:

$$X_k = \mathrm{I}\{\text{az } A[p\,..\,q] \text{ altömbnek pontosan } k \text{ eleme van}\},$$

így, feltéve, hogy az elemek egyediek

$$\mathrm{E}[X_k] = \frac{1}{n}. \tag{9.1}$$

Amikor meghívjuk a Véletlen-kiválaszt algoritmust, és az $A[q]$-t őrszem elemnek választjuk, előre nem tudjuk, hogy azonnal leáll-e a helyes választással, rekurzív módon folytatódik az $A[p\,..\,q-1]$ altömbön vagy rekurzív módon folytatódik az $A[q+1\,..\,r]$ altömbön. Ez a döntés attól függ, hogy az i-edik legkisebb elem relatíve milyen közel esik $A[q]$-hoz. Feltételezve, hogy a $T(n)$ monoton növekvő, a rekurzív hívás idejének korlátját megkapjuk a legnagyobb lehetséges bemenet melletti rekurzív híváshoz szükséges idő segítségével. Más szavakkal, a felső korlát megkapásához feltételezzük, hogy az i-edik legkisebb elem mindig a több elemet tartalmazó részbe esik. A Véletlen-kiválaszt egy adott meghívásához az X_k indikátor valószínűségi változó értéke pontosan egy k értékre 1, és minden más k esetén 0. Ha $X_k = 1$, a két altömb – melyen lehet, hogy rekurzív hívást kell végrehajtani – $k-1$, illetve $n-k$ elemű. Így a következő rekurziót kapjuk:

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n). \end{aligned}$$

Várható értékeket véve a következőket kapjuk:

$$\begin{aligned}
\mathrm{E}\,[[T(n)]] &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot (T(\max(k-1,n-k)) + O(n)\right] \\
&= \sum_{k=1}^{n} \mathrm{E}\,[X_k \cdot (T(\max(k-1,n-k))] + O(n) \qquad \text{(a várható érték linearitása miatt)} \\
&= \sum_{k=1}^{n} \mathrm{E}\,[X_k] \cdot \mathrm{E}\,[(T(\max(k-1,n-k))] + O(n) \qquad \text{(a (C.23) egyenlőség miatt)} \\
&= \sum_{k=1}^{n} (1/n) \cdot \mathrm{E}\,[(T(\max(k-1,n-k))] + O(n) \qquad \text{(a 9.1 egyenlőség miatt).}
\end{aligned}$$

A (C.23) egyenlőség alkalmazásához azt használtuk fel, hogy X_k és $T(\max(k-1,n-k)$ független valószínűségi változók. A 9.2-2. feladatban ennek a feltételnek a helyességét kell belátni.

Tekintsük a $\max(k-1,n-k)$ kifejezést. Tudjuk, hogy

$$\max(k-1,n-k) = \begin{cases} k-1, & \text{ha } k > \lceil n/2 \rceil, \\ n-k, & \text{ha } k \le \lceil n/2 \rceil. \end{cases}$$

Ha n páros, $T(\lceil n/2 \rceil)$-től $T(n-1)$-ig minden kifejezés pontosan kétszer szerepel az összegzésben. Ha n páratlan, minden kifejezés kétszer szerepel és a $T(\lfloor n/2 \rfloor)$ pedig egyszer. Így azt kapjuk, hogy

$$\mathrm{E}\,[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 r\, floor}^{n-1} \mathrm{E}\,[T(k)] + O(n).$$

A rekurzivitást helyettesítéssel oldjuk fel. Tegyük fel, hogy létezik olyan c állandó, amelyre $T(n) \le cn$, és c teljesíti a rekurzív képlet kezdeti feltételeit. Tegyük fel, hogy $T(n) = O(1)$ valamilyen állandónál kisebb n-re. Ezt az állandót később határozzuk meg. Vegyünk továbbá egy olyan a állandót, melynél a fenti $O(n)$ kifejezés (ami az algoritmus futási idejének nem rekurzív összetevőjét írja le) felső korlátja an minden $n > 0$ esetén. Ezt az induktív feltevést felhasználva a következőket kapjuk:

$$\begin{aligned}
\mathrm{E}\,[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an \\
&\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an
\end{aligned}$$

$$\begin{aligned} &= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\ &= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\ &\leq \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right). \end{aligned}$$

Ahhoz, hogy a bizonyítást befejezzük, be kell mutatni, hogy elég nagy n esetén az utolsó kifejezés legfeljebb cn, vagy azt, ami ezzel ekvivalens, hogy $cn/4 - c/2 - an \geq 0$. Ha mindkét oldalhoz hozzáadunk $c/2$-t, és kiemeljük n-t, azt kapjuk, hogy $n(c/4 - a) \geq c/2$. Ha a c konstanst úgy választjuk, hogy $c/4 - a > 0$, azaz $c > 4a$, mindkét oldalt eloszthatjuk $c/4 - a$-val, ami a következő egyenlőtlenséget adja:

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a} .$$

Így, ha feltételezzük, hogy $T(n) = O(1)$ valamely $n < 2c/(c - 4a)$ esetén, azt kapjuk, hogy $\mathrm{E}[T(n)] = O(n)$. Tehát a rendezett minta bármely eleme, így a medián is, meghatározható átlagosan lineáris időben.

Gyakorlatok

9.2-1. Mutassuk meg, hogy a VÉLETLEN-KIVÁLASZT algoritmusban 0 hosszúságú tömb esetén sosincs rekurzív hívás.

9.2-2. Bizonyítsuk be, hogy az X_k indikátor véletlen változó és a $T(\max(k-1, n-k))$ függetlenek.

9.2-3. Írjuk meg a VÉLETLEN-KIVÁLASZT egy iteratív változatát.

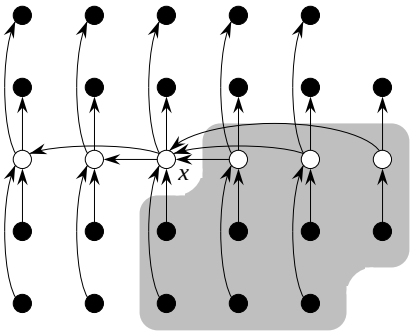
9.2-4. Tegyük fel, hogy a VÉLETLEN-KIVÁLASZT algoritmust használjuk az $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1\rangle$ tömb minimumának meghatározásához. Adjuk meg a felosztások egy olyan sorozatát, amelyik a VÉLETLEN-KIVÁLASZT legrosszabb esetének végrehajtását eredményezi.

9.3. Kiválasztás legrosszabb esetben lineáris időben

Most egy olyan algoritmust vizsgálunk meg, amelyiknek $O(n)$ a legrosszabb futási ideje. Éppúgy, mint a VÉLETLEN-KIVÁLASZT, a KIVÁLASZT algoritmus a keresett elemet szintén a bemeneti tömb rekurzív felosztásával határozza meg. Mindamellett az algoritmus kulcsa az, hogy egy jó szétválasztást *biztosít* a tömb felosztása során. A KIVÁLASZT algoritmus a gyorsrendezés FELOSZT nevű determinisztikus felosztó algoritmusát használja (lásd 7.1. alfejezet), azzal a változtatással, hogy azt az elemet, amely körül a felosztás történik, meg kell adni bemenő paraméterként.

A KIVÁLASZT algoritmus a következő lépések végrehajtásával határozza meg $n > 1$ elemből álló bemenő tömb i-edik legkisebb elemét. (Ha $n = 1$, akkor a KIVÁLASZT algoritmus i-edik legkisebb elemként a bemenő elemet adja vissza.)

1. Osszuk a bemeneti tömb n elemét $\lfloor n/5 \rfloor$ darab 5 elemből álló csoportra, és a maradék n mod 5 darab elemből alkossunk legfeljebb egy újabb csoportot.



9.1. ábra. A KIVÁLASZT algoritmus vizsgálata. Az n darab elemet kis körök jelölik, minden csoportnak egy oszlop felel meg. A csoportok mediánjai a fehér körök, és a mediánok mediánját az x címkével láttuk el. (Páros elemszám esetén az alsó mediánt használjuk.) A nyilak a nagyobb elemekről a kisebbek felé mutatnak. Látható, hogy mindegyik, x jobb oldalán levő 5 elemű csoportból 3 elem nagyobb, mint x, és minden, az x bal oldalán levő 5 elemű csoportból 3 elem kisebb, mint x. Az x elemnél nagyobb elemeket az árnyékolt háttér mutatja.

2. Az összes $\lceil n/5 \rceil$ darab csoportnak keressük meg a mediánját úgy, hogy a csoport elemeit rendezzük beszúrásos rendezéssel (mindegyik csoport legfeljebb 5 elemet tartalmaz), és aztán válasszuk ki a mediánt a csoport elemeinek rendezett listájából.
3. A KIVÁLASZT algoritmust rekurzív használatával határozzuk meg a 2. lépésben kapott $\lceil n/5 \rceil$ darab medián x mediánját. (Ha páros számú medián van, akkor x a megállapodásunk szerint az alsó medián.)
4. A módosított FELOSZT algoritmus segítségével osszuk fel a bemeneti tömböt a mediánok mediánja, azaz az x körül. Legyen k a felosztás alsó részébe került elemek számánál eggyel több úgy, hogy x a k-adik legkisebb elem. Ekkor a felosztás felső részében $n - k$ elem van.
5. Ha $i = k$, akkor a visszaadott érték legyen x. Máskülönben a KIVÁLASZT algoritmus rekurzív használatával keressük meg az i-edik legkisebb elemet az alsó részben, ha $i < k$, vagy az $(i - k)$-adik legkisebb elemet a felső részben, ha $i > k$.

A KIVÁLASZT algoritmus futási idejének vizsgálatához először meghatározunk egy alsó korlátot az x felosztó elemnél nagyobb elemek számára. A 9.1. ábra ezeket a számításokat szemlélteti. A 2. lépésben megkapott mediánok legalább fele nagyobb vagy egyenlő, mint x, a mediánok mediánja.[1] Így az $\lceil n/5 \rceil$ darab csoport legalább fele tartalmaz 3 olyan elemet, amelyik nagyobb, mint x, azt az egy csoportot kivéve, amelyik 5-nél kevesebb elemet tartalmaz, amikor n nem osztható 5-tel, és kivéve az x elemet tartalmazó csoportot. E két csoportot leszámítva azt kapjuk, hogy az x-nél nagyobb elemek száma legalább

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \;\geq\; \frac{3n}{10} - 6.$$

Hasonló módon, a x-nél kisebb elemek száma legalább $3n/10 - 6$. Tehát a legrosszabb esetben a KIVÁLASZT algoritmust legfeljebb $7n/10 + 6$ darab elemre hívjuk meg rekurzívan az 5. lépésben.

[1] Mivel feltételeztük, hogy a számok egyediek, az x kivételével minden medián nagyobb vagy kisebb, mint x.

Most már megadhatunk a KIVÁLASZT algoritmus $T(n)$ legrosszabb futási idejére egy rekurzív képletet. Az 1., 2. és 4. lépések időigénye $O(n)$. (A 2. lépésben $O(1)$ méretű halmazokra $O(n)$ alkalommal hívjuk meg a beszúrásos rendezést.) A 3. lépés időigénye $T(\lceil n/5 \rceil)$, és az 5. lépés időigénye legfeljebb $T(7n/10+6)$ feltéve, hogy T monoton növekvő. Bár először nem tűnik motiválónak, feltesszük, hogy a 140 vagy kevesebb elemből álló bemenetek $O(1)$ időt vesznek igénybe. A mágikus 140-es állandó eredetét rövidesen megmutatjuk. Így tehát a következő rekurziót kapjuk:

$$T(n) \leq \begin{cases} O(1), & \text{ha } n < 140, \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n), & \text{ha } n \geq 140. \end{cases}$$

A futási idő lineáris voltát helyettesítéssel igazoljuk. Pontosabban megmutatjuk, hogy $T(n) \leq cn$ elég nagy c állandó és minden $n > 0$ esetén. Azzal kezdjük, hogy feltételezzük, létezik elég nagy c állandó, hogy $T(n) \leq cn$ minden $n < 140$ esetén. Feltételezésünk akkor áll, ha c elég nagy. Választunk egy a állandót is úgy, hogy a fenti $O(n)$ kifejezés (ami az algoritmus futási idejének nem rekurzív összetevőjét írja le) által leírt függvény felső korlátja an minden $n > 0$ esetén. Ezt az induktív feltevést a rekurzív képlet jobb oldalába behelyettesítve a következőket kapjuk:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10+6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an), \end{aligned}$$

ami legfeljebb cn, ha

$$-cn/10 + 7c + an \leq 0. \tag{9.2}$$

A (9.2) egyenlőtlenség azonos a $c \geq 10a(n/(n-70))$ egyenlőtlenséggel, amikor $n > 70$. Mivel feltételeztük, hogy $n \geq 140$, azt kapjuk, hogy $n/(n-70) \leq 2$. Így ha c-t $c \geq 20a$ módon választjuk meg, az kielégíti a (9.2) egyenlőtlenséget. (Megjegyezzük, hogy a 140-es állandó különösebben nem speciális. Bármely, 70-nél nagyobb egészre kicserélhetjük, és c-t ennek megfelelően választhatjuk.) Tehát a KIVÁLASZT algoritmus legrosszabb futási ideje lineáris.

Éppúgy, mint az összehasonlító rendezésnél (lásd 8.1. alfejezet), a KIVÁLASZT és a VÉLETLEN-KIVÁLASZT az elemek egymáshoz viszonyított sorrendjéről csak az elemek összehasonlításával szerez információt. Emlékezzünk a 8. fejezetben olvasottakra, hogy az összehasonlító rendezés még átlagos esetben is (lásd 8-1. feladat) $\Omega(n \lg n)$ időt igényel. A 8. fejezetben bemutatott lineáris idejű rendező algoritmusok feltételezéssel éltek a bemenettel kapcsolatban. Ezzel ellentétben, az ebben a fejezetben bemutatott lineáris idejű kiválasztó algoritmusok nem éltek feltételezéssel a bemenettel kapcsolatban. Ezekre nem érvényes az $\Omega(n \lg n)$ alsó korlát, mert rendezés nélkül oldják meg a kiválasztási problémát.

Tehát a futási idő lineáris, mert ezek az algoritmusok nem rendeznek. A lineáris idejűség itt nem a bemenettel kapcsolatos feltételezésekből ered, mint a 8. fejezet rendező algoritmusai esetén. Az összehasonlító rendezés időigénye még átlagos esetben is $\Omega(n \lg n)$ (lásd 8-1. feladat), így a fejezet elején bemutatott módszer, amelyik először rendez és utána rámutat a megfelelő elemre, aszimptotikusan nem hatékony.

Gyakorlatok

9.3-1. A KIVÁLASZT algoritmusban a bemenő elemeket 5 elemből álló csoportokba osztottuk. Vajon lineáris időben működne az algoritmus akkor is, ha 7 elemű csoportokat képeznénk? Bizonyítsuk be, hogy a KIVÁLASZT algoritmus nem működne lineáris időben, ha 3 elemű csoportokat használnánk.

9.3-2. A KIVÁLASZT algoritmust elemezve mutassuk meg, hogy ha $n \geq 140$, akkor legalább $\lceil n/4 \rceil$ elem nagyobb, mint x, a mediánok mediánja, és legalább $\lceil n/4 \rceil$ elem kisebb, mint x.

9.3-3. Mutassuk meg, hogyan módosítható a gyorsrendezés úgy, hogy a legrosszabb futási ideje $O(n \lg n)$ legyen feltéve, hogy minden elem különböző.

9.3-4.⋆ Tegyük fel, hogy egy algoritmus csak összehasonlításokat használ n elem közül az i-edik legkisebb elem meghatározásához. Mutassuk meg, hogy ezzel az algoritmussal az $i - 1$ darab kisebb elemet és az $n - i$ darab nagyobb elemet is meghatározhatjuk anélkül, hogy további összehasonlításokat használnánk.

9.3-5. Tegyük fel, hogy adott egy „fekete doboz” típusú, lineáris legrosszabb futási idejű szubrutin. Adjunk meg olyan egyszerű, lineáris idejű algoritmust, amelyik a rendezett minta tetszőleges elemének a kiválasztására alkalmas.

9.3-6. Egy n elemű halmaz k-adik ***kvantilisei*** a rendezett mintából azon $k-1$ darab elemek, amelyek a rendezett halmazt k egyenlő részre osztják (az 1-et is beleértve). Adjunk meg olyan $O(n \lg k)$ időigényű algoritmust, amelyik felsorolja egy halmaz k-adik kvantiliseit.

9.3-7. Adjunk meg olyan $O(n)$ időigényű algoritmust, amelyik adott n elemű, különböző számokból álló S halmaz és adott pozitív egész $k \leq n$ szám esetén meghatározza azt a k darab számot az S halmazból, amelyek az S halmaz mediánjához legközelebb vannak.

9.3-8. Legyenek az $X[1 \,..\, n]$ és $Y[1 \,..\, n]$ n számot tartalmazó, már rendezett tömbök. Adjunk olyan $O(\lg n)$ idejű algoritmust, amelyik az X és Y tömbök $2n$ darab elemének a mediánját határozza meg.

9.3-9. Olay professzor egy olajvállalat szaktanácsadója. A vállalat olyan fővezetéket tervez, amelyik egy n kutat működtető olajmezőt szel át keletről nyugat felé. Mindegyik kút közvetlen csővel (északról vagy délről) a legrövidebb úton csatlakozik a fővezetékhez, amint azt a 9.2. ábrán láthatjuk. A kutak x és y koordinátáit ismerve, hogyan tudja a professzor a fővezeték optimális helyét meghatározni (optimális abból a szempontból, hogy a bekötő csövek hosszának az összege minimális)? Mutassuk meg, hogy az optimális hely kiválasztása elvégezhető lineáris időben.

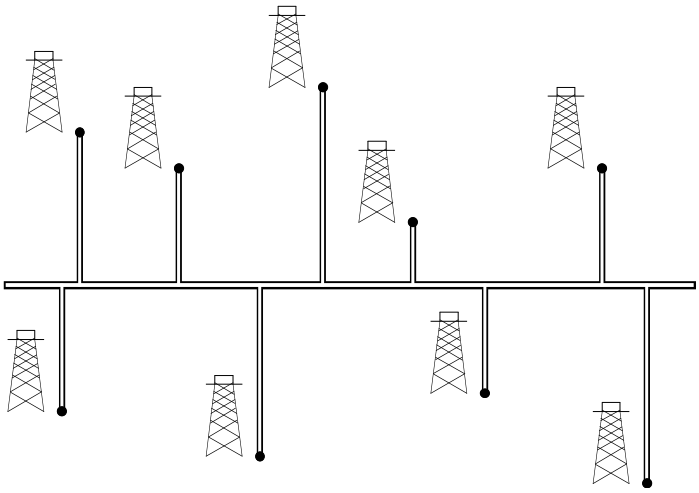
Feladatok

9-1. Az i darab rendezett legnagyobb szám

Adott n darab szám és egy összehasonlító algoritmussal szeretnénk meghatározni az i darab rendezett legnagyobb számot. Az alábbi módszerek implementálására adjunk meg olyan algoritmusokat, amelyeknek a legrosszabb futási ideje aszimptotikusan a legjobb, majd vizsgáljuk meg az algoritmusok futási idejét az n és az i függvényében.

a. Rendezzük a számokat, majd listázzuk ki az i darab legnagyobbat.

b. Készítsünk egy elsőbbségi sort a számokból, majd hívjuk meg a MAXIMUMOT-KIVÁG algoritmust i alkalommal.



9.2. ábra. Olay professzornak a kelet–nyugat irányú olajvezeték helyét kell meghatározni oly módon, hogy az észak–dél irányú bekötő csövek hosszának az összege minimális legyen.

c. Használjunk egy a rendezett mintákra adott algoritmust az *i*-edik legnagyobb szám meghatározására, a kapott szám alapján osszuk fel a számokat, és rendezzük az *i* darab legnagyobb számot.

9-2. Súlyozott medián

n darab $x_1, x_2, \ldots, x_n$ különböző szám esetén, amelyeket $w_1, w_2, \ldots, w_n$ pozitív súlyokkal láttunk el úgy, hogy $\sum_{i=1}^{n} w_i = 1$, a ***súlyozott medián*** az az x_k elem, amelyikre teljesül, hogy

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

és

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2} .$$

a. Bizonyítsuk be, hogy az $x_1, x_2, \ldots, x_n$ számok mediánja egyenlő az x_i számok azon súlyozott mediánjával, ahol a súlyok rendre $w_i = 1/n$ $i = 1, 2, \ldots, n$-re.

b. Mutassuk meg, hogyan számolható ki rendezéssel n elem súlyozott mediánja legrosszabb esetben $O(n \lg n)$ időben.

c. Mutassuk meg, hogyan számolható ki a súlyozott medián legrosszabb esetben $\Theta(n)$ id őben egy lineáris mediánkereső algoritmus segítségével, amilyen például a 9.3. alfejezet KIVÁLASZT algoritmusa.

A ***postahivatal-feladat*** megfogalmazása a következő. Adott n darab, $w_1, w_2, \ldots, w_n$ súlyokkal ellátott $p_1, p_2, \ldots, p_n$ pont. Szeretnénk meghatározni azt a p pontot (ez nem feltétlenül a bemeneti pontok egyike), amelyikre a $\sum_{i=1}^{n} w_i d(p, p_i)$ összeg minimális, ahol a $d(a, b)$ az a és b pontok közötti távolság.

d. Bizonyítsuk be, hogy az 1 dimenziós postahivatal-problémának a súlyozott medián a legjobb megoldása, ahol a pontok egyszerűen valós számok és két, a és b pont közötti távolság $d(a, b) = |a - b|$.

e. Keressük meg a 2 dimenziós postahivatal-feladat legjobb megoldását. Ebben az esetben a pontokat az (x, y) koordinátapárokkal adjuk meg, és az $a = (x_1, y_1)$ és $b = (x_2, y_2)$ pontok közötti távolság az úgynevezett ***Manhattan-távolság:*** $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3. Kis rendezett minták

n darab szám rendezett mintája i-edik elemének megtalálásához a KIVÁLASZT algoritmus által legrosszabb esetben használt összehasonlítások $T(n)$ számáról megmutattuk, hogy $T(n) = \Theta(n)$, de a Θ jelölés mögött rejlő állandó igen nagy. Ha az i az n-hez képest viszonylag kicsi, akkor megvalósíthatunk egy másfajta eljárást, amelyik a KIVÁLASZT algoritmust szubrutinként meghívja, de legrosszabb esetben is kevesebb összehasonlítást végez.

a. Adjunk meg olyan algoritmust, amelyik az i-edik legnagyobb elem meghatározásához $U_i(n)$ darab összehasonlítást végez, ahol

$$U_i(n) = \begin{cases} T(n), & \text{ha } i \geq n/2, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{egyébként.} \end{cases}$$

(*Útmutatás.* Kezdjük $\lfloor n/2 \rfloor$ különböző páronkénti összehasonlítással, majd rekurzívan azon a halmazon haladjunk tovább, amelyik a párok kisebbik elemeit tartalmazza.)

b. Mutassuk meg, hogy ha $i < n/2$, akkor $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Mutassuk meg, hogy ha i $n/2$-nél kisebb állandó, akkor $U_i(n) = n + O(\lg n)$.

d. Mutassuk meg, hogy ha $i = n/k$, $k \geq 2$ esetén, akkor $U_i(n) = n + O(T(2n/k) \lg k)$.

Megjegyzések a fejezethez

A legrosszabb esetben végzett lineáris idejű mediánkiválasztás Blum, Floyd, Pratt, Rivest és Tarjan [43] ötlete. Az átlagos eset gyors változatát Hoare [146] munkájának tulajdonítják. Floyd és Rivest [92] az átlagos esetre kifejlesztett egy javított változatot, amelyik az elemek egy kis mintájából rekurzívan kiválasztott elem körül végzi a felosztást.

Még mindig nem ismert, hogy hány összehasonlítás kell a medián meghatározásához. Az összehasonlítások alsó korlátjára Bent és John [38] a $2n$-t találta. Felső korlátra Schönhage, Paterson és Pippenger [265] a $3n$-t adta meg. Dor és Zwick [79] mindkét korlátot javította. Az ő felső korlátjuk kicsit kevesebb, mint $2,95n$ és az alsó korlátjuk kicsit több, mint $2n$. Paterson [239] más kapcsolódó munkákkal együtt bemutatja ezeket az eredményeket.

III. ADATSZERKEZETEK

Bevezetés

A halmazok szerepe ugyanúgy alapvető a számítástudományban, mint a matematikában. Amíg azonban egy matematikai halmazt időben változatlannak tekintünk, addig egy olyan halmaz, amelyen egy algoritmus végez műveleteket, bővülhet, zsugorodhat, vagy a feldolgozás során másképpen is módosulhat. Az ilyen halmazokat ***dinamikusnak*** nevezzük. A következő öt fejezetben néhány olyan alapvető módszert ismertetünk, amelyek segítségével a véges dinamikus halmazokat ábrázolhatjuk és velük műveleteket végezhetünk a számítógépen.

Az algoritmusokban számos különböző típusú, halmazokon értelmezett műveletre lehet szükség. Sok algoritmusban például elegendő annyi, hogy a halmazba beszúrjunk vagy abból töröljünk egy elemet, valamint eldöntsük egy elemnek a halmazhoz való tartozását. Azokat a dinamikus halmazokat, amelyekre ezeket a műveleteket értelmezzük, ***szótárnak*** nevezzük. Más algoritmusokban bonyolultabb műveletekre van szükség. Például a 6. fejezetben a kupac adatszerkezettel megvalósított minimumválasztó elsőbbségi sorok rendelkeznek olyan művelettel, amelyik beszúr egy elemet a halmazba és olyannal is, amelyik kiveszi onnan a legkisebbet. Egy dinamikus halmaz legalkalmasabb megvalósítását mindig a szükséges műveletek figyelembevételével választhatjuk meg.

A dinamikus halmaz elemei

A dinamikus halmazok egyik gyakori megvalósításában mutatók teszik lehetővé a halmaz elemeit ábrázoló objektumok elérését, mezőinek vizsgálatát, illetve módosítását. (A 10.3. alfejezetben tárgyaljuk az objektumok és mutatók megvalósítását olyan programozási környezetekben, amelyek nem tartalmazzák azokat mint elemi adattípusokat.) Bizonyos dinamikus halmazok objektumaiban szerepel egy azonosító ***kulcsmező.*** Ha a kulcsértékek mind különbözők, akkor a dinamikus halmaz tekinthető a kulcsértékek halmazának is. Az objektumokban szerepelhetnek ***kísérő adatok*** is, amelyeket az objektumok további mezői tartalmazzák, de ezeket a halmaz adott megvalósításában nem használjuk. Azok a mezők, amelyekre a halmazműveletek támaszkodnak, adatokat vagy pointereket tartalmazhatnak, amelyek a halmaz további objektumaira mutatnak.

Bizonyos esetekben feltesszük, hogy a dinamikus halmazban szereplő kulcsok egy olyan teljesen rendezett halmazból valók, mint például a valós számok halmaza vagy a szavak halmaza a szokásos ábécé szerinti rendezés mellett. (Egy teljesen rendezett halmaz rendelkezik a 3.1.7. pontban definiált trichotom tulajdonsággal.) Teljes rendezés esetén defini-

álhatjuk például egy halmaz legkisebb elemét, vagy beszélhetünk arról az elemről, amelyik a halmaz adott eleme után nagyság szerint közvetlenül következik.

Dinamikus halmazokon értelmezett műveletek

A dinamikus halmazokon értelmezett műveleteket két csoportba sorolhatjuk: a ***lekérdező műveletek*** csupán információt szolgáltatnak a halmazról, a ***módosító műveletek*** azonban megváltoztatják a halmazt. A következőkben felsoroljuk a leggyakoribb műveleteket. Az egyes alkalmazásokban általában ezek közül nincs mindegyikre szükség.

KERES(H, k): Lekérdező művelet, amely egy H halmazban k kulcsú elemet keres, és egy olyan elem x mutatóját adja vissza, amelyre $kulcs[x] = k$, illetve NIL-t ad vissza, ha nincs ilyen elem a H halmazban.

BESZÚR(H, x): Módosító művelet, amely bővíti a H halmazt az x által mutatott elemmel. Általában feltesszük, hogy előzőleg az x elem mindazon mezői értéket kaptak, amelyekre az adott implementációban szükség van.

TÖRÖL(H, x): Módosító művelet, amely eltávolítja a H halmazból az x által mutatott elemet. (Jegyezzük meg, hogy x egy elemre mutató pointer, tehát nem kulcs szerinti törlésről van szó.)

MINIMUM(H): Lekérdező művelet, amely a teljesen rendezett H halmazban szereplő legkisebb kulcsértékű elem mutatóját adja vissza.

MAXIMUM(H): Lekérdező művelet, amely a teljesen rendezett H halmazban szereplő legnagyobb kulcsértékű elem mutatóját adja vissza.

KÖVETKEZŐ(H, x): A lekérdező művelet annak a H halmazbeli elemnek a mutatóját adja vissza, amelynek kulcsértéke közvetlenül az x elem kulcsértéke után következik a teljes rendezés szerint, illetve NIL-t ad vissza, ha x a legnagyobb kulcsú elem H-ban.

ELŐZŐ(H, x): A lekérdező művelet annak a H halmazbeli elemnek a mutatóját adja vissza, amelynek kulcsértéke közvetlenül megelőzi az x elem kulcsértékét a teljes rendezés szerint, illetve NIL-t ad vissza, ha x a legkisebb kulcsú elem.

A KÖVETKEZŐ és ELŐZŐ lekérdező műveleteket gyakran kiterjesztik olyan halmazokra is, amelyekben több azonos kulcsú elem is előfordulhat. Kézenfekvő feltételezés, hogy ha egy n elemű halmazra végrehajtjuk előbb a MINIMUM műveletet, majd ezt követően $(n - 1)$-szer meghívjuk a KÖVETKEZŐ eljárást, akkor a halmaz elemeinek a felsorolását kapjuk a kulcsértékek növekvő sorrendjében.

A halmazműveletek végrehajtási idejét szokásos módon a halmaz elemszámának a függvényében adjuk meg. Például a 13. fejezetben ismertetésre kerülő adatszerkezeten a fenti műveletek mindegyike végrehajtható, és azok mind $O(\lg n)$ időben futnak le egy n elemű halmaz esetén.

A III. rész tartalma

A 10–14. fejezetekben olyan adatszerkezeteket ismertetünk, amelyek alkalmasak dinamikus halmazok megvalósítására; a legtöbbet közülük fel is használjuk különféle problémák megoldására adott hatékony algoritmusokban. Egy másik fontos adatszerkezet, a kupac, már a 6. fejezetben bevezetésre került.

A 10. fejezetben ismertetjük a legfontosabb tudnivalókat az olyan egyszerű adatszerkezetek használatáról, mint amilyen a verem, a sor, a láncolt lista és a gyökeres fa. Azt is megadjuk, hogy az objektumokat és a mutatókat hogyan lehet megvalósítani olyan programozási környezetekben, amelyek nem támogatják ezt a megfelelő elemi típusokkal. Az itt tárgyalt anyag jó részét már ismerhetik azok az olvasók, akik túl vannak egy bevezető szintű programozói képzésen.

A 11. fejezetben bevezetjük a hasítótáblákat, amelyekkel megvalósíthatók a BESZÚR, TÖRÖL és KERES szótárműveletek. A KERES művelet végrehajtása a legrosszabb esetben $\Theta(n)$ időt igényel, de a hasító táblák műveleteinek várható futási ideje $O(1)$. Ennek az adatkezelési módszernek az elemzése valószínűségi alapon történik, azonban a fejezet legnagyobb részében nincs szükség ilyen irányú ismeretekre.

A bináris keresőfák, amelyeket a 12. fejezetben tárgyalunk, a dinamikus halmazok minden felsorolt műveletét támogatják. Egy n elemű bináris fára minden művelet végrehajtása a legrosszabb esetben $\Theta(n)$ időt vesz igénybe, de egy véletlenszerűen felépített bináris keresőfára minden művelet várható végrehajtási ideje $O(\lg n)$. A bináris keresőfák sok más adatszerkezetnek is alapjául szolgálnak.

A bináris keresőfák egy változatát, a piros-fekete fákat, a 13. fejezetben vezetjük be. A közönséges bináris keresőfákkal szemben a piros-fekete fákra a műveletek a legrosszabb esetben is $O(\lg n)$ időt vesznek igénybe. A piros-fekete fa egy kiegyensúlyozott keresőfa; a 18. fejezet ismerteti a B-fákat, amelyek a kiegyensúlyozott bináris keresőfák egy másik változatát képviselik. Noha a piros-fekete fák működési mechanizmusa némiképpen bonyolult, az eljárások lényegét a részletek tanulmányozása nélkül is megérthetjük a fejezetből. A teljes megértéshez azonban javasoljuk az algoritmusok kódjának végigkövetését.

A 14. fejezetben azt mutatjuk meg, hogyan lehet a piros-fekete fákat további műveletek megvalósítására kiterjeszteni. Először egy olyan kiterjesztést adunk meg, amelyben egy kulcshalmaz rendezett mintáit dinamikusan tudjuk karbantartani. Ezután egy másfajta kiterjesztéssel a valós számok intervallumainak a kezelését tesszük lehetővé.

10. Elemi adatszerkezetek

Ebben a fejezetben azt vizsgáljuk, hogyan lehet a dinamikus halmazokat olyan egyszerű adatszerkezetekkel ábrázolni, amelyek mutatókat tartalmaznak. Mutatók felhasználásával sok összetett adatszerkezet megvalósítható, itt azonban csak a legegyszerűbbeket tárgyaljuk; ezek a vermek, a sorok, a láncolt listák és a gyökeres fák. Ismertetünk egy olyan módszert is, amellyel az objektumokat és a mutatókat egyaránt tömbök segítségével valósíthatjuk meg.

10.1. Vermek és sorok

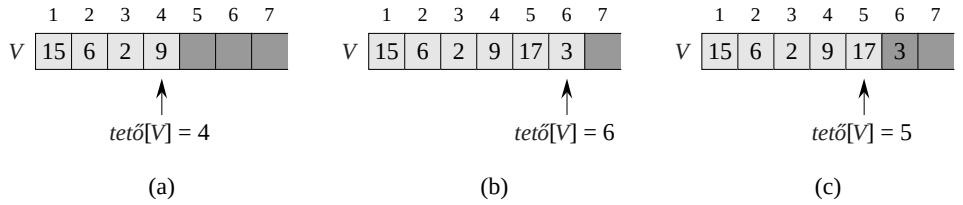
A vermek és sorok olyan dinamikus halmazok, amelyekben el őre meghatározott az az elem, amelyet a Töröl művelettel eltávolíthatunk. A ***veremből*** a legutoljára beszúrt elemet törölhetjük, így a verem ***utolsóként-be, elsőként-ki,*** vagy ***LIFO*** (last-in, first-out) eljárást valósít meg. A ***sorból*** pedig mindig a legrégebben beérkezett elemet lehet törölni, így a sor ***elsőként-be, elsőként-ki,*** vagy ***FIFO*** (first-in, first-out) stratégiát valósít meg. Több hatékony módszer is ismeretes a verem és a sor számítógépen történ ő megvalósítására. Ebben a részben azt mutatjuk be, hogyan lehet egy egyszerű tömb segítségével megvalósítani a két adatszerkezetet.

Vermek

A vermeken értelmezett Beszúr művelet neve Verembe, az argumentum nélküli Töröl művelet neve pedig Veremből. A verem adatszerkezet a LIFO-tulajdonság következtében hasonlóan működik, mint egy földbe ásott verem, amelynek csak a tetején lehet a terményeket betenni és kivenni. A veremb ől az elemeket éppen fordított sorrendben vesszük ki, mint ahogy betettük.

Egy legfeljebb n elemet tartalmazó verem megvalósítható egy $V[1\,..\,n]$ tömbbel, amint az a 10.1. ábrán látható. A tömb *tető*$[V]$ attribútuma a verembe legutoljára betett elemnek az indexe. A verem az $V[1\,..$ *tető*$[V]]$ elemekb ől áll, amelyek közül $V[1]$ a verem legalsó, $V[$*tető*$[V]]$ pedig a legfels ő eleme.

Ha *tető*$[V] = 0$, akkor a verem nem tartalmaz elemet és ***üresnek*** nevezzük. Az Üresverem lekérdező művelettel megállapítható, hogy egy verem üres-e. Ha egy üres veremb ől próbálunk elemet kivenni, akkor azt mondjuk, hogy a verem ***alulcsordul,*** amit általában



10.1. ábra. A V verem tömbös megvalósítása. A verem elemeit a világos mezők tartalmazzák. **(a)** A V verem 4 elemet tartalmaz. A tető elem a 9. **(b)** A V verem a VEREMBE(V,17) és a VEREMBE(V, 3) műveletek végrehajtása után. **(c)** A V verem a VEREMBŐL(V) művelet meghívása után a 3-at adja vissza, amely az utoljára betett elem. Annak ellenére, hogy a 3 a tömbben marad, a veremnek már nem eleme; az új tetőelem a 17.

hibának tekintünk. Ha *tető*[V] értéke meghaladja n-et, akkor a verem ***túlcsordul.*** (A következőkben a műveletek pszeudokódjában a túlcsordulást figyelmen kívül hagyjuk.)

A verem műveletei néhány sorban leírhatók.

ÜRES-VEREM(V)

```
1  if tető[V] = 0
2     then return IGAZ
3     else return HAMIS
```

VEREMBE(V, x)

```
1  tető[V] ←tető[V] + 1
2  V[tető[V]] ← x
```

VEREMBŐL(V)

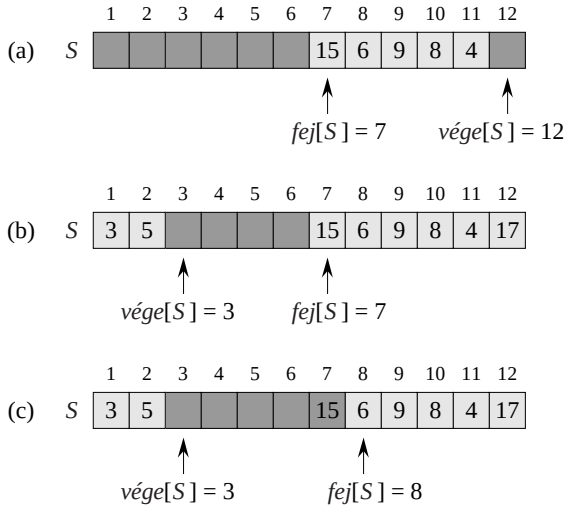
```
1  if ÜRES-VEREM(V)
2     then error „alulcsordulás”
3     else tető[V] ← tető[V] − 1
4          return V[tető[V] + 1]
```

A 10.1. ábra a VEREMBE és VEREMBŐL műveletek módosító hatását szemlélteti. Mindhárom művelet $O(1)$ idejű.

Sorok

A sorokon értelmezett BESZÚR művelet neve SORBA, a TÖRÖL művelet neve pedig SORBÓL, amelynek – a VEREMBŐL művelethez hasonlóan – nincs argumentuma. A sor a FIFO tulajdonság következtében hasonlóan működik, mint egy emberekb ől álló sor egy hivatali ablak előtt. A sor első elemét ***fejnek,*** az utolsó elemét ***végének*** nevezzük. Amikor egy elemet behelyezünk a sorba, akkor az a sor végére kerül, mint ahogyan egy újonnan érkez ő ügyfél a sor végére áll be. A sorból mindig az első elemet vesszük ki, mint ahogyan a legrégebben várakozó ügyfél az, aki soron következik. (Szerencsére itt kizárható az az eset, hogy egy elemet nem a sor végére, hanem valahová a belsejébe kell beszúrni.)

A 10.2. ábrán látható módon egy legfeljebb $n - 1$ elemű sort megvalósíthatunk egy $S[1..n]$ tömb felhasználásával. A sornak az egyik attribútuma *fej*[S], amely a sor első



10.2. ábra. Egy sor megvalósítása az $S[1..12]$ tömbbel. A sor elemeit a világos mezők tartalmazzák. **(a)** A sor öt eleme az $S[7..11]$ pozíciókban helyezkedik el. **(b)** A sor a SORBA$(S, 17)$, a SORBA$(S, 3)$ és a SORBA$(S, 5)$ műveletek végrehajtása után. **(c)** A SORBÓL(S) meghívása a 15-öt adja vissza. A 15 a tömbben marad ugyan, de a sornak nem eleme, az új fej a 6-os kulcsértékű elem.

elemét indexeli. A *vége*$[S]$ attribútum annak a helynek az indexe, amelyre a legközelebb beérkező elemet el fogjuk helyezni. A sorban az elemek rendre a *fej*$[S]$, *fej*$[S] + 1, \ldots,$ *vége*$[S] - 1$ helyeket foglalják el a ciklikus elrendezés szabályai szerint: a tömb végére érve az n-edik hely után az első hely következik. Ha *fej*$[S]$ = *vége*$[S]$, akkor a sor üres. Kezdetben *fej*$[S]$ = *vége*$[S] = 1$. Ha egy üres sorból próbálunk elemet kivenni, akkor az alulcsordulást okoz. Ha *fej*$[S]$ = *vége*$[S] + 1$, akkor a sor tele van, és ha ekkor próbálunk elemet beszúrni, az túlcsorduláshoz vezet.

A SORBA és SORBÓL műveletekben nem szerepel az alul- és túlcsordulási hiba vizsgálata. (A 10.1-4. gyakorlatban kitűzzük a két hibaellenőrzés beépítését az eljárásokba.)

```
SORBA(S, x)

1  S[vége[S]] ← x
2  if vége[S] = hossz[S]
3     then vége[S] ← 1
4     else vége[S] ← vége[S] + 1
```

```
SORBÓL(S)

1  x ← S[fej[S]]
2  if fej[S] = hossz[S]
3     then fej[S] ← 1
4     else fej[S] ← fej[S] + 1
5  return x
```

A SORBÓL és SORBA műveletek hatását a 10.2. ábra szemlélteti. Mindkét művelet $O(1)$ idejű.

Gyakorlatok

10.1-1. A 10.1. ábrán bemutatott modell felhasználásával szemléltessük a VEREMBE(V,4), VEREMBE(V,1), VEREMBE(V,3), VEREMBŐL(V), VEREMBE(V,8) és VEREMBŐL(V) műveletek mindegyikének a végrehajtását, ha a vermet a $V[1\,..\,6]$ tömbben tároljuk, és a verem kezdetben üres.

10.1-2. Hogyan valósítható meg két verem egyetlen $A[1\,..\,n]$ tömbben úgy, hogy egyik verem sem csordul túl addig, amíg együttes elemszámuk nem haladja meg az n-et? A VEREMBE és VEREMBŐL műveletek végrehajtási ideje maradjon $O(1)$.

10.1-3. A 10.2. ábrán bemutatott modell felhasználásával szemléltessük a SORBA(S, 4), SORBA(S, 1), SORBA(S, 3), SORBÓL(S), SORBA(S, 8) és SORBÓL(S) műveletek mindegyikének a végrehajtását, ha a sort a $S[1\,..\,6]$ tömbben tároljuk, és a sor kezdetben üres.

10.1-4. Írjuk meg a SORBA és a SORBÓL műveleteket úgy, hogy szerepeljen bennük az alulcsordulás és túlcsordulás vizsgálata.

10.1-5. Amint láttuk, a veremnek csak az egyik végén lehet elemet beszúrni és törölni, a sornak pedig az egyik végén lehet elemet beszúrni és a másik végén lehet törölni. Ezzel szemben a ***kétvégű sor*** mindkét végén lehet elemet beszúrni is, és törölni is. Írjunk négy olyan $O(1)$ idejű eljárást, amelyek megvalósítják a beszúrás és törlés műveleteit egy tömbbel ábrázolt kétvégű sor mindkét végén.

10.1-6. Adjuk meg a sor megvalósítását két verem felhasználásával. Elemezzük a sor műveleteinek a végrehajtási idejét.

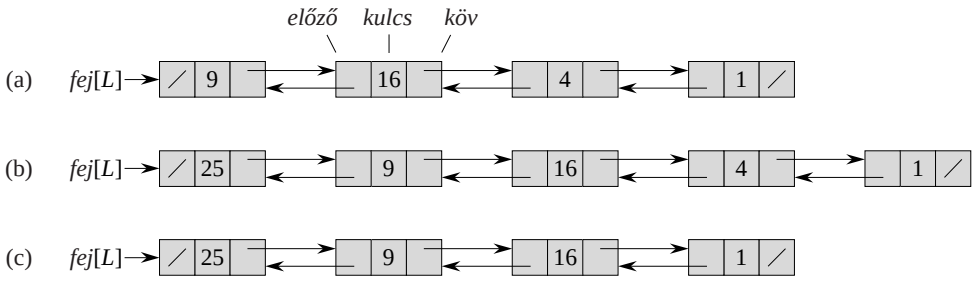
10.1-7. Adjuk meg a verem megvalósítását két sor felhasználásával. Elemezzük a verem műveleteinek végrehajtási idejét.

10.2. Láncolt listák

A ***láncolt lista*** olyan adatszerkezet, amelyben az objektumok lineáris sorrendben követik egymást. A tömböknél a lineáris sorrendet a tömbindexek határozzák meg, ezzel szemben a láncolt listákban mutatók valósítják meg az elemek lineáris elrendezettségét: a lista minden objektuma tartalmaz egy mutatót, amely a következő objektumra mutat. A láncolt lista egyszerű és rugalmas eszköz a dinamikus halmazok ábrázolására, és lehetővé teszi a III. rész bevezetőjében felsorolt műveletek (nem feltétlenül hatékony) megvalósítását.

A 10.3. ábrán az L ***kétszeresen láncolt lista*** látható, amelyben az elemek a *kulcs* mezőn mellett két mutató mezőt tartalmaznak, ezeket *köv* és *előző* azonosítja. A listaelemek tartalmazhatnak további adatokat is. A lista egy x eleme esetén *köv*$[x]$ az x rákövetkezőjére, *előző*$[x]$ az x-et megelőző elemre mutat a listában. Ha *előző*$[x]$ = NIL, akkor az x elemnek nincs megelőzője, vagyis x az első elem a listában; úgy is mondjuk, hogy x a lista ***feje.*** Ha *köv*$[x]$ = NIL, akkor x a lista utolsó eleme, vagy másképpen, x lista ***vége.*** A lista egy attribútuma az első elemre mutató *fej*$[L]$ pointer. Ha *fej*$[L]$ = NIL, akkor a lista üres.

A lista adatszerkezetnek számos változata van. Egy lista lehet egyszeresen vagy kétszeresen láncolt, lehet rendezett vagy nem rendezett, és lehet ciklikus vagy nem ciklikus. Ha egy lista ***egyszeresen láncolt,*** akkor elemeiben csak a *köv* mutató szerepel. Ha egy lista ***rendezett,*** akkor a mutatók által meghatározott sorrend megegyezik az elemekben tárolt kulcsmezők növekvő sorrendjével; a legkisebb kulcsérték a lista fejében található, a legnagyobb kulcsérték pedig a lista végében, azaz utolsó elemében szerepel. Ha egy lista ***rendezetlen,*** akkor elemeinek sorrendje tetszőleges lehet. A ***ciklikus listát*** az jellemzi, hogy a fej



10.3. ábra. (a) Az $\{1, 4, 9, 16\}$ dinamikus halmaz ábrázolása kétszeresen láncolt L listával. A lista minden eleme egy kulcsmezőt és két mutató mezőt tartalmaz; a (nyilakkal ábrázolt) pointerek rendre a megelőző és a rákövetkező elemekre mutatnak. Az utolsó elem *köv* mutatója és a fej *előző* mutatója NIL, amit 'per'-jel (slash) jelöl. A *fej*[L] pointer a lista fejére mutat. **(b)** A LISTÁBA-BESZÚR(L, x) művelet végrehajtása után, ahol *kulcs*[x] = 25, a láncolt lista a 25 kulcsértékű új objektummal bővül, és ez lesz a lista új feje. Ez az új objektum a korábbi, 9 kulcsértékű fejre mutat. **(c)** A LISTÁBÓL-TÖRÖL(L, x) művelet végrehajtásának eredménye, ahol x a 4-es kulcsú elemre mutat.

előző mutatója a lista végére, a vége *köv* pointere pedig a fejre mutat. A ciklikus listát ezért úgy is tekinthetjük, mint egy elemekből alkotott gyűrűt. Ebben a részben ezután már csak kétszeresen láncolt, rendezetlen listákkal foglalkozunk.

Keresés láncolt listában

A LISTÁBAN-KERES(L, k) eljárás lineáris módon keresi az első k kulcsú elemet az L listában, és az arra mutató pointert adja vissza, ha megtalálta az elemet. Ha nincs k kulcsú elem a listában, akkor a visszaadott érték NIL. A 10.3(a) ábrán látható listára alkalmazva, a LISTÁBAN-KERES($L, 4$) művelet a harmadik elemre mutató pointert adja vissza, míg a LISTÁBAN-KERES($L, 7$) művelet NIL-t ad vissza.

LISTÁBAN-KERES(L, k)

```
1  x ← fej[L]
2  while x ≠ NIL és kulcs[x] ≠ k
3        do x ← köv[x]
4  return x
```

Ha egy n elemű listában keresünk, akkor lehetséges, hogy minden elemet meg kell vizsgálni, ezért a LISTÁBAN-KERES eljárás végrehajtási ideje a legrosszabb esetben $\Theta(n)$.

Beszúrás láncolt listába

A LISTÁBA-BESZÚR eljárás egy megadott (kitöltött *kulcs* mezőjű) x elemet „befűz" a lista elejére úgy, ahogyan ez a 10.3(b) ábrán látható.

LISTÁBA-BESZÚR(L, x)

```
1  köv[x] ← fej[L]
2  if fej[L] ≠ NIL
3     then előző[fej[L]] ← x
```

4 *fej*[*L*] ← *x*
5 *előző*[*x*] ← NIL

Egy *n* elemű listára végrehajtott LISTÁBA-BESZÚR művelet futási ideje $O(1)$.

Törlés láncolt listából

A LISTÁBÓL-TÖRÖL eljárás eltávolít egy *x* elemet az *L* láncolt listából. A törléshez meg kell adni az *x*-re mutató pointert. Az eljárás ekkor a mutatók megfelelő átállításával „kifűzi" *x*-et a listából. Ha egy adott kulcsú elemet akarunk törölni, akkor előbb végre kell hajtani a LISTÁBAN-KERES eljárást, amely visszaadja a törlendő elem pointerét.

LISTÁBÓL-TÖRÖL(*L*, *x*)

1 **if** *előző*[*x*] ≠ NIL
2 **then** *köv*[*előző*[*x*]] ← *köv*[*x*]
3 **else** *fej*[*L*] ← *köv*[*x*]
4 **if** *köv*[*x*] ≠ NIL
5 **then** *előző*[*köv*[*x*]] ← *előző*[*x*]

A 10.3(c) ábrán látható egy elem törlése láncolt listából. A LISTÁBÓL-TÖRÖL művelet végrehajtási ideje $O(1)$, ha azonban egy adott kulcsú elemet kívánunk törölni, akkor ehhez a legrosszabb esetben $\Theta(n)$ idő szükséges, mivel előbb végre kell hajtani a LISTÁBAN-KERES eljárást.

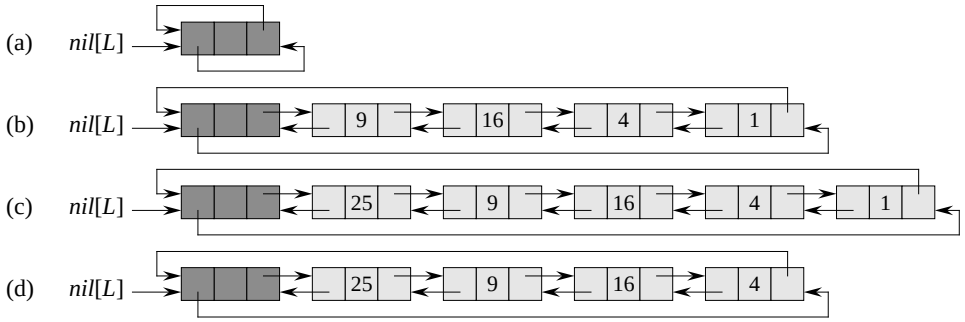
Őrszemek

A LISTÁBÓL-TÖRÖL eljárás kódja egyszerűsödne, ha nem kellene tekintettel lennünk a lista határaira a két végén.

LISTÁBÓL-TÖRÖL′(*L*, *x*)

1 *köv*[*előző*[*x*]] ← *köv*[*x*]
2 *előző*[*köv*[*x*]] ← *előző*[*x*]

Az ***őrszem*** (sentinel) olyan speciális elem, amelynek segítségével egyszerűbben lehet a lista határait ellenőrizni. Tegyük fel például, hogy az *L* lista tartalmaz egy *nil*[*L*] mutatóval azonosított objektumot. Ez a listaelem „megtestesíti" a NIL üres pointer által mutatott (tulajdonképpen nem létező) objektumot, egyébként pedig ugyanolyan felépítésű, mint a lista összes többi eleme. A műveletek kódjában ezután a NIL minden előfordulását a *nil*[*L*] őrszemre való hivatkozással helyettesíthetjük. Ennek megfelelően, amint az a 10.4. ábrán is látható, egy kétirányú lista olyan ***ciklikus kétirányú őrszemes listává*** alakul, amelyben a *nil*[*L*] őrszem a lista első és utolsó eleme között található; a *köv*[*nil*[*L*]] mutató a lista fejére, az *előző*[*nil*[*L*]] pointer pedig az utolsó elemre mutat. Ezzel összhangban az utolsó elem *köv* mezője és az első elem *előző* mezője egyaránt a *nil*[*L*] elemre mutat. A *fej*[*L*] attribútumra nincs szükségünk a továbbiakban, mivel minden rá vonatkozó hivatkozást *köv*[*nil*[*L*]]-lel helyettesíthetünk. Az üres lista csak az őrszemet tartalmazza, és annak mindkét pointere ugyanerre az elemre mutat, azaz *köv*[*nil*[*L*]] = *nil*[*L*] és *előző*[*nil*[*L*]] = *nil*[*L*].



10.4. ábra. Egy ciklikus kétszeresen láncolt őrszemes lista. A $nil[L]$ őrszem a lista feje és vége között helyezkedik el. A $fej[L]$ attribútumra tovább nincs szükség, mivel a lista feje elérhető a $köv[nil[L]]$ mutatóval. **(a)** Az üres lista. **(b)** A 10.3(a) ábrán szereplő lista őrszemmel kiegészített ciklikus változata; a lista fejében a kulcs értéke 9, a lista végének a kulcsa 1. **(c)** A lista a LISTÁBA-BESZÚR′(L, x) művelet végrehajtása után, ahol $kulcs[x] = 25$. A beszúrt új objektum lesz a lista feje. **(d)** A lista az 1-es kulcsú objektum törlése után. A lista új vége a 4-es kulcsú elem lesz.

A LISTÁBAN-KERES eljárás kódjában csak a NIL-t és $fej[L]$-et kell a fent megadott módon megváltoztatni.

LISTÁBAN-KERES′(L, k)

```
1  x ← köv[nil[L]]
2  while x ≠ nil[L] és kulcs[x] ≠ k
3        do x ← köv[x]
4  return x
```

A kétsorosra egyszerűsödött LISTÁBÓL-TÖRÖL′ eljárást már megadtuk. A következő eljárás beszúr egy elemet a listába.

LISTÁBA-BESZÚR′(L, x)

```
1  köv[x] ← köv[nil[L]]
2  előző[köv[nil[L]]] ← x
3  köv[nil[L]] ← x
4  előző[x] ← nil[L]
```

A LISTÁBA-BESZÚR′ és a LISTÁBÓL-TÖRÖL′ műveletek hatását a 10.4. ábra szemlélteti.

Az őrszemek használata ritkán csökkenti az adatszerkezetek műveleteire adható aszimptotikus időkorlátokat, az azokban szereplő állandókat azonban csökkentheti. Nem is annyira a futási idő mérséklése céljából használjuk a speciális rekordokat, hanem azért, mert általuk áttekinthetőbb és érthetőbb lesz a műveletek kódja. A láncolt lista esetén például a műveletek kódja egyszerűsödött, de csak $O(1)$ időt sikerült megtakarítani a LISTÁBA-BESZÚR′ és a LISTÁBÓL-TÖRÖL′ eljárásokban. Más esetekben azonban a speciális rekordok alkalmazásával lényegesen rövidíthető a kód a ciklusokban, és ezáltal csökkenthető a futási idő képletében – mondjuk – az n vagy n^2 együtthatója.

Vannak olyan esetek, amelyekben nem célszerű őrszemet bevezetni. Ha például sok kisméretű listával dolgozunk, akkor az őrszemek együttesen már jelentős memóriát foglalhatnak el. Ebben a könyvben csak akkor alkalmazzuk az őrszemeket, ha ezáltal valóban egyszerűsödik a kód.

Gyakorlatok

10.2-1. Megvalósítható-e a dinamikus halmazokra értelmezett BESZÚR művelet egyszeresen láncolt listákra $O(1)$ időben? Mi a helyzet a TÖRÖL művelettel?

10.2-2. Valósítsuk meg a vermet egyszerű láncolt listával. A VEREMBE és VEREMBŐL műveletek végrehajtási ideje maradjon $O(1)$.

10.2-3. Valósítsuk meg a sort egyszerű láncolt listával. A SORBA és SORBÓL műveletek végrehajtási ideje maradjon $O(1)$.

10.2-4. Amint láttuk, a LISTÁBAN-KERES′ eljárás az iteráció minden lépésében két vizsgálatot végez: ellenőrzi az $x \neq nil[L]$, valamint a $kulcs[x] \neq k$ feltétel teljesülését. Adjuk meg, hogyan lehetne az $x \neq nil[L]$ vizsgálatot kiküszöbölni a ciklusból.

10.2-5. Írjuk meg a BESZÚR, TÖRÖL és KERES szótár-műveleteket egyszeresen láncolt ciklikus listára. Mennyi az egyes eljárások futási ideje?

10.2-6. A dinamikus halmazokra értelmezett EGYESÍT művelet bemenő paraméterei az S_1 és az S_2 diszjunkt halmazok, visszaadott értéke pedig az $S = S_1 \cup S_2$ halmaz, amely S_1 és S_2 minden elemét tartalmazza. Általában megengedik, hogy az egyesítés „elrontsa” az S_1 és S_2 halmazokat. Mutassuk meg, hogyan lehet $O(1)$ idejű EGYESÍT műveletet megvalósítani alkalmas lista adatszerkezet felhasználásával.

10.2-7. Adjunk olyan nemrekurzív eljárást, amely $\Theta(n)$ időben megfordít egy n elemű egyszeresen láncolt listát. Az eljárás konstans méretű segédmemóriát használhat.

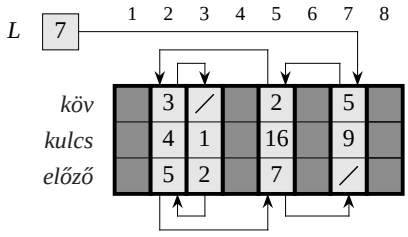
10.2-8.⋆ Valósítsuk meg a kétszeresen láncolt lista adatszerkezetet egy olyan listával, amelynek elemei a szokásos két mutató (*köv* és *előző*) helyett mindössze egyetlen $km[x]$ mutatót tartalmaznak. Tegyük fel, hogy a pointerek k-bites egészként ábrázolhatók, és definiáljuk $km[x]$-et úgy, mint a $köv[x]$ és az $előző[x]$ értékek között elvégzett k-bites „kizáró vagy” eredményét, azaz legyen $km[x] = köv[x]$ XOR $előző[x]$. (A NIL értékét 0-val ábrázoljuk.) Adjuk meg, hogy milyen információ szükséges a lista első elemének az eléréséhez. Mutassuk meg, hogyan lehet a KERES, TÖRÖL és BESZÚR műveleteket egy ilyen listára megvalósítani. Mutassuk meg azt is, hogyan lehet egy ilyen listát $O(1)$ időben megfordítani.

10.3. Mutatók és objektumok megvalósítása

Hogyan lehet mutatókat és objektumokat megvalósítani olyan nyelvekben – például Fortranban –, amelyek nem támogatják ezt a megfelelő adattípusokkal? Ebben a részben két módszert mutatunk be a láncolt adatszerkezetek megvalósítására pointer adattípus felhasználása nélkül. Az objektumok és a mutatók ábrázolására tömböket és tömbindexeket fogunk felhasználni.

Objektumok többtömbös ábrázolása

Azonos felépítésű objektumokból álló adatszerkezeteket úgy is ábrázolhatunk, hogy minden mező számára egy-egy tömböt használunk fel. Például a 10.3(a) ábrán bemutatott láncolt listát a 10.5. ábrán három tömbbel valósítottuk meg. A *kulcs* azonosítójú tömb tartalmazza a dinamikus halmazban előforduló kulcsértékeket, a mutatókat pedig a *köv* és *előző* tömbök tárolják. Egy adott tömbindex esetén $kulcs[x]$, $köv[x]$ és $előző[x]$ együtt ábrázolják a láncolt lista egy objektumát. Ilyen értelemben az x mutató nem más, mint egy közös index a *kulcs*, *köv* és *előző* tömbökben.



10.5. ábra. A 10.3(a) ábrán bemutatott láncolt lista ábrázolása a *kulcs*, *köv* és *előző* tömbökkel. A tömbök bármely három egymás alatti eleme együtt ábrázol egy objektumot. A tárolt indexek mindegyike megfelel az ábra tetején elhelyezett indexsor valamely elemének. Az indexekkel megvalósított láncolást nyilak is mutatják. A lista elemei a világos mezőjű objektumokat foglalják el. Az L változó tartalmazza a lista fejének az indexét.

A 10.3(a) ábrán a láncolt listában a 4-es kulcsú objektum rákövetkezője a 16-os kulcsú objektumnak. A 10.5. ábrán a 4-es kulcsértéket *kulcs*[2] tartalmazza, míg a 16-os kulcs *kulcs*[5]-ben szerepel, ezért *köv*[5] = 2 és *előző*[2] = 5. Az utolsó elem *köv* mezőjében és a fej *előző* mezőjében szereplő NIL pointer ábrázolására általában olyan egész értéket (például 0-t vagy −1-et) használunk, amelyet nem értelmeztünk tömbindexként. Az L változó tartalmazza a lista fejének az indexét.

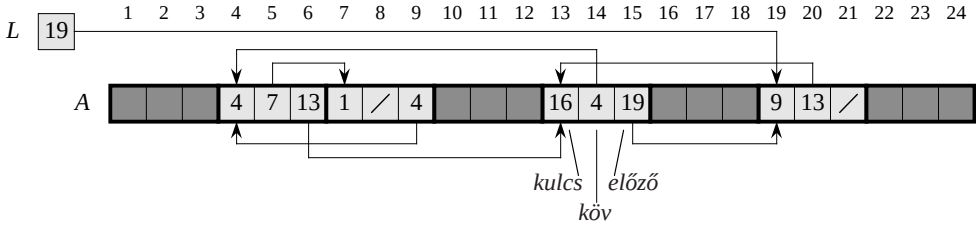
Az általunk használt pszeudokódban egyaránt zárójelekkel jelöljük egy tömb indexelését és egy objektum valamely mezőjének (attribútumának) a kiválasztását. Ebben a megvalósításban a *kulcs*[x], *köv*[x] és *előző*[x] kifejezések valóban megfelelnek mindkét jelentésnek és összhangban állnak az implementációs gyakorlattal.

Objektumok egytömbös ábrázolása

A számítógép memóriájában a szavak általában 0 és $M - 1$ közé eső egészekkel címezhetők, ahol M egy alkalmas egész szám. A legtöbb programozási nyelvben egy objektum a memória egy összefüggő területén helyezkedik el. Az objektumra mutató pointer ekkor egyszerűen az első szónak a címe, és az objektum belsejében levő helyeket úgy lehet indexelni, hogy a pointerhez hozzáadunk egy kiegészítő értéket.

Ugyanezt a stratégiát követve implementálhatunk objektumokat olyan programozási környezetekben, amelyek nem tartalmaznak pointer adattípust. A 10.6. ábra azt szemlélteti például, hogy a 10.3(a) és 10.5. ábrán bemutatott láncolt listát hogyan lehet egyetlen A tömbben tárolni. Egy objektum a folytonos $A[j\,..\,k]$ résztömbben helyezkedik el. Az objektumra a j index mutat, és egyes mezőinek a címét rendre a 0 és $k - j$ közötti kiegészítő értékek hozzáadásával kapjuk. A 10.6. ábrán a *kulcs*, *köv* és *előző* mezőknek rendre a 0, 1 és 2 kiegészítő értékek felelnek meg. Így, ha i egy objektumra mutat, akkor *köv*[i] és *előző*[i] rendre a tömb $A[i + 1]$ és $A[i + 2]$ elemeit jelölik.

Az egytömbös ábrázolás rugalmas vonása az, hogy lehetővé teszi különböző hosszúságú objektumok tárolását ugyanabban a tömbben. Az eltérő felépítésű objektumokból álló (heterogén) adatszerkezetek kezelése jóval nehezebb feladat, mint az azonos felépítésű elemeket tartalmazó (homogén) adatszerkezeteké. A könyvben tárgyalt adatszerkezetek többségét azonos felépítésű elemek alkotják, ezért céljainknak megfelel az objektumok több tömbbel való ábrázolása.



10.6. ábra. A 10.3(a) és a 10.5. ábrán bemutatott láncolt lista ábrázolása egyetlen *A* tömbben. A lista minden objektuma a tömb három egymás utáni elemét foglalja el. Egy objektum pointere az objektum első elemének az indexe, amelyhez a *kulcs*, *köv* és *előző* mezők esetén ehhez rendre a 0, 1 és 2 értékeket kell hozzáadni. A lista elemeit a világos mezőjű objektumok tartalmazzák, sorrendjüket pedig nyilak mutatják.

Objektumok lefoglalása és felszabadítása

Ha egy új elemet szeretnénk beszúrni egy kétszeresen láncolt listával ábrázolt dinamikus halmazba, akkor rendelkeznünk kell egy pointerrel, amely egy használaton kívüli objektumra mutat. Az új objektumok lefoglalása céljából előnyös, ha nyilvántartjuk azokat az objektumokat, amelyeket nem használunk a szóban forgó láncolt listában. Vannak olyan rendszerek, amelyekben külön ***szabadhelygyűjtő*** (garbage collector) eljárás feladata annak meghatározása, hogy mely objektumok szabadok. Számos alkalmazás azonban annyira egyszerű, hogy maga oldja meg a memóriakezelés feladatát. Az azonos típusú objektumok lefoglalásának és felszabadításának problémáját a kétszeresen láncolt listák többtömbös ábrázolásában mutatjuk be és tárgyaljuk.

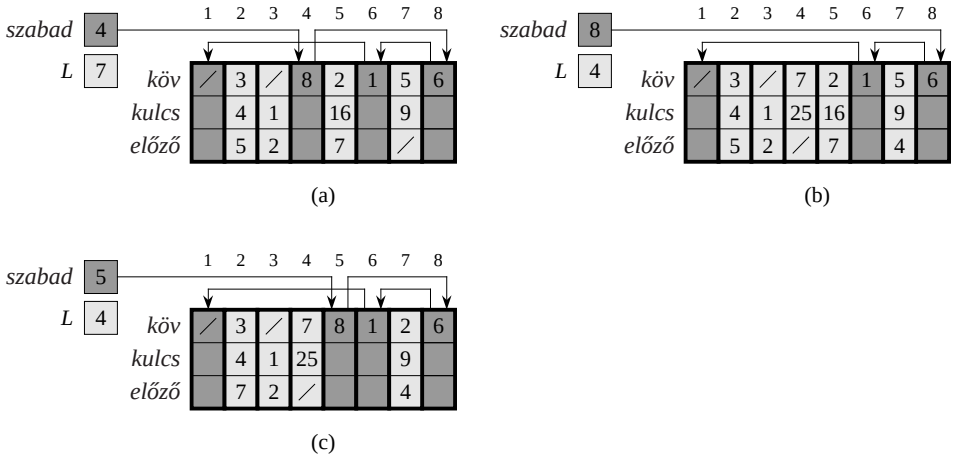
Tegyük fel, hogy a többtömbös ábrázolásban szereplő tömbök mérete m és a dinamikus halmaz $n \leq m$ elemet tartalmaz egy adott időpontban. Ekkor n objektum ábrázolja a halmaz elemeit, a többi $m - n$ objektum pedig ***szabad.*** A szabad objektumokat lehet felhasználni a dinamikus halmaz ezután beszúrásra kerülő elemeinek az ábrázolására.

A használaton kívüli objektumokat egyszeresen láncolt listában tartjuk nyilván; ezt nevezzük ***szabadlistának.*** A szabadlistában csak a *köv* tömb tartalmaz mutatókat. A szabadlista fejének az indexét a *szabad* globális változó tartalmazza. Ha az *L* láncolt listával ábrázolt dinamikus halmaz nem üres, akkor a szabadlista és *L* elemei váltakozó sorrendben követhetik egymást a tömbökben; ilyenkor a két lista „átfonja egymást”, mint például a 10.7. ábrán is. Jegyezzük meg, hogy az ábrázolás minden objektuma vagy az *L* listához, vagy a szabadlistához tartozik, de nem szerepelhet egyszerre mindkettőben.

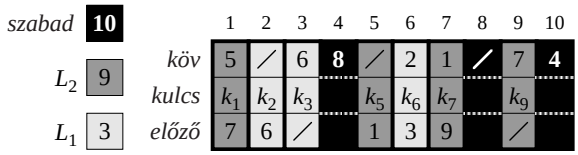
A szabadlista egy verem: az a következő lefoglalható objektum, amelyiket legutoljára szabadítottunk fel. Az objektumok lefoglalását és felszabadítását ezért a VEREMBE és VEREMBŐL műveletek listákkal megvalósított változatával oldjuk meg. Feltesszük, hogy a következő eljárásokban a *szabad* globális változó a szabadlista első elemére mutat.

OBJEKTUMOT-LEFOGLAL()

```
1  if szabad = NIL
2     then error „nincs szabadhely”
3     else  x ← szabad
4           szabad ← köv[x]
5           return x
```



10.7. ábra. Az OBJEKTUMOT-LEFOGLAL és az OBJEKTUMOT-FELSZABADÍT műveletek hatása. **(a)** A 10.5. ábrán bemutatott lista (világos mezők) és a szabadlista (sötét mezők). A szabadlista felépítését nyilak mutatják. **(b)** Az OBJEKTUM-LEFOGLAL() hívása (ami 4-et ad vissza), *kulcs*[4] beállítása 25-re és a LISTÁBA-BESZÚR$(L, 4)$ végrehajtása utáni állapot. A szabadlista feje a 8-as indexű objektum lett, amelyre előzőleg *köv*[4] mutatott. **(c)** A LISTÁBÓL-TÖRÖL$(L, 5)$ és az OBJEKTUMOT-FELSZABADÍT(5) végrehajtásának a hatása. Az 5-ös indexű objektum lett a szabadlista feje, amelynek a rákövetkezője a 8-as indexű objektum.



10.8. ábra. Két láncolt lista, az L_1 (világos) és az L_2 (szürke), valamint egy szabadlista (sötét) „összefonódása” a *köv*, *kulcs* és *előző* tömbökben.

OBJEKTUMOT-FELSZABADÍT(x)

1 $köv[x] \leftarrow szabad$
2 $szabad \leftarrow x$

A szabadlista kezdetben n objektumot tartalmaz. Ha a szabadlista kimerül, akkor a helyfoglaló eljárás hibát jelez. Gyakori az, hogy több láncolt listához egy közös szabadlista tartozik. A 10.8. ábrán látható *köv*, *kulcs* és *előző* tömbökben két láncolt lista és egy hozzájuk tartozó szabadlista szerepel együtt.

Mindkét eljárás $O(1)$ időben fut, ami igen előnyös a gyakorlatban. Az itt bemutatott szabadhelykezelés megvalósítható bármely homogén adatszerkezetre úgy, hogy a használaton kívüli objektumok egyik mezőjét úgy használjuk, mint a szabadlista *köv* mutatóját.

Gyakorlatok

10.3-1. Adjuk meg a $\langle 13, 4, 8, 19, 5, 11 \rangle$ sorozatot tartalmazó kétszeresen láncolt lista többtömbös és egytömbös ábrázolását. Mindkét reprezentációról készítsünk ábrát.

10.3-2. Írjuk meg az OBJEKTUMOT-LEFOGLAL és az OBJEKTUMOT-FELSZABADÍT eljárásokat az egytömbös ábrázolásra.

10.3-3. Miért nem használtuk fel az objektumok *előző* mezőjét az OBJEKTUMOT-LEFOGLAL és az OBJEKTUMOT-FELSZABADÍT eljárásokban?

10.3-4. Gyakran előnyös az, ha a kétszeresen láncolt lista elemeit a memória összefüggő részében tartjuk, mondjuk az első m indexpozícióban, többtömbös ábrázolásban. (Ez az eset például egy lapozott, virtuális memóriájú számítógépes környezetben.) Hogyan kell az OBJEKTUMOT-LEFOGLAL és az OBJEKTUMOT-FELSZABADÍT eljárásokat megírni az ilyen tömör ábrázolás esetén? Tegyük fel, hogy a láncolt lista elemeire kívülről nem mutatnak pointerek. (*Útmutatás.* Használjuk a verem tömbös megvalósítását.)

10.3-5. Az L kétszeresen láncolt listát az n hosszúságú *kulcs*, *előző* és *köv* tömbökben tároljuk. Ezek a tömbök tartalmazzák a kétszeresen láncolt F szabadlistát is. A szabadhelykezelést az OBJEKTUMOT-LEFOGLAL és az OBJEKTUMOT-FELSZABADÍT eljárások valósítják meg. Tegyük fel, hogy az L lista m elemű, a szabadlistában pedig $n - m$ elem van. Írjuk meg a LISTÁT-TÖMÖRÍT(L, F) eljárást, amely úgy rendezi át a tömböket, hogy az L lista elemei az $1, 2, \ldots, m$ pozíciókat foglalják el, az F szabadlista elemei pedig az $m + 1, m + 2, \ldots, n$ pozíciókon helyezkednek el. Az eljárás végrehajtási ideje legyen $\Theta(m)$, és csak konstans méretű segédmemória használatára van lehetőség. Körültekintően indokoljuk meg az eljárás helyességét.

10.4. Gyökeres fák ábrázolása

A listák ábrázolására megadott módszer, amelyet az előző részben láttunk, kiterjeszthető más homogén adatszerkezetekre is. Ebben a részben azt vizsgáljuk meg, hogy a gyökeres fákat hogyan lehet láncolt adatszerkezetekkel ábrázolni. Először a bináris fákat tekintjük, majd olyan fákra adunk módszert, amelyekben egy csúcsnak tetszőleges számú gyereke lehet.

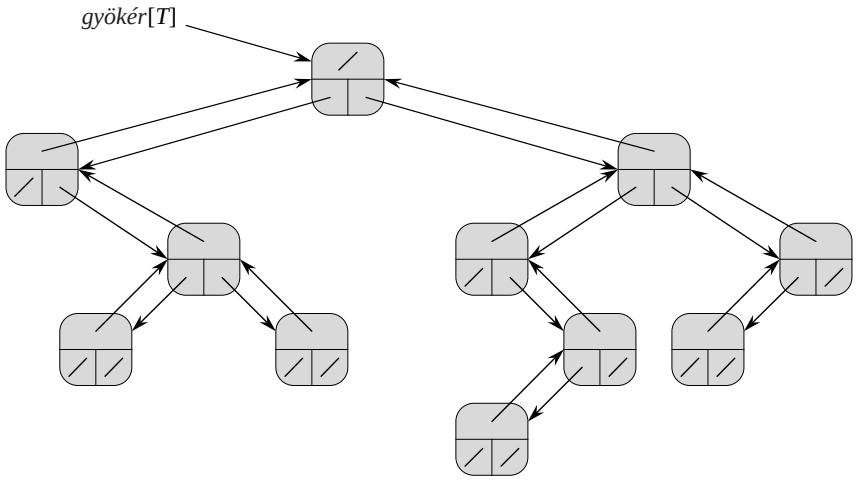
A fa minden csúcsát egy objektummal ábrázoljuk. Akárcsak a listáknál, itt is feltételezzük azt, hogy minden csúcs tartalmaz egy *kulcs* mezőt. A további (érdeklődésünkre számot tartó) mezők, amelyek más csúcsokra mutató pointereket tartalmaznak, a fa típusától függően változnak.

Bináris fák

Amint a 10.9. ábrán látható, egy T bináris fa csúcsaiban a *szülő*, *bal* és *jobb* mezőkben tárolt pointerek rendre a csúcs szülőjére, bal és jobb oldali gyerekére mutatnak. Ha *szülő*$[x]$ = NIL, akkor x a gyökér. Ha az x csúcsnak nincs bal, illetve jobb gyereke, akkor *bal*$[x]$ = NIL, illetve *jobb*$[x]$ = NIL. A T fa gyökerére a *gyökér*$[T]$ attribútum mutat. Ha *gyökér*$[T]$ = NIL, akkor a fa üres.

Nemkorlátos leágazásszámú gyökeres fák

A bináris fák ábrázolási módszere kiterjeszthető a fák minden olyan osztályára, amelyben bármely csúcs gyerekeinek a száma nem haladja meg egy k konstans értékét: a *bal* és a *jobb* mezőket helyettesítjük a *gyerek*$_1$, *gyerek*$_2$, $\ldots$, *gyerek*$_k$ mezőkkel. Ez a módszer nem alkalmazható akkor, ha nem adható felső korlát a gyerekek számára a csúcsokban, mert akkor nem tudhatjuk előre, hogy hány mezőt (a többtömbös ábrázolásban tömböt) kell a muta-



10.9. ábra. A T bináris fa ábrázolása. Bármely x csúcs a következő mezőket tartalmazza: *szülő*[x] (fent), *bal*[x] (balra lent) és *jobb*[x] (jobbra lent). A *kulcs* mezők nem szerepelnek az ábrán.

tók számára lefoglalni. Továbbá, ha korlátos is a gyerekek száma, de ez a korlát nagy érték és a legtöbb csúcsnak kevés gyereke van, akkor – noha a módszer használható – jelentős mennyiségű memóriát veszítünk.

Szerencsére, egy ügyes megfeleltetéssel bináris fával lehet ábrázolni azokat a fákat, amelyekben tetszőleges számú gyereke lehet az egyes csúcsoknak. Ennek az ábrázolási módnak még az is az előnye, hogy csak $O(n)$ memóriát használ egy n csúcsú fához. A ***bal-gyerek, jobb-testvér ábrázolást*** a 10.10. ábra szemlélteti. A csúcsok továbbra is tartalmazzák a *szülő* pointert, és a T fa gyökerére változatlanul *gyökér*[T] mutat. A fa minden csúcsában a gyerekekre mutató pointerek helyett azonban mindössze két mutatót tárolunk:

1. *bal-gyerek*[x] az x csúcs gyerekei közül bal szélen levőre mutat, és
2. *jobb-testvér*[x] az x csúcsnak arra a testvérére mutat, amelyik közvetlenül jobbra mellette található.

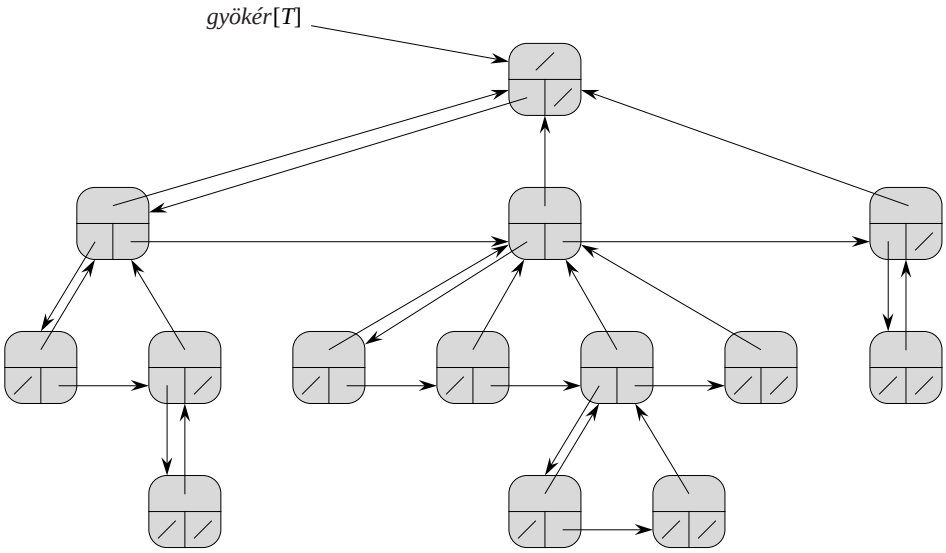
Ha az x csúcsnak nincs gyereke, akkor *bal-gyerek*[x] = NIL, ha pedig x testvérei között a jobb szélső, akkor *jobb-testvér*[x] = NIL.

További módszerek fák ábrázolására

A gyökeres fákat néha másképpen ábrázoljuk. A 6. fejezetben például a kupac adatszerkezetet, ami egy teljes bináris fa, egyetlen tömbbel és egy indexszel ábrázoljuk. A 21. fejezetben szereplő fákat mindig a gyökér felé haladva járjuk be, ezért csak a szülőre mutató pointereket használjuk; a gyerekekre mutató pointerek tárolására nincs szükség. Számos egyéb ábrázolási mód is lehetséges. Az alkalmazástól függ, hogy melyikük a legelőnyösebb.

Gyakorlatok

10.4-1. Rajzoljuk le azt a bináris fát, amelyet a következő mezők határoznak meg úgy, hogy a fa gyökerének az indexe 6.



10.10. ábra. A T bináris fa ábrázolása a bal-gyerek, jobb-testvér reprezentációval. Bármely x csúcs a következő mezőket tartalmazza: *szülő*[x] (fent), *bal-gyerek*[x] (balra lent) és *jobb-testvér*[x] (jobbra lent). A kulcsok nem szerepelnek az ábrán.

index	kulcs	bal	jobb
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2. Írjunk olyan $O(n)$ futási idejű rekurzív eljárást, amely kiírja egy n csúcsú bináris fa összes csúcsában szereplő kulcsértéket.

10.4-3. Írjunk olyan $O(n)$ futási idejű nemrekurzív eljárást, amely kiírja egy n csúcsú bináris fa összes csúcsában szereplő kulcsértéket. Használjunk ehhez segéd-adatszerkezetként egy vermet.

10.4-4. Írjunk olyan $O(n)$ futási idejű eljárást, amely kiírja az összes kulcsértéket egy tetszőleges n csúcsú fából, amelyet bal-testvér, jobb-gyerek reprezentációban adtak meg.

10.4-5.⋆ Írjunk olyan $O(n)$ futási idejű nemrekurzív eljárást, amely kiírja az összes kulcsértéket egy adott n csúcsú bináris fa csúcsaiból. Legfeljebb konstans méretű segédmemóriát használhatunk, és az eljárás még időlegesen se módosítsa a fát.

10.4-6.⋆ A bal-gyerek, jobb-testvér ábrázolásban egy tetszőleges, gyökeres fa csúcsai a következő három mutatót tartalmazzák: *bal-gyerek*, *jobb-testvér* és *szülő*. Ezek segítségével bármely csúcsból konstans időben elérhető és azonosítható a csúcs szülője, és a gyerekek

számával arányos lineáris időben elérhető és azonosítható minden gyereke. Hogyan valósítható meg a csúcsokban alkalmasan felvett két mutató és egy logikai érték segítségével az, hogy a csúcs szülőjét és gyerekeit a gyerekek számával arányos lineáris időben elérjük és azonosítsuk.

Feladatok

10-1. Listaszerkezetek összehasonlítása

Mennyi a táblázatban szereplő dinamikus halmazműveletek aszimptotikus futási ideje a legrosszabb esetben, a megadott négy listatípus mindegyikére?

	rendezetlen egyszeresen láncolt	rendezett egyszeresen láncolt	rendezetlen kétszeresen láncolt	rendezett kétszeresen láncolt
KERES(L, k)				
BESZÚR(L, x)				
TÖRÖL(L, x)				
KÖVETKEZŐ(L, x)				
ELŐZŐ(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2. Összefésülhető kupacok ábrázolása láncolt listával

Az ***összefésülhető kupac*** adatszerkezeteken a következő műveleteket valósítjuk meg: KUPACOT-LÉTREHOZ (ez létrehoz egy üres összefésülhető kupacot), BESZÚR, MINIMUM, KIVESZ-MIN és EGYESÍT.[1] Mutassuk meg, hogyan lehet az összefésülhető kupacokat láncolt listákkal megvalósítani az alábbi esetekben. Próbáljuk meg a műveleteket a lehető leghatékonyabbá tenni. Elemezzük a műveletek futási idejét a dinamikus halmaz(ok) elemszámának függvényében.

a. A listák rendezettek.

b. A listák rendezetlenek.

c. A listák rendezetlenek és az összefésülendő dinamikus halmazok diszjunktak.

10-3. Keresés rendezett tömör listában

A 10.3-4. gyakorlat teszi fel azt a kérdést, hogyan lehet egy n elemű listát tömör módon egy tömb első n pozícióján tartani. Feltesszük, hogy minden kulcs különböző és a tömör formában tárolt lista rendezett, azaz $kulcs[i] < kulcs[köv[i]]$ minden $i = 1, 2, \ldots, n$ értékre úgy, hogy $köv[i] \neq$ NIL. Ilyen feltételek mellett megmutatjuk, hogy a következő, véletlen kiválasztást tartalmazó algoritmus várhatóan $O(\sqrt{n})$ időben keres a listában.

[1] Mivel a MINIMUM és a KIVESZ-MIN műveleteket vezettük be, adatszerkezetünket ***összefésülhető minimumválasztó kupacnak*** is nevezhetnénk. Hasonlóképpen, a MAXIMUM és a KIVESZ-MAXIMUM műveletek esetén ***összefésülhető maximumválasztó kupacról*** beszélhetnénk.

TÖMÖR-LISTÁBAN-KERES(L, n, k)

```
 1  i ← fej[L]
 2  while i ≠ NIL és kulcs[i] < k
 3       do j ← VÉLETLEN(1, n)
 4          if kulcs[i] < kulcs[j] és kulcs[j] ≤ k
 5             then i ← j
 6                  if kulcs[i] = k
 7                     then return i
 8          i ← köv[i]
 9  if i = NIL vagy kulcs[i] > k
10    then return NIL
11    else  return i
```

Ha törölnénk az eljárás 3–7. sorait, akkor a rendezett láncolt listában való keresés szokásos algoritmusához jutnánk, amelyben az i index sorban végighalad a lista minden elemén. A keresés befejeződik, ha az i index „túljut” a lista végén, vagy $kulcs[i] \geq k$. Az utóbbi esetben, ha $kulcs[i] = k$, akkor nyilván megtaláltuk a k értékű kulcsot. Ha azonban $kulcs[i] > k$, akkor már nem találhatunk k értékű kulcsot, így ésszerű a keresést befejezni.

A 3–7. sorokban azt próbáljuk elérni, hogy az i index előre ugorjon egy véletlen módon kiválasztott j pozícióba. Az ugrásnak akkor van értelme, ha $kulcs[j]$ nagyobb, mint $kulcs[i]$ és nem nagyobb, mint k; ilyen esetben j egy olyan pozíciót jelöl a listában, amelybe i eljutna a hagyományos keresés során is. Minthogy a lista elemeit tömör formában, egymás mellett tároltuk, ezért biztos, hogy az 1 és n között megválasztott j index listaelemre mutat, nem pedig szabadhelyre.

A TÖMÖR-LISTÁBAN-KERES eljárás hatékonyságának közvetlen elemzése helyett azt az alkalmasan módosított TÖMÖR-LISTÁBAN-KERES′ algoritmust elemezzük, amely két külön ciklust hajt végre a keresés során. Ez az algoritmus még egy t paramétert is kap, amely felső korlátot ad az első ciklus iterációinak a számára.

TÖMÖR-LISTÁBAN-KERES′(L, n, k, t)

```
 1  i ← fej[L]
 2  for q ← 1 to t
 3       do j ← VÉLETLEN(1, n)
 4          if kulcs[i] < kulcs[j] és kulcs[j] ≤ k
 5             then i ← j
 6                  if kulcs[i] = k
 7                     then return i
 8  while i ≠ NIL és kulcs[i] < k
 9       do i ← köv[i]
10  if i = NIL vagy kulcs[i] > k
11    then return NIL
12    else  return i
```

A TÖMÖR-LISTÁBAN-KERES(L, n, k) és a TÖMÖR-LISTÁBAN-KERES′(L, n, k, t) algoritmusok végrehajtásának összehasonlításához tegyük fel, hogy a VÉLETLEN$(1, n)$ eljárás hívásai mindkét algoritmusban ugyanazt az egész értékű sorozatot eredményezik.

a. Tegyük fel, hogy a TÖMÖR-LISTÁBAN-KERES(L, n, k) eljárás 2–8. soraiban szereplő **while** ciklus t számú iterációt hajt végre. Mutassuk meg, hogy ekkor a TÖMÖR-LISTÁBAN-KERES$'(L, n, k, t)$ is ugyanazt a választ adja, és a benne szereplő **for** és **while** ciklusok együtt legalább t számú iterációt hajtanak végre.

A TÖMÖR-LISTÁBAN-KERES$'(L, n, k, t)$ eljárás hívásával kapcsolatban legyen X_t az a valószínűségi változó, amely az i index távolságát adja meg a keresett k kulcsú elem pozíciójától a **for** ciklus 2–7. sorainak t számú végrehajtása után. (A távolságot természetesen a láncolt lista *köv* mutatóin át haladva értjük.)

b. Indokoljuk meg, hogy a TÖMÖR-LISTÁBAN-KERES$'(L, n, k, t)$ eljárás várható futási ideje $O(t + E[X_t])$.

c. Mutassuk meg, hogy $E[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$. (*Útmutatás.* Alkalmazzuk a (C.24) egyenlőséget.)

d. Mutassuk meg, hogy $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t + 1)$.

e. Bizonyítsuk be, hogy $E[X_t] \leq n/(t + 1)$.

f. Mutassuk meg, hogy a TÖMÖR-LISTÁBAN-KERES$'(L, n, k, t)$ algoritmus várható futási ideje $O(t + n/t)$.

g. Mutassuk meg, hogy a TÖMÖR-LISTÁBAN-KERES eljárás várható futási ideje $O(\sqrt{n})$.

h. Miért tételeztük fel a TÖMÖR-LISTÁBAN-KERES algoritmusban, hogy a kulcsok mind különbözők? Mutassunk rá, hogy a véletlen lépések nem feltétlenül javítják a hatékonyságot aszimptotikusan, ha a lista tartalmaz ismétlődő kulcsokat.

Megjegyzések a fejezethez

Aho, Hopcroft és Ullman [6], valamint Knuth [182] kitűnő források az elemi adatszerkezetek témakörében. Számos könyv az alapvető adatszerkezetek ismertetése mellett megadja azok implementációját is valamely programozási nyelven. Az ilyen típusú könyvek között említhető például Goodrich és Tamassia [128], Main [209], Shaffer [273], valamint Weiss [310, 312, 313]. Gonnet [126] kísérleti adatokat közöl számos adatszerkezet műveleteinek hatékonyságáról.

Nem állítható bizonyossággal, hogy a vermek és a sorok úgy jelentek meg először, mint a számítástudományban alkalmazott adatszerkezetek. A megfelelő fogalmak már a számítógépek megjelenése előtt kialakultak a matematikában és a korai kézi ügyvitelben. Knuth [182] idézi A. M. Turingot, aki már 1947-ben vermet javasolt a szubrutinkapcsolatok kezelésére.

A pointer-alapú láncolt adatszerkezetek szintén a „szakmai folklór” termékének tekinthetők. Knuth szerint már a korai dobmemóriájú számítógépekben nyilvánvalóan használtak pointereket. Az A-1 nyelv, amelyet G. M. Hopper fejlesztett ki 1951-ben, az algebrai formulákat bináris fákkal ábrázolta. Knuth hangsúlyozza az A. Newell, J. C. Shaw és H. A. Simon által 1956-ban kifejlesztett IPL-II nyelv jelentőségét a pointerek fontosságának felismerése és támogatása miatt. Az IPL-III nyelv, amelyet a szerzők 1957-ben készítettek el, már a verem típusműveleteit is tartalmazta utasításai között.

11. Hasító táblázatok

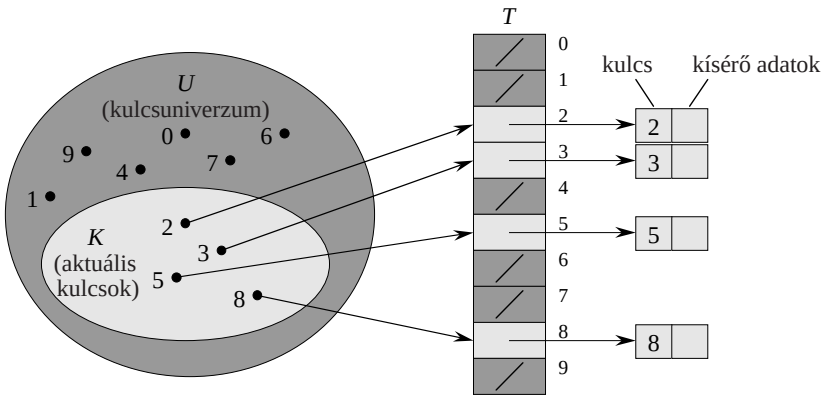
Számos olyan gyakorlati alkalmazással találkozhatunk, melyeknek csak a Besz úr, Keres és Töröl szótárműveleteket támogató dinamikus halmazra van szükségük. Például egy programozási nyelv fordítóprogramja szimbólumtáblát használ, melyben az elemek kulcsai a nyelv azonosítóinak megfelelő karakterláncok. A hasító táblázat hatékony adatszerkezet szótárak megvalósítására. Bár egy elem hasító táblázatban való keresésének ideje ugyanolyan hosszú lehet, mint egy láncolt lista esetében – a legrosszabb esetben $\Theta(n)$ –, a gyakorlatban a hasítás nagyon jól működik. Ésszerű feltételek mellett egy elem hasító táblázatban való keresésének várható ideje $O(1)$.

A hasító táblázat a megszokott tömbfogalom általánosítása. A tömbelemek közvetlen címzése lehetővé teszi, hogy egy tetszőleges pozíción levő elemet hatékonyan, $O(1)$ idő alatt vizsgálhassunk meg. A 11.1. alfejezetben részletesen tanulmányozzuk a közvetlen címzést. A közvetlen címzés akkor alkalmazható, amikor megengedhetjük magunknak, hogy olyan nagy tömböt foglaljunk le, amelyben minden lehetséges kulcsnak megfelel egy tömbelem.

Amikor az aktuálisan tárolt kulcsok száma a lehetséges kulcsok számához képest viszonylag kicsi, akkor a hasító táblázatok a tömbökben való közvetlen címzés hatékony alternatívái, mivel tipikus megvalósításaik a tárolt kulcsok számával arányos méretű tömböt használnak. A tömb elemeinek közvetlen címzésére a kulcs helyett egy, a kulcsból *kiszámított* érték szolgál. A 11.2. alfejezet a fő ötleteket mutatja be, a figyelmet a „láncolásra" fordítva, amely az „ütközések" – amikor is a hasító függvény egynél több kulcsot képez le ugyanarra a tömbindexre – egyik kezelési módszere. A 11.3. alfejezet leírja, hogyan lehet a kulcsokból a hasító függvények segítségével kiszámítani a tömbindexeket. A 11.4. alfejezet a „nyílt címzést" tekinti át, amely az ütközések egy további kezelési módszere. A vizsgálat fő iránya annak kimutatása, hogy a hasítás különösen hatékony és a gyakorlatban jól használható módszer: a legfontosabb szótárműveletek végrehajtása átlagosan $O(1)$ időt igényel. A 11.5. alfejezetben elmondjuk, hogyan támogatja a „tökéletes hasítás" a *legrosszabb esetben* $O(1)$ idejű kereséseket olyankor, ha a kulcsok halmaza statikus (azaz a kulcshalmaz a memóriába írás után többé nem változik).

11.1. Közvetlen címzésű táblázatok

A közvetlen címzés olyan egyszerű módszer, mely viszonylag kis méretű U kulcsuniverzumokra jól működik. Tegyük fel, hogy egy alkalmazáshoz olyan dinamikus halmazra van



11.1. ábra. Egy dinamikus halmaz megvalósítása a T közvetlen címzésű táblázattal. Az $U = \{0, 1, \ldots, 9\}$ univerzum minden kulcsának megfelel egy index a táblázatban. Az aktuális kulcsok $K = \{2, 3, 5, 8\}$ halmaza meghatározza a táblázatban azokat a réseket, melyek az elemekre mutató pointereket tartalmazzák. A többi, sötéten árnyékolt rés NIL-t tartalmaz.

szükségünk, melyben az elemek kulcsai az $U = \{0, 1, \ldots, m-1\}$ univerzumból valók, ahol m nem túl nagy. Tegyük fel még, hogy nincs két egyforma kulcsú elem.

A dinamikus halmaz megvalósítására egy $T[0\,..\,m-1]$ tömböt használunk, melyet ilyenkor ***közvetlen címzésű táblázatnak*** fogunk hívni. A táblázat minden helye – a ***rés*** elnevezést is használjuk rájuk – megfelel az U univerzum egy kulcsának. A 11.1. ábra bemutatja a helyzetet; a k-adik rés a halmaz k kulcsú elemére mutat. Ha a halmaz nem tartalmaz k kulcsú elemet, akkor $T[k] =$ NIL.

A szótárműveletek nagyon könnyen megvalósíthatók.

KÖZVETLEN-CÍMZÉSŰ-KERESÉS(T, k)

1 **return** $T[k]$

KÖZVETLEN-CÍMZÉSŰ-BESZÚRÁS(T, x)

1 $T[kulcs[x]] \leftarrow x$

KÖZVETLEN-CÍMZÉSŰ-TÖRLÉS(T, x)

1 $T[kulcs[x]] \leftarrow$ NIL

Mindegyik művelet gyors, a végrehajtási idejük csak $O(1)$.

Bizonyos alkalmazásokban a dinamikus halmaz elemei tárolhatók magában a közvetlen címzésű táblázatban. Ilyenkor egy elem kulcsát és kísérő adatait nem egy – a táblázat megfelelő réséhez mutatóval kapcsolt – külső objektumban tároljuk, hanem magában a résben, miáltal tárterületet takaríthatunk meg. Gyakran még az sem szükséges, hogy az objektum kulcsmezőjét tároljuk, hiszen ha megvan egy elem táblázatbeli indexe, akkor megvan a kulcsa is. Ha a kulcsokat nem a résekben tároljuk, akkor valamilyen módon el kell tudnunk dönteni, hogy egy rés üres-e.

Gyakorlatok

11.1-1. Tekintsük az S dinamikus halmazt, melyet a T közvetlen címzésű, m hosszúságú táblázatban ábrázolunk. Készítsünk olyan eljárást, mely megkeresi S legnagyobb elemét. Legrosszabb esetben mekkora az eljárás végrehajtási ideje?

11.1-2. Egy ***bitvektor*** egy biteket (nullákat és egyeseket) tartalmazó, egydimenziós tömb. Egy m hosszúságú bitvektor sokkal kevesebb helyet foglal el, mint egy m mutatót tartalmazó tömb. Írjuk le, hogyan lehetne egy dinamikus halmazt bitvektorral megvalósítani, ha tudjuk, hogy az elemek mind különbözők és nem tartalmaznak kísérő adatot. A szótárműveletek $O(1)$ idő alatt hajtódjanak végre.

11.1-3. Ajánljunk módszert az olyan közvetlen címzésű táblázatok megvalósítására, melyekben a tárolandó elemek kulcsai nem feltétlenül különbözők és az elemek kísérő adatokat is tartalmazhatnak. Mindhárom szótárművelet (BESZÚR, TÖRÖL és KERES) $O(1)$ idő alatt hajtódjon végre. (Ne feledjük, hogy a TÖRÖL műveletnek nem kulcsot, hanem egy, az objektumra mutató pointert kell argumentumként megkapnia.)

11.1-4.⋆ Egy szótárt szeretnénk megvalósítani egy nagyon nagy tömbben való közvetlen címzéssel. Kezdetben a tömb elemei akármilyen értéket tartalmazhatnak, a kezdeti értékadás nem célszerű a nagy méret miatt. Vázoljuk fel egy közvetlen címzésű szótár nagyon nagy méretű tömbben való megvalósításának sémáját. A tárolt elemek mindegyikének el kell férnie $O(1)$ helyen; a KERES, BESZÚR és TÖRÖL műveletek végrehajtási idejének, továbbá az egész adatszerkezet kezdeti érték beállítási idejének $O(1)$ kell lennie. (*Útmutatás.* Használjunk egy, a szótárban aktuálisan tárolt elemek számával megegyező méretű segédvermet, s ennek segítségével állapítsuk meg, hogy a nagy tömb egy adott eleme valódi-e vagy sem.)

11.2. Hasító táblázatok

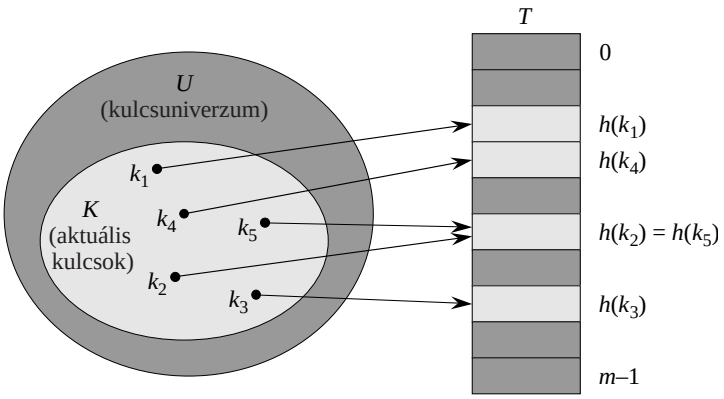
A közvetlen címzés nehézségei nyilvánvalók: ha az U univerzum nagy, akkor egy $|U|$ méretű T táblázat tárolása a ma tipikusnak tekinthető gépek memóriájában – annak korlátozott mérete miatt – nem célszerű vagy egyenesen lehetetlen. Fennáll továbbá, hogy az *aktuálisan tárolt* elemek K halmaza az U-hoz képest olyan kicsi is lehet, hogy a T által elfoglalt hely legnagyobb része kihasználatlan.

Amikor a szótárban tárolt kulcsok K halmaza sokkal kisebb a lehetséges kulcsok U univerzumánál, akkor a hasító táblázatnak jóval kevesebb memóriára van szüksége, mint a közvetlen címzésű táblázatnak. Nevezetesen a memóriaigény leszorítható $\Theta(|K|)$-ra, ugyanakkor egy elem hasító táblázatban való keresésének ideje továbbra is $O(1)$ marad. (A dolog szépséghibája, hogy ez a korlát a hasító táblázatoknál az *átlagos időre*, míg a közvetlen címzésű tábláknál a *legrosszabbra* vonatkozik.)

A közvetlen címzés esetében egy k kulcsú elem a k-adik résben tárolódik. A hasítás alkalmazása esetén ez az elem a $h(k)$ helyre kerül, vagyis egy h ***hasító függvényt*** használunk arra, hogy a rést a k kulcsból meghatározzuk. Itt h a kulcsok U univerzumát képezi le a $T[0\,..\,m-1]$ ***hasító táblázat*** réseire:

$$h : U \to \{0, 1, \ldots, m-1\}.$$

Azzal a szóhasználattal élünk, hogy a k kulcsú elem a $h(k)$ résre ***képződik le,*** illetve hogy $h(k)$ a k kulcs ***hasított értéke.*** A 11.2. ábra szemlélteti alapötletünket. A hasító függvény célja, hogy csökkentsük a szükséges tömbindexek tartományát. $|U|$ számú index helyett most csak m-re van szükség. A memóriaigény is ennek megfelelően csökken.



11.2. ábra. A h hasító függvény felhasználása a kulcsoknak egy hasító táblázat réseire való leképezésére. A k_1 és k_3 kulcsok ugyanarra a résre képződnek le, tehát ütköznek.

Üröm az örömben, hogy megtörténhet, két kulcs is ugyanarra a résre képződik le. Ezt a helyzetet ***ütközésnek*** nevezzük. Szerencsére vannak hatékony módszerek az ütközések nyomán keletkező konfliktusok feloldására.

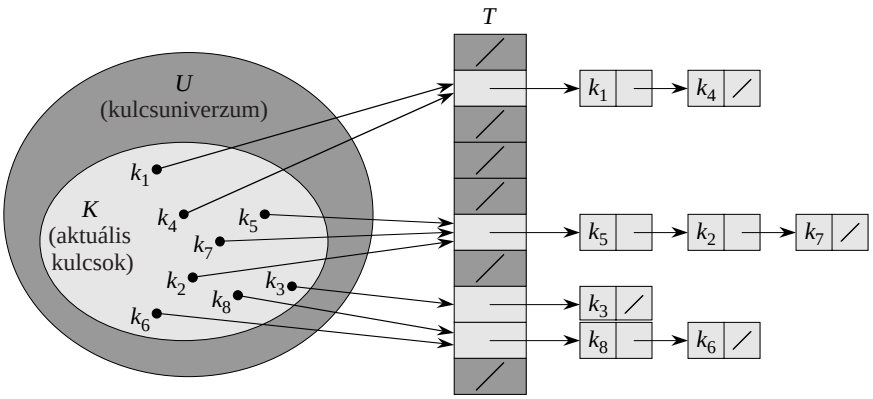
Az ideális megoldás természetesen az lenne, ha képesek lennénk az ütközések teljes kiküszöbölésére. Ezt a célt megpróbálhatjuk elérni a h hasító függvény megfelelő megválasztásával. Egy lehetséges ötlet az, hogy a h függvényt „véletlenül” választjuk, így kísérelve meg elkerülni az ütközéseket vagy legalább minimalizálni a számukat. Maga a „hasító” elnevezés is ennek a megközelítésnek a szellemét ragadja meg, a képelem címét a kulcsból „véletlen” darabolásokkal és keverésekkel állítjuk elő. (Természetesen egy hasító függvénynek determinisztikusnak kell lennie abban az értelemben, hogy egy adott k bemenetre mindig ugyanazt a $h(k)$ kimenetet kell előállítania.) Mivel $|U| > m$, ezért mindenképp kell lennie legalább két kulcsnak, amelyek ugyanarra a résre képződnek le, tehát az ütközések teljes elkerülése lehetetlen. Ezért, bár egy jól tervezett, „véletlen” hasító függvény minimalizálhatja az ütközések számát, mégiscsak szükség van az esetleg előforduló ütközések feloldásához valamilyen módszerre.

Az alfejezet további részében a legegyszerűbb ütközésfeloldási módszert mutatjuk be, az úgynevezett láncolást. A 11.4. alfejezetben alternatív módszert ismertetünk ugyanennek a feladatnak a megoldására: a nyílt címzést.

Ütközésfeloldás láncolással

A ***láncolásnál,*** ahogy ezt a 11.3. ábra mutatja, az ugyanarra a résre leképződő elemeket összefogjuk egy láncolt listába. A j-edik rés egy mutatót tartalmaz, mely a j címre leképződő elemek listájának fejére mutat. Amennyiben ilyen elemek nincsenek, akkor a j-edik rés a NIL-t tartalmazza.

A T hasító táblázat szótárműveletei könnyen megvalósíthatók a láncolt ütközésfeloldás esetében.



11.3. ábra. Ütközésfeloldás láncolással. A hasító táblázat egy $T[j]$ rése azoknak a kulcsoknak a láncolt listáját tartalmazza, melyek hasított értéke pontosan j. Például $h(k_1) = h(k_4)$ és $h(k_5) = h(k_2) = h(k_7)$.

LÁNCOLT-HASÍTÓ-BESZÚRÁS(T, x)

1 beszúrás a $T[h(kulcs[x])]$ lista elejére

LÁNCOLT-HASÍTÓ-KERESÉS(T, k)

1 a k kulcsú elem keresése a $T[h(k)]$ listában

LÁNCOLT-HASÍTÓ-TÖRLÉS

1 x törlése a $T[h(kulcs[x])]$ listából.

A beszúrás végrehajtási ideje a legrosszabb esetben $O(1)$. A beszúró eljárás részben azért gyors, mert felteszi, hogy az x beszúrandó elem nincs jelen a táblázatban; ez a feltétel szükség esetén (kiegészítő költséggel) ellenőrizhető a beszúrás előtti kereséssel. A keresés futási ideje legrosszabb esetben arányos a lista hosszával; ezt a műveletet a későbbiekben részletesen elemezzük. Egy x elem törlése $O(1)$ idő alatt elvégezhető, amennyiben a listák kétirányban vannak láncolva. (Megjegyezzük, hogy ha a LÁNCOLT-HASÍTÓ-BESZÚRÁS bemenetként az x elem helyett annak k kulcsát kapja, akkor nincs szükség keresésre. Ha a listák egy irányban láncoltak lennének, akkor a $T[h(kulcs[x])]$ listában meg kellene találnunk el őször az x-et megelőző elemet, hogy annak a *következő* elemet láncoló mutatóját az x törlése utáni helyzetnek megfelelően állíthassuk be. Ebben az esetben a törlés és a beszúrás végrehajtási ideje lényegében azonos lenne.)

A láncolásos hasítás elemzése

Milyen hatékonysággal működik a láncolásos hasítás? Különösen érdekes az a kérdés, hogy mennyi ideig tart egy adott kulcsú elem megkeresése.

Legyen T egy m rést tartalmazó hasító táblázat, melyben n elem van. Definiáljuk T-ben az α ***kitöltési tényezőt,*** mint az n/m hányadost, ami nem más, mint az egy láncba fűzött elemek átlagos száma. Elemzésünket α függvényében végezzük el, feltételezve, hogy α állandó marad. α kisebb, egyenlő és nagyobb is lehet, mint 1.

A láncolásos hasítás viselkedése a legrosszabb esetben rendkívül előnytelen: mind az n elem egy résre képződik le, egy n hosszúságú listát alkotva. Ennek következtében a keresés végrehajtási ideje a legrosszabb esetben $\Theta(n)$ plusz a hasító függvény kiszámítási ideje, vagyis nem jobb annál, mintha egyetlen láncolt listánk lenne, amely az összes elemet tartalmazza. Nyilvánvaló, hogy a hasító táblázatokat nem a legrosszabb esetben jellemző teljesítményük miatt használjuk. (A 11.5. alfejezetben leírt tökéletes hasítás azonban jó hatékonysággal rendelkezik a legrosszabb esetben – feltéve, hogy a kulcshalmaz statikus.)

A hasítás átlagos teljesítménye attól függ, hogy a h hasító függvény átlagosan mennyire egyenletesen osztja szét a tárolandó kulcsokat az m rés között. A 11.3. alfejezet az erről szóló eredményeket tárgyalja; itt feltételezzük, hogy minden elem egyforma valószínűséggel képződik le bármely résre, függetlenül attól, hogy a többiek hová kerültek. Ezt ***egyszerű egyenletes hasítási feltételnek*** nevezzük.

Ha a $T[j]$ lista hosszát n_j-vel ($j = 0, 1, \ldots, m-1$), jelöljük, akkor

$$n = n_0 + n_1 + \cdots + n_{m-1} \tag{11.1}$$

és n_j várható értéke $\mathrm{E}\left[n_j\right] = n/m = \alpha$.

Feltesszük, hogy a $h(k)$ hasított érték $O(1)$ idő alatt számítható ki, s így egy k kulcsú elem keresésének ideje lineárisan függ a $T[h(k)]$ lista $n_{h(k)}$ hosszától. Tekintsünk el a hasító függvény $O(1)$ kiszámítási idejétől, továbbá a $h(k)$ réshez való hozzáférés idejétől, s csak a keresési algoritmus által érintett elemek várható számát vizsgáljuk, vagyis a $T[h(k)]$ lista azon elemeinek számát, melyeket ellenőrzünk a k-val való egyezésük szempontjából. Két esetet vizsgálunk. Először azt, amikor a keresés sikertelen, vagyis a táblázatban nincs k kulcsú elem. Másodszor pedig azt, amikor a keresés talál k kulcsú elemet.

11.1. tétel. *Ha egy hasító táblázatban az ütközések feloldására láncolást használunk és a hasítás egyszerű egyenletes, akkor a sikertelen keresés átlagos ideje* $\Theta(1 + \alpha)$.

Bizonyítás. Az egyszerű egyenletesség feltételezése miatt bármely k kulcs, amely még nincs a táblázatban, egyforma valószínűséggel képződik le az m rés bármelyikére. Ezért egy k kulcs sikertelen keresésének átlagos ideje megegyezik annak átlagos idejével, hogy a $T[h(k)]$ listát végigkeressük. Ennek a listának az átlagos hossza $\mathrm{E}\left[n_{h(k)}\right] = \alpha$. Ezért a sikertelen keresés során megvizsgált elemek várható száma α, s így az összes szükséges idő (beszámítva a $h(k)$ kiszámítási idejét is) valóban $\Theta(1 + \alpha)$. ■

Sikeres keresés esetén a helyzet más, mivel a listákban nem azonos valószínűséggel keresünk. Annak valószínűsége, hogy egy adott listában keresünk, arányos a lista elemeinek számával. A várható keresési idő azonban továbbra is $\Theta(1 + \alpha)$.

11.2. tétel. *Ha egy hasító táblázatban a kulcsütközések feloldására láncolást használunk és a hasítás egyszerű egyenletes, akkor a sikeres keresés átlagos ideje* $\Theta(1 + \alpha)$.

Bizonyítás. Feltesszük, hogy a keresett elem egyforma valószínűséggel lehet a táblázatban tárolt n elem akármelyike. Az x elem sikeres keresése során megvizsgált elemek várható száma eggyel nagyobb, mint azoknak az elemeknek a várható száma, melyek megelőzik x-et az x-et tartalmazó listában. Az x-et megelőző elemeket x beszúrása után szúrtuk be, mivel az új elemeket a lista elejére tesszük. Ahhoz, hogy a megvizsgált elemek számának várható értékét kiszámítsuk, átlagoljuk táblázatunk n elemére az (1 + azoknak az elemeknek a várható száma, amelyeket x után adtunk x listájához) értéket. Legyen x_i a táblázatba

i-edikként beszúrt elem, és legyen $k_i = kulcs[x_i]$ ($i = 1, 2, \ldots, n$). A k_i és k_j kulcsokra definiáljuk az $x_{ij} = \mathrm{I}\left\{h(k_i) = h(k_j)\right\}$ indikátor valószínűségi változót. Mivel most a hasítás egyszerű egyenletes, $\Pr\left\{h(k_i) = h(k_j)\right\} = 1/m$, és így az 5.1. lemma szerint $\mathrm{E}\left[X_{ij}\right] = 1/m$. Ezért a sikeres keresés során vizsgált elemek számának várható értéke

$$\begin{aligned}
\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \mathrm{E}\left[X_{ij}\right]\right) && \text{(várható érték linearitása miatt)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n - i) \\
&= 1 + \frac{1}{nm}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right) \\
&= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) && \text{((A.1) egyenlőség miatt)} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}\,.
\end{aligned}$$

Így a sikeres kereséshez szükséges összes idő (beleértve most már a hasító függvény kiszámításának idejét is) $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

Milyen következtetéseket vonhatunk le az előző elemzésből? Ha a hasító táblázat réseinek száma arányos a táblázatbeli elemek számával, akkor $n = O(m)$ és így $\alpha = n/m = O(m)/m = O(1)$. Tehát a keresés átlagos ideje állandó. Mivel a beszúrás legrosszabb esetben $O(1)$ idejű (lásd a 11.2-3. gyakorlatot), továbbá a törlés legrosszabb esete is $O(1)$ idejű, ha a listák kétirányban láncoltak, ezért elmondhatjuk, hogy az összes szótárművelet megvalósítható $O(1)$ átlagos idő alatt.

Gyakorlatok

11.2-1. Tegyük fel, hogy egy h véletlen hasító függvényt használunk, mely n különböző kulcsot képez le egy m hosszúságú T táblázatba. Mi a kulcsütközések várható száma egyszerű hasítás esetén? Pontosabban szólva mi a $\{(k, l) : k \neq l \text{ és } h(k) = h(l)\}$ halmaz várható számossága?

11.2-2. Kövessük végig láncolásos ütközésfeloldás esetén az 5, 28, 19, 15, 20, 33, 12, 17, 10 kulcsok hasító táblázatba való beszúrását. Legyen a rések száma 9 és a hasító függvény $h(k) = k \bmod 9$.

11.2-3. Marley professzor azt sejtette, hogy lényeges hatékonyságjavulást érhetünk el, ha a láncolási sémát úgy módosítjuk, hogy abban a listák rendezettek legyenek. Milyen hatással van a professzor módosítása a sikeres keresés, a sikertelen keresés, a beszúrás és törlés időigényére?

11.2-4. Ajánljunk módszert arra, hogyan foglaljunk le és szabadítsunk fel memóriát az elemek számára magában a hasító táblázatban, ha a felhasználatlan réseket szabadlistába

fűzzük. Tegyük fel, hogy a rések egy jelzőbitet, továbbá vagy egy elemet és egy mutatót vagy két mutatót tartalmazhatnak. A szótárműveleteknek és a szabad-lista műveleteknek $O(1)$ várható idő alatt kell futniuk. Szükséges-e, hogy a szabad lista kétirányban legyen láncolva, vagy elegendő az egyirányú láncolás is?

11.2-5. Mutassuk meg, hogy ha U a lehetséges kulcsok univerzuma, n az adott helyzetben előforduló kulcsok száma, m a táblázat mérete és $|U| > nm$, akkor van U-nak olyan n méretű részhalmaza, melynek elemei ugyanarra a résre képződnek le, és ezért a láncolásos ütközésfeloldás legrosszabb keresési ideje $\Theta(n)$.

11.3. Hasító függvények

Ebben az alfejezetben néhány, a jó hasító függvények tervezésével kapcsolatos témával foglalkozunk, azután három előállítási sémát mutatunk be. Két séma – az osztásos és a szorzásos – lényegét tekintve heurisztikus, míg a harmadik – az univerzális – véletlenítést alkalmaz és azzal bizonyítható hatékonyságot ér el.

Hogyan működik egy jó hasító függvény?

Egy jó hasító függvény (közelítőleg) kielégíti az egyszerű egyenletességi feltételt: minden kulcs egyforma valószínűséggel képződik le az m rés bármelyikére – függetlenül attól, hová képződik le a többi kulcs. Sajnos, ezt a feltételt rendszerint nem lehet ellenőrizni, mivel ritkán ismerjük a kulcsok választásának eloszlásfüggvényét, és a kulcsok választása nem mindig független egymástól.

Néha ismerjük az eloszlást. Ha a kulcsokról például feltesszük, hogy olyan k véletlen valós számok, melyek egymástól függetlenül és egyenletesen oszlanak el a $0 \leq k < 1$ tartományban, akkor a

$$h(k) = \lfloor km \rfloor$$

hasító függvényről megmutatható, hogy kielégíti az egyszerű egyenletes hasítás feltételét.

A gyakorlatban heurisztikus módszereket használhatunk a jól működő hasító függvények készítésére. Ilyenkor hasznos, ha van valamilyen számszerű információnk az eloszlásról. Tekintsük például a fordítóprogramokban használt szimbólumtáblát, melyben a kulcsok a program azonosítóit jelölő karakterláncok. Általános jelenség, hogy egymással közeli kapcsolatban lévő szimbólumok – mint például `pt` és `pts` – jelennek meg ugyanabban a programban. Egy jó hasító függvény minimalizálja annak a lehetőségét, hogy az ilyen változatok ugyanarra a résre képződjenek le.

Jó az az általános megközelítés, hogy a hasító függvény értékét úgy állítjuk elő, hogy várhatóan független legyen az adatokban esetleg meglévő mintáktól. Például a (későbbiekben tárgyalt) „osztásos módszer” a hasító függvény értékét úgy számolja ki, hogy veszi a kulcs egy megadott prímszámra vonatkozó maradékát. Hacsak az adott prímszám nem illeszkedik a kulcseloszlásban levő mintákhoz, akkor ez a módszer jó eredményt ad.

Végezetül megjegyezzük, hogy a hasító függvények bizonyos alkalmazásai az egyszerű egyenletességnél szigorúbb feltételezést is igényelhetnek. Elvárhatjuk például, hogy a valamilyen értelemben egymáshoz „közel” lévő kulcsokhoz tartozó értékek távol legyenek egymástól. (Ez a tulajdonság különösen kívánatos például, ha a 11.4. alfejezetben definiált lineáris kipróbálás módszerét használjuk.)

A kulcsok természetes számokkal való megjelenítése

A legtöbb hasító függvény azt tételezi fel, hogy a kulcsok univerzuma a természetes számok $\mathbf{N} = \{0, 1, 2, \ldots\}$ halmaza. Ha a kulcsok nem természetes számok, akkor keresnünk kell valamilyen módot arra, hogy természetes számként jelenítsük meg őket. Például a karaktersorozat kulcsokat tekinthetjük megfelelő számrendszerben felírt egészeknek. A `pt` azonosítót megjeleníthetjük, mint a (112,116) egész számpárt, mivel a `p` betű ASCII kódja 112, a `t` betűé pedig 116. Ezt a számpárt kétjegyű, 128-as számrendszerben felírt egészként tekintve `pt` a $(112 \cdot 128) + 116 = 14\,452$ számmal azonosítható. A legtöbb alkalmazás esetében természetes módon adódik valamilyen hasonlóan egyszerű módszer arra, hogy a kulcsokat (esetleg nagyon nagy) egészekkel jelenítsük meg. Ezért a továbbiakban feltételezzük, hogy a kulcsok természetes számok.

11.3.1. Az osztásos módszer

A hasító függvények megadására szolgáló ***osztásos módszer*** esetén egy k kulcsot úgy képezünk le az m rés valamelyikére, hogy vesszük k m-mel való osztásának maradékát. Azaz a hasító függvény a következő:

$$h(k) = k \bmod m.$$

Ha például a hasító táblázat mérete $m = 12$ és a kulcs $k = 100$, akkor $h(k) = 4$. Mivel ez a módszer egyetlen osztást igényel, ezért az osztásos hasítás meglehetősen gyors.

Amikor az osztásos módszert használjuk, akkor általában elkerülünk bizonyos m értékeket. Ne legyen például m kettőhatvány, mert ha $m = 2^p$, akkor $h(k)$ éppen k p darab legalacsonyabb helyi értékű bitje. Hacsak előzetesen nem tudjuk valahonnan, hogy valószínűségeloszlásunk szerint az összes lehetséges legalacsonyabb helyi értékű p bit egyformán valószínű, akkor jobb olyan hasító függvényt választani, mely a kulcs összes bitjét ől függ.

Gyakran jó értékek m számára a kettőhatványokhoz nem túl közeli prímek. Tegyük fel például, hogy egy olyan hasító táblázatot kezelünk láncolt ütközésfeloldással, melyben körülbelül $n = 2000$ karakterlánc van, 8 bites karakterekkel. Mivel a sikertelen keresésnél átlagosan 3 elem megvizsgálását nem tartjuk soknak, ezért egy $m = 701$ méretű hasító táblázatot foglalunk le. A 701 számot azért választottuk, mert egy $\alpha = 2000/3$-hoz közeli prím, mely nincs közel semmilyen kettőhatványhoz. A k kulcsokat egészként tekintve hasító függvényünk a következő lesz:

$$h(k) = k \bmod 701.$$

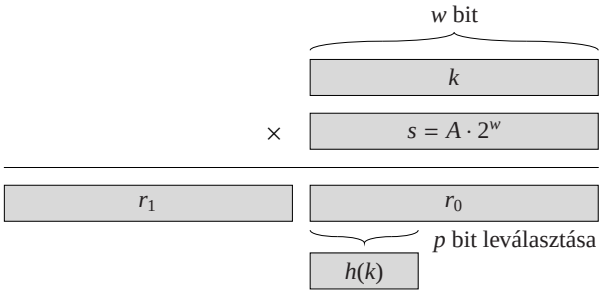
11.3.2. A szorzásos módszer

A hasító függvények előállításának ***szorzásos módszere*** két lépésből áll. Az elsőben beszorozzuk a k kulcsot valamely A $(0 < A < 1)$ állandóval és vesszük kA törtrészét. Ezután ezt az értéket beszorozzuk m-mel és vesszük az eredmény alsó egészrészét. Röviden a hasító függvény értéke a következő:

$$h(k) = \lfloor m(k\,A \bmod 1) \rfloor,$$

ahol „$k\,A \bmod 1$” a kA törtrészét, vagyis $(kA - \lfloor kA \rfloor)$-t jelenti.

A szorzásos módszer előnye, hogy itt az m értéke nem kritikus. Általában valamilyen kettőhatványnak választjuk: $m = 2^p$ valamilyen p egészre. Így – ahogy azt rövidesen látni



11.4. ábra. A hasító függvény kiszámításának szorzásos módszere. A k kulcs w bites ábrázolását beszorozzuk a w-bites $s = A \cdot 2^w$ értékkel. A szorzat w bites alsó részének p legmagasabb helyi értékű bitje adja a kívánt $h(k)$ függvényértéket.

fogjuk – a legtöbb számítógépen könnyen megvalósíthatjuk a függvényt. Legyen gépünk szóhossza w bit és tegyük fel, hogy k befér egy egyszeres szóba. A 11.4. ábra szerint el őször beszorozzuk k-t a w bites $A \cdot 2^w$ értékkel. A szorzat egy $2w$ bites $r_1 2^w + r_0$ érték, ahol r_1 a szorzat magasabb helyi értékű szava, míg r_0 az alacsonyabb helyi értékű. A kívánt p bitnyi függvényérték az r_0 szó p darab felső bitje.

Bár ez a módszer az A állandó minden értékére működik, bizonyos értékekre jobban, mint másokra. Az optimális választás a hasítandó adatok jellegét ől függ. Knuth [123] azt írja, hogy az

$$A \approx \frac{\sqrt{5} - 1}{2} = 0,6180339887\ldots \tag{11.2}$$

valószínűleg jól fog működni.

Példaként legyen $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ és $w = 32$. Knuth ajánlása szerint úgy választjuk meg A-t, hogy a $(\sqrt{5} - 1)/2$-höz legközelebbi $s/2^{32}$ alakú tört legyen, ezért $A = 2\,654\,435\,769/2^{32}$. Ekkor $k \cdot s = 327\,706\,022\,297\,664 = (76\,300 \cdot 2^{32}) + 17\,612\,864$, és így $r_1 = 76\,300$ és $r_0 = 17\,612\,864$. r_0 legfelső 14 bitje adja a $h(k) = 67$ értéket adja.

★ 11.3.3. Az univerzális hasítás

Ha egy rosszakarónk válogatja ki a hasító táblázatunkba kerül ő kulcsokat, akkor megválaszthat úgy n kulcsot, hogy azok mind ugyanarra a résre képz ődjenek le, s így a visszakeresés átlagos ideje $\Theta(n)$ legyen. Bármely rögzített hasító függvény sebezhet ő az ilyenfajta legrosszabb viselkedés szempontjából. A hasonló esetek elkerülésére az egyedüli hatásos megoldás, ha a hasító függvényt *véletlenül*, az aktuálisan tárolandó kulcsoktól *független* módon választjuk meg. Ez a megközelítés, melyet ***univerzális hasításnak*** nevezünk, jó átlagos teljesítményre vezet, függetlenül attól, milyen kulcsokat választanak rosszakaróink.

Az univerzális hasítás alapgondolata az, hogy a hasító függvényt egy gondosan megtervezett függvényhalmazból a futás során, véletlenül választjuk ki. A véletlenítés, mint ahogy azt a gyorsrendezésnél is láttuk, biztosítja, hogy ne legyen olyan el őre megadható bemenet, mely mindig a legrosszabb viselkedést váltja ki. A véletlenítés következtében ugyanis még ugyanarra a bemenetre is másképp viselkedhet az algoritmus különböz ő lefutásai során. Ez a megközelítés jó viselkedést garantál az átlagos esetben, függetlenül a bemenetül kapott kulcsoktól. A fordítóprogramokban lev ő szimbólumtáblák példájához visszatérve azt

tapasztaljuk, hogy a programozói azonosító választás így nem vezet törvényszerűen rossz teljesítményű lefutáshoz. A rossz eset most csak akkor áll elő, ha a számítógép választ olyan véletlen hasító függvényt, mely az azonosítókat nem egyenletesen osztja szét, de ennek valószínűsége kicsi és minden hasonló méretű azonosító halmaz esetében ugyanaz.

Legyen $\mathcal{H}$ hasító függvények egy véges osztálya, melyek egy adott U kulcsuniverzumot a $\{0, 1, \ldots, m-1\}$ tartományba képeznek le. Egy ilyen osztályt ***univerzálisnak*** hívunk, ha tetszőleges $k, l \in U$ különböző elemekből álló kulcspár esetében azoknak a $h \in \mathcal{H}$ hasító függvényeknek a száma, melyekre $h(k) = h(l)$ pontosan $|\mathcal{H}|/m$. Kissé átfogalmazva ez azt jelenti, hogy egy véletlenül választott $h \in \mathcal{H}$ esetén a k és l kulcsok közötti ütközés valószínűsége nem nagyobb, mint $1/m$, ami a $\{0, 1, \ldots, m-1\}$ halmazból véletlenül kiválasztott $h(x)$ és $h(y)$ egyenlőségének valószínűsége.

A következő tétel azt mutatja, hogy egy univerzális hasító függvény osztály jó átlagos viselkedéshez vezet. Emlékeztetünk arra, hogy n_i a $T[i]$ lista hossza.

11.3. tétel. *Tegyük fel, hogy h egy univerzális hasító függvény osztályból való és n kulcs m méretű T hasító táblázatba való elhelyezésére használjuk, továbbá az ütközéseket láncolással oldjuk fel. Ha a k kulcs nincs a táblázatban, akkor a k kulcsot tartalmazó lánc* $\mathrm{E}\left[n_{h(k)}\right]$ *várható hossza legfeljebb α. Ha a k kulcs a táblázatban van, akkor a k kulcsot tartalmazó lánc várható hossza legfeljebb $1 + \alpha$.*

Bizonyítás. Megjegyezzük, hogy a várható értékeket a hasító függvények választására vonatkozóan vesszük, és nem függenek a kulcsok eloszlására vonatkozó feltevésektől. A különböző k és l különböző kulcsokra definiáljuk az $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$ indikátor valószínűségi változókat. Mivel definíció szerint bármely kulcspár legfeljebb $1/m$ valószínűséggel ütközik, így $\Pr\{h(k) = h(l)\} \leq 1/m$, ezért az 5.1. lemmából következik, hogy $\mathrm{E}[X_{kl}] \leq 1/m$.

Most minden k kulcsra definiáljuk az Y_k valószínűségi változót mint azon k-tól különböző kulcsok számát, amelyek ugyanarra a résre képződnek le, mint k, azaz

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

Így

$$\begin{aligned} \mathrm{E}[Y_k] &= \mathrm{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} \mathrm{E}[X_{kl}] \qquad \text{(a várható érték linearitása miatt)} \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}\,. \end{aligned}$$

A bizonyítás hátralévő része attól függ, vajon a k kulcs benne van-e a T táblázatban.

- Ha $k \notin T$, akkor $n_{h(k)} = Y_k$ és $|\{l : l \in T \text{ és } l \neq k\}| = n$. Így $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}[Y_k] \leq n/m = \alpha$.
- Ha $k \in T$, akkor mivel a k kulcs benne van a $T[h(k)]$ listában és az Y_k értéke nem veszi figyelembe a k kulcsot, $n_{h(k)} = Y_k + 1$ és $|\{l : l \in T \text{ és } l \neq k\}| = n - 1$. Így $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}[Y_k] + 1 \leq (n-1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$. ■

A következő állítás szerint az univerzális hasítás rendelkezik a kívánt tulajdonsággal: most már nem tud egy ellenfél olyan műveletsorozatot kiválasztani, amely biztosan a futási idő legrosszabb esetére vezet. Ha a futás során jól véletlenítjük a hasító függvény megválasztását, akkor biztosíthatjuk, hogy a várható futási idő minden műveletsorozatra jó legyen.

11.4. következmény. *Az univerzális hasítás láncolásos ütközésfeloldás és m rést tartalmazó táblázat esetén tetszőleges n hosszúságú,* BESZÚR, KERES *és* TÖRÖL *műveletekből álló, $O(m)$* BESZÚR *műveletet tartalmazó sorozatot $\Theta(n)$ várható idő alatt dolgoz fel.*

Bizonyítás. Mivel a beszúrások száma $O(m)$, ezért $n = O(m)$, és így $\alpha = O(1)$. A BESZÚR és TÖRÖL műveletek konstans idő alatt hajtódnak végre, a KERES művelet várható végrehajtási ideje pedig – a 11.3. tétel szerint – $O(1)$. A várható érték linearitása miatt az egész műveletsorozat végrehajtásának várható értéke $O(n)$. ■

Hasító függvények univerzális osztályának tervezése

Meglehetősen könnyű feladat egy univerzális hasító függvény osztály megtervezése, amint azt egy kis számelmélet segítségével mindjárt be is bizonyítjuk. Ha nem ismeri eléggé a számelméletet, célszerű átnézni a 31. fejezetet.

Először válasszunk egy olyan p prímszámot, amely elég nagy ahhoz, hogy minden k kulcs benne legyen a $[0, p-1]$ intervallumban. Vezessük be a következő két jelölést: $\mathbf{Z}_p = \{0, 1, \ldots, p-1\}$ és $\mathbf{Z}_p^\star = \{1, 2, \ldots, p-1\}$. Mivel p prímszám, a 31. fejezetben megadott módszerekkel tudunk egyenleteket megoldani mod p. Mivel feltesszük, hogy a kulcsuniverzum mérete nagyobb, mint a táblázat réseinek a száma, ezért $p > m$.

Most definiáljuk a $h_{a,b}$ függvényt minden $a \in \mathbf{Z}_p^\star$ és minden $b \in \mathbf{Z}_p^\star$ elemre, egy lineáris transzformáció, majd azt követő modulo p és modulo m redukciók segítségével:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m\,. \tag{11.3}$$

Ha például $p = 17$ és $m = 6$, akkor $h_{3,4}(8) = 5$. Az ilyen hasító függvények osztálya

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^\star \text{ és } b \in \mathbf{Z}_p\}. \tag{11.4}$$

Minden $h_{a,b}$ hasító függvény leképezi $\mathbf{Z}_p$-t $\mathbf{Z}_m$-re. Ennek a függvényosztálynak megvan az a hasznos tulajdonsága, hogy a kimenet m mérete tetszőleges lehet – nem szükségképpen prím. Ezt a tulajdonságot majd a 11.5. alfejezetben fel fogjuk használni. Mivel a $(p-1)$-féleképpen, b pedig p-féleképpen választható meg, a $\mathcal{H}_{p,m}$ osztályban $p(p-1)$ hasító függvény van.

11.5. tétel. *A hasító függvényeknek a (11.3) és (11.4) egyenlőségekkel definiált $\mathcal{H}_{p,m}$ osztálya univerzális.*

Bizonyítás. Tekintsük a $\mathbf{Z}_p$-beli k és l kulcsokat, melyekre $k \neq l$. Adott $h_{a,b}$ hasító függvényre legyen

$$\begin{aligned} r &= (ak + b) \bmod p, \\ s &= (al + b) \bmod p. \end{aligned}$$

Először belátjuk, hogy $r \neq s$. Miért? Vegyük észre, hogy

$$r - s \equiv a(k - l) \bmod p.$$

Innen adódik, hogy $r \neq s$, mivel p prím, a és $(k - l)$ nem nullák modulo p, és így a 31.6. tétel szerint a szorzatuk sem nulla modulo p. Ezért bármely $\mathcal{H}_{p,m}$-beli $h_{a,b}$ függvény szerint különböző k és l bemenetek különböző r és s értékekre képződnek le modulo p; a „mod p szinten" nincs ütközés. Továbbá az (a, b) párok minden lehetséges $p(p-1)$ választása $a \neq 0$ esetén *különböző* (r, s)-t eredményez, melyre $r \neq s$, mivel adott r és s esetén meg tudjuk oldani a-ra és b-re az

$$\begin{aligned} a &= ((r - s)((k - l)^{-1} \bmod p)) \bmod p, \\ b &= (r - ak) \bmod p \end{aligned}$$

kongruenciarendszert, ahol $((k - l)^{-1} \bmod p)$ a $k - 1$ egyetlen multiplikatív inverzét jelöli, modulo p. Mivel csak $p(p - 1)$ darab olyan (r, s) pár van, amelyre $r \neq s$, ezért az $a \neq 0$ tulajdonságú (a, b) párok és az $r \neq s$ tulajdonságú (r, s) párok kölcsönösen egyértelműen megfeleltethetők egymásnak. Így ha bármely bemeneti k és l értékre egyenlő valószínűséggel választjuk az (a, b) párt a $\mathbf{Z}_p^* \times \mathbf{Z}_p$ halmazból, akkor az eredményül kapott (r, s) pár elemei egyforma valószínűséggel veszik fel a különböző modulo p értékeket.

Innen adódik, hogy annak valószínűsége, hogy a különböző k és l értékek ütközni fognak, $r \equiv s \pmod m$, ha r és s modulo p véletlenül választott értékek. Adott r esetén az s fennmaradó lehetséges $p - 1$ értékére az olyan s-ek száma, melyekre $s \neq r$ és $s \equiv r \pmod m$, legfeljebb

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p + m - 1)/m) - 1 \quad \text{((3.6) egyenlőtlenség szerint)} \\ &= (p - 1)/m. \end{aligned}$$

Annak valószínűsége, hogy a modulo m redukált s és r ütköznek, legfeljebb

$$((p - 1)/m)/(p - 1) = 1/m.$$

Ezért a különböző $k, l \in \mathbf{Z}_p$ értékekre

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m,$$

és így $\mathcal{H}_{p,m}$ valóban univerzális. ■

Gyakorlatok

11.3-1. Tegyük fel, hogy egy n hosszúságú láncolt listában keresünk, ahol az elemekben a k kulcs mellett egy $h(k)$ hasító függvény érték is van. Minden kulcs hosszú karaktersorozat. Hogyan lehetne a hasító függvény értékeket hasznosítani egykulcsú elemnek a listában való keresésénél?

11.3-2. Tegyük fel, hogy r hosszúságú, 128-as számrendszerbeli számokként tekintett karakterláncokat képezünk le m résre, osztásos hasító függvénnyel. Az m szám könnyen ábrázolható, mint egyetlen 32 bites gépi szó, de az r hosszú karakterláncok 128 alapú számként ábrázolva több szót is foglalnak. Hogyan valósítsuk meg a hasító függvényt kiszámító osztásos módszert, ha közben – a karakterlánc szavain kívül – legfeljebb rögzített számú segédszót használhatunk fel?

11.3-3. Tekintsük az osztásos módszer azon változatát, ahol $h(k) = k \bmod m$, továbbá $m = 2^p - 1$ és a k kulcs 2^p alakú számrendszerbeli számként tekintett karakterlánc. Mutassuk meg, hogy ha az x sorozatot az y sorozatból a karaktereinek permutálásával kapjuk

meg, akkor x és y ugyanarra az értékre képződik le. Adjunk meg olyan alkalmazást, ahol a hasító függvénynek ez a tulajdonsága nemkívánatos.

11.3-4. Tekintsünk egy $m = 1000$ méretű hasító táblázatot és a $h(k) = \lfloor m(k\,A \bmod 1)\rfloor$ hasító függvényt az $A = (\sqrt{5} - 1)/2$ állandó mellett. Határozzuk meg azokat a helyeket, ahová a 61, 62, 63, 64 és 65 kulcsok leképződnek.

11.3-5.⋆ A véges U halmazt a véges B halmazra képező hasítófüggvények $\mathcal{H}$ osztályát ***ϵ-univerzálisnak*** nevezzük, ha a különböző $k, l \in \mathbf{Z}_p$ értékekre

$$\Pr\{h(k) = h(l)\} \leq \epsilon,$$

ahol a valószínűség arra vonatkozik, hogy a h függvényt véletlenül választjuk a $\mathcal{H}$ osztályból. Mutassuk meg, hogy hasító függvények ϵ-univerzális osztályára

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

11.3-6.⋆ Legyen U a $\mathbf{Z}_p$-ből vett n-esek halmaza, és legyen $B = \mathbf{Z}_p$, ahol p prímszám. Ha $b \in \mathbf{Z}_p$, akkor a $h_b = U \to B$ hasító függvényt az U-ból vett $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ n elemű bemenetre a

$$h_b(\langle a_0, a_1, \ldots, a_{n-1}\rangle) = \sum_{j=0}^{n-1} a_j b^j$$

módon definiáljuk, és legyen $\mathcal{H} = \{h_b : b \in \mathbf{Z}_b\}$. Mutassuk meg, hogy $\mathcal{H}$ a 11.3-5. gyakorlatban adott definíció szerint $((n-1)/p)$-univerzális. (*Útmutatás.* Lásd a 31.4-4. gyakorlatot.)

11.4. Nyílt címzés

A ***nyílt címzés*** esetében az elemeket magában a hasító táblázatban tároljuk. A táblázat elemeinek tartalma vagy a dinamikus halmaz egy eleme, vagy pedig a NIL. Egy elem keresésénél rendre végignézzük a táblázat réseit mindaddig, amíg vagy megtaláljuk a kívánt elemet, vagy pedig világossá nem válik, hogy az nincs benne a táblázatban. A nyílt címzésnél nincsenek táblázaton kívül tárolt elemek és listák, mint a láncolásnál, ezért a nyílt címzéses hasító táblázat egy idő után „betelhet”, s ilyenkor további elemek már nem szúrhatók bele. Ebből persze adódik, hogy az α kitöltési tényező soha nem haladhatja meg az 1-et.

Bár az ugyanoda képződő elemeket – a még üres rések felhasználásával – összefűzhetnénk egy hasító táblázaton belüli listákba (lásd a 11.2-4. gyakorlatot), ezt mégsem tesszük, mert a nyílt címzés előnye éppen az, hogy elkerül mindenféle mutatót. A mutatók láncának követése helyett itt egymásután *kiszámítjuk* a megvizsgálandó rések címeit. A mutatók megtakarításából adódó többletmemória lehetővé teszi, hogy ugyanakkora területen nagyobb résszámú hasító táblázatot tudjunk tárolni, potenciálisan kevesebb ütközéssel és gyorsabb visszakereséssel.

A nyílt címzésnél a beszúrást úgy hajtjuk végre, hogy a hasító táblázat réseit egymás után megvizsgáljuk, ***kipróbáljuk*** mindaddig, amíg rátalálunk egy üresre, amibe aztán az adott kulcsot elhelyezzük. A kipróbálandó pozíciók sorrendje a rögzített $0, 1, \ldots, m-1$ helyett (mely $\Theta(n)$ keresési időre vezetne) a *beszúrandó kulcs függvénye*. Annak meghatározására, hogy melyik rést kell kipróbálni, kiterjesztjük a hasító függvény értelmezési tarto-

mányát egy új (0-tól induló) komponenssel, a kipróbálási számmal. Így a hasító függvény a következő alakú lesz:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Megköveteljük még, hogy minden k kulcsra az úgynevezett

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$$

kipróbálási sorozat a $\langle 0, 1, \ldots, m-1\rangle$ egy permutációja legyen. Ebből következik, hogy a táblázat kitöltése során előbb-utóbb minden pozíció számításba jön, mint egy új kulcs helye. A következő pszeudokódban írt eljárásban feltételezzük, hogy a T hasító táblázatban csak kulcsok vannak, kísérő adatok nélkül; a k kulcsot tartalmazó elemet azonosítjuk magával a kulccsal. A rések vagy egy kulcsot tartalmaznak, vagy pedig a NIL-t (ha a rés üres).

HASÍTÓ-BESZÚR(T, k)

```
1  i ← 0
2  repeat j ← h(k, i)
3          if T[j] = NIL
4             then T[j] ← k
5                  return
6             else i ← i + 1
7          until i = m
8  error „hasító táblázat túlcsordulás”
```

A keresési algoritmus valamely k kulcsra pontosan ugyanazokat a rés-sorozatokat próbálja ki, amelyeket a beszúrás algoritmusa is végigvizsgált, amikor a k-t beszúrta. Éppen ezért a keresési algoritmus megállhat (sikertelenül), amikor egy üres elemet talál, hiszen a k kulcsnak a beszúrás során ebbe a résbe kellett volna belekerülnie, tehát kipróbálási sorozatának későbbi tagjaiban már biztosan nem lehet benne. (Megjegyezzük, hogy ez az okfejtés feltételezi, hogy nincsenek a hasító táblázatból kitörölt kulcsok.) A HASÍTÓ-KERES eljárás bemenete a T hasító táblázat és a k kulcs, a visszatérési érték a k kulcsot tartalmazó rés j sorszáma (ha van ilyen) vagy NIL (ha nincs).

HASÍTÓ-KERES(T, k)

```
1  i ← 0
2  repeat j ← h(k, i)
3          if T[j] = k
4             then return j
5          i ← i + 1
6  until T[j] = NIL vagy i = m
7  return NIL
```

A nyílt címzéses hasító táblázatokból való törlés kissé nehezebb. Amikor töröljük az i-edik résben lévő kulcsot, nem elegendő a NIL beírásával jelezni annak ürességét, mert ha csak ezt tennénk, akkor lehetetlen lenne minden olyan kulcs visszakeresése, melynek beszúrása során az i-edik rést kipróbáltuk és foglaltnak találtuk. Egy lehetséges megoldás az, hogy a töröltség jelölésére a NIL helyett egy másik speciális értéket, a TÖRÖLT-et használjuk.

Ennek megfelelően a HASÍTÓ-BESZÚR eljárást úgy kell módosítani, hogy a TÖRÖLT értékű réseket az üresekhez hasonló módon kezelje, megengedve hogy kulcs kerüljön beléjük. A HASÍTÓ-KERES eljárást nem is kell megváltoztatni, mivel az a TÖRÖLT értékek esetében éppen megfelelően működik, azaz továbblép. Ha a TÖRÖLT értéket alkalmazzuk, akkor a keresési idő már nem lesz az α kitöltési tényező függvénye. Ezért amikor törölnünk is kell elemeket, akkor az ütközések feloldására legtöbbször a láncolást használjuk.

Elemzésünk során élünk az ***egyenletes hasítási*** feltételezéssel, tehát feltesszük, hogy minden kulcsra a $\{0, 1, \ldots, m-1\}$ elemek $m!$ permutációja közül mindegyik ugyanolyan valószínűséggel fordul elő, mint kipróbálási sorozat. Az egyenletes hasítási feltételezés a korábban definiált egyszerű egyenletes hasítási feltételezés általánosítása arra a helyzetre, amikor a hasító függvény értéke nem egyetlen szám, hanem egy egész kipróbálási sorozat. A tiszta egyenletes hasítási módszert nehéz megvalósítani, ezért a gyakorlatban alkalmas közelítéseket használnak (mint például a későbbiekben definiált dupla hasítást).

Három olyan módszer van, amit a nyílt címzésnél széles körben használnak kipróbálási sorozatok előállítására: a lineáris kipróbálás, a négyzetes kipróbálás és a dupla hasítás. Mindegyikük garantálja, hogy bármely k kulcs esetében az $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ sorozat a $\langle 0, 1, \ldots, m-1\rangle$ sorozat permutációja legyen. Ezen módszerek egyike sem teljesíti az egyenletes hasítási feltételt, hiszen egyikük sem tud m^2-nél több kipróbálási sorozatot létrehozni (az egyenletes esetben elvárt $m!$ helyett). A dupla hasító adja a legtöbb kipróbálási sorozatot, és ahogy az elvárható, ez adja a legjobb eredményt.

Lineáris kipróbálás

Ha adott egy $h' : U \to \{0, 1, \ldots, m-1\}$ közönséges hasító függvény, akkor a ***lineáris kipróbálás*** módszere az alábbi hasító függvényt használja ($i = 0, 1, \ldots, m-1$):

$$h(k, i) = (h'(k) + i) \bmod m.$$

Egy adott k kulcs esetén az első kipróbált rés $T[h'(k)]$. A következő kipróbált elem a $T[h'(k)+1]$, és így tovább egészen a $T[m-1]$ résig. Ezután ciklikusan folytatjuk a $T[0], T[1], \ldots$ résekkel, s legvégül $T[h'(k)-1]$-et próbáljuk ki. Mivel a kezdeti próbapozíció már meghatározza az egész kipróbálási sorozatot, ezért itt csak m kipróbálási sorozat fordul elő.

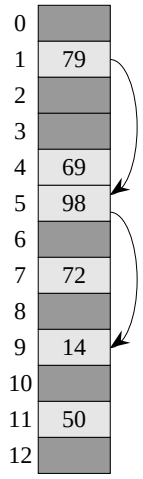
A lineáris kipróbálást könnyű megvalósítani, de kellemetlen hátránya is van, az ***elsődleges klaszterezés*** jelensége. Hosszú, elemek által teljesen kitöltött sorozatok jönnek létre, melyek megnövelik az átlagos keresési időt. A klaszterek azért jönnek létre, mert az i tele rést követő üres rés a következő hasításnál $(i+1)/m$ valószínűséggel tele lesz. Az elfoglalt résekből álló hosszú sorozatok még hosszabbak lesznek, és a keresési idő nő.

Négyzetes kipróbálás

A ***négyzetes kipróbálás*** a következő alakú hasító függvényt használja:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \tag{11.5}$$

ahol (ahogy a lineáris kipróbálásnál) h' egy kisegítő hasító függvény, c_1 és $c_2 \neq 0$ segédállandók és $i = 0, 1, \ldots, m-1$. Az először kipróbált pozíció itt is $T[h'(k)]$; a későbbi pozíciók az őket megelőzőkből a próbaszámtól négyzetesen függő mennyiséggel való eltolással



11.5. ábra. Beszúrás dupla hasítás esetén. Hasító táblázatunk legyen 13 hosszú, $h_1(k) = k \bmod 13$ és $h_2(k) = 1 + (k \bmod 11)$. Mivel $14 \equiv 1 \bmod 13$ és $14 \equiv 3 \bmod 11$, ezért a 14 kulcsot a 9. résbe szúrjuk be, mivel az 1. és 5. rés megvizsgálásakor azokat már foglaltnak találtuk.

kaphatók meg. Ez a módszer sokkal hatékonyabban működik, mint a lineáris kipróbálás. Ahhoz viszont, hogy ki tudjuk használni az egész táblázatot, megkötéseket kell tennünk c_1-re, c_2-re és m-re. A 12-3. feladat módszert ad ezeknek a paramétereknek a beállítására. Itt is igaz, hogy ha két kulcshoz ugyanaz a kezdeti próbapozíció tartozik, akkor a teljes kipróbálási sorozatuk megegyezik, hiszen $h(k_1, 0) = h(k_2, 0)$-ból már következik $h(k_1, i) = h(k_2, i)$. Ez a klaszterezés egy szelídebb változatához, az úgynevezett ***másodlagos klaszterezéshez*** vezet. Ugyanúgy mint a lineáris kipróbálásnál, a kezdeti próbahely itt is meghatározza a teljes kipróbálási sorozatot, ezért csak m különböző sorozat fordulhat elő.

Dupla hasítás

A dupla hasítás az egyik legjobb nyílt címzéses módszer, mivel az általa használt kipróbálási sorozatok a véletlen permutációk sok tulajdonságával rendelkeznek. A ***dupla hasítás*** a következő alakú hasító függvényt használja:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

ahol h_1 és h_2 kisegítő hasító függvények. A kezdeti kipróbálási pozíció most is $T[h_1(k)]$; az egymás után következő próbapozíciók pedig az előző pozíciók $h_2(k)$-val való eltolásával jönnek létre (modulo m). A lineáris, illetve négyzetes kipróbálás módszerével ellentétben itt a próbasorozatok kétféle módon is függenek a k kulcstól, mivel a kezdő pozíció, az eltolás, illetve mindkettő változhat. A 11.5. ábra példát mutat arra, hogyan kell beszúrni a dupla hasítás alkalmazása esetén.

A $h_2(k)$ értéknek relatív prímnek kell lennie a táblázat m méretéhez képest. Egy kényelmes út ennek a feltételnek a biztosítására, ha m-et kettőhatványnak választjuk, h_2-t pedig úgy, hogy mindig páratlan számot állítson elő. Egy másik lehetőség, ha m prím és h_2 mindig m-nél kisebb pozitív egészet ad eredményül. Válasszuk például m-et prímnek és legyen

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod m'), \end{aligned}$$

ahol m' egy m-nél kicsivel kisebb egész (mondjuk $m-1$ vagy $m-2$). Ha például $k = 123456$ és $m = 701$, $m' = 700$, akkor $h_1(k) = 80$ és $h_2(k) = 257$, így az első kipróbálási pozíció a 80, s minden 257. (modulo m) elemet megvizsgálunk, amíg a kulcsot meg nem találjuk vagy addig, amíg minden rést meg nem vizsgáltunk.

A dupla hasítás a lineáris és négyzetes hasítás javítása, ahol a $\Theta(m)$ kipróbálási sorozat helyett $\Theta(m^2)$-et használhatunk, mivel minden lehetséges $(h_1(k), h_2(k))$ értékpár különböző sorozathoz vezet. Az is javítás, hogy amikor változtatjuk a kulcsot, akkor a kezdeti $h_1(k)$ próbapozíció és a $h_2(k)$ eltolás egymástól függetlenül változik. Úgy tűnik, hogy a dupla hasítás hatékonysága nagyon közel van az „ideális” egyenletes hasítási séma hatékonyságához.

A nyílt címzéses hasítás elemzése

Elemzésünket, csakúgy mint a láncolás esetében, az α kitöltési tényező függvényében fogjuk elvégezni, miközben n és m tartanak a végtelenhez. Idézzük vissza, hogy ha m rést tartalmazó táblázatunkban n elemet tárolunk, akkor az elemek száma résenként átlagosan $\alpha = n/m$. Természetesen a nyílt címzésnél egy rés legfeljebb egy elemet tartalmazhat, ezért $n \leq m$, amiből $\alpha \leq 1$ következik.

Tegyük fel, hogy egyenletes hasítási módszert használunk. Az idealizált sémában bármelyik rögzített kulcs esetén a $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ kipróbálási sorozat egyforma valószínűséggel lehet a $\langle 0, 1, \ldots, m-1\rangle$ bármelyik permutációja. Tehát bármelyik lehetséges kipróbálási sorozat egyforma valószínűséggel esélyes arra, hogy egy beszúrásnál vagy keresésnél használjuk. Természetesen egy adott kulcsnak csak egyetlen, rögzített kipróbálási sorozata van; az egyforma valószínűséget itt úgy értjük, hogy a kulcsok terén vett valószínűségeloszlás és a hasító függvénynek a kulcsokra való hatása következtében lesznek egyforma valószínűségűek a lehetséges kipróbálási sorozatok.

Rátérünk most a nyílt címzéses hasítás várható próbaszámának elemzésére, élve azzal a feltevéssel, hogy hasító függvényünk egyenletes. A sikertelen keresés próbaszámának elemzésével kezdjük.

11.6. tétel. *Ha adott egy nyílt címzéses hasító táblázat az $\alpha = n/m < 1$ kitöltöttségi aránnyal, akkor a sikertelen keresés várható próbaszáma az egyenletes hasítási feltétel teljesülése esetén legfeljebb $1/(1-\alpha)$.*

Bizonyítás. A sikertelen keresésnél a legutolsó próba kivételével minden érintett rés foglalt, és nem az adott kulcsot tartalmazza, míg a legutolsóként kipróbált rés üres. Definiáljuk az X valószínűségi változót mint sikertelen keresés során végzett próbák számát, és definiáljuk az A_i $(i = 1, 2, ...)$ eseményt úgy, hogy van i-edik próba, és az foglalt résre vonatkozott. Ekkor az $\{X \geq i\}$ esemény az események $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$ metszete. Az $\Pr\{X \geq i\}$ valószínűségre az $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$ valószínűség becslésével adunk korlátot. A C.2-6. gyakorlat szerint

$$\begin{aligned} \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}. \end{aligned}$$

Mivel n elem és m rés van, $\Pr\{A_1\} = n/m$. Ha $j > 1$, akkor annak valószínűsége, hogy van j-edik próba és az foglalt résre vonatkozik – feltéve, hogy az első $j-1$ próba foglalt résre vonatkozott – $(n-j+1)/(m-j+1)$. Ez a valószínűség abból adódik, hogy a fennmaradó $(n-(j-1))$ elemet $(m-(j-1))$ eddig nem vizsgált résben keressük, és az egyenletes hasítás feltétele szerint a keresett valószínűség ezeknek a számoknak a hányadosa. Mivel $n < m$, így $(n-j)/(m-j) \leq n/m$ minden olyan j-re, amelyre $0 \leq j < m$; ezért minden olyan i-re, amelyre $1 \leq i \leq m$, fennáll

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}.\end{aligned}$$

A (C.24) egyenlőség felhasználásával korlátot adunk a próbák várható számára:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha}\,.\end{aligned}$$

■

A fenti $1 + \alpha + \alpha^2 + \alpha^3 + \cdots$ összegnek van intuitív jelentése. Egy próbát mindenképp csinálunk. Körülbelül α valószínűséggel az első próba foglalt rést talál, és ezért szükség van a második próbára. Körülbelül α^2 valószínűséggel az első két rés foglalt, ezért szükséges a harmadik próba és így tovább.

Ha α állandó, akkor a 11.6. tétel szerint a sikertelen keresés futási ideje $O(1)$. Ha például a hasító táblázat félig van kitöltve, akkor a sikertelen keresés próbáinak átlagos száma legfeljebb $1/(1-0,5) = 2$. Ha 90 százalékban telített, akkor ez az átlagos próbaszám legfeljebb $1/(1-0,9) = 10$.

A 11.6. tétel szinte közvetlenül megadja a HASÍTÓ-BESZÚR eljárás hatékonyságát.

11.7. következmény. *Egy α kitöltési tényezőjű nyílt címzéses hasító táblázatba való beszúrás várható próbaszáma az egyenletes hasítási feltétel teljesülése esetén legfeljebb* $1/(1-\alpha)$.

Bizonyítás. Egy elem csak akkor szúrható be, ha a táblázatban van még üres hely, azaz $\alpha < 1$. A beszúrás során először el kell végezni egy sikertelen keresést a kulcsra, amit követ az elemnek a megtalált első üres helyre való elhelyezése. Így a várható próbák száma valóban legfeljebb $1/(1-\alpha)$. ■

A sikeres keresés várható próbaszámának kiszámítása egy kicsit több munkát igényel.

11.8. tétel. *Legyen adott egy nyílt címzéses hasító táblázat az $\alpha < 1$ kitöltöttségi aránnyal. Tegyük fel az egyenletes hasítási feltételt és azt, hogy a táblázat minden elemét egyforma valószínűséggel keressük. Ekkor a sikeres keresés várható próbaszáma*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} .$$

Bizonyítás. Egy k kulcsot keresve ugyanazt a kipróbálási útvonalat követjük, mint amikor az azt tartalmazó elemet beszúrtuk. A 11.7. következmény szerint ha k az $(i+1)$-edik elemként került be a hasító táblázatba, akkor a próbák várható száma a keresésnél legfeljebb $1/(1-i/m) = m/(m-i)$. A hasító táblázatban levő n kulcsra átlagolva azt kapjuk, hogy a sikeres keresés átlagos próbaszáma:

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}), \end{aligned}$$

ahol $H_i = \sum_{j=1}^{i} 1/j$ az i-edik harmonikus szám (ahogy ezt az (A.7) egyenlőségben definiáltuk). Felhasználva az összeg integrállal való becslésének módszerét, amelyet az A.2 alfejezetben írtunk le, az

$$\begin{aligned} \frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} 1/k \\ &\le \frac{1}{\alpha} \int_{m-n}^{n} (1/x)dx \quad \text{(A.12. egyenlőség miatt)} \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

felső becslést kapjuk a sikeres keresés várható próbaszámára. ■

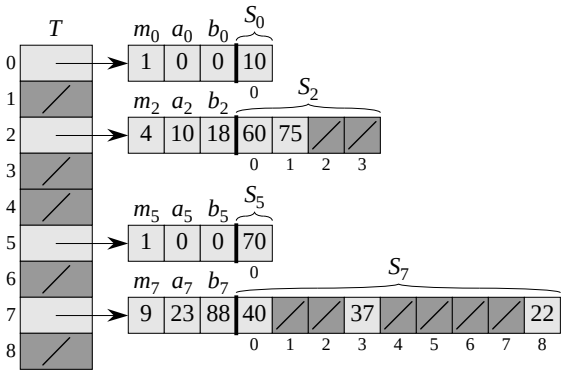
A tételből azt kapjuk, hogy a sikeres keresések várható próbaszáma egy félig kitöltött hasító táblázat esetén legfeljebb 1,387, míg 90 százalékos telítettségnél legfeljebb 2,559.

Gyakorlatok

11.4-1. Adott egy nyílt címzéses, $m = 11$ hosszúságú (üres) hasító táblázat a $h'(k) = k \bmod m$ elsődleges hasító függvénnyel, s tekintsük a beszúrandó 10, 22, 31, 4, 15, 28, 17, 88, 59 kulcsokat. Szemléltessük a beszúrás eredményét akkor, ha a lineáris kipróbálást, a $c_1 = 1$ és $c_2 = 3$ állandókat használó négyzetes kipróbálást, illetve a $h_2(k) = 1 + (k \bmod (m-1))$ másodlagos hasító függvényű dupla hasítási módszert alkalmazzuk.

11.4-2. Készítsük el a korábban ismertetett HASÍTÓ-TÖRÖL eljárás pszeudokódját, s módosítsuk a HASÍTÓ-BESZÚR és HASÍTÓ-KERES algoritmusokat is, beépítve a speciális TÖRÖLT érték kezelését.

11.4-3.⋆ Tegyük fel, hogy az ütközések feloldására dupla hasítási módszert használunk, a $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$ hasító függvénnyel. Mutassuk meg, hogy a $\langle h(k,0), h(k,1),$



11.6. ábra. Tökéletes hasítás alkalmazása a $K = \{10, 22, 37, 40, 60, 70, 75\}$ halmaz tárolására. A külső hasító függvény $h(k) = ((ak + b) \bmod p) \bmod m$, ahol $a = 3$, $b = 42$, $p = 101$ és $m = 9$. Például $h(75) = 2$, ezért a 75-ös kulcs a T táblázat 2-es résére képződik le. Az S_j másodlagos hasító tábla az összes olyan kulcsot tartalmazza, amelyek a j résre képződnek le. Az S_j hasító tábla mérete m_j, és a kapcsolódó hasító függvény $h_j(k) = ((a_j k + b_j)) \bmod p \bmod m_j$. Mivel $h_2 = 1$, a 75-ös kulcsot az S_2 másodlagos hasító tábla 1-es résében tároljuk. Egyik hasító táblában sincs ütközés, és így a keresés legrosszabb esetben is csak konstans időt vesz igénybe.

$\ldots, h(k, m-1)\rangle$ kipróbálási sorozata a $\langle 0, 1, \ldots, m-1\rangle$ sorozatnak akkor és csak akkor lesz permutációja, ha $h_2(k)$ m-hez képest relatív prím. (*Útmutatás.* Lásd a 31. fejezetet.)

11.4-4. Adott egy nyílt címzéses hasító táblázat az egyenletes hasítási feltétellel és az $\alpha = 1/2$ kitöltöttségi aránnyal. Adjunk felső korlátot a sikertelen, illetve a sikeres keresés várható próbaszámára. Mit kapunk a 3/4 és 7/8 kitöltési tényezők esetén?

11.4-5.$\star$ Adott egy nyílt címzéses hasító táblázat az egyenletes hasítási feltétellel és α kitöltési tényezővel. Keressük meg azt a nem nulla α értéket, melyre a sikertelen keresés várható próbaszáma kétszerese a sikeres keresés várható próbaszámának. Használjuk a sikeres keresés várható próbaszámára vonatkozó $(1/\alpha)\ln(1/(1-\alpha))$ becslést.

$\star$ 11.5. Tökéletes hasítás

Bár a hasítást rendszerint kitűnő várható teljesítménye miatt alkalmazzák, a hasítás azért is alkalmazható, hogy kitűnő teljesítményt kapjunk a *legrosszabb esetben:* amikor a kulcshalmaz ***statikus,*** azaz ha a kulcsokat már beírtuk a táblázatba, többé nem változnak. Bizonyos alkalmazásokban természetes módon statikus a kulcshalmaz: tekintsük egy adott programozási nyelv fenntartott szavainak halmazát, vagy egy CD-ROM fájlneveinek halmazát. Egy hasítási módszert ***tökéletes hasításnak*** nevezünk, ha az egy kereséshez szükséges memóriaolvasások száma $O(1)$.

A tökéletes hasítás létrehozásának alapötlete egyszerű. Kétszintű hasítást alkalmazunk úgy, hogy mindkét szinten univerzális a hasítás. A 11.6. ábra szemlélteti ezt a megközelítést.

Az első szint lényegében ugyanaz, mint a láncolásos hasításnál: n kulcsot hasítunk m résbe úgy, hogy gondosan megválasztjuk az univerzális hasítófüggvények osztályát.

Ahelyett azonban, hogy a j-edik résbe kerülő kulcsokat láncolnánk, egy kis S_j ***másodlagos hasító táblát*** alkalmazunk, amelyhez egy h_j kiegészítő hasító tábla tartozik. A h_j

hasító függvény gondos megválasztásával biztosítjuk, hogy a második szinten ne legyen ütközés.

Annak biztosításához, hogy ne legyen ütközés a második szinten, szükség van arra, hogy az S_j hasító tábla m_j mérete a j-edik résre képződő kulcsok számának négyzetével legyen egyenlő. Úgy tűnhet, hogy mivel m_j négyzetesen függ n_j-től, a teljes memóriaigény nagyon nagy. Megmutatjuk, hogy ha jól választjuk meg az első szint hasítófüggvényét, akkor a várható teljes memóriaigény még mindig csak $O(1)$.

Olyan hasító függvényeket használunk, amelyeket a 11.3.3. pontban szereplő univerzális osztályokból választunk. Az első szint hasító függvényét a $\mathcal{H}_{p,m}$ osztályból választjuk, ahol p – a 11.3.3. ponthoz hasonlóan – minden kulcsnál nagyobb prímszám. A j résre képződő kulcsokat újra hasítjuk az m_j méretű S_j másodlagos hasító táblázatba, a $\mathcal{H}_{p,m_j}$ osztályból.[1]

Két lépésben oldjuk meg a hasítást. Először meghatározzuk, hogyan biztosítható, hogy a második szinten ne legyen ütközés. Azután megmutatjuk, hogy az elsődleges hasító táblákhoz és a másodlagos hasító táblákhoz összesen felhasznált memória nagysága $O(n)$.

11.9. tétel. *Ha egy $m = n^2$ méretű hasítótáblában n kulcsot tárolunk, hasító függvények univerzális osztályából véletlenül választott h hasító függvényt alkalmazva, akkor az ütközés valószínűsége kisebb, mint 1/2.*

Bizonyítás. $\binom{n}{m}$ olyan kulcspár van, amelyek ütközést okozhatnak; ha h-t véletlenül választjuk hasító függvények egy $\mathcal{H}$ osztályából, akkor minden pár $1/m$ valószínűséggel ütközik. Legyen X valószínűségi változó, amelyik megadja az ütközések számát. Ha $n = m^2$, akkor az ütközések számának várható értéke

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2. \end{aligned}$$

(Megjegyezzük, hogy ez az elemzés hasonló a születésnap paradoxonnak az 5.4.1. pontban elvégzett elemzéséhez.) A (C.29) Markov-egyenlőtlenségnek, amely szerint $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, az alkalmazása a $t = 1$ választással teljessé teszi a bizonyítást. ■

A 11.9. tételben leírt helyzetben, ahol $m = n^2$, a $\mathcal{H}$ osztályból véletlenül választott h hasító függvény valószínűbb, mint az, hogy lesz ütközés. Ezért ha adott az n hasítandó kulcsot tartalmazó K halmaz (ne felejtsük el, hogy K statikus), néhány próbálkozással könnyű találni olyan h hasító függvényt, amelyik ütközésmentes.

Ha azonban n nagy, a hasító tábla $m = n^2$ mérete nagy. Ezért kétszintes megközelítést alkalmazunk, és a 11.9. tételt csak az azonos résre leképzett kulcsok hasítására alkalmazzuk. A külső vagy első szinten a h hasító függvényt arra használjuk, hogy a kulcsokat $m = n$ résre képezzék le. Ekkor ha n_j kulcsot képezünk le a j-edik résre, akkor egy $m_j = n_j^2$ méretű S_j másodlagos hasító táblázatot alkalmazunk, hogy ütközésmentes, konstans idejű keresést kapjunk.

[1] Ha $n_j = m_j = 1$, akkor a j réshez nincs szükségünk hasító függvényre; ha a $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ hasító függvényt használjuk az ilyen résre, akkor éppen az $a = b = 0$ értékeket alkalmazzuk.

Ezután nézzük azt, hogyan érjük el, hogy a teljes memóriaigény csak $O(n)$ legyen. Mivel a j-edik másodlagos hasító tábla m_j mérete négyzetesen nő a tárolandó kulcsok n_j számával, fennáll a kockázat, hogy a memóriaigény túl nagy lesz.

Ha az első szintű táblázat mérete $m = n$, akkor az elsődleges hasító táblázat, az m_j méretű másodlagos táblázatok, valamint a 11.3.3. pontban szereplő $\mathcal{H}_{p,m}$ osztályból vett másodlagos hasító függvényeket meghatározó a_j és b_j paraméterek (kivéve, amikor $n_j = 1$ és az $a = b = 0$ értékeket használjuk) együttes memóriaigénye $O(n)$. A következő tétel és következmény korlátot tartalmaznak az összes másodlagos hasító táblázat várható együttes méretére. Egy további következmény annak valószínűségére ad korlátot, hogy a másodlagos táblázatok együttes mérete szuperlineáris.

11.10. tétel. *Ha egy $m = n$ méretű hasítótáblában n kulcsot tárolunk, hasító függvények univerzális osztályából véletlenül választott h hasító függvényt alkalmazva, akkor*

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,$$

ahol n_j a j-edik résre leképződő kulcsok száma.

Bizonyítás. A következő azonossággal kezdünk, amely minden a nemnegatív egészre teljesül:

$$a^2 = a + 2\binom{a}{2}.$$

Ekkor

$$\begin{aligned}
\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{((11.6) egyenlőség)}\\
&= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(várható érték linearitása)}\\
&= \mathrm{E}[n] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{((11.1) egyenlőség)}\\
&= n + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(mivel } n \text{ nem valószínűségi változó).}
\end{aligned}$$

A $\sum_{j=0}^{m-1}\binom{n_i}{2}$ összeg kiértékeléséhez vegyük észre, hogy ez éppen az ütközések száma. Az univerzális hasítás tulajdonságai szerint ennek az összegnek a várható értéke legfeljebb

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

mivel $m = n$. Így

$$\begin{aligned}
\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &\le n + 2\,\frac{n-1}{2}\\
&= 2n - 1\\
&< 2n.
\end{aligned}$$

■

11.11. következmény. *Ha egy $m = n$ méretű hasítótáblában n kulcsot tárolunk, hasító függvények univerzális osztályából véletlenül választott h hasító függvényt alkalmazva, és minden másodlagos hasító táblázat méretét az $m_j = n_j^2$ (ahol $j = 1, 2, \ldots n$) értékre állítjuk be, akkor tökéletes hasítás esetén minden hasítótábla várható memóriaigénye kisebb, mint $2n$.*

Bizonyítás. Ha $j = 0,\ 1, \ldots, m-1$, akkor $m_j = n_j^2$, ezért a 11.10. tétel szerint

$$\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n\,, \tag{11.6}$$

ami teljessé teszi a bizonyítást. ■

11.12. következmény. *Ha egy $m = n$ méretű hasítótáblában n kulcsot tárolunk, hasító függvények univerzális osztályából véletlenül választott h hasító függvényt alkalmazva, és minden másodlagos hasító táblázat méretét az $m_j = n_j^2$ (ahol $j = 1, 2, \ldots n$) értékre állítjuk be, akkor annak valószínűsége, hogy a másodlagos hasító táblákhoz összesen felhasznált memória nagysága $4n$-nél nagyobb, (1/2)-nél kisebb.*

Bizonyítás. Ismételten alkalmazzuk a (C.29) Markov-egyenlőtlenséget, amely szerint $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, ezúttal a (11.7) egyenlőtlenségre, a $X = \sum_{j=0}^{m-1} m_j$ és $t = 4n$ választással:

$$\begin{aligned}
\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} &\leq \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\
&< \frac{2n}{4n} \\
&= \frac{1}{2}.
\end{aligned}$$ ■

A 11.12. következményből látjuk, hogy néhány, az univerzális osztályból véletlenül kiválasztott hasító függvényt tesztelve gyorsan találunk olyat, amelyik csak elég kis tárterületet igényel.

Gyakorlatok

11.5-1.⋆ Tegyük fel, hogy n kulcsot szúrunk be egy m méretű hasító táblába, nyílt címzést és egyenletes hasítást alkalmazva. Legyen $p(n, m)$ annak a valószínűsége, hogy nem lesz ütközés. Mutassuk meg, hogy $p(n, m) \leq e^{-n(n-1)/2m}$. (*Útmutatás.* Lásd a (3.11) egyenlőséget.) Használjuk ki, hogy amikor n túllépi a $\sqrt{m}$ értéket, akkor az ütközések elkerülésének valószínűsége gyorsan tart nullához.

Feladatok

11-1. A leghosszabb próba becslése

Egy m méretű hasító táblázatban n elemet tárolunk, ahol $n \leq m/2$. Nyílt címzést használunk az ütközések feloldására.

a. Mutassuk meg, hogy az egyenletes hasítási feltétel teljesülése esetén minden $i = 1, 2, \ldots n$-re annak valószínűsége, hogy az i-edik beszúráshoz k-nál határozottan több próbára van szükség, legfeljebb 2^{-k}.

b. Mutassuk meg, hogy minden $i = 1, 2, \ldots n$-re annak valószínűsége, hogy az i-edik beszúráshoz $2 \lg n$-nél több próbára van szükség, legfeljebb $1/n^2$.

Jelölje az X_i azt a valószínűségi változót, mely megadja az i-edik beszúráshoz szükséges próbák számát. A (b) részben megmutattuk, hogy $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Jelölje az $X = \max_{1 \leq i \leq n} X_i$ azt a valószínűségi változót, mely megadja az n beszúráshoz szükséges próbák maximális számát.

c. Mutassuk meg, hogy $\Pr\{X > 2 \lg n\} \leq 1/n$.

d. Mutassuk meg, hogy a leghosszabb próbasorozat várható hosszára $\mathrm{E}[X] = O(\lg n)$.

11-2. A leghosszabb lánc becslése

Tegyük fel, hogy van egy n méretű hasító táblázatunk, ahol az ütközések feloldására láncolást használunk. Tegyük még fel, hogy a táblázatban pontosan n kulcs van elhelyezve. Minden kulcs egyforma valószínűséggel képződik le bármelyik résre. Legyen M az egy

réshez tartozó kulcsok maximális száma az összes elem beszúrása után. Feladatunk, hogy $\mathrm{E}[M]$-re, az M várható értékére belássuk az $O(\lg n / \lg \lg n)$ felső becslést.

a. Bizonyítsuk be, hogy annak a valószínűsége, hogy valamely kiválasztott résre pontosan k darab kulcs képződik le, a következő:

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

b. Legyen P_k annak valószínűsége, hogy $M = k$, vagyis hogy a legtöbb kulcsot tartalmazó lánc hossza egyenlő k-val. Mutassuk meg, hogy $P_k \leq nQ_k$.

c. A Stirling-formula, azaz a (2.11) képlet segítségével mutassuk meg, hogy $Q_k < e^k / k^k$.

d. Mutassuk meg, hogy létezik olyan $c > 1$ konstans, hogy a $k_0 = c \lg n / \lg \lg n$ indexre $Q_{k_0} < 1/n^3$. Vezessük le ebből, hogy minden a $k \geq k_0$-ra $P_k < 1/n^2$ is igaz.

e. Bizonyítsuk be, hogy

$$\mathrm{E}[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Vezessük le ebből, hogy $\mathrm{E}[M] = O((\lg n)/(\lg \lg n))$.

11-3. Négyzetes kipróbálás

Tegyük fel, hogy adott egy k kulcs, amit egy $0, 1, \ldots, m-1$ pozíciókkal rendelkező hasító táblázatban keresünk, s tegyük fel még, hogy adott egy h hasító függvény is, mely a kulcsuniverzumot a $\{0, 1, \ldots, m-1\}$ halmazba képzi. A keresés sémája az alábbi:

1. Számítsuk ki az $i \leftarrow h(k)$ értéket, és állítsuk be j-t $j \leftarrow 0$.
2. Próbáljuk ki az adott k kulccsal az i-edik pozíciót. Ha megtaláltuk, illetve ez a pozíció üres, fejezzük be a keresést.

3. Legyen $j \leftarrow (j + 1) \bmod m$ és $i \leftarrow (i + j) \bmod m$, azután vissza a 2. lépésre.

Tételezzük fel, hogy m kettőhatvány.

a. Mutassuk meg, hogy ez a séma speciális esete az általános „négyzetes kipróbálásnak” és állapítsuk meg a (11.5) képletnek megfelelő c_1 és c_2 állandókat.

b. Bizonyítsuk be, hogy ez az algoritmus a legrosszabb esetben a táblázat minden pozícióját megvizsgálja.

11-4. k-univerzális hasítás és hitelesítés

Legyen $\mathcal{H} = \{h\}$ hasító függvények olyan osztálya, melyben minden h az U kulcsuniverzumot képezi le a $\{0, 1, \ldots, m-1\}$ halmazra. Azt mondjuk, hogy $\mathcal{H}$ ***k-univerzális,*** ha bármely rögzített, k hosszúságú, különböző kulcsokból álló $\langle x_1, x_2, \ldots, x_k \rangle$ sorozatra és bármely $\mathcal{H}$-ból véletlenül választott h függvényre a $\langle h(x_1), h(x_2), \ldots, h(x_k) \rangle$ sorozat egyenletes valószínűséggel veszi fel értékeit a $\{0, 1 \ldots, m-1\}$ halmaz feletti összesen m^k darab k hosszúságú sorozat közül.

a. Mutassuk meg, hogy ha $\mathcal{H}$ 2-univerzális, akkor univerzális is.

b. Tegyük fel, hogy az U univerzum a $\mathbf{Z} * p = \{0, 1, \ldots, p-1\}$ halmazból vett n-esek halmaza, ahol p prím. Tekintsük az univerzum egy elemét: $x \in \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. Minden $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ n-esre definiáljuk a következő hasító függvényt:

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p.$$

Legyen $\mathcal{H} = \{h_a\}$. Mutassuk meg, hogy $\mathcal{H}$ univerzális, de nem 2-univerzális.

c. Tegyük fel, hogy a $\mathcal{H}$ függvényt a (b) résztől kissé eltérő módon definiáljuk: minden $a \in U$ és minden $b \in \mathbf{Z}_p$ elemre legyen

$$h'_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

és legyen $\mathcal{H}' = \{h_{a,b}\}$. Mutassuk meg, hogy $\mathcal{H}'$ univerzális. (*Útmutatás.*) Tekintsünk rögzített $x \in U$ és $y \in U$ kulcsokat, melyekre bizonyos i-re $x_i \neq y_i$. Hogyan változik $h'_{a,b}(x)$ és $h'_{a,b}(y)$ értéke, ha a_i és b végigfut a $\mathbf{Z}_p$ halmazon?

d. Tegyük fel, hogy Bob és Aliz titokban megegyeznek, hogy a 2-univerzális hasító függvények $\mathcal{H}$ osztályából a $h_{a,b}$ függvényt választják. Később Aliz küld egy m üzenetet az interneten Bobnak, ahol $m \in U$. Aliz hitelesíti ezt az üzenetet azzal, hogy küld egy $t = h_{a,b}(m)$ hitelesítő címkét, és Bob ellenőrzi, hogy a kapott (m, t) pár kielégíti a $t = h_{a,b}(m)$ egyenlőséget. Tegyük fel, hogy egy ellenfél útközben elfogja az (m, t)-t, és megpróbálja Bobot félrevezetni azzal, hogy helyettük (m', t')-t küld neki. Mutassuk meg, hogy annak valószínűsége, hogy az ellenfél félre tudja vezetni Bobot, hogy fogadja el (m', t')-t, legfeljebb p attól függetlenül, mekkora számítási kapacitása van az ellenfélnek – még akkor is, ha az ellenfél ismeri a felhasznált hasító függvények $\mathcal{H}$ családját.

Megjegyzések a fejezethez

Knuth [185] és Gonnet [126] kitűnő hivatkozások a hasítást alkalmazó algoritmusok területén. Knuth H. P. Luhn-nak (1953) tulajdonítja a hasító táblázatok felfedezését, valamint az ütközésfeloldás láncolásos módszerét is. Körülbelül ebből az időből, G. M. Amdahltól ered a nyílt címzés ötlete.

A hasítófüggvények univerzális osztályának a fogalmát Carter és Wegman [52] vezették be 1979-ben.

Fredman, Komlós és Szemerédi [96] javasolták a 11.5. alfejezetben bemutatott tökéletes hasítási sémát. Módszerüket Dietzfelbinger és társai [73] kiterjesztették dinamikus halmazokra úgy, hogy a beszúrások és törlések amortizált várható végrehajtási ideje $O(1)$.

12. Bináris keresőfák

A keresőfák olyan adatszerkezetek, amelyekre a dinamikus halmazok számos műveletét értelmezzük, többek között a KERES, MINIMUM, MAXIMUM, ELŐZŐ, KÖVETKEZŐ, BESZÚR és TÖRÖL műveleteket. Ezért a keresőfák mind szótárként, mind elsőbbségi sorként használhatók.

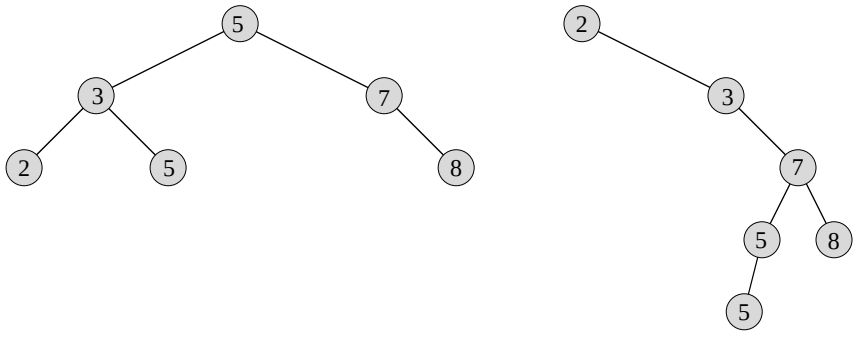
A bináris keresőfák alapműveletei a fa magasságával arányos végrehajtási idejűek. Egy n csúcsú teljes bináris keresőfa esetén ezek a műveletek a legrosszabb esetben $\Theta(\lg n)$ idő alatt hajtódnak végre. Ha viszont a fa egy n csúcsú lánc, akkor ugyanezek a műveletek a legrosszabb esetben $\Theta(n)$ végrehajtási időt igényelnek. A 12.4. alfejezetben látni fogjuk, hogy egy véletlen építésű bináris keresőfa magassága $O(\lg n)$, ezért ebben az esetben a dinamikus halmaz alapműveleteinek időigénye $\Theta(\lg n)$ átlagosan.

A gyakorlatban nem mindig garantálható a bináris keresőfák felépítésének véletlenszerűsége, de vannak a bináris keresőfáknak olyan változatai is, melyeknél biztosítható, hogy az alapműveletek hatékonysága a legrosszabb esetben is elég jó legyen. A 13. fejezetben bemutatunk egy ilyen változatot, a piros-fekete fát, melynek magassága $O(\lg n)$. A 18. fejezetben bevezetjük a B-fákat, melyek különösen jól használhatók véletlen elérésű, másodlagos (lemez) tárolókon kezelt adatbázisok esetén.

A bináris keresőfák alapvető tulajdonságainak tárgyalása után a következő alfejezetek bemutatják, hogyan lehet egy bináris keresőfát úgy bejárni, hogy rendezetten írjuk ki a benne levő értékeket, hogyan kereshetünk meg bennük egy konkrét értéket, hogyan találhatjuk meg a legkisebb vagy legnagyobb elemet, hogyan találhatjuk meg egy elem megelőzőjét vagy rákövetkezőjét, és hogyan tudunk egy bináris keresőfába beszúrni, illetve belőle törölni. A fák alapvető matematikai tulajdonságait a B függelék tartalmazza.

12.1. Mi a bináris keresőfa?

A bináris keresőfák, ahogy azt a nevük is sugallja, bináris faként szervezett objektumok. A 12.1. ábrán két példát is láthatunk rájuk. A keresőfákat láncolt struktúraként ábrázolhatjuk, ahol minden csúcs egy önálló objektum. Minden csúcs a *kulcs* mező és a hozzákapcsolódó adatok mellett tartalmaz egy *bal*, egy *jobb* és egy *szülő* nevű mezőt is, amelyek rendre a csúcs bal oldali és jobb oldali gyerekére, illetve szülőjére mutatnak. Ha hiányzik valamelyik gyerek vagy a szülő, akkor a megfelelő mező a NIL értéket tartalmazza. A gyökér az egyedüli olyan csúcs a fában, melynek szülő mutatója NIL.



12.1. ábra. Bináris keresőfák. Bármely x csúcs esetén annak bal oldali részfája legfeljebb $kulcs[x]$, míg jobb oldali részfája legalább $kulcs[x]$ nagyságú kulcsokat tartalmaz. Ugyanazt a kulcshalmazt különböző bináris keresőfákkal is ábrázolhatjuk. A legtöbb keresőfa művelet végrehajtási ideje a legrosszabb esetben a fa magasságával arányos. (**a**) Egy 6 csúcsú, 2 magasságú keresőfa. (**b**) Egy kevésbé hatékony, 4 magasságú bináris keresőfa, ugyanazokkal a kulcsokkal.

Egy bináris keresőfában levő kulcsok mindig úgy helyezkednek el, hogy kielégítsék az úgynevezett ***bináris-keresőfa tulajdonságot:***

> Legyen x az adott bináris keresőfa egy tetszőleges csúcsa. Ha y az x bal oldali részfájának egy csúcsa, akkor $kulcs[y] \leq kulcs[x]$. Ha y az x jobb oldalára esik, akkor $kulcs[x] \leq kulcs[y]$.

Így a 12.1(a) ábrán a gyökér kulcsa 5, a bal oldalán lévő 2, 3, 5 kulcsok nem nagyobbak 5-nél, míg a jobb oldali 7 és 8 kulcsok nem kisebbek 5-nél. Ez a tulajdonság a fa összes csúcsára teljesül. Például a 12.1(a) ábrán a 3-as kulcs nem kisebb a bal oldalán levő 2-es kulcsnál, és nem nagyobb a jobb oldalán lévő 5-ös kulcsnál.

A bináris-keresőfa tulajdonság lehetővé teszi, hogy egy bináris keresőfa kulcsait egy egyszerű rekurzív algoritmussal, az úgynevezett ***inorder bejárással*** rendezett sorrendben írhassuk ki. Az algoritmus neve onnan származik, hogy a fa gyökerében levő kulcsot a bal oldali részfájában lévő értékek után és a jobb oldali részfájában levő értékek előtt – azok között – írjuk ki. (Hasonlóan ehhez a ***preorder bejárás*** esetében a gyökér kulcsát a részfáinak kulcsai előtt, míg a ***posztorder bejárás*** esetében azok után írjuk ki.) Ahhoz, hogy a most következő algoritmust egy T bináris keresőfa összes értékének kiírására használhassuk, az INORDER-FABEJÁRÁS($gyökér[T]$) hívást kell leírnunk.

```
INORDER-FABEJÁRÁS(x)

1  if x ≠ NIL
2    then INORDER-FABEJÁRÁS(bal[x])
3         print kulcs[x]
4         INORDER-FABEJÁRÁS(jobb[x])
```

Példaként tekintsük a 12.1. ábrán szereplő két bináris keresőfát. Mindkettőnek az inorder bejárása a 2, 3, 5, 5, 7, 8 sorrendben írja ki a kulcsokat. Az algoritmus helyessége a bináris-keresőfa tulajdonságból indukcióval könnyen adódik.

Egy n csúcsú bináris keresőfa bejárása $\Theta(n)$ ideig tart, mivel a kezdőhívás után az eljárás a fa minden csúcspontja esetében pontosan kétszer (rekurzívan) meghívja önmagát, egyszer a bal oldali részfára, egyszer pedig a jobb oldali részfára. A következő tétel formálisan is bizonyítja, hogy az inorder fabejárás lineáris végrehajtási idejű.

12.1. tétel. *Ha x egy n csúcspontú részfa gyökere, akkor az* INORDER-FABEJÁRÁS(x) *hívás* $\Theta(n)$ *ideig tart.*

Bizonyítás. Jelölje $T(n)$ az INORDER-FABEJÁRÁS végrehajtási idejét egy n csúcspontot tartalmazó részfára történő meghívása esetén. Az INORDER-FABEJÁRÁS egy rövid időt fordít egy üres fa feldolgozására (teszteli az $x \neq$ NIL feltételt), így $T(0) = c$, ahol c egy pozitív konstans.

Ha $n > 0$, akkor az INORDER-FABEJÁRÁS egy olyan x csúcsra hívódik meg, amelynek bal részfája k és jobb részfája $n - k - 1$ csúcsot tartalmaz. Az INORDER-FABEJÁRÁS(x) végrehajtási ideje $T(n) = T(k) + T(n - k - 1) + d$, ahol d egy pozitív konstans, amely az INORDER-FABEJÁRÁS(x) válaszideje, nevezetesen a rekurzív hívásokra fordított idő.

A helyettesítő módszert alkalmazva megmutatjuk, hogy $T(n) = (c + d)n + c$, amely bizonyítja, hogy $T(n) = \Theta(n)$. Ha $n = 0$, akkor $(c + d) * 0 + c = c = T(0)$. Ha $n > 0$, akkor

$$\begin{aligned} T(n) &= T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c, \end{aligned}$$

mellyel a bizonyítás teljes. ■

Gyakorlatok

12.1-1. Rajzoljunk 2, 3, 4, 5 és 6 magasságú bináris keresőfákat, melyek az $\{1, 4, 5, 10, 16, 17, 21\}$ halmazban levő kulcsokat tartalmazzák.

12.1-2. Mi a különbség a bináris-keresőfa tulajdonság és a minimumkupac tulajdonság között? (Lásd 6.1. alfejezet.) Használható-e a minimumkupac tulajdonság arra, hogy egy n csúcsú bináris fában levő kulcsokat $O(n)$ idő alatt rendezetten kiírjuk? Ha igen, hogyan; ha nem, miért nem?

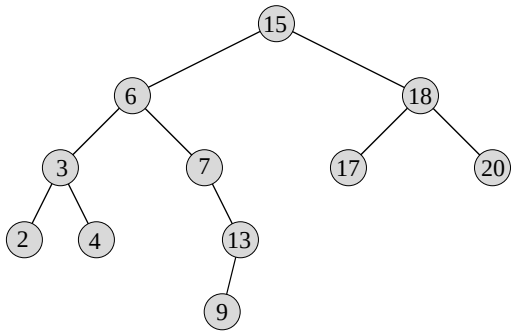
12.1-3. Adjunk nemrekurzív algoritmust az inorder bejárásra. (*Útmutatás.* Az egyszerű megoldás egy vermet használ segédmemóriaként, a bonyolultabb, de elegáns megoldás nem használ vermet, hanem feltételezi, hogy két mutató egyenlősége eldönthető.)

12.1-4. Adjunk a preorder, illetve a posztorder fabejárásra rekurzív algoritmust, amely egy n csúcsú fát $\Theta(n)$ idő alatt jár végig.

12.1-5. Annak alapján, hogy az összehasonlításos modellben n elem rendezésének ideje a legrosszabb esetben $\Omega(n \lg n)$, lássuk be, hogy egy n elemű listából a legrosszabb esetben ugyancsak $\Omega(n \lg n)$ idő alatt készíthető bináris keresőfa.

12.2. Keresés bináris keresőfában

A bináris keresőfákon használt leggyakoribb művelet a fában tárolt kulcsok keresése. A KERES művelet mellett a bináris keresőfákon más kereső műveletek is megvalósíthatók, így



12.2. ábra. Keresés bináris keresőfában. A fában lévő 13-as kulcs keresése során a gyökérből induló 15 → 6 → 7 → 13 utat követjük. A fa minimális kulcsa 2, melyet a gyökérből a *bal* mutatókat követve érhetünk el. A fa maximális kulcsa 20, melyet szintén a gyökérből indulva, a *jobb* mutatókat követve találhatunk meg. A 15-ös kulcsú csúcs rákövetkezője a 17-es kulcsú csúcs, mivel az a 15 bal oldali részfájának minimális kulcsa. A 13-as kulcsú csúcsnak nincs jobb oldali részfája, ezért annak rákövetkezője az a legalacsonyabban lévő ős, amelynek bal oldali gyereke is ős a 13-nak. Esetünkben a 15-ös kulcsú csúcs a rákövetkező.

például a MINIMUM, MAXIMUM, KÖVETKEZŐ és ELŐZŐ. Ebben az alfejezetben a most említett műveleteket vizsgáljuk, és megmutatjuk, hogy azok egy h magasságú bináris keresőfa esetén $O(h)$ idő alatt megvalósíthatók.

Adott kulcsú elem keresése

Egy adott kulcs bináris keresőfában való keresésére az alábbi FÁBAN-KERES algoritmust használjuk. Ha megadjuk a fa gyökerének mutatóját és a k kulcsértéket, akkor a FÁBAN-KERES a k kulcsú elem mutatóját adja vissza, ha van ilyen elem, illetve NIL-t ad vissza, ha ilyen elem nincs.

FÁBAN-KERES(x, k)

1 **if** x = NIL vagy $k = kulcs[x]$
2 **then return** x
3 **if** $k < kulcs[x]$
4 **then return** FÁBAN-KERES($bal[x], k$)
5 **else return** FÁBAN-KERES($jobb[x], k$)

Az algoritmus a gyökérnél kezdi a keresést, és egy lefelé haladó utat jár be a fában, ahogy ez a 12.2. ábrán látható. Minden érintett x csúcsnál összehasonlítja k-t $kulcs[x]$-szel. Ha a két kulcs egyenlő, a keresés befejeződik. Ha k kisebb, mint a $kulcs[x]$ érték, akkor x bal oldali részfájában folytatódik a keresés, hiszen a bináris-keresőfa tulajdonság miatt k a jobb oldali részfában már nem lehet. A szimmetria miatt, ha k nagyobb, mint $kulcs[x]$, akkor a keresés a jobb oldali részfában folytatódik. A rekurzió során az algoritmus által érintett csúcsok egy gyökérből induló utat alkotnak, ezért a FÁBAN-KERES végrehajtási ideje $O(h)$, ahol h a fa magassága.

Ugyanez az eljárás megírható iteratív algoritmusként is, a rekurziónak **while** ciklussá való „kibontásával”. A legtöbb számítógépen az utóbbi változat a hatékonyabb.

FÁBAN-ITERATÍVAN-KERES(x, k)

```
1  while x ≠ NIL vagy k ≠ kulcs[x]
2        do if k < kulcs[x]
3              then x ←bal[x]
4              else  x ←jobb[x]
5  return x
```

Minimum és maximum keresése

Egy bináris keresőfa minimális kulcsú eleme mindig könnyen megtalálható, ha addig követjük a bal oldali mutatókat, amíg NIL mutatót nem találunk, ahogy ezt a 12.2. ábra szemlélteti. A következő eljárás az x gyökerű részfa minimális elemének mutatóját adja eredményül.

FÁBAN-MINIMUM(x)

```
1  while bal[x] ≠ NIL
2        do x ←bal[x]
3  return x
```

A FÁBAN-MINIMUM algoritmus helyessége a bináris-keresőfa tulajdonságból következik. Ha az x csúcsnak nincs bal oldali részfája, akkor – mivel a jobb oldali részfájának minden kulcsa legalább olyan nagy, mint $kulcs[x]$ – az x gyökerű részfának valóban $kulcs[x]$ a minimuma. Ha az x csúcsnak van bal oldali részfája, akkor – mivel a jobb oldali részfának nem lehet $kulcs[x]$-nél kisebb kulcsú eleme és a bal oldali részfa minden kulcsa nem nagyobb $kulcs[x]$-nél – az x gyökerű fa minimális kulcsa a $bal[x]$ mutatójú részfában található meg.

A FÁBAN-MAXIMUM pszeudokódja az előző algoritmus szimmetrikus párja.

FÁBAN-MAXIMUM(x)

```
1  while jobb[x] ≠ NIL
2        do x ←jobb[x]
3  return x
```

h a magasságú fára mindkét algoritmus $O(h)$ idő alatt fut le, hasonlóan a FÁBAN-KERES eljáráshoz, mivel mindkettő egy gyökérből kiinduló úton halad végig.

Rákövetkező és megelőző meghatározása

Legyen adott egy bináris keresőfa egy csúcsa. Néha fontos, hogy meg tudjuk találni a növekvő sorrend szerinti rákövetkezőjét, melyet a fa inorder bejárása határoz meg. Ha a fában minden kulcs különböző, akkor az x csúcs rákövetkezője az a csúcs, melynek kulcsa a legkisebb olyan kulcs a fában, mely nagyobb $kulcs[x]$-nél. A bináris keresőfák szerkezete lehetővé teszi, hogy a rákövetkezőt kulcs-összehasonlítások nélkül megtalálhassuk. A most következő eljárás az x csúcs rákövetkezőjét adja vissza, ha a bináris keresőfában létezik ilyen, és NIL-t ad vissza, ha x a fa legnagyobb kulcsú eleme.

```
FÁBAN-KÖVETKEZŐ(x)
1  if jobb[x] ≠ NIL
2      then return FÁBAN-MINIMUM(jobb[x])
3  y ← p[x]
4  while y ≠ NIL és x = jobb[y]
5      do x ← y
6         y ← p[y]
7  return y
```

A FÁBAN-KÖVETKEZŐ eljárás kódja két részre bontható. Ha az x jobb oldali részfája nem üres, akkor x rákövetkezője éppen az x jobb oldali részfájának legkisebb eleme, amit a 2. sorban a FÁBAN-MINIMUM($jobb[x]$) hívással találunk meg. Például a 12.2. ábrán a 15-ös kulcsú csúcs rákövetkezője a 17-es kulcsú csúcs.

Másrészt, amint azt a 12.2-6. gyakorlat is kitűzi, ha x jobb oldali részfája üres és az x rákövetkezője az y, akkor y az x olyan legalacsonyabban lévő őse, amelynek bal oldali gyereke is őse x-nek. A 12.2. ábrán a 13-as kulcsú elem rákövetkezője a 15-ös kulcsú elem. Az y-t egyszerűen úgy találhatjuk meg, hogy a fában x-ből mindaddig felfelé megyünk, amíg egy olyan csúcsot nem találunk, amely szülője bal oldali gyereke. Ezt a FÁBAN-KÖVETKEZŐ eljárás 3–7. sorai valósítják meg.

A FÁBAN-KÖVETKEZŐ algoritmus futásideje h magasságú fák esetén $O(h)$, mivel mindkét programágon a fa egy útját követjük, az egyik esetben felülről lefelé, a másikban alulról fölfelé. A FÁBAN-ELŐZŐ algoritmus, amely a FÁBAN-KÖVETKEZŐ értelemszerű módosítása, ugyancsak $O(h)$ idő alatt fut.

Abban az esetben, ha a kulcsok nem különbözők, akkor az x csúcs rákövetkezőjét és megelőzőjét a FÁBAN-KÖVETKEZŐ(x) és FÁBAN-ELŐZŐ(x) hívások eredményeként kapott csúcsokkal definiáljuk.

Eddigi eredményeinket a következő tételben összegezhetjük.

12.2. tétel. *A dinamikus halmazokra vonatkozó* KERES, MINIMUM, MAXIMUM, KÖVETKEZŐ *és* ELŐZŐ *műveletek h magasságú bináris keresőfákon elvégezhetők $O(h)$ idő alatt.*

Gyakorlatok

12.2-1. Tegyük fel, hogy van egy olyan bináris keresőfánk, melyben a kulcsok az 1 és 1000 közötti számok közül valók. A 363 értéket akarjuk a fában megkeresni. Az alábbi sorozatok közül melyek nem lehetnek keresési sorozatok?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2. Írjuk meg a FÁBAN-MINIMUM és a FÁBAN-MAXIMUM eljárások rekurzív változatát.

12.2-3. Írjuk meg a FÁBAN-ELŐZŐ eljárást.

12.2-4. Bunyan professzor azt hitte, hogy felfedezte a bináris keresőfák egy érdekes tulajdonságát. Ez a tulajdonság a következő. Tegyük fel, hogy egy k kulcs keresése levélnél

fejeződik be, és tekintsük a következő három halmazt: A a keresési úttól balra lévő kulcsok, B a keresési úton lévő kulcsok, C pedig a keresési úttól jobbra lévő kulcsok halmaza. Bunyan professzor azt sejtette, hogy a három halmazból akárhogy kivéve egy-egy $a \in A$, $b \in B$, $c \in C$ elemet, azoknak ki kell elégíteniük az $a \leq b \leq c$ egyenlőtlenséget. Adjunk a professzor sejtését cáfoló, lehető legkisebb méretű ellenpéldát.

12.2-5. Mutassuk meg, ha egy bináris keresőfa egy csúcsának van két gyereke, akkor a rákövetkezőjének nincs bal oldali gyereke és a megelőzőjének nincs jobb oldali gyereke.

12.2-6. Tekintsünk egy T bináris keresőfát, amelynek a kulcsai különbözőek. Mutassuk meg, ha az x jobb oldali részfája üres a T-ben és az x rákövetkezője y, akkor y az x olyan legalacsonyabban lévő őse, amelynek bal oldali gyereke is őse x-nek. (Emlékezzünk, hogy minden csúcs egyben a saját maga őse is.)

12.2-7. Egy n csúcsú fa inorder bejárását megvalósíthatjuk úgy, hogy először a FÁBAN-MINIMUM eljárással megkeressük a minimális elemét, majd $(n-1)$-szer meghívjuk a FÁBAN-KÖVETKEZŐ eljárást. Bizonyítsuk be, hogy ez az algoritmus $\Theta(n)$ idő alatt fut le.

12.2-8. Bizonyítsuk be, hogy ha egy h magasságú bináris keresőfára egymás után k-szor meghívjuk a FÁBAN-KÖVETKEZŐ eljárást, akkor az a kiindulási pont megválasztásától függetlenül $O(k+h)$ idő alatt fut le.

12.2-9. Legyen T egy bináris keresőfa különböző kulcsokkal. Legyen az x ennek egy levele, és legyen y az x szülője. Mutassuk meg, hogy $kulcs[y]$ vagy a legkisebb kulcs a $kulcs[x]$-nél nagyobb T-beli kulcsok között, vagy a $kulcs[x]$-nél kisebb kulcsok között a legnagyobb.

12.3. Beszúrás és törlés

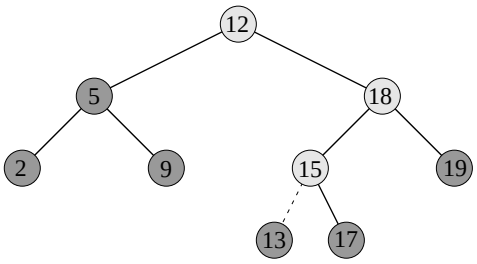
A beszúrás és törlés műveleteinek végrehajtása megváltoztatja a bináris keresőfával ábrázolt dinamikus halmazt. A módosítást természetesen úgy kell végrehajtani, hogy a bináris-keresőfa tulajdonság megmaradjon. Amint látni fogjuk, egy új elem beszúrása viszonylag egyszerű, a törlés kezelése azonban már kissé bonyolultabb.

Beszúrás

Egy új v értéket a T bináris keresőfába a FÁBA-BESZÚR eljárással illeszthetünk be. Az eljárás paraméterül egy z csúcsot kap, melyre $kulcs[z] = v$, $bal[z] = \text{NIL}$ és $jobb[z] = \text{NIL}$. Hogy z a fában a megfelelő helyre kerüljön, az eljárás módosítja T-t és z bizonyos mezőit.

A FÁBA-BESZÚR eljárás működését a 12.3. ábra szemlélteti. Hasonlóan a FÁBAN-KERES és a FÁBAN-ITERATÍVAN-KERES eljáráshoz, a FÁBA-BESZÚR is a gyökértől indul el és egy onnan leszálló utat jár be. Az x az úton végighaladó mutató, az y pedig mindig az x szülőjére mutat. A kezdőértékadást követő 3–7. sorokban a **while** ciklus végzi a két mutató lefelé mozgatását a fában. Mindaddig haladunk lefelé, $kulcs[z]$ és $kulcs[x]$ összehasonlításának az eredményétől függően hol balra, hol jobbra, amíg az x egyenlő nem lesz NIL-lel. Pontosan ennek a NIL-nek a helye az, ahová a z elemet be kell illeszteni. A 8–13. sorok a z elem befűzését végzik a mutatók megfelelő beállításával.

A FÁBA-BESZÚR eljárás – a bináris keresőfákon értelmezett többi egyszerű művelethez hasonlóan – h magasságú fákon $O(h)$ idő alatt végrehajtódik.



12.3. ábra. A 13 kulcsértékű elem beillesztése egy bináris keresőfába. A halványan árnyékolt csúcsok a gyökérből a beszúrás helyéhez vezető utat mutatják. A szaggatott vonal azt az új mutatót szemlélteti, amely az új elemet a fába illeszti.

FÁBA-BESZÚR(T, z)

```
 1  y ← NIL
 2  x ← gyökér[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if kulcs[z] < kulcs[x]
 6              then x ← bal[x]
 7              else  x ←jobb[x]
 8  p[z] ← y
 9  if y = NIL
10        then gyökér[T] ← z                ▷ Üres T fa esetén.
11        else if kulcs[z] < kulcs[y]
12                then bal[y] ← z
13                else  jobb[y] ← z
```

Törlés

Egy bináris keresőfa valamely z csúcsának törlését végző eljárás a törlendő z csúcs mutatóját kapja argumentumként. Ahogyan azt a 12.4. ábra mutatja, az eljárás három esetet vizsgál meg. Ha z-nek nincs gyereke, akkor egyszerűen módosítjuk a szülőjét, $p[z]$-t, z helyett NIL-re. Ha z-nek egy gyereke van, akkor őt „kivágjuk” oly módon, hogy a szülője és gyereke között építünk ki új kapcsolatot. Végül, ha a csúcsnak két gyereke van, akkor z-nek azt az y legközelebbi rákövetkezőjét „vágjuk ki”, melynek már nincs bal oldali gyereke (lásd a 12.2-5. gyakorlatot) és z tartalmát az y tartalmával helyettesítjük.

A FÁBÓL-TÖRÖL kódjában a három esetet kissé másként szervezzük. Az 1–3. sorokban az algoritmus meghatározza, hogy mely y csúcsot kell kivágni. Az y vagy maga a z bemenő csúcs (ha z-nek legfeljebb 1 gyereke van) vagy z rákövetkezője (ha z-nek két gyereke van). Ezután a 4–6. sorokban x-et vagy y nem-NIL gyermekére állítjuk, vagy ha nincs ilyen, akkor NIL-re. Az y kivágása a 7–13. sorokban történik a $p[y]$ és x mutatóinak módosításával. Az y kivágása azért bonyolultabb kissé, mert pontosan kell kezelni a fa határaira vonatkozó feltételeket, vagyis azokat az eseteket, amikor x = NIL, illetve amikor y a gyökérrel azonos. Végül, ha a kivágott y csúcs a z rákövetkezője volt, akkor y tartalmát 14–16. sorokban átmásoljuk z-be, felülírva annak korábbi tartalmát. Az eljárás az y csúcsot a 17. sorban ered-

ményként visszaadja, és így azt a hívó eljárás a szabad listán keresztül újra felhasználhatóvá teheti. Az eljárás $O(h)$ idő alatt fut le, ahol h a fa magassága.

```
FÁBÓL-TÖRÖL(T, z)
 1  if bal[z] = NIL vagy jobb[z] = NIL
 2      then y ← z
 3      else y ← FÁBAN-KÖVETKEZŐ(z)
 4  if bal[y] ≠ NIL
 5      then x ← bal[y]
 6      else x ← jobb[y]
 7  if x ≠ NIL
 8      then p[x] ← p[y]
 9  if p[y] = NIL
10      then gyökér[T] ← x
11      else if y = bal[p[y]]
12              then bal[p[y]] ← x
13              else jobb[p[y]] ← x
14  if y ≠ z
15      then kulcs[z] ←kulcs[y]
16              ▷ ha y-nak más mezői is vannak, azok átmásolása
17  return y
```

Eddigi eredményeinket a következő tételben összegezhetjük.

12.3. tétel. *A dinamikus halmazokra vonatkozó* BESZÚR *és* TÖRÖL *műveletek h magasságú bináris keresőfákon elvégezhetők $O(h)$ idő alatt.*

Gyakorlatok

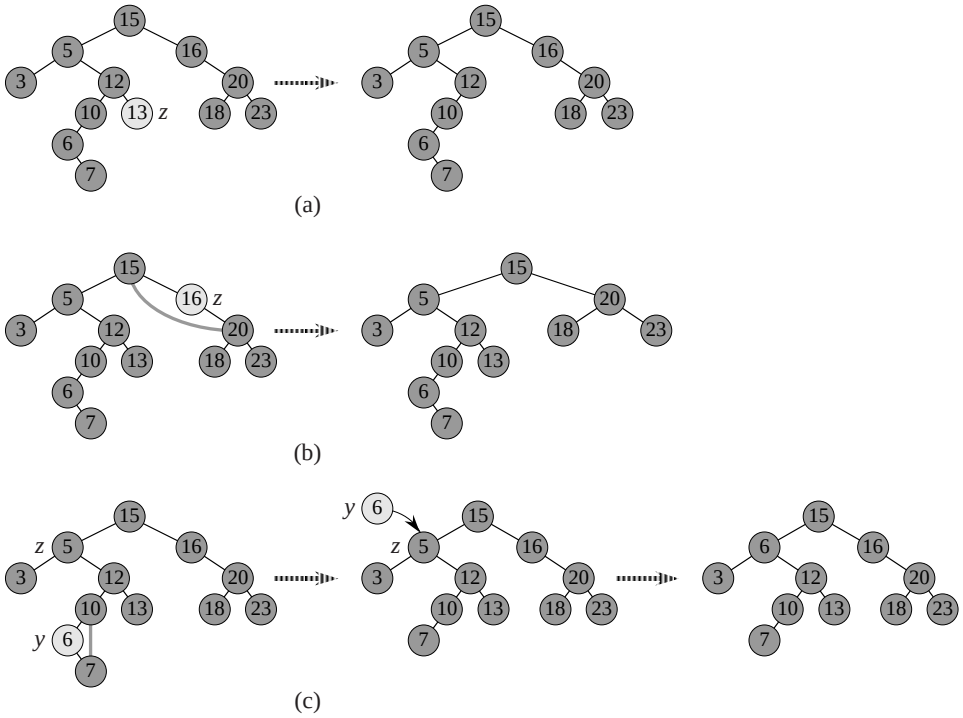
12.3-1. Adjuk meg a FÁBA-BESZÚR eljárás rekurzív változatát.

12.3-2. Tegyük fel, hogy egy bináris keresőfát úgy hoztunk létre, hogy az üres fába egymástól különböző értékeket szúrtunk be egymás után. Lássuk be, hogy egy érték keresése során megvizsgált csúcsok száma eggyel több, mint azoknak a csúcsoknak a száma, amelyeket ennek az értéknek a fába való beszúrása során kellett megvizsgálni.

12.3-3. Egy halmaz elemeit rendezhetjük úgy, hogy először építünk egy, a halmaz elemeit tartalmazó bináris keresőfát (az elemeknek az üres fába történő egymás utáni beszúrásával), majd a fa tartalmát az inorder bejárással kiírjuk. Mi ennek a rendezési eljárásnak a legrosszabb és legjobb futási ideje?

12.3-4. Tegyük fel, hogy egy bináris keresőfa egy y csúcsára egy másik adatszerkezetből is mutat pointer. Milyen probléma keletkezhet, ha a fából a FÁBÓL-TÖRÖL eljárással töröljük y megelőzőjét, a z-t? Hogyan kellene átírni a FÁBÓL-TÖRÖL eljárást, hogy ezt a problémát megoldjuk?

12.3-5. Igaz-e, hogy a törlés művelete kommutatív abban az értelemben, hogy egy bináris keresőfából előbb x-et, majd y-t törölve ugyanahhoz a fához jutunk, mint ha előbb töröljük a fából y-t, és azután az x-et? Lássuk be a kommutativitást, ha igaz, illetve adjunk ellenpéldát, ha nem igaz.



12.4. ábra. A z csúcs törlése egy bináris keresőfából. Mindhárom esetnél halványan árnyékoltuk az aktuálisan törlendő elemet. (**a**) Ha z-nek nincs gyereke, akkor azonnal eltávolítjuk. (**b**) Ha z-nek egy gyereke van, akkor z-t kivágjuk. (**c**) Ha z-nek két gyereke van, akkor azt a legközelebbi rákövetkezőjét vágjuk ki, melynek nincs bal oldali gyereke, majd z tartalmát helyettesítjük y tartalmával.

12.3-6. Amikor a FÁBÓL-TÖRÖL eljárásban z-nek két gyereke van, akkor a törlést nemcsak a rákövetkezőjének, hanem a megelőzőjének kivágásával is végezhetnénk. Néhányan úgy tartják, hogy a pártatlan stratégia, amikor egyforma esélyt adunk a megelőzőnek és a rákövetkezőnek, jobb gyakorlati eredményekre vezet. Hogyan kellene a FÁBÓL-TÖRÖL eljárást módosítani, hogy ezt a pártatlan stratégiát valósítsa meg?

★ 12.4. Véletlen építésű bináris keresőfák

Megmutattuk, hogy a bináris keresőfákon értelmezett összes alapművelet h magasságú fákon $O(h)$ idő alatt hajtódik végre. Azonban a bináris keresőfák magassága változik azáltal, hogy elemeket szúrunk be és törlünk. Ha például az elemeket szigorúan növekvő sorrendben szúrjuk be, akkor a fa tulajdonképpen egy lánc lesz $n - 1$ magassággal. Másrészt a B.5-4. gyakorlat mutatja, hogy $h \geq \lfloor \lg n \rfloor$. Hasonlóan a gyorsrendezéshez, belátható, hogy az átlagos eset viselkedése közelebb áll a legjobb, mint a legrosszabb esethez.

Ha egy bináris keresőfa beszúrások és törlések sorozatával épül fel, akkor sajnos keveset tudunk mondani az átlagos magasságáról. Ha viszont a felépítés során csak beszúrásokat

használunk, akkor az elemzés könnyebben kezelhetővé válik. Ezért vezetjük be a ***véletlen építésű bináris keresőfa*** fogalmát. Legyen adva n egymástól különböző kulcs, amelyekből bináris keresőfát építünk úgy, hogy a kulcsokat valamilyen sorrendben egymás után beszúrjuk a kezdetben üres fába. Ha itt minden sorrend, vagyis az n kulcsnak mind az $n!$ permutációja egyformán valószínű, akkor a kapott fát véletlen építésű bináris keresőfának nevezzük. (A 12.4-3. gyakorlat annak megmutatását tűzi ki célul, hogy a most definiált fogalom nem egyezik meg azzal, amelyben azt tételezzük fel, hogy az n kulcsot tartalmazó bináris keresőfák mind egyformán valószínűek.) Ebben az alfejezetben megmutatjuk, hogy egy n kulcsból véletlen módon épített bináris keresőfa magassága $O(\lg n)$. A továbbiakban feltesszük, hogy minden kulcs különböző.

Először három valószínűségi változót definiálunk, hogy mérni tudjuk a véletlen építésű bináris keresőfák magasságát. Egy n kulcsú véletlen építésű bináris keresőfa magasságát jelöljük X_n-nel, és definiáljuk az ***exponenciális magasságot*** Y_n-nel úgy, hogy $Y_n = 2^{X_n}$. Jelölje az R_n valószínűségi változó a gyökérnek választott kulcs rangját az n darab kulcs halmazában, amelyekből egy bináris keresőfát építünk. R_n egyenlő valószínűséggel veszi fel az $\{1, 2, \ldots, n\}$ halmaz bármely elemét. Ha $R_n = i$, akkor a gyökér bal részfája $i - 1$, míg jobb részfája $n - i$ kulcsot tartalmaz egy véletlen építésű bináris keresőfában. Mivel egy bináris fa magassága eggyel nagyobb a gyökérhez kapcsolódó két részfa nagyobbikának magasságánál, ezért egy bináris fa exponenciális magassága kétszerese a nagyobbik részfa exponenciális magasságának. Ha tudjuk, hogy $R_n = i$, akkor

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}).$$

Az alap esetekben, az egy csúcsú fa exponenciális magassága legyen $Y_1 = 1$, mivel $2^0 = 1$, és tekintsük definíciónak, hogy $Y_0 = 0$.

Ezután definiáljuk a $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$ indikátor valószínűségi változókat, ahol

$$Z_{n,i} = \mathrm{I}\{R_n = i\}.$$

Mivel R_n egyenlő valószínűséggel vesz fel értéket az $\{1, 2, \ldots, n\}$ halmazból, ezért $\Pr\{R_n = i\} = 1/n$, ahol $i = 1, 2, \ldots, n$, és az 5.1. lemma szerint

$$\mathrm{E}[Z_{n,i}] = \frac{1}{n}, \tag{12.1}$$

ahol $i = 1, 2, \ldots, n$. Mivel pontosan egy $Z_{n,i}$ érték 1 és az összes többi 0, ezért

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right).$$

Megmutatjuk, hogy $\mathrm{E}[Y_n]$ egy polinomja n-nek, amelyből következik, hogy $\mathrm{E}[X_n] = O(\lg n)$.

A $Z_{n,i} = \mathrm{I}\{R_n = i\}$ indikátor valószínűségi változó független az Y_{i-1} és Y_{n-i} értékektől. $R_n = i$ választás esetén, a bal részfába, melynek exponenciális magassága Y_{i-1}, i-nél kisebb rangú $i - 1$ kulcs kerül véletlenszerűen. Ez a részfa ugyanolyan, mint bármely más $i - 1$ kulcsot tartalmazó véletlen építésű bináris keresőfa. Eltekintve a tartalmazott kulcsok számától, ennek a részfának a struktúrájára egyáltalán nincs hatással az $R_n = i$ választás, mivel az Y_{i-1} valószínűségi változó független $Z_{n,i}$-től. Hasonlóan, a jobb részfa, melynek

exponenciális magassága Y_{n-i}, $n-i$ darab olyan kulcsból épül fel véletlenszerűen, melynek rangja nagyobb, mint i. Ennek struktúrája is független az R_n értékétől, azaz az Y_{n-i} és $Z_{n,i}$ valószínűségi változók függetlenek. Tehát,

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))] && \text{(várható érték linearitása miatt)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(változók függetlensége miatt)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{((12.1) egyenlőség miatt)} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}[\max(Y_{i-1}, Y_{n-i})] && \text{((C.21) egyenlőség miatt)} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(C.3-4. gyakorlat alapján).}
\end{aligned}$$

Az utolsó összegben az $\mathrm{E}[Y_0], \mathrm{E}[Y_1], \ldots, \mathrm{E}[Y_{n-1}]$ tagok kétszer fordulnak elő, egyszer $\mathrm{E}[Y_{i-1}]$-ként és még egyszer $\mathrm{E}[Y_{n-i}]$ alakban, tehát

$$\mathrm{E}[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i]\,. \tag{12.2}$$

A helyettesítési módszerrel megmutatjuk, hogy minden pozitív egész n esetén a (12.2) rekurzív egyenlőtlenség megoldása

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}.$$

A bizonyításban felhasználjuk a következő azonosságot:

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}. \tag{12.3}$$

(A 12.4-1. gyakorlat ennek bizonyítását tűzi ki.)

Az alap esetre az állítás igaz, mivel érvényesek a következő korlátok:

$$\begin{aligned}
0 &= Y_0 \\
&\le \frac{1}{4}\binom{3}{3} \\
&= \frac{1}{4}
\end{aligned}$$

és

$$\begin{aligned} 1 &= Y_1 \\ &= \mathrm{E}[Y_1] \\ &\leq \frac{1}{4}\binom{1+3}{3} \\ &= 1. \end{aligned}$$

Általános esetben pedig azt kapjuk, hogy

$$\begin{aligned} \mathrm{E}[Y_n] &\leq \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\ &\leq \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(indukciós feltétel miatt)} \\ &= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\ &= \frac{1}{n}\binom{n+3}{4} && \text{(a (12.3) egyenlőség szerint)} \\ &= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\ &= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\ &= \frac{1}{4}\binom{n+3}{3}. \end{aligned}$$

Sikerült megbecsülnünk $\mathrm{E}[Y_n]$-t, de az eredeti célkitűzésünk az volt, hogy felső korlátot adjunk $\mathrm{E}[X_n]$-re. Mivel a 12.4-4. gyakorlat annak bizonyítását kéri, hogy az $f(x) = 2^x$ konvex (lásd C.3. alfejezet), ezért alkalmazhatjuk a Jensen-egyenlőtlenséget (C.25), amely szerint

$$2^{\mathrm{E}[X_n]} \leq \mathrm{E}\left[2^{X_n}\right] = \mathrm{E}[Y_n],$$

amely azt eredményezi, hogy

$$\begin{aligned} 2^{\mathrm{E}[X_n]} &\leq \frac{1}{4}\binom{n+3}{3} \\ &= \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6} \\ &= \frac{n^3+6n^2+11n+6}{24}. \end{aligned}$$

Mindkét oldal logaritmusát véve azt kapjuk, hogy $\mathrm{E}[X_n] = O(\lg n)$. Ezáltal bizonyítottuk a következő állítást.

12.4. tétel. *Egy n különböző kulcsot tartalmazó véletlen építésű bináris keresőfa várható magassága* $O(\lg n)$.

Gyakorlatok

12.4-1. Bizonyítsuk be a (12.3) egyenlőséget.

12.4-2. Írjunk le egy olyan n csúcsú bináris keresőfát, melyben a csúcsok átlagos mélysége $\Theta(\lg n)$, de a fa magassága $\omega(\lg n)$. Adjunk egy aszimptotikus felső határt egy n csúcsú bináris keresőfa magasságára, ha a csúcsok átlagos mélysége $\Theta(\lg n)$.

12.4-3. Bizonyítsuk be, hogy az n csúcsú véletlen módon választott bináris keresőfa fogalma (ahol minden n csúcsú bináris keresőfát egyforma valószínűséggel választhatunk) nem esik egybe a véletlen építésű bináris keresőfa fogalmával, amelyet ebben a részben definiáltunk. (*Útmutatás.* Soroljuk fel a lehetőségeket $n = 3$ esetén.)

12.4-4. Mutassuk meg, hogy az $f(x) = 2^x$ függvény konvex.

12.4-5.⋆ Alkalmazzuk a VÉLETLEN-GYORSRENDEZ eljárást egy n különböző értéket tartalmazó bemenő sorozatra. Bizonyítsuk be, hogy tetszőleges $k > 0$ szám esetén az a lehetséges $n!$ bemenő permutáció – $O(1/n^k)$ kivételével – $O(n \lg n)$ futási időre vezet.

Feladatok

12-1. Bináris keresőfák egyenlő kulcsokkal

Az egyforma kulcsok problémákat vetnek fel a bináris keresőfák megvalósításánál.

a. Milyen a FÁBA-BESZÚR eljárás aszimptotikus viselkedése, ha a kezdetben üres fába n egyedet szúrunk be azonos kulccsal?

A FÁBA-BESZÚR olyan javítását javasoljuk, melyben az 5. sor, illetve a 11. sor előtt megvizsgáljuk a $kulcs[z] = kulcs[x]$, illetve a $kulcs[z] = kulcs[y]$ feltételeket. Ha az egyenlőség fennáll, akkor az alábbi stratégiák valamelyikét alkalmazzuk. Állapítsuk meg az egyes stratégiák aszimptotikus viselkedését, abban az esetben, ha a kezdetben üres fába n darab azonos kulcsú elemet szúrunk be. (A stratégiákat az 5. sor előtti vizsgálat esetére írjuk le, amikor $kulcs[z]$-t és $kulcs[x]$-et hasonlítjuk össze. A 11. sorhoz szükséges módosítást az x-nek y-ra való cseréjével kaphatjuk meg.)

b. Vegyünk fel minden x csúcs esetén egy új $b[x]$ mezőt, amelyben egy jelzőbitet tárolunk. Az x-et a jelzőbit értékétől függően állítsuk tovább, hol $bal[x]$-re, hol $jobb[x]$-re. A jelzőbit váltakozva a HAMIS és IGAZ értékeket veszi fel, mindannyiszor váltva, amikor az x csúcs a beszúrás során egy vele azonos kulcsú csúcsot érint.

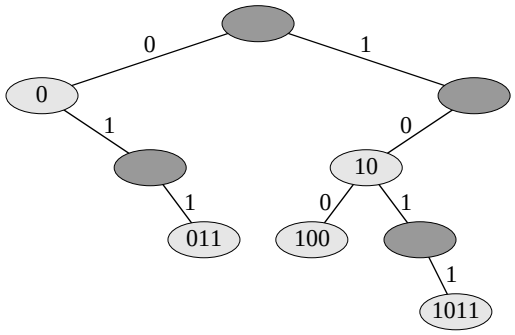
c. Az egyenlő kulcsú elemeket az x elemben lévő listában tároljuk, és z-t ide szúrjuk be.

d. Az x értékét véletlen módon állítsuk $bal[x]$-re vagy $jobb[x]$-re. (Adjuk meg a hatékonyságot a legrosszabb esetben, és informális módon vezessük le az átlagos hatékonyságot.)

12-2. Radix fák

Legyenek $a = a_0 a_1 \ldots a_p$ és $b = b_0 b_1 \ldots b_q$ valamely rendezett karakterhalmaz elemeiből képzett karaktersorozatok. Azt mondjuk, hogy az a sorozat ***lexikografikusan kisebb, mint*** b, ha

1. létezik olyan j egész, ahol $0 \leq j \leq \min(p, q)$ úgy, hogy $a_i = b_i$ minden $i = 0, 1, \ldots, (j-1)$-re és $a_j < b_j$, vagy
2. $p < q$ és $a_i = b_i$ minden $i = 0, 1, \ldots, p$-re.



12.5. ábra. Az 1011, 10, 011, 100 és 0 bitsorozatokat tartalmazó radix fa. A csúcsokban levő értéket a gyökérből hozzájuk vezető út határozza meg. Ezért nincs is szükség maguknak a kulcsoknak a tényleges tárolására, csak szemléltetés céljából szerepelnek az ábrán. Azok a csúcsok, amelyekhez tartozó kulcsok nincsenek benne a fában, sötétek; az ilyen csúcsok csak azért szerepelnek, hogy létrehozzák az utat a többiek irányába.

Ha például a és b bitsorozatok, akkor 10100 < 10110 teljesül az első szabály miatt (a $j = 3$ választással), míg 10100 < 101000 a 2. szabály miatt. Ez a rendezés hasonló a szótárakban lévő rendezettséghez.

A 12.5. ábrán bemutatott ***radix fa*** típusú adatszerkezetben az 1011, 10, 011, 100 és 0 bitsorozatok szerepelnek. Amikor az $a = a_0 a_1 \ldots a_p$ kulcsot keressük, akkor az i-edik szinten balra lépünk, ha $a_i = 0$ és jobbra lépünk, ha $a_i = 1$. Legyen S egy olyan halmaz, amely egymástól különböző bitsorozatokat tartalmaz, és ezek összhosszúsága n. Mutassuk meg, hogyan használható a radix fa az S halmaz $\Theta(n)$ idejű lexikografikus rendezésére. A 12.5. ábrán látható példában a rendezés kimenetének a 0,011,10,100,1011 sorozatnak kell lennie.

12-3. Véletlen építésű bináris keresőfák átlagos csúcsmélysége

Ebben a feladatban azt mutatjuk meg, hogy egy véletlen építésű n csúcsú bináris keresőfa átlagos csúcsmélysége $O(\lg n)$. Bár ez az eredmény gyengébb, mint a 12.4. tétel állítása, de a használt technika feltárja a bináris keresőfa építése és a 7.3. alfejezetben definiált VÉLETLENGYORSRENDEZ működése közötti meglepő hasonlóságot.

Definiáljuk a T fa $P(T)$ ***teljes úthosszát,*** mint a T-beli x csúcsok mélységeinek összegét. Az x csúcs mélységét $d(x, T)$-vel jelöljük.

a. Igazoljuk, hogy egy T fa csúcsainak átlagos mélysége

$$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Így elég belátni, hogy $P(T)$ várható értéke $O(n \lg n)$.

b. Jelölje T_L és T_R a T fa bal oldali és jobb oldali részfáját. Bizonyítsuk be, ha T csúcsainak száma n, akkor

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

c. Jelölje $P(n)$ a véletlen építésű n csúcsú bináris keresőfa átlagos teljes úthosszát. Mutassuk meg, hogy

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1).$$

d. Mutassuk meg, hogy $P(n)$ átírható a következő alakba:

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n).$$

e. A véletlen gyorsrendező hasonló elemzését felidézve, amely a 7-2. feladatban szerepel, vonjuk le azt a következtetést, hogy $P(n) = O(n \lg n)$.

A gyorsrendező minden rekurzív hívásakor kiválasztunk egy véletlen elemet, amely a rendezendő elemeket részhalmazokra bontja. Egy bináris keresőfa bármely csúcsa is ugyanígy bontja részekre az általa meghatározott részfa elemeinek a halmazát.

f. Adjuk meg a gyorsrendező egy olyan megvalósítását, melyben a rendezéshez szükséges összehasonlítások pontosan ugyanazok, mint amikor az elemeket egy bináris keresőfába szúrjuk be. (Az összehasonlítások elvégzésének sorrendje lehet különböző, de ugyanazokat az összehasonlításokat kell elvégezni.)

12-4. Különböző bináris fák száma

Jelölje b_n az n csúcsú különböző bináris fák számát. Ebben a feladatban képletet adunk b_n-re és megállapítjuk annak aszimptotikus viselkedését.

a. Mutassuk meg, hogy $b_0 = 1$ és minden $n \geq 1$ esetén

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

b. Emlékeztetve a 4-5. feladatban szereplő generátorfüggvények definíciójára, legyen $B(x)$ a következő generátorfüggvény:

$$B(x) = \sum_{n=0}^{\infty} b_n x^n.$$

Mutassuk meg, hogy $B(x) = xB(x)^2 + 1$ és innen

$$B(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x}).$$

Egy $f(x)$ függvénynek az a pont körüli ***Taylor-sorát*** a következő egyenlőséggel adhatjuk meg:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x-a)}{k!}(x-a)^k,$$

ahol $f^{(k)}(x)$ az f k-adik deriváltjának értéke az x helyen.

c. Az $\sqrt{1-4x}$ függvény $x = 0$ körüli Taylor-sorának segítségével mutassuk meg, hogy

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(ez az n-edik ***Catalan-szám***). (A Taylor-sor helyett használhatjuk a (C.4) binomiális sor valós számokra történő általánosítását. Tetszőleges n valós és k egész szám esetén $\binom{n}{k}$-t az $n(n-1)\ldots(n-k+1)/k!$ képlettel értelmezzük, ha $k \geq 0$, egyébként pedig 0-nak vesszük.)

d. Mutassuk meg, hogy

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O\left(\frac{1}{n}\right)\right).$$

Megjegyzések a fejezethez

Knuth [185] jó áttekintést ad az egyszerű bináris keresőfákról és azok számos lehetséges változatáról. A bináris keresőfákat többen egymástól függetlenül fedezték fel az 50-es évek végén. Az angol nyelvű szakirodalomban a radix fákat gyakran „tri”-knek is nevezik, ami a „retrieval” szóra utal. Ezeket Knuth [185] szintén tárgyalja.

A 15.5. alfejezetben megmutatjuk, miként kell egy optimális bináris keresőfát konstruálni, amikor a fa építése előtt már ismert a keresések gyakorisága, azaz minden kulcsra adott, hogy milyen gyakran kell keresni, illetve, hogy milyen gyakran esnek a keresett értékek a fában szereplő kulcsok közé. Az adott keresések halmazára egy olyan bináris keresőfát konstruálunk, amelyben a legkevesebb csúcsot kell megvizsgálni.

A 12.4. alfejezetben szereplő bizonyítás, amely korlátot ad a véletlen építésű bináris keresőfák várható magasságára, Aslamnak [23] tulajdonítható. Martínez és Roura [211] véletlen algoritmust ad a bináris keresőfákba történő beszúrásra és törlésre, mely műveletek eredménye egy véletlen bináris keresőfa. Azonban az ő véletlen bináris keresőfa definíciójuk némileg eltér a jelen fejezetben tárgyalt véletlen építésű bináris keresőfáktól.

13. Piros-fekete fák

A 12. fejezetben megmutattuk, hogy az alapvető dinamikus-halmaz műveletek – KERES, KÖVETKEZŐ, ELŐZŐ, MINIMUM, MAXIMUM, BESZÚR, TÖRÖL – mindegyike megvalósítható $O(h)$ idejű algoritmussal, ha a halmazt h magasságú bináris keresőfával ábrázoljuk. Tehát a halmaz-műveletek gyorsak, ha a magasság kicsi, de hatékonyságuk a legrosszabb esetben nem lesz jobb, mint láncolt lista esetén. A piros-fekete fa a sokféle „kiegyensúlyozott" keresőfa egyike: lehetővé teszi, hogy az alapvető dinamikus-halmaz műveletek a legrosszabb esetben is $O(\lg n)$ idejűek legyenek.

13.1. Piros-fekete fák tulajdonságai

A ***piros-fekete fa*** olyan bináris keresőfa, melynek minden csúcsa egy extra bit információt tartalmaz, ez a csúcs színe, amelynek értékei: PIROS vagy FEKETE. A csúcsok színezésének korlátozásával biztosítható, hogy piros-fekete fában bármely, a gyökértől levélig vezető út hossza nem lehet nagyobb, mint a legrövidebb ilyen út hosszának kétszerese. Tehát az ilyen fák megközelítőleg ***kiegyensúlyozottak***.

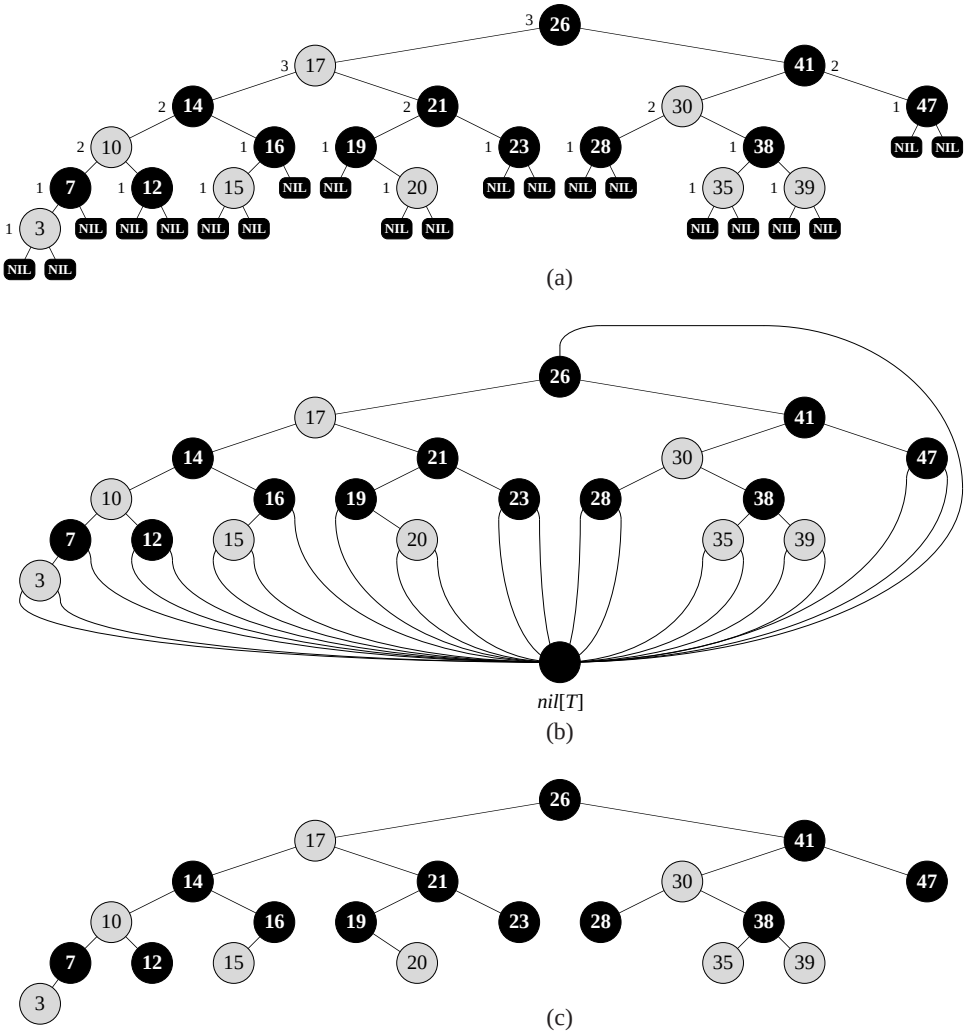
A fa minden csúcsa tartalmazza a *szín*, *kulcs*, *bal*, *jobb* és *szülő* mezőket. Ha egy csúcshoz tartozó gyerek vagy szülő nem létezik, akkor a megfelelő mező a NIL értéket tartalmazza. Úgy tekintjük, hogy az ilyen NIL mutató értékek a bináris keresőfa külső (levél) csúcsaira mutatnak, míg a fa kulcsot tartalmazó csúcsai a belső csúcsok.

Egy bináris keresőfa piros-fekete fa, ha teljesül rá a következő ***piros-fekete tulajdonság:***

1. Minden csúcs színe vagy piros, vagy fekete.
2. A gyökércsúcs színe fekete.
3. Minden levél (NIL) színe fekete.
4. Minden piros csúcsnak mindkét gyereke fekete.
5. Bármely csúcsból bármely levélig vezető úton ugyanannyi fekete csúcs van.

A 13.1(a) ábra egy piros-fekete fát mutat.

Piros-fekete fa algoritmusokban a peremfeltételek kényelmes kezelése végett egyetlen őrszem csúcsot használunk a NIL érték ábrázolására (lásd a 10.2. alfejezetet). Egy T piros-



13.1. ábra. A sötét csúcsok az ábrán fekete csúcsok, a világosan satírozottak pedig a piros csúcsok. Minden csúcs vagy piros, vagy fekete színű. Minden piros csúcs mindkét gyereke fekete, és minden csúcsból induló, levélig vezető úton ugyanannyi fekete csúcs található. **(a)** Minden levél (NIL) színe fekete. Minden nem levél csúcs mellé odaírtuk a csúcs fekete-magasságát, a NIL csúcsok fekete-magassága 0. **(b)** Ugyanaz a piros-fekete fa, de minden NIL csúcsot a *nil*[*T*] őrszemmel helyettesítettünk, amely mindig fekete színű, és a fekete-magasságát nem jelöltük. A gyökércsúcs szülője szintén az őrszem. **(c)** Ugyanaz a piros-fekete fa, de a leveleket és a gyökér szülőjét teljesen elhagytuk. Ezt az ábrázolási formát fogjuk használni a fejezet hátralévő részében.

fekete fa esetén a *nil*[*T*] egy, a fa közönséges csúcsaival megegyező mezőket tartalmazó csúcs. A *szín* mezőjének értéke FEKETE, és a többi mezőjének – *szülő*, *kulcs*, *bal*, *jobb* – értéke tetszőleges lehet. Amint azt a 13.1(b) ábra mutatja, minden NIL csúcsot helyettesítünk a *nil*[*T*] őrszemmel.

Az őrszem használata lehetővé teszi, hogy bármely *x* csúcs NIL gyerekét olyan közönséges csúcsnak tekintsük, amelynek szülője *x*. Bevezethetnénk minden NIL számára egyedi

őrszemet, azonban ez tárpazarló lenne. Ehelyett egyetlen $nil[T]$ őrszemet használunk az összes NIL érték – az összes levél és a gyökér szülője ábrázolására. A *szülő*, *kulcs*, *bal*, *jobb* mezők értéke lényegtelen, de eljárásokban kényelmi okok miatt értéküket beállíthatjuk.

Számunkra általában a piros-fekete fa belső csúcsai érdekesek, mert ezek tartalmazzák a kulcs értékeket. A fejezet hátralévő részében elhagyjuk a leveleket, amint azt a 13.1(c) ábra mutatja.

Egy x csúcs ***fekete-magasságának*** nevezzük az x csúcsból induló, levélig vezető úton található, x-et nem tartalmazó fekete csúcsok számát, és $fm(x)$-szel jelöljük ezt az értéket. Az 5. tulajdonság miatt a fekete-magasság jól definiált, mivel minden ilyen út azonos számú fekete csúcsot tartalmaz. Egy piros-fekete fa fekete-magasságát a fa gyökércsúcsának fekete-magasságaként definiáljuk.

A következő lemma megmutatja, hogy miért jók a piros-fekete fák.

13.1. lemma. *Bármely n belső csúcsot tartalmazó piros-fekete fa magassága legfeljebb* $2\lg(n+1)$.

Bizonyítás. Először megmutatjuk, hogy a fa minden x gyökerű részfája legalább $2^{fm(x)} - 1$ belső csúcsot tartalmaz. Az állítást x magassága szerinti matematikai indukcióval bizonyítjuk. Ha x magassága 0, akkor x levél ($nil[T]$), tehát az x gyökerű részfának valóban $2^{fm(x)} - 1 = 2^0 - 1 = 0$ belső csúcsa van. Tegyük fel, hogy x magassága pozitív, és két gyereke van. Mindkét gyerekének a fekete-magassága vagy $fm(x)$, vagy $fm(x) - 1$ attól függően, hogy a színe piros vagy fekete. Mivel x gyerekeinek magassága kisebb, mint x magassága, így az indukciós feltevést alkalmazva azt kapjuk, hogy mindkét részfa legalább $2^{fm(x)-1} - 1$ belső csúcsot tartalmaz. Tehát az x gyökerű részfa belső csúcsainak száma legalább $(2^{fm(x)-1} - 1) + (2^{fm(x)-1} - 1) + 1 = 2^{fm(x)} - 1$, ami az állítás bizonyítását jelenti.

A lemma bizonyításához legyen x magassága h. A 4. tulajdonság szerint minden olyan út, amelyik a gyökértől egy levélig halad, legalább feleannyi fekete csúcsot tartalmaz, mint ezen út csúcsainak száma, nem számítva a gyökeret. Tehát a gyökér fekete-magassága legalább $h/2$, így

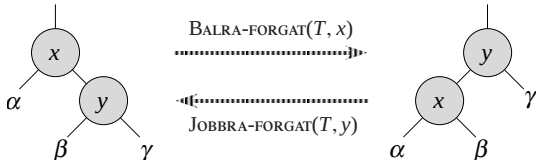
$$n \geq 2^{h/2} - 1.$$

Az egyenlőtlenség mindkét oldalához 1-et adva, majd logaritmusát véve kapjuk, hogy $\lg(n+1) \geq h/2$, vagy $h \leq 2\lg(n+1)$. ■

Ezen lemma közvetlen következménye, hogy a KERES, MINIMUM, MAXIMUM, KÖVETKEZŐ, ELŐZŐ dinamikus-halmaz műveletek mindegyike megvalósítható $O(\lg n)$ időben piros-fekete fákkal, mivel a futási idő $O(h)$, ha a fa magassága h (mint láttuk a 12. fejezetben), és egy n csúcsú piros-fekete fa magassága $O(\lg n)$ (a 12. fejezet algoritmusaiban a NIL hivatkozások helyettesítendőek $nil[T]$-vel). Bár a 12. fejezetben adott FÁBA-BESZÚR és FÁBÓL-TÖRÖL algoritmusok futási ideje szintén $O(\lg n)$, ha az input piros-fekete fa, azonban közvetlenül nem támogatják a BESZÚR és a TÖRÖL műveleteket, mert az algoritmus által módosított fa nem feltétlenül lesz ismét piros-fekete fa. Látni fogjuk majd a 13.3. és 13.4. alfejezetekben, hogy ezek a műveletek is megvalósíthatók $O(\lg n)$ időben piros-fekete fákkal.

Gyakorlatok

13.1-1. Rajzoljuk meg az $\{1, 2, \ldots, 15\}$ kulcsokat tartalmazó 3 magasságú teljes bináris keresőfát. Egészítsük ki NIL levelekkel és színezzük be a csúcsokat úgy, hogy a fa piros-fekete magassága 2, 3, illetve 4 legyen.



13.2. ábra. Forgató műveletek piros-fekete fákon. A BALRA-FORGAT(T, x) művelet az ábra bal oldalán látható konfigurációt a jobb oldali konfigurációba viszi át konstans számú mutató megváltoztatásával. A jobb oldali konfigurációt a BALRA-FORGAT(T, x) inverz művelet a bal oldalon látható konfigurációba transzformálja. Az α, β és γ betűk tetszőleges részfákat jelölnek. Mindkét forgató művelet megőrzi a bináris keresőfa tulajdonságot: az α részfában minden kulcs kisebb vagy egyenlő, mint a *kulcs*[x], ami kisebb vagy egyenlő, mint bármely kulcs a β részfában, továbbá ezek mindegyike kisebb vagy egyenlő, mint *kulcs*[y], ami kisebb vagy egyenlő, mint bármely kulcs a γ részfában.

13.1-2. Rajzoljuk fel azt a piros-fekete fát, amelyet akkor kapunk, ha a 13.1. ábrán látható fára alkalmazzuk a FÁBA-BESZÚR algoritmust a 36 kulcsértékkel. Piros-fekete fát kapunk-e, ha a beszúrt új csúcs színét pirosra színezzük? Mi lesz, ha feketére színezzük a beszúrt csúcsot?

13.1-3. Definiáljuk a ***gyenge piros-fekete fát*** úgy, mint olyan bináris keresőfát, amely kielégíti a piros-fekete tulajdonság 1., 3., 4. és 5. feltételét. Másképpen szólva, a gyökér lehet akár piros, akár fekete. Tekintsünk egy T gyenge piros-fekete fát, amelynek gyökere piros. Piros-fekete fát kapunk-e, ha T gyökerét feketére változtatjuk, de más módosítást nem végzünk a fán?

13.1-4. Tegyük fel, hogy egy piros-fekete fa minden fekete csúcsa „elnyeli” piros színű gyerekeit, úgy, hogy a piros csúcs gyerekei a fekete színű szülő gyerekei lesznek. (Tekintsünk el attól, hogy mi történik a kulcsokkal.) Mi lehet egy fekete csúcs lehetséges fokszáma a piros gyerekeinek beolvasztása után? Mit tudunk mondani a keletkezett fa leveleinek mélységéről?

13.1-5. Igazoljuk, hogy minden piros-fekete fában a leghosszabb olyan út hossza, amely egy adott x csúcsból valamely levélig vezet, legfeljebb kétszerese az x-től levélig vezető legrövidebb út hosszának.

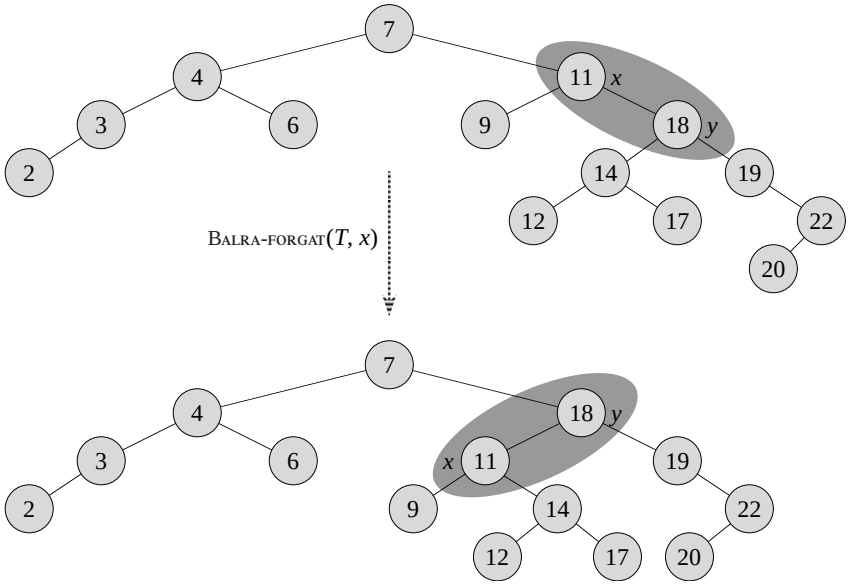
13.1-6. Mennyi a legtöbb csúcsot tartalmazó k fekete-magasságú piros-fekete fa belső csúcsainak száma? Mennyi lehet a legkevesebb érték?

13.1-7. Adjunk meg olyan n csúcsú piros-fekete fát, amelyben a lehető legnagyobb a piros és a fekete belső csúcsok aránya. Mi lesz ez az arány? Milyen fa esetén lesz az arány a legkisebb, és mi lesz ez az érték?

13.2. Forgatások

Minden n csúcsú piros-fekete fa esetén a FÁBA-BESZÚR és FÁBÓL-TÖRÖL műveletek időigénye $O(\lg n)$. Az algoritmusok azonban módosítják a fa szerkezetét, így az eredményül kapott fa nem feltétlenül teljesíti a 13.1. alfejezetben adott piros-fekete tulajdonságot. A piros-fekete tulajdonság helyreállítása végett meg kell változtatnunk bizonyos csúcsok színét, és esetleg módosítani kell a fa szerkezetét is.

Olyan ***forgatás***műveletet alkalmazunk a fára, amely lokális művelet, megváltoztatja a fa szerkezetét, de megőrzi a bináris keresőfa tulajdonságot. A 13.2. ábra mutatja a kétféle, balra, illetve jobbra forgatás műveletek hatását. Feltételezzük, hogy az x csúcsra végrehaj-



13.3. ábra. Egy példa arra, hogyan módosítja a BALRA-FORGAT(T, x) eljárás a bináris keresőfát. A NIL leveleket nem tüntettük fel. A két fa inorder bejárása ugyanazt a kulcssorozatot adja.

tandó balra forgatás előtt az x csúcs jobb gyereke nem $nil[T]$; x bármely olyan csúcsa lehet a fának, amelynek jobb gyereke nem $nil[T]$. A balra forgatás az $x-y$ kapcsolat körül történik. Ennek hatására y lesz a részfa új gyökere, x lesz y bal gyereke és y bal gyereke lesz x jobb gyereke.

A BALRA-FORGAT algoritmus feltételezi, hogy $jobb[x] \neq nil[T]$, továbbá hogy a gyökér szülője $nil[T]$.

BALRA-FORGAT(T, x)

```
 1  y ← jobb[x]                          ▷ y beállítása
 2  jobb[x] ← bal[y]                     ▷ x jobb részfája legyen y bal részfája
 3  if bal[y] ≠ nil[T]
 4     then szülő[bal[y]] ← x
 5  szülő[y] ← szülő[x]                  ▷ legyen x szülője y
 6  if szülő[x] = nil[T]
 7     then gyökér[T] ← y
 8     else if x = bal[szülő[x]]
 9             then bal[szülő[x]] ← y
10             else jobb[szülő[x]] ← y
11  bal[y] ← x                           ▷ x legyen y bal részfája
12  szülő[x] ← y
```

A 13.3. ábra szemlélteti a BALRA-FORGAT műveletet. A JOBBRA-FORGAT művelet hasonlóan valósítható meg. Mindkét művelet időigénye $O(1)$. Forgatás csak mutató értékeket változtat, a csúcsok minden más mezője változatlan marad.

Gyakorlatok

13.2-1. Írjuk meg a JOBBRA-FORGAT művelet algoritmusát.

13.2-2. Indokoljuk meg, hogy minden n-csúcsú bináris keresőfának pontosan $n-1$ lehetséges forgatása van.

13.2-3. Legyenek a, b és c a 13.2. ábra bal oldalán látható fa α, β, illetve γ részfáinak tetszőleges csúcsai. Hogyan változik e három csúcs magassága, ha balra forgatást hajtunk végre az x csúcsra?

13.2-4. Mutassuk meg, hogy bármely n-csúcsú bináris keresőfa $O(n)$ forgatással átalakítható bármely másik bináris keresőfává. (*Útmutatás.* Először mutassuk meg, hogy legfeljebb $n-1$ forgatással a fa átalakítható jobbra tartó lánccá.)

13.2-5.$\star$ Azt mondjuk, hogy egy T_1 bináris keresőfa jobbra-alakítható a T_2 bináris keresőfává, ha T_2 megkapható T_1-ből JOBBRA-FORGAT műveletek alkalmazásával. Adjunk olyan T_1 és T_2 fát, hogy T_1 nem jobbra-alakítható T_2-vé. Ezután mutassuk meg, hogy ha T_1 jobbra-alakítható T_2-vé, akkor átalakítható $O(n^2)$ JOBBRA-FORGAT művelet alkalmazásával.

13.3. Beszúrás

Piros-fekete fa bővítése egy új csúccsal elvégezhető $O(\lg n)$ időben, ha a fa n csúcsot tartalmaz. A FÁBA-BESZÚR (12.3. alfejezet) eljárás kissé módosított változatával bővítjük a T fát a z csúccsal, mintha a fa közönséges bináris keresőfa lenne, majd a z csúcs színét pirosra állítjuk. A piros-fekete tulajdonság biztosítására meghívjuk a PF-FÁBA-BESZÚR-JAVÍT segéd eljárást, amely csúcsok átszínezését és forgatásokat végez. A PF-FÁBA-BESZÚR(T, z) eljáráshívás a T fába szúrja a z csúcsot, amelynek kulcsmezőjét előzetesen már beállítottuk.

```
PF-FÁBA-BESZÚR(T, z)
 1  y ← nil[T]
 2  x ← gyökér[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5          if kulcs[z] < kulcs[x]
 6             then x ← bal[x]
 7             else x ← jobb[x]
 8  szülő[z] = y
 9  if y = nil[T]
10     then gyökér[T] ← z
11     else if kulcs[z] < kulcs[y]
12             then bal[y] ← z
13             else jobb[y] ← z
14  bal[z] ← nil[T]
15  jobb[z] ← nil[T]
16  szín[z] ← PIROS
17  PF-FÁBA-BESZÚR-JAVÍT(T, z)
```

Négy különbség van a FÁBA-BESZÚR és a PF-FÁBA-BESZÚR eljárás között. Először, a FÁBA-BESZÚR eljárásban szereplő minden *nil* érték helyett a *nil*$[T]$ érték van. Másodszor, a 14–15.

sorokban *bal*[*z*] és *jobb*[*z*] értékét *nil*[*T*]-re állítjuk, hogy fenntartsuk a fa helyes szerkezetét. Harmadszor, *z* színét pirosra állítjuk be a 17. sorban. Negyedszer, mivel *z* színének pirosra állítása megsértheti valamelyik piros-fekete tulajdonságot, ezért a 17. sorban meghívjuk a PF-FÁBA-BESZÚR-JAVÍT eljárást a piros-fekete tulajdonság helyreállítására.

PF-FÁBA-BESZÚR-JAVÍT(T, z)

```
 1  while szín[szülő[z]] = PIROS
 2        do if szülő[z] = bal[szülő[szülő[z]]]
 3              then y ← jobb[szülő[szülő[z]]]
 4                   if szín[y] = PIROS
 5                      then szín[szülő[z]] ← FEKETE                    ▷ 1. eset
 6                           szín[y] ← FEKETE                           ▷ 1. eset
 7                           szín[szülő[szülő[z]]] ← PIROS              ▷ 1. eset
 8                           z ← szülő[szülő[z]]                        ▷ 1. eset
 9                      else if z = jobb[szülő[z]]
10                              then z ← szülő[z]                       ▷ 2. eset
11                                   BALRA-FORGAT((T, z)                ▷ 2. eset
12                              szín[szülő[z]] ← FEKETE                 ▷ 3. eset
13                              szín[szülő[szülő[z]]] ← PIROS           ▷ 3. eset
14                              JOBBRA-FORGAT(T, szülő[szülő[z]])       ▷ 3. eset
15                      else ugyanaz, mint a then ág, csak a „bal” és „jobb” felcserélve
16  szín[gyökér[T]] ← FEKETE
```

A PF-FÁBA-BESZÚR-JAVÍT eljárás működésének megértéséhez a kódot három fő részre bontva vizsgáljuk. Először, meghatározzuk, hogy a piros-fekete tulajdonság milyen sérülést szenvedett a *z* csúcs beszúrása és pirosra festése következtében. Másodszor, megvizsgáljuk, hogy mi a célja az 1–15. sorokban leírt ***while*** ciklusnak. Végül azt vizsgáljuk, hogy a **while** ciklus milyen három[1] esetre bontható, és ezek az esetek hogyan teljesítik a kitűzött célt. A 13.4. ábra a PF-FÁBA-BESZÚR-JAVÍT eljárás működését mutatja be egy példa piros-fekete fára.

A piros-fekete tulajdonságok közül melyek sérülhettek meg PF-FÁBA-BESZÚR-JAVÍT eljárás hívása előtt? Az 1. és a 3. tulajdonság bizonyára továbbra is teljesül, mert az új piros csúcs mindkét gyereke a *nil*[*T*] őrszem. Az 5. tulajdonság, amely azt mondja, hogy a fekete csúcsok száma azonos minden adott csúcsból induló, és levélig vezető úton, szintén továbbra is teljesül, mert az új piros *z* csúcs fekete őrszem csúcs helyébe kerül, és mindkét gyereke az őrszem lesz. Tehát legfeljebb a 2. tulajdonság, amely szerint a gyökér színe fekete, és a 4. tulajdonság, amely szerint piros csúcsnak nem lehet piros színű gyereke, nem teljesül. Mindkét esetben az okozhatja a sérülést, hogy *z* színét pirosra állítottuk. A 2. tulajdonság akkor sérül, ha *z* a gyökér lesz, a 4. pedig akkor, ha *z* szülője is piros. A 13.4.(a) ábra azt az esetet mutatja, amikor a 4. tulajdonság sérül *z* beszúrása után.

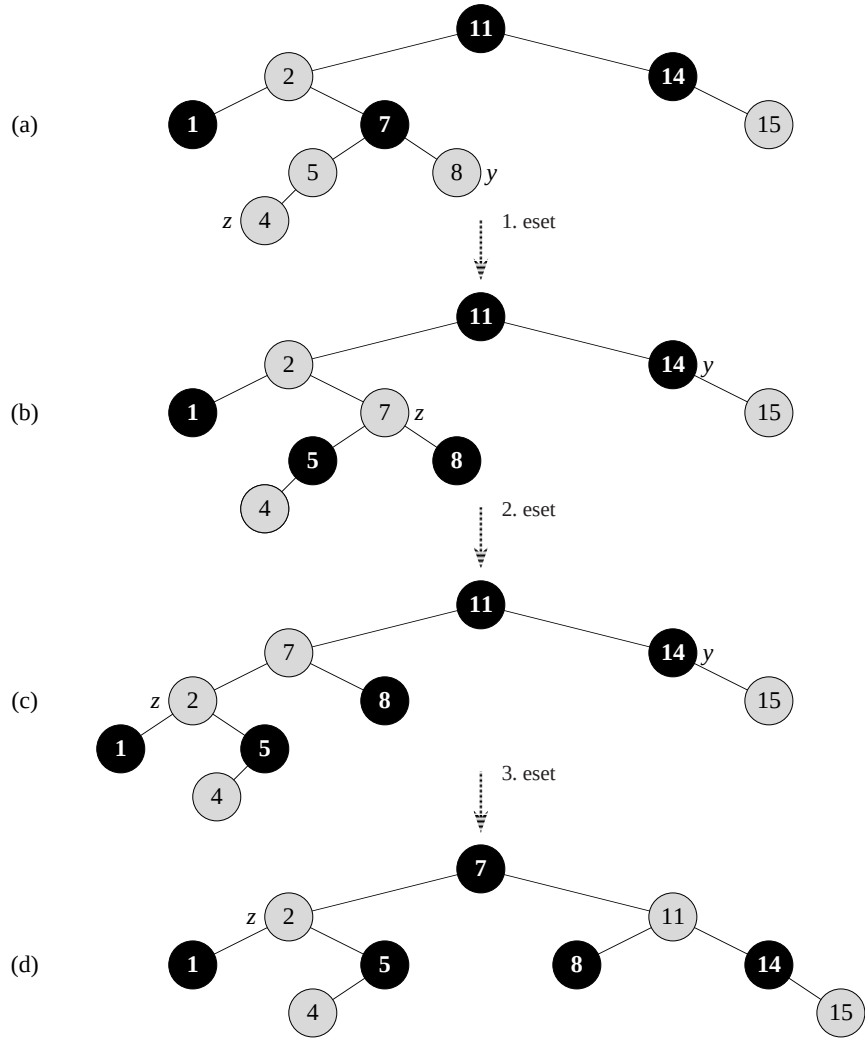
A 11–15. sorokban leírt ***while*** ciklus teljesíti a következő háromrészes invariánst.

A ciklusmag minden végrehajtása előtt teljesül:

a. *z* színe piros.

b. Ha *szülő*[*z*] a gyökér, akkor *szülő*[*z*] színe fekete.

[1] A második eset után következhet a 3. eset, tehát ezek nem kölcsönösen kizáró esetek.



13.4. ábra. A PF-FÁBA-BESZÚR-JAVÍT művelet hatása. **(a)** A z csúcs a beszúrással keletkező új csúcs. Mivel mind z, mind *szülő*[z] színe is piros, a 4. tulajdonság sérül. Mivel z nagybátyja az y csúcs piros, ezért az 1. eset alkalmazandó. Ennek hatására csúcsok átszíneződnek, és a z mutató a fában feljebb kerül, az eredményt a **(b)** rész mutatja. z és a szülője ismét piros, de z nagybátyja az y csúcs fekete. Mivel z jobb gyereke *szülő*[z]-nek, így a 2. esetet kell alkalmazni. Balra forgatást kell végezni, aminek a hatását a **(c)** rész mutatja. Most már z bal gyereke a szülőjének, így a 3. esetről van szó. Jobbra forgatással alakul ki a **(d)** részben látható fa, amely már szabályos piros-fekete fa.

c. Ha sérül valamelyik piros-fekete tulajdonság, akkor csak egy sérülhet, vagy a 2., vagy a 4. tulajdonság. Ha a 2. tulajdonság sérül, akkor ez azért van, mert z a gyökér, és a színe piros. Ha a 4. tulajdonság sérül, akkor ez azért van, mert mind z, mind *szülő*[z] színe piros.

Annak igazolására, hogy a PF-FÁBA-BESZÚR-JAVÍT helyreállítja a piros-fekete tulajdonságot, a (c) rész fontosabb, mint az (a) és a (b) rész, amelyeket arra használunk, hogy

megértsük a kódban előforduló eseteket. Mivel a z csúcsra és annak szomszédos csúcsaira összpontosítunk, fontos tudnunk az (a) részből, hogy z piros színű. A (b) részt használjuk annak megmutatására, hogy a *szülő*[*szülő*[z]] csúcs létezik, amikor a 2., 3., 7., 8., 13. és 14. sorokban hivatkozunk rá.

Emlékeztetünk, hogy meg kell mutatni, hogy a ciklusinvariáns teljesül a ciklusmag első végrehajtása előtt, hogy teljesül a ciklusmag minden végrehajtására, továbbá a ciklusinvariáns hasznos tulajdonságot fejez ki a ciklus befejezése után.

Az előkészítés és a befejezés elemzésével kezdjük. Aztán miután megvizsgáltuk a ciklusmag részletes működését, megmutatjuk, hogy a ciklusmag minden végrehajtása teljesíti a ciklusinvariánst. Azt is kimutatjuk, hogy a ciklusmag minden végrehajtásának két lehetséges kimenete adódik: a z mutató vagy feljebb kerül a fában, vagy valamilyen forgatás hajtódik végre, és a ciklus befejeződik.

Teljesül: A ciklus első végrehajtása előtt olyan piros-fekete fával indultunk, amelynek egyik tulajdonsága sem sérült, aztán bővítettük ezt a fát a z piros csúccsal. Megmutatjuk, hogy a ciklusinvariáns mindhárom állítása teljesül, amikor a PF-FÁBA-BESZÚR-JAVÍT eljárás meghívódik.

a. PF-FÁBA-BESZÚR-JAVÍT hívásakor z a beszúrt új csúcs, amelynek színe piros.

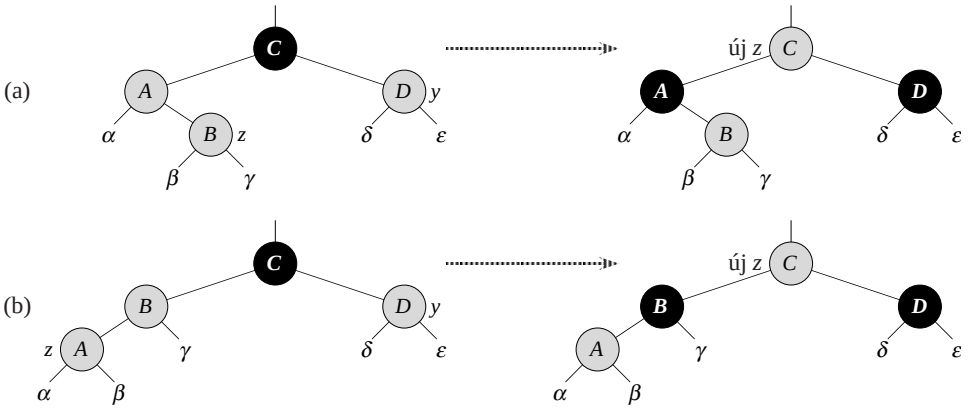
b. Ha *szülő*[z] a gyökér, akkor *szülő*[z] színe fekete volt kezdéskor, és nem változott PF-FÁBA-BESZÚR-JAVÍT hívása előtt.

c. Azt már láttuk, hogy az 1., 3. és 5. tulajdonság teljesül, amikor meghívódik PF-FÁBA-BESZÚR-JAVÍT.
   Ha sérül a 2. tulajdonság, akkor szükségképpen a piros gyökér az új z csúcs, amely az egyetlen belső csúcs a fában. Mivel z-nek a szülője és mind a két gyereke őrszem, aminek a színe fekete, így a 4. tulajdonság nem sérül. Tehát a teljes fában csak a 2. tulajdonság az egyetlen piros-fekete tulajdonság, amely sérült.
   Ha a 4. tulajdonság sérül, akkor mivel z gyerekei a fekete őrszemek, és z beszúrása előtt nem sérült egyetlen tulajdonság sem, ezért a sérülés oka csak az lehet, hogy mind z, mind *szülő*[z] piros. Továbbá, más piros-fekete tulajdonság nem sérül.

Befejeződik: Amikor a ciklus befejeződik, akkor ennek az az oka, hogy *szülő*[z] színe fekete. (Ha z a gyökér, akkor *szülő*[z] az őrszem *nil*[T], amely fekete.) Tehát a 4. tulajdonság nem sérül az ismétlés befejezése után. A ciklusinvariáns miatt csak legfeljebb a 2. tulajdonság sérülhet. A 16. sor helyreállítja ezt a tulajdonságot is, tehát PF-FÁBA-BESZÚR-JAVÍT befejeződésekor minden piros-fekete tulajdonság teljesül.

Megmarad: Ténylegesen hat esetet kell tekinteni a ***while*** ciklusban, attól függően, hogy z szülője, a *szülő*[z] bal, avagy jobb gyereke-e z nagyszülőjének, a *szülő*[*szülő*[z]] csúcsnak, amelyet a 2. sorban határozunk meg. Csak arra az esetre írtuk meg a kódot, amikor *szülő*[z] bal gyerek. A *szülő*[*szülő*[z]] csúcs létezik a ciklusinvariáns (b) része miatt, és ha *szülő*[z] a gyökér, akkor *szülő*[z] fekete. Mivel a ciklusmag csak akkor hajtódik végre, ha *szülő*[z] piros, így *szülő*[z] nem lehet a gyökér. Tehát *szülő*[*szülő*[z]] létezik.

Az 1. esetet a 2. és 3. esettől az különbözteti meg, hogy milyen a színe z nagybátyjának. A 3. sor az y mutatót beállítja z-nek a *jobb*[*szülő*[*szülő*[z]]] nagybátyjára, és a 4. sorban lekérdezi ennek színét. Ha y piros, akkor az 1. eset hajtódik végre. Egyébként a vezérlés a 2. és 3. esetre adódik át. Mindhárom esetben z-nek a *szülő*[*szülő*[z]] nagyszülője fekete, mivel *szülő*[z] piros, és a 4. tulajdonság a fában csak z és *szülő*[z] között sérülhet.



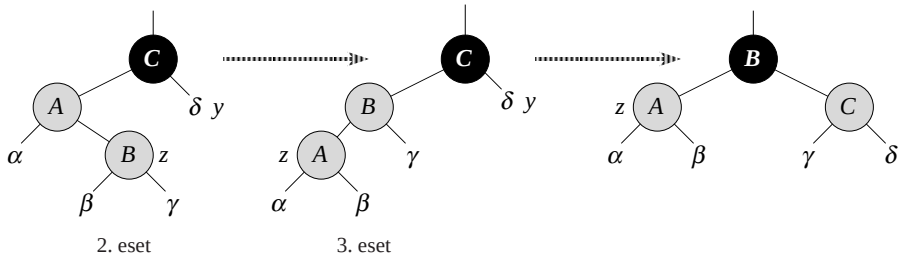
13.5. ábra. A PF-FÁBA-BESZÚR eljárás 1. esete. A 4. tulajdonság nem teljesül, mivel mind *z*, mind *szülő*[*z*] színe piros. Ugyanaz a teendő akár **(a)** *z* jobb gyerek, akár **(b)** *z* bal gyerek. Az $\alpha, \beta, \gamma, \delta$ és ϵ részfák mindegyikének a gyökere fekete, és fekete-magasságuk azonos. Az 1. eset csak a csúcsok színét változtatja meg úgy, hogy teljesüljön az 5. tulajdonság. A **while** ciklus ismétlődik *z* nagyapjára, a *szülő*[*szülő*[*z*]] csúcsra mint új *z* csúcsra. Ezután a 4. tulajdonság csak az új piros *z* apjára nem teljesülhet, ha az is piros.

1. eset: z nagybátyja, y piros

A 13.5. ábra mutatja az 1. eset helyzetét (5–8. sorok). Az 1. eset akkor hajtódik végre, ha mind *szülő*[*z*], mind *y* piros. Mivel *szülő*[*szülő*[*z*]] fekete, így feketére színezhetjük mind a *szülő*[*z*], mind *y* csúcsot, ezzel kiküszöböljük, hogy *z* és *szülő*[*z*] piros volt, továbbá pirosra színezzük a *szülő*[*szülő*[*z*]] csúcsot, tehát biztosítjuk az 5. tulajdonság teljesülését. Ezután ismételjük a ***while*** ciklust a *z* csúcsot *szülő*[*szülő*[*z*]]-re változtatva. Tehát a *z* mutató két szinttel feljebb kerül a fában.

Megmutatjuk, hogy az 1. eset végrehajtása után a ciklusinvariáns teljesül a ciklusmag következő végrehajtása előtt. *z*-vel jelöljük az aktuális csúcsot az aktuális ismétlésben, és $z' = $ *szülő*[*szülő*[*z*]]-vel a *z* következő ismétlésbeli értékét az 1. sorban végrehajtott teszteléskor.

a. Mivel az aktuális ismétlés pirosra színezi a *szülő*[*szülő*[*z*]] csúcsot, ezért z' piros lesz a következő ismétlés kezdetekor.
b. A *szülő*[z'] csúcs a *szülő*[*szülő*[*szülő*[*z*]]] csúcs az aktuális ismétlésben, amelynek színe nem változik. Ha ez a gyökér csúcs, akkor színe fekete volt az ismétlés előtt, és fekete marad a következő ismétlés megkezdésekor.
c. Már korábban beláttuk, hogy az 1. eset megőrzi az 5. tulajdonságot, és nyilvánvalóan nem okozza az 1. és a 3. tulajdonság sérülését.
   Ha a z' csúcs a gyökér a következő ismétlés megkezdésekor, akkor az 1. eset végrehajtása helyreállítja az egyetlen 4. tulajdonság sérülését ebben az ismétlésben. Mivel z' piros, és ez a gyökér, a 2. tulajdonság lesz az egyetlen, amely sérül, és a sérülést a z' csúcs okozza.
   Ha z' nem a gyökércsúcs a következő ismétlés kezdetekor, akkor az 1. eset végrehajtása miatt nem sérül a 2. tulajdonság. Az 1. eset korrigálja az egyetlen 4. tulajdonság sérülését, és befejeződik az ismétlés. Ezután pirosra festi z'-t, és meghagyja a *szülő*[z']. Ha *szülő*[z'] piros, akkor z' átszínezése a 4. tulajdonság sérülését okozza z' és *szülő*[z'] vonatkozásában.



13.6. ábra. A PF-FÁBA-BESZÚR-JAVÍT eljárás 2. és 3. esete. Mint az 1. esetben, a 3. tulajdonság nem teljesül, mert mind z, mind *szülő*[z] színe piros. Az α, β, γ és δ részfák mindegyikének gyökere fekete (α, β és γ esetében a 4. tulajdonság miatt, és δ esetében azért, mert különben az 1. eset adódna), és fekete-magasságuk azonos. A 2. esetből balra forgatással kapjuk a 3. esetet, amely megőrzi a 5. tulajdonságot. A 3. esetben néhány csúcs színét megváltoztatjuk, majd forgatással elérjük a 5. tulajdonság teljesülését. Ezután a **while** ciklus befejeződik, mert a 4. tulajdonság teljesül, nem lesz két egymást követő, szülő-gyerek csúcs piros.

2. eset: z nagybátyja, y fekete, és z jobb gyerek
3. eset: z nagybátyja, y fekete, és z bal gyerek

A 2. és 3. esetben z nagybátyja, y fekete. A két esetet az különbözteti meg, hogy z a *szülő*[z] csúcsnak bal, avagy jobb gyereke. A 10–11. sorok tartalmazzák a 2. esetet, amit a 13.6. ábra mutat a 3. esettel együtt. A 2. esetben a z csúcs jobb gyereke a szülőjének. Ekkor egy balra forgatást hajtunk végre, amivel a 3. esetre alakítjuk át a fát (12–14. sorok), amikor is a z csúcs bal gyerek. Mivel mind z, mind *szülő*[z] piros, ezért a forgatás nem változtatja sem a fekete-magasságot, sem az 5. tulajdonság teljesülését. Akár közvetlenül jutunk a 3. esetbe, akár a 2. eseten keresztül, z nagybátyja y fekete, mivel egyébként az 1. esetet hajtanánk végre. Ráadásul a *szülő*[*szülő*[z]] csúcs létezik, mivel már beláttuk, hogy ez a csúcs létezik a 2. és 3. sor végrehajtásakor, majd z-vel egyet felfelé léptünk a 10. sorban, aztán egyet lefelé a 11. sorban, így a *szülő*[*szülő*[z]] értéke változatlan marad. A 3. esetben néhány csúcsot átszínezünk, és jobbra forgatást hajtunk végre, aminek hatására az 5. tulajdonság teljesül, és ezzel készen is vagyunk, mert nincs további két egymást követő piros csúcs. A ***while*** ciklus magja többször már nem hajtódik végre, mert *szülő*[z] színe fekete.

Most megmutatjuk, hogy a 2. és 3. eset megőrzi a ciklusinvariánst. (Amint éppen most mutattuk meg, hogy *szülő*[z] színe fekete lesz az 1. sorbeli következő teszteléskor, és a ciklusmag ezután már nem hajtódik végre.)

a. A 2. eset z értékét *szülő*[z]-re változtatja, amely piros. A 2. és 3. eset további változtatást nem végez z-n és annak színén.
b. A 3. eset *szülő*[z] színét feketére változtatja, így ha *szülő*[z] a gyökér a következő ismétlés kezdetekor, akkor színe fekete.
c. Mint az 1. esetben, az 1., 3. és 5. tulajdonság megőrződik a 2. és 3. esetben is. Mivel z nem a gyökér a 2. és a 3. esetben, így a 2. tulajdonság nem sérülhet. A 2. és 3. eset nem eredményezheti a 2. tulajdonság sérülését, mivel az egyetlen csúcs, amely piros színű lesz, egy fekete csúcs gyerekévé lesz a 3. esetbeli forgatással.
   A 2. és 3. eset kijavítja az egyedül sérült 4. tulajdonságot, és nem eredményez további sérülést.

Azzal, hogy megmutattuk, hogy a ciklusinvariáns megőrződik a ciklusmag minden végrehajtásakor, bebizonyítottuk, hogy a PF-FÁBA-BESZÚR-JAVÍT eljárás helyreállítja a piros-fekete tulajdonságot.

Elemzés

Mi a PF-FÁBA-BESZÚR algoritmus futási ideje? Mivel minden n csúcsú piros-fekete fa magassága $O(\lg n)$, így a PF-FÁBA-BESZÚR eljárás 1–16. sorainak végrehajtása $O(\lg n)$ időt igényel. A PF-FÁBA-BESZÚR-JAVÍT eljárásban a **while** ciklusmag csak az 1. esetben ismétlődik, amikor is z két szinttel feljebb kerül a fában. Tehát a **while** ciklus teljes végrehajtási ideje $O(\lg n)$. Így a PF-FÁBA-BESZÚR algoritmus $O(\lg n)$ idejű. Vegyük észre, hogy az algoritmus legfeljebb kétszer végez forgatást, mivel a **while** ciklus befejeződik a 2. vagy 3. eset végrehajtása után.

Gyakorlatok

13.3-1. A PF-FÁBA-BESZÚR algoritmus 16. sorában az új z csúcs színét pirosra állítjuk. Vegyük észre, hogy ha z színét feketére állítanánk, akkor nem sérülne a piros-fekete fák 4. tulajdonsága. Miért nem fekete színt kap az új z csúcs?

13.3-2. Rajzoljuk meg azt a piros-fekete fát, amely a kezdetben üres fából úgy keletkezik, hogy egymás után bővítjük a fát a 41, 38, 31, 12, 19, 8 kulcsokkal.

13.3-3. Tegyük fel, hogy a 13.5. és 13.6. ábrákon az $\alpha, \beta, \gamma, \delta$ és ϵ részfák fekete-magassága k. Címkézzük meg mindkét ábrán a részfákat a fekete-magasságukkal, és igazoljuk, hogy a végrehajtott transzformációk megőrzik a 5. tulajdonságot.

13.3-4. Tanár professzor azon aggódik, hogy a PF-FÁBA-BESZÚR-JAVÍT eljárás a *szín*[*nil*[T]] értékét PIROS-ra állíthatja, és így a ciklus nem fejeződik be, amikor az 1. sorbeli tesztelésnél z értéke a gyökér. Mutassuk meg, hogy a professzor aggodalma alaptalan, mert a PF-FÁBA-BESZÚR-JAVÍT eljárás sohasem állítja *szín*[*nil*[T]] értékét PIROS értékre.

13.3-5. Legyen T olyan piros-fekete fa, amely az üres fából n csúcs bővítésével keletkezik a PF-FÁBA-BESZÚR eljárással. Igazoljuk, hogy ha $n > 1$, akkor T-nek van legalább egy piros csúcsa.

13.3-6. Hogyan lehet hatékonyan megvalósítani a PF-FÁBA-BESZÚR algoritmust, ha a fa csúcsai nem tartalmaznak *szülő* mezőt?

13.4. Törlés

Mint a többi alapvető művelet, a törlés is $O(\lg n)$ időt igényel n csúcsot tartalmazó piros-fekete fa esetén. Csúcs törlése piros-fekete fából csak kissé bonyolultabb, mint a beszúrás.

A PF-FÁBÓL-TÖRÖL eljárás a FÁBÓL-TÖRÖL eljárás (12.3. alfejezet) kisebb módosításával kapható. Miután kitöröltük a kívánt csúcsot, végrehajtjuk a PF-FÁBÓL-TÖRÖL-JAVÍT eljárást, amely helyreállítja a fa piros-fekete tulajdonságát csúcsok átszínezésével és forgatásokkal.

A PF-FÁBÓL-TÖRÖL eljárás három ponton különbözik a FÁBÓL-TÖRÖL eljárástól. Először is, a PF-FÁBÓL-TÖRÖL eljárásban minden NIL helyett a *nil*[T] őrszem hivatkozás szerepel. Másodszor, annak eldöntése, hogy z NIL-e a 7. sorban helyettesítődik a *szülő*[x] $\leftarrow$ *szülő*[y] feltétel nélküli értékadással. Tehát, ha az x csúcs a *nil*[T] strázsa, akkor az szülő mezője a kikapcsolandó y csúcs szülőjére mutat. Harmadszor, a 16–17. sorokban meghívjuk

a PF-FÁBÓL-TÖRÖL-JAVÍT eljárást, ha y színe fekete. Ha y piros, akkor a piros-fekete tulajdonság továbbra is teljesül, miután töröltük az y csúcsot a következő okok miatt:

- egyetlen csúcs fekete-magassága sem változik,
- nem keletkezik új piros-piros szülő-gyerek kapcsolat, és
- mivel y nem lehetett a gyökér, ha színe piros volt, ezért a gyökér színe fekete marad.

A PF-FÁBÓL-TÖRÖL-JAVÍT eljárásnak átadott x csúcs értéke kétféle lehet: vagy a ténylegesen törlendő y csúcsnak az egyetlen nem $nil[T]$ gyereke, ha y-nak volt nem őrszem gyereke, vagy a $nil[T]$ őrszem, ha y-nak nem volt gyereke. Az utóbbi esetben a 7. sorban végrehajtott feltétlen értékadás biztosítja, hogy x szülője ekkor is a törlendő y csúcs szülője lesz, akár belső csúcs x, akár a $nil[T]$ őrszem.

PF-FÁBÓL-TÖRÖL(T, z)

```
 1  if bal[z] = nil[T] or jobb[z] = nil[T]
 2     then y ← z
 3     else y ← FA-KÖVETKEZŐ(z)
 4  if bal[y] ≠ nil[T]
 5     then x ← bal[y]
 6     else x ← jobb[y]
 7  szülő[x] ← szülő[y]
 8  if szülő[y] = nil[T]
 9     then gyökér[T] ← x
10     else if y = bal[szülő[y]]
11             then bal[szülő[y]] ← x
12             else jobb[szülő[y]] ← x
13  if y ≠ z
14     then kulcs[z] ← kulcs[y]
15          ▷ Másoljuk át y többi mezőjét is z-be
16  if szín[y] = FEKETE
17     then PF-FÁBÓL-TÖRÖL-JAVÍT(T, x)
18  return y
```

Vizsgáljuk meg, hogy a PF-FÁBÓL-TÖRÖL-JAVÍT hogyan állítja helyre a fa piros-fekete tulajdonságát.

Ha a ténylegesen kitörlendő y csúcs színe fekete, akkor három probléma keletkezhet. Először, ha y a gyökér volt, és piros gyereke lesz az új gyökér, mert sérül a 2. tulajdonság. Másodszor, ha mind x, mind $szülő[y]$ (amely most megegyezik $szülő[x]$-nal) piros volt, mert sérül a 4. tulajdonság. Harmadszor, y eltávolítása után minden olyan út, amely az y csúcson keresztül haladt, eggyel kevesebb fekete csúcsot fog tartalmazni. Tehát minden olyan csúcsra sérül az 5. tulajdonság, amely őse y-nak. Ezt a problémát kiküszöbölhetjük, ha az x csúcsot úgy tekintjük, hogy egy extra fekete értéket tartalmaz. Vagyis, ha minden olyan út fekete csúcsainak számához hozzáadnánk egyet, amely keresztül megy az x csúcson, akkor teljesülne az 5. tulajdonság. Amikor eltávolítjuk a fekete y csúcsot, akkor fekete értékét továbbadjuk az x fiának. Az egyetlen probléma, hogy x már eredetileg is fekete lehetett, így színe se nem piros, se nem fekete, ami sérti az 1. tulajdonságot. Ehelyett x vagy „kétszeresen

fekete", vagy „piros-és-fekete", ezáltal 2 vagy 1 értéket ad az x-et tartalmazó minden út fekete csúcsszámához. Az x csúcs *szín* mezője továbbra is PIROS (a piros-és-fekete esetben), vagy FEKETE (a kétszeresen fekete esetben). Más szóval, a csúcs extra fekete volta nem a *szín* mezőjének értékéből derül ki, hanem az x mutatóból.

PF-FÁBÓL-TÖRÖL-JAVÍT(T, x)

```
 1  while x ≠ gyökér[T] and szín[x] = FEKETE
 2        do if x = bal[szülő[x]]
 3              then w ← jobb[szülő[x]]
 4                   if szín[w] = PIROS
 5                      then szín[w] ← FEKETE                          ▷ 1. eset
 6                           szín[szülő[x]] ← PIROS                    ▷ 1. eset
 7                           BALRA-FORGAT(T, szülő[x])                 ▷ 1. eset
 8                           w ← jobb[szülő[x]]                        ▷ 1. eset
 9                   if szín[bal[w]] = FEKETE  and szín[jobb[w]] = FEKETE
10                      then szín[w] ← PIROS                           ▷ 2. eset
11                           x ← szülő[x]                              ▷ 2. eset
12                      else if szín[jobb[w]] = FEKETE
13                              then szín[bal[w]] ← FEKETE             ▷ 3. eset
14                                   szín[w] ← PIROS                   ▷ 3. eset
15                                   JOBBRA-FORGAT(T, w)               ▷ 3. eset
16                                   w ← jobb[szülő[x]]                ▷ 3. eset
17                           szín[w] ← szín[szülő[x]]                  ▷ 4. eset
18                           szín[szülő[x]] ← FEKETE                   ▷ 4. eset
19                           szín[jobb[w]] ← FEKETE                    ▷ 4. eset
20                           BALRA-FORGAT(T, szülő[x])                 ▷ 4. eset
21                           x ← gyökér[T]                             ▷ 4. eset
22              else ( ugyanaz, mint a then ág, csak a „bal" és „jobb" felcserélve)
23  szín[x] ← FEKETE
```

A PF-FÁBÓL-TÖRÖL-JAVÍT eljárás helyreállítja az 1., 2. és 3. tulajdonságot. A 13.4-1. és 13.4-2. gyakorlatok azt kérik, hogy mutassuk meg, hogy az eljárás helyreállítja a 2. és 4. tulajdonságot, így az alfejezet további részében csak az 1. tulajdonságra összpontosítunk. Az 1–22. sorok alkotta **while** ciklus feladata, hogy az extra fekete csúcsot a fában felfelé mozgassa, amíg

1. x olyan csúcsra mutat, amely piros-és-fekete színű, ekkor a 23. sorban (egyszeresen) feketére színezzük, vagy
2. x a gyökérre mutat, és ekkor az extra fekete egyszerűen elhagyható, vagy
3. alkalmas forgatásokat és átszínezéseket hajthatunk végre.

A **while** ciklusmagban x mindig olyan csúcsra mutat, amely nem a gyökér és kétszeresen fekete. A 2. sorban eldöntjük, hogy x az apjának bal avagy jobb gyereke. (Az eljárásban csak azt az esetet dolgoztuk ki, amikor x bal gyerek; a másik eset – 22. sor – hasonló.) A w mutató mindig az x csúcs testvérére mutat. Mivel az x csúcs kétszeresen fekete, így w nem

lehet *nil*[T], mert egyébként a *szülő*[x] csúcsból induló, (egyszeresen fekete) w levélig tartó úton kevesebb fekete csúcs lenne, mint a *szülő*[x]-től x-ig haladó úton.

Az algoritmusban megkülönböztetett négy[2] esetet a 13.7. ábra szemlélteti. Mielőtt az esetek részletes vizsgálatába kezdenénk, megvizsgáljuk általánosan, hogyan mutatható meg, hogy az alkalmazott átalakítások mindegyik esetben megőrzik a 5. tulajdonságot. A kulcs ötlet az, hogy mindegyik esetben a feltüntetett részfa gyökerétől az $\alpha, \beta, \ldots, \zeta$ részfákhoz vezető úton található fekete csúcsok száma (beleértve a gyökeret is és x extra feketeségét is) a transzformáció hatására nem változik. Tehát ha az 5. tulajdonság teljesül a transzformáció előtt, akkor teljesülni fog utána is. Például a 13.7(a) ábrán, amely a 1. esetet szemlélteti, a gyökértől akár az α, akár a β részfához vezető úton a fekete csúcsok száma 3, a transzformáció előtt is és utána is. (Ne feledjük, x kétszeresen fekete.) Hasonlóan, a fekete csúcsok száma a gyökértől a γ, δ, ϵ és ζ részfákig vezető utakon 2, a transzformáció előtt is és utána is. A 13.7(b) ábrán a számoláskor figyelembe kell venni a részfa gyökerének c színét is, ami lehet piros is és fekete is. Ha a *pszín*(PIROS) = 0 és *pszín*(FEKETE) = 1 definíciót alkalmazzuk, akkor a fekete csúcsok száma a gyökértől az α részfáig 2 + *pszín*(c), a transzformáció előtt is és utána is. Ekkor, a transzformáció után az új x csúcs *szín* mezőjének értéke c, de a csúcs valójában vagy piros-és-fekete (ha c = PIROS), vagy kétszeresen fekete (ha c = FEKETE). A többi eset is hasonlóan bizonyítható (lásd 13.4-5 gyakorlat).

1. eset: x-nek a w testvére piros

Az 1. eset (5–8. sorok a PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban, és a 13.7(a) ábra) akkor áll elő, ha az x csúcs testvére, a w csúcs piros színű. Mivel w-nek biztosan van fekete gyereke, felcserélhetjük a w és a *szülő*[x] csúcsok színét, majd balra forgathatjuk a *szülő*[x] csúcsot anélkül, hogy a piros-fekete tulajdonságok megmaradnának. Az x csúcs új w testvére – amely a forgatás előtt w egyik gyereke volt – most fekete, tehát az 1. esetet a 2., 3. vagy 4. esetre transzformáltuk.

A 2., 3. és 4. esetek akkor fordulnak elő, amikor a w csúcs fekete; az eseteket w gyerekeinek színe alapján különböztetjük meg.

2. eset: x-nek a w testvére fekete, és w mindkét gyereke fekete

A 2. esetben (10–11. sorok a PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban, és a 13.7(b) ábra) a w csúcs mindkét gyereke fekete. Mivel w szintén fekete, így elvehetünk egy feketét x-től és w-től, ezzel x egyszeresen fekete, w pedig piros lesz. Az x-től és w-től történt elvételt szeretnénk kiegyenlíteni azzal, hogy *szülő*[x]-nek adunk egy extra feketét, amely eredetileg lehetett akár piros, akár fekete. Ezt a **while** ciklusnak x helyett a *szülő*[x] csúcsra történő ismétlésével érjük el. Vegyük észre, hogy ha közvetlenül az 1. eset után hajtjuk végre a 2. eset kódját, akkor az új x csúcs színe piros-és-fekete lesz, mivel az eredeti *szülő*[x] piros volt. Az új x csúcs c színértéke PIROS, és így az ismétlés véget ér az ismétlési feltétel tesztelésével. Az új x csúcs a 23. sorban (egyszeres) fekete színt kap.

3. eset: x-nek a w testvére fekete, w bal gyereke piros, jobb gyereke fekete

A 3. esetben, (13–16. sorok a PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban, és a 13.7(c) ábra) w fekete, w bal gyereke piros és jobb gyereke fekete. Felcserélve w és bal gyereke *bal*[w] színét, majd jobbra forgatást végrehajtva a w csúcsra, a piros-fekete tulajdonság változatlanul

[2]Ugyanúgy, mint PF-BŐVÍT-JAVÍT esetén, az esetek a PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban sem kölcsönösen kizárók.

teljesülni fog. Az x csúcs új w testvére fekete lesz, és w-nek jobb gyereke piros lesz, így az esetet a 4. esetre transzformáltuk.

4. eset: x-nek a w testvére fekete, és w jobb gyereke fekete
A 4. eset (17–21. sorok a PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban, és a 13.7(d) ábra) akkor áll elő, amikor w fekete és ennek jobb gyereke piros. A w, *szülő*[x] és *jobb*[w] csúcsok színének megváltoztatásával majd *szülő*[x] körül balra forgatást végrehajtva eltörölhetjük az x csúcsról az extra feketét anélkül, hogy megsértenénk a piros-fekete tulajdonságot. Ezután x felveszi a gyökér értékét, tehát a **while** ciklus véget ér az ismétlési feltétel tesztelésekor.

Elemzés

Mi a futási ideje a PF-FÁBÓL-TÖRÖL algoritmusnak? Mivel minden n csúcsú piros-fekete fa magassága $O(\lg n)$, így az algoritmus teljes futási ideje a PF-FÁBÓL-TÖRÖL-JAVÍT költsége nélkül $O(\lg n)$. A PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban az 1., 2. és 4. esetben az ismétlés véget ér, miután konstans számú csúcs színét megváltoztattuk, és legfeljebb három forgatást végrehajtottunk. Csak a 2. esetben következhet be a ciklusmag ismétlése, ekkor azonban az x csúcs feljebb kerül a fában, így ez legfeljebb $O(\lg n)$-szer ismétlődhet. A 2. esetben forgatás nem történik. Tehát a PF-FÁBÓL-TÖRÖL-JAVÍT eljárás időigénye $O(\lg n)$, és legfeljebb három forgatást végez, így a PF-FÁBÓL-TÖRÖL algoritmus teljes futási ideje $O(\lg n)$.

Gyakorlatok

13.4-1. Mutassuk meg, hogy a PF-FÁBÓL-TÖRÖL-JAVÍT eljárás végrehajtása után a fa gyökerének színe fekete lesz.

13.4-2. Mutassuk meg, hogy ha a PF-FÁBÓL-TÖRÖL eljárásban mind x, mind *szülő*[y] piros, akkor a PF-FÁBÓL-TÖRÖL-JAVÍT(T, x) eljáráshívás helyreállítja a 4. tulajdonságot.

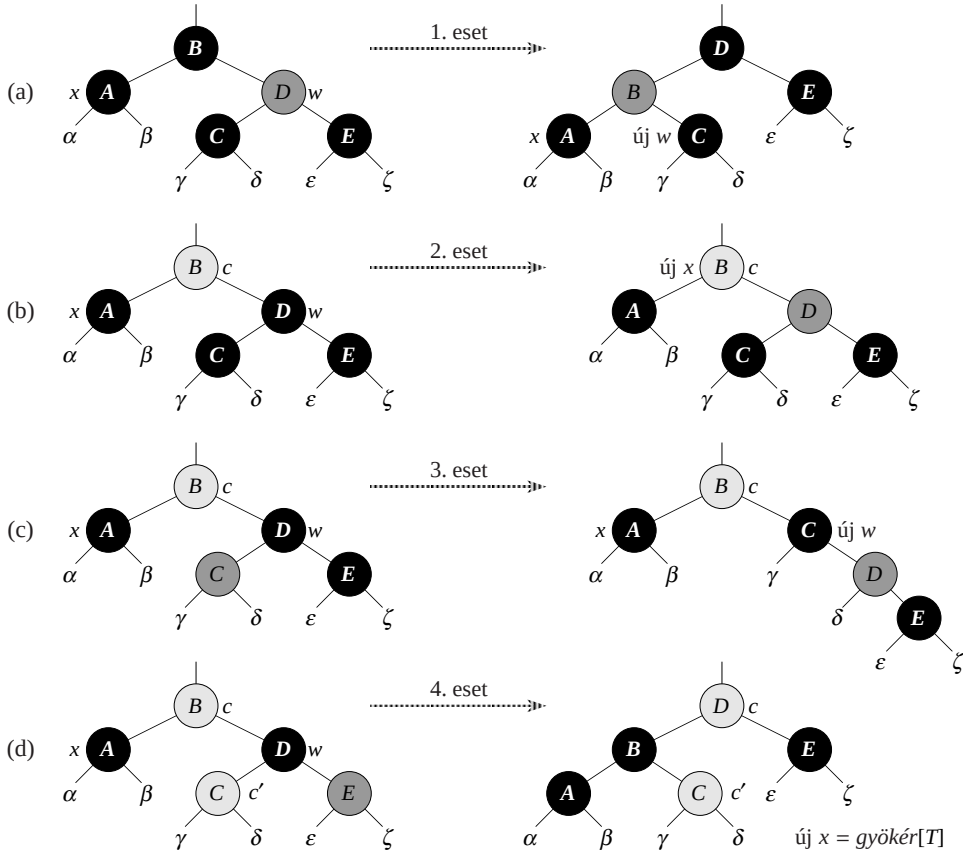
13.4-3. A 13.3-2. gyakorlatban szerepel az a piros-fekete fa, amelyet úgy kapunk, hogy az üres fába egymás után beszúrjuk a $41, 38, 31, 12, 19, 8$ kulcsokat. Rajzoljuk fel azokat a piros-fekete fákat, amelyeket ebből a fából úgy kapunk, hogy egymás után töröljük a $8, 12, 19, 31, 38, 41$ kulcsokat.

13.4-4. A PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban melyik sorokban fordul elő a *nil*[T] őrszem vizsgálata vagy módosítása?

13.4-5. A 13.7. ábrán látható fák mindegyikére számolja ki az $\alpha, \beta, \ldots, \zeta$ részfákhoz vezető utakon található fekete csúcsok számát, és igazoljuk, hogy ez nem változik a végrehajtott transzformáció hatására. Olyan csúcsokra, amelyek jelzett színe c vagy c', használja a *pszín*(c), illetve *pszín*(c') szimbolikus kifejezést a számítás során.

13.4-6. Skelton és Baron professzorok aggódnak, hogy a PF-FÁBÓL-TÖRÖL-JAVÍT eljárásban az 1. eset végrehajtása előtt *szülő*[x] színe esetleg fekete is lehet. Ha a professzoroknak igaza lenne, akkor az 5-6. sorok hibásak lennének. Mutassuk meg, hogy *szülő*[x] színének feketének kell lenni, tehát a professzoroknak nincs miért aggódniuk.

13.4-7. Tegyük fel, hogy egy piros-fekete fát a PF-FÁBA-BESZÚR eljárás bővíti az x csúccsal, majd közvetlenül ez után a PF-FÁBÓL-TÖRÖL eljárás törli ezt a csúcsot. Visszakapjuk-e az eredeti fát? Indokoljuk a választ.

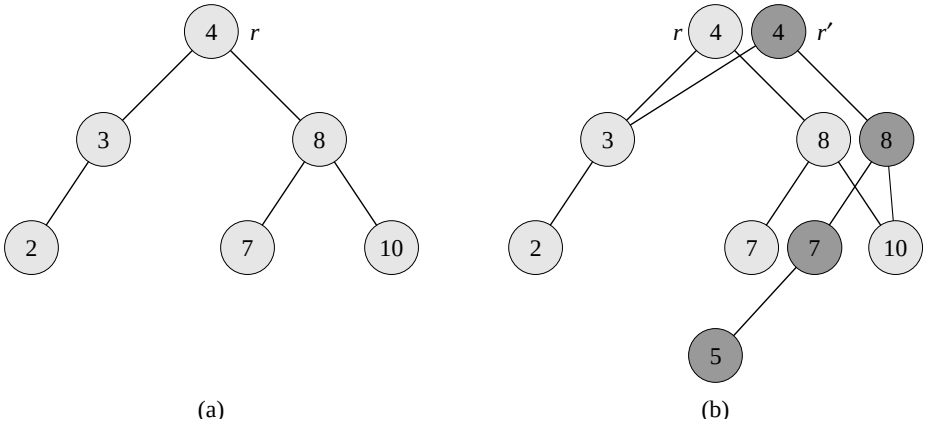


13.7. ábra. A PF-FÁBÓL-TÖRÖL-JAVÍT eljárás **while** ciklusának esetei. Az ábrán a sötét csúcsok színe FEKETE, a satírozottaké piros, a világos csúcsok színe lehet akár PIROS, akár FEKETE, a szín értékét c vagy c-vel jelöljük. Az $\alpha, \beta, \ldots, \zeta$ betűk tetszőleges részfákat jelölnek. Mindegyik esetben az ábra bal oldalán álló fát az algoritmus a jobb oldali fává transzformálja csúcsok átszínezésével és forgatások végrehajtásával. Az x csúcson mindig van egy extra fekete, és színe vagy kétszeresen fekete, vagy piros-és-fekete. A ciklus csak a 2. esetben nem terminál. **(a)** Az 1. esetből a 2., 3. vagy 4. esetet kapjuk a B és D csúcsok színének felcserélésével, és a B csúcs körüli balra forgatással. **(b)** A 2. esetben az x csúcsbeli extra fekete feljebb kerül a fában azáltal, hogy a D csúcs piros színt kap, és x értéke a D csúcsra változik. Ha az 1. eset végrehajtása után érünk a 2. esethez, akkor a ciklus terminál, mert a c szín PIROS. **(c)** A 3. esetből a 4. eset keletkezik a C és D csúcsok színének felcserélésével és jobbra forgatásával. **(d)** A 4. esetben csúcsok színének megváltoztatásával, majd B körül balra forgatást végrehajtva eltörölhetjük az x csúcsról az extra feketét (anélkül, hogy megsértenénk a piros-fekete tulajdonságot). Ezután az ismétlés befejeződik.

Feladatok

13-1. Megmaradó dinamikus halmazok

Előfordulhat, hogy egy algoritmus végrehajtása során valamely dinamikus halmaz módosítás előtti állapotát is meg kell őrizni. Az ilyen halmazt nevezzük ***megmaradó*** halmaznak. Az egyik lehetséges megvalósítása a megmaradó halmaznak az, amikor minden módosítás előtt másolatot készítünk a halmazról. Ez a megoldás azonban lassítja az algoritmust, és a tárigény is növekszik. Ennél jobb megoldás is létezik.



13.8. ábra. (a) Egy, a $2, 3, 4, 7, 8, 10$ kulcsokat tartalmazó bináris keresőfa. **(b)** Az 5 kulcs bővítésével keletkező megmaradó bináris keresőfa. A halmaz legfrissebb változata azokat a kulcsokat tartalmazza, amelyek elérhetők az r' gyökérből, a halmaz előző változata pedig azokat a kulcsokat tartalmazza, amelyek az r gyökérből érhetők el. fa sötét csúcsai az 5 kulcs bővítése során keletkeznek.

Tekintsük az S megmaradó halmazt a BESZÚR, TÖRÖL és KERES műveletekkel. A megvalósításhoz a 13.8(a) ábrán látható bináris keresőfát használunk. A halmaz minden változatához külön gyökér tartozik. Az 5 kulcs bővítése végett létrehozunk egy új csúcsot 5 kulccsal. Ez a csúcs bal gyereke lesz egy új, 7 kulcsot tartalmazó csúcsnak, mivel nem módosíthatjuk azt a régi csúcsot, amely a 7 kulcsot tartalmazza. Hasonlóan, a 7 kulcsot tartalmazó új csúcs bal gyereke lesz egy új, 8 kulcsot tartalmazó csúcsnak, amelynek jobb gyereke a régi, 10 kulcsot tartalmazó csúcs lesz. A 8 kulcsot tartalmazó új csúcs viszont jobb gyereke lesz az új r' gyökérnek, amely kulcsa 4 és bal gyereke a régi, 3 kulcsot tartalmazó csúcs. Tehát a fának csak egy részéről készül másolat, a fa többi régi csúcsát felhasználjuk az új fában is. Ez látható a 13.8(b) ábrán.

Tegyük fel, hogy a fa minden csúcsa tartalmaz *kulcs*, *bal* és *jobb* mezőket, de nincs *szülő* mező. (Lásd még a 13.3-5. gyakorlatot.)

a. Általános megmaradó keresőfa esetén azonosítsuk azokat a csúcsokat, amelyek egy k kulcs bővítése, vagy egy y csúcs törlése esetén megváltoznak.

b. Írjunk olyan MEGMARADÓ-FA-BESZÚR eljárást, amely adott T megmaradó fa, és bővítendő k kulcs esetén eredményül azt a T' új megmaradó fát adja, amely T-nek a k kulccsal való bővítése.

c. Ha a T megmaradó bináris keresőfa magassága h, akkor mekkora a MEGMARADÓ-FA-BESZÚR algoritmus tárigénye és futási ideje? (A tárigény arányos a létrehozott új csúcsok számával.)

d. Tegyük fel, hogy a fa csúcsai *szülő* mezőt is tartalmaznak. Így a MEGMARADÓ-FA-BESZÚR algoritmusnak további másolásokat kell végrehajtani. Bizonyítsuk be, hogy a MEGMARADÓ-FA-BESZÚR algoritmus futási ideje és tárigénye $\Omega(n)$, ha a fa csúcsainak száma n.

e. Mutassuk meg hogyan lehet piros-fekete fát használva megvalósítani a bővítés és a törlés műveleteket úgy, hogy időigényük és tárigényük legrosszabb esetben is $O(\lg n)$ legyen.

13-2. Egyesítés művelet piros-fekete fákon

Az ***egyesítés*** művelet bemenő paramétere két dinamikus halmaz, S_1 és S_2 és egy x elem, amelyekre teljesül, hogy minden $x_1 \in S_1$ és $x_2 \in S_2$ esetén $kulcs[x_1] \leq kulcs[x] \leq kulcs[x_2]$. A művelet eredménye az $S = S_1 \cup \{x\} \cup S_2$ halmaz. A feladat annak vizsgálata, hogyan lehet az egyesítés műveletet megvalósítani piros-fekete fákkal.

a. Adott T piros-fekete fa esetén az $fm[T]$ mezőben tároljuk a fa fekete-magasságát. Igazoljuk, hogy ez az érték fenntartható a PF-FÁBA-BESZÚR és PF-FÁBÓL-TÖRÖL algoritmusokkal úgy, hogy nincs szükség pótlólagos tárra a fa csúcsában, és az id őigény aszimptotikus mértéke nem változik. Mutassuk meg, hogy a T fa bejárása során minden csúcs fekete-magasságát meg tudjuk határozni csúcsonként $O(1)$ id őben.

A PF-FÁKAT-EGYESÍT(T_1, x, T_2) műveletet kívánjuk megvalósítani, amely megszünteti a T_1 és T_2 fákat és a $T = T_1 \cup \{x\} \cup T_2$ piros-fekete fát eredményezi. Legyen n a T_1 és T_2 fák csúcsszámainak összege.

b. Az általánosság megszorítása nélkül feltehetjük, hogy $fm[T_1] \geq fm[T_2]$. Adjunk olyan $O(\lg n)$ idejű algoritmust, amely megad a T_1 fában egy olyan fekete y csúcsot, amely a legnagyobb olyan kulcsú csúcs, amelynek fekete-magassága $fm[T_2]$.

c. Legyen T_y az y gyökerű részfája T_1-nek. Mutassuk meg, hogyan lehet T_y-t helyettesíteni a $T_y \cup \{x\} \cup T_2$ fával $O(1)$ időben, fenntartva a bináris keresőfa tulajdonságot.

d. Milyen színt adjunk az x csúcsnak, hogy teljesüljön az 1., 3. és 5. piros-fekete tulajdonság? Mutassuk meg, hogy a 2. és 4. piros-fekete tulajdonság hogyan állítható helyre $O(\lg n)$ időben.

e. Mutassuk meg, hogy az általánosság megszorítása nélkül teljesül a *b*. rész feltételezése. Írjuk le a szimmetrikus esetet, amikor is $fm[T_1] \leq fm[T_2]$.

f. Igazoljuk, hogy a PF-FÁKAT-EGYESÍT algoritmus futási ideje $O(\lg n)$.

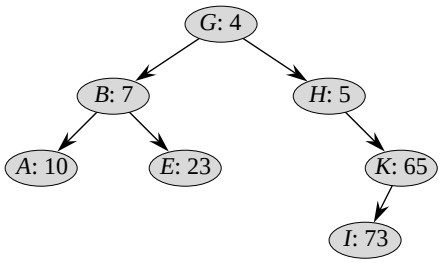
13-3. AVL fák

Az ***AVL fa*** olyan bináris keresőfa, amely ***magasság-kiegyensúlyozott:*** minden x csúcsra teljesül, hogy x bal részfájának és jobb részfájának magassága legfeljebb 1-gyel különbözik. AVL fa megvalósításához minden csúcs tartalmaz egy extra $h[x]$ mezőt, amely az x csúcs magasságát tartalmazza. Mint minden bináris fa esetén $gyökér[T]$ a fa gyökércsúcsa.

a. Bizonyítsuk be, hogy minden n csúcsú AVL fa magassága $O(\lg n)$. (*Útmutatás.* Bizonyítsuk be, hogy ha az AVL fa magassága h, akkor legalább F_h csúcsot tartalmaz, ahol F_h a h-adik Fibonacci-szám.)

b. AVL fába beszúráskor először helyezzük el a keresőfában megfelelő helyre az új csúcsot. Ennek következtében a fa nem biztos, hogy magasság-kiegyensúlyozott marad. Pontosabban, előfordulhat, hogy egy csúcs bal és jobb részfája magasságának különbsége 2 lesz. Adjunk olyan EGYENSÚLYOZ(x) eljárást, amelynek x paramétere olyan csúcs, amelyre igaz, hogy a bal és jobb részfájának magassága legfeljebb 2-vel különbözik, azaz $|h[jobb[x]] - h[bal[x]]| \leq 2$, és úgy alakítja át az x-gyökerű részfát, hogy magasság-kiegyensúlyozott legyen. (*Útmutatás.* Végezzünk forgatást.)

c. A (b) részt felhasználva adjunk olyan AVL-BESZÚR(x, z) rekurzív eljárást, amelynek x paramétere egy AVL fa egy csúcsa, z az új beszúrandó csúcs (aminek a kulcs mezője már beállítottuk), és beszúrja a z csúcsot az x gyökerű fába, fenntartva az x gyökerű fa



13.9. ábra. Egy fapac példa. Minden x csúcs a $kulcs[x] : prioritás[x]$ párt tartalmazza. Például a gyökér a G kulcsot és a 4 prioritási értéket tartalmazza.

AVL tulajdonságát. Mint a 12.3. fejezetben a FÁBA-BESZÚR eljárásnál, most is feltételezzük, hogy $kulcs[z]$ már kapott értéket, továbbá $bal[z] = \text{NIL}$, $jobb[z] = \text{NIL}$ és $h[z] = 0$. Tehát a z csúcsnak a T AVL fába való beszúrását az AVL-BESZÚR($gyökér[T], z$) eljáráshívás végzi.

d. Mutassuk meg, hogy n csúcsú fára az AVL-BESZÚR eljárás futási ideje $O(\lg n)$, és $O(1)$ forgatást végez.

13-4. Fapacok

Ha n elemet szúrunk be egy bináris keresőfába, akkor a fa nagyon kiegyensúlyozatlan lehet, ami hosszú keresési időt eredményez. A véletlen építésű bináris keresőfa azonban, amint azt a 12.4. alfejezetben láttuk, közel kiegyensúlyozott lesz. Tehát az a stratégia, amely adott elemeket úgy szúr be bináris keresőfába, hogy először elkészíti az elemek véletlen permutációját, és azután ebben a sorrendben szúrja be az elemeket a fába, átlagosan kiegyensúlyozott fát eredményez.

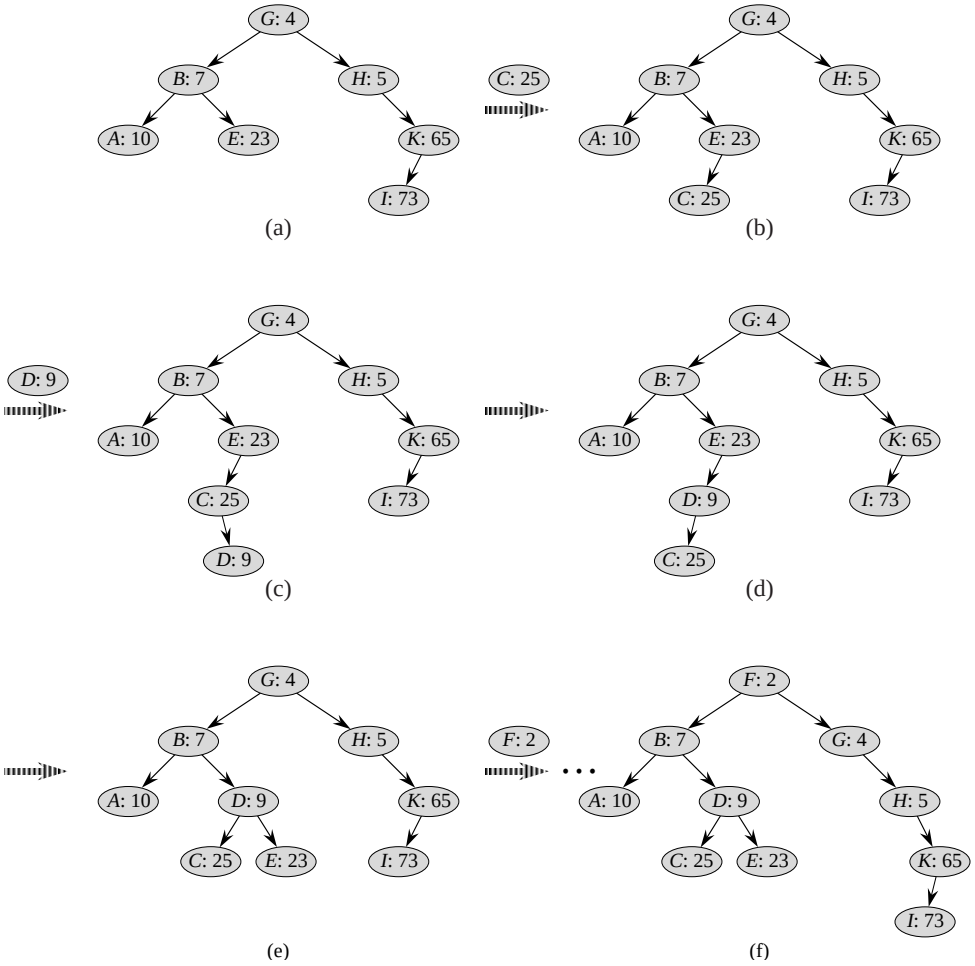
Mi a helyzet, ha nem adottak előre a beszúrandó elemek? Ha az elemeket egyesével kapjuk, akkor is tudunk belőlük építeni véletlen bináris keresőfát?

Olyan adatszerkezetet vizsgálunk, amely igenlő választ ad a kérdésre. A ***fapac*** olyan bináris keresőfa, amelyben a csúcsok elrendezése módosított. A 13.9. ábra egy példát mutat erre. Mint rendesen, minden x csúcs tartalmaz $kulcs[x]$ mezőt. Ezen felül, minden csúcshoz hozzárendelünk egy $prioritás[x]$ értéket, ami a csúcsoktól függetlenül választott véletlen szám. Feltételezzük, hogy mind a prioritási értékek, mind a kulcs értékek különbözőek. Fapacban a csúcsok olyan elrendezésűek, hogy a kulcsokra nézve teljesül a bináris keresőfa tulajdonság, a prioritási értékekre pedig a minimumos kupac tulajdonság teljesül.

- Ha v bal gyereke u-nak, akkor $kulcs[v] < kulcs[u]$.
- Ha v jobb gyereke u-nak, akkor $kulcs[v] > kulcs[u]$.
- Ha v gyereke u-nak, akkor $prioritás[v] > prioritás[u]$.

(A tulajdonságok kombinációja miatt használjuk a „fapac” elnevezést, mivel mind a bináris keresőfa, mind a kupac tulajdonság teljesül a fára.)

Fapac megértéséhez a következőképpen gondolkodjunk róla. Tegyük fel, hogy a megfelelő kulcsú $x_1, x_2, \ldots, x_n$ csúcsokat szúrjuk be a fapacba. Az eredményül kapott fapac az a fa, amelyet úgy kapnánk, hogy közönséges bináris keresőfába szúrnánk be a csúcsokat a (véletlenül választott) prioritási értékeik szerinti sorrendben, tehát ha $prioritás[x_i] < prioritás[x_j]$, akkor x_i-t előbb szúrjuk be, mint x_j-t.

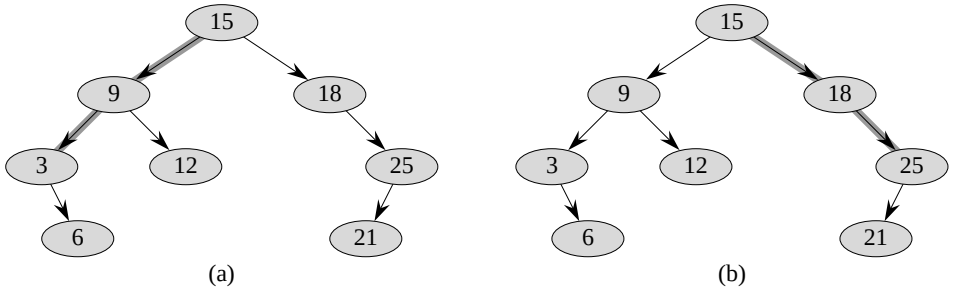


13.10. ábra. A FAPAC-BESZÚR művelet. **(a)** A beszúrás előtti eredeti fapac. **(b)** A *C* kulcsú, 25 prioritású csúcs beszúrása utáni fapac. **(c)-(d)** A *D* kulcsú, 9 prioritású csúcs beszúrása során keletkező közbülső állapotok. **(e)** A (c) és (d) részbeli beszúrások után keletkező fapac. **(f)** Az *F* kulcsú és 2 prioritású csúcs beszúrása után keletkező fapac.

a. Mutassuk meg, hogy csúcsok adott $x_1, x_2, \ldots, x_n$ halmazára egyetlen olyan fapac van, amely az adott csúcsokat tartalmazza. A csúcsokhoz tartozó kulcsok és prioritások mind különbözőek.

b. Mutassuk meg, hogy minden n csúcsú fapac magasságának várható értéke $\Theta(\lg n)$, és így a keresés futási ideje $\Theta(\lg n)$.

Mutassuk meg, hogyan lehet egy új csúcsot beszúrni egy meglévő fapacba. Először az új csúcshoz hozzárendelünk egy véletlen prioritást. Ezután meghívjuk a FAPAC-BESZÚR algoritmust, melynek működését a 13.10. ábra mutatja.



13.11. ábra. Bináris keresőfa gerincei. A bal oldali gerinc az **(a)** részen a satírozott út, a jobb oldali gerinc a **(b)** részen a satírozott út.

c. Magyarázzuk el a FAPAC-BESZÚR működését. Először szöveges leírását adjuk az algoritmusnak, majd írjuk meg a pszeudokódját. (*Útmutatás.* Először végezzünk közönséges bináris keresőfa beszúrást, majd forgatásokkal állítsuk helyre a minimumos kupac tulajdonságot.)

d. Mutassuk meg, hogy FAPAC-BESZÚR eljárás futási idejének várható értéke $\Theta(\lg n)$.

FAPAC-BESZÚR egy keresést és forgatások sorozatát hajtja végre. Ámbár e két művelet futási idejének várható értéke azonos, a gyakorlatban különböző költséget jelent. A keresés csak kiolvas információt a fa csúcsaiból anélkül, hogy módosítaná. Ellentétben a forgatással, amely szülő és gyerek mutató értékeket módosít a fapacban. A legtöbb számítógépen az olvasás művelet sokkal gyorsabb, mint az írás. Ezért azt szeretnénk, hogy a FAPAC-BESZÚR eljárás kevés forgatást végezzen. Meg fogjuk mutatni, hogy a forgatások számának várható értéke konstans.

Ennek érdekében be kell vezetnünk néhány meghatározást, amit a 13.11. ábra szemléltet. Egy T bináris keresőfa ***bal oldali gerince*** az az út, amely a gyökértől a legkisebb kulcsú csúcsig vezet. Más szóval, a bal oldali gerinc a gyökértől induló út, amely csak balra vezető éleket tartalmaz. Hasonlóan, T ***jobb oldali gerince*** az az út, amely a gyökértől indul, és csak jobbra vezető éleket tartalmaz. Gerinc hosszán az úton található csúcsok számát értjük.

e. Tekintsünk egy T fapacot közvetlenül az után, hogy egy x csúcsot beszúrtunk a FAPAC-BESZÚR eljárással. Legyen C az x-gyökerű részfa bal oldali gerincének, D pedig a jobb oldali gerincének a hossza. Bizonyítsuk be, hogy x beszúrása utáni forgatások száma $C + D$.

Ki fogjuk számítani C és D várható értékét. Az általánosság megszorítása nélkül feltehetjük, hogy a kulcsok értékei az $1, 2, \ldots, n$ számok, mivel csak egymással való összehasonlítást végzünk rajtuk.

Az $x \neq y$ csúcsokra legyen $k = kulcs[x]$ és $i = kulcs[y]$. Indikátor valószínűségi változókat definiálunk.

$$X_{i,k} = \mathrm{I}\{y \text{ rajta van } x \text{ bal részfájának a jobb oldali gerincén } (T\text{-ben})\}$$

f. Mutassuk meg, hogy $X_{i,k} = 1$ akkor és csak akkor teljesül, ha $prioritás[y] > prioritás[x]$, $kulcs[y] < kulcs[x]$, és minden z-re, ha $kulcs[y] < kulcs[z] < kulcs[x]$, akkor $prioritás[y] < prioritás[z]$.

g. Mutassuk meg, hogy

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)} . \tag{13.1}$$

h. Mutassuk meg, hogy

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k} . \tag{13.2}$$

i. Hasonló módon mutassuk meg, hogy

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1} .$$

j. Végső következtetésként mutassuk meg, hogy a beszúráskor végrehajtandó forgatások számának várható értéke kisebb, mint 2.

Megjegyzések a fejezethez

Keresőfák kiegyensúlyozásának ötlete Adelszon-Velszkijtől és Landistól [2] származik, akik 1962-ben megalkották az AVL-fa fogalmát, amely a 13.3 feladatban leírt kiegyensúlyozott keresőfa. Keresőfák egy másik osztálya a „2-3 fák”, amelyet J. E. Hopcroft vezetett be 1970-ben (nem publikált). Itt az egyensúly a fa fokának korlátozásával tartható fenn. A 2-3 fák általánosítása a Bayer és McCreight [32] által bevezetett B-fák, amelyet a 18. fejezetben tárgyalunk.

A piros-fekete fák fogalma Bayertől származik [31], aki a „szimmetrikus bináris B-fák” elnevezést használta. Guibas és Sedgewick [135] vizsgálta alaposan az ilyen fák tulajdonságait, és tőlük származik a piros-fekete színezési elnevezés. A piros-fekete fáknak egy egyszerűen kódolható változatát adta Anderson [15]. Weiss [311] AA-fának hívja ezt a változatot. Az AA-fa olyan piros-fekete fa, amelyben a bal gyerek sohasem lehet piros.

A fapac fogalmát Seidel és Aragon [271] javasolta. Ezt használja alapértelmezésben szótárak megvalósítására a LEDA rendszer, amely adatszerkezetek és algoritmusok megvalósításainak kiváló gyűjteménye.

Sok változata van a kiegyensúlyozott keresőfáknak, többek között a súly-korlátos fák [230], a k-szomszédos fák [213], a bűnbak fák [108]. A sokféle kiegyensúlyozott keresőfa közül talán a legérdekesebb a Sleator és Tarjan [281] által megalkotott „splay” fa, amely „önszervező” bináris keresőfa. (A splay fák kiváló leírása megtalálható Tarjan [292] munkájában.) Splay fában nem kell egyensúlyi érteket, mint például a színt, tárolni. Ehelyett valahányszor el kell érni a fa egy csúcsát, mindannyiszor végrehajtjuk a „splay” műveletet a fán (ez is forgatásokon alapszik). Minden művelet amortizált költsége (lásd a 17. fejezetet) $O(\lg n)$ minden n csúcsú fára.

Az ugrólista (Skiplist) [251] a kiegyensúlyozott bináris keresőfa egy alternatívája. Az ugrólista olyan rendezett lánc, amelyben több mutató is lehet minden cellában. Minden szótár művelet futási idejének várható értéke $O(\lg n)$ minden n elemet tartalmazó ugrólistára.

14. Adatszerkezetek kibővítése

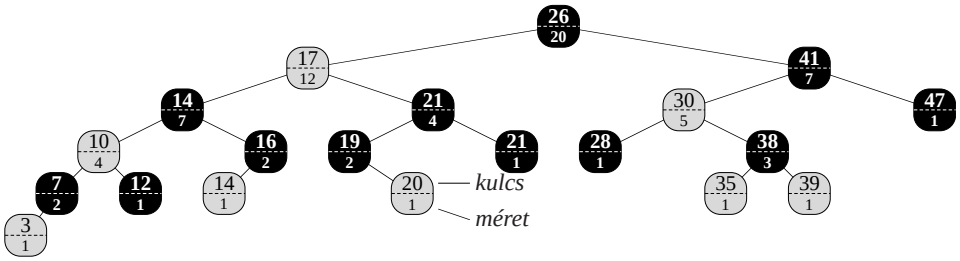
Vannak olyan helyzetek, amikor a feladat megoldása csak olyan adatszerkezetet kíván, amely bármely tankönyvben megtalálható – mint például láncolt lista, hasítótábla, vagy bináris keresőfa. Más esetekben a szükséges adattípusok kifejlesztése némi kreativitást kíván meg. Csak ritkán fordul elő, hogy teljesen új típusú adatszerkezet megalkotása váljék szükségessé. Igen gyakran elegendő csupán a már ismert adatszerkezetet kibővíteni további információkkal. Ezután olyan új műveleteket programozhatunk, amelyek támogatják a probléma megoldását. Az adatszerkezet kibővítése azonban nem mindig nyilvánvaló, mert az eredeti műveleteknek aktualizálni kell, és fenn kell tartani a pótlólagos információkat is.

Ebben a fejezetben két olyan adatszerkezettel foglalkozunk, amelyek a piros-fekete fák kibővítésével kaphatók. A 14.1. alfejezetben olyan adatszerkezetet vizsgálunk, amely támogatja dinamikus-halmazok rendezett mintára vonatkozó műveleteit. Gyorsan meg tudjuk találni adott halmaz i-edik legkisebb elemét és adott elem rendezés szerinti rangját. A 14.2. alfejezet általánosítja az adatszerkezetek kibővítését, és egy olyan tételt ad, amely egyszerűsíti a piros-fekete fák kibővítését. A 14.3. alfejezetben felhasználjuk ezt az eredményt olyan adatszerkezet kifejlesztésére, amely támogatja intervallumokat tartalmazó dinamikus-halmazok kezelését. Ilyenek például az idő-intervallumok halmazai. Adott intervallumhoz gyorsan tudunk találni olyan intervallumot a halmazban, amelynek van közös része az adott intervallummal.

14.1. Dinamikus rendezett minta

A rendezett minta fogalmát a 9. fejezetben vezettük be. Egy n elemű halmaz i-edik ($i \in \{1, \ldots, n\}$) legkisebb elemén a halmaz i-edik legkisebb kulcsú elemét értjük. Láttuk, hogy nem rendezett halmaz i-edik legkisebb eleme megkereshető $O(n)$ idejű algoritmussal. Ebben az alfejezetben megmutatjuk, hogyan oldható meg ez a feladat $O(\lg n)$ időben piros-fekete fák módosításával. Azt is látni fogjuk, hogy adott elem rangja – a rendezett halmazbeli pozíciója – hogyan határozható meg $O(\lg n)$ idejű algoritmussal.

A 14.1. ábra olyan adatszerkezetet mutat, amely támogatja gyors rendezettminta-műveletek megvalósítását. A ***rendezettminta-fa*** olyan T piros-fekete fa, amelynek minden csúcsában kiegészítő információt is tárolunk. A piros-fekete fák szokásos mezőin, azaz a *kulcs*[x], *szülő*[x], *bal*[x], *jobb*[x] mezőkön kívül van egy *méret*[x] mező is. Ez a mező az



14.1. ábra. Egy rendezettminta-fa, amely kibővített piros-fekete fa. A világos csúcsok pirosak, a sötét csúcsok pedig fekete csúcsok. A szokásos mezőkön kívül minden x csúcs tartalmaz egy *méret*[x] mezőt, amely az x gyökerű részfa csúcsainak számát tartalmazza.

x gyökerű részfa (belső) csúcsainak számát tartalmazza (beleértve saját magát is), tehát a részfa méretét. Ha *méret*[NIL] $= 0$ definíciót használjuk, akkor a következő azonossághoz jutunk

$$méret[x] = méret[bal[x]] + méret[jobb[x]] + 1.$$

Nem kell kikötni, hogy a kulcsok különbözőek legyenek a rendezett minta-fában. (Például a 14.1. ábrán látható fában a 14 és a 21 kulcs kétszer szerepel.) Ha egy kulcs többször is előfordul, akkor a rang nem egyértelműen definiált. A többértelműséget úgy szüntetjük meg rendezettminta-fa esetére, hogy az elem rangja legyen az inorder bejárás szerinti sorszáma. Például a 14.1 ábrán a fekete csúcsban lévő 14 kulcs rangja 5, míg a piros csúcsbelié 6.

Adott rangú elem keresése

Mielőtt megmutatnánk, hogy a *méret* információ hogyan tartható fenn beszúrás és törlés során, lássunk két olyan rendezett-minta műveletet, amelyek ezt a kiegészítő információt használják. Először azt a műveletet tekintsük, amely megadja a halmaz adott rangú elemét. Ez az RM-KIVÁLASZT(x, i) eljárás.

RM-KIVÁLASZT(x, i)

```
1  r ← méret[bal[x]] + 1
2  if i = r
3     then return x
4  elseif i < r
5     then return RM-KIVÁLASZT(bal[x], i)
6  else return RM-KIVÁLASZT(jobb[x], i − r)
```

A művelet eredménye arra a csúcsra mutató érték, amely az x gyökerű fában az i-edik legkisebb kulcsú elemet tartalmazza. Egy T rendezettminta-fa i-edik legkisebb elemét az RM-KIVÁLASZT(*gyökér*[T], i) eljáráshívás adja. Az RM-KIVÁLASZT algoritmus ötlete nagyon hasonló ahhoz, mint amit a 9. fejezetben megismertünk. A *méret*[*bal*[x]] érték a rendezésben x-et megelőző elemek száma az x gyökerű részfában. Tehát *méret*[*bal*[x]] + 1 nem más, mint x rangja az x gyökerű részfában.

Az RM-KIVÁLASZT eljárás 1. sorában az r változó felveszi x rangját az x gyökerű részfában. Ha $i = r$ akkor x az i-edik legkisebb elem, tehát az eljárás terminál a 3. sorban. Ha

$i < r$, akkor az i-edik legkisebb elem x bal részfájában van, tehát rekurzív hívás következik az 5. sorban a $bal[x]$ részfára. Ha $i > r$, akkor az i-edik legkisebb elem x jobb részfájában van. Mivel r olyan elem van az x gyökerű részfában, amelyek megelőzik az inorder bejárás szerint x jobb részfájában lévő elemeket, így az i-edik legkisebb elem az x gyökerű részfában nem más, mint az $(i-r)$-edik legkisebb elem a $jobb[x]$ gyökerű részfában. Ezt az elemet a 6. sorban található rekurzív hívással határozzuk meg.

Az RM-KIVÁLASZT algoritmus működésének szemléltetésére tekintsük a 14.1. ábrán látható rendezettminta-fa 17-edik legkisebb elemének kiválasztását. Kezdetben x a fa gyökere, amelynek kulcsa 26, és $i = 17$. Mivel a 26 kulcsot tartalmazó csúcs bal részfájának mérete 12, így rangja 13. Tehát tudjuk, hogy az a csúcs, amelynek rangja 17, a $17 - 13 = 4$-edik legkisebb elem a 26-ot tartalmazó csúcs jobb részfájában. A rekurzív hívás után x a 41 kulcsú elem lesz, és $i = 4$. Mivel a 41 kulcsú csúcs bal részfájának mérete 5, így rangja 6. Tehát tudjuk, hogy a 4 rangú elem a 4-edik legkisebb elem x bal részfájában. A rekurzív hívás után x a 30 kulcsú elemre mutat, amelynek rangja az x gyökerű részfában 2. Ismételt rekurzióval keressük azt az elemet, amelynek rangja $4 - 2 = 2$ abban a részfában, amelynek gyökere a 38 kulcsot tartalmazza. Azt találjuk, hogy ennek a csúcsnak a rangja 2, tehát az eljárás a 38 kulcsot tartalmazó csúcsra mutató mutatót ad eredményül.

Mivel minden rekurzív hívás egy csúccsal lefelé halad a fában, az RM-KIVÁLASZT algoritmus teljes futási ideje a legrosszabb esetben arányos a fa magasságával. A rendezettminta-fa piros-fekete fa, így magassága $O(\lg n)$, ha a fa n csúcsot tartalmaz. Tehát az RM-KIVÁLASZT algoritmus futási ideje $O(\lg n)$ minden n elemű dinamikus halmazra.

Elem rangjának meghatározása

Az RM-RANG algoritmus adott T rendezettminta-fa és annak egy x csúcsára meghatározza az x csúcs pozícióját a T fa inorder bejárásával kapott sorozatban.

RM-RANG(T, x)

```
1  r ← méret[bal[x]] + 1
2  y ← x
3  while y ≠ gyökér[T]
4        do if y = jobb[szülő[y]]
5              then r ← r + méret[bal[szülő[y]]] + 1
6           y ← szülő[y]
7  return r
```

Az eljárás a következőképpen működik. Az x csúcs rangja úgy tekinthető, mint az inorder bejárásban x-et megelőző elemek száma plusz 1. Teljesül a következő invariáns tulajdonság:

> A 3–6. sorokban megfogalmazott **while** ciklus minden végrehajtása előtt a $kulcs[x]$ elem rangja r az y gyökerű részfában.

Ezt az invariánst használjuk annak bizonyítására, hogy RM-RANG helyesen működik.

Teljesül: A ciklusmag első végrehajtása előtt az 1. sorban r felveszi a $kulcs[x]$ elemnek a rangját az x gyökerű részfában. A 2. sorbeli $y \leftarrow x$ értékadás miatt az invariáns teljesül a 3. sorban végrehajtott ellenőrzésnél.

Megmarad: A **while** ciklus minden egyes végrehajtásának végén végrehajtjuk az $y \leftarrow$ *szülő*[y] értékadást. Tehát azt kell megmutatnunk, hogy a ciklusmag végrehajtása el őtt r a *kulcs*[x] elem rangja az y gyökerű részfában, és a ciklusmag végén r a *kulcs*[x] elem rangja a *szülő*[y] gyökerű részfában. A **while** ciklus minden egyes végrehajtásában a *szülő*[y] gyökerű részfát vizsgáljuk. Ekkor már összeszámláltuk azokat a csúcsokat, amelyek az y gyökerű részfában vannak, és megel őzik az inorder bejárás szerint az x-et, tehát növelni kell az r értékét azon csúcsok számával, amelyek abban a részfában vannak, amelynek gyökere y testvére, és még eggyel, ha az *szülő*[y] is megel őzi az x-et. Ha az y csúcs bal gyereke szülőjének, akkor sem *szülő*[y], sem az y jobb részfájában lévő elemek nem előzik meg x-et, tehát r értéke változatlan marad. Ellenkez ő esetben y szülőjének jobb gyereke, tehát *szülő*[y] bal részfájában lév ő elemek, és *szülő*[y] maga is megelőzi x-et az inorder bejárás szerint. Ezért az 5. sorban r aktuális értékéhez hozzáadjuk a *méret*[*bal*[*szülő*[y]]] + 1 értéket.

Befejeződik: Ha y = *gyökér*[T], akkor az ismétlés befejez ődik, és ekkor az y gyökerű részfa a teljes T fa. Tehát r a *kulcs*[x] elem rangja a teljes fában.

Például ha az RM-RANG algoritmust alkalmazzuk arra a rendezett-minta-fára, amelyet a 14.1. ábra mutat, keresve a 38 kulcsú elem rangját, akkor a következ ő *kulcs*[y] és r értékek sorozatát kapjuk a **while** ciklus ismétlése során.

ismétlés	*kulcs*[y]	r
1	38	2
2	30	4
3	41	4
4	26	17

Eredményül 17-et kapunk.

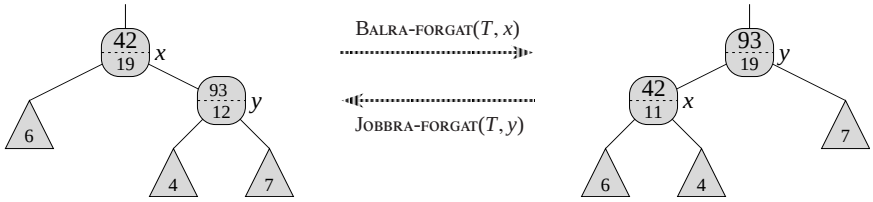
Mivel a **while** ciklusmag minden ismétlése $O(1)$ id őt igényel, és y egy szinttel feljebb kerül a fában minden ismétléssel, így az RM-RANG algoritmus futási ideje a legrosszabb esetben is arányos a fa magasságával, ami $O(\lg n)$ minden n csúcsú rendezett-minta-fára.

Részfák méretének fenntartása

A piros-fekete fák csúcsainak kib ővítése *méret* mezővel azt eredményezi, hogy az RM-KIVÁLASZT és az RM-RANG algoritmusok gyorsan képesek kiszámítani a kívánt eredményt. Azonban a *méret* kiegészítő információt a piros-fekete fák módosító műveleteinek hatékonyan fenn kell tartania, különben eddigi munkánk semmibe vész. Megmutatjuk, hogy a részfák mérete fenntartható mind a beszúrás, mind a törlés során úgy, hogy a műveletek aszimptotikus időigénye nem változik.

A 13.3. alfejezetben láttuk, hogy a piros-fekete fák esetén a beszúrás két menetben történik. Az első menetben a gyökértől haladunk lefelé egy olyan csúcsig, amelynek gyereke lesz az új csúcs. A második menetben felfelé haladunk a fában, és csúcsok színét változtatjuk, valamint forgatást végzünk, hogy fenntartsuk a piros-fekete tulajdonság teljesülését.

Az első menetben az aktualizálás csak annyit jelent, hogy azon x csúcsok *méret*[x] mezőit kell növelni eggyel, amelyeken áthaladunk a keresés során, az új csúcs mérete pedig 1 lesz. Mivel a keresőúton $O(\lg n)$ csúcs van, így a *méret* információ fenntartásának pótlólagos költsége $O(\lg n)$.



14.2. ábra. A *méret* információ aktualizálása forgatás során. Azon két csúcs *méret* mezőit kell aktualizálni, amely csúcsokat összekötő kapcsolat körül a forgatást végezzük. Az aktualizálás lokális, csak az *x*, *y* csúcsok, továbbá a háromszöggel jelölt részfák gyökerének *méret* mezőire van szükség.

A második menetben egyedül forgatás okozhat a piros-fekete fa szerkezetén változást, és legfeljebb két forgatás hajtunk végre. Továbbá, minden forgatás lokális művelet, csak annak a két csúcsnak a mérete változik, amely kapcsolat körül a forgatás történik. Tehát a 13.2. alfejezetben tárgyalt BALRA-FORGAT eljárást ki kell kiegészíteni az alábbi két sorral:

12 *méret*[*y*] ← *méret*[*x*]
13 *méret*[*x*] ← *méret*[*bal*[*x*]] + *méret*[*jobb*[*x*]] + 1

A 14.2. ábra szemlélteti a forgatás után végrehajtandó aktualizálást. A JOBBRA-FORGAT eljárás módosítása a szimmetriának megfelelően történik.

Mivel legfeljebb két forgatást kell végezni piros-fekete fába történő beszúrás során, így $O(1)$ a pótlólagos időigénye a *méret* információ fenntartásának a második menetben. Tehát n csúcsú rendezett-minta-fába való beszúrás teljes időigénye $O(\lg n)$ – ami aszimptotikusan megegyezik a közönséges piros-fekete fák esetével.

Piros-fekete fa törlés algoritmusa szintén két menetből áll: az első menet törlést végez a bináris keresőfából, a második menet legfeljebb három forgatást végez, egyébként nem eredményez szerkezeti változást (lásd a 13.4. alfejezetet). Az első menet eltávolít a fából egy *y* csúcsot. A *méret* információ aktualizálása végett egyszerűen az *y* csúcsból felfelé a gyökérig haladva csökkentjük eggyel minden, az úton található csúcs méretet. Mivel ezen út hossza $O(\lg n)$ minden n csúcsú piros-fekete fában, így az első menet pótlólagos időigénye $O(\lg n)$. A második menetben végzett $O(1)$ forgatás kezelése hasonló a beszúrás esetéhez. Tehát mind a beszúrás, mind a törlés, a *méret* információ fenntartásával együtt $O(\lg n)$ időt igényel n csúcsú rendezettminta-fa esetén.

Gyakorlatok

14.1-1. Mutassuk be, hogyan működik az RM-KIVÁLASZT(*gyökér*[*T*], 10) művelet a 14.1. ábrán látható *T* piros-fekete fára.

14.1-2. Mutassuk be, hogyan működik az RM-RANG(*T*, *x*) művelet a 14.1. ábrán látható *T* piros-fekete fára, ha *x* a 35 kulcsú csúcs.

14.1-3. Írjuk meg az RM-KIVÁLASZT algoritmus nemrekurzív változatát.

14.1-4. Írjunk olyan rekurzív RM-KULCS-RANG(*T*, *k*) eljárást, amely a *T* rendezettminta-fa és a *k* kulcs bemenetre kiszámítja *k* rangját a *T* fa által ábrázolt dinamikus-halmazban. Feltehető, hogy *T*-ben minden kulcs különböző.

14.1-5. Adott egy n csúcsú rendezettminta-fa *x* csúcsa és egy *i* természetes szám. Hogyan számítható ki *x*-nek a lineáris rendezés szerinti *i*-edik követője $O(\lg n)$ időben?

14.1-6. Vegyük észre, hogy az RM-KIVÁLASZT és az RM-RANG algoritmusban valahányszor hivatkozunk egy x csúcs *méret* mezőjére, csak arra használjuk, hogy megkapjuk x rangját az x gyökerű részfában. Ennek megfelelően, tegyük fel, hogy minden csúcsban tároljuk ezt az információt. Mutassuk meg, hogyan tartható fenn ez az információ beszúrás és törlés során. (Emlékezzünk arra, hogy ez a két művelet forgatást igényelhet.)

14.1-7. Mutassuk meg, hogy rendezettminta-fa használatával hogyan számítható ki egy n elemű sorozat inverzióinak száma $O(n \lg n)$ időben. (Lásd az 2-4. feladatot.)

14.1-8.⋆ Tekintsük egy kör n darab húrját, amelyek a végpontjaikkal adottak. Adjunk olyan $O(n \lg n)$ idejű algoritmust, amely kiszámítja, hogy hány húrpár metszi egymást a körön belül. (Például ha az n húr mindegyike átmérő, amely keresztül megy a középponton, akkor a válasz $\binom{n}{2}$.) Tegyük fel, hogy bármely két húrnak nincs közös végpontja.

14.2. Hogyan bővítsünk adatszerkezetet

Algoritmusok tervezése során gyakran előfordul, hogy a funkcionalitás bővítése érdekében egy adott adatszerkezetet bővíteni kell. Nevezzük a kiindulási adatszerkezetet alap-adatszerkezetnek. A bővítést használjuk a következőkben is olyan adatszerkezet tervezésére, amely támogatja intervallumokon végzendő műveleteket. Ebben az alfejezetben a bővítés során végzendő lépéseket vizsgáljuk. Bebizonyítunk egy olyan tételt, amely sok esetben egyszerűen használható piros-fekete fák bővítésénél.

Adatszerkezet bővítése négy lépésre bontható:

1. az alap-adatszerkezet megválasztása,
2. az alap-adatszerkezetben fenntartandó kiegészítő információk meghatározása,
3. annak igazolása, hogy a kiegészítő információk fenntarthatók a lap-adatszerkezet módosító műveletei során, és
4. új műveletek kifejlesztése.

Mint minden leíró tervezési módszer esetén, nem kell mindig vakon követni a lépéseket a megadott sorrendben. A legtöbb tervezés próbálgatást is magában foglal, és az egyes lépések megvalósítása párhuzamosan halad. Nincs értelme például a kiegészítő információkat megadni, és új műveleteket tervezni (2. és 4. lépés), ha nem tudjuk hatékonyan fenntartani a kiegészítő információkat. Azonban ez a négylépéses módszer jól irányítja figyelmünket azon erőfeszítésre, amely során adatszerkezetet bővítünk. Továbbá, ez egy jó módszert kínál bővített adatszerkezetek dokumentálására.

Ezeket a lépéseket követtük a 14.1. alfejezetben, amikor a rendezett-minta-fát terveztük. Az első lépésben piros-fekete fát választottunk alap-adatszerkezetként. Az vezetett a piros-fekete fák alkalmasságának felismeréséhez, hogy tudtuk, ez az adatszerkezet támogatja olyan műveletek hatékony megvalósítását, mint a MINIMUM, MAXIMUM, KÖVETKEZŐ és ELŐZŐ.

Második lépésként bevezettük a *méret* kiegészítő információt, amely minden x csúcsra az x gyökerű részfa csúcsainak száma. Általában, a kiegészítő információ hatékonyabbá teszi az alkalmazott műveleteket. Például az RM-KIVÁLASZT és az RM-RANG műveleteket meg tudnánk valósítani pusztán a csúcsban lévő kulcsot használva, de ekkor nem kapnánk $O(\lg n)$ idejű algoritmust. Néha a kiegészítő információ mutató és nem adat, mint a 14.2-1. gyakorlatban.

Harmadik lépésként megmutattuk, hogy a beszúrás és a törlés során fenntartható a *méret* kiegészítő információ úgy, hogy a futási idő $O(\lg n)$ marad. Kedvező esetben csak kisebb változtatás szükséges a kiegészítő információk fenntartásához. Például ha a fa minden csúcsában tárolnánk a csúcs rangját, akkor az RM-KIVÁLASZT és az RM-RANG eljárás gyorsan futna, de egy új legkisebb elem beszúrása után minden csúcsban módosítani kellene ezt az adatot. Ehelyett a méretet tárolva, minden új elem beszúrása után csak $O(\lg n)$ csúcsot kell módosítani.

A negyedik lépésben fejlesztettük ki az RM-KIVÁLASZT és az RM-RANG műveleteket. Ezen műveletek szükségessége miatt foglalkoztunk mindenekelőtt adatszerkezet bővítésével. Alkalmanként nem új műveleteket tervezünk, hanem a meglévőket javítjuk, mint a 14.2-1. gyakorlatban.

Piros-fekete fák bővítése

Ha a bővített adatszerkezet alap-adatszerkezete piros-fekete fa, akkor bizonyítható, hogy bizonyos típusú kiegészítő információk mindig fenntarthatók beszúrás és törlés során, ezzel nagyon egyszerűvé téve a harmadik lépést. A következő tétel bizonyítása hasonló, mint a 14.1. alfejezetben annak igazolása volt, hogy a rendezettminta-fa esetén a *méret* információ fenntartható.

14.1. tétel (piros-fekete fák bővítése). *Legyen f piros-fekete fák olyan mezője, amely bővítése az adatszerkezetnek, és teljesül, hogy $f[x]$ értéke kiszámítható az x, $bal[x]$, és $jobb[x]$ csúcsokban lévő információkból, beleértve az $f[bal[x]]$, és $f[jobb[x]]$ értékeket. Ekkor f értéke fenntartható a beszúrás és a törlés során úgy, hogy az algoritmusok aszimptotikus hatékonysága nem változik, azaz $O(\lg n)$ minden n csúcsú fára.*

Bizonyítás. A bizonyítás alapötlete az, hogy egy x csúcs f mezőjének megváltozása csak x őseire van kihatással. Azaz, ha $f[x]$ értékét változtatjuk, akkor csak $f[szülő[x]]$ értékét kell aktualizálni, ezután csak $f[szülő[szülő[x]]]$ értékét kell aktualizálni, és így tovább, amíg a gyökér f értékét nem aktualizáltuk. Ha a gyökér f értékét aktualizáltuk, akkor már nincs olyan egyéb csúcs, amelynek f mezője az új értéktől függne, így az aktualizálás véget ér. Mivel minden n csúcsú piros-fekete fa magassága $O(\lg n)$, így egy csúcs f mezőjének megváltoztatása esetén $O(\lg n)$ időben helyreállítható minden olyan csúcs értéke, amely függ a megváltoztatottól.

Egy T fába egy x csúcs beszúrása két menetben történik (lásd 13.3.). Az első menetben az x csúcsot beillesztjük a már létező $szülő[x]$ gyerekeként. $f[x]$ értéke kiszámítható $O(1)$ időben, mert a feltételezés szerint ez csak az x csúcs egyéb mezőitől és x gyerekeitől függ, azonban x mindkét gyereke a $nil[T]$ őrszem. Miután $f[x]$ értékét kiszámítottuk, a változás hatása felfelé terjed a fában. Tehát az első menet teljes időigénye $O(\lg n)$. A második menetben csak a forgatások eredményeznek szerkezeti változást. Egy forgatásban két csúcs változik, így az f mező aktualizálásának teljes időköltsége forgatásonként $O(\lg n)$. Mivel a beszúrás során legfeljebb két forgatást kell végezni, így a beszúrás teljes időigénye $O(\lg n)$.

Mint a beszúrás, a törlés is két menetben történik. (Lásd a 13.4. alfejezetet.) Az első menetben a fa szerkezete akkor változik, amikor a törlendő csúcsot helyettesítjük a követőjével, és azután vagy a törlendő csúcsot, vagy a követőjét kikapcsoljuk a fából. Mivel a változtatás lokális, ezért az aktualizálás időköltsége legfeljebb $O(\lg n)$. A piros-fekete tu-

lajdonság helyreállítása a második menetben legfeljebb három forgatást igényel, és minden forgatás után $O(\lg n)$ idő szükséges az f értékek újraszámítására a fában felfelé haladva. Tehát a törlés teljes időigénye $O(\lg n)$, csakúgy, mint a beszúrásé. ■

Sok esetben, mint például a rendezett-minta-fák esetén a *méret* információ fenntartása, a forgatás utáni aktualizálás csak $O(1)$ időt igényel, ellentétben a 14.1. tételben bizonyított $O(\lg n)$ helyett. A 14.2-4. gyakorlat egy másik példát mutat erre.

Gyakorlatok

14.2-1. Mutassuk meg, hogy a MINIMUM, MAXIMUM, KÖVETKEZŐ és ELŐZŐ dinamikus-halmaz műveletek megvalósíthatók legrosszabb esetben is $O(1)$ idejű algoritmussal, ha a halmazt rendezett-minta-fával ábrázoljuk. A többi művelet aszimptotikus hatékonysága nem változhat. (*Útmutatás.* A csúcsokhoz adjunk hozzá mutatókat.)

14.2-2. Tegyük fel, hogy piros-fekete fa minden csúcsát kiegészítjük a csúcs feketemagasságával. Fenntartható-e ez az érték úgy, hogy a műveletek aszimptotikus hatékonysága ne változzon? Mutassuk meg, hogyan, vagy indokoljuk meg, miért nem.

14.2-3. Fenntartható-e hatékonyan piros-fekete fában minden csúcs magassága mint kiegészítő információ? Mutassuk meg, hogyan, vagy indokoljuk meg, miért nem.

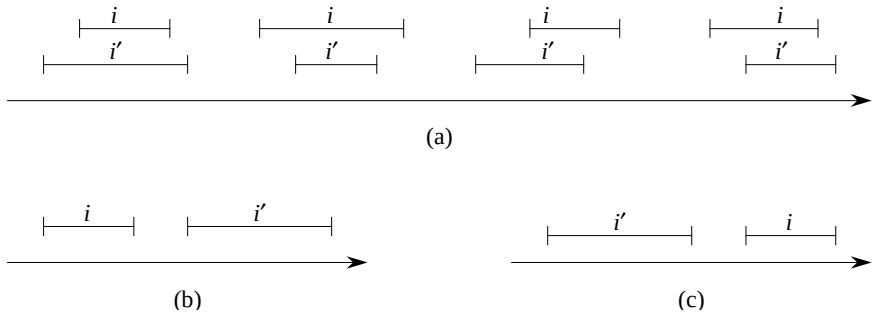
14.2-4.⋆ Legyen $\otimes$ egy asszociatív kétváltozós művelet, és legyen a a piros-fekete fa egy mezője. Tegyük fel, hogy a fa minden x csúcsát ki akarjuk egészíteni egy f mezővel, amelynek értékét az $f[x] = a[x_1] \otimes a[x_2] \otimes \cdots \otimes a[x_m]$ összefüggéssel számítjuk, ahol $x_1, x_2, \ldots, x_m$ az x gyökerű részfa csúcsának inorder felsorolása. Mutassuk meg, hogy az f mező aktualizálható forgatás után $O(1)$ időben. Ennek bizonyítását módosítva mutassuk meg, hogy rendezett-minta-fában a *méret* információ aktualizálható forgatás után $O(1)$ időben.

14.2-5.⋆ Piros-fekete fát szeretnénk úgy bővíteni, hogy támogassa azt az új RM-LISTÁZ(T, x, a, b) műveletet, amely kilistázza az x gyökerű részfa mindazon k kulcsát, amelyekre teljesül az $a \le k \le b$ egyenlőtlenség. Mutassuk meg, hogyan lehet az RM-LISTÁZ műveletet megvalósítani $\Theta(m + \lg n)$ időben, ahol m a listázandó kulcsok száma, n pedig a fa belső csúcsainak száma. (*Útmutatás.* Nem szükséges kiegészítő mezőt bevezetni a piros-fekete fában.)

14.3. Intervallum-fák

Ebben az alfejezetben piros-fekete fák olyan bővítésével foglalkozunk, amelyek támogatják intervallumokat tartalmazó dinamikus halmazok műveleteit. ***Zárt intervallumon*** egy $[t_1, t_2]$ rendezett valós számpárt értünk, ahol $t_1 \le t_2$. A $[t_1, t_2]$ intervallum a $\{t \in \mathbf{R} : t_1 \le t \le t_2\}$ halmazt ábrázolja. ***Nyitott,*** illetve ***félig nyitott*** intervallum esetén mindkét, illetve valamelyik végpont nem szerepel a halmazban. A továbbiakban feltesszük, hogy a szóban forgó intervallum mindig zárt. Nyitott, illetve félig nyitott intervallumok esete hasonlóan kezelhető.

Az intervallumokkal kényelmesen ábrázolhatók események, amelyek időben egy folytonos szakaszt képeznek. Például kérdést tehetünk fel, hogy intervallum adatbázisban mely események történtek egy adott intervallumon belül. A tárgyalandó adatszerkezet hatékony eszközt nyújt ilyen intervallum adatbázis kezelésére.



14.3. ábra. Intervallum trichotómia az i és i' zárt intervallumokra. **(a)** Ha i és i' átfedi egymást, akkor négy eset lehetséges; mindegyikben teljesülnek az $alsó[i] \leq felső[i]$ és $alsó[i'] \leq felső[i]$ egyenlőtlenségek. **(b)** Az intervallumok nem átfedők és $felső[i] < alsó[i']$. **(c)** Az intervallumok nem átfedők és $felső[i'] < alsó[i]$.

Egy $[t_1, t_2]$ intervallumot ábrázolhatunk egy olyan i objektummal, amelynek két mezője van, az intervallum két végpontja: (***alsó végpont***) $alsó[i] = t_1$ és (***felső végpont***) $felső[i] = t_2$. Azt mondjuk, hogy az i és i' intervallumok átfedik egymást, ha az $i \cap i' \neq \emptyset$, vagyis $alsó[i] \leq felső[i']$ és $alsó[i'] \leq felső[i]$. Bármely két, i és i' intervallum teljesíti az ***intervallum trichotómia*** tulajdonságot, azaz a következő három állítás közül pontosan egyik teljesül rájuk:

a. az i és i' intervallumok átfedik egymást,

b. $felső[i] < alsó[i']$,

c. $felső[i'] < alsó[i]$.

A 14.3. ábra mutatja a három lehetséges esetet.

Az ***intervallum-fa*** olyan piros-fekete fa, amely lehetővé teszi olyan dinamikus halmaz kezelését, amelynek minden x eleme tartalmaz egy $int[x]$ intervallumot. Intervallum-fák a következő műveleteket támogatják.

INTERVALLUMOT-BESZÚR(T, x): bővíti a T intervallum-fát az x csúccsal, amelynek *int* mezője egy intervallumot tartalmaz.

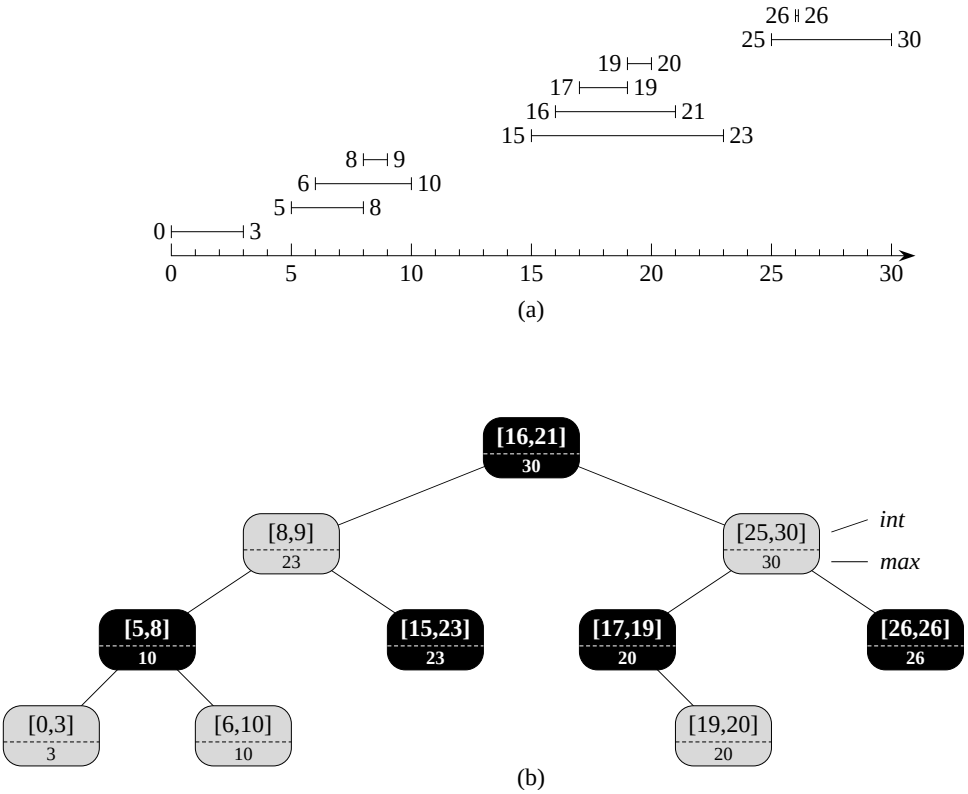
INTERVALLUMOT-TÖRÖL(T, x): törli a T intervallum-fából az x csúcsot.

INTERVALLUMOT-KERES(T, i): eredményül a T fa egy olyan x csúcsára mutató mutatót ad, amelyre teljesül, hogy $int[x]$ átfedi az i intervallumot, illetve $nil[T]$ az eredmény, ha nincs a fában ilyen csúcs.

A 14.4. ábra azt mutatja, hogyan lehet intervallumok egy halmazát ábrázolni intervallum-fával. A továbbiakban a 14.2. alfejezetben tárgyalt négylépéses tervezési módszert fogjuk követni az intervallum-fa tervezése és a műveletek megvalósítása során.

1. lépés: az alap-adatszerkezet megválasztása

Olyan piros-fekete fát használunk alap-adatszerkezetként, amelynek minden x csúcsa tartalmaz egy $int[x]$ intervallumot, és az x csúcs kulcsa az $alsó[int[x]]$ érték, azaz az intervallum alsó végpontja.



14.4. ábra. Egy intervallum-fa. **(a)** 10 intervallum, amelyek alulról felfelé haladva az alsó végpontjuk szerint rendezettek. **(b)** A 10 intervallumot ábrázoló intervallum-fa. A fa inorder bejárása az intervallumok alsó végpontja szerint rendezett sorozatot ad.

2. lépés: a kiegészítő információk meghatározása

Az intervallum mellett minden x csúcs tartalmazza a $max[x]$ kiegészítő információt, amely az x gyökerű részfában lévő intervallumok végpontjainak a maximuma. Mivel minden intervallum felső végpontja nagyobb vagy egyenlő, mint az alsó végpontja, így $max[x]$ az x gyökerű részfában lévő intervallumok felső végpontjainak a maximuma.

3. lépés: a kiegészítő információ fenntartása

Meg kell mutatnunk, hogy a beszúrás és a törlés elvégezhető $O(\lg n)$ időben minden n csúcsú intervallum-fán. Vegyük észre, hogy $max[x]$ értékét ki tudjuk számítani, ha adott $int[x]$ és az x csúcs gyerekeiben lévő max értékek:

$$max[x] = max(felső[int[x]], max[bal[x]], max[jobb[x]]).$$

Tehát a 14.1. tétel szerint a beszúrás és a törlés futási ideje $O(\lg n)$. Valójában a max érték $O(1)$ időben aktualizálható minden forgatás után, amit a 14.2-4. és 14.3-1. gyakorlatok is mutatnak.

4. lépés: az új műveletek kifejlesztése

Ténylegesen csak az INTERVALLUMOT-KERES(T, i) az új művelet, amely megkeres a T fában egy olyan intervallumot, amely átfedi az adott i intervallumot. Ha nincs ilyen intervallum a fában, akkor az eredmény a $nil[T]$ őrszem.

INTERVALLUMOT-KERES(T, i)

```
1  x ← gyökér[T]
2  while x ≠ nil[T] and i nem fedi át int[x]-et
3        do if bal[x] ≠ NIL and max[bal[x]] ≥ alsó[i]
4              then x ← bal[x]
5              else x ← jobb[x]
6  return x
```

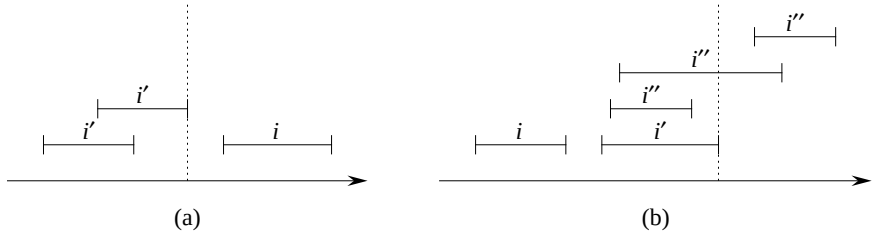
Az i-t átfedő intervallum keresése úgy indul, hogy x a fa gyökere, és a keresés a fában lefelé halad. A keresés akkor fejeződik be, ha találtunk egy csúcsot, amelyben lévő intervallum átfedi i-t, vagy x a $nil[T]$ őrszemre mutat. Mivel a ciklusmag végrehajtásának ideje $O(1)$, és minden n csúcsú piros-fekete fa magassága $O(\lg n)$, így az INTERVALLUMOT-KERES algoritmus futási ideje $O(\lg n)$.

Mielőtt bebizonyítanánk az algoritmus helyességét, nézzük meg, hogyan működik az algoritmus a 14.4. ábrán látható intervallum-fára. Tegyük fel, hogy olyan intervallumot keresünk, amely átfedi az $i = [22, 25]$ intervallumot. Kezdetben x a gyökér, amely a $[16, 21]$ intervallumot tartalmazza, tehát nem fedi i-t. Mivel $max[bal[x]] = 23$ nagyobb, mint $alsó[i] = 22$, a ciklus úgy folytatódik, hogy x a bal gyerek lesz, amelyben a [8,9] intervallum van, tehát ez sem fedi át i-t. Ekkor $max[bal[x]] = 10$, ami kisebb, mint $alsó[i] = 22$, tehát az ismétlés úgy folytatódik, hogy x jobb gyereke lesz az új x csúcs. Az ebben lévő $[15, 23]$ intervallum átfedi i-t, tehát az eljárás véget ér, és ezt az intervallumot eredményezi.

Sikertelen keresésre példaként vegyük azt az esetet, amikor a 14.4. ábrán látható intervallum-fában keresünk olyan intervallumot, amely átfedi az $i = [11, 14]$ intervallumot. Ismét úgy kezdünk, hogy x a fa gyökere. Az itt lévő intervallum $[16, 21]$ nem fedi át i-t, továbbá $max[bal[x]] = 23$ nagyobb, mint $alsó[i]$, így balra megyünk, ahol a $[8, 9]$ intervallum van. (Jegyezzük meg, hogy nincs olyan intervallum a jobb részfában, amely átfedné i-t – majd később látni fogjuk, hogy miért.) A $[8, 9]$ intervallum nem fedi át i-t, és $max[bal[x]] = 10$, ami kisebb, mint $alsó[i] = 11$, tehát jobbra megyünk. (Jegyezzük meg, hogy nincs olyan intervallum a bal részfában, amely átfedné i-t.) A $[15, 23]$ intervallum nem fedi át i-t és a bal gyerek $nil[T]$, így az ismétlés véget ér, az eredmény $nil[T]$.

Az INTERVALLUMOT-KERES algoritmus helyességének igazolásához meg kell értenünk, hogy miért elegendő egyetlen, a gyökértől induló utat vizsgálni. Az alapötlet az, hogy minden x csúcsban, ha $int[x]$ nem fedi át i-t, akkor a keresés jó irányban folytatódik: mindig találunk átfedő intervallumot, ha egyáltalán létezik a fában ilyen. A következő tétel pontosabban fejezi ki ezt a tulajdonságot.

14.2. tétel. *Az* INTERVALLUMOT-KERES(T, i) *eljáráshívás minden végrehajtása vagy olyan csúcsot ad, amelyben lévő intervallum átfedi i-t, vagy a $nil[T]$-t, és ekkor nincs a T fában i-t átfedő intervallum.*



14.5. ábra. A 14.2. tétel bizonyításában szereplő intervallumok. A függőleges szaggatott vonal mindkét esetben a $max[bal[x]]$ értéket jelöli. **(a)** A keresés jobbra tart. Nincs olyan i' intervallum x bal részfájában, amely átfedné i-t. **(b)** A keresés balra tart. Az x csúcs bal részfájában van olyan intervallum, amely átfedi i-t (ezt az esetet nem mutatja az ábra), vagy van olyan i' intervallum x bal részfájában, hogy $felső[i'] = max[bal[x]]$. Mivel i és i' nem átfedők, i nem átfedő egyetlen olyan i'' intervallummal sem, amely x jobb részfájában van, mivel $alsó[i'] \leq alsó[i'']$.

Bizonyítás. A 2–5. sorokban leírt ***while*** ciklus akkor ér véget, ha vagy $x = nil[T]$, vagy i átfedi az $int[x]$ intervallumot. Ezért az első esetre összpontosítunk, amikor a ***while*** ciklus úgy fejeződik be, hogy $x = nil[T]$.
A következő invariánst használjuk a 2–5. sorokban a ***while*** ciklusra:

> Ha a T fa tartalmaz olyan intervallumot, amely átfedi i-t, akkor az x-gyökerű részfában is van i-t átfedő intervallum.

Ezt a ciklusinvariánst az alábbiak szerint használjuk.

Teljesül: A ciklusmag első végrehajtása előtt az 1. sorban x értéke felveszi a T fa gyökerét, tehát az invariáns teljesül.

Megmarad: A ***while*** ciklus minden ismétlésében vagy a 4., vagy az 5. sor végrehajtódik. Meg fogjuk mutatni, hogy mindkét esetben teljesül az invariáns.

Ha az 5. sor hajtódik végre, akkor a 3. sorban végrehajtott teszt miatt teljesül, hogy $bal[x] = nil[T]$ vagy $max[bal[x]] < alsó[i]$. Ha $bal[x] = nil[T]$, akkor a $bal[x]$ gyökerű részfa biztosan nem tartalmaz i-t átfedő intervallumot, mivel egyáltalán nem tartalmaz intervallumot, tehát mivel x felveszi a $jobb[x]$ értéket, teljesül az invariáns. Tegyük fel, hogy $bal[x] \neq nil[T]$ és $max[bal[x]] < alsó[i]$. Mint a 14.5(a) ábra mutatja, minden olyan i' intervallumra, amely x bal részfájában van, teljesül, hogy

$$\begin{aligned} felső[i'] &\leq max[bal[x]] \\ &< alsó[i]. \end{aligned}$$

Az intervallum trichotómia miatt i' és i nem átfedők. Ezért x bal részfája nem tartalmazhat i-t átfedő intervallumot, tehát mivel x felveszi a $jobb[x]$ értéket, teljesül az invariáns.

Ha a 4. sor hajtódik végre, akkor meg fogjuk mutatni, hogy az invariáns tagadása teljesül. Vagyis ha nincs i-t átfedő intervallum a $bal[x]$ gyökerű részfában, akkor nincs i-t átfedő intervallum a teljes fában. Mivel a 4. sor végrehajtódott, a 3. sorbeli feltétel ellenőrzés miatt $max[bal[x]] \geq alsó[i]$. Továbbá a max mező definíciója miatt x bal részfájában kell lennie olyan i' intervallumnak, amelyre teljesülnek a következők.

$$\begin{aligned} felső[i'] &= max[bal[x]] \\ &\geq alsó[i]. \end{aligned}$$

(A 14.5(b) ábra szemlélteti ezt az esetet.) Mivel i és i' nem átfedők, és nem igaz, hogy $felső[i'] < alsó[i]$, így az intervallum trichotómia miatt azt kapjuk, hogy $felső[i] < alsó[i']$. Intervallum-fákban a kulcs az intervallum alsó végpontja, így a keres őfa tulajdonság miatt teljesül, hogy x jobb részfájában található minden i'' intervallumra igaz, hogy

$$\begin{aligned} felső[i] &< alsó[i'] \\ &\leq alsó[i''] . \end{aligned}$$

Az intervallum trichotómia miatt i és i'' nem átfedők. Arra a következtetésre jutottunk, hogy akár létezik i-t átfedő intervallum x bal részfájában, akár nem, az $x \leftarrow bal[x]$ értékadás fenntartja az invariáns teljesülését.

Befejeződik: Ha a ciklus az $x = nil[T]$ miatt fejeződik be, akkor nincs olyan intervallum az x-gyökerű részfában, amely átfedné i-t. Az invariáns ellenkez őjéből következik, hogy T nem tartalmaz i-t átfedő intervallumot. Tehát helyes, hogy az eredmény $x = nil[T]$. ■

Tehát az INTERVALLUMOT-KERES algoritmus helyesen működik.

Gyakorlatok

14.3-1. Írjuk meg a BALRA-FORGAT algoritmus intervallum-fákra használható olyan változatát, amely a max mezőt $O(1)$ időben aktualizálja.

14.3-2. Írjuk meg az INTERVALLUMOT-KERES algoritmust arra az esetre, amikor minden intervallum nyitott.

14.3-3. Tervezzünk olyan hatékony algoritmust, amely adott i intervallumhoz megad egy olyan i-t átfedő intervallumot, amelynek az alsó végpontja a legkisebb. Az eredmény $nil[T]$ legyen, ha nincs i-t átfedő intervallum.

14.3-4. Adott egy T intervallum-fa és egy i intervallum. Mutassuk meg, hogyan lehet $O(\min(n, k \lg n))$ időben felsorolni az összes olyan intervallumot, amely átfedi i-t, ahol k az átfedő intervallumok száma, n pedig a fa csúcsainak száma. (*Változat.* Keressünk olyan megoldást, amely nem módosítja a fát.)

14.3-5. Módosítsuk az intervallum-fa algoritmusokat úgy, hogy az támogassa az INTERVALLUMOT-PONTOSAN-KERES(T, i) művelet olyan megvalósítását, amely a T fa olyan x csúcsát adja eredményül, amelyre $alsó[int[x]] = alsó[i]$, $felső[int[x]] = felső[i]$, illetve $nil[T]$-t, ha T-ben nincs ilyen csúcs. Minden művelet algoritmusa $O(\lg n)$ idejű legyen n csúcsú fára, beleértve az INTERVALLUMOT-PONTOSAN-KERES műveletet is.

14.3-6. Mutassuk meg, hogyan lehet megvalósítani olyan dinamikus halmazt, amelynek elemei számok, és van egy MIN-TÁV művelete, amely adott Q halmaz esetén a halmaz két legközelebbi elemének a különbségét adja. Például, ha $Q = \{1, 5, 9, 15, 18, 22\}$, akkor az MIN-TÁV($Q$) művelet eredménye $18 - 15 = 3$, mivel a két legközelebbi szám Q-ban 15 és 18. Készítsük el a BESZÚR, TÖRÖL, KERES és MIN-TÁV műveletek lehető leghatékonyabb megvalósítását, és elemezzük az algoritmusok időigényét.

14.3-7.⋆ A VLSI adatbázisok gyakran integrált áramkörök, mint téglalapok halmazát ábrázolják. Tegyük fel, hogy minden téglalap olyan, hogy oldalai párhuzamosak az x, illetve az y tengellyel, tehát minden téglalap ábrázolható a téglalapot alkotó pontok minimális és maximális x és y koordinátáival. Tervezzünk olyan $O(n \lg n)$ idejű algoritmust, amely eldönti, hogy az így ábrázolt n elemű halmaz tartalmaz-e két, egymást átfed ő téglalapot. Az algoritmusnak nem kell megadnia az összes átfedő téglalapot. Két téglalapot akkor is egymást

átfedőnek tekintünk, ha egyik teljesen tartalmazza a másikat, de a téglalapok határai nem metszik egymást. (*Útmutatás*. Pásztázza be egy „seprő” egyenessel a téglalapok halmazát.)

Feladatok

14-1. Maximális átfedő pont

Tegyük fel, hogy nyilván akarunk tartani intervallumok egy halmazához egy ***maximális átfedő pontot,*** amely olyan pont, amelyet a legtöbb intervallum átfed.

a. Mutassuk meg, hogy mindig van olyan maximális átfedő pont, amely valamelyik intervallum végpontja.

b. Tervezzünk olyan adatszerkezetet, amely hatékonyan támogatja az INTERVALLUMOT-BESZÚR, INTERVALLUMOT-TÖRÖL és MÁP-KERES műveleteket, ahol az utóbbi művelet egy maximális átfedő pontot ad. (*Útmutatás*. Használjunk olyan piros-fekete fát, amely minden intervallum mindkét végpontját tartalmazza. A +1 értékeket rendeljük a bal végpontokhoz, a −1 értéket pedig a jobb végpontokhoz. Bővítsük ki a fa pontjait olyan extra információval, amely segíti maximális átfedő pont fenntartását.)

14-2. Jozefusz-permutációk

A ***Jozefusz-probléma*** a következőt jelenti. Tegyük fel, hogy egy kör alakú asztal körül n ember foglal helyet, és adott egy $m \leq n$ pozitív egész szám. Egy kijelölt helytől indulva, körbe haladva minden m-edik embert kizárunk a társaságból. A kizárt embereket nem számítjuk a számlálás során körbe haladva. Ezt a kizárást mindaddig folytatjuk, amíg mindenkit ki nem zártunk. Az n ember kizárásának sorrendje az $1, \ldots, n$ számok egy permutációját definiálja, ezt nevezzük ***(n,m)-Jozefusz-permutációnak.*** Például a (7, 3)-Jozefusz-permutáció a (3, 6, 2, 7, 5, 1, 4) számsorozat lesz.

a. Tegyük fel, hogy m állandó. Tervezzünk olyan $O(n)$ idejű algoritmust, amely adott n pozitív egész számra kiszámítja az (n, m)-Jozefusz-permutációt.

b. Tegyük fel, hogy m nem állandó érték. Tervezzünk olyan $O(n \lg n)$ idejű algoritmust, amely adott n és m pozitív egész számokra kiszámítja az (n, m)-Jozefusz-permutációt.

Megjegyzések a fejezethez

Preparata és Shamos [247] számos, az irodalomban előforduló intervallum-fát tárgyal, idézve H. Edelsbrunner (1980) és E. M. McCreight (1981) munkáit. A könyv részletesen tárgyal olyan intervallum-fát, amely intervallumok n elemű halmazára és adott intervallumra $O(k + \lg n)$ időben felsorolja az adott intervallumot átfedő k darab intervallumokat.

IV. FEJLETT ELEMZÉSI ÉS TERVEZÉSI MÓDSZEREK

Bevezetés

Ez a rész hatékony algoritmusok tervezésének és elemzésének három fontos módszerét, a dinamikus programozást (15. fejezet), a mohó módszert (16. fejezet) és az amortizált elemzést (17. fejezet) tárgyalja. Ezek a módszerek – a korábban megismert oszd-meg-és-uralkodj, véletlenítés és rekurzió módszerekhez hasonlóan – széles körben alkalmazhatók. Az új módszerek a korábbiaknál bonyolultabbak. Az ebben a részben feldolgozott témák a későbbiekben vissza fognak térni.

A dinamikus programozást tipikusan olyan helyzetekben alkalmazzák, amikor döntések sorozatát kell meghozni ahhoz, hogy elérjünk egy optimális megoldást. Gyakran az a helyzet, hogy mihelyst meghoztunk egy döntést, a keletkező részprobléma hasonló az eredetihez. A dinamikus programozás akkor hatékony, ha egy adott részprobléma több különböző döntési sorozat nyomán is keletkezhet. A kulcskérdés az, hogy hogyan tároljuk, azaz „memorizáljuk” a részprobléma megoldását arra az esetre, ha ismét előfordulna. A 15. fejezet azt mutatja be, hogy ez az egyszerű gondolat hogyan képes néha időben exponenciális algoritmusokat polinom idejű algoritmusokká alakítani.

A dinamikus programozáshoz hasonlóan a mohó módszert is jellemzően akkor alkalmazzuk, amikor döntések egy sorozatát kell meghozni ahhoz, hogy elérjünk egy optimális megoldást. A mohó módszer alapgondolata, hogy minden döntést lokálisan optimális módon hozzunk meg. Egy egyszerű példa erre a pénzváltás: ahhoz, hogy egy dollárban megadott összeget minimális számú pénzdarabbal kifizessünk, mindig a lehetséges legnagyobb címletet kell használni annyiszor, ahányszor a maradék összeg azt megengedi. Nagyon sok olyan probléma van, amire a mohó módszer sokkal gyorsabban ad optimális megoldást, mint ahogy azt a dinamikus programozás tenné. Azonban nem mindig könnyű megmondani, hogy a mohó megközelítés hatékony lesz-e. A 16. fejezet áttekinti a matroidelméletet, ami gyakran segít a kérdés eldöntésében.

Az amortizált elemzés olyan algoritmusok vizsgálatának eszköze, amelyek hasonló műveletek sorozatát hajtják végre. Ahelyett, hogy a műveletek sorozatának költségét az egyes műveletek költségének becsléséből adnánk meg, az amortizált elemzés az egész sorozat tényleges költségének becslését képes megadni. Hogy ez az elv hatékony lehet, annak egyik oka az, hogy többnyire lehetetlen, hogy műveletek egy sorozatában minden művelet a lehető leghosszabb ideig tartson. Míg néhány művelet drága lehet, addig sok másik meg éppen olcsó. Az amortizált elemzés nem egyszerűen az elemzés egy eszköze, hanem egyúttal egy mód is arra, hogy algoritmusok tervezéséről hogyan gondolkodjunk, mivel egy algoritmus tervezése és futási idejének elemzése gyakran szorosan összefonódik. A 17. fejezet egy algoritmus amortizált elemzésének három ekvivalens módját ismerteti.

15. Dinamikus programozás

A dinamikus programozás éppúgy, mint az oszd-meg-és-uralkodj módszer, a feladatot részfeladatokra való osztással oldja meg. (A „programozás” szó ebben az összefüggésben egy táblázatos módszerre utal, nem egy számítógépes program írására.) Mint azt a 2. fejezetben láttuk, az oszd-meg-és-uralkodj módszer felosztja a feladatot független részfeladatokra, melyeket ismételten megold, és a részfeladatok megoldásait az eredeti feladat megoldása céljából egyesíti. Ezzel szemben a dinamikus programozás akkor alkalmazható, ha a részproblémák nem függetlenek, azaz közös részproblémáik vannak. Ilyen helyzetben az oszd-meg-és-uralkodj módszer a szükségesnél többet dolgozik, mert ismételten megoldja a részproblémák közös részproblémáit. A dinamikus programozás minden egyes részfeladatot és annak minden részfeladatát pontosan egyszer oldja meg, az eredményt egy táblázatban tárolja, és ezáltal elkerüli az ismételt számítást, ha a részfeladat megint felmerül.

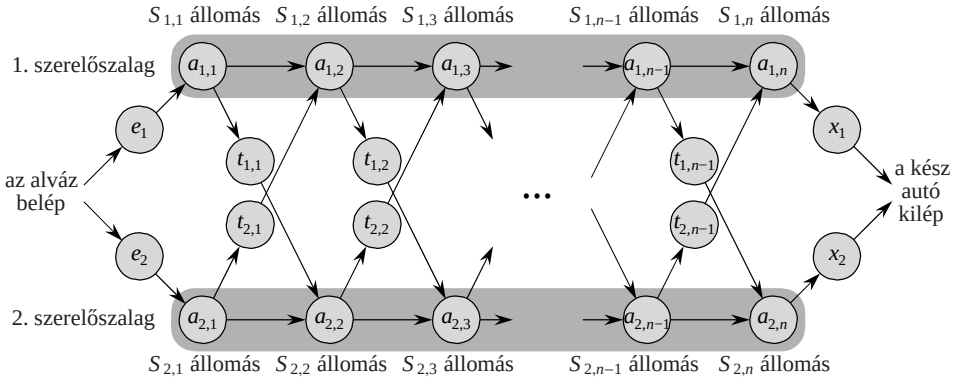
A dinamikus programozást ***optimalizálási feladatok*** megoldására használjuk. Ilyen feladatoknak sok megengedett megoldása lehet. Mindegyiknek van értéke, és mi az optimális (minimális vagy maximális) értékűt kívánjuk megtalálni. Egy ilyen megoldást *egy optimális megoldásnak* nevezzük, nem pedig *az optimális megoldásnak*, mivel más megoldások is elérhetik az optimális értéket.

Egy dinamikus programozási algoritmus kifejlesztése négy lépésre bontható fel:

1. Jellemezzük az optimális megoldás szerkezetét.
2. Rekurzív módon definiáljuk az optimális megoldás értékét.
3. Kiszámítjuk az optimális megoldás értékét alulról felfelé történő módon.
4. A kiszámított információk alapján megszerkesztünk egy optimális megoldást.

Az 1–3. lépések jelentik az alapját annak, ahogy a dinamikus programozás meg tud oldani egy feladatot. A 4. lépés elhagyható, ha csak az optimális értékre vagyunk kíváncsiak. Ha végre kell hajtanunk a 4. lépést, akkor a 3. lépésben gyakran olyan kiegészítő információkat határozunk meg, amelyek megkönnyítik az optimális megoldás előállítását.

A következő alfejezetekben a dinamikus programozást néhány optimalizálási probléma megoldására használjuk fel. A 15.1. alfejezetben két, autókat gyártó szerelőszalag ütemezését vizsgáljuk abban az esetben, amikor a szerelés alatt álló autó minden állomás után vagy továbbra is ugyanazon a szalagon marad vagy átmehet a másik szalagra. A 15.2. alfejezetben azt a kérdést vetjük fel, hogy mátrixok egy véges sorozatát hogyan lehet összeszorozni úgy, hogy a lehető legkevesebb skalár szorzást végezzük. Ezen példák alapján a 15.3. alfejezetben tárgyaljuk azt a két alapvető tulajdonságot, amivel egy feladatnak rendelkeznie kell



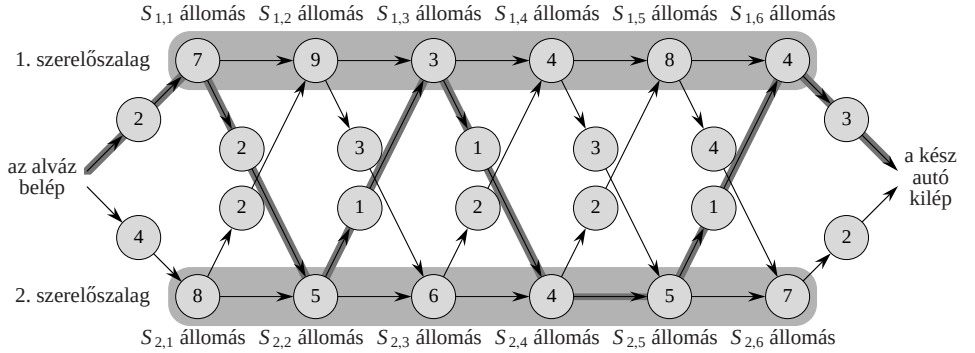
15.1. ábra. Egy üzemen való leggyorsabb áthaladás megkeresése. Két szerelőszalag van, mindkettőben n állomás; az i-edik szalag j-edik állomását S_{ij} jelöli, és a szerelés itt a_{ij} ideig tart. Az autó alváza belép az üzembe, majd felkerül az i-edik szalagra (ahol i 1 vagy 2), ami e_i ideig tart. Miután az alváz keresztül ment a j-edik állomáson, valamelyik szalag $(j + 1)$-edik állomására kerül. Ugyanazon a szalagon nincs költsége az alváz továbbításának, de t_{ij} ideig tart, ha a másik szalagról kerül az S_{ij} állomásra. Az n-edik állomás elhagyása után x_i ideig tart, amíg a kész autó elhagyja az üzemet. A feladat az, hogy kiválasszuk, hogy mely állomásokon szereljenek az első, illetve a második szalagon, hogy egy autó a lehetséges minimális idő alatt haladjon keresztül az üzemen.

ahhoz, hogy esetében a dinamikus programozás használható megoldási módszer legyen. A 15.4. alfejezetben azt mutatjuk meg, hogy két sorozat leghosszabb közös részsorozatát hogyan lehet megtalálni. Végül a 15.5. alfejezetben dinamikus programozással optimális bináris kereső fákat szerkesztünk, amikor ismert a keresendő kulcsok eloszlása.

15.1. Szerelőszalag ütemezése

A dinamikus programozásra vonatkozó első példánk egy ipari problémát old meg. A Colonel Motors Corporation autókat állít elő az egyik gyárában, amelynek két szerelőszalagja van, mint azt a 15.1. ábra mutatja. Mindkét szerelőszalag elején egy alváz jelenik meg, melyhez adott számú állomáson hozzászerelik az alkatrészeket, majd a szalag végén távozik a kész autó. Mindegyik szalagnak n állomása van, melyek indexei $j = 1, 2, \ldots, n$. S_{ij} jelöli az i-edik ($i = 1$ vagy 2) szalag j-edik állomását. Az első szalag j-edik állomása (S_{1j}) ugyanazt a funkciót látja el, mint a második szalag j-edik állomása (S_{2j}). Azonban a szalagokat különböző időpontokban különböző technológiával építették, ezért előfordulhat, hogy a két szalag azonos állomásán a terméknek különböző időt kell eltöltenie. Az S_{ij} állomáson a műveleti időt a_{ij} jelöli. A 15.1. ábra azt mutatja, ahogy az alváz belép valamelyik szalag első állomására, és innen hogyan halad tovább állomásról állomásra. Ahhoz is kell egy bizonyos idő, hogy az alváz a szalagra kerüljön, és ahhoz is, hogy az elkészült autó elhagyja a szalagot. Ezeket az i-edik szalag esetén e_i, illetve x_i jelöli.

Normális esetben, amikor az alváz belép egy szalagra, akkor csak ezen a szalagon megy végig. Ugyanazon a szalagon egy állomásról a következő állomásra a termék elhanyagolható idő alatt megy át. Esetenként sürgős rendelések érkeznek. Ilyenkor a vevő azt kívánja, hogy az autó olyan gyorsan készüljön el, amilyen gyorsan csak lehet. Ilyen esetekben az autó az n-edik állomáson továbbra is a megfelelő sorrendben halad végig, de a gyár vezetője a félig kész kocsit bármely állomás után átrakhatja a másik szalagra. Az i-edik szalag S_{ij} ál-



(a)

j	1	2	3	4	5	6	
$f_1[j]$	9	18	20	24	32	35	$f^* = 38$
$f_2[j]$	12	16	22	25	30	37	

j	2	3	4	5	6	
$l_1[j]$	1	2	1	1	2	$l^* = 1$
$l_2[j]$	1	2	1	2	2	

(b)

15.2. ábra. (a) Példa a szerelőszalag feladatára, ahol az ábrán feltüntettük az e_i, a_{ij}, t_{ij} és x_i költségeket. A vastagon rajzolt út a legrövidebb. (b) Az (a)-beli példa esetében az $f[j]$, f^*, $l_i[j]$ és l^* értékek.

lomása után az alváznak a másik szalagra való átrakása t_{ij} időbe kerül ($\{j = 1, 2, \ldots, n-1\}$, mivel n állomás után a kocsi már kész). A feladat az, hogy meghatározzuk, hogy mely állomásokat érintse az egyik és melyeket a másik szalagon, hogy az átfutási idő minimális legyen. A 15.2(a) ábra példájában az 1. szalagon teljesíti az 1., 3. és 6. állomásokat, a 2., 4., 5. állomásokat pedig a 2. szalagon.

A számítások nem végezhetők el a természetes „nyers erővel”, azaz teljes leszámlálással sok állomás esetén. Ha adott, hogy mely állomásokat kell igénybe venni az első szalagon és melyeket a másodikon, akkor az átfutási idő $\Theta(n)$ idő alatt kiszámítható. Sajnos azonban 2^n választási lehetőség van, amit úgy láthatunk be, hogy az első szalagon igénybe vett állomások halmaza egy részhalmaza $1, 2, \ldots, n$-nek, és ezt 2^n-féleképpen lehet megválasztani. Ezért a teljes leszámlálás számítási ideje $\Omega(2^n)$, ami nagy n mellett elfogadhatatlan.

1. lépés: a legrövidebb gyártási útszerkezete

A dinamikus programozás első lépése, hogy jellemezzük az optimális megoldást. A szerelőszalag ütemezése esetében ezt a következőképpen tehetjük meg. Tekintsük a lehetséges legrövidebb utat, ahogy egy alváz a kiindulási helyzetből túl jut az S_{1j} állomáson. Ha $j = 1$, akkor csak egy lehetőség van, és így könnyű meghatározni a szükséges időt. Azonban $j = 2, 3, \ldots, n$ esetén két lehetőség van: Az alváz érkezhet közvetlenül az $S_{1,j-1}$ állomásról, amikor is ugyanannak a szalagnak a $(j-1)$-edik állomásáról a j-edik állomására való átmenetel ideje elhanyagolható. A másik lehetőség, hogy az alváz $S_{2,j-1}$ felől érkezik. Az átszállítás ideje $t_{2,j-1}$. Bár külön-külön meg kell vizsgálnunk ezt a két lehetőséget, látni fogjuk, hogy sok közös vonásuk van.

Először tegyük fel, hogy az S_{1j} állomáson való túljutás leggyorsabb módja az, hogy oda az $S_{1,j-1}$ felől érkezünk. A dolog kulcsa az az észrevétel, hogy az alváznak a legrövidebb

módon kell túljutnia az $S_{1,j-1}$ állomáson. Miért? Azért, mert ha volna egy gyorsabb mód, hogy az alváz túljusson az $S_{1,j-1}$ állomáson, akkor ezt használva magán S_{1j}-n is hamarább jutna túl, ami ellentmondás.

Hasonlóképpen tegyük fel, hogy a leggyorsabb út az, amikor az alváz $S_{2,j-1}$ felől érkezik. Ekkor a kiindulási ponttól indulva a legrövidebb módon kell túljutnia $S_{2,j-1}$-en. Az indoklás hasonló, mint az előbb: ha volna egy gyorsabb mód, hogy az alváz túljusson az $S_{2,j-1}$ állomáson, akkor ezt használva magán S_{1j}-n is hamarább jutna túl, ami ismét ellentmondás.

Általánosabban fogalmazva azt mondhatjuk, hogy a szerelőszalag ütemezésének (hogy megtaláljuk a legrövidebb módot az S_{ij} állomáson való túljutásra) optimális megoldása tartalmazza egy részfeladat (vagy az $S_{1,j-1}$, vagy az $S_{2,j-1}$ állomáson való leggyorsabb áthaladás) optimális megoldását. Erre a tulajdonságra ***optimális részstruktúraként*** fogunk hivatkozni. Mint azt a 15.3. alfejezetben látni fogjuk, ez egyike azoknak a dolgoknak, amelyek a dinamikus programozás alkalmazhatóságát fémjelzik.

Az optimális részstruktúra tulajdonságot használjuk fel annak megmutatására, hogy részfeladatok optimális megoldásaiból megszerkeszthető a feladat optimális megoldása. A szerelőszalag ütemezése esetén a következő módon érvelhetünk. Ha a leggyorsabb módon túljutunk az S_{1j} állomáson, akkor túl kell jutnunk a $(j-1)$-edik állomáson vagy az első vagy a második szalagon. Ezért a leggyorsabb út az S_{1j} állomáson keresztül vagy

- a leggyorsabb út az $S_{1,j-1}$ állomáson keresztül, és onnan közvetlenül megyünk az S_{1j} állomásra, vagy
- a leggyorsabb út az $S_{2,j-1}$ állomáson keresztül, ahonnan az alvázat át kell szállítani az első szalag S_{1j} állomására.

Mindkét esetben természetesen el kell végezni a munkát az S_{1j} állomáson. Hasonlóképpen a leggyorsabb út az S_{2j} állomáson keresztül vagy

- a leggyorsabb út az $S_{2,j-1}$ állomáson keresztül, és onnan közvetlenül megyünk az S_{2j} állomásra,
- a leggyorsabb út az $S_{1,j-1}$ állomáson keresztül, ahonnan az alvázat át kell szállítani a második szalag S_{2j} állomására.

Ahhoz, hogy bármelyik szalag j-edik állomásán keresztülvezető legrövidebb utat megtaláljuk, mindkét szalag $(j-1)$-edik állomásán keresztüli legrövidebb út megkeresésére van szükség.

Tehát a szerelőszalag ütemezésének optimális megoldását felépíthetjük a részfeladatok optimális megoldásaiból.

2. lépés: a rekurzív megoldás

A dinamikus programozás alkalmazásának második lépése, hogy az optimális megoldás értékét a részfeladatok optimális értékeiből rekurzívan fejezzük ki. A szerelőszalag ütemezése esetén a részfeladatok legyenek mindkét szalag j-edik állomásán való leggyorsabb áthaladás megkeresésének a feladatai $j = 1, 2, \ldots, n$ mellett. Jelölje $f_i[j]$ azt a legrövidebb időt, ami alatt az alváz túl tud jutni az S_{ij} állomáson.

A végső célunk annak a legrövidebb időnek a meghatározása, ami alatt az alváz az egész üzemen keresztül tud haladni. Ezt az időt f^* jelöli. Az alváznak át kell haladnia az

első vagy a második szalag n-edik állomásán, és utána el kell hagynia a gyárat. Mivel a két lehetőség közül a gyorsabbik az egész üzemen átvezető leggyorsabb lehetőség, azt kapjuk, hogy

$$f^* = \min\{f_1[n] + x_1,\ f_2[n] + x_2\}. \tag{15.1}$$

$f_1[1]$ és $f_2[1]$ értékét szintén könnyű meghatározni. Mindkét szalag esetében az első állomáson való keresztülhaladás egyszerűen azt jelenti, hogy az alváz egyenesen erre az állomásra megy, azaz

$$f_1[1] = e_1 + a_{11}, \tag{15.2}$$
$$f_2[1] = e_2 + a_{21}. \tag{15.3}$$

Most vizsgáljuk meg, hogy miként kell kiszámítani $f_i[j]$-t $j = 2, 3, \ldots, n$ és $i = 1, 2$ esetén. Emlékeztetünk rá, hogy $f_1[j]$-t vagy az $S_{1,j-1}$ állomáson áthaladó legrövidebb út és innen közvetlenül S_{1j}-n való áthaladás adja, vagy az $S_{2,j-1}$ állomáson áthaladó legrövidebb út, majd az alváznak a második szalagról az első re való átszállítása és ismét az S_{1j}-n való áthaladás adja. Az első esetben $f_1[j] = f_1[j-1] + a_{1j}$, a másodikban pedig $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1j}$. Tehát

$$f_1[j] = \min\{f_1[j-1] + a_{1j},\ f_2[j-1] + t_{2,j-1} + a_{1j}\}, \tag{15.4}$$

ha $j = 2, 3, \ldots, n$. Ehhez hasonlóan

$$f_2[j] = \min\{f_2[j-1] + a_{2j},\ f_1[j-1] + t_{1,j-1} + a_{2j}\}, \tag{15.5}$$

ha $j = 2, 3, \ldots, n$. A (15.2)–(15.5) egyenletekből a következő rekurzív egyenleteket kapjuk:

$$f_1[j] = \begin{cases} e_1 + a_{11}, & \text{ha } j = 1, \\ \min\{f_1[j-1] + a_{1j},\ f_2[j-1] + t_{2,j-1} + a_{1j}\}, & \text{ha } j \geq 2, \end{cases} \tag{15.6}$$

és

$$f_2[j] = \begin{cases} e_2 + a_{21}, & \text{ha } j = 1, \\ \min\{f_2[j-1] + a_{2j},\ f_1[j-1] + t_{1,j-1} + a_{2j}\}, & \text{ha } j \geq 2. \end{cases} \tag{15.7}$$

A 15.2(b) ábra az (a) rész példájához a (15.6) és (15.7) egyenletek alapján számított $f_1[j]$ és $f_2[j]$ értékeket mutatja f^*-gal együtt.

Az $f_i[j]$ mennyiségek a részfeladatok optimális értékei. Annak érdekében, hogy vissza tudjuk keresni a legrövidebb utat, definiáljuk $l_i[j]$-t mint azt a szerelőszalagot, amelyiknek a $j-1$-edik állomását használtuk az S_{ij}-n való leggyorsabb keresztülhaladáskor. Itt $i = 1, 2$ és $j = 2, 3, \ldots, n$. (Nem definiáljuk $l_i[1]$-et, mert S_{1i}-t egyik szalagon sem előzi meg másik állomás.) Az l^* szalag pedig definíció szerint az, amelyiknek az utolsó állomását használjuk az egész üzemen való leggyorsabb áthaladáskor. Az $l_i[j]$ értékek segítségével lehet nyomon követni a legrövidebb utat. A 15.2(b) ábrán feltüntetett l^*, $l_i[j]$ értékek segítségével rekonstruálható az ábra (a) részén bejelölt legrövidebb út. Mivel $l^* = 1$, ezért az S_{16} állomást használjuk. Mivel $l_1[6] = 2$, ezért ide az S_{25} állomásról érkeztünk. Így folytatva $l_2[5] = 2$ az S_{24}, $l_2[4] = 1$ az S_{13}, $l_1[3] = 2$ az S_{22} és $l_2[2] = 1$ az S_{11} állomás használatát jelenti.

3. lépés: a legrövidebb átfutási idő kiszámítása

Ezen a ponton már egyszerű egy rekurzív algoritmust írni a (15.1), (15.6) és (15.7) egyenletek alapján a legrövidebb átfutási idő kiszámítására. Azonban van egy probléma az ilyen algoritmussal, nevezetesen n-ben exponenciális ideig fut. Hogy ennek okát lássuk, legyen $r_i(j)$ az a szám, ahányszor az algoritmus a lefutása során hivatkozik $f_i[j]$-re. A (15.1) egyenlet alapján

$$r_1(n) = r_2(n) = 1. \tag{15.8}$$

A (15.6) és (15.7) rekurzív egyenletek szerint pedig

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1), \tag{15.9}$$

ahol $j = 1, 2, \ldots, n-1$. A 15.1-2. gyakorlat annak bizonyítását kéri, hogy $r_i(j) = 2^{n-j}$. Tehát egyedül $f_1[1]$-re 2^{n-1}-szer hivatkozunk. A 15.1-3. gyakorlat annak megmutatását írja elő, hogy az összes $f_i[j]$-re való hivatkozások száma $\Theta(2^n)$.

Sokkal jobban járunk, ha az $f_i[j]$ értékeket más sorrend szerint számoljuk, mint az a rekurzív módszerből adódik. Vegyük észre, hogy $j \geq 2$ esetén $f_i[j]$ csak $f_1[j-1]$-től és $f_2[j-1]$-től függ. Ha az $f_i[j]$ értékeket az állomások j indexének *növekvő* sorrendjében – a 15.2(b) ábrán balról jobbra – számoljuk, akkor a legrövidebb átfutási id ő meghatározása $\Theta(n)$ ideig tart. A LEGRÖVIDEBB-ÁTFUTÁSI-IDŐ eljárás bemenő paraméterei az a_{ij}, t_{ij}, e_i és x_i értékek, valamint mindkét szalag állomásainak n száma.

LEGRÖVIDEBB-ÁTFUTÁSI-IDŐ(a, t, e, x, n)

```
 1  f_1[1] ← e_1 + a_11
 2  f_2[1] ← e_2 + a_21
 3  for j ← 2 to n
 4      do if f_1[j − 1] + a_1j ≤ f_2[j − 1] + t_2,j−1 + a_1j
 5          then f_1[j] ← f_1[j − 1] + a_1,j
 6               l_1[j] ← 1
 7          else f_1[j] ← f_2[j − 1] + t_2,j−1 + a_1j
 8               l_1[j] ← 2
 9          if f_2[j − 1] + a_2j ≤ f_1[j − 1] + t_1,j1 + a_2j
10          then f_2[j] ← f_2[j − 1] + a_2,j
11               l_2[j] ← 2
12          else f_2[j] ← f_1[j − 1] + t_1,j−1 + a_2,j
13          l_2[j] ← 1
14  if f_1[n] + x_1 ≤ f_2[n] + x_2
15      then f* = f_1[n] + x_1
16           l* = 1
17      else f* = f_2[n] + x_2
18           l* = 2
```

LEGRÖVIDEBB-ÁTFUTÁSI-IDŐ eljárás a következőképpen dolgozik. Az első két sorban $f_1[1]$ és $f_2[1]$ kiszámítása történik a (15.2) és (15.3) képlet alapján. Ezután a **for** ciklus a 3–13. sorokban az $f_i[j]$, $l_i[j]$ értékeket határozza meg $i = 1, 2$ és $j = 2, 3, \ldots, n$ esetén. A 4–8. sorok $f_1[j]$-t és $l_1[j]$-t számítják a (15.4) képlet alapján, míg a 9–13. sorok $f_2[j]$-t és $l_2[j]$-t (15.5) alapján. Végül a 14–18. sorok f^*-ot és l^*-ot adják meg a (15.1) képlet felhasználásával.

Mivel az 1–2. és 14–18. sorok elvégzése konstans ideig tart, hasonlóképpen a 3–13. sorok **for** ciklusának mind az $n - 1$ végrehajtása, az egész eljárás $\Theta(n)$ idejű.

$f_i[j]$ és $l_i[j]$ kiszámítását úgy is tekinthetjük, hogy kitöltjük egy táblázat elemeit. Ha a 15.2(b) ábrára gondolunk, akkor az $f_i[j]$-t és $l_i[j]$-t tartalmazó táblát balról jobbra és oszloponként felülről lefelé töltjük ki. $f_i[j]$ kiszámításához csak $f_1[j-1]$-re és $f_2[j-1]$-re van szükségünk. Tudván, hogy ezeket már kiszámoltuk és tároltuk, az adott pillanatban a meghatározásuk mindössze abból áll, hogy kiolvassuk őket a táblázatból.

4. lépés: a legrövidebb átfutási idejű út megszerkesztése

$f_i[j]$, f^*, $l_i[j]$ és l^* kiszámítását követően meg kell szerkeszteni az üzemen való legrövidebb áthaladást biztosító utat. A 15.2. ábra példája esetében a fentiekben már megmutattuk, hogyan kell ezt csinálni.

Az alábbi eljárás az út állomásait az indexek csökkenő sorrendjében nyomtatja ki. A 15.1-1. gyakorlat azt kívánja, hogy úgy változtassuk meg, hogy a nyomtatás növekvő sorrendben történjen.

```
ÁLLOMÁSOK-NYOMTATÁSA(l, n)
1  i ← l*
2  print i„-edik szalag ”n„-edik állomás”
3  for j ← n downto 2
4       do i ← l_i[j]
5          print i„-edik szalag (”j − 1„)-edik állomás”
```

A 15.2. ábra példáján az ÁLLOMÁSOK-NYOMTATÁSA eljárás a következő eredményt adja:

1. szalag, 6. állomás
2. szalag, 5. állomás
2. szalag, 4. állomás
1. szalag, 3. állomás
2. szalag, 2. állomás
1. szalag, 1. állomás.

Gyakorlatok

15.1-1. Hogyan kell az ÁLLOMÁSOK-NYOMTATÁSA eljárást módosítani, hogy az állomásokat számuk növekvő sorrendjében nyomtassa ki. (*Útmutatás.* Használjunk rekurziót.)

15.1-2. A (15.8) és (15.9) egyenlet és a helyettesítés módszerének felhasználásával mutassuk meg, hogy a rekurzív eljárásban az $f_i[j]$-re történő hivatkozások $r_i(j)$ száma 2^{n-j}.

15.1-3. A 15.1-2. gyakorlat eredményére támaszkodva mutassuk meg, hogy az $f_i[j]$ értékekre történő összes hivatkozások száma, vagyis $\sum_{i=1}^{2}\sum_{j=1}^{n} r_i(j)$, pontosan $2^{n+1} - 2$.

15.1-4. Az $f_i[j]$ és $l_i[j]$ értékeket tartalmazó táblázatoknak együttesen $4n - 2$ eleme van. Mutassa meg, hogy hogyan lehet $2n + 2$ elemre redukálni a tárolási igényt úgy, hogy f^* számítása és a legrövidebb út nyomtatása még lehetséges maradjon.

15.1-5. Aggódó professzor azt sejti, hogy vannak olyan e_i, a_{ij} és t_{ij} értékek, amelyek mellett a LEGRÖVIDEBB-ÁTFUTÁSI-IDŐ eljárás valamely j állomásra az $l_1[j] = 2$ és $l_2[j] = 1$ eredményt adja. Mutassuk meg, hogy ha a t_{ij} átrakási idők nemnegatívak, akkor a professzor aggodalma felesleges.

15.2. Mátrixok véges sorozatainak szorzása

Következő példánk a dinamikus programozás alkalmazására egy algoritmus, mely mátrixok véges sorozatát szorozza össze. Tegyük fel, hogy $A_1, A_2, \ldots, A_n$ adott n mátrix, és ezek

$$A_1 A_2 \cdots A_n \tag{15.10}$$

szorzatát kívánjuk kiszámítani. A (15.10) kifejezés értékét megkaphatjuk a mátrixok szokásos szorzásával, amit szubrutinként használunk, miután a kifejezést zárójelekkel teljesen egyértelművé tettük. Mátrixok egy szorzata ***teljesen zárójelezett,*** ha vagy egyetlen mátrix, vagy két zárójelbe rakott teljesen zárójelezett mátrixszorzat szorzata. A mátrixok szorzása asszociatív, így bármilyen zárójelezés ugyanazt az eredményt adja. Például, ha a sorozat (A_1, A_2, A_3, A_4), akkor az $A_1A_2A_3A_4$ szorzatot a következő öt különböző módon lehet teljesen zárójelezni:

$$(A_1(A_2(A_3A_4))),$$
$$(A_1((A_2A_3)A_4)),$$
$$((A_1A_2)(A_3A_4)),$$
$$((A_1(A_2A_3))A_4),$$
$$(((A_1A_2)A_3)A_4).$$

Az a mód, ahogy egy szorzatot zárójelezünk, alapvetően befolyásolja a kifejezés kiértékelésének költségét. Tekintsük először két mátrix összeszorzásának költségét. A szokásos algoritmust a következő pszeudokóddal adhatjuk meg. A *sor* és *oszlop* attribútum a mátrix sorainak és oszlopainak számát adja meg.

MÁTRIXSZORZÁS(A, B)

```
1  if oszlop[A] ≠ sor[B]
2    then error „nem összeillő dimenzió”
3    else for i ← 1 to sor[A]
4         do for j ← 1 to oszlop[B]
5            do C[i, j] ← 0
6               for k ← 1 to oszlop[A]
7                  do C[i, j] ← C[i, j] + A[i, k]B[k, j]
8         return C
```

Az A és B mátrixot csak akkor szorozhatjuk össze, ha ***összeillők,*** azaz A oszlopainak száma egyenlő B sorainak számával. Ha A egy $p \times q$ méretű mátrix, B pedig $q \times r$ méretű, akkor az eredményül kapott C mátrix mérete $p \times r$. A C kiszámításához szükséges idő döntő része az algoritmus 7. sorában található skalár szorzások mennyisége, ami pqr. A következőkben a számítási időt a szorzások számával fejezzük ki.

Annak illusztrálására, hogy a zárójelezés mennyire befolyásolja a számítás költségét, tekintsük azt a példát, ahol $n = 3$ és az A_1, A_2, A_3 mátrixok mérete rendre 10×100, 100×5 és 5×50. Ha a számítást az $((A_1A_2)A_3)$ zárójelezés szerint végezzük el, akkor $10 \cdot 100 \cdot 5 = 5000$ szorzást végzünk a 10×5 méretű A_1A_2 mátrix kiszámításához, és további $10 \cdot 5 \cdot 50 = 2500$ szorzást, hogy ezt a mátrixot A_3-mal összeszorozzuk, vagyis összesen 7500 szorzásra van

szükségünk. Ha azonban a mátrixok szorzását az $(A_1(A_2A_3))$ zárójelezés szerint végezzük, akkor $100 \cdot 5 \cdot 50 = 25\,000$ szorzásra van szükség a 100×50 méretű A_2A_3 mátrix kiszámításához, és további $10 \cdot 100 \cdot 50 = 50\,000$-re, hogy ezt A_1-gyel összeszorozzuk. Vagyis az első zárójelezés tízszer gyorsabb eljárást ad.

A ***véges sok mátrix összeszorzásának problémája*** a következő: adott mátrixoknak egy $(A_1, A_2, \ldots, A_n)$ véges sorozata, ahol az A_i mátrix mérete $p_{i-1} \times p_i$ $(i = 1, \ldots, n)$. Keresendő az $A_1A_2 \cdots A_n$ szorzat azon teljes zárójelezése, amely minimalizálja a szorzat kiszámításához szükséges skalár szorzások számát.

Hangsúlyozzuk, hogy a véges sok mátrix összeszorzásának problémájában nem szorzunk össze mátrixokat. Célunk csak az összeszorzás legkisebb költséggel bíró sorrendjének meghatározása. Általában az optimális sorrend meghatározására fordított idő több, mint amennyit ezen sorrend alkalmazásával később meg lehet takarítani (ilyen megtakarítás az, amikor csak 7500 szorzást végzünk 75 000 helyett).

A zárójelezések számának meghatározása

Mielőtt megoldanánk dinamikus programozás segítségével a véges sok mátrix összeszorzásának problémáját, meg kell győződnünk, hogy az összes zárójelezés megvizsgálása nem hatékony eljárás. Legyen n mátrix zárójelezéseinek száma $P(n)$. Ha $n = 1$, akkor csak egyetlen mátrixunk van, amit csak egyetlen módon lehet teljesen zárójelezni. Tegyük fel, hogy $n \geq 2$. A szorzat két teljesen zárójelezett szorzat szorzata. Ha egy sorozatot a k-adik és $(k+1)$-edik eleme $(k = 1, 2, \ldots, n-1)$ közt szétvágunk, akkor a két részsorozatra a zárójelezések számát egymástól függetlenül határozhatjuk meg, az alábbi rekurzív formulához jutunk:

$$P(n) = \begin{cases} 1, & \text{ha } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{ha } n \geq 2. \end{cases} \tag{15.11}$$

A 12-4. feladat azt állította, hogy egy hasonló rekurzív egyenlet megoldásai a ***Catalan-számok,*** melyek $\Omega(4^n/n^{3/2})$ szerint nőnek. Az ennél egyszerűbb 15.2-3. gyakorlat annak megmutatását kéri, hogy (15.11) megoldásai $\Omega(2^n)$ nagyságrendűek. Tehát a megoldások száma exponenciális n-ben, azaz az összes zárójelezés megvizsgálásának „nyers erő” módszere nem hatékony eljárás az optimális sorrend meghatározására.

1. lépés: Az optimális zárójelezés szerkezete

Az első szükséges lépés a dinamikus programozás paradigmája szerint az optimális részstruktúra megtalálása, majd alkalmazása arra, hogy a részfeladatok optimális megoldásából létrehozzuk a probléma optimális megoldását. A mátrixszorzási feladatra ez a következőképpen végezhető el. A kényelem kedvéért jelölje $A_{i..j}$ az $A_iA_{i+1} \cdots A_j$ szorzat eredményét. Vegyük észre, hogy ha a feladat nem triviális, azaz $i < j$, akkor az optimális zárójelezés két részre vágja az $A_iA_{i+1} \cdots A_j$ szorzatot valamely A_k és A_{k+1} mátrix között, ahol $i \leq k < j$. Azaz valamely k mellett először kiszámítjuk az $A_{i..k}$ és $A_{k+1..j}$ mátrixokat, majd ezeket szorozzuk össze, hogy megkapjuk az $A_{i..j}$ végeredményt. Tehát az optimális zárójelezés költsége az $A_{i..k}$ és $A_{k+1..j}$ mátrixok kiszámításának és összeszorzásának együttes költsége.

A kulcsfontosságú észrevétel, hogy az $A_iA_{i+1}\cdots A_k$ részsorozat zárójelezésének is optimálisnak kell lennie ebben az *optimális* zárójelezésben. Miért? Ha az $A_iA_{i+1}\cdots A_k$ részsorozatnak volna egy olcsóbb zárójelezése, akkor az $A_iA_{i+1}\cdots A_j$ optimális zárójelezésében kicserélve az első rész zárójelezését erre az olcsóbbra egy olyan zárójelezését kapnánk a teljes $A_iA_{i+1}\cdots A_j$ sorozatnak, ami az optimálisnál jobb, ez pedig ellentmondás. Hasonló igaz az $A_{k+1}A_{k+2}\cdots A_j$ részsorozatra is, azaz az ő zárójelezésének is optimálisnak kell lenni $A_iA_{i+1}\cdots A_j$ optimális zárójelezésében.

Most felhasználjuk az optimális részstruktúrákat ahhoz, hogy megmutassuk, hogy a részfeladatok optimális megoldásaiból megszerkeszthető a teljes feladat egy optimális megoldása. Láttuk, hogy minden nem triviális esetben a szorzatot két részszorzattá kell szétvágni, és hogy bármely optimális megoldás tartalmaz önmagában optimális megoldásokat ezen részfeladatokra. Tehát a mátrixlánc-szorzás egy konkrét esetének optimális megoldását felépíthetjük úgy, hogy a feladatot két részre vágjuk (optimálisan zárójelezve $A_iA_{i+1}\ldots A_k$-t és $A_{k+1}, A_{k+2}\ldots A_j$-t), és az így keletkező részfeladatok optimális megoldásait összetesszük. Biztosítanunk kell, hogy amikor keressük a kettévágás megfelelő helyét, akkor minden lehetséges pozíciót megvizsgálunk, így tehát az optimálisat is.

2. lépés: Egy rekurzív megoldás

Most az optimális értéket rekurzívan fejezzük ki a részproblémák optimális megoldásai segítségével. Véges sok mátrix összeszorzásának problémája esetén a részprobléma az $A_iA_{i+1}\cdots A_j$ alakú szorzat optimális zárójelezésének meghatározása lesz, ahol $1 \le i \le j \le n$. Legyen $m[i, j]$ az $A_{i..j}$ mátrix kiszámításához minimálisan szükséges skalár szorzások száma. Tehát $A_{1..n}$ kiszámításának lehetséges legkisebb költsége $m[1, n]$.

Az $m[i, j]$ mennyiség kiszámítását rekurzív módon az alábbiak szerint végezhetjük el. Ha $i = j$, akkor $A_{i..i} = A_i$, azaz nincs szükség szorzásra a szorzat meghatározásához. Tehát $m[i, i] = 0$, $i = 1, 2, \ldots, n$. Ha $i < j$, akkor felhasználjuk az optimális megoldások szerkezetét, amit a dinamikus programozás paradigmájának első lépésében határoztunk meg. Tegyük fel, hogy az optimális zárójelezés az A_k és A_{k+1} mátrixok között vágja szét az $A_iA_{i+1}\cdots A_j$ szorzatot, ahol $i \le k < j$. Ekkor tehát $m[i, j]$ egyenlő az $A_{i..k}$ és $A_{k+1..j}$ mátrixok kiszámítása minimális költségének és ezen két mátrix összeszorzása költségének összegével. Mivel az A_i mátrixok mérete $p_{i-1} \times p_i$, az $A_{i..k}A_{k+1..j}$ szorzat kiszámítása $p_{i-1}p_kp_j$ szorzást igényel, ezért azt kapjuk, hogy

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j.$$

Ez a rekurzív egyenlet feltételezi, hogy ismerjük k értékét, habár ez nem igaz. Azonban csak $j - i$ különböző értéke lehet k-nak, nevezetesen $k = i, i+1, \ldots, j-1$. Mivel az optimális zárójelezés feltétlenül használja valamelyiket, ezért az a teendőnk, hogy valamennyit megvizsgáljuk, és a legjobbat kiválasszuk. Így az $A_iA_{i+1}\cdots A_j$ szorzat zárójelezése minimális költségére a következő rekurzív definíciót kapjuk:

$$m[i, j] = \begin{cases} 0, & \text{ha } i = j, \\ \min_{i \le k < j}\{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\}, & \text{ha } i < j. \end{cases} \tag{15.12}$$

Az $m[i, j]$ mennyiségek a részproblémák optimális értékét adják meg. Az optimális megoldás megtalálása érdekében definiáljuk az $s[i, j]$ mennyiséget, ami nem más, mint az a k index, ahol az optimális zárójelezés az $A_iA_{i+1}\cdots A_j$ szorzatot ketté vágja, azaz $s[i, j]$ az a k érték, amelyre $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ teljesül.

3. lépés: Az optimális költség kiszámítása

Ezen a ponton már egyszerű dolog egy rekurzív algoritmust írni a (15.12) képlet alapján, amely meghatározza az $A_1 A_2 \cdots A_n$ szorzat $m[1, n]$ minimális költségét. Azonban, mint azt a 15.3. alfejezetben látni fogjuk, ez az algoritmus exponenciális időt igényel, azaz nem jobb, mint az összes zárójelezés megvizsgálásának durva módszere.

Azonban azt a fontos észrevételt tehetjük most, hogy viszonylag kevés részfeladatunk van csak: pontosan egy minden olyan i, j választásra, amely kielégíti az $1 \leq i \leq j \leq n$ egyenlőtlenséget, azaz összesen $n(n+1)/2 = \Theta(n^2)$. Egy rekurzív algoritmus mindegyik részfeladattal többször is foglalkozhat a rekurziós fa különböző ágaiban. A dinamikus programozás alkalmazhatóságának második jellemzője a részfeladatok ilyen átfedése (az első, hogy a részfeladatok optimális megoldásai lépnek fel az optimális megoldásban).

Ahelyett, hogy a (15.12) egyenlet megoldását rekurzív módon számolnánk ki, végrehajtjuk a dinamikus programozás paradigmájának harmadik lépését, és az optimális költséget egy alulról felfelé történő megközelítéssel határozzuk meg. A következő pszeudokód feltételezi, hogy az A_i mátrix mérete $p_{i-1} \times p_i$ minden $i = 1, 2, \ldots, n$ esetén. A bemenet a $p = \langle p_0, p_1, \ldots, p_n \rangle$ sorozat, ahol $hossz[p] = n+1$. Az eljárás használja az $m[1\,..\,n, 1\,..\,n]$ és $s[1\,..\,n, 1\,..\,n]$ tömböket a költségek, illetve az optimális költséget adó k indexek tárolására. Az utóbbi tömböt az optimális megoldás megszerkesztésére használjuk fel.

Az alulról felfelé való megközelítés helyes megvalósítása megköveteli, hogy meghatározzuk, az $m[i, j]$ tömb mely elemeit használjuk. A (15.12) egyenletből látható, hogy a $j - i + 1$ mátrix összeszorzása $m[i, j]$ költségének kiszámításához csak olyan szorzatok költségére van szükség, melyek $j - i + 1$-nél kevesebb mátrixot tartalmaznak. Azaz $k = i, i+1, \ldots, j-1$ esetén az $A_{i..k}$ mátrix $k - i + 1 < j - i + 1$ mátrixnak, az $A_{k..j}$ mátrix pedig $j - k < j - i + 1$ mátrixnak a szorzata. Így az algoritmusnak az m tömböt a zárójelezési probléma megoldása során a szorzatok növekvő hossza szerint kell kitöltenie.

MÁTRIX-SZORZÁS-SORREND(p)

```
 1  n ← hossz[p] − 1
 2  for i ← 1 to n
 3      do m[i, i] ← 0
 4  for l ← 2 to n        ▷ l a szorzat hossza.
 5      do for i ← 1 to n − l + 1
 6             do j ← i + l − 1
 7                m[i, j] ← ∞
 8                for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                          then m[i, j] ← q
12                               s[i, j] ← k
13  return m és s
```

Az algoritmus először az $m[i, i] \leftarrow 0$ ($i = 1, 2, \ldots, n$) műveleteket hajtja végre a 2. és 3. sorban, mivel 0 a költsége az 1 hosszú szorzatoknak. Azután a (15.12) rekurzív egyenletet felhasználva az $m[i, i+1]$ ($i = 1, 2, \ldots, n-1$) értékeket, azaz az $l = 2$ hosszú szorzatok minimális költségeit számítja ki, a 4–12. sorokban található ciklus első lefutásakor. A ciklus

második lefutásakor következnek az $m[i, i+2]$ ($i = 1, 2, \ldots, n-2$) mennyiségek, azaz az $l = 3$ hosszú szorzatok minimális költségei és így tovább. Minden lépésben, amikor az $m[i, j]$ mennyiséget határozzuk meg a 9–12. sorban, az csak a tömb $m[i, k]$ és $m[k+1, j]$ elemeitől függ, amelyeket már kiszámítottunk.

A 15.3. ábra ezt az eljárást $n = 6$ mátrix szorzatán mutatja be. Mivel az $m[i, j]$ mennyiséget csak $i \leq j$ esetére definiáltuk, ezért az m tömb elemei közül csak a főátló fölé esőket használjuk. Az ábra elforgatva mutatja a tömböt úgy, hogy a főátló vízszintes helyzetben van. A mátrixok sorozata az ábra alján látható. Ebben az elrendezésben az $A_i A_{i+1} \cdots A_j$ szorzat $m[i, j]$ költsége azon két egyenes metszéspontjában található, amelyek közül az egyik A_i-től északkeleti, a másik pedig A_j-től északnyugati irányban fut. Minden vízszintes sor azonos hosszúságú szorzatok költségét tartalmazza. A MÁTRIX-SZORZÁS-SORREND eljárás a tömböt alulról felfelé és a sorokon belül balról jobbra számítja ki. Az $m[i, j]$ érték kiszámításához a $p_{i-1} p_k p_j$ ($k = i, i+1, \ldots, j-1$) alakú szorzatokat és a tőle délnyugatra és délkeletre lévő értékeket használjuk fel.

A MÁTRIX-SZORZÁS-SORREND eljárás három mélységben egymásba skatulyázott ciklusainak egyszerű vizsgálata $O(n^3)$ futási időt ad, mivel a három ciklusváltozó mindegyike (l, i és k) legfeljebb n értéket vehet fel. A 15.2-4. gyakorlat azt kéri, hogy mutassuk meg, hogy a módszer futási ideje valóban $\Omega(n^3)$. Az algoritmus $\Theta(n^2)$ memóriát igényel az m és s tömb tárolásához. Tehát a MÁTRIX-SZORZÁS-SORREND eljárás sokkal hatékonyabb, mint az összes lehetséges zárójelezés megvizsgálásának exponenciális módszere.

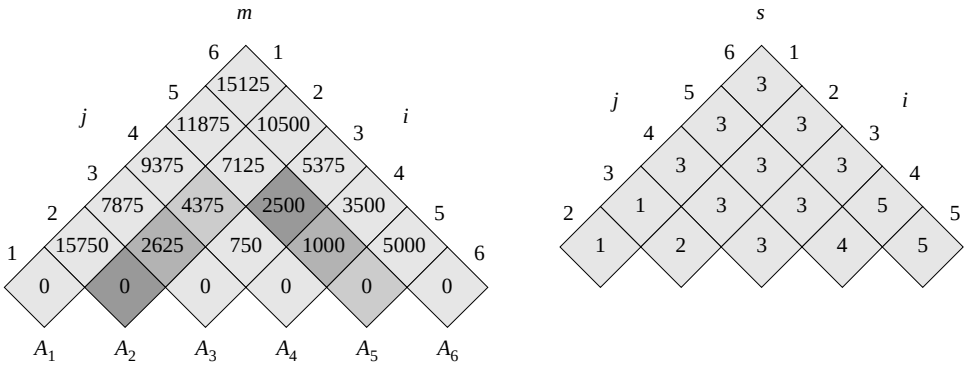
4. lépés: Az optimális megoldás előállítása

Habár a MÁTRIX-SZORZÁS-SORREND eljárás megadja véges sok mátrix összeszorzásakor a skalár szorzások optimális számát, közvetlenül nem mutatja meg a mátrixok összeszorzásának mikéntjét. Könnyű az $s[1..n, 1..n]$ tömb felhasználásával a mátrixok összeszorzásának legjobb módját meghatározni. Minden $s[i, j]$ elem azt a k indexet tartalmazza, ahol az optimális zárójelezés az $A_i A_{i+1} \cdots A_j$ szorzatot kettévágja A_k és A_{k+1} között. Tehát a teljes $A_{1..n}$ szorzat optimális kiszámításakor a végső mátrixszorzás $A_{1..s[1,n]} A_{s[1,n]+1..n}$. A korábbi mátrixszorzásokat rekurzívan számíthatjuk ki: $s[1, s[1, n]]$ adja meg az utolsó szorzás helyét $A_{1..s[1,n]}$ számításakor, $s[s[1, n] + 1, n]$ pedig az $A_{s[1,n]+1..n}$ mátrix esetében. Az alábbi rekurzív eljárás az adott $A = (A_1, A_2, \ldots, A_n)$ mátrixok és rögzített i és j indexek mellett a MÁTRIX-SZORZÁS-SORREND által kiszámított s tömb segítségével az optimális $A_{i..j}$ szorzatot nyomtatja ki. A kezdeti hívás OPTIMÁLIS-ZÁRÓJELEZÉS-NYOMTATÁSA$(s, 1, n)$.

OPTIMÁLIS-ZÁRÓJELEZÉS-NYOMTATÁSA(s, i, j)

```
1  if j = i
2     then print „A”_i
3     else print „(”
4          OPTIMÁLIS-ZÁRÓJELEZÉS-NYOMTATÁSA(s, i, s[i, j])
5          OPTIMÁLIS-ZÁRÓJELEZÉS-NYOMTATÁSA(s, s[i, j] + 1, j)
6          print „)”
```

Az OPTIMÁLIS-ZÁRÓJELEZÉS-NYOMTATÁSA$(s, 1, 6)$ hívás az $A_1, A_2, \ldots, A_6$ mátrixok szorzatát az $((A_1(A_2A_3))((A_4A_5)A_6))$ zárójelezés szerint számítja a 15.3. ábra példájában.



15.3. ábra. A MÁTRIX-SZORZÁS-SORREND eljárás által kiszámított m és s tömbök, ha $n = 6$ és a mátrixok méretei a következők:

mátrix	méret
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

A tömböket úgy forgattuk, hogy a főátló vízszintes legyen. Az m tömbben csak a főátlót és a felső háromszöget, az s tömbben pedig csak a felső háromszöget használjuk. A hat mátrix összeszorzásához minimálisan $m[1, 6] = 15125$ skalár szorzás kell. A világosabban satírozott elemek azon párjaiból, ahol a satírozás azonos, az eljárás 9. sorában a következőt kapjuk:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} = 7125.$$

Gyakorlatok

15.2-1. Keressük meg azon mátrixok összeszorzásának optimális zárójelezését, ahol a méretek sorozata (5, 10, 3, 12, 5, 50, 6).

15.2-2. Adjunk meg rekurzív MÁTRIX-LÁNC-SZORZÁS(A, s, i, j) algoritmust, amely optimális módon végre is hajtja mátrixok véges sorozatainak összeszorzását, feltéve, hogy adott $(A_1, A_2, \ldots, A_n)$, a MÁTRIX-SZORZÁS-SORREND által számított s tömb és az i és j index. (A kezdeti hívás MÁTRIX-LÁNC-SZORZÁS($A, s, 1, n$).)

15.2-3. A helyettesítés módszerével mutassuk meg, hogy a (15.11) rekurzív egyenlet megoldásának nagyságrendje $\Omega(2^n)$.

15.2-4. Legyen $R(i, j)$ az a szám, ahányszor a MÁTRIX-SZORZÁS-SORREND eljárás az $m[i, j]$ elemet felhasználja. Mutassuk meg, hogy az egész tömbre vonatkozóan

$$\sum_{i=1}^{n} \sum_{j=1}^{n} R(i, j) = \frac{n^3 - n}{3}.$$

(*Útmutatás.* Hasznos lehet az (A.3) azonosság.)

15.2-5. Mutassuk meg, hogy egy n-elemű kifejezés bármely teljes zárójelezése pontosan $n - 1$ zárójelpárt tartalmaz.

15.3. A dinamikus programozás elemei

Habár két példán keresztül bemutattuk a dinamikus programozást, mégis felmerülhet az Olvasóban a kérdés, hogy a módszer egyáltalán hol alkalmazható. Mérnöki szempontból mikor kell egy probléma dinamikus programozási megoldását keresnünk? Ebben az alfejezetben azt a két tulajdonságot vizsgáljuk, amivel egy optimalizálási problémának rendelkeznie kell ahhoz, hogy a dinamikus programozás alkalmazható legyen. Ezek az optimális részstruktúrák és az átfedő részproblémák. Ugyancsak áttekintjük a módszer egy variánsát, amit feljegyzéses módszernek[1] nevezünk, és ami az átfedő részfeladatokból származó előnyöket használja ki.

Optimális részstruktúra

A dinamikus programozás alkalmazásához vezető első lépés az optimális megoldás szerkezetének jellemzése. Azt mondjuk, hogy a feladat ***optimális részstruktúrájú,*** ha a probléma egy optimális megoldása önmagán belül a részfeladatok optimális megoldásait tartalmazza. Ha a feladat ilyen tulajdonságú, akkor ez jó jel arra, hogy a dinamikus programozás alkalmazható lehet. (De azt is jelentheti, hogy a mohó stratégia alkalmazható, lásd 16. fejezet.) Mivel a dinamikus programozásban az optimális megoldást részfeladatok optimális megoldásaiból építjük fel, biztosítanunk kell, hogy a részfeladatok általunk vizsgált halmaza tartalmazza azokat, amelyekből az optimális megoldás áll.

A jelen fejezetben tárgyalt mindkét probléma esetében kimutattuk az optimális részstruktúra tulajdonságot. A 15.1. alfejezetben azt láttuk, hogy bármelyik szerelőszalag j-edik állomásán áthaladó legrövidebb út tartalmazza valamelyik szerelőszalag $(j-1)$-edik állomásán áthaladó legrövidebb utat. A 15.2. alfejezetben megfigyeltük, hogy $A_iA_{i+1}\cdots A_j$ optimális zárójelezése, mely a szorzatot az A_k és A_{k+1} között vágja ketté, az $A_iA_{i+1}\cdots A_k$ és $A_{k+1}A_{k+2}\cdots A_j$ optimális zárójelezését tartalmazza.

Az alábbi séma adható meg annak kimutatására, hogy egy probléma rendelkezik az optimális részstruktúra tulajdonságával:

1. Meg kell mutatni, hogy a probléma megoldása során döntéseket kell hozni, mint például annak kiválasztása, hogy az előző állomás melyik szerelőszalagon legyen, vagy annak az indexnek a kiválasztása, ahol a mátrixok láncát ketté kell vágni. Ezen döntés után még egy vagy több részproblémát meg kell oldani.
2. Azt kell feltételezni a probléma esetében, hogy az optimális megoldáshoz vezető döntés adott. Nem kell foglalkozni azzal, hogy hogyan kell ezt a döntést meghatározni, egyszerűen fel kell tenni, hogy ismerjük.
3. Feltéve, hogy adott ez a döntés, meg kell határozni, hogy milyen részfeladatok keletkeznek, és hogyan lehet a fellépő részfeladatok terét a legjobban jellemezni.
4. A „szétvágás és összeragasztás” módszerével meg kell mutatni, hogy a részfeladatok azon megoldásai, amelyeket az optimális megoldás tartalmaz, maguk is optimálisak. Ezt úgy érjük el, hogy feltesszük, hogy a részfeladatok megoldásai nem optimálisak, és ebből ellentmondásra jutunk. Nevezetesen „kivágjuk” a részfeladat nem optimális megoldását és „beragasztunk” helyette egy optimálisat. Meg kell mutatni, hogy így egy jobb

[1] Angolul *memoization* és nem *memorization*. A memoization kifejezés a *memo* szóból származik, ami arra utal, hogy feljegyezzük azokat az értékeket, amelyeket később csak kiolvasunk.

megoldáshoz jutunk, ami ellentmond annak, hogy optimálisból indultunk ki. Ha történetesen több részfeladat is volna, akkor ezek rendszerint annyira hasonlóak egymáshoz, hogy a szétvágás és összeragasztás módszerén alapuló érvelés kis erőfeszítéssel módosítható úgy, hogy mindegyikre alkalmazható legyen.

A részfeladatok terének jellemzésekor egy jó ökölszabály, hogy ezt a teret olyan egyszerűnek kell megtartani, amennyire csak lehet, és csak akkor szabad kiterjeszteni, ha szükséges. Például a szerelőszalag ütemezésekor ez a tér az üzembe való belépéstől az $S_1[j]$, illetve $S_2[j]$ állomáson való leggyorsabb áthaladásig tartó útból állt. Ez a tér jól működött, és nem volt semmi szükség részfeladatok egy általánosabb terének bevezetésére.

Megfordítva, tegyük fel, hogy a mátrixok láncainak összeszorzásakor a részfeladatok terét az $A_1A_2 \cdots A_j$ alakú szorzatokra szűkítjük le. Az optimális megoldás ezt valamely alkalmas $1 \leq k < j$ mellett A_k és A_{k+1} között vágja ketté. Hacsak nem tudjuk biztosítani valahogy, hogy k mindig $j-1$, a kapott részfeladatok $A_1A_2 \cdots A_k$ és $A_{k+1}A_{k+2} \cdots A_j$ alakúak. Az utóbbi viszont nem $A_1A_2 \cdots A_j$ alakú. Ezért ezen feladat esetében meg kellett engedni, hogy a részfeladat mindkét végén változzék, azaz az $A_iA_{i+1} \cdots A_j$ részfeladatban az i és a j index is változzék.

Különböző feladatok esetében az optimális részstruktúra kétféleképpen változik:

1. hány részprobléma megoldását használjuk az eredeti feladat optimális megoldásában,
2. hány különböző lehetőséget kell megvizsgálni, míg a felhasználandó részfeladatokat kiválasztjuk.

A szerelőszalag ütemezése esetében csak egyetlenegy részfeladatot használunk, amit két lehetőség közül kell kiválasztani. Az S_{ij} állomáson való leggyorsabb áthaladást *vagy* az S_{1j-1}-en való leggyorsabb áthaladást, *vagy* az S_{2j-1}-en való leggyorsabb áthaladást tartalmazza. Bármelyiket vegyük is, az adja azt az egyetlen részfeladatot, amit optimálisan meg kell oldanunk. A mátrixok szorzásánál $A_iA_{i+1} \cdots A_j$ arra ad példát, hogy az optimális megoldásban két részfeladat megoldása van, amelyeket $j - i$ lehetőség közül kell kiválasztani. Adott A_k mátrix esetében két részfeladatunk van – $A_iA_{i+1} \cdots A_k$ és $A_{k+1}A_{k+2} \cdots A_j$ zárójelezése –, és nekünk *mindkettőt* meg kell oldani optimálisan. Mihelyst megoldottuk optimálisan a részfeladatokat, a $j - i$ lehetőség közül ki kell választani k értékét.

Informálisan a dinamikus programozás futási ideje két tényező szorzatától függ: az összes részfeladat számától és attól, hogy részfeladatonként hány választási lehetőségünk van. A szerelőszalag ütemezésének esetében $\Theta(n)$ részprobléma van összesen és két lehetőséget kell megvizsgálni mindegyiknél, ami összesen $\Theta(n)$ futási időt ad. A mátrixok szorzásánál $\Theta(n^2)$ részprobléma és mindegyiknél legfeljebb $n - 1$ részfeladat van, ami együtt $O(n^3)$ futási időt eredményez.

A dinamikus programozás az optimális részstruktúra tulajdonságot alulról felfelé használja, ami azt jelenti, hogy előbb kell a részfeladatok optimális megoldását meghatározni, és csak utána tudjuk az egész feladat optimális megoldását megtalálni. A feladat optimális megoldásának megtalálása maga után vonja, hogy választanunk kell a részfeladatok között, hogy melyik megoldása szerepeljen az egész feladat optimális megoldásában. A megoldás költsége általában a részfeladat költsége meg egy közvetlenül a választásnak tulajdonítható költség. Például a szerelőszalagnál először megoldottuk az $S_{1,j-1}$ és $S_{2,j-1}$ állomáson való leggyorsabb áthaladás problémáját, és azután választottuk ki, hogy melyik előzze meg az S_{ij} állomást. A választáshoz rendelhető költség attól függ, hogy váltunk-e szerelőszalagot a $j-1$-edik és j-edik állomás között. Ez a költség a_{ij}, ha ugyanazon a szalagon maradunk, és

$t_{i',j-1} + a_{ij}$, $(i' \neq i,)$ ha váltanunk kell a szalagot. A mátrixok szorzásánál, az $A_iA_{i+1}\cdots A_j$ optimális megoldásakor, amikor az A_k mátrixot kiválasztjuk, hogy ott vágjuk ketté a szorzatot, a választás költsége $p_{i-1}p_kp_j$.

A 16. fejezetben fogjuk vizsgálni a mohó algoritmust, aminek sok, a dinamikus programozáséhoz hasonló vonása van. Például azok a feladatok, amelyekre a mohó módszer alkalmazható, rendelkeznek az optimális részstruktúra tulajdonságával. Az egyik lényeges különbség a dinamikus programozás és a mohó módszer között, hogy az utóbbiban az optimális részstruktúra tulajdonságot felülről lefelé használjuk. Ahelyett, hogy először megkeresné a részproblémák optimális megoldását és utána választana közöttük, a mohó módszer először eldönti, hogy mi a legjobb választás az adott pillanatban, és csak ezután oldja meg a keletkező részfeladatot.

Finom részletek

Vigyázni kell arra, hogy olyankor ne tegyük fel az optimális részstruktúra tulajdonság meglétét, amikor az nem áll fenn. A következő két példában adott egy $G = (V, E)$ irányított gráf és annak két csúcsa, u és v.

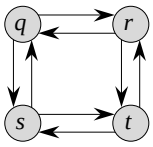
Súlyozatlan legrövidebb út:[2] Keressük meg a legkevesebb élből álló, u-ból v-be vezető utat. Egy ilyen útnak egyszerűnek kell lennie, hiszen egy kör elhagyása az útból csökkenti az élek számát.

Súlyozatlan leghosszabb egyszerű út: Keressük meg az u-ból v-be menő, legtöbb élből álló egyszerű utat. Ki kell kötni azt, hogy az út egyszerű legyen, különben egy körön akárhányszor végig mehetünk, és így tetszőlegesen sok élből álló utat előállíthatunk.

A súlyozatlan legrövidebb út probléma rendelkezik az optimális részstruktúra tulajdonságával. Tegyük fel, hogy $u \neq v$, azaz a feladat nem triviális. Ekkor minden u-ból v-be menő p útnak van egy w közbülső pontja, ami akár u vagy v is lehet. Ekkor az $u \overset{p}{\rightsquigarrow} v$ út a $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ részutakra bontható. Világos, hogy a p-beli élek száma egyenlő a p_1-beli és p_2-beli élek számának összegével. Azt állítjuk, hogy ha p optimális, vagyis a legrövidebb út u-ból v-be, akkor p_1-nek az u-ból w-be vezető legrövidebb útnak kell lennie. Miért? Ismét a „szétvágás és összeragasztás” érvét használjuk: Ha volna egy másik p_1' út u-ból w-be, amely kevesebb élből állna, mint p_1, akkor kivághatnánk p_1-et és beragaszthatnánk helyette p_1'-t, és az így nyert $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ út kevesebb élből állna, mint p, pedig u-t v-vel kötné össze, ez pedig ellentmondana p optimalitásának. Hasonlóképpen p_2-nek a legrövidebb w-ből v-be vezető útnak kell lennie. Így az u-ból v-be menő legrövidebb út megkapható úgy, hogy veszünk egy közbülső w csúcsot, és meghatározzuk az u-ból w-be w-ből v-be vezető legrövidebb utat. Ezután azt a közbülső csúcsot kell választanunk, amelyik így – összességében – a legrövidebb utat eredményezi. A 25.2. alfejezetben ezen optimális részstruktúra egy variánsát használjuk fel, hogy súlyozott, irányított gráfban minden csúcspár között a legrövidebb utat meghatározzuk.

Csábító feltenni, hogy a leghosszabb egyszerű út problémája is rendelkezik az optimális részstruktúra tulajdonságával. Végül is, ha az $u \overset{p}{\rightsquigarrow} v$ leghosszabb utat a $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ részutakra bontjuk fel, akkor nem kellene p_1-nek az u-ból w-be és p_2-nek a w-ből v-be vezető hosszabb útnak lennie? A válasz nem! A 15.4. ábra példa arra, hogy miért van ez így.

[2] A súlyozatlan szót arra használjuk, hogy megkülönböztessük ezt a feladatot attól, amikor a legrövidebb utat súlyozott élek mellett keressük. Ezt az utóbbi problémát a 24. és 25. fejezetben fogjuk tárgyalni. A súlyozatlan feladat megoldására a 22. fejezetben a szélességi keresést fogjuk használni.



15.4. ábra. Egy irányított gráf, ami mutatja, hogy a leghosszabb egyszerű út nem rendelkezik az optimális részstruktúra tulajdonságával. A $q \to r \to t$ a leghosszabb egyszerű út q-ból t-be, de sem $q \to r$, sem $r \to t$ nem a leghosszabb egyszerű út q-ból r-be, illetve r-ből t-be.

Tekintsük a $q \to r \to t$ utat, ami egy leghosszabb egyszerű út q-ból t-be. Igaz, hogy $q \to r$ a leghosszabb egyszerű út q-ból r-be? Nem, a $q \to s \to t \to r$ út hosszabb. Hasonlóképpen $r \to t$ nem a leghosszabb egyszerű út r-ből t-be, az $r \to q \to s \to t$ út hosszabb.

Ez az egyszerű probléma nemcsak azt mutatja, hogy a leghosszabb egyszerű út problémája nem rendelkezik az optimális részstruktúra tulajdonságával, hanem azt is, hogy a részfeladatok optimális megoldásaiból nem is tudunk megszerkeszteni egy „legális” megoldást. Ha összerakjuk a két részfeladat $q \to s \to t \to r$ és $r \to q \to s \to t$ megoldását, akkor a $q \to s \to t \to r \to q \to s \to t$ utat kapjuk, amelyik nem egyszerű. Tehát a leghosszabb egyszerű út meghatározása valóban nem rendelkezik az optimális részstruktúra tulajdonságával. Soha nem találtak hozzá hatékony dinamikus programozási algoritmust. Valójában a feladat NP-teljes, ami – mint azt a 34. fejezetben látni fogjuk – azt jelenti, hogy nem valószínű, hogy polinomiális időben megoldható.

Milyen a részfeladatok struktúrája a leghosszabb egyszerű útnál, hogy annyira különbözik a legrövidebb út problémája részfeladatainak struktúrájától? Bár mindkét esetben két részfeladat megoldását használjuk fel, a leghosszabb egyszerű út meghatározásánál a részfeladatok nem ***függetlenek*** egymástól, míg a legrövidebb út esetében azok. Mit értünk a részfeladatok függetlenségén? Ez azt jelenti, hogy az egyik feladat megoldása nincs hatással a másik feladat megoldására. A példaként szereplő 15.4. ábrán a feladat a q-ból t-be menő leghosszabb egyszerű út megtalálása, melynek két részfeladata a q-ból r-be és az r-ből t-be menő leghosszabb egyszerű út megkeresése. Az első optimális megoldásaként $q \to s \to t \to r$ adódott, amiben használtuk az s és t csúcsot is. Emiatt a második részfeladatban nem használhatjuk ezt a két csúcsot, különben a két részfeladat megoldásából a teljes feladatra kapott megoldás nem lesz egyszerű út. Ha azonban a t csúcsot nem használhatjuk, akkor a második részproblémát egyáltalán nem is tudjuk megoldani, hiszen t az ott keresett út kötelező végpontja, nem pedig egy olyan csúcs, aminél a két megoldást összeillesztjük (ami az r csúcs volt). Az, hogy az s és t csúcsot használjuk az egyik részfeladatban, kizárja, hogy a másikban használjuk őket. Ahhoz azonban, hogy a másikat megoldjuk, legalább az egyiket használni kell, ahhoz pedig, hogy optimálisan oldjuk meg, mindkettőt. Tehát ezek a részproblémák nem függetlenek egymástól. Másképpen fogalmazva, az, hogy bizonyos erőforrásokat (itt a csúcsokat) az egyik részfeladatban használjuk, azt eredményezi, hogy ugyanezek az erőforrások nem hozzáférhetők más részfeladatok esetében.

Miért függetlenek akkor a részproblémák a legrövidebb út feladat esetében? A válasz az, hogy a probléma természeténél fogva a részfeladatok nem osztoznak azonos erőforráson. Az állítjuk, hogy ha w az u-ból v-be vezető legrövidebb út egy közbülső pontja, akkor bármely u-ból w-be menő $u \overset{p_1}{\rightsquigarrow} w$ és w-ből v-be vezető $w \overset{p_2}{\rightsquigarrow} v$ legrövidebb út összeillesztésével egy u-ból v-be vezető legrövidebb utat kapunk. Könnyen megbizonyosodhatunk róla, hogy w-n kívül p_1 más csúcsa nem ismétlődik meg p_2-ben. Miért? Tegyük fel, hogy egy

$x \neq w$ csúcs fellép mind p_1-ben, mind p_2-ben. Így tehát p_1-et $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ utakra, p_2-t pedig $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$ utakra dekomponálhatjuk. A p út éleinek száma az optimális részstruktúra tulajdonság miatt annyi, mint p_1 és p_2 élei számának összege, mondjuk e. Azonban az $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ út is u-ból v-be vezet, és legfeljebb $e - 2$ éle van, ami ellentmond p optimalitásának. Tehát a legrövidebb út problémában a részfeladatok függetlenek egymástól.

Mind a 15.1., mind a 15.2. alfejezetben szereplő probléma esetében a részfeladatok függetlenek egymástól. A mátrixok szorzásánál a két részfeladat $A_i A_{i+1} \cdots A_k$ és $A_k A_{k+1} \cdots A_j$. Ez a két részsorozat diszjunkt, így ugyanaz a mátrix nem léphet fel mindkettőben. A szerelőszalag ütemezésénél az S_{ij} állomáson való leggyorsabb áthaladáshoz az $S_{1,j-1}$ és $S_{2,j-1}$ állomásokon való leggyorsabb áthaladást vizsgáltuk. Mivel az $S_{i,j}$ állomáson való leggyorsabb áthaladás meghatározásakor csak az egyik részfeladat megoldását illesztjük bele a megoldásba, ez automatikusan független a megoldásban felhasznált többitől.

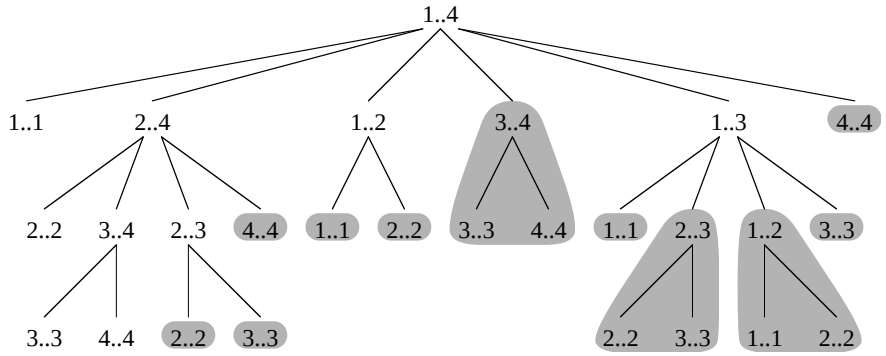
Átfedő részfeladatok

A másik tulajdonság, amivel egy problémának rendelkeznie kell ahhoz, hogy a dinamikus programozás alkalmazható legyen az, hogy a részfeladatok halmaza „kicsi” legyen abban az értelemben, hogy a probléma rekurzív algoritmusa ismételten ugyanazokat a részfeladatokat oldja meg ahelyett, hogy mindig új részfeladatot állítana elő. Tipikus az, ha az összes különböző részfeladat száma az input méretében polinomiális. Ha a rekurzív algoritmus ismételten visszatér ugyanarra a részfeladatra, akkor azt mondjuk, hogy az optimalizálási feladat ***részfeladatai átfedőek.***[3] Ezzel szemben az olyan feladat esetében, amire az oszd-meg-és-uralkodj megközelítés alkalmazható, a rekurzió minden lépésben teljesen új részfeladatokat szokott generálni. A dinamikus programozás kihasználja az átfedő részfeladatok nyújtotta azon előnyt, hogy elegendő mindegyiket csak egyszer megoldani, az eredményt tárolni, és amikor a részfeladatra ismét szükség van, akkor a tárolt eredményt időben állandó költséggel elővenni.

A 15.1. alfejezetben röviden megvizsgáltuk, hogy hogyan lehetséges, hogy egy rekurzív eljárás az $f_i[j]$ értékre 2^{n-j}-szer hivatkozik ($j = 1, 2, \ldots, n$). A táblázatos megoldásunk a rekurzív eljárás exponenciális idejét lineárisra szorítja le.

Az átfedő részfeladatok részletesebb illusztrálására vizsgáljuk meg ismét a véges sok mátrix szorzásának problémáját. Visszatekintve a 15.3. ábrára vegyük észre, hogy a MÁTRIX-SZORZÁS-SORREND eljárás az alsóbb sorokban található részfeladatok megoldását ismételten felhasználja a magasabb sorokban lévő részfeladatok megoldása során. Például az $m[3, 4]$ elemre négyszer van szükség, nevezetesen $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ és $m[3, 6]$ kiszámításakor. Ha mindegyik esetben kiszámítanánk az $m[3, 4]$ mennyiséget ahelyett, hogy csak egyszerűen a tárolt értéket kiolvasnánk, akkor a futási idő drámai módon megnőne. Hogy lássuk ezt, tekintsük azt a (rossz hatásfokú) rekurzív eljárást, amelyik minden $A_{i..j} = A_i A_{i+1} \cdots A_j$ szorzat elvégzéséhez a minimálisan szükséges szorzások számát mindig meghatározza. Az eljárás közvetlenül a (15.12) rekurzív egyenleten alapszik.

[3] Furcsának tűnhet, hogy a részfeladatokat egyszerre nevezzük átfedőnek és függetlennek. Habár ez a két elnevezés ellentmondónak hangzik, két különböző fogalmat takar. Két részfeladat független, ha nem használnak közös erőforrást. Két részfeladat átfedő, ha azonosak annak ellenére, hogy különböző feladatok részfeladatai.



15.5. ábra. A REKURZÍV-SZORZÁS-SORREND$(p, 1, 4)$ rekurziós fája. Minden csúcs tartalmazza az i és j paramétert. A satírozott részfák esetében a FELJEGYZÉSES-MÁTRIX-SORREND$(p, 1, 4)$ eljárás csak kiolvassa a tárolt eredményt.

REKURZÍV-MÁTRIX-LÁNC(p, i, j)

```
1  if i = j
2    then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5      do q ← REKURZÍV-MÁTRIX-LÁNC(p, i, k)
             + REKURZÍV-MÁTRIX-LÁNC(p, k + 1, j) + p_{i−1} p_k p_j
6        if q < m[i, j]
7          then m[i, j] ← q
8  return m[i, j]
```

A 15.5. ábra mutatja a REKURZÍV-MÁTRIX-LÁNC$(p, 1, 4)$ hívás nyomán keletkezett rekurziós fát. Minden csúcs címkéje az i és j paraméter értéke. Figyeljük meg, hogy egyes értékpárok többször is előfordulnak.

Valóban meg tudjuk mutatni, hogy ennek a rekurzív eljárásnak $m[1, n]$ meghatározásához szükséges $T(n)$ futási ideje legalább exponenciális n-ben. Tegyük fel, hogy mind az 1–2., mind a 6–7. sor műveleteinek végrehajtása legalább egy id őegység. Az eljárás vizsgálata azt mutatja, hogy

$$\begin{aligned} T(1) &\geq 1, \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1), \qquad \text{ha } n > 1. \end{aligned}$$

Vegyük észre, hogy minden $i \overset{k=1}{=} 1, 2, \ldots, n-1$ esetén a $T(i)$ tag egyszer mint $T(k)$ és még egyszer mint $T(n-k)$ lép fel. Ha még összegezzük az 1-eseket is, azt kapjuk, hogy

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n. \tag{15.13}$$

A helyettesítés módszerével megmutatjuk, hogy $T(n) = \Omega(2^n)$. Pontosabban, azt látjuk be, hogy minden $n \geq 1$ esetén $T(n) \geq 2^{n-1}$. Az állítás $n = 1$-re igaz, hiszen $T(1) \geq 1 = 2^0$. Tegyük fel, hogy az állítás igaz minden $T(i)$ esetén, ha $i < n$. Ekkor $n \geq 2$ mellett kapjuk, hogy

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^{i} + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^{n} - 2) + n \\ &\geq 2^{n-1}. \end{aligned}$$

Tehát a REKURZÍV-MÁTRIX-LÁNC$(p, 1, n)$ eljárás által végzett számítás mennyisége n-ben legalább exponenciális.

Hasonlítsuk össze a felülről lefelé haladó rekurzív algoritmust az alulról felfelé menő dinamikus programozási eljárással. Utóbbi sokkal hatékonyabb, mert kihasználja az átfedő részfeladatok nyújtotta előnyöket. $\Theta(n^2)$ különböző részfeladat van, és a dinamikus programozás mindegyiket pontosan egyszer oldja meg. Ezzel ellentétben a rekurzív algoritmus minden részfeladatot minden alkalommal megold, amikor előfordul a rekurziós fában. Ha egy probléma természetes rekurzív megoldásának rekurziós fája ismételten tartalmazza ugyanazt a részfeladatot, és a részfeladatok száma kicsi, célszerű megvizsgálni, hogy a feladat dinamikus programozással megoldható-e.

Egy optimális megoldás előállítása

Praktikus okoknál fogva gyakran egy táblázatban tároljuk, hogy az egyes részfeladatoknál milyen döntést hoztunk. Így nem kell ezt az információt a tárolt költségek alapján rekonstruálni. A szerelőszalag ütemezésénél az S_{ij}-n átvezető leggyorsabb út S_{ij}-t megelőző állomását $l_i[j]$-ben tároltuk. Azonban ugyanezt az információt némi számítási többletmunka árán is megkaphatjuk, ha az egész $f_i[j]$ táblázatot tároltuk. Ugyanis, ha $f_1[j] = f_1[j-1] + a_{1j}$, akkor ez az állomás $S_{1,j-1}$, ha pedig $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1j}$, akkor $S_{2,j-1}$. Tehát ebben az esetben a megelőző állomás rekonstruálása csak $O(1)$ időt igényel még abban az esetben is, ha $l_i[j]$-t nem tároltuk.

A mátrixok szorzásánál azonban az $s[i, j]$ táblázat tárolása jelentős munkát takarít meg számunkra az optimális megoldás rekonstruálásakor. Tegyük fel, hogy $s[i, j]$-t nem, csak a részfeladatok optimális értékét tartalmazó $m[i, j]$ táblázatot tároltuk. Most $j - i$ lehetőség van arra, hogy melyik részfeladatot kell használni $A_iA_{i+1} \cdots A_j$ felbontásánál, és $j - i$ nem állandó. Ezért $\Theta(j - i) = \omega(1)$ ideig tart rekonstruálni, hogy melyik részfeladatról van szó. Az $s[i, j]$ táblázat tárolásával minden szorzat szétvágási helye $O(1)$ idő alatt megkapható.

A feljegyzéses módszer

Létezik a dinamikus programozásnak egy olyan változata, amely gyakran ugyanolyan hatékony, mint a szokásos, pedig felülről lefelé halad. Az ötlet az, hogy ***feljegyzéssel*** kell kiegészíteni a természetes, ámde lassú rekurzív algoritmust. A közönséges dinamikus programozáshoz hasonlóan a részfeladatok megoldását itt is egy tömbben tároljuk, de a tömb kitöltésének stratégiája jobban hasonlít a rekurzív algoritmusra.

A feljegyzéses rekurzív algoritmus minden részprobléma megoldása számára fenntart egy helyet a tömbben. A tömb minden eleme kezdetben egy speciális értéket vesz fel, ami

azt jelzi, hogy az elembe még nem jegyeztük fel a kívánt értéket. Amikor a rekurzív eljárás során először találkozunk a részfeladattal, a megoldását kiszámítjuk és feljegyezzük a tömbbe. Minden további alkalommal, amikor a részfeladatra szükség van, a tömbben tárolt értéket használjuk fel.[4]

A következő eljárás a REKURZÍV-MÁTRIX-LÁNC feljegyzéses változata.

```
FELJEGYZÉSES-MÁTRIX-LÁNC(p)
1  n ← hossz[p] − 1
2  for i ← 1 to n
3      do for j ← i to n
4             do m[i, j] ← ∞
5  return LÁNCOT-KERES(p, 1, n)

LÁNCOT-KERES(p, i, j)
1  if m[i, j] < ∞
2    then return m[i, j]
3  if i = j
4    then m[i, j] ← 0
5    else for k ← i to j − 1
6           do q ← LÁNCOT-KERES(p, i, k) + LÁNCOT-KERES(p, k + 1, j) + p_{i−1} p_k p_j
7              if q < m[i, j]
8                then m[i, j] ← q
9  return m[i, j]
```

A FELJEGYZÉSES-MÁTRIX-LÁNC a MÁTRIX-SZORZÁS-SORREND-hez hasonlóan az $m[1\,..\,n,\ 1\,..\,n]$ tömböt használja az $A_{i..j}$ mátrix kiszámításához szükséges szorzások minimális $m[i, j]$ számának tárolására. A tömb minden elemének értéke kezdetben ∞, ami azt jelzi, hogy a valódi értékeket még nem jegyeztük fel. Ha a LÁNCOT-KERES(p, i, j) hívásakor $m[i, j] < \infty$ (1. sor), akkor az eljárás egyszerűen visszaadja a már korábban kiszámított értéket (2. sor). Ellenkező esetben a költséget ugyanúgy kiszámoljuk, mint a REKURZÍV-MÁTRIX-LÁNC eljárásban, az $m[i, j]$ elemben tároljuk, és eredményként visszaadjuk. (A ∞ jól megfelel annak jelzésére, hogy az adott elembe még nem jegyeztük fel $m[i, j]$-t, mert ez ugyanaz az érték, amit $m[i, j]$ kap az inicializáláskor a REKURZÍV-MÁTRIX-LÁNC eljárás 3. sorában.) Így a LÁNCOT-KERES(p, i, j) mindig az $m[i, j]$ értéket adja vissza, de azt csak akkor számolja ki, amikor először hívták meg az i, j paraméterekkel.

A 15.5. ábra mutatja, hogy a FELJEGYZÉSES-MÁTRIX-LÁNC hogyan takarít meg időt a REKURZÍV-MÁTRIX-LÁNC-hoz viszonyítva: a satírozott részfák azok, amelyeket kiszámítás helyett csak elővesz a tömbből.

A FELJEGYZÉSES-MÁTRIX-LÁNC, akárcsak a dinamikus programozás MÁTRIX-SZORZÁS-SORREND eljárása, $O(n^3)$ ideig fut. A $\Theta(n^2)$ mátrixelemet a FELJEGYZÉSES-MÁTRIX-LÁNC sorában egyszer inicializáljuk. A LÁNCOT-KERES hívásait két kategóriába sorolhatjuk:

1. amikor $m[i, j] = \infty$, és így a 3–9. sorokat hajtjuk végre, és
2. amikor $m[i, j] < \infty$, és így a LÁNCOT-KERES a 2. sorában tér vissza.

[4] Ez a megközelítés feltételezi, hogy a részfeladatok paraméterei előre ismertek, és a tömbbeli pozíciók és a részfeladatok kapcsolata meghatározott. A feljegyzéses módszer másik változatában a részfeladatok paramétereire mint kulcsokra alapozott hasító eljárást használunk.

$\Theta(n^2)$ első típusú hívás van, minden elemhez pontosan egy. Minden második típusú hívást egy első típusú hívás tesz meg rekurzív módon. Ha egy LÁNCOT-KERES rekurzív hívást tesz, akkor $O(n)$ számúra van szüksége. Ezért $O(n^3)$ számú második típusú hívás van, melyek mindegyike $O(1)$ ideig tart, az első típusúak ideje pedig $O(n)$, plusz a rekurzív hívásokra fordított idő. Innen kapjuk a $O(n^3)$ teljes futási időt. Tehát a feljegyzéses eljárásunk az $\Omega(2^n)$-es módszert egy $O(n^3)$ algoritmussá javítja.

Összefoglalva: a véges sok mátrix összeszorzásának problémája $O(n^3)$ idő alatt megoldható akár felülről lefelé a feljegyzéses algoritmussal, akár alulról felfelé a dinamikus programozással. Mindkét módszer kihasználja az átfedő részfeladatok által nyújtott előnyöket. Összesen csak $\Theta(n^2)$ különböző részfeladat van, és ezeket mindkét módszer csak egyszer oldja meg. Feljegyzés nélkül a természetes rekurzív módszer exponenciális ideig fut, mivel a részfeladatokat ismételten megoldja.

Általában, ha az összes részfeladatot legalább egyszer meg kell oldani, akkor az alulról felfelé haladó dinamikus programozás egy konstans tényezővel jobb a feljegyzéses algoritmusnál, mert a rekurzióhoz szükséges adminisztráció elmarad, és a tömb kezelése is egyszerűbb. Ezenkívül vannak olyan problémák, amelyeknél a tömbelemek elérésének a dinamikus programozásnál szokásos módja kiaknázható a futási idő vagy a szükséges memória további csökkentésére. Ezzel szemben, ha a részfeladatok közül nem kell mindet megoldani, akkor a feljegyzéses megoldásnak megvan az az előnye, hogy csak azokat a részfeladatokat oldja meg, amelyekre valóban szükség van.

Gyakorlatok

15.3-1. Mi a hatékonyabb eljárás a mátrixok szorzásánál a szorzási műveletek optimális számának meghatározására: leszámlálni az összes zárójelezést és mindegyikhez meghatározni a szorzások számát, avagy futtatni a REKURZÍV-MÁTRIX-LÁNC eljárást? Indokoljuk meg a választ.

15.3-2. Rajzoljuk le a 2.3.1. pont ÖSSZEFÉSÜLŐ-RENDEZÉS eljárásának rekurziós fáját 16 elem esetén. Magyarázzuk meg, hogy miért nem hatékony a feljegyzéses módszer az ÖSSZEFÉSÜLŐ-RENDEZÉS-hez hasonló, jó oszd-meg-és-uralkodj algoritmusok felgyorsítására.

15.3-3. Tekintsük a mátrixok összeszorzása feladatának azt a változatát, amikor a szorzási műveletek számát nem minimalizálni, hanem éppen maximalizálni akarjuk. Rendelkezik-e ez a feladat az optimális részstruktúra tulajdonságával?

15.3-4. Írjuk le, hogyan lépnek fel az átfedő részfeladatok a szerelőszalag ütemezésénél?

15.3-5. Azt mondtuk, hogy a dinamikus programozásnál először meg kell oldani a részfeladatokat, és csak utána lehet kiválasztani, hogy melyiket használjuk az optimális megoldásban. Capulet professzor azt állítja, hogy nem mindig szükséges valamennyi részfeladatot megoldani. Azt állítja, hogy a mátrixok szorzásánál előre meg lehet mondani, hogy az $A_i A_{i+1} \cdots A_j$ szorzatot annál a k indexnél kell kettévágni, ahol a $p_{i-1} p_k p_j$ szorzat a legkisebb. Adjunk példát arra, hogy ez a mohó eljárás nem ad optimális megoldást.

15.4. A leghosszabb közös részsorozat

Biológiai alkalmazásokban gyakran össze akarjuk hasonlítani két vagy több élőlény DNS-ét. A DNS-t bizonyos, ***bázisnak*** nevezett molekulák szekvenciái alkotják. A lehetséges bázisok: adenin, guanin, citozin és timin. A bázisokat a kezdőbetűjükkel jelöljük. A DNS

szerkezete leírható a véges {A,C,G,T} halmazon vett sztringgel. (A C Függelékben található a sztring definíciója.) Például az egyik élőlény DNS-e lehet S_1 = ACCGGTCGAG TGCGCGGAAGCCGGCCGAA, míg a másiké S_2 = GTCGTTCGGAATGCCGTTGCTCTG TAAA. Két élőlény DNS-ét például abból a célból hasonlíthatjuk össze, hogy megállapítsuk, hogy mennyire hasonlók, azaz valamilyen mérték szerint a két lény mennyire közel áll egymáshoz. A hasonlóságot sokféleképpen lehet definiálni. Például azt mondhatjuk, hogy két DNS szekvencia hasonló, ha az egyik a másiknak részsztringje. (A 32. fejezet ad ezen probléma megoldására algoritmust.) Példánkban sem S_1, sem S_2 nem része a másiknak. Azt is mondhatjuk, hogy két szekvencia akkor hasonló, ha kevés változtatással vihet ők át egymásba. (A 15-3. feladat foglalkozik ezzel a kérdéssel.) A hasonlóság megállapításának egy harmadik módja, hogy keresünk egy olyan S_3 szekvenciát, amelyik olyan bázisokból áll, amelyek ugyanabban a sorrendben, de nem feltétlenül egymás után fordulnak el ő mind S_1-ben, mind S_2-ben. Minél hosszabb S_3, annál nagyobb a hasonlóság. A példánkban a leghosszabb S_3 szekvencia GTCGTCGGAAGCCGGCCGAA.

A mondottak a leghosszabb közös részsorozat problémáját jelentik. Egy sorozat részsorozata egy olyan sorozat, amit az adott sorozatból néhány (esetleg nulla) elem elhagyásával nyerünk. Formálisan, ha az adott sorozat $X = (x_1, x_2, \ldots, x_m)$, akkor egy másik $Z = (z_1, z_2, \ldots, z_k)$ sorozat akkor ***részsorozata*** X-nek, ha létezik X indexeinek egy szigorúan növő $(i_1, i_2, \ldots, i_k)$ sorozata, hogy minden $j = 1, 2, \ldots, k$ esetén $x_{i_j} = z_j$. Például $Z = (B, C, D, B)$ részsorozata az $X = (A, B, C, B, D, A, B)$-nek, és ekkor az indexek megfelelő sorozata (2, 3, 5, 7).

Adott két sorozat, X és Y. Azt mondjuk, hogy egy Z sorozat ***közös részsorozatuk,*** ha Z részsorozata X-nek is és Y-nak is. Például, ha $X = (A, B, C, B, D, A, B)$ és $Y = (B, D, C, A, B, A)$, akkor a (B, C, A) sorozat közös részsorozata X-nek és Y-nak. Azonban (B, C, A) nem a *leghosszabb* közös részsorozat, mert a hossza csak 3, míg a (B, C, B, A) is közös részsorozat és a hossza 4. (A továbbiakban a leghosszabb közös részsorozat kifejezést LKR-ként rövidítjük.) A (B, C, B, A) sorozat X és Y LKR-je, miként a (B, D, B, A) sorozat is, hiszen 5 hosszú közös részsorozat nincs.

A ***leghosszabb közös részsorozat problémában*** adott két sorozat $X = (x_1, x_2, \ldots, x_m)$ és $Y = (y_1, y_2, \ldots, y_n)$. A feladat: a leghosszabb közös részsorozatukat megtalálni. Ebben az alfejezetben megmutatjuk, hogy az LKR probléma hatékonyan megoldható a dinamikus programozás segítségével.

1. lépés: A leghosszabb közös részsorozat jellemzése

A feladat megoldásának durva módszere leszámlálni X összes részsorozatát és mindegyiket ellenőrizni, hogy részsorozata-e Y-nak és a megtalált leghosszabb részsorozatot tárolni. X mindegyik részsorozata az $\{1, 2, \ldots, m\}$ indexhalmaz egy részhalmazának felel meg. Mivel X-nek 2^m részsorozata van, ez az eljárás exponenciális időt igényel, ami használhatatlanná teszi hosszabb sorozatok esetén.

Azonban az LKR probléma rendelkezik az optimális részstruktúra tulajdonsággal, ahogy azt a következő tétel mutatja. Mint látni fogjuk, a részfeladatok természetes osztálya a két bemenő sorozat „prefixei” párjainak felel meg. Pontosabban szólva az $X = (x_1, x_2, \ldots, x_m)$ sorozat i-edik ***prefixe*** $X_i = (x_1, x_2, \ldots, x_i)$ $(i = 0, 1, \ldots, m)$. Például, ha $X = (A, B, C, B, D, A, B)$, akkor $X_4 = (A, B, C, B)$ és X_0 az üres sorozat.

15.1. tétel (az LKR optimális részstruktúrája). *Legyen* $X = (x_1, x_2, \ldots, x_m)$ *és* $Y = (y_1, y_2, \ldots, y_n)$ *két sorozat és* $Z = (z_1, z_2, \ldots, z_k)$ *ezek egy LKR-je. Ekkor igazak a következő állítások:*

1. *Ha* $x_m = y_n$, *akkor* $z_k = x_m = y_n$, *és* Z_{k-1} *az* X_{m-1} *és* Y_{n-1} *egy LKR-je.*
2. *Ha* $x_m \neq y_n$, *akkor* $z_k \neq x_m$ *esetén* Z *az* X_{m-1} *és* Y *egy LKR-je.*
3. *Ha* $x_m \neq y_n$, *akkor* $z_k \neq y_n$ *esetén* Z *az* X *és* Y_{n-1} *egy LKR-je.*

Bizonyítás. (1) Ha $z_k \neq x_m$, akkor az $x_m = y_n$ elemet hozzátehetnénk Z-hez, és így X és Y egy $k+1$ hosszú közös részsorozatát kapnánk, ami ellentmond Z választásának. Ezért szükségképpen $z_k = x_m = y_n$. Most a Z_{k-1} prefix hossza $k-1$ és közös részsorozata X_{m-1}-nek és Y_{n-1}-nek. Meg akarjuk mutatni, hogy ezek egy LKR-je is. Ha W egy legalább k hosszú közös részsorozata X_{m-1}-nek és Y_{n-1}-nek, akkor ehhez hozzáfűzve az $x_m = y_n$ elemet, X és Y k-nál hosszabb közös részsorozatát kapjuk, ami ellentmond Z választásának.

(2) Ha $z_k \neq x_m$, akkor Z közös részsorozata X_{m-1}-nek és Y-nak. Ha ezeknek volna egy W közös részsorozata, mely hosszabb, mint k, akkor ez közös részsorozata volna X_m-nek és Y-nak, ami ismét ellentmond Z választásának.

(3) A bizonyítás ugyanúgy megy, mint a (2) esetben. ■

A 15.1. tétel által adott jellemzés mutatja, hogy két sorozat LKR-je a sorozatok prefixeinek egy LKR-jét tartalmazza önmagán belül. Tehát az LKR problémának megvan az optimális részstruktúra tulajdonsága. A rekurzív megoldás átfedő részfeladatokkal is rendelkezik, mint azt azonnal látni fogjuk.

2. lépés: részfeladatok rekurzív megoldása

A 15.1. tétel azt mondja, hogy csak egy vagy két részfeladat van, amit meg kell vizsgálni $X = (x_1, x_2, \ldots, x_m)$ és $Y = (y_1, y_2, \ldots, y_n)$ LKR-je megkeresésekor. Ha $x_m = y_n$, akkor X_{m-1} és Y_{n-1} LKR-jét kell megkeresni, amelyhez csatolva az $x_m = y_n$ elemet kapjuk X és Y LKR-jét. Ha $x_m \neq y_n$, akkor két részfeladatot kell megoldanunk: X_{m-1} és Y, illetve X és Y_{n-1} LKR-jét kell megkeresni. Akármelyik is a hosszabb, az az X és Y LKR-je. Mivel ezek az esetek kimerítik az összes lehetőséget, tudjuk, hogy egy részfeladat megoldása benne van X és Y LKR-jében.

Azonnal látható, hogy az LKR problémában a részfeladatok átfedőek. X és Y LKR-jének megkeresésekor szükségünk lehet X és Y_{n-1}, illetve X_{m-1} és Y LKR-jének megkeresésére. Ezek közös részfeladata X_{m-1} és Y_{n-1} LKR-jének meghatározása. Sok más részfeladatnak is van másokkal közös rész-részfeladata.

A mátrixok szorzásának problémájához hasonlóan most is szükségünk van az optimális költségre vonatkozó rekurzív összefüggés meghatározására. Legyen $c[i, j]$ X_i és Y_j LKR-jének hossza. Ha akár i, akár j nulla, azaz a sorozatok legalább egyikének hossza 0, akkor természetesen az LKR hossza is 0. Az LKR optimális részstruktúra tulajdonságából a következő rekurzív képletet nyerjük:

$$c[i, j] = \begin{cases} 0, & \text{ha } i = 0 \text{ vagy } j = 0, \\ c[i-1, j-1] + 1, & \text{ha } i, j > 0 \text{ és } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\}, & \text{ha } i, j > 0 \text{ és } x_i \neq y_j. \end{cases} \tag{15.14}$$

Vegyük észre, hogy ebben a rekurzív képletben egy, a feladatra vonatkozó feltétel megadja, hogy mely részproblémákat kell vizsgálni. Ha $x_i = y_j$, akkor kizárólag X_{i-1} és Y_{j-1} LKR-jét kell megtalálni. Különben pedig két részfeladatot kell vizsgálni, X_i és Y_{j-1}, valamint X_{i-1} és Y_j LKR-jét kell előállítani. A szerelőszalag ütemezésével és a mátrixok szorzásával foglalkozó, korábban vizsgált dinamikus programozási algoritmusokban egyetlen részfeladatot sem zártunk ki valamely, a feladatra vonatkozó feltétel alapján. Az LKR meghatározása nem az egyetlen dinamikus programozási eljárás, amely alkalmas feltétel alapján kizár részfeladatokat. Például az editálási távolság feladat (15-3. feladat) is rendelkezik ezzel a tulajdonsággal.

3. lépés: LKR hosszának kiszámítása

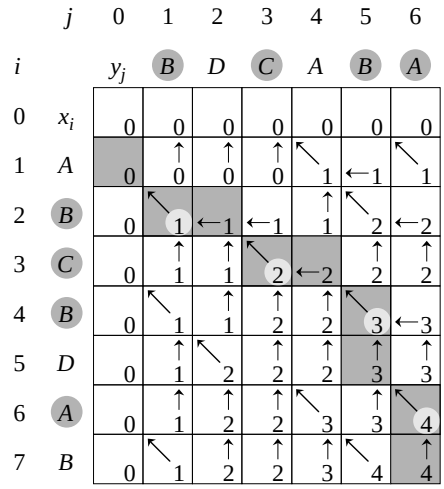
A (15.14) egyenletre alapozva könnyen írhatunk egy exponenciális ideig futó rekurzív algoritmust két sorozat LKR-jének előállítására. Mivel azonban csak $\Theta(mn)$ különböző részfeladat van, a dinamikus programozást használhatjuk a megoldás alulról felfelé történő kiszámítására.

Az LKR-hossz eljárás bemenete az X és Y sorozat. A $c[i, j]$ értékeket a $c[0\,..\,m, 0\,..\,n]$ tömbben tárolja, melynek elemeit sorfolytonosan számítja ki. (Azaz először c első sorát tölti fel balról jobbra, azután a második sort stb.) Az optimális megoldás megkeresésének leegyszerűsítése végett egy $b[1\,..\,m, 1\,..\,n]$ tömböt is kitölt. $b[i, j]$ a c tömbnek arra az elemére mutat, ahonnan $c[i, j]$ meghatározásakor a legjobb értéket kaptuk. Az eljárás a b és c tömböket adja vissza, és $c[m, n]$ X és Y LKR-jének hossza.

LKR-hossz(X, Y)

```
 1  m ← hossz[X]
 2  n ← hossz[Y]
 3  for i ← 1 to m
 4      do c[i, 0] ← 0
 5  for j ← 0 to n
 6      do c[0, j] ← 0
 7  for i ← 1 to m
 8      do for j ← 1 to n
 9             do if x_i = y_j
10                    then c[i, j] ← c[i − 1, j − 1] + 1
11                         b[i, j] ← „↖”
12                    else if c[i − 1, j] ≥ c[i, j − 1]
13                           then c[i, j] ← c[i − 1, j]
14                                b[i, j] ← „↑”
15                           else c[i, j] ← c[i, j − 1]
16                                b[i, j] ← „←”
17  return c és b
```

A 15.6. ábra mutatja az LKR-hossz eljárás által készített tömböket, ha $X = (A, B, C, B, D, A, B)$ és $Y = (B, D, C, A, B, A)$. Az eljárás futási ideje $O(mn)$, hiszen minden tömbelem kiszámításának ideje $O(1)$.



15.6. ábra. Az $X = (A, B, C, B, D, A, B)$ és $Y = (B, D, C, A, B, A)$ esetére az LKR-hossz által meghatározott c és b tömb. Az i sor és j oszlop kereszteződésében álló négyzet tartalmazza $c[i, j]$ értékét és a $b[i, j]$ értékének megfelelő nyilat. A tömb jobb alsó sarkában lévő $c[7, 6]$-ban található 4 érték X és Y (B, C, B, A) LKR-jének a hossza. Ha $i, j > 0$, akkor $c[i, j]$ értéke csak a már korábban kiszámított $c[i-1, j]$, $c[i, j-1]$ és $c[i-1, j-1]$ értékétől, valamint attól függ, hogy igaz-e az $x_i = y_j$ egyenlőség. Az LKR elemeinek előállításához a jobb alsó sarokból indulva kell követni a $b[i, j]$ nyilakat. A ténylegesen bejárt út satírozva látható. Az útban minden „↖” nyíl az LKR egy elemének felel meg, amire teljesül, hogy $x_i = y_j$, és ez az ábrán kiemelten látható.

4. lépés: Egy LKR megszerkesztése

Az LKR-hossz eljárás által készített b tömb szolgál az $X = (x_1, x_2, \ldots, x_m)$ és $Y = (y_1, y_2, \ldots, y_n)$ egy LKR-jének gyors előállítására. A nyilak irányában a $b[m, n]$ elemből kiindulva végig kell csak haladni a tömbön. Amikor egy „↖” jelet találunk $b[i, j]$-ben, az azt jelenti, hogy az $x_i = y_j$ elem benne van az LKR-ben. Evvel a módszerrel az LKR elemeit fordított sorrendben kapjuk meg. A következő rekurzív eljárás X és Y LKR-jének elemeit a helyes, eredeti sorrendben nyomtatjuk ki. A kezdeti hívása LKR-NYOMTAT(b, X,*hossz*[X],*hossz*[Y]).

LKR-NYOMTAT(b, X, i, j)

```
1  if i = 0 vagy j = 0
2     then return
3  if b[i, j] = „↖”
4     then LKR-NYOMTAT(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = „↑”
7     then LKR-NYOMTAT(b, X, i − 1, j)
8  else LKR-NYOMTAT(b, X, i, j − 1)
```

A 15.6. ábra b tömbje esetén ez az eljárás a „*BCBA*” sorozatot nyomtatja ki. A szükséges műveleti idő $O(m+n)$, hiszen a rekurzió minden lépésében i és j legalább egyikének csökken az értéke.

A program javítása

Gyakori, hogy miután az ember kifejlesztett egy algoritmust, azt veszi észre, hogy az általa használt idő vagy memória csökkenthető. Ez különösen igaz a dinamikus programozásból közvetlenül származó eljárásokra. Egyes módosítások egyszerűsíthetik a programot és javíthatják a konstans tényezőket, de aszimptotikusan nem javítják annak teljesítményét. Más változtatások azonban aszimptotikusan jelentős megtakarítást hozhatnak időben és memóriában.

Például a b tömb használatára nincs szükség. Minden $c[i, j]$ elem csak három másik elemtől függ, melyek $c[i-1, j-1]$, $c[i-1, j]$ és $c[i, j-1]$. Ha ismert $c[i, j]$, akkor $O(1)$ időben meg tudjuk mondani, hogy melyiket használtuk anélkül, hogy a b tömbhöz hozzá kellene nyúlni. Így az LKR-NYOMTAT nevű eljáráshoz hasonló módon $O(m + n)$ lépésben tudunk egy LKR-t előállítani. (A 15.4-2. gyakorlat kéri a pszeudoprogram megírását.) Habár így megtakarítunk $\Theta(mn)$ memóriát, de a szükséges memória aszimptotikusan nem csökken, hiszen a c tömbhöz mindenképpen szükség van $\Theta(mn)$ memóriára.

Az LKR-HOSSZ eljárásban azonban aszimptotikusan is csökkenthetjük a memóriaigényt, mivel egyszerre csak a c tömb két sorára van szükség: az éppen kiszámolás alattira és az azt megelőzőre. (Valójában valamivel több memória, mint c egy sora elegendő az LKR hosszának kiszámításához, lásd a 15.4-4. gyakorlatot.) Ez a javítás azonban csak akkor működik, ha csak az LKR hosszára vagyunk kíváncsiak. Ha azt meg is kell szerkeszteni, akkor ez a kisebb tömb nem ad elegendő információt ahhoz, hogy ezt $O(m + n)$ lépésben megtehessük.

Gyakorlatok

15.4-1. Határozzuk meg (1,0,0,1,0,1,0,1) és (0,1,0,1,1,0,1,1,0) egy LKR-jét.

15.4-2. Mutassuk meg, hogy hogyan lehet megkapni egy LKR-t a b tömb használata nélkül a c tömbből az eredeti $X = (x_1, x_2, \ldots, x_m)$ és $Y = (y_1, y_2, \ldots, y_n)$ sorozatok segítségével $O(m + n)$ lépésben.

15.4-3. Adjuk meg az LKR-HOSSZ eljárás feljegyzéses változatát, amely $O(mn)$ ideig fut.

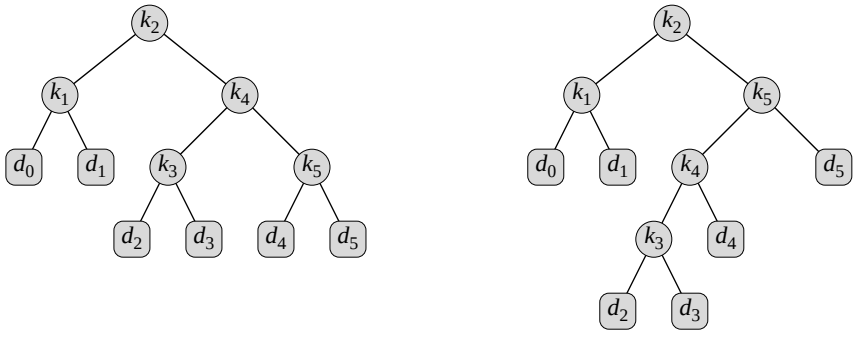
15.4-4. Mutassuk meg, hogy hogyan lehet az LKR hosszát kiszámítani a c tömb $2\min(m, n)$ elemét és további $O(1)$ memóriát használva. Majd mutassuk meg, hogy ugyanehhez elég a tömb $\min(m, n)$ eleme és $O(1)$ további memória.

15.4-5. Adjunk meg $O(n^2)$ idejű algoritmust egy n számból álló sorozat leghosszabb monoton növő részsorozatának meghatározására.

15.4-6.⋆ Adjunk meg $O(n \log n)$ idejű algoritmust n számból álló sorozat leghosszabb monoton növő részsorozatának meghatározására. (*Útmutatás.* Vegyük észre, hogy ha egy i hosszú részsorozat része lehet a keresett leghosszabb monoton részsorozatnak, akkor utolsó eleme legalább akkora, mint egy ugyancsak szóba jövő $i-1$ hosszú részsorozat utolsó eleme. Az eredeti sorozaton keresztül összekapcsolva tartsuk nyilván ezen jelölteket.)

15.5. Optimális bináris kereső fák

Tegyük fel, hogy angol szöveget franciára fordító programot akarunk kifejleszteni. Minden angol szó mindegyik szövegbeli előfordulásakor ki kell keresni a francia megfelelőjét. Ezen műveletet úgy is kivitelezhetjük, hogy egy bináris keresőfát építünk n angol szóból, mint kulcsokból és a francia megfelelőjükből, mint kapcsolt adatokból. Mivel minden



(a) (b)

15.7. ábra. Két bináris keresési fa $n = 5$ kulcs esetén, amikor a valószínűségek a következők:

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

(a) Bináris kereső fa, melyben a keresés költségének várható értéke 2,80. **(b)** Bináris kereső fa, melyben a keresés költségének várható értéke 2,75. Ez a fa optimális.

szóra külön keresnünk kell a fában, ezért a teljes keresési időt olyan rövidnek szeretnénk, amennyire csak lehetséges. A piros-fekete vagy más kiegyensúlyozott fával előfordulásonként $O(\log n)$ keresési időt tudnánk biztosítani. Azonban a szavak előfordulási gyakorisága különböző. Előfordulhatna, hogy olyan gyakran használt szó, mint „the” a gyökértől távol lenne, míg az igen ritka „mycophagist” pedig a gyökér közelében. Egy ilyen elrendezés jelentősen lelassítaná a keresést, mert a fában egy kulcs megtalálásához eggyel több csúcsot kell felkeresni, mint a kulcsot tartalmazó csúcs mélysége. Tehát a gyakori szavakat a gyökérhez közel szeretnénk elhelyezni.[5] Továbbá a szövegben lehetnek olyan szavak, amelyeknek egyáltalán nincs francia megfelelője, ezért ezeknek a szavaknak benne sem kell lenni a fában. Feltéve, hogy ismerjük a szavak előfordulási gyakoriságát, hogyan kell a fát megszerkeszteni, hogy minimalizáljuk a felkeresett csúcsok számát?

Amire szükségünk van, azt úgy hívják, hogy ***optimális bináris kereső fa.*** Formálisan megfogalmazva arról van szó, hogy adott különböző kulcsoknak egy $K = (k_1, k_2, \ldots, k_n)$ rendezett sorozata (ahol $k_1 < k_2 < \ldots < k_n$), és ezekből akarunk egy bináris fát felépíteni. Minden k_i kulcshoz ismert annak p_i előfordulási valószínűsége. Egyes keresések azonban olyan értékekre vonatkozhatnak, melyek nincsenek benne K-ban. Emiatt van $n + 1$ további pótkulcsunk is, melyeket $d_0, d_1, \ldots, d_n$ jelöl. d_0 képviseli az összes k_1-nél kisebb értéket, d_n az összes k_n-nél nagyobbat, míg $i = 1, \ldots, n-1$ esetén d_i a szigorúan k_i és k_{i+1} közé esőket. Minden d_i pótkulcsra is ismert annak q_i előfordulási valószínűsége. A 15.7. ábra mutat két bináris fát az $n = 5$ esetre. Minden k_i kulcs belső pont, míg minden pótkulcs végpont. Minden keresés vagy sikeres, azaz megtalálja valamelyik k_i kulcsot, vagy nem sikeres, azaz egy pótkulcsra fut rá. Emiatt

[5] Ha a szöveg témája az ehető gombák, akkor lehet, hogy a „mycophagist”-ot is a gyökérhez közel szeretnénk látni.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1. \tag{15.15}$$

Mivel ismertek a kulcsok és pótkulcsok előfordulási valószínűségei, egy adott T bináris fára megállapítható a keresés költségének várható értéke. Tegyük fel, hogy kulcs megtalálásának a költsége a keresés során megvizsgált kulcsok száma, azaz a *kulcs mélysége a fában* + 1. Ekkor a T-beli keresés várható értéke

$$\begin{aligned} \mathrm{E}[\text{a keresés költsége } T\text{-ben}] &= \sum_{i=1}^{n} (\text{mélység}_T(k_i)+1)p_i + \sum_{i=0}^{n} (\text{mélység}_T(d_i)+1)q_i \\ &= 1 + \sum_{i=1}^{n} \text{mélység}_T(k_i)p_i + \sum_{i=0}^{n} \text{mélység}_T(d_i)q_i, \end{aligned} \tag{15.16}$$

ahol mélység_T jelöli a csúcs T-beli mélységét. Az utolsó egyenlőség (15.5)-ből következik. A 15.7(a) ábra alapján a keresés költségének várható értéke csúcsonként az alábbi:

csúcs	mélység	valószínűség	költség
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40
Összesen			2,80

Adott valószínűségek mellett az a célunk, hogy olyan fát szerkesszünk, amelyre a keresés várható költsége minimális. Az ilyen fát ***optimális bináris kereső fának*** nevezzük. A 15.7(b) ábrán látható egy, az ábra feliratában található valószínűségekhez tartozó optimális bináris kereső fa, melynek költsége 2.75. A példa azt is mutatja, hogy egy optimális keresős fa mélysége nem feltétlenül a lehető legkisebb. Az sem igaz, hogy a legnagyobb valószínűségű kulcsot feltétlenül a gyökérbe kell tenni. Az összes kulcs közül k_5 valószínűsége a legnagyobb, mégis az optimális keresős fa gyökerében k_2 ül. (Azon fák körében, melyek gyökerében k_5 van, a minimális költség 2,85.)

Ugyanúgy, mint a mátrixok szorzásánál, az összes lehetőség egyenkénti vizsgálata nem ad hatékony algoritmust. Bármely n csúcsból álló bináris fa csúcsait megcímkézhetjük a $k_1, k_2, \ldots, k_n$ kulcsokkal, és utána hozzáadhatjuk a pótkulcsokat mint leveleket. A 12-4. feladatban láttuk, hogy a bináris fák száma n csúcs esetén $\Omega(4^n/n^{3/2})$. Tehát exponenciálisan sok bináris fát kellene megvizsgálnunk egy teljes leszámlálás esetén. Nem meglepő, hogy a feladatot dinamikus programozással oldjuk meg.

1. lépés: az optimális bináris kereső fa szerkezete

Egy részfákra vonatkozó megfigyeléssel kell kezdenünk az optimális bináris keres ő fa optimális részstruktúrájának leírását. Tekintsük egy bináris fa egy részfáját. Ennek a kulcsok $k_i, \ldots, k_j$ összefüggő tartományát kell tartalmaznia valamely alkalmas $1 \leq i \leq j \leq n$ esetén. Továbbá annak a fának, amelyik a $k_i, \ldots, k_j$ kulcsokat tartalmazza, ugyancsak tartalmaznia kell a $d_{i-1}, \ldots, d_j$ pótkulcsokat mint leveleket.

Az optimális részstruktúra azt jelenti, hogy ha a T fa egy T' részfája, mely a $k_i, \ldots, k_j$ kulcsokat tartalmazza, optimális az ugyanezen kulcsokon és a $d_{i-1}, \ldots, d_j$ pótkulcsokon értelmezett részfeladaton. A szokásos szétvágás és összeragasztás érvelés itt is működik. Ha volna egy T'' részfa, amelynek várható költsége kisebb volna T' várható költségénél, akkor ezt téve T' helyére olyan fát kapnánk, ami jobb T-nél, ez pedig ellentmond T optimalitásának.

Erre a tulajdonságra ismét csak azért van szükségünk, hogy megmutassuk, hogy meg tudunk szerkeszteni egy optimális megoldást a részfeladatok optimális megoldásaiból. A $k_i, \ldots, k_j$ kulcsokat tartalmazó optimális részfában valamelyik kulcs, mondjuk k_r ($i \leq r \leq j$) lesz a gyökérben. A gyökér bal oldali részfája a $k_i, \ldots, k_{r-1}$ (és velük együtt a $d_{i-1}, \ldots, d_{r-1}$ pótkulcsokból) kulcsokból, a jobb oldali részfa pedig a $k_{r+1}, \ldots, k_j$ kulcsokból (és a $d_r, \ldots, d_j$ pórkulcsokból) áll. Ha a gyökér minden k_r ($i \leq r \leq j$) jelöltje esetében meghatározzuk a $k_i, \ldots, k_{r-1}$, valamint a $k_{r+1}, \ldots, k_j$ kulcsokat tartalmazó optimális kereső fát, garantált, hogy az egész feladat optimális megoldását is meg fogjuk találni.

Meg kell jegyezni valamit az „üres” részfákkal kapcsolatban. Tegyük fel, hogy a $k_i, \ldots, k_j$ kulcsokat tartalmazó részfa gyökerében k_i ül. Ekkor a gyökér bal oldali részfájának kulcsai $k_i, \ldots, k_{i-1}$. Természetes volna ezt a részfát úgy értelmezni, hogy nem tartalmaz kulcsot. Azonban észben kell tartanunk, hogy a részfákban pótkulcsok is vannak. Ezért azt a konvenciót alkalmazzuk, hogy az a részfa, aminek a kulcsai $k_i, \ldots, k_{i-1}$, az egyetlen d_{i-1} pótkulcsból áll. Hasonlóképpen, ha k_j van a gyökérben, akkor ennek jobb oldali, $k_{j+1}, \ldots, k_j$ kulcsokból álló részfájában egyedül a d_j pótkulcs található.

2. lépés: Egy rekurzív megoldás

Most már rekurzívan is tudjuk definiálni az optimális megoldást. A részfeladatok közül vizsgáljuk azt, amelyik a $k_i, \ldots, k_j$ kulcsokat tartalmazza, ahol $i \geq 1$, $j \leq n$ és $j \geq i - 1$. (Ha $j = i - 1$, akkor nincs kulcs a fában, csak a d_{i-1} pótkulcs.) Legyen $e[i, j]$ a $k_i, \ldots, k_j$ kulcsokból álló optimális bináris kereső fában a keresés várható költsége. Mi végül az $e[1, n]$ értékre vagyunk kíváncsiak.

Egyszerű az az eset, amikor $j = i - 1$. Ekkor csak a d_{i-1} pótkulcsunk van, és a keresés várható költsége $e[i, i - 1] = q_{i-1}$.

Amikor $j \geq i$, ki kell választani $k_i, \ldots, k_j$ közül azt a k_r-t, ami a gyökérben lesz. Ha a választás megtörtént, akkor ezen feltétel mellett a $k_i, \ldots, k_{r-1}$, illetve $k_{r+1}, \ldots, k_j$ kulcsokat tartalmazó bal, illetve jobb oldali részfák segítségével szerkesztünk egy-egy optimális bináris kereső fát. Mi történik egy részfa várható költségével, amikor az egy csúcshoz tartozó részfává válik? Minden kulcs mélysége növekszik eggyel. A (15.16) képlet szerint a várható költség a részfához tartozó valószínűségek összegével növekszik. A $k_i, \ldots, k_j$ kulcsok esetén ezt az összeget jelölje

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l. \tag{15.17}$$

Tehát, ha k_r van a $k_i, \ldots, k_j$ kulcsok optimális részfájának gyökerében, akkor

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)).$$

Vegyük észre, hogy

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j).$$

Így $e[i, j]$ az

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \tag{15.18}$$

alakba írható.

A (15.18) rekurzív egyenlet feltételezi, hogy tudjuk, hogy melyik k_r kulcs kerül a gyökérbe. Természetesen azt kell választani, amelyik a legkisebb várható költséget adja. Ezért a rekurzív összefüggésünk végső alakja:

$$e[i, j] = \begin{cases} q_{i-1}, & \text{ha } j = i-1, \\ \min_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + w(i, j) \}, & \text{ha } i \le j. \end{cases} \tag{15.19}$$

Az $e[i, j]$ értékek adják az optimális keresések költségének várható értékét. Hogy meg tudjuk szerkeszteni az optimális bináris kereső fát, bevezetjük a *gyökér* tömböt, amelynek *gyökér*$[i, j]$ $(1 \le i \le j \le n)$ eleme megadja a $k_i, \ldots, k_j$ kulcsokat tartalmazó optimális részfa gyökerében lévő k_r kulcs r indexét. Habár látni fogjuk, hogy a *gyökér*$[i, j]$ értékeket hogyan kell kiszámítani, az optimális bináris kereső fa ezen értékekkel történő megszerkesztése a 15.5-1. gyakorlatra marad.

3. lépés: az optimális bináris fában való keresés várható költsége

Az eddigiekben már felfigyelhettünk a mátrixok szorzásának és az optimális bináris kereső fa meghatározásának problémájában néhány közös jellegzetességre. Mindkét esetben a részproblémákat definiáló indexek halmaza egymást követő értékekből áll. A (15.19) rekurzív egyenlet közvetlen megoldása nem hatékony, éppúgy, ahogy a mátrixok láncainak szorzásakor sem volt a közvetlen rekurzív megoldás az. Helyette az $e[i, j]$ értékeket egy $e[1\,..\,n+1, 0\,..\,n]$ tömbben tároljuk. Az első indexnek $n+1$-ig kell futnia, mert csak így kapjuk meg azt a fát, ami egyedül a d_n pótkulcsból áll, és aminek az értékét $e[n+1, n]$-ben tároljuk. Hasonlóképpen a második index 0-tól indul, mert ekkor jutunk csak hozzá az egyedül a d_0 pótkulcsból álló fához, aminek az értékét $e[1, 0]$-ban tároljuk. Csak azokat az $e[i, j]$ elemeket használjuk, amelyekre $j \ge i-1$. Ugyancsak használunk egy *gyökér*$[i, j]$ tömböt a $k_i, \ldots, k_j$ kulcsokat tartalmazó részfa gyökerének tárolására, melynek csak az $1 \le i \le j \le n$ indexű elemeit használjuk.

A hatékonyság érdekében még egy tömbre lesz szükségünk. Ahelyett, hogy $e[i, j]$ meghatározásakor $w(i, j)$ kiszámítását elölről kezdenénk, ami $\Theta(j-i)$ összeadást igényelne, ezen értékeket a $w[1\,..\,n+1, 0\,..\,n]$ tömbben tároljuk. Minden $1 \le i \le n$ esetén $w[i, i-1] = q_{i-1}$. Ha pedig $j \ge i$, akkor

$$w[i, j] = w[i, j-1] + p_j + q_j. \tag{15.20}$$

Tehát a $\Theta(n^2)$ számú $w[i, j]$ értéket egyenként $\Theta(1)$ idő alatt lehet kiszámítani.

Az alábbi programrészlet bemenő adatai n, a kulcsok száma, valamint a $p_1, \ldots, p_n$ és $q_0, \ldots, q_n$ valószínűségek, eredményül pedig az e és *gyökér* tömböket adja vissza.

OPTIMÁLIS-FA(p, q, n)

```
 1  for i ← 1 to n + 1
 2      do e[i, i − 1] ← q_{i−1}
 3         w[i, i − 1] ← q_{i−1}
 4  for l ← 1 to n
 5      do for i ← 1 to n − l + 1
 6         do j ← i + l − 1
 7            e[i, j] ← ∞
 8            w[i, j] ← w[i, j − 1] + p_j + q_j
 9            for r ← i to j
10                do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
11                   if t < e[i, j]
12                              then e[i, j] ← t
13                                   gyökér[i, j] ← r
14  return e és gyökér
```

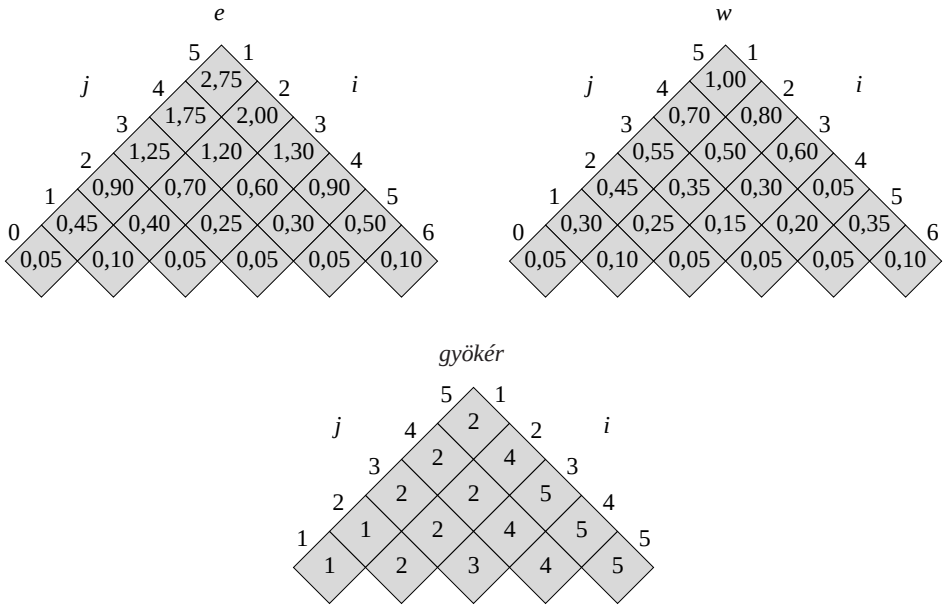
A fenti leírás és a 15.2. alfejezetbeli MÁTRIX-SZORZÁS-SORREND algoritmushoz való hasonlóság alapján az eljárás működése jól érthető. Az 1–3. sorokban található **for** ciklus inicializálja az $e[i, i-1]$ és $w[i, i-1]$ értékeket. A 4–13. sorok **for** ciklusa a (15.19) és (15.20) rekurzív képletek alapján minden $1 \le i \le j \le n$ esetén kiszámítja $e[i, j]$-t és $w[i, j]$-t. Az első iterációban, amikor $l = 1$, a ciklus az $e[i, i]$-t és $w[i, i]$-t ($i = 1, \ldots, n$) határozza meg. A második iterációban, amikor $l = 2$, $e[i, i+1]$ és $w[i, i+1]$ kerül sorra $i = 1, \ldots, n-1$ mellett és így tovább. A legbelső **for** ciklus, mely a 9–13. sorokban található, kipróbálja az összes lehetséges r indexet, hogy melyik k_r kulcs legyen $k_i, \ldots, k_j$ optimális részfájának gyökerében. Ez a ciklus a *gyökér*$[i, j]$-be teszi az r indexet, ha egy jobb kulcsot talált.

A 15.8 ábra az OPTIMÁLIS-FA eljárás által számított $e[i, j]$, $w[i, j]$, *gyökér*$[i, j]$ táblázatokat mutatja a kulcsoknak a 15.7. ábrán található eloszlása esetén. A táblázatokat úgy forgattuk el, hogy az átlók vízszintesen helyezkednek el, akárcsak a mátrixok szorzásának ábráján. A OPTIMÁLIS-FA eljárás a sorokat alulról fölfelé, a soron belül az értékeket balról jobbra számítja ki.

Az OPTIMÁLIS-FA futási ideje $\Theta(n^3)$, akárcsak a MÁTRIX-SZORZÁS-SORREND eljárásé. Könnyű látni, hogy a számítás mennyisége $O(n^3)$, hiszen a három egymásba skatulyázott ciklus mindegyike legfeljebb n-szer fut le. A ciklusok indexhatárai nem egyeznek meg pontosan a MÁTRIX-SZORZÁS-SORREND eljáráséival, de legfeljebb 1 távolságra vannak attól. Így az OPTIMÁLIS-FA futási ideje ugyanúgy, mint a MÁTRIX-SZORZÁS-SORREND eljárásé, $\Omega(n^3)$.

Gyakorlatok

15.5-1. Írjuk meg az OPTIMÁLIS-FA-SZERKESZTÉSE(*gyökér*) eljárás pszeudokódját, amely adott *gyökér* tömb esetén megadja az optimális kereső fa szerkezetét. A 15.8. ábra példája esetében az eljárásnak a következőt kell nyomtatnia a 15.7(b) ábrán megadott optimális bináris kereső fa esetén:



15.8. ábra. Az OPTIMÁLIS-FA eljárás által számított $e[i, j]$, $w[i, j]$, $gyökér[i, j]$ táblázatok a kulcsok 15.7. ábrán található eloszlása esetén. A táblákat úgy forgattuk, hogy az átlók vízszintesen láthatók.

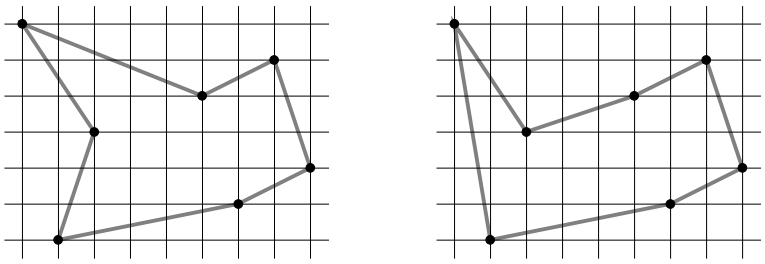
k_2	a gyökér	
k_1	a bal oldali gyermeke	k_2-nek
d_0	a bal oldali gyermeke	k_1-nek
d_1	a jobb oldali gyermeke	k_1-nek
k_5	a jobb oldali gyermeke	k_2-nek
k_4	a bal oldali gyermeke	k_5-nek
k_3	a bal oldali gyermeke	k_4-nek
d_2	a bal oldali gyermeke	k_3-nak
d_3	a jobb oldali gyermeke	k_3-nak
d_4	a jobb oldali gyermeke	k_4-nek
d_5	a jobb oldali gyermeke	k_5-nek

15.5-2. Határozzuk meg az optimális bináris kereső fa szerkezetét és a várható költségét, ha $n = 7$ és a kulcsok valószínűsége a következő:

i	0	1	2	3	4	5	6	7
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,04
q_i	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05.

15.5-3. Tegyük fel, hogy ahelyett, hogy a $w[i, j]$ értékeket karbantartanánk, az eljárás 8. sorában a (15.17) egyenlet alapján közvetlenül számoljuk ki, és ezt a kiszámított értéket használjuk a 10. sorban. Hogyan befolyásolja ez a OPTIMÁLIS-FA futási idejét?

15.5-4. ⋆ Knuth [184] megmutatta, hogy az optimális részfáknak mindig vannak olyan gyökerei, hogy $gyökér[i, j-1] \le gyökér[i, j] \le gyökér[i+1, j]$ minden $1 \le i < j \le n$ esetén. Ezt a tényt felhasználva módosítsuk a OPTIMÁLIS-FA eljárást, hogy futási ideje $\Theta(n^2)$ legyen.



(a) (b)

15.9. ábra. Egy síkbeli rácson lévő hét pont. **(a)** A 24,89 hosszú legrövidebb körút. Ez a körút nem kétirányú. **(b)** A legrövidebb kétirányú körút, melynek hossza 25,58.

Feladatok

15-1. Kétirányú euklideszi utazóügynök feladat

Az ***euklideszi utazóügynök feladatban*** adott a síkon n pont és a feladat az őket összekötő legrövidebb körút megkeresése. A 15.9(a) ábra egy 7 pontból álló feladat esetén mutatja a megoldást. Az általános feladat NP-teljes, és ezért azt hisszük, hogy a megoldása polinomiálisnál több időt igényel (lásd 34. fejezet).

J. L. Bentley javasolta, hogy egyszerűsítsük le a problémát olyan módon, hogy csak az úgynevezett ***kétirányú körutakat*** engedjük meg. Egy körutat kétirányúnak nevezünk, ha a bal szélső pontból indul, mindig balról jobbra halad, és amikor elérte a jobb szélső pontot, akkor hasonló módon jobbról balra haladva tér vissza a kiindulási helyre. A 15.9(b) ábra mutatja ugyanazon a 7 ponton a legrövidebb kétirányú körutat. Ebben az esetben létezik polinom idejű algoritmus.

Adjunk $O(n^2)$ idejű algoritmust az optimális kétirányú körút meghatározására. Feltehetjük, hogy nincs két olyan pont, aminek azonos lenne az x koordinátája. (*Útmutatás.* Balról jobbra tartsuk nyilván az optimális körút két lehetséges ágát.)

15-2. Arányos nyomtatás

Egy bekezdést kell arányosan kinyomtatni a nyomtatón. Az input szöveg n szó sorozata. A szavak karakterben mért hossza rendre $l_1, l_2, \ldots, l_n$. A bekezdést olyan sorokba akarjuk arányosan kinyomtatni, amelyek mindegyikében legfeljebb M karakterre van hely. Az „arányosságra” a következő kritériumot adjuk meg. Ha egy sor az i-től j-ig terjedő szavakat tartalmazza, akkor ezek között mindig egy szóköz van, míg a sor végén további $M - j + i - \sum_{k=i}^{j} l_k$ extra szóköz. Az utóbbi mennyiségnek természetesen nemnegatívnak kell lennie, hogy a szavak beleférjenek a sorba. Az utolsó sor kivételével a sorok végén található extra szóközök száma köbeinek összegét akarjuk minimalizálni. Adjunk meg egy dinamikus programozási algoritmust egy n szóból álló bekezdés arányos nyomtatására. Elemezzük algoritmusunk futási idejét és a szükséges memória nagyságát.

15-3. Editálási távolság

Amikor egy $x[1\,..\,m]$ „forrás” karaktersorozatot kicserélünk egy $y[1\,..\,n]$ „eredmény” karaktersorozatra, akkor számos mód van ennek végrehajtására. Az a feladatunk, hogy megadjuk transzformációk egy olyan sorozatát, ami x-et y-ra cseréli. A közbülső állapotok tárolására egy elegendően hosszú z tömböt használunk. Kezdetben z üres, és az eljárás végén a

$z[j] = y[j]$ ($j = 1, \ldots, n$) egyenleteknek kell teljesülniük. Az eljárás folyamán i az x tömb, j pedig a z aktuális indexe. Kezdetben $i = j = 1$. x minden karakterét meg kell vizsgálni, ami azt jelenti, hogy a transzformációs műveletek sorozatának végén az $i = m+1$ egyenlőségnek kell fennállnia.

Hat különböző transzformációs művelet van:

Másolás. Egy karakter átmásolása a $z[j] \leftarrow x[i]$ értékadás útján, majd i és j eggyel való növelése. A művelet megvizsgálja $x[i]$-t.

Csere. Az $x[i]$ karakter cseréje egy másik c karakterre a $z[j] \leftarrow c$ értékadás útján, majd i és j eggyel való növelése. A művelet megvizsgálja $x[i]$-t.

Törlés. Az $x[i]$ karakter törlése úgy, hogy i-t növeljük eggyel, míg j változatlan. A művelet megvizsgálja $x[i]$-t.

Beszúrás. Egy c karaktert beszúrunk a z tömbbe a $z[j] \leftarrow c$ értékadással. A j indexet növeljük eggyel, i változatlan. A művelet nem vizsgálja meg x egyetlen karakterét sem.

Felcserélés. Az x tömb következő két karakterét átmásoljuk z-be, de fordított sorrendben. A kiadott értékadások: $z[j] \leftarrow x[i+1]$, $z[j+1] \leftarrow x[i]$, $i \leftarrow i+2$, $j \leftarrow j+2$. A művelet megvizsgálja mind $x[i]$-t, mind $x[i+1]$-et.

Befejezés. x további karaktereinek elhagyása az $i \leftarrow m+1$ értékadással, ami csak az utolsó művelet lehet, befejezzük a transzformációt. Ez a művelet megvizsgálja x összes még megvizsgálatlan elemét.

Példaként megmutatjuk, hogyan lehet az `algorithm` szót kicserélni az `altruistic` szóra. Az aláhúzott karakter mindig a művelet utáni $x[i]$, illetve $z[j]$:

művelet	x	z
kezdeti karaktersorozatok	a̲lgorithm	̲
másol	al̲gorithm	a̲
másol	alg̲orithm	al̲
cserél t-re	alg̲orithm	alt̲
törlés	algo̲rithm	alt̲
másol	algor̲ithm	altr̲
beszúrja u-t	algor̲ithm	altru̲
beszúrja i-t	algor̲ithm	altrui̲
beszúrja t-t	algor̲ithm	altruis̲
felcserél	algorit̲hm	altruisti̲
beszúrja c-t	algorit̲hm	altruistic̲
befejez	algorithm̲	altruistic̲

A műveleteknek számos más sorozata létezik, amely ugyanerre az eredményre vezet.

A műveletek mindegyikének meghatározott költsége van, mely a konkrét alkalmazástól függ ugyan, de feltételezhető, hogy egy alkalmazás során minden művelet költsége állandó és ismert. Továbbá feltehető, hogy a másol és a csere művelet egyedi költsége kisebb, mint a törlés és beszúrás együttes költsége, különben ezt a két műveletet nem kellene használni. A műveletek egy sorozatának költsége a benne szereplő műveletek költségeinek összege. A fenti példában az `algorithm` szónak az `altruistic` szóvá való átalakításának költsége

$$(3 \cdot \text{költség(másolás)}) + \text{költség(csere)} + \text{költség(törlés)} + (4 \cdot \text{költség(beszúrás)}) + \text{költség(felcserélés)} + \text{költség(befejezés)}.$$

a. Adott két sorozat $x[1\,..\,m]$ és $y[1\,..\,n]$ és minden művelet költsége. Az y sorozatnak az x sorozattól mért ***editálási távolsága*** az x-et y-ba transzformáló műveleti sorozatok közül a legolcsóbb költsége. Adjunk meg egy dinamikus programozási algoritmust, amely meghatározza $y[1\,..\,n]$ $x[1\,..\,m]$-től mért editálási távolságát, és kinyomtatja az optimális transzformációk sorozatát. Elemezzük az algoritmus futási idejét és memóriaigényét.

Az editálási távolság meghatározásának feladata általánosítása két DNS szekvencia sorba állításának (l. pl. [272, Section 3.2])). A DNS szekvenciák hasonlóság mérésének számos módszere alapul a sorba állításon. Az egyik ilyen módszer üres pozíciókat szúr be a két szekvenciába (beleértve azok mindkét végét is) úgy, hogy a két sorozat hossza azonos legyen, és azonos indexű pozíció ne legyen mindkettőben üres (azaz ne létezzen olyan j pozíció, hogy $x'[j]$ és $y'[j]$ is üres). Az eredeti szekvenciákat x és y, a módosítottakat x' és y' jelöli. Minden j pozícióhoz egy pontszámot rendelünk a következő módon:

- $+1$, ha $x'[j] = y'[j]$, és egyik sem üres pozíció,
- -1, ha $x'[j] \neq y'[j]$, és egyik sem üres pozíció,
- -2, ha vagy $x'[j]$ vagy $y'[j]$ üres pozíció.

A sorba állítás pontszáma az egyes pozíciók pontjainak összege. Például az $x = GATCGGCAT$ és $y = CAATGTGAATC$ szekvenciák egy lehetséges sorba állítása

G		A	T	C	G		G	C	A	T	
C	A	A	T		G	T	G	A	A	T	C
-	⋆	+	+	⋆	+	⋆	+	-	+	+	⋆

Egy pozíció alatt a + jel $+1$-et, a $-$ jel (-1)-et, ⋆ pedig (-2)-t jelent. Így ennek a sorba állításnak a teljes pontszáma $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Értelmezzük az optimális sorba állítás megtalálását mint egy olyan editálási távolság meghatározását, amelyben csak a másolás, csere, törlés, beszúrás, felcserélés és befejezés transzformációs műveleteket használhatjuk.

15-4. Vállalati fogadás tervezése

Stewart professzor egy vállalat elnökének tanácsadója, aki a társaság dolgozói részére egy fogadást akar rendezni. A társaság hierarchikus rendszerben dolgozik, azaz a főnök-beosztott reláció egy fát alkot, melynek gyökere az elnök. A személyzeti osztály minden alkalmazottnak egy kedélyességi értéket adott, ami egy valós szám. Hogy minden jelenlevő nyugodtan élvezhesse az összejövetelt, az elnök azt szeretné, hogy egyetlen alkalmazottnak se legyen ott a közvetlen főnöke.

Stewart professzor megkapta a vállalat szerkezetének leírását a 10.4. alfejezetben tárgyalt módon. A fa minden csúcsa tartalmazza még az alkalmazott nevét és kedélyességi értékét. Adjunk meg egy algoritmust a meghívandók listájának elkészítésére. A cél a vendégek kedélyességi összértékének maximalizálása. Elemezzük az algoritmus futási idejét.

15-5. Viterbi algoritmusa

Beszéd felismerésére is felhasználható a dinamikus programozás, ha megfelelő módon alkalmazzuk egy $G = (V, E)$ irányított gráfra. Minden $(u, v) \in E$ élt hangok egy Σ véges halmazából vett $\sigma(u, v)$ hanggal címkéztünk meg. Az irányított gráf egy személy által beszélt korlátozott nyelv formális modellje. A gráfban egy kitüntetett $v_0 \in V$ csúcsból induló

minden irányított út hangok valamely, a modell szerint lehetséges sorozatának felel meg. Egy irányított út címkéje az úthoz tartozó élek címkéinek láncolata lesz.

a. Adjunk meg egy hatékony algoritmust, amely ha adott az élein egy Σ ábécéből vett címkékkel rendelkező irányított gráf, ennek egy rögzített v_0 csúcsa, valamint a Σ ábécéből képzett $s = (\sigma_1, \sigma_2, \ldots, \sigma_k)$ sorozat, akkor olyan G-beli, v_0-ból induló utat ad meg, amelynek a címkéje s, feltéve, hogy ilyen út létezik. Ellenkező esetben az eljárás válasza legyen NINCS-ILYEN-ÚT. Elemezzük az algoritmus futási idejét. (*Útmutatás.* A 22. fejezet fogalmai hasznosak lehetnek.)

Tegyük fel, hogy minden $(u, v) \in E$ élhez tartozik egy nemnegatív $p(u, v)$ érték, amely az élen való áthaladásnak, azaz a megfelelő hang kiejtésének a valószínűsége. Minden csúcsnál a kimenő élek valószínűségeinek összege 1. Egy út valószínűsége a benne szereplő élek valószínűségeinek szorzata. Egy v_0-ból induló út valószínűségét úgy tekinthetjük, mint annak a valószínűségét, hogy egy ugyancsak v_0-ból induló „véletlen bolyongás” éppen ezt az utat járja be, feltéve, hogy minden u csúcsban annak az élnek a kiválasztása, amelyiken továbbhaladunk véletlen módon, az u-ból induló élek valószínűségei szerint történik.

b. Egészítsük ki az (a) részben adott választ úgy, hogy ha az algoritmus megad egy utat, akkor ez a *legvalószínűbb út* legyen, amely v_0-ból indul és a címkéje s. Elemezzük az algoritmus futási idejét.

15-6. Mozgás a sakktáblán

Tegyük fel, hogy adott egy $n \times n$ méretű sakktábla és azon egy figura, melyet a tábla alsó széléről kell elvinni a felső szélére úgy, hogy minden kockából a következő három kocka valamelyikébe léphetünk:

1. a pillanatnyi kocka fölötti kockába,
2. a pillanatnyi kocka fölötti kockától balra lévő kockába, feltéve, hogy nem állunk a bal szélső oszlopban,
3. a pillanatnyi kocka fölötti kockától jobbra lévő kockába, feltéve, hogy nem állunk a jobb szélső oszlopban.

Amikor az x kockából az y kockába lépünk, $p(x, y)$ dollárt kapunk. A $p(x, y)$ dollárt minden olyan (x, y) párra megkapjuk, amelyre az x-ről y-ra való lépés megengedett. Azonban $p(x, y)$ nem feltétlenül pozitív.

Adjunk meg egy algoritmust arra, hogy a legalsó sorból bármely kockájáról a legfelső sorba vigyünk egy figurát, és annyi dollárt gyűjtünk be, amennyit csak lehetséges. Az algoritmus szabadon választhatja meg az alsó sor egy mezőjét kiindulási pontként és a felső sor egy mezőjét célként. Mi az algoritmus futási ideje?

15-7. Ütemezés a haszon maximalizálása érdekében

Egy gépen n számú, $a_1, a_2, \ldots, a_n$-nel jelölt munkát kell elvégezni. Minden a_j munka esetében a megmunkálási idő t_j, a munkán elérhető haszon p_j, a határidő pedig d_j. A gép egyszerre csak egy munkán tud dolgozni. Semelyik a_j munka megmunkálása sem szakítható meg, hanem annak a megkezdéstől t_j ideig, azaz a befejezéséig folynia kell. Az a_j munkáért járó γ_j profitot pontosan akkor kapjuk meg, ha a munka határidőre elkészült, különben az általunk elért haszon 0. Adjunk meg egy algoritmust a teljes hasznot maximalizáló ütemezés meghatározására, feltéve, hogy minden megmunkálási idő 1 és n közé eső egész. Mi az algoritmus futási ideje?

Megjegyzések a fejezethez

A dinamikus programozást R. Bellman kezdte módszeresen tanulmányozni 1955-ben. A „programozás” szó itt is és a lineáris programozás esetében is a táblázatos megoldási módszer használatára utal. Habár olyan optimalizálási technikák, amelyek a dinamikus programozás elemeit felhasználták, már korábban is ismertek voltak, Bellman fektette le a terület szilárd matematikai alapjait [34].

Hu és Shing [159, 160] adott egy $O(n \log n)$ idejű algoritmust a mátrixok véges sorozatainak szorzására.

A leghosszabb közös részsorozatra vonatkozó $O(mn)$ algoritmus folklór. Knuth [63] vetette fel, hogy vajon létezik-e szubkvadratikus eljárás az LKR feladatra. Masek és Paterson [212] igenlően válaszolta meg ezt a kérdést, miután megadtak egy $O(mn/\log n)$ futási idejű módszert, ahol $(n \leq m)$ és a sorozatok elemei egy korlátos halmazból származnak. Arra a speciális esetre, amikor egy elem sem fordul elő egynél többször egy sorozatban, Szymanski [288] megmutatta, hogy a feladat $O((m + n) \log(m + n))$ idő alatt is megoldható. Ezen eredmények többsége az editálási távolság meghatározására is igaz marad (15-3. feladat).

Gilbert és Moore [114] korai, változó hosszú bináris kódokról szóló dolgozata alkalmazható olyan optimális bináris kereső fák meghatározására, ahol minden p_i valószínűség 0. Ez a cikk egy $O(n^3)$ algoritmust ír le. A 15.5. alfejezet algoritmusa Aho, Hopcroft, Ullman [5] dolgozatából való. A 15.5-4. gyakorlat Knuthtól [184] származik. Hu és Tucker [161] talált egy $O(n^2)$ idejű és $O(n)$ memóriát igénylő algoritmust arra az említett esetre, amikor minden p_i valószínűség 0. Ezt követően Knuth [185] a futási időt $O(n \log n)$-re szorította le.

16. Mohó algoritmusok

Optimalizálási probléma megoldására szolgáló algoritmus gyakran olyan lépések sorozatából áll, ahol minden lépésben adott halmazból választhatunk. Sok optimalizálási probléma esetén a dinamikus programozási megoldás túl sok esetet vizsgál annak érdekében, hogy az optimális választást meghatározza. Ennél egyszerűbb, hatékonyabb algoritmus is létezik. A ***mohó algoritmus*** mindig az adott lépésben optimálisnak látszó választást teszi. Vagyis, a lokális optimumot választja abban a reményben, hogy ez globális optimumhoz fog majd vezetni. Ebben a fejezetben olyan optimalizálási problémákkal foglalkozunk, amelyek megoldhatók mohó algoritmussal. A fejezet olvasása előtt ajánlatos elolvasni a dinamikus programozásról szóló 15. fejezetet, különösen a 15.3. alfejezetet.

Mohó algoritmus nem mindig ad optimális megoldást, azonban sok probléma megoldható mohó algoritmussal. Először a 16.1. alfejezetben egy olyan egyszerű, de nem triviális problémát vizsgálunk, az eseménykiválasztás problémáját, amelyre a mohó algoritmus hatékony megoldást ad. A mohó algoritmushoz úgy jutunk, hogy először dinamikus programozási megoldást adunk, aztán megmutatjuk, hogy a mohó választás mindig optimális megoldást eredményez. A 16.2. alfejezetben áttekintjük a mohó stratégia elemeit, ami mohó algoritmusok helyességének közvetlenebb bizonyítását teszi lehetővé, mint a 16.2. alfejezetben tárgyalt dinamikus programozási módszer. A 16.3. alfejezetben a mohó technika egy fontos alkalmazását mutatjuk be: az adattömörítő (Huffman-) kódokat. A 16.4. alfejezetben az úgynevezett „matroidok” elméletének elemeit ismertetjük. Ez olyan kombinatorikus struktúrák elmélete, amelyek esetében a mohó algoritmus mindig optimális megoldást szolgáltat. Végül a 16.5. alfejezet a matroidok egy felhasználását illusztrálja az egységnyi idejű ütemezési probléma megoldása kapcsán.

A mohó stratégia egy igen hatékony eszköz, amely problémák széles körére alkalmazható. A további fejezetekben több olyan algoritmust látunk majd, amelyek a mohó stratégia alkalmazásainak tekinthetők, például a minimális feszítőfa algoritmusok (23. fejezet), Dijkstra algoritmusa az egy csúcsból induló legrövidebb utak meghatározására (24. fejezet), és a Chvátal-féle halmazlefedési heurisztika (35. fejezet). A minimális feszítőfák algoritmusai klasszikus példák a mohó stratégiára. Jóllehet e fejezet és a 23. fejezet egymástól függetlenül is megérthető, hasznos lehet a kettőt együtt olvasni.

16.1. Egy eseménykiválasztási probléma

Az első probléma, amit vizsgálunk, közös erőforrást igénylő, egymással versengő események ütemezése, azzal a céllal, hogy kiválasszunk egy maximális elemszámú, kölcsönösen kompatibilis eseményekből álló eseményhalmazt. Tegyük fel, hogy adott ***események*** egy $S = \{a_1, a_2, \ldots, a_n\}$ n elemű halmaza, amelyek egy közös erőforrást, például egy előadótermet kívánnak használni, amit egy időben csak egyik használhat. Minden a_i eseményhez adott az s_i ***kezdő időpont*** és az f_i ***befejező időpont,*** ahol $s_i \leq f_i$. Ha az a_i eseményt kiválasztjuk, akkor ez az esemény az $[s_i, f_i)$ félig nyitott időintervallumot foglalja le. Az a_i és a_j események ***kompatibilisek,*** ha az $[s_i, f_i)$ és $[s_j, f_j)$ intervallumok nem fedik egymást (azaz a_i és a_j kompatibilisek, ha $s_i \geq f_j$ vagy $s_j \geq f_i$). Az eseménykiválasztási probléma azt jelenti, hogy kiválasztandó kölcsönösen kompatibilis eseményeknek egy legnagyobb elemszámú halmaza. Például tekintsük azt az S eseményhalmazt, amelynek elemeit a befejezési idejük szerint nemcsökkenő sorrendbe rendeztünk.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(Hamarosan látni fogjuk, hogy miért célszerű így rendezni az eseményeket.) Az $\{a_3, a_9, a_{11}\}$ részhalmaz kölcsönösen kompatibilis eseményeket tartalmaz. Azonban nem maximális, mert az $\{a_1, a_4, a_8, a_{11}\}$ részhalmaz elemszáma nagyobb. Az $\{a_1, a_4, a_8, a_{11}\}$ részhalmaz ténylegesen a legnagyobb elemszámú kölcsönösen kompatibilis események halmaza, és egy másik ilyen legnagyobb elemszámú részhalmaz az $\{a_2, a_4, a_9, a_{11}\}$ halmaz.

Ezt a feladatot több lépésben oldjuk meg. Dinamikus programozási megoldással kezdünk, amelyben két részprobléma optimális megoldását kombináljuk, hogy az eredeti probléma optimális megoldását kapjuk. Sok választási lehetőséget tekintünk, amikor meghatározzuk, hogy mely részproblémákból épül fel az optimális megoldás. Aztán megállapítjuk, hogy csak egy választást kell nézni – a mohó választást –, és amikor a mohó választást tesszük, akkor az egyik részprobléma üres, tehát csak egy nem üres részprobléma marad. Erre az észrevételre alapozva egy rekurzív mohó algoritmust fejlesztünk ki az eseménykiválasztási feladat megoldására. Azzal tesszük teljessé a mohó algoritmus kifejlesztését, hogy a rekurzív algoritmust átalakítjuk iteratív algoritmussá. Azoknak a lépéseknek a sorozata, amelyen keresztülmegyünk ebben az alfejezetben, egy kicsit bonyolultabb annál, mint amit általában alkalmazunk mohó algoritmusok kifejlesztésénél, de jól szemlélteti a dinamikus programozás és a mohó algoritmus viszonyát.

Az eseménykiválasztási probléma optimális részproblémák szerkezete

Mint már mondtuk, eseménykiválasztási feladat dinamikus programozási megoldásával indulunk. Mint a 15. fejezetben, az első lépésünk az, hogy megtaláljuk az optimális szerkezetet, és felépítsük a feladat optimális megoldását a részproblémák optimális megoldásaiból.

A 15. fejezetben már láttuk, hogy részproblémák alkalmas terét kell definiálnunk. Kezdjük azzal, hogy definiáljuk a következő halmazokat.

$$S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\},$$

tehát $S_{i,j}$ azokat az S-beli eseményeket tartalmazza, amelyek a_i befejeződése után kezdődhetnek, és befejeződnek a_j kezdete előtt. Valójában $S_{i,j}$ azokat az eseményeket tartalmazza,

amelyek kompatibilisek mind a_i-vel, mind a_j-vel, és szintén kompatibilisek az összes olyan eseménnyel, amely nem később fejeződik be, mint amikor a_i befejeződik, és azokkal, amelyek a_j kezdeténél nem korábban kezdődnek. A teljes probléma kezeléséhez egészítsük ki az eseményhalmazt az a_0 és a_{n+1} eseményekkel, ahol $f_0 = 0$, $s_{n+1} = \infty$. Ekkor $S = S_{0n+1}$, és a részproblémák indexeinek tartománya: $0 \leq i, j \leq n+1$.

Még tovább szűkíthetjük i és j tartományát a következőképpen. Tegyük fel, hogy az események a befejezésük szerint monoton nemcsökkenő sorrendbe rendezettek.

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}. \tag{16.1}$$

Azt állítjuk, hogy $S_{i,j} = \emptyset$, valahányszor $i \geq j$. Miért? Tegyük fel, hogy van olyan $a_k \in S_{i,j}$ esemény, hogy $i > j$, azaz a_i hátrább van a sorrendben, mint a_j. Ekkor azt kapnánk, hogy $f_i \leq s_k < f_k \leq s_j < f_j$. Tehát $f_i < f_j$ lenne, ami ellentmond azon feltevésünknek, hogy a_i hátrább van a sorrendben, mint a_j. Azt kaptuk, hogy feltételezve, hogy az események a befejezésük szerint monoton nemcsökkenő sorrendbe rendezettek, az $S_{i,j}$, $0 \leq i < j \leq n+1$ részproblémák közül kell maximális elemszámú, kölcsönösen kompatibilis eseményhalmazt kiválasztani, tudva, hogy minden más $S_{i,j}$ halmaz üres.

Az eseménykiválasztási probléma részprobléma szerkezetének meghatározásához tekintsünk egy nem üres $S_{i,j}$ részproblémát[1], és tegyük fel, hogy valamely a_k eleme a megoldásnak, azaz $f_i \leq s_k < f_k \leq s_j$. Az a_k eseményt használva két részproblémát kaphatunk, $S_{i,k}$-t (amely azon események halmaza, amelyek a_i befejezése után kezdődnek, és befejeződnek a_k kezdete előtt) és $S_{k,j}$-t (amely azon események halmaza, amelyek a_k befejezése után kezdődnek, és befejeződnek a_j kezdete előtt). Nyilvánvaló, hogy $S_{i,k}$ és $S_{k,j}$ részhalmaza az $S_{i,j}$ eseményhalmaznak. $S_{i,j}$ megoldását megkapjuk, ha az $S_{i,k}$ és $S_{k,j}$ megoldásának egyesítéséhez hozzávesszük az a_k eseményt. Tehát az $S_{i,j}$ megoldásának elemszámát kapjuk, ha az $S_{i,k}$ megoldásának elemszámához hozzáadjuk $S_{k,j}$ megoldásának elemszámát és még egyet (a_k miatt).

Az optimális részproblémák szerkezet a következő lesz. Tegyük fel, hogy $A_{i,j}$ egy optimális megoldása az $S_{i,j}$ részproblémának és $a_k \in A_{i,j}$. Ekkor az $A_{i,k}$ megoldás optimális megoldása kell legyen az $S_{i,k}$ részproblémának, és az $A_{k,j}$ megoldás optimális megoldása kell legyen az $S_{k,j}$ részproblémának. A szokásos kivágás-beillesztés módszer alkalmazható a bizonyításhoz. Ha lenne olyan $A'_{i,k}$ megoldása $S_{i,k}$-nak, amely több eseményt tartalmazna, mint $A_{i,k}$, akkor $A_{i,j}$-ben $A_{i,k}$ helyett $A'_{i,k}$-t véve $S_{i,j}$-nek egy olyan megoldását kapnánk, amely több eseményt tartalmazna, mint $A_{i,j}$. Mivel feltettük, hogy $A_{i,j}$ optimális, ezért ellentmondásra jutottunk. Hasonlóan, ha lenne olyan $A'_{k,j}$ megoldása $S_{k,j}$-nek, amely több eseményt tartalmazna, mint $A_{k,j}$, akkor $A_{i,j}$-ben $A_{k,j}$ helyett $A'_{k,j}$-t véve $S_{i,j}$-nek egy olyan megoldását kapnánk, amely több eseményt tartalmazna, mint $A_{i,j}$.

Most az optimális részproblémák szerkezet felhasználásával megmutatjuk, hogy az eredeti probléma optimális megoldása felépíthető a részproblémák optimális megoldásaiból. Láttuk, hogy egy nem üres $S_{i,j}$ részprobléma minden megoldása tartalmaz valamely a_k eseményt, és minden optimális megoldás tartalmazza az $S_{i,k}$ és $S_{k,j}$ részproblémák optimális megoldását. Tehát felépíthetjük egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó megoldását az $S_{i,j}$ részproblémának úgy, hogy két részproblémára bontjuk (az $S_{i,k}$ és $S_{k,j}$ részproblémák maximális elemszámú megoldásának megkeresésé-

[1] Az $S_{i,j}$ halmazra néha azt mondjuk, hogy részprobléma, és nem események halmaza. A szövegkörnyezetből mindig világos lesz, hogy ha $S_{i,j}$-re hivatkozunk, akkor mint események halmazát értjük, avagy egy részproblémát, amelynek a bemenete ez a halmaz.

vel), majd megkeressük két részprobléma maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó $A_{i,k}$ és $A_{k,j}$ megoldását, aztán az alábbi formában megalkotjuk a kölcsönösen kompatibilis eseményekből álló $A_{i,j}$ maximális elemszámú megoldást.

$$A_{i,j} = A_{i,k} \cup \{a_k\} \cup A_{k,j}. \tag{16.2}$$

Az eredeti probléma optimális megoldását $S_{0,n+1}$ megoldása adja.

Rekurzív megoldás

A dinamikus programozási megoldás kifejlesztésének második lépéseként rekurzív módon definiáljuk az optimális megoldás értékét. Az eseménykiválasztási probléma esetén legyen $c[i, j]$ az $S_{i,j}$ részprobléma maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmaz elemszáma. Azt tudjuk, hogy $c[i, j] = 0$, ha $S_{i,j} = \emptyset$, és $c[i, j] = 0$, ha $i > j$.

Tekintsünk egy $S_{i,j}$ nem üres részhalmazt. Amint láttuk, ha a_k benne van az $S_{i,j}$ egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazában, akkor az $S_{i,k}$ és $S_{k,j}$ részproblémák egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazait használhatjuk. A (16.2) egyenlőséget felhasználva kapjuk a következő rekurzív összefüggést.

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Ez a rekurzív egyenlet feltételezi, hogy ismerjük a k értéket, de ez nem így van. Összesen $j - i - 1$ lehetséges értéket vehet fel k, nevezetesen $k = i + 1, \ldots, j - 1$. Mivel $S_{i,j}$ a maximális elemszámú részhalmaza valamelyik k-ra előáll, ezért ellenőrizzük az összes lehetséges értékre, hogy a legjobbat kiválasszuk. Tehát $c[i, j]$ teljes rekurzív alakja a következő lesz:

$$c[i, j] = \begin{cases} 0, & \text{ha } S_{i,j} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{i,j}}} \{c[i, k] + c[k, j] + 1\}, & \text{ha } S_{i,j} \neq \emptyset. \end{cases} \tag{16.3}$$

A dinamikus programozási megoldás átalakítása mohó megoldássá

Ezen a ponton egyszerű gyakorlati feladat lehetne táblázatkitöltős, dinamikus programozási algoritmus megírása a (16.3) rekurziós képlet alapján. Valóban, a 16.1-1. gyakorlat ezt kéri. Azonban, van két kulcsfontosságú észrevétel, ami leegyszerűsíti a megoldást.

16.1. tétel. *Tekintsünk egy $S_{i,j}$ nem üres részproblémát, és legyen a_m a legkisebb befejezési idejű esemény $S_{i,j}$-ben.*

$$f_m = \min\{f_k : a_k \in S_{i,j}\}.$$

Ekkor

1. *a_m eleme $S_{i,j}$ valamely maximális elemszámú, kölcsönösen kompatibilis eseményekből álló részhalmazának.*
2. *Az $S_{i,m}$ részprobléma üres, tehát a_m választásával legfeljebb az $S_{m,j}$ nem üres.*

Bizonyítás. Először a második részt bizonyítjuk, mert az egyszerűbb. Tegyük fel, hogy $S_{i,m}$ nem üres, tehát van olyan a_k esemény, hogy $f_i \leq s_k < f_k \leq s_m < f_m$. Mivel a_k eleme $S_{i,j}$-nek, és befejezési ideje kisebb, mint a_m-é, ami ellentmond a_m választásának. Tehát azt kaptuk, hogy $S_{i,m}$ üres.

Az első rész bizonyításához tegyük fel, hogy $A_{i,j}$ egy maximális elemszámú, kölcsönösen kompatibilis eseményekből álló részhalmaza $S_{i,j}$-nek, és tekintsük $S_{i,j}$ elemeinek a befejezési idejük szerinti monoton nemcsökkenő felsorolását. Legyen a_k az első ebben a felsorolásban. Ha $a_k = a_m$, akkor készen vagyunk, mert megmutattuk, hogy a_m eleme $S_{i,j}$ valamely maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazának. Ha $a_k \neq a_m$, akkor tekintsük az $A'_{i,j} = A_{i,j} - \{a_k\} \cup \{a_m\}$ részhalmazt. Az $A'_{i,j}$-beli események diszjunktak, mert $A_{i,j}$ elemei diszjunktak, és a_k a legkorábban befejeződő esemény $A_{i,j}$-ben, továbbá $f_m \leq f_k$. Mivel $A'_{i,j}$ ugyanannyi eseményt tartalmaz, mint $A_{i,j}$, ezért $A'_{i,j}$ is egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmaza $S_{i,j}$-nek, amely tartalmazza a_m-et. ■

Miért fontos a 16.1. tétel? Emlékeztetünk a 15.3. alfejezetre, amely szerint az optimális részproblémák szerkezetét az befolyásolja, hogy hány részproblémától függ az eredeti probléma, és hány választást kell végezni, hogy meghatározzuk, melyik részproblémát kell felhasználni. A dinamikus programozási megoldásunkban két részproblémát használunk az optimális megoldáshoz, és $j - i - 1$ választást kell tenni az $S_{i,j}$ részprobléma megoldásához. A 16.1. tétel jelentősen csökkenti mindkét értéket. Csak egy részprobléma kell az optimális megoldáshoz (a másik biztosan üres), és $S_{i,j}$ megoldása során csak egy választást kell nézni, ami az $S_{i,j}$ legkorábban befejeződő eseménye. Szerencsére könnyen meg tudjuk határozni ezt az eseményt.

Azon túl, hogy csökkentette a részproblémák és a választások számát, a 16.1. tétel más előnnyel is jár. Minden részproblémát felülről lefelé haladó módon meg tudunk oldani, ellentétben a tipikus dinamikus programozási módszerrel, ahol alulról felfelé kell haladni. Az $S_{i,j}$ részprobléma megoldását úgy kapjuk, vesszük $S_{i,j}$ legkorábban befejeződő a_m eseményét, és hozzávesszük az $S_{m,j}$ részprobléma egy optimális megoldásához. Mivel tudjuk, hogy a_m választásával $S_{m,j}$ optimális megoldása biztosan része $S_{i,j}$ egy optimális megoldásának, ezért nem kell megoldani $S_{m,j}$-t $S_{i,j}$ megoldása előtt. $S_{i,j}$ -t úgy oldhatjuk meg, hogy kiválasztjuk a legkorábban befejeződő a_m eseményt $S_{i,j}$-ből, és aztán megoldjuk $S_{m,j}$ -t.

Jegyezzük meg azt is, hogy van séma a megoldandó részproblémákra. Az eredeti probléma az $S = S_{0,n+1}$. Tegyük fel, hogy az a_{m_1} eseményt választottuk, amely a legkorábban befejeződő eseménye $S_{0,n+1}$-nek. (Mivel az események befejezési idejük szerint monoton nemcsökkenő sorrendbe rendezettek, és $f_0 = 0$, így $m_1 = 1$.) A következő részproblémánk $S_{m_1,n+1}$ lesz. Tegyük fel, hogy a_{m_2}-t választottuk $S_{m_1,n+1}$-ből, amely a legkorábban befejeződő eseménye. (Nem feltétlenül teljesül, hogy $m_2 = 2$.) A következő részproblémánk $S_{m_2,n+1}$ lesz. Ezt folytatva látjuk, hogy minden részproblémánk $S_{m_i,n+1}$ alakú lesz, valamely m_i eseménysorszámra. Más szóval, minden részproblémát a legkésőbb befejeződő esemény, és egy másik esemény sorszáma határoz meg, ahol az utóbbi részproblémáról részproblémára változik.

A választandó eseményre is van sémánk. Mivel mindig $S_{m_i,n+1}$ -nek a legkorábban befejeződő eseményét választjuk, így a részproblémákhoz kiválasztott események sorozata a befejezési idő szerint szigorúan monoton növekvő lesz. Továbbá, minden eseményt csak egyszer kell vizsgálni, a befejezési idejük szerint monoton nemcsökkenő sorrendben.

Egy részprobléma megoldásához mindig azt az a_m eseményt választjuk ki, amely a legkorábban befejeződik, és legálisan beoszthető. Tehát a választás „mohó” abban az értelemben, hogy intuitíve a legnagyobb lehetőséget hagyja a fennmaradt események ütemezésére. Tehát az a mohó választás, amely maximalizálja az ütemezésre fennmaradt időt.

Rekurzív mohó algoritmus

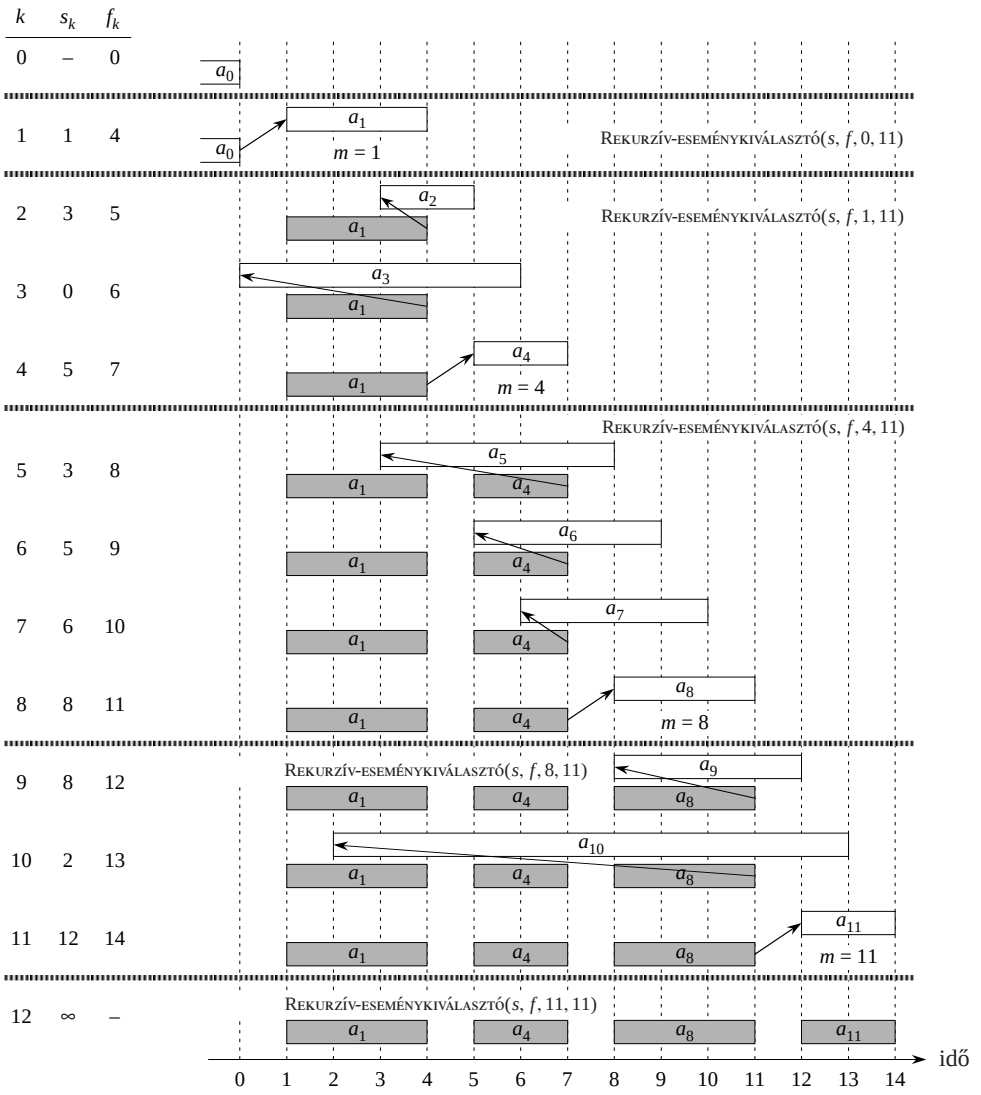
Miután láttuk, hogyan adhatunk dinamikus programozási megoldást, amely felülről lefelé haladó módszer, itt az ideje, hogy megadjunk egy tisztán mohó, alulról felfelé haladó módszerű algoritmust. A REKURZÍV-ESEMÉNYKIVÁLASZTÓ eljárás közvetlenül kapható rekurzív megoldása a problémának. Ennek bemenő paraméterei az események kezdő és befejező időpontjait tartalmazó s és f tömb, továbbá a megoldandó $S_{i,n+1}$ részproblémát meghatározó i és n sorszám. (Az n paraméter az utolsó a_n esemény indexe, és nem az $n + 1$ fiktív esemény, amely szintén eleme a részproblémának.) Az eljárás $S_{i,n+1}$ egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazát adja eredményül. Feltételezzük, hogy az n bemeneti esemény befejezési idő szerint monoton nemcsökkenő sorrendbe rendezett a 16.1. képletnek megfelelően. Ha a rendezettség nem teljesülne, akkor $O(n \log n)$ időben rendezhetjük őket. A kiindulási probléma megoldását a REKURZÍV-ESEMÉNYKIVÁLASZTÓ$(s, f, 0, n)$ eljáráshívás adja.

```
REKURZÍV-ESEMÉNYKIVÁLASZTÓ(s, f, i, n)

1  m ← i + 1
2  while m ≤ n és s_m < f_i                ▷ S_{i,n+1} első választható eseményét keressük
3      do m ← m + 1
4  if m ≤ n
5     then return {a_m} ∪ REKURZÍV-ESEMÉNYKIVÁLASZTÓ(s, f, m, n)
6     else return ∅
```

A 16.1. ábra mutatja az algoritmus által végzett műveleteket. A REKURZÍV-ESEMÉNYKIVÁLASZTÓ(s, f, m, n) egy adott meghívásakor a 2–3. sorokban a **while** ciklus megkeresi az $S_{i,n+1}$ első választható eseményét. A ciklus sorban az $a_{i+1}, a_{i+2}, \ldots, a_n$ eseményeket vizsgálja, amíg meg nem találja az első olyan a_m eseményt, amely kompatibilis a_i-vel, azaz $s_m \geq f_i$ teljesül. Ha a ciklus úgy ér véget, hogy talált ilyen eseményt, akkor az eljáráshívással befejeződik az 5. sorban végrehajtott **return** utasítással, ami visszaadja az $\{a_m\}$ és a REKURZÍV-ESEMÉNYKIVÁLASZTÓ(s, f, m, n) rekurzív hívás által visszaadott halmazok egyesítését. Az utóbbi halmaz az $S_{m,n+1}$ részprobléma megoldása. A ciklus úgy is befejeződhet, hogy az $m > n$ feltétel teljesül, amikor is nincs olyan esemény, amely kompatibilis lenne s_i-vel. Ebben az esetben $S_{i,n+1} = \emptyset$, és az eljárás az $\emptyset$ értéket adja vissza a 6. sorban.

Feltéve, hogy az események befejezési idejük szerint monoton nemcsökkenően rendezettek, a REKURZÍV-ESEMÉNYKIVÁLASZTÓ$(s, f, 0, n)$ eljáráshívás futási ideje $\Theta(n)$. Ezt a következőképpen láthatjuk be. A rekurzív hívásokban minden eseményt pontosan egyszer vizsgálunk a **while** ciklus feltételvizsgálatakor a 2. sorban. Pontosabban, az a_k eseményt az utolsó olyan hívás vizsgálja, amelyre $i < k$.



16.1. ábra. A Rekurzív-eseménykiválasztó algoritmus működése a korábban megadott 11 eseményre. Egy rekurzív hívás során vizsgált események két horizontális vonal között láthatóak. A fiktív a_0 esemény befejezési ideje 0, az első Rekurzív-eseménykiválasztó$(s, f, 0, 11)$ eljáráshíváskor az a_1 esemény választódik ki. A már korábban kiválasztott események satírozottak, az éppen vizsgált esemény pedig fehér. Ha egy esemény kezdő időpontja előbb van, mint a legutoljára beválasztott esemény befejező időpontja (a közöttük meghúzott nyíl balra mutat), akkor azt elvetjük. Egyébként (ha a nyíl egyenesen felfelé vagy jobbra mutat) beválasztjuk. Az utolsó Rekurzív-eseménykiválasztó$(s, f, 11, 11)$ rekurzív hívás a $\emptyset$ értékkel tér vissza. Az eredményül kapjuk a kiválasztott események $\{a_1, a_4, a_8, a_{11}\}$ halmazát.

Iteratív mohó algoritmus

A rekurzív eljárásunkat egyszerűen átalakíthatjuk iteratív algoritmussá. A Rekurzív-eseménykiválasztó eljárás majdnem jobb-rekurzív (lásd a 7-4. feladatot), önmagát hívó rekurzív hívással végződik, amit követ egy egyesítés művelet. Jobb-rekurzív eljárás átalakí-

tása iteratívvá általában egyszerű feladat, valójában több programozási nyelv fordítóprogramja ezt automatikusan elvégzi. Amint látjuk, a REKURZÍV-ESEMÉNYKIVÁLASZTÓ eljárás minden $S_{i,n+1}$ részproblémára működik, tehát azokra, amelyek a legnagyobb befejezésű eseményeket tartalmazzák.

A MOHÓ-ESEMÉNYKIVÁLASZTÓ eljárás egy iteratív változata a REKURZÍV-ESEMÉNYKIVÁLASZTÓ eljárásnak. Ez ismét feltételezi, hogy a bemeneti események befejezési idejük szerint monoton nemcsökkenő sorrendbe rendezettek. Az eljárás az A halmazban gyűjti össze a kiválasztott eseményeket, és ezt a halmazt adja eredményül a végén.

```
MOHÓ-ESEMÉNYKIVÁLASZTÓ(s, f)
1  n ← hossz[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5      do if s_m ≥ f_i
6            then A ← A ∪ {a_m}
7                 i ← m
8  return A
```

Az eljárás a következőképpen működik. Az i változó tartalmazza az A-ba legutoljára beválasztott esemény indexét, aminek az a_i esemény felel meg a rekurzív változatban. Mivel az eseményeket befejezési idejük szerinti monoton nemcsökkenő sorrendben vizsgáljuk, ezért f_i mindig a legnagyobb befejezési idejű esemény az A halmazban. Tehát

$$f_i = \max\{f_k : a_k \in A\} . \tag{16.4}$$

A 2–3. sorban kiválasztjuk az a_1 eseményt, előkészítve ezzel az A halmazt, hogy egyedül az a_1 eseményt tartalmazza, az i változó pedig ezen esemény sorszámát veszi fel kezdetben. A ***for*** ciklus a 4–7. sorokban megkeresi a legkorábban befejeződő eseményt az $S_{i,n+1}$ halmazban. A ciklus egymás után vizsgálja az a_m eseményeket, és hozzáadja az A halmazhoz, ha kompatibilis az összes A-beli eseménnyel. Annak ellenőrzése, hogy a_m kompatibilis az összes A-ban lévő eseménnyel, a 16.4. egyenlőség miatt elegendő azt ellenőrizni (5. sor), hogy az s_m kezdő időpont nem korábbi, mint az A-ba legutoljára beválasztott esemény f_i befejező időpontja. Ha az a_m esemény kompatibilis, akkor a 6–7. sorokban hozzávesszük a_m-et A-hoz, és i felveszi az m értéket. A MOHÓ-ESEMÉNYKIVÁLASZTÓ(s, f) eljáráshívás pontosan azt a halmazt adja, mint a REKURZÍV-ESEMÉNYKIVÁLASZTÓ$(s, f, 0, n)$ hívás.

A MOHÓ-ESEMÉNYKIVÁLASZTÓ algoritmus, a REKURZÍV-ESEMÉNYKIVÁLASZTÓ algoritmushoz hasonlóan, $\Theta(n)$ időben megoldja n bemeneti eseményre a feladatot, feltéve, hogy az események kezdetben a befejezési idejük szerint monoton nemcsökkenő sorrendben vannak.

Gyakorlatok

16.1-1. Adjunk dinamikus programozási algoritmust az eseménykiválasztási probléma megoldására, felhasználva a (16.3) rekurzív egyenletet. Az algoritmus számítsa ki a fent definiált $c[i, j]$ elemszám értékeket és adja meg az események egy maximális elemszámú A halmazát is. Tegyük fel, hogy a bemenet a 16.1. egyenlőtlenségeket teljesítő módon van rendezve. Hasonlítsuk össze az algoritmus futási idejét a MOHÓ-ESEMÉNYKIVÁLASZTÓ algoritmus futási idejével.

16.1-2. Tegyük fel, hogy nem a legkorábban befejező, hanem mindig a legkésőbb kezdődő eseményt választjuk ki, amely kompatibilis minden eddig beválasztott eseménnyel. Mutassuk meg, hogyan lesz ez mohó algoritmus, és bizonyítsuk be, hogy optimális megoldást ad.

16.1-3. Tegyük fel, hogy eseményeket kell ütemezni nagyszámú előadóterembe. Az összes esemény egy olyan ütemezését keressük, amely a lehető legkevesebb termet igényli. Készítsünk olyan hatékony mohó algoritmust, amely meghatározza, hogy az egyes eseményeket melyik terembe kell beosztani.

(Ez a probléma úgy is ismert, mint ***intervallumgráf-színezési probléma.*** Megalkothatjuk azt az intervallum-gráfot, amelynek csúcsai az adott események, élei pedig a nem kompatibilis eseményeket kötik össze. Színezzük ki a gráfot a legkevesebb színnel úgy, hogy bármely két olyan csúcs színe, amelyeket él köt össze, különböző legyen. Az így kapott színek száma megegyezik a legkevesebb előadóterem számával, amely szükséges az események ütemezéséhez.)

16.1-4. Nem minden mohó stratégia szolgáltat maximális elemszámú olyan halmazt, melynek elemei kölcsönösen kompatibilisek. Adjunk olyan példát, amely megmutatja, hogy az a mohó választás, amely mindig a legrövidebb ideig tartó olyan eseményt választja ki, amely kompatibilis az addig kiválasztott eseményekkel, nem ad optimális megoldást. Hasonló módon mutassuk meg, hogy az a stratégia, amely mindig azt az eseményt választja, amelynek a legkevesebb még megmaradt eseménnyel van átfedése, szintén nem ad optimális megoldást.

16.2. A mohó stratégia elemei

A mohó algoritmus úgy alkotja meg a probléma optimális megoldását, hogy választások sorozatát hajtja végre. Az algoritmus során minden döntési pontban azt az esetet választja, amely az adott pillanatban optimálisnak látszik. Ez a heurisztikus stratégia nem mindig ad optimális megoldást, azonban néha igen, mint azt láttuk az eseménykiválasztási probléma esetén. Ebben a szakaszban a mohó stratégia néhány általános tulajdonságát fogjuk megvizsgálni.

Az a módszer, amit a 16.1. alfejezetben követtünk mohó algoritmus kifejlesztésére, egy kicsit bonyolultabb az általános esetnél. A következő lépések sorozatán mentünk keresztül.

1. A probléma optimális szerkezetének meghatározása.
2. Rekurzív megoldás kifejlesztése.
3. Annak bizonyítása, hogy minden rekurzív lépésben az egyik optimális választás a mohó választás. Tehát mindig biztonságos a mohó választás.
4. Annak igazolása, hogy a mohó választás olyan részproblémákat eredményez, amelyek közül legfeljebb az egyik nem üres.
5. A mohó stratégiát megvalósító rekurzív algoritmus kifejlesztése.
6. A rekurzív algoritmus átalakítása iteratív algoritmussá.

Ezen lépéseken keresztülhaladva láttuk a mohó algoritmus dinamikus programozási alátámasztását. A gyakorlatban azonban általában egyszerűsítjük a fenti lépéseket mohó algoritmus tervezésekor. A részproblémák kifejlesztésekor arra figyelünk, hogy a mohó válasz-

tás egyetlen részproblémát eredményezzen, amelynek optimális megoldását kell megadni. Például az eseménykiválasztási feladatnál először olyan $S_{i,j}$ részproblémákat határoztunk meg, ahol i és j is változó érték lehetett. Ezután rájöttünk, hogy ha mindig mohó választást végzünk, akkor redukálhatjuk a részproblémákat $S_{i,n+1}$ alakúakra.

Másképpen kifejezve, az optimális részproblémák szerkezetét a mohó választás figyelembevételével alakíthattuk ki. Tehát elhagyhattuk a második indexet, és az $S_i = \{a_k \in S : f_i \leq s_k\}$ alakú részproblémákhoz jutottunk. Ezután bebizonyíthattuk, hogy a mohó választás (az első befejeződő a_m esemény S_i-ben), kombinálva S_m egy optimális megoldásával, az eredeti S_i probléma optimális megoldását adja. Általánosabban, mohó algoritmus tervezését az alábbi lépések végrehajtásával végezzük.

1. Fogalmazzuk meg az optimalizációs feladatot úgy, hogy minden egyes választás hatására egy megoldandó részprobléma keletkezzék.
2. Bizonyítsuk be, hogy mindig van olyan optimális megoldása az eredeti problémának, amely tartalmazza a mohó választást, tehát a mohó választás mindig biztonságos.
3. Mutassuk meg, hogy a mohó választással olyan részprobléma keletkezik, amelynek egy optimális megoldásához hozzávéve a mohó választást, az eredeti probléma egy optimális megoldását kapjuk.

Ezt a közvetlenebb módszert alkalmazzuk a fejezet hátralévő részében. Mindazonáltal, minden mohó algoritmushoz majdnem mindig van bonyolultabb dinamikus programozási megoldás.

Meg tudjuk-e mondani, hogy adott optimalizációs feladatnak van-e mohó algoritmusú megoldása? Erre nem tudunk általános választ adni, de a mohó-választási tulajdonság és az optimális részproblémák tulajdonság két kulcsfontosságú összetevő. Ha meg tudjuk mutatni, hogy a feladat rendelkezik e két tulajdonsággal, nagy eséllyel ki tudunk fejleszteni mohó algoritmusú megoldást.

Mohó-választási tulajdonság

Az első alkotóelem a ***mohó-választási tulajdonság:*** globális optimális megoldás elérhető lokális optimum (mohó) választásával. Más szóval, amikor arról döntünk, hogy melyik választást tegyük, azt választjuk, amelyik az adott pillanatban a legjobbnak tűnik, nem törődve a részproblémák megoldásaival.

Ez az a pont, ahol a mohó stratégia különbözik a dinamikus programozástól. Dinamikus programozás esetén minden lépésben választást hajtunk végre, de a választás függhet a részproblémák megoldásától. Következésképpen, a dinamikus programozási módszerrel a problémát alulról felfelé haladó módon oldjuk meg, egyszerűbbtől összetettebb részproblémák felé haladva. A mohó algoritmus során az adott pillanatban legjobbnak tűnő választást hajtjuk végre, bármi is legyen az, és azután oldjuk meg a választás hatására fellépő részproblémát. A mohó algoritmus során végrehajtott választás függhet az addig elvégzett választásoktól, de nem függhet későbbi választásoktól vagy részproblémák megoldásától. Tehát ellentétben a dinamikus programozással, amely a részproblémákat alulról felfelé haladva oldja meg, a mohó stratégia általában felülről lefelé halad, egymás után végrehajtva mohó választásokat, amellyel a problémát sorra kisebb méretűre redukálja.

Természetesen bizonyítanunk kell, hogy a lépésenkénti mohó választásokkal globálisan optimális megoldáshoz jutunk, és ez az, ami leleményességet igényel. Tipikusan, mint

a 16.1. tétel esetén, a bizonyítás részproblémák globális optimális megoldását vizsgálja. Megmutatja, hogy az optimális megoldás módosítható úgy, hogy az a mohó választást tartalmazza, és hogy ez a választás redukálja a problémát hasonló, de kisebb méretű részproblémára.

A mohó-választási tulajdonság gyakran hatékonyságot eredményez a részprobléma választásával. Például az eseménykiválasztási feladatnál, feltételezve, hogy az események befejezési idejük szerint monoton nemcsökkenő sorrendbe rendezettek, minden eseményt csak egyszer kell vizsgálni. Gyakran az a helyzet, hogy a bemeneti adatokat alkalmasan előfeldolgozva, vagy alkalmas adatszerkezetet használva (ami gyakran prioritási sor), a mohó választás gyorsan elvégezhető, és ezáltal hatékony algoritmust kapunk.

Optimális részproblémák tulajdonság

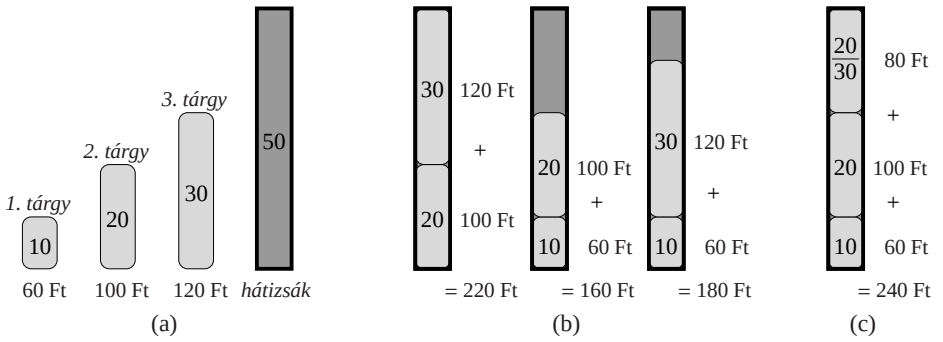
Egy probléma teljesíti az ***optimális részproblémák tulajdonságot,*** ha az optimális megoldás felépíthető a részproblémák optimális megoldásából. Ez az alkotóelem kulcsfontosságú mind a dinamikus programozás, mind a mohó stratégia alkalmazhatóságának megállapításánál. Az optimális részproblémákra példaként emlékeztetünk arra, ahogy a 16.1. alfejezetben megmutattuk, hogy ha $S_{i,j}$ egy optimális megoldása tartalmazza az a_k eseményt, akkor az szükségképpen tartalmazza $S_{i,k}$ és $S_{k,j}$ egy optimális megoldását. Ezen optimális szerkezet alapján, ha tudjuk, hogy melyik a_k eseményt kell választani, akkor $S_{i,j}$ egy optimális megoldása megalkotható a_k, továbbá $S_{i,k}$ és $S_{k,j}$ egy optimális megoldásából. Az optimális részproblémák ezen tulajdonságát észre véve meg tudtuk adni a 16.3. rekurzív egyenletet, ami az optimális megoldás értékét adja meg.

Általában sokkal közvetlenebb alkalmazását használjuk az optimális részproblémák tulajdonságnak mohó algoritmus kifejlesztése során. Mint már említettük, szerencsénk van, amikor feltételezzük, hogy az eredeti probléma mohó választása megfelelő részproblémát eredményez. Csak azt kell belátni, hogy a részprobléma optimális megoldása, kombinálva a már elvégzett mohó választással, az eredeti probléma optimális megoldását adja. Ez a séma implicit módon használ részproblémák szerinti indukciót annak bizonyítására, hogy minden lépésben mohó választást végezve optimális megoldást kapunk.

Mohó stratégia vagy dinamikus programozás

Mivel az optimális részproblémák tulajdonságot kihasználjuk mind a mohó, mind a dinamikus programozási stratégiáknál, előfordulhat, hogy dinamikus programozási megoldást próbálunk adni akkor, amikor mohó megoldás is célravezető lenne, és fordítva, tévesen mohó megoldással próbálkozunk akkor, amikor valójában dinamikus programozási módszert kellene alkalmazni. A finom különbségek illusztrálására tekintsük a következő klasszikus optimalizálási probléma két változatát.

A 0-1 ***hátizsák feladat*** a következőt jelenti. Adott n darab tárgy, az i-edik tárgy használati értéke v_i, a súlya pedig w_i, ahol v_i és w_i egész számok. Kiválasztandó a tárgyaknak olyan részhalmaza, amelyek használati értékének összege a lehető legnagyobb, de a súlyuk összege nem nagyobb, mint a hátizsák W kapacitása, amely egész szám. Mely tárgyakat rakjuk a hátizsákba? (Ezt a problémát azért nevezzük 0-1 hátizsák feladatnak, mert minden tárgyat vagy beválasztunk, vagy elhagyunk, nem tehetjük meg, hogy egy tárgy töredékét, vagy többszörösét választjuk.)



16.2. ábra. A mohó stratégia nem működik a 0-1 hátizsák feladatra. **(a)** Az ábrán szereplő három tárgyat tartalmazó halmaz olyan részhalmazát kell kiválasztani, hogy a részhalmaz tárgyainak összsúlya legfeljebb 50 legyen. **(b)** Az optimális megoldás a 2. és 3. tárgyakat tartalmazza. Minden olyan megoldás, amely tartalmazza a 1. tárgyat, annak ellenére, hogy ennek a legnagyobb az érték per súly hányadosa, nem lesz optimális. **(c)** A töredékes hátizsák feladat esetén az a stratégia, amely szerint a legnagyobb érték per súly szerint választunk, optimális megoldást ad.

A ***töredékes hátizsák feladat*** csak abban különbözik az előzőtől, hogy a tárgyak töredéke is választható, nem kell 0-1 bináris választást tenni. Úgy tekinthetjük, hogy 0-1 hátizsák feladat esetén a tárgyak aranytömbök, míg a töredékes hátizsák feladatnál aranyporból meríthetünk.

Mindkét hátizsák feladat teljesíti az optimális részproblémák tulajdonságot . A 0-1 feladat esetén tekintsünk egy olyan választást, amely a legnagyobb használati értéket adja, de a tárgyak összsúlya nem haladja meg a W értéket. Ha kivesszük a j-edik tárgyat a hátizsákból, akkor a bennmaradt tárgyak használati értéke a legnagyobb lesz azon feltétel mellett, hogy az összsúly nem nagyobb, mint $W - w_j$, és $n - 1$ tárgyból választhatunk, kizárva az eredeti tárgyak közül a j-ediket. Ha a töredékes hátizsák feladatnál egy optimális választásból kiveszünk a j tárgyból w mennyiséget, akkor a megmaradt választás optimális lesz arra az esetre, amikor legfeljebb $W - w$ összsúlyt érhetünk el, és a j-edik tárgyból legfeljebb $w_j - w$ mennyiséget választhatunk.

Bár a két feladat hasonló, a töredékes hátizsák feladat megoldható mohó stratégiával, a 0-1 feladat azonban nem. A töredékes feladat megoldásához előbb számítsuk ki minden tárgyra a v_i/w_i használati érték per súly hányadost. A mohó stratégiát követve először a legnagyobb hányadosú tárgyból választunk amennyit csak lehet. Ha elfogyott, de még nem telt meg a hátizsák, akkor a következő legnagyobb hányadosú tárgyból választunk amennyit csak lehet, és így tovább, amíg a hátizsák meg nem telik. Mivel a tárgyakat az érték per súly hányados szerint kell rendeznie, a mohó algoritmus futási ideje $O(n \lg n)$ lesz. Annak bizonyítása, hogy a töredékes hátizsák feladat teljesíti a mohó-választási tulajdonságot, a 16.2-1 gyakorlat tárgya.

Annak bemutatására, hogy a mohó stratégia nem működik a 0-1 hátizsák feladatra, tekintsük a 16.2(a) ábrán látható esetet. Három tárgyunk van, és a hátizsák mérete 50 egységnyi. Az 1. tárgy súlya 10, használati értéke 60, a 2. tárgy súlya 20, használati értéke 100, a 3. tárgy súlya 30, használati értéke pedig 120 egység. Tehát az 1. tárgy érték per súly hányadosa 6, a 2. tárgyé 2, a 3. tárgyé pedig 4. Így a mohó stratégia először az 1. tárgyat választaná. Azonban a 16.2(b) ábrán látható elemzés szerint az optimális megoldásban a 2. és a 3. tárgy szerepel, kihagyva az 1. tárgyat. Mindkét választás, amelyben az 1. tárgy szerepel, nem optimális.

A megfelelő töredékes feladatra azonban a mohó stratégia, amely először az 1. tárgyat választja, optimális megoldást ad, mint azt a 16.2(c) ábra mutatja. A 0-1 feladat esetén az 1. tárgy választása nem vezet optimális megoldáshoz, mert ezután nem tudjuk telerakni a hátizsákot, és az üresen maradt rész csökkenti a hátizsák lehetséges érték per súly hányadost. A 0-1 feladatnál amikor egy tárgy beválasztásáról döntünk, akkor előbb össze kell hasonlítani annak a két részproblémának a megoldását, amely a tárgy beválasztásával, illetve kihagyásával adódik. Az így megfogalmazott probléma sok, egymást átfedő részproblémát eredményez, ami a dinamikus programozást fémjelzi. Valóban, a 0-1 feladat megoldható dinamikus programozási módszerrel (lásd a 16.2-2. gyakorlatot).

Gyakorlatok

16.2-1. Mutassuk meg, hogy a töredékes hátizsák feladat teljesíti a mohó-választási tulajdonságot.

16.2-2. Adjunk olyan dinamikus programozási megoldást a 0-1 hátizsák feladatra, amelynek futási ideje $O(n\,W)$, ahol n a tárgyak száma, W pedig a hátizsák kapacitása.

16.2-3. Tegyük fel, hogy a 0-1 hátizsák feladat esetén a tárgyak növekvő súly szerinti sorrendje megegyezik a csökkenő érték szerinti sorrendjükkel. Adjunk hatékony algoritmust ezen változat megoldására, és igazoljuk az algoritmus helyességét.

16.2-4. Midas professzor autóval utazik Newark városából Renóba. Ha autójának üzemanyagtartálya tele van, akkor n mérföldet tud megtenni. Van olyan térképe, amely az útvonalon található benzinkutak egymástól mért távolságát is tartalmazza. A professzor a lehető legkevesebbszer akar megállni üzemanyag-felvétel miatt. Adjunk hatékony módszert annak meghatározására, hogy a professzornak mely benzinkutaknál kell megállnia, és bizonyítsuk be, hogy a stratégia optimális megoldást ad.

16.2-5. Adott a számegyenesen pontoknak egy $\{x_1, x_2, \ldots, x_n\}$ halmaza. Adjunk olyan hatékony algoritmust, amely megad egy minimális számú, egységnyi hosszú zárt intervallumokból álló halmazt, amelyek tartalmazzák a pontokat. Indokoljuk meg az algoritmusa helyességét.

16.2-6.$\star$ Mutassuk meg, hogyan lehet a töredékes hátizsák feladatot megoldani $O(n)$ időben.

16.2-7. Tegyük fel, hogy adott egész számoknak két, A és B halmaza, és mindegyik n elemet tartalmaz. A két halmaz elemei tetszőleges sorrendbe rakhatóak. Legyen a_i az A halmaz i-edik eleme a felsorolásban, és legyen b_j B halmaz j-edik eleme a felsorolásban. Ekkor a nyeremény $\prod_{i=1}^{n} a_i^{b_i}$. Adjunk olyan algoritmust, amely kiszámítja a lehető legnagyobb nyeremény értékét. Bizonyítsuk be, hogy az algoritmus maximális nyereményt ad, és adjuk meg az algoritmus futási idejét.

16.3. Huffman-kód

A Huffman-kód széles körben használt és nagyon hatékony módszer adatállományok tömörítésére. Az elérhető megtakarítás 20%-tól 90%-ig terjedhet, a tömörítendő adatállomány sajátosságainak függvényében. A kódolandó adatállományt karaktersorozatnak tekintjük. A Huffman-féle mohó algoritmus egy táblázatot használ az egyes karakterek előfordulási gyakoriságára, hogy meghatározza, hogyan lehet a karaktereket optimálisan ábrázolni bináris jelsorozattal.

	a	b	c	d	e	f
Gyakoriság (ezrekben)	45	13	12	16	9	5
Fix hosszú kódszó	000	001	010	011	100	101
Változó hosszú kódszó	0	101	100	111	1101	1100

16.3. ábra. Karakterkódolási probléma. Az adatállomány 100 000 karakterből áll, és csak az a–f karakterek fordulnak elő az állományban a feltüntetett gyakoriságokkal. Ha minden karaktert 3 bites kódszóval kódolunk, akkor 300 000 bitre van szükség. Az ábrán látható változó hosszú kódszavakat használva az állományt 224 000 bittel kódolhatjuk.

Tegyük fel, hogy egy 100 000 karaktert tartalmazó adatállományt akarunk tömörítetten tárolni. Tudjuk, hogy az egyes karakterek előfordulási gyakorisága megfelel a 16.3. ábrán látható táblázatnak. Vagyis, hat különböző karakter fordul elő az állományban, és az a karakter 45 000-szer fordul elő az állományban.

Sokféleképpen ábrázolható egy ilyen típusú információhalmaz. Mi ***bináris karakterkód*** (vagy röviden ***kód***) tervezésének problémáját vizsgáljuk, amikor is minden karaktert egy bináris jelsorozattal ábrázolunk. Ha ***fix hosszú kódot*** használunk, akkor 3 bitre van szükség a hatféle karakter kódolására: $a = 000, b = 001, \ldots, f = 101$. Ez a módszer 300 000 bitet igényel a teljes állomány kódolására. Csinálhatjuk jobban is? A ***változó hosszú kód*** alkalmazása tekintélyes megtakarítást eredményez, ha gyakori karaktereknek rövid, ritkán előforduló karaktereknek hosszabb kódszavat feleltetünk meg. A 16.3. ábra egy ilyen kódolást mutat: itt az egybites 0 kód az a karaktert ábrázolja, a négybites 1100 kód pedig az f karakter kódja. Ez a kódolás

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,000$$

bitet igényel az állomány tárolására, ami hozzávetőleg 25% megtakarítást eredményez. Valójában ez optimális kódolást jelent, mint majd látni fogjuk.

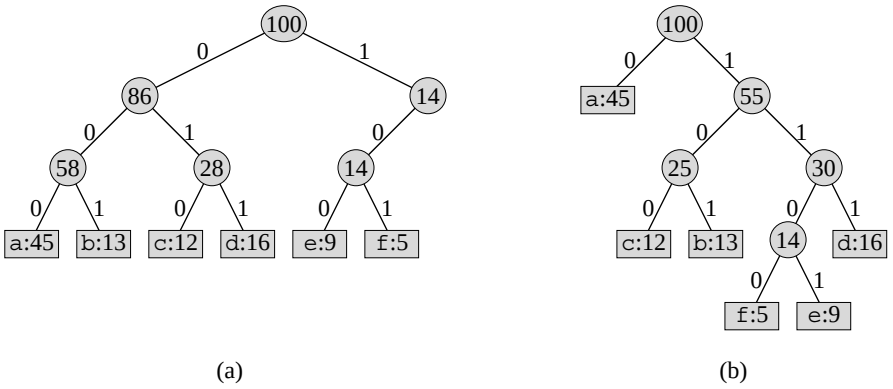
Prefix-kódok

A továbbiakban csak olyan kódszavakat tekintünk, amelyekre igaz, hogy egyik sem kezdőszelete a másiknak. Az ilyen kódolást ***prefix-kódnak*** nevezzük.[2] Megmutatható (bár mi ezt nem tesszük meg), hogy karakterkóddal elérhető optimális adattömörítés mindig megadható prefix-kóddal is, így az általánosság megszorítása nélkül elegendő prefix-kódokat tekinteni.

A prefix-kódok előnyösek, mert egyszerűsítik a kódolást (tömörítést) és a dekódolást. A kódolás minden bináris karakterkódra egyszerű: csak egymás után kell írni az egyes karakterek bináris kódját. Például a 16.3. ábrán adott változó hosszú karakterkód esetén az *abc* három karaktert tartalmazó állomány kódja $0 \cdot 101 \cdot 100 = 0101100$, ahol a „$\cdot$” pont az egymásután írás művelet (konkatenáció) jele.

A dekódolás is meglehetősen egyszerű prefix-kód esetén. Mivel nincs olyan kódszó, amely kezdőszelete lenne egy másiknak, így egyértelmű, hogy a kódolt állomány melyik kódszóval kezdődik. Egyszerűen megállapítjuk, hogy a kódolt állomány melyik kódszóval kezdődik, aztán helyettesítjük ezt azzal a karakterrel, amelynek ez a kódja, és ezt az eljárást addig végezzük, amíg a kódolt állományon végig nem értünk. Példánkat tekintve,

[2] A „prefix-mentes” elnevezés helyesebb lenne, de a „prefix-kód” általánosan használt az irodalomban.



(a) (b)

16.4. ábra. A 16.3. ábrán adott kódolásokhoz tartozó bináris fák. Minden levél címkeként tartalmazza a kódolandó karaktert és annak előfordulási gyakoriságát. A belső csúcsok az adott gyökerű részfában található gyakoriságok összegét tartalmazzák. **(a)** A fix hosszú kódhoz tartozó fa; $a = 000, \ldots, f = 101$. **(b)** Az optimális prefix-kódhoz tartozó fa; $a = 0, b = 101, \ldots, f = 1100$.

a 001011101 jelsorozat egyértelműen bontható fel a $0 \cdot 0 \cdot 101 \cdot 1101$ kódszavak sorozatára, tehát a dekódolás az *aabe* sorozatot eredményezi.

A dekódolási eljáráshoz szükség van a prefix-kód olyan alkalmas ábrázolására, amely lehetővé teszi, hogy a kódszót könnyen azonosítani tudjuk. Az olyan bináris fa, amelynek levelei a kódolandó karakterek, egy ilyen alkalmas ábrázolás. Ekkor egy karakter kódját a fa gyökerétől az adott karakterig vezető út ábrázolja, a 0 azt jelenti, hogy balra megyünk, az 1 pedig, hogy jobbra megyünk az úton a fában. A 16.4. ábra a példánkban szereplő két kódot ábrázolja. Vegyük észre, hogy ezek a fák nem bináris keresőfák, a levelek nem rendezetten találhatók, a belső csúcsok pedig nem tartalmaznak karakterkulcsokat.

Egy adatállomány optimális kódját mindig *teljes* bináris fa ábrázolja, tehát olyan fa, amelyben minden nem levél csúcsnak két gyereke van (lásd a 16.3-1. gyakorlatot). A példánkban szereplő fix hosszú kód nem optimális, mert a 16.4. ábrán látható fája nem teljes bináris fa: van olyan kódszó, amely 10-zel kezdődik, de nincs olyan, amely 11-gyel kezdődne. Mivel a továbbiakban szorítkozhatunk teljes bináris fákra, azt mondhatjuk, hogy ha C az az ábécé, amelynek elemei a kódolandó karakterek, akkor az optimális prefix-kód fájának pontosan $|C|$ levele és pontosan $|C| - 1$ belső csúcsa van. (Lásd a B.5-3. gyakorlatot.)

Ha adott egy prefix-kód T fája, akkor egyszerű kiszámítani, hogy az adatállomány kódolásához hány bit szükséges. A C ábécé minden c karakterére jelölje $f(c)$ a c karakter előfordulási gyakoriságát az állományban, $d_T(c)$ pedig jelölje a c-t tartalmazó levél mélységét a T fában. Vegyük észre, hogy $d_T(c)$ megegyezik a c karakter kódjának hosszával. A kódoláshoz szükséges bitek száma ekkor

$$B(T) = \sum_{c \in C} f(c) d_T(c), \tag{16.5}$$

és ezt az értéket a T fa ***költségének*** nevezzük.

Huffman-kód szerkesztése

Huffman találta ki azt a mohó algoritmust, amely optimális prefix-kódot készít, amit ***Huffman-kódnak*** nevezünk. A 16.2. szakasz megállapításait figyelembe véve az algoritmus helyességének bizonyítása a mohó-választási és az optimális részproblémák tulajdonságon alapszik. Ahelyett, hogy a kód kifejlesztése előtt bebizonyítanánk e két tulajdonság teljesülését, először a kódot adjuk meg. Ezt azért tesszük, hogy világosan lássuk, az algoritmus hogyan használja a mohó választást.

A következő, pszeudokód formájában adott algoritmusban feltételezzük, hogy C a karakterek n elemű halmaza, és minden $c \in C$ karakterhez adott annak $f[c]$ gyakorisága. Az algoritmus alulról felfelé haladva építi fel azt a T fát, amely az optimális kód fája. Az algoritmus úgy indul, hogy kezdetben $|C|$ számú csúcs van, amelyek mindegyike levél, majd $|C| - 1$ számú „összevonás" végrehajtásával alakítja ki a végső fát. Az f-szerint kulcsolt Q prioritási sort használjuk az összevonandó két legkisebb gyakoriságú elem azonosítására. Két elem összevonásának eredménye egy új elem, amelynek gyakorisága a két összevont elem gyakoriságának összege.

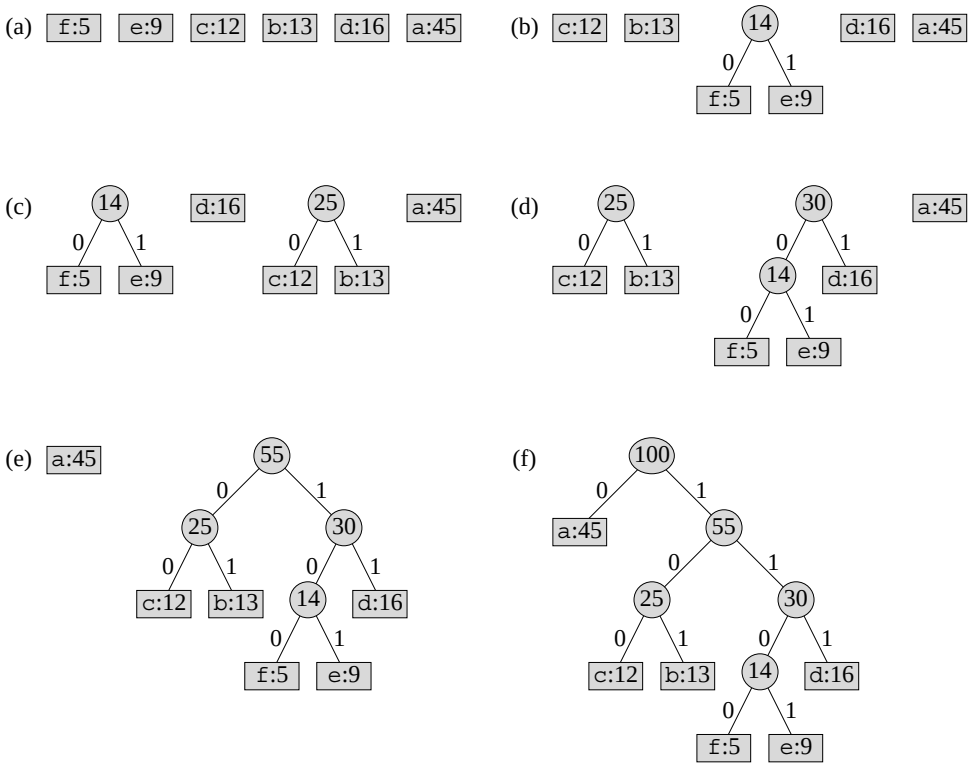
HUFFMAN(C)

```
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4      do új z csúcs létesítése
5         bal[z] ← x ← KIVESZ-MIN(Q)
6         jobb[z] ← y ← KIVESZ-MIN(Q)
7         f[z] ← f[x] + f[y]
8         BESZÚR(Q, z)
9  return KIVESZ-MIN(Q)          ▷ A fa gyökerének visszaadása.
```

A példánkban szereplő adatokra a Huffman-algoritmus a 16.5. ábrán látható módon működik. Mivel hat kódolandó karakter van, a sor mérete kezdetben $n = 6$, és 5 összevonási lépés szükséges a fa felépítéséhez. A végén kapott fa megfelel az optimális prefix-kódnak. Minden karakter kódja a gyökértől a megfelelő levélig vezető úton lévő élek címkéinek sorozata.

Az algoritmusban a 2. sor inicializálja a Q prioritási sort a C-beli karakterekkel. A 3–8. sorokban adott **for** ciklus ismétlődően kiválasztja a Q sorból az x és y két legkisebb gyakoriságú csúcsot, és beteszi a sorba azt a z új csúcsot, amely x és y összevonását ábrázolja. A z új csúcs gyakorisága x és y gyakoriságának összege lesz, amit a 7. sorban számítunk ki. A z csúcs bal gyereke x, jobb gyereke pedig az y csúcs lesz. (Itt a sorrend nem lényeges, bármely csúcs bal és jobb gyereke felcserélhető, különböző, de azonos költségű fát eredményezve.) $n - 1$ számú összevonás végrehajtása után a sorban egy csúcs marad (a kódfa gyökere), az algoritmus a 9. sor végrehajtásával ezt adja eredményül.

A Huffman-algoritmus időigényének elemzésénél feltételezzük, hogy a felhasznált prioritási sort bináris kupaccal ábrázoljuk (lásd a 6. fejezetet). A Q sor inicializálása a 2. sorban $O(n)$ időt igényel, ha a C ábécé n elemű, feltéve, hogy a 6.3. alfejezetben ismertetett KUPACOT-ÉPÍT eljárást használjuk. A 3–8. sorokban adott **for** ciklus pontosan $(n - 1)$-szer hajtódik végre, és mivel a prioritási sor minden művelete $O(\lg n)$ időt igényel, a ciklus teljes



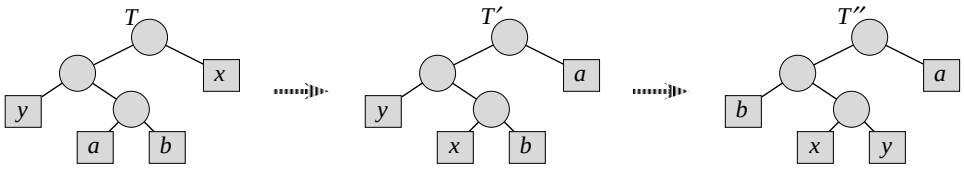
16.5. ábra. A Huffman-algoritmus lépései a 16.3. ábrán szereplő gyakoriságokra. Minden részábra tartalmazza a sor aktuális tartalmát gyakoriság szerint növekvően. Minden lépésben a két legkisebb gyakoriságú csúcsot kapcsoljuk össze. A leveleket téglalapok jelölik, feltüntetve bennük a karaktert és annak gyakoriságát. A belső csúcsokat kör jelöli, amelyekben a csúcs két gyereke gyakoriságának összege van. Minden él, amely egy belső csúcsot annak bal fiával köt össze, a 0 címkét, illetve ha a jobb fiával köti össze, akkor az 1 címkét viseli. Minden karakter kódszava az a jelsorozat, amelyet úgy kapunk, hogy a gyökérről a karaktert tartalmazó levélig vezető úton az élek címkéit egymás után írjuk. **(a)** A kezdeti állapot $n = 6$ csúccsal. **(b)–(e)** A közbülső állapotok. (**f**) A fa az eljárás végén.

futási ideje $O(n, \lg n)$. Tehát a HUFFMAN futási ideje $O(n, \lg n)$ minden n karaktert tartalmazó C halmazra.

A Huffman-algoritmus helyessége

A HUFFMAN mohó algoritmus helyességének igazolásához megmutatjuk, hogy az optimális prefix-kód meghatározása teljesíti a mohó-választási és az optimális részproblémák tulajdonságokat. A következő lemma azt bizonyítja, hogy a mohó-választási tulajdonság teljesül.

16.2. lemma. *Legyen C tetszőleges karakterhalmaz, és legyen $f[c]$ a $c \in C$ karakter gyakorisága. Legyen x és y a két legkisebb gyakoriságú karakter C-ben. Ekkor létezik olyan optimális prefix-kód, amely esetén az x-hez és y-hoz tartozó kódszó hossza megegyezik, és a két kódszó csak az utolsó bitben különbözik.*



16.6. ábra. A 16.2. lemma bizonyításának kulcslépése. Az optimális prefix-kód T fájában b és c a két legmélyebb testvércsúcs. Az x és y az a két levélcsúcs, amelyet a Huffman-algoritmus elsőnek von össze. Ezek bárhol lehetnek a fában. A b és x csúcsok felcserélésével kapjuk a T' fát. Ezután a c és y csúcsokat felcserélve adódik a T'' fa. Mivel egyik lépés hatására sem növekszik a fa költsége, a kapott T'' fa is optimális lesz.

Bizonyítás. A bizonyítás alapötlete az, hogy vegyünk egy optimális prefix-kódot ábrázoló T fát és módosítsuk úgy, hogy a fában x és y a két legmélyebben lévő testvércsúcs legyen. Ha ezt meg tudjuk tenni, akkor a hozzájuk tartozó kódszavak valóban azonos hosszúságúak lesznek, és csak az utolsó bitben különböznek.

Legyen a és b a T fában a két legmélyebb testvércsúcs. Az általánosság megszorítása nélkül feltehetjük, hogy $f[a] \leq f[b]$ és $f[x] \leq f[y]$. Mivel $f[x] \leq f[y]$ a két legkisebb gyakoriság, valamint $f[a] \leq f[b]$ tetszőleges gyakoriságok, így azt kapjuk, hogy $f[x] \leq f[a]$ és $f[y] \leq f[b]$. A 16.6. ábrán látható módon felcseréljük a T fában a és x helyét, ezzel kapjuk a T' fát, majd ebből a fából, felcserélve a b és y csúcsok helyét, kapjuk a T'' fát. A (16.3) egyenlet szerint a T és a T' fák költségének különbsége

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_{T'}(x) - f[a] d_{T'}(a) \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_T(a) - f[a] d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0. \end{aligned}$$

Az egyenlőtlenség azért teljesül, mert $f[a] - f[x]$ és $d_T(a) - d_T(x)$ nemnegatív. Pontosabban, $f[a] - f[x]$ nemnegatív, mert x egy legkisebb gyakoriságú karakter, és $d_T(a) - d_T(x)$ azért nemnegatív, mert a maximális mélységű a T fában. Hasonlóan bizonyítható, hogy b és y felcserélése esetén sem növekszik a költség, így $B(T') - B(T'')$ nemnegatív. Tehát $B(T'') \leq B(T)$, és mivel T optimális, így $B(T) \leq B(T'')$, tehát $B(T'') = B(T)$. Tehát T'' olyan optimális fa, amelyben x és y maximális mélységű testvércsúcsok, amiből a lemma állítása következik. ■

A 16.2. lemmából következik, hogy az optimális fa felépítése, az általánosság megszorítása nélkül, kezdhető a mohó választással, azaz a két legkisebb gyakoriságú karakter összevonásával. Miért tekinthető ez mohó választásnak? Azért, mert tekinthetjük a két összevont elem gyakoriságának összegét egy összevonás költségeként. A 16.3-3. gyakorlat mutatja, hogy a felépített fa teljes költsége megegyezik az összevonási lépések költségeinek összegével. HUFFMAN az összes lehetséges lépések közül mindig azt választja, amelyik a legkisebb mértékben járul hozzá a költséghez.

A következő lemma azt mutatja, hogy az optimális prefix-kód konstrukciója teljesíti az optimális részproblémák tulajdonságot.

16.3. lemma. *Legyen C tetszőleges ábécé, és minden $c \in C$ karakter gyakorisága $f[c]$. Legyen x és y a két legkisebb gyakoriságú karakter C-ben. Tekintsük azt a C' ábécét, amelyet C-ből úgy kapunk, hogy eltávolítjuk az x és y karaktert, majd hozzáadunk egy új z karaktert, tehát $C' = C - \{x, y\} \cup \{z\}$. Az f gyakoriságok C'-re megegyeznek a C-beli gyakoriságokkal, kivéve z esetét, amelyre $f[z] = f[x] + f[y]$. Legyen T' olyan fa, amely a C' ábécé egy optimális prefix-kódját ábrázolja. Ekkor az a T fa, amelyet úgy kapunk, hogy a z levélcsúcshoz hozzákapcsoljuk gyerek csúcsként x-et és y-t, olyan fa lesz, amely a C ábécé optimális prefix-kódját ábrázolja.*

Bizonyítás. Először megmutatjuk, hogy a T fa $B(T)$ költsége kifejezhető a T' fa $B(T')$ költségével a 16.5. egyenlet alapján. Minden $c \in C - \{x, y\}$ esetén $d_T(c) = d_{T'}(c)$, így $f[c]d_T(c) = f[c]d_{T'}(c)$. Mivel $d_T(x) = d_T(y) = d_{T'}(z) + 1$, így azt kapjuk, hogy

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]), \end{aligned}$$

amiből az következik, hogy

$$B(T) = B(T') + f[x] + f[y],$$

vagy ami ezzel egyenértékű

$$B(T') = B(T) - f[x] - f[y].$$

Indirekt módon bizonyítunk. Tegyük fel, hogy T nem optimális prefix-kódfa a C ábécére. Ekkor létezik olyan T'' kódfa C-re, hogy $B(T'') < B(T)$. Az általánosság megszorítása nélkül (a 16.2. lemma alapján) feltehetjük, hogy x és y testvérek. Legyen T''' az a fa, amelyet T''-ből úgy kapunk, hogy eltávolítjuk az x és y csúcsokat, és ezek közös z szülőjének gyakorisága az $f[z] = f[x] + f[y]$ érték lesz. Ekkor

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T'), \end{aligned}$$

ami ellentmond annak, hogy T' a C' ábécé optimális prefix-kódját ábrázolja. Tehát T szükségképpen a C ábécé optimális prefix-kódját ábrázolja. ■

16.4. tétel. *A* HUFFMAN *eljárás optimális prefix-kódot állít elő.*

Bizonyítás. Az állítás közvetlenül következik a 16.2. és a 16.3. lemmákból. ■

Gyakorlatok

16.3-1. Bizonyítsuk be, hogy minden olyan bináris fa, amely nem teljes, nem lehet optimális prefix-kód fája.

16.3-2. Mi az optimális Huffman-kódja az alábbi ábécének, ha a karakterek gyakoriságait az első nyolc Fibonacci-szám adja: a:1, b:1, c:2, d:3, e:5, a:8, g:13, h:21? Tudjuk ezt általánosítani arra az esetre, amikor a gyakoriságok megegyeznek az első n Fibonacci-számmal?

16.3-3. Bizonyítsuk be, hogy a fa teljes költsége kiszámítható úgy is, mint a belső csúcsok költségének összege, ahol belső csúcs költségén a csúcshoz tartozó két fiú gyakoriságának összegét értjük.
16.3-4. Bizonyítsuk be, hogy ha a karaktereket gyakoriság szerint monoton csökkenő sorrendbe rendezzük, akkor létezik olyan optimális prefix-kód, hogy a megfelelő kódszavak hosszai monoton nemcsökkenő sorozatot képeznek.
16.3-5. Tegyük fel, hogy adott a $C = \{0, 1, \ldots, n-1\}$ ábécé egy optimális prefix-kódja. A kódot a lehető legkevesebb bit átküldésével akarjuk továbbítani. Mutassuk meg, hogyan lehet C optimális prefix-kódját ábrázolni legfeljebb $2n - 1 + n\lceil \lg n \rceil$ bit felhasználásával. (*Útmutatás*. $2n - 1$ bittel leírható a fa szerkezete, amely a fa bejárásával kapható.)
16.3-6. Általánosítsuk a Huffman-algoritmust ternáris kódszavakra (vagyis olyan kódolásra, amelyben a 0, 1, 2 jeleket használhatjuk) és bizonyítsuk be, hogy az optimális ternáris kódot eredményez.
16.3-7. Tegyük fel, hogy egy adatállomány 8 bites karakterek sorozatából áll, és az összes 256 lehetséges karakter nagyjából hasonlóan gyakori: a legnagyobb gyakoriság kisebb, mint a legkisebb gyakoriság kétszerese. Mutassuk meg, hogy ebben az esetben a Huffman-kód nem hatékonyabb, mint a közönséges 8 bites fix hosszú kód.
16.3-8. Mutassuk meg, hogy 8 bites véletlen eloszlású adatállományt semmilyen tömörítő séma szerinti kódolás sem tudja akár egy bittel is jobban tömöríteni. (*Útmutatás*. Hasonlítsuk össze az állományok számát a lehetséges kódolt állományok számával.)

★ 16.4. A mohó módszerek elméleti alapjai

A mohó algoritmusokhoz egy szép elmélet kapcsolódik, amit ebben az alfejezetben vázolunk. Ez az elmélet hasznosan alkalmazható annak kimutatására, hogy a mohó algoritmus optimális megoldást szolgáltat. Az elmélet az úgynevezett „matroid” kombinatorikus struktúra fogalmára épül. Bár ez az elmélet nem fedi le az összes olyan esetet, amikor a mohó stratégia alkalmazható (mint például a 16.1. alfejezet eseménykiválasztási problémáját, vagy a 16.3. alfejezet Huffman-kód problémáját), azonban számos, a gyakorlatban fontos esetet lefed. Továbbá, ez az elmélet gyorsan fejlődik, egyre újabb alkalmazásokat teremt. A fejezet végén található megjegyzések több ide vonatkozó hivatkozást tartalmaznak.

Matroidok

A ***matroid*** olyan $M = (S, \mathcal{I})$ rendezett pár, amelyre teljesül az alábbi három feltétel.

1. S véges halmaz.
2. $\mathcal{I}$ olyan nem üres halmaz, amelynek elemei S részhalmazai, ezeket S ***független*** részhalmazainak nevezzük. Minden $B \in \mathcal{I}$ és $A \subseteq B$ halmaz esetén $A \in \mathcal{I}$. Azt mondjuk, hogy $\mathcal{I}$ teljesíti az ***öröklési*** tulajdonságot, ha teljesül rá ez a feltétel. A $\emptyset$ üres részhalmaz nyilván mindig eleme $\mathcal{I}$-nek.
3. Ha $A \in \mathcal{I}$ és $B \in \mathcal{I}$, valamint $|A| < |B|$, akkor van olyan $x \in B - A$ elem, hogy $A \cup \{x\} \in \mathcal{I}$. Ekkor azt mondjuk, hogy M teljesíti a ***felcserélési tulajdonságot.***

A „matroid” szó Hassler Whitneytől származik, aki a ***mátrix matroidokat*** tanulmányozta. Ekkor az S halmaz elemei egy adott mátrix sorai, és sorok egy halmaza független részhal-

maz, ha a sorok a hagyományos értelemben lineárisan függetlenek. Könnyen igazolható, hogy így valóban matroidot kapunk (lásd a 16.4-2. gyakorlatot).

Matroidok egy másik illusztrálása végett tekintsük az $M_G = (S_G, \mathcal{I}_G)$ ***gráfmatroidot,*** amelyet egy adott $G = (V, E)$ irányítatlan gráf definiál az alábbi módon.

- Az S_G halmaz megegyezik a G gráf éleinek E halmazával.
- Az élek egy $A \subseteq E$ halmaza független részhalmaza S-nek, azaz $A \in \mathcal{I}_G$, akkor és csak akkor, ha A körmentes. Vagyis, élek egy halmaza akkor és csak akkor független részhalmaz, ha erdőt alkot.

Az M_G gráf matroid szorosan kapcsolódik a minimális feszítőfa problémához, amit a 23. fejezetben tárgyalunk részletesen.

16.5. tétel. *Ha G irányítatlan gráf, akkor $M_G = (S_G, \mathcal{I}_G)$ matroid.*

Bizonyítás. Nyilvánvalóan $S_G = E$ véges halmaz. Továbbá $\mathcal{I}_G$ teljesíti az öröklési tulajdonságot, mivel minden erdő minden részhalmaza is erdő. Másképpen fogalmazva, élek körmentes halmazából éleket elhagyva nem keletkezhet kör.

Tehát az maradt hátra, hogy megmutassuk, M_G teljesíti a felcserélési tulajdonságot. Tegyük fel, hogy A és B erdő a G gráfban és $|A| < |B|$. Tehát A és B élek körmentes halma és B több élt tartalmaz, mint A.

A B.2. tételből következik, hogy ha egy erdő k élt tartalmaz, akkor pontosan $|V| - k$ fából áll. (Másképpen ez úgy is bizonyítható, hogy el őször tekintsük azt az erdőt, amely $|V|$ csúcsot tartalmaz élek nélkül. Ezután minden hozzávett él eggyel csökkenti a fák számát.) Tehát az A erdő $|V| - |A|$ számú, a B erdő pedig $|V| - |B|$ számú fát tartalmaz.

Mivel a B erdő kevesebb fát tartalmaz, mint az A erdő, így B biztosan tartalmaz olyan T fát, amelynek csúcsai az A erdő két különböző fájában vannak. Továbbá, mivel T összefüggő, így van olyan (u, v) éle, hogy u és v az A erdő különböző fájában van. Mivel az (u, v) él az A erdő két különböző fáját köti össze, így hozzávétele A-hoz nem eredményez kört. Tehát M_G teljesíti a felcserélési tulajdonságot, amivel igazoltuk, hogy M_G matroid. ■

Adott $M = (S, \mathcal{I})$ matroid esetén egy $x \notin A$ elemet $A \in \mathcal{I}$ ***bővítőjének*** nevezünk, ha x hozzávétele A-hoz független részhalmazt eredményez, azaz $A \cup \{x\} \in \mathcal{I}$. Példaként tekintsünk egy M_G gráf matroidot. Egy független A élhalmaznak egy e él bővítője, ha nem eleme A-nak és hozzávétele A-hoz nem eredményez kört.

Ha A független részhalmaza az M matroidnak, és nincs bővítője, akkor azt mondjuk, hogy A ***maximális.*** Tehát A maximális, ha nem tartalmazza M egyetlen bővebb független részhalmaza sem. A következő tulajdonság sokszor hasznosan alkalmazható.

16.6. tétel. *Egy matroid minden maximális független részhalmaza ugyanannyi elemet tartalmaz.*

Bizonyítás. Tegyük fel az ellenkezőjét, tehát, hogy A maximális független részhalmaza az M matroidnak és van olyan B maximális független részhalmaza M-nek, amelynek több eleme van, mint A-nak. Ekkor a felcserélési tulajdonság miatt van olyan $x \in B - A$ elem, hogy $A \cup \{x\} \in \mathcal{I}$, tehát x bővítője A-nak, ami ellentmond A maximális voltának. ■

Ezen tétel illusztrálására tekintsünk egy adott G irányítatlan összefüggő gráfhoz tartozó M_G gráf matroidot. M_G minden maximális független részhalmazának pontosan $|V| - 1$

élt kell tartalmaznia, amelyek összekötik G minden csúcsát. Az ilyen fát G ***feszítőfájának*** nevezzük.

Azt mondjuk, hogy az $M = (S, \mathcal{I})$ matroid ***súlyozott,*** ha adva van egy w súlyfüggvény, amely S minden x eleméhez a $w(x)$ szigorúan pozitív súlyértéket rendeli. A w súlyfüggvény összegzéssel kiterjeszthető S részhalmazaira:

$$w(A) = \sum_{x \in A} w(x)$$

minden $A \subseteq S$ részhalmazra. Például, ha $w(e)$ értéke egy M_G gráf matroid esetén az e él hossza, akkor $w(A)$ az A-beli élek hosszainak teljes összege.

Mohó algoritmusok súlyozott matroidokra

Sok probléma, amelyre a mohó stratégia optimális megoldást szolgáltat, megfogalmazható súlyozott matroid maximális súlyú független részhalmazának megkereséseként. Vagyis adott az $M = (S, \mathcal{I})$ súlyozott matroid, és keressük egy olyan $A \in \mathcal{I}$ független részhalmazát, amelyre a $w(A)$ érték a legnagyobb. Az olyan részhalmazt, amely független és a $w(A)$ súlyérték a lehető legnagyobb, a matroid ***optimális*** részhalmazának nevezzük. Mivel a $w(x)$ súlyérték minden $x \in S$ elemre pozitív, így az optimális részhalmaz mindig maximális független részhalmaz is, ami mindig segíti, hogy A a lehető legnagyobb legyen.

Például a ***minimális feszítőfa probléma*** esetén adott egy $G = (V, E)$ irányítatlan és összefüggő gráf és egy w hosszfüggvény, amely minden e élhez a $w(e)$ (pozitív) hosszértéket rendeli. (Itt a „hossz” kifejezést használjuk, amely az eredeti élsúlyérték a gráfban, és nem a „súly” szót, amelyet a kapcsolódó súlyozott matroid számára tartunk fenn.) A feladat: megkeresni éleknek egy olyan halmazát, amely összeköti a gráf minden csúcsát, és az élek hosszainak összege minimális. Annak érdekében, hogy e problémát optimális részhalmaz megkereséseként tekinthessük, vegyük azt a súlyozott M_G matroidot a w' súlyfüggvénnyel, amelyre $w'(e) = w_0 - w(e)$, ahol w_0 nagyobb, mint az élhosszak maximuma. Ebben a súlyozott matroidban minden súly pozitív, és egy optimális részhalmaz olyan feszítőfa lesz, amely éleinek az eredeti gráfban vett összhossza minimális. Még pontosabban, minden maximális független A részhalmaz megfelel egy feszítőfának, és mivel

$$w'(A) = (|V| - 1)w_0 - w(A)$$

minden A maximális független részhalmazra, így minden olyan független részhalmazra, amelyre $w'(A)$ maximális, a $w(A)$ érték minimális lesz. Tehát minden olyan algoritmus, amely optimális részhalmazt tud keresni, tetszőleges súlyozott matroidban, az a minimális feszítőfa problémát is meg tudja oldani.

A 23. fejezetben adunk algoritmust a minimális feszítőfa probléma megoldására, de itt olyan mohó algoritmust is kifejlesztünk, amely tetszőleges súlyozott matroidra megoldást ad. Az algoritmus bemenete egy $M = (S, \mathcal{I})$ súlyozott matroid a hozzá tartozó w pozitív súlyfüggvénnyel, a kimenete pedig egy A optimális részhalmaz. Az eljárásunkban az M matroid komponenseire az $S[M]$ és $\mathcal{I}[M]$ kifejezésekkel hivatkozunk, a súlyfüggvényt pedig w-vel jelöljük. Az algoritmus mohó, mert az $x \in S$ elemeket a súlyuk szerint nemnövekvő sorrendben vizsgálja, és azonnal hozzáveszi az A halmazhoz, ha az $A \cup \{x\}$ független részhalmaz lesz.

Mohó(M, w)

```
1  A ← ∅
2  S[M] rendezése a w súly szerint nemnövekvő sorrendbe
3  for x ∈ S[M], nemnövekvő w[x] szerinti sorrendben
4      do if A ∪ {x} ∈ I[M]
5            then A ← A ∪ {x}
6  return A
```

Az algoritmus sorraveszi az S halmaz minden elemét súly szerint nemnövekvő sorrendben. Ha a vizsgált x elem hozzávehető az A halmazhoz úgy, hogy az független részhalmaz maradjon, akkor hozzá is veszi, egyébként elveti. Mivel az üres halmaz a matroid definíciója szerint független, és az x elemet csak akkor vesszük hozzá A-hoz, ha $A \cup \{x\}$ független, így az A halmaz mindig független. Tehát a Mohó eljárás mindig független részhalmazt ad eredményül. Hamarosan látni fogjuk, hogy az A részhalmaz súlya a lehető legnagyobb, tehát A optimális részhalmaz.

A Mohó algoritmus futási ideje egyszerűen számítható. Legyen $n = |S|$. A Mohó algoritmus rendezési fázisa $O(n \lg n)$ időt igényel. A 4. sor pontosan n-szer hajtódik végre, az S halmaz minden elemére egyszer. A 4. sor minden végrehajtása annak ellenőrzését igényli, hogy az $A \cup \{x\}$ részhalmaz független-e. Ha minden ilyen ellenőrzés $O(f(n))$ időt igényel, akkor az algoritmus teljes időigénye $O(n \lg n + n f(n))$.

Bebizonyítjuk, hogy a Mohó algoritmus optimális megoldást ad.

16.7. lemma (matroidok teljesítik a mohó-választási tulajdonságot). *Legyen $M = (S, \mathcal{I})$ súlyozott matroid w súlyfüggvénnyel, és tegyük fel, hogy S a súly szerint nemnövekvő sorrendbe rendezett. Legyen x az első olyan eleme S-nek, amelyre $\{x\}$ független részhalmaz, ha egyáltalán van ilyen. Ha van ilyen x elem, akkor létezik S-nek olyan A optimális részhalmaza, amelynek x eleme.*

Bizonyítás. Ha nincs az állításban szereplő x, akkor az egyetlen független részhalmaz az üres halmaz, és nincs mit bizonyítani. Egyébként legyen B egy nem üres optimális részhalmaz. Tegyük fel, hogy $x \notin B$, mert különben legyen $A = B$, és készen vagyunk.

B-nek nincs olyan eleme, amelynek súlya nagyobb lenne, mint $w(x)$. Ugyanis, ha $y \in B$, akkor $\{y\}$ független részhalmaz, mivel $B \in \mathcal{I}$ és $\mathcal{I}$-re teljesül az öröklési tulajdonság. Az x elem választása ezért biztosítja, hogy $w(x) \geq w(y)$ teljesül minden $y \in B$ elemre.

A keresett A halmazt a következőképpen konstruáljuk. Legyen kezdetnek $A = \{x\}$. A nyilvánvalóan független x választása miatt. A felcserélési tulajdonságot kihasználva ismétlődően válasszunk egy olyan elemet a B halmazból, ami nincs A-ban, és vegyük hozzá A-hoz mindaddig, amíg $|A| = |B|$ nem teljesül. Ekkor teljesül, hogy $A = B - \{y\} \cup \{x\}$ valamely $y \in B$ elemre, tehát azt kapjuk, hogy

$$w(A) = w(B) - w(y) + w(x) \geq w(B).$$

Mivel B optimális volt, így A is optimális, és $x \in A$, ezzel a lemmát bebizonyítottuk. ■

A következőkben bebizonyítjuk, hogy ha egy elem kezdetben nem választható, akkor később sem lesz választható.

16.8. lemma. *Legyen $M = (S, \mathcal{I})$ tetszőleges matroid. Ha x olyan eleme S-nek, amely bővítője S valamely független A részhalmazának, akkor x bővítője $\emptyset$-nak is.*

Bizonyítás. Mivel x bővítője A-nak, így $A \cup \{x\}$ független részhalmaz. Mivel $\mathcal{I}$ eleget tesz az öröklési tulajdonságnak, ezért $\{x\}$ is független részhalmaz, tehát x bővítője $\emptyset$-nak. ■

16.9. következmény. *Legyen $M = (S, \mathcal{I})$ tetszőleges matroid. Ha x olyan eleme S-nek, amely nem bővítője a $\emptyset$ üres halmaznak, akkor x nem bővítője S egyetlen független A részhalmazának sem.*

Bizonyítás. Az állítás egyszerűen a 16.8. lemma fordítottja. ■

A 16.9. következmény azt mondja, hogy minden elem, amelyik kezdetben nem választható, később sem lesz választható. Tehát a Monó algoritmus nem hibázik, amikor kihagyja S azon elemét, amely kezdetben nem bővítője az üres részhalmaznak, mert az ilyen elem később sem lesz választható.

16.10. lemma (matroidok teljesítik az optimális részproblémák tulajdonságot). *Legyen x az első elem, amit a* Monó *algoritmus választ az $M = (S, \mathcal{I})$ súlyozott matroidra. Az x-et tartalmazó, maximális súlyú független részhalmaz megtalálása redukálódik annak az $M' = (S', \mathcal{I}')$ súlyozott matroid maximális súlyú független részhalmazának megkeresésére, ahol:*

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\}, \\ \mathcal{I}' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\}, \end{aligned}$$

és az M' matroid súlyfüggvénye M súlyfüggvényének a megszorítása az S' halmazra. (Az M' matroidot az M matroid x elemre vett ***kontrakciójának*** *nevezzük.)*

Bizonyítás. Ha A tetszőleges olyan maximális súlyú független részhalmaza M-nek, amely tartalmazza x-et, akkor az $A' = A - \{x\}$ független részhalmaza az M' matroidnak. Fordítva, ha A' független részhalmaza M'-nek, akkor az $A = A' \cup \{x\}$ halmaz független részhalmaza lesz M-nek. Mivel $w(A) = w(A') + w(x)$, így minden maximális súlyú megoldás M-ben, amely x-et tartalmazza, maximális súlyú megoldást ad M'-ben, és megfordítva. ■

16.11. tétel (a mohó algoritmus helyessége matroidokon). *Ha $M = (S, \mathcal{I})$ súlyozott matroid w súlyfüggvénnyel, akkor a* Monó(M, w) *eljáráshívás optimális részhalmazt eredményez.*

Bizonyítás. A 16.9. következmény szerint, minden olyan eleme S-nek, amelyet kihagyunk az elején, mert nem bővítője $\emptyset$-nak, elhagyható, mert később sem lesz használható. Amikor az első x elemet választjuk, akkor a 16.7. lemma biztosítja, hogy a Monó algoritmus nem téved, amikor hozzáveszi x-et A-hoz, mert létezik olyan optimális részhalmaz, amelynek x eleme. Végül, a 16.10. lemma szerint a fennmaradt probléma nem más, mint optimális részhalmaz megkeresése abban az M' súlyozott matroidban, amely M-nek x-re vett kontrakciója. Miután a Monó algoritmus A értékét $\{x\}$-re állítja, minden további lépése úgy interpretálható, hogy azok az $M' = (S', \mathcal{I}')$ matroidra vonatkoznak, mivel ha B független részhalmaz M'-ben, akkor $B \cup \{x\}$ független részhalmaz M-ben minden $B \in \mathcal{I}'$-re. Tehát a Monó algoritmus további működése az M' matroid egy maximális súlyú független részhalmazát fogja szolgáltatni, és így a Monó algoritmus teljes hatása az M matroid egy maximális súlyú független részhalmazát adja. ■

Gyakorlatok

16.4-1. Mutassuk meg, hogy $(S, \mathcal{I}_k)$ matroid, ha S tetszőleges véges halmaz és $\mathcal{I}_k$ az S halmaz legfeljebb k elemű részhalmazait tartalmazza ($k \leq |S|$).

16.4-2.⋆ Adott egy valós számokból álló $n \times n$ méretű T mátrix. Mutassuk meg, hogy $(S, \mathcal{I})$ matroid, ahol S a T mátrix oszlopainak halmaza, és $A \in \mathcal{I}$ akkor és csak akkor, ha az A-beli oszlopok lineárisan függetlenek.

16.4-3.⋆ Mutassuk meg, hogy ha $(S, \mathcal{I})$ matroid, akkor $(S, \mathcal{I}')$ is matroid, ahol $\mathcal{I}' = \{A' : S - A'$ tartalmaz valamely maximális $A \in \mathcal{I}$ részhalmazt$\}$. Vagyis, $(S', \mathcal{I}')$ maximális független részhalmazai éppen $(S, \mathcal{I})$ maximális független részhalmazainak a komplementerei.

16.4-4.⋆ Legyen S véges halmaz és legyen $S_1, S_2, \ldots, S_k$ az S halmaz egy nemüres, diszjunkt halmazokból álló partíciója. Definiáljuk az $(S, \mathcal{I})$ struktúrát az alábbiak szerint: $\mathcal{I} = \{A : |A \cap S_i| \leq 1; i = 1, 2, \ldots k\}$. Mutassuk meg, hogy $(S, \mathcal{I})$ matroid. Vagyis, hogy azon A halmazok, amelyeknek minden partícióval legfeljebb egy közös elemük van, egy matroid független részhalmazait alkotják.

16.4-5. Mutassuk meg, hogyan kell a súlyfüggvényt transzformálni ahhoz, hogy adott súlyozott matroidra megoldjuk a minimális súlyú maximális független részhalmaz keresésének problémáját, visszavezetve ezt a standard súlyozott matroid problémára. Indokoljuk meg körültekintően, hogy a transzformációja helyes.

⋆ 16.5. Egy ütemezési probléma

Ebben az alfejezetben egy érdekes problémát vizsgálunk, amely megoldható matroidok segítségével. A probléma az egységnyi végrehajtási idejű munkák optimális ütemezése egy processzorra. Minden munkához tartozik egy határidő és egy büntetés, amit akkor kell fizetni, ha a munkát nem végezzük el az ütemezés szerint határidőre. A probléma bonyolultnak látszik, azonban megoldható meglepően egyszerű módon mohó algoritmussal.

Az ***egységidejű munka*** olyan feladat, mint egy program, amit a számítógép pontosan egy időegység alatt tud végrehajtani. Adott egységidejű munkák egy S halmaza, ennek egy ***ütemezésén*** az S halmaz elemeinek egy permutációját értjük, amely megadja, hogy az egyes munkákat milyen sorrendben kell végrehajtani. Az ütemezésben az első munka végrehajtása a 0. időpontban kezdődik és az 1. időpontban ér véget, a második munka az 1. időpontban kezdődik és a 2. időpontban ér véget és így tovább.

Az ***egységidejű határidős-büntetéses munkák egy processzorra való ütemezése*** probléma bemeneti adatai a következők:

- az egységidejű munkák $S = \{a_1, a_2, \ldots, a_n\}$ halmaza,
- a $d_1, d_2, \ldots, d_n$ egészértékű ***határidők,*** ahol d_i az a_i munka megkövetelt befejezési határideje, és $1 \leq d_i \leq n$,
- a nemnegatív $w_1, w_2, \ldots, w_n$ súlyok, vagy ***büntetések,*** a w_i büntetést csak akkor kell fizetni, ha az a_i munkát az ütemezés szerint nem végezzük el a d_i határidőre.

A feladat S olyan ütemezésének megkeresése, amely esetén a be nem tartott határidők miatt fizetendő büntetések összege a lehető legkisebb.

Tekintsünk egy tetszőleges ütemezést. Azt mondjuk, hogy egy munka ***lekéső*** az ütemezésben, ha végrehajtása később fejeződik be, mint a határideje. Egyébként azt mondjuk, hogy a munkának ***korai*** az ütemezése. Minden ütemezés átalakítható ***korai-előbb formára,***

ahol minden korai munka megelőz minden lekéső munkát. Ennek igazolására tekintsünk egy a_i korai munkát, amely később van az ütemezésben, mint az a_j lekéső munka. Ekkor az ütemezésben felcserélhetjük az a_i és a_j munkák pozícióját anélkül, hogy a_i korai és a_j lekéső volta megváltozna.

Hasonlóan mutatható meg, hogy minden ütemezés ***kanonikus formára*** hozható, amelyben minden korai munka megelőz minden lekéső munkát, és a korai munkák határidejük szerint nemcsökkenő sorrendben vannak. Ennek elérése végett először hozzuk az ütemezést korai-előbb formára. Majd mindaddig, amíg van olyan a_i és a_j korai munka, amelyek a k és a $k+1$, időpontokra vannak beosztva, és $d_j < d_i$, cseréljük fel a_i és a_j pozícióját az ütemezésben. Mivel az a_j munka korai a csere előtt, így $k+1 \leq d_j$. Tehát $k+1 < d_i$, és ezért a_i korai marad a cserét követően is. Az a_j munkát előbbre vittük a cserével, így az természetesen a csere után is korai lesz.

Az optimális ütemezés megtalálása redukálódott munkák olyan A halmazának megtalálására, amelyek mindegyike korai az optimális ütemezésben. Ha megtaláltuk A-t, akkor az ütemezés elkészíthető úgy, hogy az ütemezésben előbb A elemeit soroljuk fel nemcsökkenő határidő szerint, majd a kimaradt $(S - A)$ lekéső munkákat tetszőleges sorrendben ez után írjuk, ami így egy kanonikus ütemezés lesz.

Azt mondjuk, hogy munkák egy A halmaza ***független,*** ha van az A-beli munkáknak olyan ütemezése, amelyben nincs lekéső. Nyilvánvaló, hogy minden ütemezés korai munkáinak halmaza független. Legyen $\mathcal{I}$ a munkák független halmazainak összessége.

Hogyan határozható meg, hogy munkák egy A halmaza független-e? Ezen probléma megoldása érdekében minden $t = 1, 2, \ldots, n$ értékre jelölje $N_t(A)$ azon A-beli munkák számát, amelyek határideje nem nagyobb, mint t.

16.12. lemma. *Munkák minden A halmazára az alábbi állítások ekvivalensek.*

1. *Az A halmaz független.*
2. *Minden $t = 1, 2, \ldots, n$ értékre $N_t(A) \leq t$.*
3. *Ha az A-beli munkákat határidejük szerint monoton nemcsökkenő sorrendbe rendezzük, akkor nem lesz lekéső munka.*

Bizonyítás. Világos, hogy ha $N_t(A) > t$, akkor nem lehet az A-beli munkákat úgy beosztani, hogy ne legyen lekéső, mert több mint t munka van, amelyeket a t ideig be kellene fejezni. Tehát (1)-ből következik (2). Ha (2) teljesül, akkor (3) is teljesül, mert nem akadunk el, ha a munkákat határidejük szerint nemcsökkenő sorrendbe rendezzük, hisz (2)-ből következik, hogy az i-edik legnagyobb határidő legfeljebb i. Végül az, hogy (3)-ból következik (1) triviális. ■

A 16.11. lemma (2) állítása alapján egyszerűen kiszámítható, hogy munkák egy adott halmaza független-e (lásd a 16.5-2. gyakorlatot).

A lekéső munkák büntetésösszegének minimalizálása ekvivalens a korai munkák büntetésösszegének maximalizálásával. A következő tétel ezért biztosítja, hogy használhatjuk a mohó algoritmust munkák olyan A független halmazának megkeresésére, amelyre a büntetések összege minimális.

16.13. tétel. *Ha S egységidejű határidős munkák egy halmaza, és $\mathcal{I}$ az összes független munkák halmaza, akkor az $(S, \mathcal{I})$ pár matroidot alkot.*

Bizonyítás. Munkák független halmazának minden részhalmaza is független, tehát az öröklési tulajdonság teljesül. A felcserélési tulajdonság teljesülésének bizonyításához legyen B és A munkák olyan független halmaza, hogy $|B| > |A|$. Legyen k a legnagyobb olyan t, amelyre $N_t(B) \leq N_t(A)$. (Ilyen t biztosan létezik, mert $N_0(A) = N_0(B) = 0$.) Mivel $N_n(B) = |B|$ és $N_n(A) = |A|$, továbbá $|B| > |A|$, így $k < n$ és $N_j(B) > N_j(A)$ minden $k + 1 \leq j \leq n$ munkára. Tehát B több olyan munkát tartalmaz, amelyeket a $k + 1$ időpontig be kell fejezni, mint az A halmaz. Legyen $a_i \in B - A$ olyan munka, amelynek határideje a legnagyobb, de nem későbbi, mint $k + 1$. Tekintsük az $A' = A \cup \{a_i\}$ halmazt.

Megmutatjuk a 16.12. lemma (2) feltételét használva, hogy A' munkák független halmaza. Minden t-re $N_t(A') = N_t(A) \leq t$ ha $0 \leq t \leq k$, mivel A független. Ha $k < t \leq n$, akkor $N_t(A') \leq N_t(B) \leq t$, mivel B független. Ezért A' független és ezzel végeztünk annak bizonyításával, hogy $(S, \mathcal{I})$ matroid. ■

A 16.11. tétel biztosítja, hogy használhatjuk a mohó algoritmust maximális súlyú független A halmaz megkeresésére. Ezután elkészíthetjük az optimális ütemezést úgy, hogy benne az A-beli munkák legyenek koraiak. Ez a módszer hatékony algoritmust eredményez az egységidejű határidős-büntetéses munkák egy processzorra való ütemezésére. A Mohó algoritmust használva az időigény $O(n^2)$, mivel az $O(n)$ számú függetlenségi ellenőrzés mindegyike $O(n)$ időt igényel (lásd a 16.5-2. gyakorlatot). Egy gyorsabb megvalósítás ötlete megtalálható a 16-4. feladatban.

A 16.7. ábra egy példát mutat egységidejű határidős-büntetéses munkák egy processzorra való ütemezésére. Ebben a példában a mohó algoritmus az a_1, a_2, a_3 és a_4 munkákat választja, azután elveti az a_5 és a_6 munkákat, végül választja az a_7 munkát. Az optimális ütemezés a

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$$

sorozat lesz, a fizetendő büntetés pedig $w_5 + w_6 = 50$.

Gyakorlatok

16.5-1. Oldjuk meg a 16.7. ábrán adott feladatot módosított adatokkal; minden w_i büntetés legyen $80 - w_i$.

16.5-2. Mutassuk meg, hogy hogyan használható fel a 16.12. lemma (2) feltétele arra, hogy $O(|A|)$ időben eldöntsük, hogy munkák egy A halmaza független-e.

Feladatok

16-1. Pénzváltás

Tekintsük a pénzváltás problémáját, ami azt jelenti, hogy adott az n felváltandó érték, amit a lehető legkevesebb érmével kell felváltani.

a. Tervezzünk olyan mohó algoritmust, amely pénzváltást végez, feltéve, hogy a felhasználható érmék értékei: 25, 10, 5 és 1. Mutassuk meg, hogy az algoritmus optimális megoldást ad.

b. Tegyük fel, hogy a felhasználható érmék értékei: $c^0, c^1, \ldots, c^k$ valamely $c > 1$ és $k \geq 1$ pozitív egész számokra. Mutassuk meg, hogy a mohó algoritmus mindig optimális megoldást ad.

	Munka						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

16.7. ábra. Egy példa egységidejű határidős-büntetéses munkák egy processzorra való ütemezésére.

c. Adjunk meg egy olyan érmehalmazt, amelyre a mohó algoritmus nem ad optimális megoldást. Az 1 értékű érme szerepeljen, hogy minden n-re legyen megoldás.

d. Adjunk olyan $O(n, k)$ idejű algoritmust, amely minden olyan esetre megoldást ad, amikor k különböző érme van és van közöttük 1 értékű érme.

16-2. Az átlagos befejezési idő minimalizálását célzó ütemezés

Tegyük fel, hogy adott egy $S = \{a_1, a_2, \ldots, a_n\}$ programhalmaz, ahol az a_i program végrehajtási ideje p_i. Egy számítógépen hajtunk végre minden programot, de egyszerre csak egyet tudunk futtatni. Adott ütemezés esetén legyen c_i az a_i program ***befejezésének időpontja.*** Olyan ütemezést keresünk, amelyre a befejezési idők átlaga a lehető legkisebb, tehát az $(1/n)\sum_{i=1}^{n} c_i$ célfüggvényt kell minimalizálni. Például ha két program van, a_1 és a_2, a $p_1 = 3$ és $p_2 = 5$ végrehajtási időkkel, és az ütemezés szerint először a_2-t, majd a_1-et hajtjuk végre, akkor $c_2 = 5$ és $c_1 = 8$, tehát a befejezési idők átlaga (5+8)/2 = 6,5.

a. Adjunk olyan algoritmust, amely előállít egy olyan ütemezést, amelyre a befejezési idők átlaga minimális. Minden program nem megszakíthatóan fut, tehát ha az a_i program végrehajtása megkezdődött, akkor folyamatosan fut p_i ideig, amíg be nem fejeződik. Bizonyítsuk be, hogy az algoritmus minimalizálja a befejezési idők átlagát, és mutassuk ki az algoritmus futási idejét.

b. Tegyük fel, hogy egyszerre nem minden program áll rendelkezésre. Pontosabban, minden programhoz adott az az r_i ***belépési időpont***, amely előtt az a_i program végrehajtása nem kezdhető el. Tegyük fel továbbá, hogy megengedjük a végrehajtás közbeni ***megszakítást,*** tehát egy program végrehajtása megszakítható, majd egy később időpontban folytatható. Például, a 6 végrehajtási idejű a_i program futása elkezdhető az 1. időpontban és megszakítható a 4. időpontban, majd folytatható a 10. időponttól a 11. időpontig, aztán ismét futhat a 11. időponttól a 15. időpontig. Az a_i program teljes futása 6 időegységet vett igénybe, de futása három részre lett felosztva. Tehát a_i befejezési időpontja 15. Adjunk olyan algoritmust a módosított feltételekre, amely előállít egy olyan ütemezést, amelyre a befejezési idők átlaga minimális. Bizonyítsuk be, hogy az algoritmus minimalizálja a befejezési idők átlagát, és mutassuk ki az algoritmus futási idejét.

16-3. Körmentes részgráfok

a. Legyen $G = (V, E)$ irányítatlan gráf. A matroid definícióját használva mutassuk meg, hogy $(E, \mathcal{I})$ matroid lesz, ha $A \in \mathcal{I}$ akkor és csak akkor, ha A élek olyan részhalmaza, amely körmentes.

b. A $G = (V, E)$ irányítatlan gráf ***illeszkedési mátrixa*** az a $|V| \times |E|$ méretű M mátrix, amelyre $M_{ve} = 1$, ha az e él valamelyik végpontja v, egyébként $M_{ve} = 0$. Igazoljuk, hogy az M mátrix oszlopainak egy halmaza akkor és csak akkor lineárisan független, ha a megfelelő élek halmaza körmentes. Ezután a 16.4-2. gyakorlatot felhasználva adjunk alternatív bizonyítást az (a) feladatra, tehát, hogy $(E, \mathcal{I})$ matroid.

c. Tegyük fel, hogy a $G = (V, E)$ irányítatlan gráf minden e éléhez adott a $w(e)$ nemnegatív súlyérték. Adjunk hatékony algoritmust E egy olyan körmentes élhalmazának megkeresésére, amelyre az élek összsúlya maximális.

d. Legyen $G = (V, E)$ tetszőleges irányított gráf, és definiáljuk az $(E, \mathcal{I})$ párt úgy, hogy $A \in \mathcal{I}$ akkor és csak akkor, ha A nem tartalmaz irányított kört. Adjunk olyan irányított G gráfot, amelyhez tartozó $(E, \mathcal{I})$ nem lesz matroid. Mondjuk meg, hogy melyik matroid tulajdonság nem teljesül.

e. A $G = (V, E)$ irányított gráf ***illeszkedési mátrixa*** az a $|V| \times |E|$ méretű M mátrix, amelyre $M_{ve} = -1$, ha az e él a v csúcsból indul, $M_{ve} = 1$, ha az e él a v csúcsba vezet, egyébként $M_{ve} = 0$. Mutassuk meg, hogy ha az M mátrix oszlopainak egy halmaza lineárisan független, akkor a megfelelő élek halmaza nem tartalmaz irányított kört.

f. A 16.4-2. gyakorlat szerint tetszőleges M mátrix esetén azon oszlopok halmazai, amelyek lineárisan függetlenek, matroidot alkotnak. Indokoljuk meg körültekintően, hogy a (d) és (e) rész állításai miért nem ellentmondóak. Mi okozza azt, hogy nem tökéletes a megfeleltetés a körmentes él-halmaz és az incidencia mátrixban neki megfelelő oszlophalmaz lineárisan függetlensége között?

16-4. Ütemezési változatok

Tekintsük az alábbi algoritmust, amely megoldja a 16.5. alfejezetben tárgyalt egységidejű határidős-büntetéses munkák egy processzorra való ütemezésének a problémáját. Legyen kezdetben mind az n időszelet, ahol az i-edik időszelet az az egységnyi hosszú intervallum, amely az i időpontban végződik. Vizsgáljuk a munkákat büntetés szerint monoton csökkenő sorrendben. A a_j munka vizsgálatakor, ha létezik olyan időszelet, amely nem későbbi, mint a a_j munka d_j határideje, azaz még szabad, akkor osszuk be a legkésőbbi ilyen időszeletre az a_j munkát, lefoglalva ezzel ezt az időszeletet. Ha nincs ilyen szabad időszelet, akkor osszuk be az a_j munkát az utolsó, még szabad időszeletre.

a. Indokoljuk meg, hogy ez az algoritmus mindig optimális megoldást ad.

b. Használjuk a 21.3. alfejezetben ismertetett gyors diszjunkthalmaz-erdő adatszerkezetet az algoritmus hatékony megvalósítására. Tegyük fel, hogy a munkák már eleve monoton csökkenő sorrendbe rendezettek a büntetéseik szerint. Elemezzük a kapott algoritmus időigényét.

Megjegyzések a fejezethez

A mohó algoritmusokról és a matroidokról többet olvashat Lawler [196], valamint Papadimitriou és Steiglitz [237] munkáiban. Mohó algoritmus kombinatorikus optimalizálás témában először Edmonds cikkében [85] jelent meg, bár a matroidok elmélete Whitney [314] 1935-ben megjelent cikkéig nyúlik vissza. Az eseménykiválasztási problémát megoldó mohó algoritmus helyességének a bizonyítása Gavril [112] bizonyítását követi. Az ütemezési problémát Lawler [196], Horowitz és Sahni [158], valamint Brassard és Bratley [47] tanulmányozta. A Huffman-kód 1952-ből származik [162]; Lelewer és Hirschberg [200] könyve áttekintést ad az 1987-ig ismert adattömörítő technikákról. A matroidok kiterjesztése a greedoidok elmélete, amiben úttörő munkát végzett Korte és Lovász [189, 190, 191, 192], akik jelentősen általánosították az ebben a könyvben ismertetetteket.

17. Amortizációs elemzés

Egy ***amortizációs elemzés*** során adatszerkezetekkel végzett műveletek végrehajtásához szükséges időt átlagoljuk az összes elvégzett művelet esetére. Az amortizációs elemzés annak kimutatására használatos, hogy egy művelet átlagos költsége kicsi, miközben a műveletsorozatban egy-egy végrehajtása drága lehet. Az amortizációs elemzés abban különbözik az átlagos eset vizsgálatától, hogy itt a véletlen semmilyen szerepet nem játszik; az amortizációs elemzés az *egyes műveletek átlagos költségére* garantál felső korlátot a *legrosszabb esetben.*

A jelen fejezet első három alfejezete az amortizációs elemzés három leggyakoribb módszerét mutatja be. A 17.1. alfejezetben ismertetjük az összesítéses elemzést, amelyben egy n műveletből álló sorozat elvégzésének összköltségére határozunk meg egy $T(n)$ felső korlátot. A műveletek átlagos költsége ekkor $T(n)/n$. Ezt az átlagos költséget tekintjük az egyes műveletek amortizációs költségének, így minden műveletnek ugyanakkora lesz az amortizációs költsége.

A 17.2. alfejezet a könyvelési módszert mutatja be, amelyben minden művelet amortizációs költségét meghatározzuk. Ha többfajta művelet van, akkor a különböző típusú műveletek amortizációs költsége különböző lehet. A könyvelési módszer a sorozat elején túlszámláz egyes műveleteket, és ezt nyilvántartja az adatszerkezet meghatározott elemein mint „előre kifizetett hitelt”. Ezt a hitelt később a sorozatban olyan műveletek kifizetésére használjuk, amelyekre a tényleges költségnél kevesebbet számlázunk.

A 17.3. alfejezetben tárgyaljuk a potenciál módszert, amelyben a könyvelési módszerhez hasonlóan minden művelet amortizációs költségét meghatározzuk, és a sorozat elején egyes műveleteket túlszámlázhatunk, amit később más műveletek alulszámlázásával egyenlítünk ki. A potenciál módszer a hitelt az adatszerkezet egészének „potenciális energiájaként” tartja nyilván ahelyett, hogy azt az adathalmaz egyes elemeihez kötné.

Két példát használunk a három módszer vizsgálatára. Az egyik egy TÖBBSZÖRÖS-VEREMBŐL nevű művelettel kiegészített verem. Ez a művelet egyszerre több elemet vesz le a verem tetejéről. A másik egy bináris számláló, amely 0-tól számlál az egyetlen NÖVEL nevű művelet segítségével.

A fejezet olvasása közben ne felejtsük el, hogy az amortizációs elemzés során használt költségek csak az elemzés célját szolgálják. Nem kell és nem is szabad megjelenniük egy valódi programban. Például, ha egy x elemhez valamilyen hitel tartozik a könyvelési módszer során, akkor nem szükséges, hogy a megfelelő összeget a programban egy *hitel*[x] mezőhöz hozzárendeljük.

Egy speciális adatszerkezetnek az amortizációs elemzés által nyújtott mélyebb megértése segíthet az adatszerkezet jobb megtervezésében. Például a 17.4. alfejezetben a potenciál módszert fogjuk használni egy dinamikusan kiterjedő és összehúzódó tömb elemzésére.

17.1. Összesítéses elemzés

Az ***összesítéses elemzésnél*** minden n-re megmutatjuk, hogy egy n műveletből álló sorozat a legrosszabb esetben összesen $T(n)$ ideig tart. Tehát a *legrosszabb esetben* a műveletenkénti átlagos költség vagy ***amortizációs költség*** $T(n)/n$. Hangsúlyozandó, hogy ugyanaz az amortizációs költség vonatkozik minden műveletre, még akkor is, ha különböző típusú műveletek vannak a sorozatban. A másik két módszer – a könyvelési és a potenciál módszer – azonban különböző típusú műveletekhez különböző amortizációs költséget rendelhet.

Veremműveletek

Az összesítéses elemzés első példájában egy vermet elemzünk, amelyet egy új művelettel egészítettünk ki. A 10.1. alfejezetben bevezettük a két alapvető veremműveletet, amelyek mindegyike $O(1)$ ideig tart:

VEREMBE(S, x) az x objektumot berakja az S verembe.

VEREMBŐL(S) az S verem tetején lévő elemet kiveszi és eredményül visszaadja.

Mivel mindkét művelet $O(1)$ ideig tart, tekintsük mindkettő költségét 1-nek. A teljes költsége n VEREMBE és VEREMBŐL műveletnek n, tehát n művelet tényleges futási ideje $\Theta(n)$.

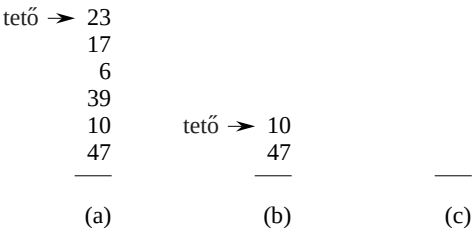
Most hozzávesszük a veremhez a TÖBBSZÖRÖS-VEREMBŐL(S, k) műveletet, amely az S verem tetejéről k objektumot vesz le, vagy kiüríti a teljes vermet, ha az kevesebb mint k objektumot tartalmaz. A következő pszeudokódban az ÜRES-VEREM művelet IGAZ választ ad, ha pillanatnyilag nincs objektum a veremben, különben pedig HAMIS-t.

```
TÖBBSZÖRÖS-VEREMBŐL(S, k)
1  while ¬ÜRES-VEREM(S) és k ≠ 0
2        do VEREMBŐL(S)
3           k ← k – 1
```

A TÖBBSZÖRÖS-VEREMBŐL műveletre a 17.1. ábra mutat példát.

Mi a TÖBBSZÖRÖS-VEREMBŐL(S, k) futási ideje, ha a veremben s objektum van? A valódi futási idő lineáris a ténylegesen végrehajtott VEREMBŐL műveletek számában, és ezért elegendő a TÖBBSZÖRÖS-VEREMBŐL-t a VEREMBE és VEREMBŐL művelet absztrakt 1 költsége szerint elemezni. A ***while*** ciklus iterációinak száma $\min(s, k)$, ami nem más, mint a veremből kivett objektumok száma. A ciklus minden iterációjában meghívjuk egyszer a VEREMBŐL eljárást a 2. sorban. Így tehát a TÖBBSZÖRÖS-VEREMBŐL teljes költsége $\min(s, k)$, és a tényleges futási idő ennek a mennyiségnek lineáris függvénye.

Elemezzük VEREMBE, VEREMBŐL és TÖBBSZÖRÖS-VEREMBŐL műveletek egy n hosszú sorozatát, amelyet egy eredetileg üres veremre alkalmazunk. A legrosszabb esetben a TÖBBSZÖRÖS-VEREMBŐL költsége $O(n)$, hiszen a verem mérete legfeljebb n. Tehát most valamennyi művelet költsége a legrosszabb esetben $O(n)$, és így az egész sorozat költsége $O(n^2)$, mivel $O(n)$ TÖBBSZÖRÖS-VEREMBŐL művelet lehet a sorozatban és mindegyik $O(n)$ ideig tart. Habár



17.1. ábra. A TÖBBSZÖRÖS-VEREMBŐL hatása egy S veremre, amelynek kezdeti állapotát az **(a)** oszlop mutatja. A verem tetején lévő 4 objektumot kivette a TÖBBSZÖRÖS-VEREMBŐL(S, 4) hívás, amelynek eredménye a **(b)** oszlopban található. A következő művelet a TÖBBSZÖRÖS-VEREMBŐL(S, 7), amely – mint az a **(c)** oszlopban látható – kiüríti a vermet, mert abban kevesebb mint 7 elem maradt.

ez a gondolatmenet helyes, az $O(n^2)$ eredmény, amelyet az egyes műveletek legrosszabb esetbeli költségei alapján kaptunk, nem éles.

Összesítéses elemzést használva pontosabb felső korlátot kaphatunk, ami az egész n hosszú műveletsorozatot figyelembe veszi. Habár egy TÖBBSZÖRÖS-VEREMBŐL művelet drága lehet, n VEREMBE, VEREMBŐL és TÖBBSZÖRÖS-VEREMBŐL műveletnek egy eredetileg üres veremre alkalmazott sorozata legfeljebb $O(n)$ költségű lehet. Miért? Minden objektumot a verembe való berakása után legfeljebb egyszer lehet kivenni. Ezért a VEREMBŐL eljárás hívásainak száma, beleértve azokat is, amelyek a TÖBBSZÖRÖS-VEREMBŐL eljárás alatt történnek, legfeljebb annyi, mint amennyi a VEREMBE hívásainak száma, mely utóbbi legfeljebb n. Tehát minden n-re, VEREMBE, VEREMBŐL és TÖBBSZÖRÖS-VEREMBŐL műveletek bármely n hosszú sorozata esetén a futási idő összesen $O(n)$. Egy művelet átlagos költsége $O(n)/n = O(1)$. Összesítéses elemzésnél minden művelet amortizációs költségén az átlagos költséget értjük. Ezért ebben a példában mindhárom veremművelet amortizációs költsége $O(1)$.

Bár azt mutattuk meg, hogy egy veremművelet átlagos költsége és emiatt átlagos futási ideje is $O(1)$, nem használtunk semmilyen valószínűségszámításon alapuló érvelést. Azt igazoltuk, hogy $O(n)$ az n művelet teljes költségének felső korlátja a *legrosszabb esetben.* Ezt n-nel osztva kaptuk egy művelet átlagos költségét, vagyis az amortizációs költséget.

Bináris számláló növelése

Az összesítéses elemzést illusztráló másik példaként tekintsük a 0-tól induló k bites számláló megvalósításának a feladatát. Bitek egy $A[0\,..\,k-1]$ vektorát használjuk számlálóként, ahol $hossz[A] = k$. Egy, a számlálóban tárolt x bináris szám legalacsonyabb helyi értékű bitje $A[0]$-ban van, a legmagasabb helyi értékű pedig $A[k-1]$-ben, tehát $x = \sum_{i=0}^{k-1} A[i]2^i$. Kezdetben $x = 0$, és így $A[i] = 0$ ($i = 0, 1, \ldots, k-1$). Arra, hogy x-hez hozzáadjunk 1-et (mod 2^k), a következő eljárást használjuk.

```
NÖVEL(A)
1  i ← 0
2  while i < hossz[A] és A[i] = 1
3        do A[i] ← 0
4           i ← i + 1
5  if i < hossz[A]
6    then A[i] ← 1
```

A számláló értéke	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	A teljes költség
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

17.2. ábra. Egy 8 bites számláló, amint értéke 0-ról 16-ra változik 16 NÖVEL művelet következtében. A megváltozott bitek satírozva láthatók. A bitek megváltoztatásának költsége a jobb oldalon látható. Megfigyelhető, hogy a teljes költség sohasem több, mint az összes NÖVEL hívás számának kétszerese.

A 17.2. ábra mutatja, hogy mi történik egy bináris számlálóban, ha az 0-ból indulva 16 növelés után a 16 értékhez jut. A 2–4. sorokban található **while** ciklus minden iterációjának kezdetén az i-edik pozícióhoz akarjuk az 1-et hozzáadni. Ha $A[i] = 1$, akkor az 1-gyel való növelés az i bitet 0-vá változtatja és átvitelként 1-et ad, amit a ciklus következő iterációjában az $(i+1)$ pozícióhoz kell adni. Különben a ciklus véget ért, és ha $i < k$, akkor mivel $A[i] = 0$, 1-nek az i-edik pozícióhoz adása, a 0-nak 1-re váltása a 6. sorban történik meg. A NÖVEL művelet költsége lineáris a megváltoztatott bitek számában.

A verem példához hasonlóan a felületes vizsgálattal kapott korlát helyes, de nem éles. A NÖVEL művelet a legrosszabb esetben, amikor az A vektor minden bitje 1, $\Theta(k)$ ideig tart. Így a kezdetben 0 számlálóra alkalmazott n NÖVEL művelet a legrosszabb esetben $O(nk)$ időt vesz igénybe.

Élesíthetjük azonban a korlátot azáltal, ha észrevesszük, hogy az n NÖVEL műveletből álló sorozat nem változtatja meg mindig az összes bitet. Így a korlát $O(n)$-re javul. Ugyanis, ahogy az a 17.2. ábrán látszik, az $A[0]$ bit mindig megváltozik. A következő legalacsonyabb helyi értékű, az $A[1]$ bit csak minden második alkalommal változik meg: egy kezdetben 0 számlálóra alkalmazott n egymás utáni NÖVEL művelet $A[1]$-et $\lfloor n/2 \rfloor$-szer változtatja meg. Hasonlóképpen $A[2]$ csak minden negyedik alkalommal érintett a műveletben, vagyis $\lfloor n/4 \rfloor$-szer. Általában $i = 0, 1, \ldots, k-1$ esetén az $A[i]$ bit megváltozásainak száma $\lfloor n/2^i \rfloor$. Ha $i \geq k$, akkor a bit sohasem vesz részt a műveletben. A bitek változásainak a számára az (A.6) képlet alapján a

$$\begin{aligned} \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n \end{aligned}$$

korlátot kapjuk. Tehát egy kezdetben a 0-t tartalmazó számlálóra alkalmazott n egymást követő NÖVEL művelet futási ideje a legrosszabb esetben $O(n)$. A műveletek átlagos költsége, és ezért az egyes műveletek amortizációs költsége $O(n)/n = O(1)$.

Gyakorlatok

17.1-1. Igaz maradna-e a veremműveletek amortizációs költségére vonatkozó $O(1)$ korlát, ha a veremműveletek közé beiktatnánk a verembe k elemet tevő TÖBBSZÖRÖS-VEREMBE műveletet is?

17.1-2. Mutassuk meg, hogy ha egy CSÖKKENT műveletet is hozzávennénk a k bites számláló példájához, akkor n művelet költsége akár $\Theta(nk)$ is lehetne.

17.1-3. Műveletek egy n hosszú sorozatát hajtjuk végre egy adatszerkezeten. Az i-edik művelet költsége i, ha i éppen egy kettőhatvány, minden más esetben 1. Alkalmazzunk összesítéses elemzést a műveletenkénti amortizációs költség meghatározására.

17.2. A könyvelési módszer

Az amortizációs elemzés ***könyvelési módszerében*** különböző műveletekre különböző költséget számítunk fel, megengedve azt is, hogy egyes műveletek esetében a tényleges költségnél többet vagy kevesebbet számlázzunk. Az az összeg, amit egy adott műveletre felszámítunk, az adott művelet ***amortizációs költsége.*** Ha egy művelet amortizációs költsége meghaladja a tényleges költségét, akkor a különbséget az adatszerkezet bizonyos objektumaihoz rendeljük mint ***hitelt.*** A hitelt később arra fordíthatjuk, hogy olyan műveletekért fizessünk, amelyek amortizációs költsége kisebb a tényleges költségnél. Tehát egy művelet amortizációs költségét úgy tekinthetjük, mint ami megoszlik a tényleges költség és a hitel között, mely utóbbit hol letétbe helyezzük, hol felhasználjuk. Ez nagyon eltér tehát az összesítéses elemzéstől, ahol minden művelet amortizációs költsége azonos.

A műveletek amortizációs költségét gondosan kell megválasztani. Ha meg akarjuk mutatni az amortizációs költségek felhasználásával, hogy a legrosszabb esetben az átlagos műveletenkénti költség kicsi, akkor az adott műveletsorozat amortizációs összköltsége a tényleges összköltség felső korlátja kell hogy legyen. Továbbá, éppúgy mint az összesítéses elemzésnél, ennek a relációnak minden műveletsorozatra érvényesnek kell lennie. Ha az i-edik művelet tényleges költségét c_i-vel, amortizációs költségét $\widehat{c_i}$-val jelöljük, fenn kell állnia a

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i \tag{17.1}$$

egyenlőtlenségnek, minden n hosszú műveletsorozatra. Az adatszerkezetben tárolt teljes hitel a teljes amortizációs költség és a teljes tényleges költség különbsége, vagyis $\sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i$. A (17.1) egyenlőtlenség szerint az adatszerkezethez rendelt teljes hitelnek minden pillanatban nemnegatívnak kell lennie. Ha megengednénk, hogy a teljes hitel negatívvá váljon (annak eredményeként, hogy korábbi műveleteket alulszámláznánk azt ígérve, hogy a tartozást később visszafizetjük), akkor az eddig a pillanatig felmerült teljes amortizációs költség az ugyaneddig a pillanatig felmerült teljes tényleges költség alatt volna, tehát az eddig a pillanatig tartó műveletsorozatnak a teljes amortizációs költsége nem volna felső korlátja a tényleges költségének. Vigyáznunk kell tehát, hogy az adathalmazon a teljes hitel soha ne váljon negatívvá.

Veremműveletek

Az amortizációs elemzés könyvelési módszerét illusztrálandó, tekintsük ismét a verem példát. Emlékeztetünk arra, hogy a műveletek tényleges költsége

VEREMBE	1,
VEREMBŐL	1,
TÖBBSZÖRÖS-VEREMBŐL	min(k, s),

ahol k a TÖBBSZÖRÖS-VEREMBŐL bemenő paramétere, s pedig a verem mérete a híváskor. Rendeljük a következő amortizációs költségeket a műveletekhez:

VEREMBE	2,
VEREMBŐL	0,
TÖBBSZÖRÖS-VEREMBŐL	0.

Figyeljük meg, hogy a TÖBBSZÖRÖS-VEREMBŐL amortizációs költsége állandó (0), míg a tényleges költsége változó. Most mindhárom amortizációs költség nagyságrendje $O(1)$, de a különböző műveletek amortizációs költségei általában aszimptotikusan eltérhetnek egymástól.

Megmutatjuk, hogy bármely műveletsorozatot ki tudunk fizetni ezekkel az amortizációs költségekkel. Tegyük fel, hogy egy tallér felel meg a költség egységének. Kezdetben a verem üres. Emlékeztetünk a 10.1. alfejezetben említett, a verem típusú adatszerkezet és egy vendéglőbeli tányérokból álló halom közti analógiára. Amikor egy tányért teszünk a halom tetejére, akkor 1 tallért fizetünk a tényleges költségért, és a felszámított 2 tallérból marad 1 tallér hitel, amit a tányér tetejére teszünk. Minden pillanatban tehát minden tányéron ott áll 1 tallér hitel.

A tányéron található tallér a halomból való kivétel előre kifizetett költsége. Amikor egy VEREMBŐL műveletet végrehajtunk, nem számítunk fel semmit, és a tényleges költséget a halomban tárolt hitelből fizetjük ki. Amikor kivesszük a tányért, akkor az egytallérnyi hitelt levesszük róla, és vele kifizetjük a művelet tényleges költségét. Így azzal, hogy a VEREMBE műveletet egy kicsit túlszámlázzuk, elkerüljük, hogy bármit is fel kelljen számítani a VEREMBŐL műveletért.

Továbbá nem kell felszámítanunk semmit a TÖBBSZÖRÖS-VEREMBŐL műveletért sem. Az első tányér kivételekor a rajta levő tallérral fizetjük ki a VEREMBŐL művelet költségét. A második és a még későbbi tányérok esetében ugyanígy járunk el. Azaz elég pénzt szedtünk be előre, hogy bármikor kifizethessük a TÖBBSZÖRÖS-VEREMBŐL műveletet. Más szavakkal, mivel a halomban minden tányéron van egy tallér hitel, és a halomban mindig nemnegatív számú tányér van, biztos, hogy a hitel összege mindenkor nemnegatív. Tehát a VEREMBE, VEREMBŐL és TÖBBSZÖRÖS-VEREMBŐL műveletek *bármely* sorozata esetén a teljes amortizációs költség felső korlátja a teljes tényleges költségnek. Mivel a teljes amortizációs költség $O(n)$, ugyanennyi a teljes tényleges költség nagyságrendje is.

Bináris számláló növelése

A könyvelési módszert bemutató másik példa a 0-ból induló bináris számláló NÖVEL műveletének elemzése. Mint ahogy azt korábban már megállapítottuk, ennek a műveletnek a futási ideje arányos a megváltoztatott bitek számával, ezért ezt a mennyiséget fogjuk költségnek tekinteni a példában. Ismét egy tallér feleljen meg a költség egységének, vagyis egy bit megváltoztatásának.

Az elemzés céljából számítsunk fel 2 tallér amortizációs költséget akkor, ha egy bitet 1-re állítunk át. Ekkor az egyik tallérral kifizetjük a megváltoztatás költségét, a másik tallért pedig a bitre helyezzük hitelként, amelyet később akkor használunk fel, mikor a bitet visszaváltoztatjuk 0-ra. Minden pillanatban a számláló minden egyesének van 1 tallér hitele, és így nem kell felszámítani semmit, amikor a bitet visszaállítjuk 0-ra, mert azt ebből a tallérból fizetjük ki.

A NÖVELÉS amortizációs költsége most már meghatározható. A **while** ciklusban a bit 1-ről 0-ra való visszaállításának költségét a biten található tallérból egyenlítjük ki. Legfeljebb egy bitet állítunk 1-re a NÖVEL algoritmusának 6. sorában, így a művelet amortizációs költsége legfeljebb 2 tallér. Az egyesek száma a számlálóban sohasem negatív, tehát a hitel is mindig nemnegatív. Így n NÖVEL művelet teljes amortizációs költsége, ami $O(n)$, felső korlátja a teljes tényleges költségnek.

Gyakorlatok

17.2-1. Veremműveletek sorozatát hajtjuk végre egy vermen, melynek mérete sohasem nagyobb k-nál. Minden k művelet után lemásoljuk az egész vermet mentés céljából. Az amortizációs költségek alkalmas megválasztásával mutassuk meg, hogy n veremművelet költsége, a másolást is beleértve, $O(n)$.

17.2-2. Oldjuk meg újra a 17.1-3. gyakorlatot úgy, hogy az elemzést a könyvelési módszerrel végezzük el.

17.2-3. Tegyük fel, hogy nemcsak növelni akarjuk a számlálót, hanem nullára visszaállítani is, azaz minden bitjét 0-val egyenlővé tenni. Mutassuk meg, hogyan lehet a számlálót mint bitvektort úgy létrehozni, hogy összesen n NÖVEL és VISSZAÁLLÍT futási ideje $O(n)$ legyen, ha a számláló a 0-ból indul. (*Útmutatás.* Tartsunk fenn egy mutatót a legmagasabb helyi értékű egyesnek.)

17.3. A potenciál módszer

Az amortizációs elemzés ***potenciál módszere*** az előre kifizetett munkát nem az adatszerkezet egyes elemeihez rendelt hitelként tartja nyilván, hanem mint „potenciális energiát” vagy egyszerűen „potenciált,” ami a jövőbeli műveletek kifizetésére használható. Ezt a potenciált az adatszerkezet egészéhez és nem annak egyes elemeihez rendeljük hozzá.

A potenciál módszer a következőképpen működik. A kezdeti adatszerkezet legyen D_0, amelyen n műveletet hajtunk végre. Minden $i = 1, 2, \ldots, n$ esetén legyen c_i az i-edik művelet tényleges költsége, D_i pedig az az adatszerkezet, amelyet az i-edik művelet eredményeként kapunk D_{i-1}-ből. A Φ ***potenciál függvény*** minden D_i adatszerkezethez egy valós $\Phi(D_i)$ számot rendel, ami a D_i adatszerkezethez rendelt ***potenciál.*** Az i-edik művelet $\widehat{c}_i$ ***amortizációs költségét*** a Φ potenciálfüggvényre vonatkozóan a

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{17.2}$$

egyenlettel definiáljuk. Tehát minden művelet amortizációs költsége a tényleges költség és a művelet által okozott potenciálnövekedés összege. A (17.2) egyenlőség n művelet teljes amortizációs költségére a

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \qquad (17.3)\end{aligned}$$

összefüggést adja. Itt a második egyenlőség (A.9) alapján következik, hiszen a $\Phi(D_i)$ értékek teleszkópszerűen kiejtik egymást.

Ha egy olyan Φ potenciál függvényt tudunk definiálni, hogy $\Phi(D_n) \geq \Phi(D_0)$, akkor a $\sum_{i=1}^{n} \widehat{c_i}$ teljes amortizációs költség felső korlátja a teljes tényleges költségnek. A gyakorlatban nem tudjuk mindig előre, hogy hány műveletet kell elvégezni. Ezért ha megköveteljük, hogy minden i-re $\Phi(D_i) \geq \Phi(D_0)$ teljesüljön, akkor biztosan előre fizetünk, éppúgy, mint a könyvelési módszernél. Sokszor kényelmes $\Phi(D_0)$-t nullának definiálni, és megmutatni, hogy $\Phi(D_i) \geq 0$ minden i-re. (A 17.3-1. gyakorlat mutat egy könnyű módot arra, hogy hogyan kezelhetők az olyan esetek, amikor $\Phi(D_0) \neq 0$.)

Szemléletesen, ha az i-edik művelet $\Phi(D_i) - \Phi(D_{i-1})$ potenciálváltozása pozitív, akkor az i-edik művelet $\widehat{c_i}$ amortizációs költsége túlszámlázást takar, és az adatszerkezet potenciálja nőtt. Ha a potenciálváltozás negatív, akkor az i-edik műveletet alulszámláztuk, és a tényleges költséget a potenciál csökkenése árán fizettük ki.

A (17.2) és (17.3) egyenlettel definiált amortizációs költség a Φ potenciálfüggvény megválasztásától függ. Különböző potenciálfüggvények különböző amortizációs költséget adhatnak, amelyek mind felső korlátjai a tényleges költségnek. A potenciálfüggvény megválasztásánál gyakran kell egyensúlyt teremteni különböző szempontok között; hogy melyik a legjobb potenciálfüggvény, az attól függ, hogy milyen időkorlátot kívánunk bizonyítani.

Veremműveletek

Ismét a VEREMBE, VEREMBŐL és TÖBBSZÖRÖS-VEREMBŐL veremműveletek példájához fordulunk, hogy bemutassuk a potenciál módszert. A Φ potenciálfüggvényt a veremben lévő objektumok számaként definiáljuk. A kezdeti D_0 üres veremre $\Phi(D_0) = 0$. Mivel a veremben lévő objektumok száma sohasem negatív, ezért az i-edik művelet után keletkező D_i veremnek nemnegatív a potenciálja, azaz

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0).\end{aligned}$$

Tehát n művelet teljes amortizációs költsége ezen Φ függvény mellett a műveletek teljes tényleges költségének felső korlátja.

Számítsuk ki a különböző műveletek amortizációs költségét. Ha az i-edik művelet egy s objektumot tartalmazó verem esetén VEREMBE, akkor a potenciálváltozás

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1.\end{aligned}$$

A (17.2) egyenlőség szerint a VEREMBE művelet amortizációs költsége így

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

Tegyük fel, hogy TÖBBSZÖRÖS-VEREMBŐL(S, k) a vermen végzett i-edik művelet, amely $k' = \min(k, s)$ objektumot vesz ki a veremből. A művelet tényleges költsége k', és a potenciálváltozás

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Így a TÖBBSZÖRÖS-VEREMBŐL művelet amortizációs költsége

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0. \end{aligned}$$

Ehhez hasonlóan az egyszerű VEREMBŐL művelet amortizációs költsége 0.

Mindhárom művelet amortizációs költsége $O(1)$, ezért az n műveletből álló sorozat amortizációs költsége $O(n)$. Mivel már megmutattuk, hogy $\Phi(D_i) \geq \Phi(D_0)$, n művelet teljes amortizációs költsége felső korlátja a tényleges költségüknek, azaz a legrosszabb esetben $O(n)$.

Bináris számláló növelése

A potenciál módszert bemutató másik módszer ismét a bináris számláló növelése. Legyen a potenciálfüggvény az i-edik NÖVEL művelet után a számlálóban található egyesek száma, amit b_i-vel jelölünk.

Számítsuk ki a NÖVEL művelet amortizációs költségét. Tegyük fel, hogy az i-edik NÖVEL művelet t_i bitet változtat vissza 1-ről 0-ra. Ezért a művelet tényleges költsége legfeljebb $t_i + 1$, ami az említett t_i bit megváltoztatását és legfeljebb egy bitnek 0-ról 1-re való fordítását takarja. Ha $b_i = 0$, akkor az i-edik művelet mind a k bitet 0-ra állítja, és ezért $b_{i-1} = t_i = k$. Ha $b_i > 0$, akkor $b_i = b_{i-1} - t_i + 1$. Mindkét esetben azt kapjuk, hogy $b_i \leq b_{i-1} - t_i + 1$, és így a potenciálváltozásra

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i. \end{aligned}$$

Az amortizációs költség tehát

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2. \end{aligned}$$

Ha a számláló nulláról indul, akkor $\Phi(D_0) = 0$. Mivel minden i esetén $\Phi(D_i) \geq 0$, n NÖVEL művelet sorozatának teljes amortizációs költsége felső korlátja a tényleges költségnek, és ez a legrosszabb esetben $O(n)$.

A potenciál módszer segítségével könnyű elemezni a számlálót akkor is, amikor nem nullából indul. Kezdetben b_0 egyes van benne, n NÖVEL művelet után pedig b_n, ahol $0 \leq b_0, b_n \leq k$. (k a számlálóban található bitek száma.) A (17.3) egyenlet átrendezéséből nyerjük, hogy

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c_i} - \Phi(D_n) + \Phi(D_0). \tag{17.4}$$

Tudjuk, hogy $\widehat{c_i} \leq 2$ minden $i = 1, 2, \ldots, n$ esetén. Mivel $\Phi(D_0) = b_0$ és $\Phi(D_n) = b_n$, n NÖVEL művelet teljes tényleges költségére azt kapjuk, hogy

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0. \end{aligned}$$

Figyeljük meg azt is, hogy mivel $b_0 \leq k$, ezért ha $k = O(n)$, akkor a teljes tényleges költség $O(n)$. Más szóval, ha legalább $n = \Omega(k)$ NÖVEL műveletet végzünk, akkor a teljes tényleges költség $O(n)$ lesz, függetlenül attól, hogy a számláló honnan indult.

Gyakorlatok

17.3-1. Tegyük fel, hogy egy olyan Φ potenciálfüggvényünk van, amelyre minden i esetén igaz, hogy $\Phi(D_i) \geq \Phi(D_0)$, de $\Phi(D_0) \neq 0$. Mutassuk meg, hogy ekkor létezik egy Φ' potenciálfüggvény, amelyre $\Phi'(D_0) = 0$ és minden $i \geq 1$ esetén $\Phi'(D_i) \geq 0$, továbbá a Φ' szerinti amortizációs költség azonos a Φ szerint számítottal.

17.3-2. Oldjuk meg ismét a 17.1-3. gyakorlatot a potenciál módszer segítségével.

17.3-3. Tekintsük az n elemen megadott egyszerű bináris min-kupac adatszerkezetet, amely a BESZÚR és KIVESZ-MIN műveleteket hajtja végre a legrosszabb esetben $O(\log n)$ idő alatt. Adjunk meg egy olyan Φ potenciálfüggvényt, amelynél a BESZÚR amortizációs költsége $O(\log n)$ és a KIVESZ-MIN műveleté pedig $O(1)$, és mutassuk meg, hogy ez működik. A futási idő nagyságrendje meghatározható.

17.3-4. Mi a teljes költsége n VEREMBE, VEREMBŐL és TÖBBSZÖRÖS-VEREMBŐL veremműveletnek, ha kezdetben s_0, a végén s_n objektum van a veremben?

17.3-5. Tegyük fel, hogy a számláló nulla helyett egy olyan számnál kezd működni, amelynek bináris felírásában b darab egyes van. Mutassuk meg, hogy n NÖVEL művelet végrehajtásának költsége $O(n)$, feltéve, hogy $n = \Omega(b)$. (Ne tegyük fel, hogy b állandó.)

17.3-6. Mutassuk meg, hogy hogyan lehet kialakítani egy (várakozási) sor típusú adatszerkezetet két közönséges verem segítségével úgy (lásd 10.1-6. gyakorlat), hogy mind a SORBA, mind a SORBÓL művelet amortizációs költsége $O(1)$.

17.3-7. Tervezzünk adatszerkezetet, amely a következő két műveletet hajtja végre egész számok egy S halmazán:

BESZÚR(S, x) beszúrja x-et S-be.

NAGYOBBIK-FELET-TÖRÖL(S) törli S-ből a legnagyobb $\lceil |S|/2 \rceil$ elemet.

Magyarázzuk el, hogyan kell megvalósítani ezt az adatszerkezetet úgy, hogy bármely m művelet futási ideje $O(m)$ legyen.

17.4. Dinamikus táblák

Egyes alkalmazásokban nem tudjuk előre, hány objektumot kell tárolni egy tömbben. Előfordulhat, hogy nem adunk elég helyet a tömbnek. Ekkor a méretét meg kell növelni, és minden benne tárolt objektumot át kell másolni az új, nagyobb tömbbe. Hasonlóképpen előfordulhat, hogy sok elemet töröltünk belőle, és ekkor érdemes lehet lekicsinyíteni. Ebben az alfejezetben azt a problémát vizsgáljuk, hogy egy tömböt hogyan lehet dinamikusan

kiterjeszteni és összehúzni. Az ilyen tömböt dinamikus táblának nevezzük. Az amortizációs elemzés segítségével megmutatjuk, hogy a beszúrás és törlés amortizációs költsége csak $O(1)$, annak ellenére, hogy egy művelet tényleges költsége nagy, amikor egy kiterjesztést vagy összehúzást vált ki. Továbbá látni fogjuk, hogy hogyan lehet garantálni, hogy a dinamikus tábla nem használt része soha ne haladja meg a teljes méret egy meghatározott hányadát.

Feltesszük, hogy a dinamikus tábla a TÁBLÁBA-BESZÚR és TÁBLÁBÓL-TÖRÖL műveletet használja. A táblában tárolt objektumok mind ugyanakkora helyet foglalnak el, amelyet egységnyinek tekintünk. A TÁBLÁBA-BESZÚR művelet berak egy elemet a táblába, a TÁBLÁBÓL-TÖRÖL pedig kivesz egyet, és ezzel egy helyet felszabadít. Hogy a tábla adatszerkezetét milyen módszerrel szervezzük meg, lényegtelen. Használható a verem (10.1. alfejezet), a kupac (6. fejezet) vagy a hasító tábla (11. fejezet). De használhatunk tömböt vagy tömbök halmazát is, mint azt a 10.3. alfejezetben tettük.

Kényelmes lesz egy olyan fogalom használata, amit a hasítás elemzésekor (11. fejezet) vezettünk be. Egy nem üres T tábla $\alpha(T)$ ***feltöltési tényezője*** a benne tárolt elemek számának és a tábla méretének a hányadosa. Az üres tábla mérete 0 és feltöltési tényezője 1. Ha egy dinamikus tábla feltöltési tényezőjének van egy állandó alsó korlátja, akkor a tábla nem használt területe sohasem nagyobb, mint a teljes terület egy konstansszorosa.

Először egy olyan dinamikus táblát elemzünk, amiben csak beszúrás lehetséges. Azután az általánosabb esetet vizsgáljuk, amikor mind a beszúrás, mind a törlés megengedett.

17.4.1. A tábla kiterjesztése

Tegyük fel, hogy a tábla tárolását a helyek tömbjeként valósítjuk meg. A tábla betelik, ha minden helyét felhasználtuk, ami ekvivalens azzal, hogy a feltöltési tényezője 1.[1] Egyes szoftverkörnyezetek esetében, ha egy teli táblába kívánunk egy további elemet berakni, akkor az egyetlen lehetőség a hibával történő leállás. Feltesszük azonban, hogy a mi szoftverkörnyezetünk, mint számos modern környezet, rendelkezik egy olyan memóriakezelő rendszerrel, amely kérésre le tud foglalni és fel tud szabadítani blokkokat. Így amikor egy teli táblába szúrunk be egy elemet, ***ki tudjuk terjeszteni*** a táblát olyan módon, hogy a régi táblánál több hely legyen benne. Mivel megköveteljük, hogy a tábla mindig folytonosan helyezkedjen el a memóriában, a kiterjesztésnél le kell foglalnunk egy nagyobb tömböt az új táblának, majd át kell másolnunk az elemeket a régi táblából az új táblába.

Elterjedt heurisztika, hogy az új tábla kétszer akkora legyen, mint a régi. Ha csak beszúrásokat végzünk, akkor a tábla feltöltési tényezője mindig legalább 1/2, és a kihasználatlan terület sohasem haladja meg a tábla méretének felét.

A következő pszeudokódban feltesszük, hogy T a táblát jelentő objektum. A *tábla*$[T]$ mező a táblára mutató pointer. A *szám*$[T]$ a T-ben tárolt elemek száma, a *méret*$[T]$ a táblában lévő helyek száma. Kezdetben, amikor a tábla üres, *szám*$[T] =$ *méret*$[T] = 0$.

[1] Néhány helyzetben, ilyen például a nyílt címzésű hasító tábla, a táblát akkor is telinek kell tekintenünk, ha a feltöltési tényezője egy 1-nél határozottan kisebb állandó. Lásd a 17.4-1. gyakorlatot.

TÁBLÁBA-BESZÚR(T, x)

```
 1  if méret[T] = 0
 2    then foglaljunk le egy helyet tábla[T]-nek
 3         méret[T] ← 1
 4  if szám[T] = méret[T]
 5    then foglaljunk le 2 · méret[T] helyet új-táblá-nak
 6         tábla[T] valamennyi elemét szúrjuk be új-táblá-ba
 7         szabadítsuk fel tábla[T]-t
 8         tábla[T] ← új-tábla
 9         méret[T] ← 2 · méret[T]
10  szúrjuk be x-et tábla[T]-be
11  szám[T] ← szám[T] + 1
```

Látható, hogy két „beszúrás” eljárásunk van: az egyik a TÁBLÁBA-BESZÚR maga, a másik az ***elemi beszúrás*** a táblába a 6. és 10. sorban. A TÁBLÁBA-BESZÚR futási idejét az elemi beszúrások számában vizsgálhatjuk. Egy elemi beszúrás költsége 1. Feltesszük, hogy a TÁBLÁBA-BESZÚR tényleges futási ideje lineáris kifejezése az elemi beszúrás végrehajtási idejének, mivel a 2. sorban a kezdeti tábla lefoglalásának ideje állandó, az új tábla lefoglalásának (5. sor) és a régi tábla felszabadításának (7. sor) végrehajtási ideje pedig nem ad jelentős többletet a 6. sorban végzett másolásokhoz képest. Azt az eseményt, amikor az 5–9. sorokban található **then** ágat végrehajtjuk, ***kiterjesztésnek*** nevezzük.

Elemezzük egy eredetileg üres táblára alkalmazott n egymást követő TÁBLÁBA-BESZÚR műveletsorozatát. Mi az i-edik művelet c_i költsége? Ha van hely a táblában (vagy ez az első művelet), akkor $c_i = 1$, mivel csak egy elemi beszúrást kell végezni az eljárás 10. sorában. Ha azonban a tábla tele van, akkor kiterjesztést kell végezni és $c_i = i$: a régi tábla elemeinek átmásolása az új táblába az algoritmus 6. sorában $i - 1$ költséggel jár, és 1 költsége van még az új elem elhelyezésének a 10. sorban. Ha n műveletet hajtunk végre, akkor egy művelet a legrosszabb esetben $O(n)$ ideig tart, ami az n művelet teljes futási idejére a $O(n^2)$ felső korláthoz vezet.

Ez a korlát nem éles, mert az n TÁBLÁBA-BESZÚR művelet során a kiterjesztés csak ritkán fordul elő. Ugyanis az i-edik művelet csak akkor váltja ki a tábla kiterjesztését, ha $i-1$ kettőhatvány. Egy művelet amortizációs költsége csak $O(1)$, amint azt az összesítéses elemzéssel ki lehet mutatni. Az i-edik művelet költsége

$$c_i = \begin{cases} i, & \text{ha } i-1 \text{ kettőhatvány,} \\ 1, & \text{különben.} \end{cases}$$

Innen n TÁBLÁBA-BESZÚR műveletköltségére azt kapjuk, hogy

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \\ &< n + 2n \\ &= 3n, \end{aligned}$$

mert legfeljebb n db 1 költségű művelet van, és a többi művelet költsége mértani sorozatot alkot. Mivel n TÁBLÁBA-BESZÚR művelet teljes költsége legfeljebb $3n$, ezért egy művelet amortizációs költsége 3.

A könyvelési módszer segítségével megérthetjük, hogy miért kell az amortizációs költségnek 3-nak lennie. Úgy képzelhetjük, hogy minden művelet 3 elemi beszúrásért fizet: egy a saját maga elhelyezése a pillanatnyi táblában, egy másik az ő áthelyezése az új táblába, amikor a táblát ki kell terjeszteni, és a harmadik egy olyan elem áthelyezése, amelyet a tábla egy korábbi kiterjesztése alkalmával már áthelyeztünk. Például tegyük fel, hogy közvetlenül egy kiterjesztés után a tábla mérete m. Ekkor a táblában lévő elemek száma $m/2$, és a tábla nem tartalmaz hitelt. 3 tallért számolunk fel egy beszúrásért. Maga az elemi beszúrás 1 tallérba kerül. Egy másik tallért mint hitelt, ráhelyezünk erre az elemre. A harmadik tallért a már a táblában lévő $m/2$ elem valamelyikére helyezzük. A tábla betöltése $m/2 - 1$ beszúrást kíván, és így, amikor a tábla megtelik és benne m elem van, minden elemnek van egy tallérja, amivel a kiterjesztés ráeső részét ki tudja fizetni.

A potenciál módszer szintén felhasználható a TÁBLÁBA-BESZÚR művelet elemzésére, és a 17.4.2. pontban egy olyan TÁBLÁBÓL-TÖRÖL művelet megtervezésére fogjuk használni, amelynek szintén $O(1)$ lesz az amortizációs költsége. A Φ potenciálfüggvényt úgy definiáljuk, hogy közvetlenül egy kiterjesztés után 0 legyen, de a tábla méretével legyen egyenl ő akkorra, amikor a tábla megtelik – így ki lehet vele fizetni a következ ő kiterjesztés költségét. A függvény megválasztásának egyik lehet ősége:

$$\Phi[T] \;=\; 2 \cdot \mathit{szám}[T] - \mathit{méret}[T]. \tag{17.5}$$

Egy kiterjesztés után közvetlenül $\mathit{szám}[T] = \mathit{méret}[T]/2$, vagyis $\Phi(T) = 0$. Közvetlenül előtte pedig $\mathit{szám}[T] = \mathit{méret}[T]$, és így $\Phi(T) = \mathit{szám}[T]$, ahogy kívántuk. A potenciál kezdeti értéke 0, és mivel a tábla legalább félig mindig tele van, azaz $\mathit{szám}[T] \geq \mathit{méret}[T]/2$, ezért $\Phi(T)$ mindig nemnegatív. Tehát n TÁBLÁBA-BESZÚR művelet amortizációs költségének összege a tényleges költségek összegének fels ő korlátja.

Az elemzéshez bevezetjük a következ ő jelöléseket: az i-edik művelet után közvetlenül $\mathit{szám}_i$ a táblában tárolt elemek száma, $\mathit{méret}_i$ a tábla mérete, Φ_i a potenciálfüggvény értéke. Kezdetben $\mathit{szám}_0 = \mathit{méret}_0 = \Phi_0 = 0$.

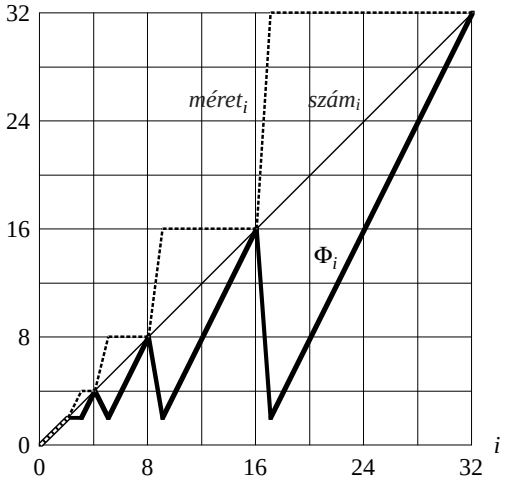
Ha az i-edik TÁBLÁBA-BESZÚR művelet nem váltja ki a tábla kiterjesztését, akkor $\mathit{méret}_i = \mathit{méret}_{i-1}$, és így a művelet amortizációs költsége

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot \mathit{szám}_i - \mathit{méret}_i) - (2 \cdot \mathit{szám}_{i-1} - \mathit{méret}_{i-1}) \\
&= 1 + (2 \cdot \mathit{szám}_i - \mathit{méret}_i) - (2 \cdot (\mathit{szám}_i - 1) - \mathit{méret}_i) \\
&= 3.
\end{aligned}$$

Ha az i-edik művelet kiváltja a tábla kiterjesztését, akkor $\mathit{méret}_i = 2 \cdot \mathit{méret}_{i-1}$, és $\mathit{méret}_{i-1} = \mathit{szám}_{i-1} = \mathit{szám}_i - 1$, ebből pedig $\mathit{méret}_i = 2 \cdot \mathit{szám}_i 1$. Ezért az amortizációs költség

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= \mathit{szám}_i + (2 \cdot \mathit{szám}_i - \mathit{méret}_i) - (2 \cdot \mathit{szám}_{i-1} - \mathit{méret}_{i-1}) \\
&= \mathit{szám}_i + (2 \cdot \mathit{szám}_i - (2 \cdot \mathit{szám}_i - 2)) - (2(\mathit{szám}_i - 1) - (\mathit{szám}_i - 1)) \\
&= \mathit{szám}_i + 2 - (\mathit{szám}_i - 1) \\
&= 3.
\end{aligned}$$

A 17.3. ábra $\mathit{szám}_i$, $\mathit{méret}_i$ és Φ_i értékét mutatja i függvényében. Figyeljük meg, hogy hogyan növekszik a potenciál értéke, hogy ki tudjuk fizetni a kiterjesztés költségét.



17.3. ábra. *n* TÁBLÁBA-BESZÚR műveletsorozatának hatása a táblában tárolt elemek *szám*$_i$ és helyek *méret*$_i$ számára és a $\Phi_i = 2 \cdot szám_i - méret_i$ potenciálra. Mindhárom mennyiséget közvetlenül az *i*-edik művelet után mérjük. A vékony vonal *szám*$_i$-t, a szaggatott vonal *méret*$_i$-t, a vastag vonal Φ_i-t ábrázolja. Figyeljük meg, hogy egy kiterjesztés előtt közvetlenül a potenciál értéke eléri a táblában tárolt elemek számát, és ezért ki tudjuk fizetni az elemek áthelyezését az új táblába. Ezután a potenciál 0-ra zuhan vissza, de azonnal 2-vel növekszik, amikor a kiterjesztést kiváltó elemet berakjuk a táblába.

17.4.2. Tábla kiterjesztés és összehúzás

A TÁBLÁBÓL-TÖRÖL művelet megvalósításához egyszerűen csak el kell távolítani a táblából a meghatározott elemet. Azonban sokszor kívánatos, hogy ***összehúzzuk*** a táblát, ha a feltöltési tényezője túl kicsivé válik, és így elkerüljük, hogy a kihasználatlan terület túl nagy legyen. A tábla összehúzása hasonló a tábla kiterjesztéséhez: amikor az elemek száma túlzottan lecsökken, akkor egy új, kisebb táblát foglalunk le, és a régi tábla valamennyi elemét az új táblába másoljuk. A régi tábla által elfoglalt terület felszabadítható azáltal, hogy visszaadjuk a memóriakezelő rendszernek. Alapvetően két tulajdonságot szeretnénk megőrizni:

- a dinamikus tábla feltöltési tényezőjének legyen pozitív állandó alsó korlátja, és
- egy táblaművelet amortizációs költségének legyen pozitív állandó felső korlátja.

Feltesszük, hogy a költséget az elemi beszúrások és törlések száma egy konstans erejéig meghatározza.

A kiterjesztés és összehúzás természetes stratégiája, hogy a tábla méretét megkettőzzük, ha egy teli táblába kell egy elemet beszúrni, és megfelezzük, ha egy törlés után a felénél kisebb mértékig lenne feltöltve. Ez a stratégia biztosítja, hogy a feltöltési tényező sohasem esik 1/2 alá, de sajnos azzal járhat, hogy az amortizációs költség elég magas lesz. Tekintsük az események következő lehetséges lefolyását. Egy T táblán n műveletet hajtunk végre, ahol n egy kettőhatvány. Az első $n/2$ művelet beszúrás, aminek a teljes költsége az előző elemzés szerint $\Theta(n)$. Ennek a beszúrási sorozatnak a végén $szám[T] = méret[T] = n/2$. A második $n/2$ művelet esetén a következő sorozatot hajtjuk végre:

B, T, T, B, B, T, T, B, B, ...,

ahol B a beszúrást, T a törlést jelenti. Az első beszúrás kiváltja a tábla méretének n-re való kiterjesztését. A következő két törlés után a táblát össze kell húzni $n/2$-re. A két következő beszúrás ismét a tábla kiterjesztését okozza és így tovább. Minden kiterjesztés és összehúzás költsége $\Theta(n)$, és $\Theta(n)$ van belőlük. Így n művelet teljes költsége $\Theta(n^2)$, és ezért egy művelet amortizációs költsége $\Theta(n)$.

Nyilvánvaló, mi okozza a nehézséget, ha ezt a stratégiát alkalmazzuk: kiterjesztés után nem végzünk elég törlést ahhoz, hogy az összehúzást ki tudjuk fizetni. Hasonlóan, összehúzás után nem elég a beszúrások száma a kiterjesztés finanszírozására.

Javíthatjuk a stratégiát azáltal, ha megengedjük, hogy a feltöltési tényező 1/2 alá csökkenjen. Továbbra is megkettőzzük a tábla méretét, ha egy teli táblába kellene beszúrni egy elemet, de csak akkor felezzük meg, ha kevesebb mint a negyedéig lenne tele, ahelyett, hogy ez a határ a fele volna, mint előzőleg. Ezért 1/4 a feltöltési tényező alsó korlátja. Emögött az a gondolat húzódik meg, hogy egy kiterjesztés után a feltöltési tényező 1/2. Tehát a tábla elemeinek felét törölni kell ahhoz, hogy egy összehúzásra sor kerüljön, hiszen ezt csak akkor csináljuk, amikor a feltöltési tényező 1/4 alá esik. Hasonlóképpen, egy összehúzás után a feltöltési tényező szintén 1/2, tehát az elemek számát meg kell kettőzni ahhoz, hogy egy kiterjesztés szükségessé váljon, mert ez csak akkor következik be, ha a feltöltési tényező az 1-et meghaladná.

Elhagyjuk a TÁBLÁBÓL-TÖRÖL kódját, hiszen az a TÁBLÁBA-BESZÚR kódjához hasonló. Az elemzés kényelmesebbé tétele kedvéért feltesszük, hogy ha a táblát az összes elem elhagyja, akkor a tábla által foglalt helyet felszabadítjuk, azaz ha $szám[T] = 0$, akkor $méret[T] = 0$.

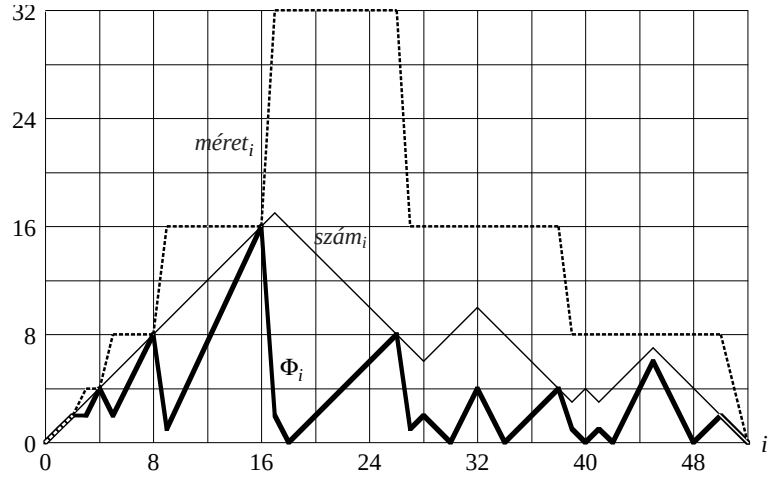
Most már fel tudjuk használni a potenciál módszert n TÁBLÁBA-BESZÚR és TÁBLÁBÓL-TÖRÖL műveletből álló sorozat költségének elemzésére. Azzal kezdjük, hogy definiáljuk a Φ potenciálfüggvényt, ami 0 közvetlenül egy kiterjesztés vagy összehúzás után. A feltöltési tényező, amely egy nem üres tábla esetén $\alpha(T) = szám[T]/méret[T]$, 1/4 és 1 között fog mozogni. Mivel üres tábla esetén $szám[T] = méret[T] = 0$ és $\alpha(T) = 1$, mindig igaz, hogy $szám[T] = \alpha(T) \cdot méret[T]$, függetlenül attól, hogy a tábla üres-e. A következő potenciálfüggvényt fogjuk használni:

$$\Phi(T) = \begin{cases} 2 \cdot szám[T] - méret[T], & \text{ha } \alpha(T) \geq 1/2, \\ méret[T]/2 - szám[T], & \text{ha } \alpha(T) < 1/2. \end{cases} \tag{17.6}$$

Vegyük észre, hogy az üres tábla potenciálja 0, és a potenciál sohasem negatív. Így a sorozat Φ-re vonatkozó teljes amortizációs költsége a tényleges költség felső korlátja.

Mielőtt a pontos elemzést elvégeznénk, egy kitérőt teszünk, hogy a potenciálfüggvény néhány tulajdonságát megvizsgáljuk. Amikor a feltöltési tényező 1/2, akkor a potenciál 0. Ha a tényező 1, akkor $méret[T] = szám[T]$, amiből $\Phi[T] = szám[T]$, és így a potenciálból kifizethető a kiterjesztés, ha egy elemet be kell szúrni. Ha a feltöltési tényező 1/4, akkor $méret[T] = 4 \cdot szám[T]$, ami ismét maga után vonja, hogy $\Phi[T] = szám[T]$, és ezért a potenciálból ki lehet fizetni az összehúzást, ha egy elemet törölni kell. A 17.4. ábra mutatja, hogy hogyan viselkedik a potenciál egy műveletsorozat végrehajtásakor.

n TÁBLÁBA-BESZÚR és TÁBLÁBÓL-TÖRÖL műveletből álló sorozat elemzéséhez bevezetjük a következő jelöléseket: az i-edik művelet költsége c_i, Φ-re vonatkozó amortizációs költsége $\widehat{c_i}$, valamint az i-edik művelet után közvetlenül $szám_i$ a táblában tárolt elemek száma, $méret_i$ a tábla mérete, Φ_i a potenciálfüggvény értéke, α_i pedig a feltöltési tényező. Kezdetben $szám_0 = méret_0 = \Phi_0 = 0$ és $\alpha_0 = 1$.



17.4. ábra. n TÁBLÁBA-BESZÚR és TÁBLÁBÓL-TÖRÖL műveletből álló sorozat hatása a táblában tárolt elemek *szám*$_i$ és helyek *méret*$_i$ számára és a

$$\Phi_i = \begin{cases} 2 \cdot szám_i - méret_i, & \text{ha } \alpha_i \geq 1/2, \\ méret_i/2 - szám_i, & \text{ha } \alpha_i < 1/2. \end{cases}$$

potenciálra. Mindhárom mennyiséget közvetlenül az i-edik művelet után mérjük. A vékony vonal *szám*$_i$-t, a szaggatott vonal *méret*$_i$-t, a vastag vonal Φ_i-t ábrázolja. Figyeljük meg, hogy közvetlenül a kiterjesztés előtt a potenciál értéke eléri a táblában tárolt elemek számát és ezért ki tudjuk fizetni az elemek áthelyezését az új táblába. Hasonlóképpen közvetlenül egy összehúzás előtt a potenciál értéke eléri a táblában tárolt elemek számát.

Azzal az esettel kezdjük, amikor az i-edik művelet egy TÁBLÁBA-BESZÚR. Ha $\alpha_{i-1} \geq 1/2$, akkor az elemzés azonos azzal, amit a tábla kiterjesztésénél végeztünk a 17.4.1. pontban. Függetlenül attól, hogy sor kerül-e kiterjesztésre, a $\widehat{c_i}$ amortizációs költség legfeljebb 3. Ha $\alpha_{i-1} < 1/2$, akkor a kiterjesztés nem kerülhet szóba, hiszen ahhoz $\alpha_{i-1} = 1$ kell. Ha még $\alpha_i < 1/2$ is igaz, akkor az i-edik művelet amortizációs költsége

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \left(\frac{méret_i}{2} - szám_i\right) - \left(\frac{méret_{i-1}}{2} - szám_{i-1}\right) \\ &= 1 + \left(\frac{méret_i}{2} - szám_i\right) - \left(\frac{méret_i}{2} - (szám_i - 1)\right) \\ &= 0. \end{aligned}$$

Ha $\alpha_{i-1} < 1/2$, de $\alpha_i \geq 1/2$, akkor

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot szám_i - méret_i) - \left(\frac{méret_{i-1}}{2} - szám_{i-1}\right) \\ &= 1 + (2(szám_{i-1} + 1) - méret_{i-1}) - \left(\frac{méret_{i-1}}{2} - szám_{i-1}\right) \\ &= 3 \cdot szám_{i-1} - \frac{3}{2} méret_{i-1} + 3 \end{aligned}$$

$$\begin{aligned} &= 3\alpha_{i-1}méret_{i-1} - \frac{3}{2}méret_{i-1} + 3 \\ &< \frac{3}{2}méret_{i-1} - \frac{3}{2}méret_{i-1} + 3 \\ &= 3. \end{aligned}$$

Tehát a TÁBLÁBA-BESZÚR művelet amortizációs költsége legfeljebb 3.

Most térjünk át arra az esetre, amikor az i-edik művelet egy TÁBLÁBÓL-TÖRÖL. Ebben az esetben $szám_i = szám_{i-1} - 1$. Ha $\alpha_{i-1} < 1/2$, akkor figyelembe kell venni, hogy sor kerül-e összehúzásra. Ha nem, akkor $méret_i = méret_{i-1}$, és a művelet amortizációs költsége

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \left(\frac{méret_i}{2} - szám_i\right) - \left(\frac{méret_{i-1}}{2} - szám_{i-1}\right) \\ &= 1 + \left(\frac{méret_i}{2} - szám_i\right) - \left(\frac{méret_i}{2} - (szám_i + 1)\right) \\ &= 2. \end{aligned}$$

Ha $\alpha_{i-1} < 1/2$ és az i-edik művelet kiváltja a tábla összehúzását, akkor $c_i = szám_i + 1$, hiszen egy elemet törlünk és $szám_i$ elemet átrakunk az új táblába. Ekkor $méret_i/2 = méret_{i-1}/4 = szám_{i-1} = szám_i + 1$ és a művelet amortizációs költsége

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= (szám_i + 1) + \left(\frac{méret_i}{2} - szám_i\right) - \left(\frac{méret_{i-1}}{2} - szám_{i-1}\right) \\ &= (szám_i + 1) + ((szám_i + 1) - szám_i) - ((2 \cdot szám_i + 2) - (szám_i + 1)) \\ &= 1. \end{aligned}$$

Ha az i-edik művelet TÁBLÁBÓL-TÖRÖL és $\alpha_{i-1} \geq 1/2$, akkor az amortizációs költségnek is van egy állandó felső korlátja. Ennek az esetnek a vizsgálata a 17.4-2. gyakorlatra marad.

Összefoglalva, mivel minden művelet amortizációs költségét felülről korlátozza egy állandó, ezért a dinamikus tábla bármely n műveletének tényleges költsége $O(n)$.

Gyakorlatok

17.4-1. Tegyük fel, hogy egy dinamikus, nyílt címzésű hasító táblát akarunk létrehozni. Miért kell a táblát telinek tekintenünk olyankor, amikor a feltöltési tényező elér egy bizonyos, 1-nél határozottan kisebb α értéket? Írjuk le röviden, hogy hogyan lehet egy dinamikus, nyílt címzésű hasító táblába beszúrni olyan módon, hogy a műveletenkénti amortizációs költség várható értéke $O(1)$ legyen. Miért nem feltétlenül $O(1)$ a beszúrásonkénti költség várható értéke minden beszúrás esetén?

17.4-2. Mutassuk meg, hogy ha a dinamikus tábla i-edik művelete egy TÁBLÁBÓL-TÖRÖL és $\alpha_{i-1} \geq 1/2$, akkor a művelet (17.6) potenciálfüggvény szerinti amortizációs költségét egy konstans felülről korlátozza.

17.4-3. Tegyük fel, hogy a helyett a stratégia helyett, hogy a tábla méretét a felére húzzuk össze, amikor a feltöltési tényezője 1/4 alá esik, azt alkalmazzuk, hogy a méretét a 2/3 részére csökkentjük, amikor a feltöltési tényező 1/3-nál kisebb lesz. A

$$\Phi([T]) = |2 \cdot szám[T] - méret[T]|$$

potenciálfüggvényt használva mutassuk meg, hogy ekkor TÁBLÁBÓL-TÖRÖL művelet amortizációs költségének van állandó felső korlátja.

Feladatok

17-1. Fordított bitsorrendű bináris számláló

A 30. fejezet tárgyalja a gyors Fourier-transzformációnak vagy FFT-nek nevezett fontos módszert. Az FFT algoritmus első lépése, hogy meghatározza egy $n = 2^k$ hosszú (k nemnegatív egész) $A[0\,..\,n-1]$ vektor ***fordított bitsorrendű permutációját.*** Ez a permutáció azokat az elemeket cseréli fel, amelyek indexeit kettes számrendszerben felírva a biteket éppen fordított sorrendben kapjuk.

Egy a indexet egy k hosszú $(a_{k-1}, a_{k-2}, \ldots, a_0)$ bitsorozattal írhatunk le, ahol $a = \sum_{i=0}^{k-1} a_i 2^i$. Definíció szerint

$$\text{ford}_k((a_{k-1}, a_{k-2}, \ldots, a_0)) = (a_0, a_1, \ldots, a_{k-1}),$$

és így

$$\text{ford}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

Például, ha $n = 16$ (vagy ami ezzel ekvivalens: $k = 4$), akkor $\text{ford}_4(3) = 12$, mivel három 4 bites felírása 0011, aminek a megfordítása 1100, ez pedig tizenkét 4 bites felírása.

a. Adott egy $\Theta(k)$ ideig futó ford_k függvény. Írjunk egy olyan algoritmust, ami $O(nk)$ idő alatt készíti el egy $n = 2^k$ hosszú vektor fordított bitsorrendű permutációját.

Amortizációs elemzésen alapuló algoritmust használhatunk a fordított bitsorrendű permutációhoz szükséges futási idő javítására. Egy „fordított bitsorrendű számlálót” és FORDÍTOTT-BITSORRENDŰ-NÖVEL eljárást működtetünk. Az utóbbi a $\text{ford}_k(\text{ford}_k(a) + 1)$ értéket adja eredményül, ha a fordított bitsorrendű számláló értéke a. Ha például $k = 4$, és a fordított bitsorrendű számláló 0-ból indul, akkor a FORDÍTOTT-BITSORRENDŰ-NÖVEL eljárás ismételt hívásával a következő sorozat áll elő:

$$0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots .$$

b. Tegyük fel, hogy a számítógépünk k bites szavakat tárol, és azokon egységnyi idő alatt tud elvégezni olyan manipulációkat, mint tetszőleges hellyel balra vagy jobbra való eltolás, bitenkénti konjunkció és diszjunkció stb. Adjuk meg a FORDÍTOTT-BITSORRENDŰ-NÖVEL eljárás egy olyan megvalósítását, amely lehetővé teszi a fordított bitsorrendű permutáció $O(n)$ idejű meghatározását egy n elemű tömb esetében.

c. Tegyük fel, hogy egységnyi idő alatt csak egy hellyel tudunk egy szót balra vagy jobbra elcsúsztatni. Ebben az esetben is lehetséges a fordított bitsorrendű permutáció $O(n)$ idejű előállítása?

17-2. Dinamikus bináris keresés

A bináris keresés egy rendezett vektorban logaritmikus ideig tart, de egy új elem beszúrása lineáris a vektor méretében. Megjavíthatjuk az utóbbit azáltal, ha az elemeket több rendezett vektorban tároljuk.

Tegyük fel, hogy a KERES és BESZÚR műveletet akarjuk megvalósítani egy n elemű halmazon. Legyen $k = \lceil \log(n+1) \rceil$, és legyenek n kettes számrendszerbeli jegyei $(n_{k-1}, n_{k-2}, \ldots, n_0)$. Ekkor k rendezett vektorunk van, $A_0, A_1, \ldots, A_{k-1}$, melyekben az elemeket tároljuk. Az A_i vektor hossza $i = 0, 1, \ldots, k-1$ esetén 2^i. Minden vektor vagy tele van, vagy üres, attól függően, hogy $n_i = 1$ vagy $n_i = 0$. Tehát a k vektorban tárolt összes elem száma $\sum_{i=0}^{k-1} n_i 2^i = n$. Habár a vektorok egyenként rendezettek, a különböző vektorokban található elemek között nincs összefüggés.

a. Írjuk le, hogy ebben az adatszerkezetben hogyan kell végrehajtani a KERES műveletet. Elemezzük a futási időt a legrosszabb esetben.

b. Hogyan lehet ebben az adatszerkezetben beszúrni egy új elemet? Elemezzük a legrosszabb esetet és az amortizációs futási időt.

c. Vizsgáljuk meg a TÖRÖL művelet megvalósításának módját.

17-3. Amortizációs kiegyensúlyozott fák

Tekintsünk egy közönséges bináris keresőfát, amelyet minden x csúcsánál kiegészítettünk egy *méret*$[x]$ mezővel, amelyben az x gyökerű részfában tárolt kulcsok száma található. Legyen α egy állandó az $1/2 \leq \alpha < 1$ tartományban. Azt mondjuk, hogy egy adott x csúcs ***α-kiegyensúlyozott,*** ha

$$méret[bal[x]] \leq \alpha \cdot méret[x]$$

és

$$méret[jobb[x]] \leq \alpha \cdot méret[x].$$

A teljes fa ***α-kiegyensúlyozott,*** ha minden csúcsa α-kiegyensúlyozott. G. Varghese javasolta a következő amortizációs módszert a fa kiegyensúlyozottságának fenntartására.

a. Az 1/2-kiegyensúlyozott fa bizonyos értelemben annyira kiegyensúlyozott, amennyire csak lehet. Adott egy x csúcs egy tetszőleges bináris keresőfában. Mutassuk meg, hogy hogyan lehet az x gyökerű részfát újra felépíteni, hogy 1/2-kiegyensúlyozottá váljon. Az algoritmus futási ideje legyen $\Theta(méret[x])$, és $O(méret[x])$ kiegészítő memóriát használjon.

b. Mutassuk meg, hogy egy keresés az n csúcsú α-kiegyensúlyozott bináris keresőfában a legrosszabb esetben $O(\log n)$ ideig tart.

A feladat hátralévő részében feltesszük, hogy α határozottan nagyobb 1/2-nél. Tegyük fel, hogy a BESZÚR és TÖRÖL ugyanúgy történik, mint az az n csúcsú bináris keresőfák esetén szokásos, azzal az egy kivétellel, hogy minden ilyen művelet után, ha egy részfa már nem α-kiegyensúlyozott, akkor a legmagasabb szinten lévő gyökérrel rendelkező ilyen részfát átrendezzük úgy, hogy 1/2-kiegyensúlyozott legyen.

A potenciál módszert felhasználva elemezzük ezt az átrendezési eljárást. A T bináris keresőfa egy x csúcsára legyen

$$\Delta(x) = \left| méret\,[bal[x]] - méret\,[jobb[x]] \right|,$$

T potenciálfüggvénye pedig legyen

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

ahol c egy α-tól függő elég nagy szám.

c. Mutassuk meg, hogy bármely bináris keresőfa potenciálja nemnegatív, és az 1/2-kiegyensúlyozott fa potenciálja 0.

d. Tegyük fel, hogy egy m csúcsú részfa átrendezése m egység potenciállal kifizethető. Milyen nagynak kell lennie c-nek α-ban kifejezve, hogy egy nem α kiegyensúlyozott fa átrendezése $O(1)$ amortizációs ideig tartson?

e. Mutassuk meg, hogy egy csúcs beszúrása, illetve törlése egy n csúcsú α-kiegyensúlyozott fa esetén $O(\log n)$ amortizációs ideig tart.

17-4. Piros-fekete fák átépítésének költsége

A piros-fekete fáknak négy olyan alapvető műveletük van, melyek ***módosítják a szerkezetet:*** csúcsok beszúrása, csúcsok törlése, forgatás és a színek változtatása. Láttuk, hogy a PF-FÁBA-BESZÚR és PF-FÁBÓL-TÖRÖL műveleteknek csak $O(1)$ csúcs beszúrására, törlésére, ill. forgatásra van szüksége ahhoz, hogy a piros-fekete tulajdonságok megmaradjanak, de ennél jóval több színváltozással járhatnak.

a. Írjunk le egy olyan n csúcsú piros-fekete fát, amelyben az $(n + 1)$-edik csúcsra meghívott PF-FÁBA-BESZÚR művelet $\Omega(\log n)$ színváltozással jár. Utána írjunk le egy olyan n csúcsú piros-fekete fát, amelyben egy adott csúcsra meghívott PF-FÁBÓL-TÖRÖL művelet $\Omega(\log n)$ színváltozással jár.

Megmutatjuk, hogy bár a legrosszabb esetben a műveletenkénti színváltozások száma logaritmikus is lehet, egy eredetileg üres piros-fekete fára meghívott m PF-FÁBA-BESZÚR és PF-FÁBÓL-TÖRÖL művelet a legrosszabb esetben is legfeljebb $O(m)$ szerkezeti módosítással jár.

b. A PF-FÁBA-BESZÚR-JAVÍT és PF-FÁBÓL-TÖRÖL-JAVÍT kódjának fő ciklusában előforduló esetek közül némelyik ***befejező:*** ha ide jutunk, a ciklus állandó számú további művelet elvégzése után befejeződik. Állapítsuk meg a PF-FÁBA-BESZÚR-JAVÍT és PF-FÁBÓL-TÖRÖL-JAVÍT eseteiről, melyek befejezőek és melyek nem. (*Útmutatás.* Tekintsük a 13.5., 13.6. és 13.7. ábrákat.)

Először azt elemezzük, hogy milyen szerkezeti módosítások történnek, ha csak beszúrásaink vannak. Legyen T egy piros-fekete fa, és jelöljük $\Phi(T)$-vel a T-beli piros csúcsok számát. Tegyük fel, hogy egy egységnyi potenciállal tudunk fizetni a PF-FÁBA-BESZÚR-JAVÍT három esetének bármelyike által végrehajtott szerkezeti módosításokért.

c. Legyen T' az a fa, melyet úgy kapunk T-ből, hogy a PF-FÁBA-BESZÚR-JAVÍT első esetét alkalmazzuk. Lássuk be, hogy $\Phi(T') = \Phi(T) - 1$.

d. Egy csúcs beszúrása egy piros-fekete fába a PF-FÁBA-BESZÚR művelettel három részre tagolható. Soroljuk fel, hogy milyen szerkezeti módosítások és potenciál változások történnek a PF-FÁBA-BESZÚR 1–16. sorában, a PF-FÁBA-BESZÚR-JAVÍT nem befejező, ill. befejező eseteiben.

e. A (d) pont felhasználásával lássuk be, hogy bármely PF-FÁBA-BESZÚR meghívásával járó szerkezeti módosítások amortizációs száma $O(1)$.

Most azt szeretnénk bebizonyítani, hogy akkor is $O(m)$ szerkezeti módosítás történik, ha beszúrások és törlések is vannak. Definiáljuk minden x csúcsra a következő függvényt:

$$w(x) = \begin{cases} 0, & \text{ha } x \text{ piros,} \\ 1, & \text{ha } x \text{ fekete és nincs piros gyereke,} \\ 0, & \text{ha } x \text{ fekete és egy piros gyereke van,} \\ 2, & \text{ha } x \text{ fekete és két piros gyereke van.} \end{cases}$$

Új definíciónk egy T piros-fekete fa potenciáljára legyen

$$\Phi(T) = \sum_{x \in T} w(x),$$

és legyen T' az a fa, melyet a PF-FÁBA-BESZÚR-JAVÍT vagy PF-FÁBÓL-TÖRÖL-JAVÍT egy nem befejező esetének alkalmazásával kapunk T-ből.

f. Mutassuk meg, hogy $\Phi(T') \leq \Phi(T) - 1$ a PF-FÁBA-BESZÚR-JAVÍT minden nem befejező esetében. Lássuk be, hogy a PF-FÁBA-BESZÚR-JAVÍT bármely hívásakor történő szerkezeti módosítások amortizációs száma $O(1)$.

g. Mutassuk meg, hogy $\Phi(T') \leq \Phi(T) - 1$ a PF-FÁBÓL-TÖRÖL-JAVÍT minden nem befejező esetében. Lássuk be, hogy a PF-FÁBÓL-TÖRÖL-JAVÍT bármely hívásakor történő szerkezeti módosítások amortizációs száma $O(1)$.

h. Fejezzük be annak bizonyítását, hogy m PF-FÁBA-BESZÚR és PF-FÁBÓL-TÖRÖL művelet a legrosszabb esetben $O(m)$ szerkezeti módosítással jár.

Megjegyzések a fejezethez

Összesítéses elemzést Aho, Hopcroft és Ullman [5] használt. Tarjan [293] áttekinti az amortizációs elemzés könyvelési és potenciál módszerét, és számos alkalmazást említ. Szerinte a könyvelési módszert több szerző is megtalálta. Ezek között említi M. R. Brown, R. E. Tarjan, S. Huddleston és K. Mehlhorn nevét. A potenciál módszer felfedezőjének D. D. Sleatort tartja. Az „amortizációs” kifejezést D. D. Sleator és R. E. Tarjan vezette be.

Bizonyos problémáknál a potenciálfüggvények jól használhatók alsó korlátok bizonyítására is. A probléma minden állapotára definiálunk egy potenciálfüggvényt, mely az állapothoz egy valós számot rendel. Ezután meghatározzuk a kezdeti állapot Φ_{kezd} és a végállapot $\Phi_{vég}$ potenciálját, valamint az egy lépésben történő maximális potenciálváltozást, $\Delta\Phi_{max}$-ot. A lépések száma ezekkel kifejezve legalább $\left|\Phi_{vég} - \Phi_{kezd}\right| / \Delta\Phi_{max}$. A potenciálfüggvények I/O bonyolultság alsó korlátjának bizonyításához való használatára Cormen [71], Floyd [91], ill. Aggarwal és Vitter [4] cikkeiben találhatunk példát. Krumme, Cybenko és Venkataraman [194] ***pletykálás*** (egy gráf minden csúcsából minden csúcsba el kell juttatni egy egyedi üzenetet) alsó korlátjának bizonyítására alkalmaztak potenciálfüggvényeket.

V. Fejlett adatszerkezetek

Bevezetés

Ebben a részben visszatérünk azoknak az adatszerkezeteknek a vizsgálatához, amelyek a dinamikus halmazok műveleteire jól alkalmazhatók, de ezt most magasabb szinten végezzük, mint a III. részben. Például két fejezetben is nagy szerepe lesz a 17. fejezetben tanulmányozott amortizált elemzési módszereknek.

A 18. fejezetben bevezetjük a B-fák fogalmát: ezek olyan kiegyensúlyozott keresőfák, amelyeket kimondottan úgy terveztek, hogy mágneslemezeken hatékonyan tárolhatók legyenek. Mivel a mágneslemezek működése sokkal lassabb, mint a véletlen-hozzáférésű memóriáké, a B-fák hatékonyságát nemcsak azzal mérjük, hogy mennyi számolási időt igényelnek a dinamikus halmazon végzett műveletek, hanem a végrehajtott lemezműveletek számát is figyelembe vesszük. A lemezhozzáférések száma a B-fák mindegyik műveletében a B-fa magasságával növekszik, de a B-fa magasságát a B-fa műveletek alacsony szinten tartják.

A 19. és 20. fejezetekben az összefésülhető kupacok olyan megvalósításait adjuk meg, amelyekben a BESZÚR, a MIN, a MINIMUMOT-KIVÁG és az EGYESÍT [2] műveletek könnyen végrehajthatók. Az EGYESÍT művelet egyesít, vagy összefésül két kupacot. Az ezekben a fejezetekben vizsgált adatszerkezetekre ezeken kívül a TÖRÖL és a KULCSOT-CSÖKKENT művelet is alkalmazható.

A binomiális kupacok, amelyekkel a 19. fejezetben foglalkozunk, ezeket a műveleteket legrosszabb esetben is végrehajtják $O(\lg n)$ idő alatt, ahol n a bemenő kupac elemszáma (az EGYESÍT esetén a két bemenő kupac elemszámának az összege). Ha az EGYESÍT műveletet hajtjuk végre, a binomiális kupacok sokkal jobbak, mint a 6. fejezetben tárgyalt bináris kupacok, hiszen két bináris kupac egyesítésére a legrosszabb esetben $\Theta(n)$ az időigény.

A 20. fejezetben a Fibonacci-kupacokkal foglalkozunk. Ezek, legalábbis elméleti szempontból, tovább javítják a binomiális kupacokat. A Fibonacci-kupacok hatékonyságának mérésére amortizált időkorlátokat használunk. A BESZÚR, a MIN és az EGYESÍT műveletekre a Fibonacci-kupacokon csak $O(1)$ az aktuális és amortizált időkorlát, a MINIMUMOT-KIVÁG és a TÖRÖL műveletek amortizált időigénye pedig $O(\lg n)$. A Fibonacci-kupacok legfontosabb előnye azonban az, hogy a KULCSOT-CSÖKKENT művelet amortizált ideje csak $O(1)$. A KULCSOT-CSÖKKENT művelet alacsony amortizált időigénye az, ami miatt a gráfproblé-

[2] A 10-2. feladatban már definiáltunk egy összefésülhető kupacot a MINIMUM és a MINIMUMOT-KIVÁG algoritmusokra, ezért ezt a kupacot **összefésülhető min-kupacnak** is nevezhetjük. Hasonlóan, a MAXIMUM és MAXIMUMOT-KIVÁG műveletekre megadható kupacot ***összefésülhető max-kupacnak*** is nevezhetjük. Ha másképp nem jelöljük, összefésülhető kupacon mindig összefésülhető min-kupacot fogunk érteni.

mák megoldásaiban néhány aszimptotikusan leggyorsabb algoritmus Fibonacci-kupacokat használ.

Végül a 21. fejezetben diszjunkt halmazokra adunk adatszerkezeteket. Egy n elemű univerzumot dinamikus halmazokra bontunk fel. El őször mindegyik elem a saját egyelemű halmazába kerül. Az EGYESÍT művelet egyesít két halmazt, és a HALMAZT-KERES függvény meghatározza azt a halmazt, amelyben az adott pillanatban a keresett elem benne van. Ha a halmazokat egy egyszerű gyökeres fával ábrázoljuk, akkor egy meglep ően gyors algoritmust kapunk: egy m műveletb ől álló sorozat id őigénye $O(m\alpha(n))$, ahol $\alpha(n)$ egy rendkívül lassan növekedő függvény – $\alpha(n)$ legfeljebb 4 minden elképzelhet ő alkalmazásban. Ezt az időkorlátot adó amortizált elemzés olyan bonyolult, mint amilyen egyszerű az adatszerkezet.

Az ebben a részben tárgyalt adatszerkezeteken kívül, természetesen, vannak más „fejlett” adatszerkezetek is. Ilyen fejlett adatszerkezetek a következ ők:

- ***Dinamikus fák,*** amelyeket Sleator és Tarjan [281] definiált, és Tarjan [292] tanulmányozott, diszjunkt gyökeres fák erdejéb ől állnak. Mindegyik fa minden egyes éléhez egy valós értékű költség van hozzárendelve. A dinamikus fákra olyan keres ő függvények alkalmazhatók, amelyek a szül ők, a gyökércsúcsok, él-költségek, és egy adott pontból a gyökércsúcsba vezető legkisebb költségű út megkeresésére szolgálnak. A fákat a következő műveletekkel lehet átalakítani: élek átvágása, egy adott csúcsból a gyökérbe vezető út élei költségének megváltoztatása, egy gyökérnek egy másik fához történ ő hozzákapcsolása, egy fa egy belső csúcsának a fa gyökércsúcsává alakítása. A dinamikus fák egyik megvalósításában ezeknek a műveleteknek az amortizált id őkorlátja $O(\lg n)$; egy bonyolultabb megvalósításban a legrosszabb esetre az $O(\lg n)$ id őkorlátot kapták. Dinamikus fákat használnak néhány aszimptotikusan leggyorsabb hálózati adatáram algoritmusban.
- ***Ferde fák,*** amelyeket Sleator és Tarjan [282] fejlesztett ki, és Tarjan [292] tanulmányozta őket. Ezek a bináris keresőfáknak olyan alakjai, amelyekre a szabványos fakeresési algoritmusok $O(\lg n)$ amortizált id őben futnak. A ferde fák egyik alkalmazása leegyszerűsíti a dinamikus fákat.
- A ***megmaradó*** adatszerkezetek az adatszerkezet korábbi változataira is megengednek lekérdezéseket, és néha módosításokat is. Driscoll, Sarnak, Sleator és Tarjan [82] ad olyan eljárásokat, amelyek összekapcsolt adatszerkezeteket kis id ő- és memóriaköltséggel megmaradó adatszerkezetekké alakítanak. A 13-1. feladatban egy egyszerű példát találhatunk a megmaradó dinamikus halmazokra.
- Számos adatszerkezet lehetővé tesz egy gyorsabb implementációt a kulcsok egy korlátozott univerzumára vonatkozó szótár (BESZ ÚR, TÖRÖL és KERES) műveletek számára. Az ezekből a korlátokból adódó el őnyök kihasználásával jobb legrosszabb aszimptotikus futási időket érnek el, mint az összehasonlításokon alapuló adatszerkezetek. Van Emde Boas [301] definiált egy olyan adatszerkezetet, amelyre a MINIMUM, MAXIMUM, BESZÚR, TÖRÖL, KERES, MINIMUMOT-KIVÁG, MAXIMUMOT-KIVÁG, MEGELŐZŐ és KÖVETŐ műveletek id őigénye a legrosszabb esetben $O(\lg\lg n)$, azzal a feltétellel, hogy a kulcsok univerzuma az $\{1, 2, \ldots, n\}$ halmaz. Fredman és Willard bevezette az ***egyesített fákat*** [99], amelyek az első olyan adatszerkezetek voltak, amelyek gyorsabb szótár műveleteket tettek lehetővé, ha az univerzumot az egész számokra korlátozzuk. Megmutatták, hogy hogyan lehet ezeket a műveleteket $O(\lg n / \lg\lg n)$ id ővel implementálni. Számos

későbbi adatszerkezet, mint például az ***exponenciális keresőfa*** [16] szintén jobb korlátokat ad néhány, vagy akár az összes szótárműveletre, ezekkel az adatszerkezetekkel ebben a könyvben az egyes fejezetekhez fűzött megjegyzésekben találkozunk.

- A ***dinamikus gráf adatszerkezetekben*** olyan műveletek is végrehajthatók, amelyek azt is lehetővé teszik, hogy a csúcsokat vagy éleket beszúró vagy törlő műveletek a gráf struktúráját is megváltoztathassák. Ilyen műveletek például a csúcsok összekapcsolása [144], élek összekapcsolása, minimális feszítő fák [143], kétszeres összekapcsolás és a tranzitív lezárás [142].

A könyv fejezeteihez fűzött megjegyzésekben további adatszerkezeteket találunk.

18. B-fák

A B-fák olyan kiegyensúlyozott keresőfák, amelyeket úgy terveztek, hogy hatékonyan lehessen alkalmazni őket a mágneslemezeken vagy más közvetlen hozzáférésű másodlagos tárolóberendezéseken. A B-fák hasonlók a piros-fekete fákhoz (13. fejezet), de a lemezen kevesebb beviteli és kiviteli műveletet igényelnek. Sok adatbázisrendszer használja a B-fákat vagy a B-fák változatait az információ tárolására.

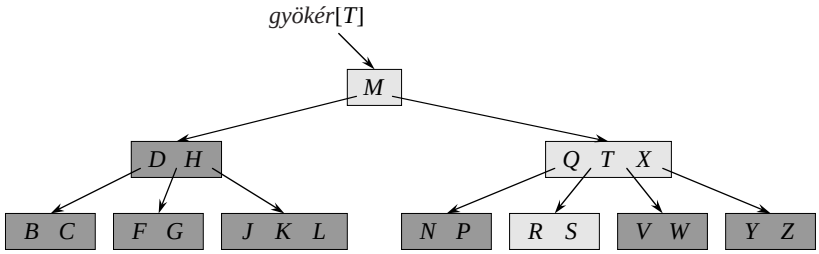
A B-fák elsősorban abban különböznek a piros-fekete fáktól, hogy a B-fában a csúcsoknak sok gyerekük lehet, a gyerekek száma néhánytól több ezerig terjedhet. Azaz, egy B-fában az „elágazási tényező” igen nagy lehet, bár erre a felhasznált mágneslemez jellemzői egy felső korlátot adnak. A B-fák hasonlók a piros-fekete fákhoz abban, hogy minden n-csúcsú B-fának a magassága $O(\lg n)$, bár egy B-fa magassága lényegesen kisebb, mint egy piros-fekete fa magassága, mivel a B-fa elágazási tényezője sokkal nagyobb lehet. Így a B-fákkal is sok $O(\lg n)$ idejű dinamikushalmaz-műveletet tudunk megvalósítani.

A B-fák a bináris keresőfákból általánosítással származtathatók. A 18.1. ábrán egy egyszerű B-fa látható. Ha a B-fa egy x csúcsa $n[x]$ kulcsot tartalmaz, akkor az x-nek $n[x] + 1$ gyereke van. Az x csúcsban a kulcsokat arra használjuk, hogy az x által kezelt kulcstartományt $n[x] + 1$ résztartományra osztjuk úgy, hogy mindegyik résztartományhoz az x-nek egy gyereke tartozzon. Ha a B-fákban egy kulcsot keresünk, akkor $(n[x] + 1)$-féle választásunk lehet, azért, mert a keresett kulcsot az x csúcsban $n[x]$ kulccsal hasonlítjuk össze. A levélcsúcsok szerkezete különbözik a belső csúcsok szerkezetétől; ezzel a különbséggel majd a 18.1. alfejezetben foglalkozunk.

A 18.1. alfejezetben pontosan definiáljuk a B-fákat, és bebizonyítjuk, hogy a B-fa magassága a benne levő csúcsok számának növelésekor legfeljebb a csúcsszámok logaritmusával nő. A 18.2. alfejezetben leírjuk, hogy B-fában hogyan kell kulcsot keresni, és hogyan kell egy B-fába kulcsot beszúrni. A 18.3. alfejezetben pedig egy kulcs törlésével foglalkozunk. Azonban mielőtt ezekkel foglalkoznánk, fel kell tennünk azt a kérdést, hogy vajon miért kell egy mágneslemezen működő adatszerkezetet másképpen értékelni, mint egy, a központi memóriában használható adatszerkezetet.

Adatszerkezetek a másodlagos tárolókon

Sok különböző olyan technológia ismert, amellyel egy számítógéprendszer memóriakapacitását növelni lehet. Egy számítógéprendszer ***elsődleges memóriája*** (vagy ***főmemóriája***) rendszerint szilikonlapkákból áll. Az egy bitre eső előállítási költség ennél a technológiá-



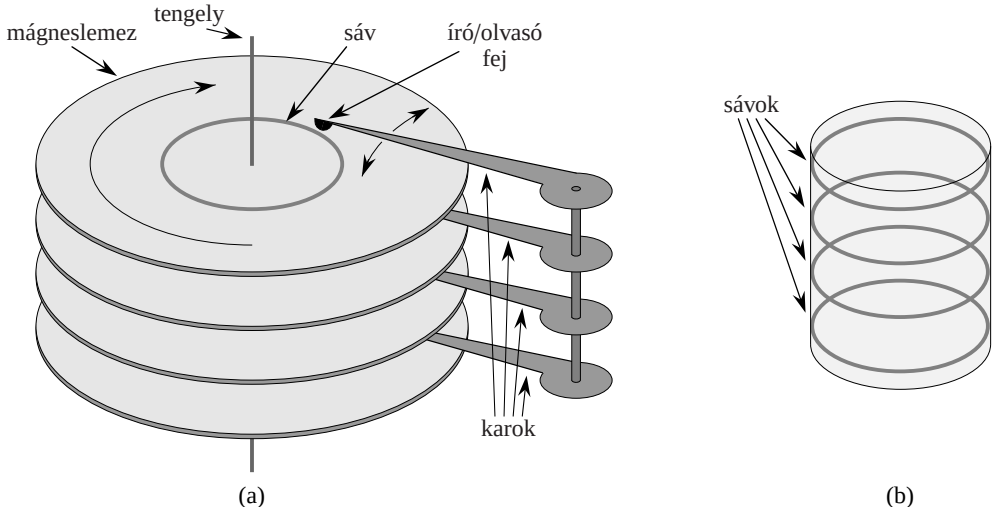
18.1. ábra. Egy B-fa, amelyben a kulcsok az angol ábécé mássalhangzói. Egy belső x csúcsnak $n[x]$ kulcsa és $n[x] + 1$ gyereke van. A fában mindegyik levélnek ugyanakkora a mélysége. Ha az R betűt keressük, akkor a világosszürke színű csúcsokat fogjuk megvizsgálni.

nál két nagyságrenddel nagyobb, mint a mágnesszalagok vagy a mágneslemezek mágneses tárolási módszere esetén. A legtöbb számítógéprendszer mágneslemezes ***másodlagos tárolókat*** is tartalmaz; az ilyen másodlagos tárolók kapacitása gyakran legalább két nagyságrenddel nagyobb, mint az elsődleges memóriáé.

A 18.2(a) ábrán egy tipikus mágneslemezegység látható. Az egység több ***lemezből*** áll, ezek egy közös ***tengelyen*** állandó sebességgel forognak. Mindegyik lemez felülete mágnesezhető anyaggal van bevonva. Mindegyik lemezre egy ***kar*** végén levő ***fejjel*** lehet írni vagy olvasni. A karok egymáshoz vannak „fixen” rögzítve, a karok a rajtuk levő fejeket a lemezek tengelye felé vagy a lemezek széle irányába tudják mozgatni. Ha egy fej mozdulatlan, akkor az általa a mágneslemez felületén bejárt utat ***sávnak*** nevezzük. Az író/olvasó fejek függőlegesen mindig ugyanabban a pozícióban vannak, ezért az alattuk levő sávok mindegyikéhez egyidejűleg férhetnek hozzá. A 18.2(b) ábra mutatja ezeknek a sávoknak a halmazát, amit ***cilindernek*** nevezünk.

Bár a mágneslemezek olcsóbbak és nagyobb kapacitásúak, mint a főmemória, a lemezek sokkal lassabbak, mivel mozgó alkatrészeket tartalmaznak. Kétféle mechanikus mozgás is van: a mágneslemezek forgása és a karok elmozdulása. E könyv írásának idején a kereskedelemben kapható lemezek forgássebessége 5400–15 000 RPM között van (RPM = rotation per minute, percenkénti forgásszám), és 7200 RPM az átlagos érték. Bár a 7200 RPM nagynak tűnik, egy fordulás 8,33 ms ideig tart, ami 5 nagyságrenddel nagyobb, mint a szilikonmemória átlagosnak tekinthető 100 ns-os hozzáférési ideje. Más megvilágításban, mialatt arra várunk, hogy egy adott adat egy teljes fordulat megtétele után ismét az író/olvasó fej alá érkezzen, 100 000-szer tudunk hozzáférni a főmemóriához. Átlagban csak egy fél fordulatra kell várnunk, de a szilikonmemóriához és a mágneslemezekhez való hozzáférések számának különbsége így is óriási. A karok mozgatása is eltart bizonyos ideig. E könyv írásának idején a kereskedelemben kapható lemezek átlagos hozzáférési ideje 3 és 9 ms között van.

Azért, hogy a mechanikus mozgásokra való várakozás idejét csökkentsük, egy lemezhozzáférés nem egy adatot, hanem egyidejűleg sok adatot mozgat. Az információ egyenlő nagyságú ***lapokra*** van felosztva, amelyek egymás után, egy cilinderen belül helyezkednek el, és mindegyik lemezírás vagy -olvasás egy vagy több egész lapra vonatkozik. Egy tipikus mágneslemezen egy lap hossza 2^{11}–2^{14} bájt. Ha az író/olvasó fej pozicionálása pontosan megtörtént, és a mágneslemez a kívánt lap elejéhez ért, akkor a mágneslemez írása és olvasása (a mágneslemez forgásától eltekintve) teljesen elektronikus, és így gyorsan lehet írni és olvasni nagy tömegű adatot.



18.2. ábra. (a) Egy tipikus mágneslemez egység. Több mágneslemezből áll, amelyek egy tengely körül forognak. Mindegyik lemez egy kar végén levő fejjel írható és olvasható. A karok egymáshoz vannak kapcsolva, így a fejeket egyszerre mozgatják. A karok egy közös forgástengely körül mozognak. A sáv az a felület, amit az író/olvasó fej bejár akkor, amikor a kar mozdulatlan. **(b)** A cilindert az egymás alatti sávok halmaza alkotja.

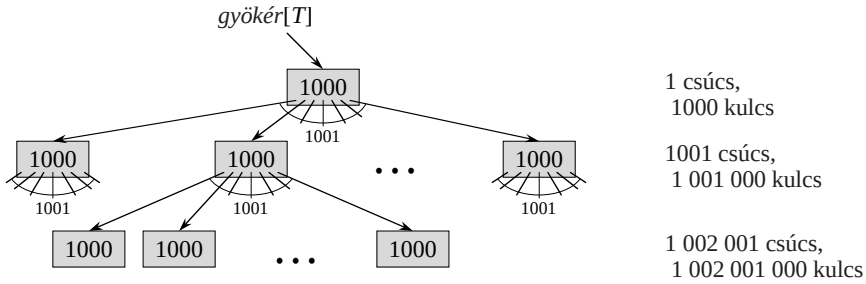
Gyakran egy lap elérése és beolvasása tovább tart, mint a lapon levő összes adatnak a számítógéppel történő teljes feldolgozása. Ezért ebben a fejezetben a futási időt két fő összetevőre bontjuk fel:

- a lemezelérések számára és
- a központi egység (számolási) idejére.

A lemezelérések számán a szükséges lapírások vagy lapolvasások számát értjük. Megjegyezzük azonban, hogy a mágneslemez hozzáférési ideje nem konstans – hiszen a pillanatnyi sáv és az elérendő sáv távolságától és a mágneslemez pillanatnyi forgási állapotától függ. A lapírások és lapolvasások darabszámát ezért csak a mágneslemez elérésére fordított idő közelítő becslésére használhatjuk.

A B-fák tipikus alkalmazásaiban a kezelt adatmennyiség akkora, hogy rendszerint nem fér el egyszerre a főmemóriában. A B-fa algoritmusok csak azokat a kiválasztott lapokat olvassák be a mágneslemezről a memóriába, amelyekre szükség van, és csak a megváltoztatott tartalmú lapokat írják vissza a memóriából a mágneslemezre. A B-fa algoritmusokat úgy tervezték, hogy mindig csak egy állandó darabszámú lap van a főmemóriában, így a főmemória mérete nem korlátozza a kezelhető B-fák méretét.

A mágneslemez-műveleteket a következő algoritmussal modellezzük. Legyen x egy objektumra mutató pointer. Ha az objektum pillanatnyilag a számítógép főmemóriájában van, akkor ennek az objektumnak a mezőire a szokásos módon hivatkozhatunk: például *kulcs*[x]. Ha az x-szel jelölt objektum a mágneslemezen van, akkor először a LEMEZRŐL-OLVAS(x) műveletet kell végrehajtani, ez beolvassa az x objektumot a főmemóriába, és az x mezőire csak ezután lehet hivatkozni. (Feltesszük, hogy ha x már a főmemóriában van, akkor a LEMEZRŐL-OLVAS(x) művelet nem végez lemezolvasást, azaz egy NOP, „no-operation” műveletnek felel meg.) Ehhez hasonlóan, a LEMEZRE-ÍR(x) mű-



18.3. ábra. Egy 2 magasságú B-fa, amelynek több mint egymilliárd kulcsa van. Mindegyik belső csúcsnak és levélnek 1000 kulcsa van. Az 1 mélységben 1001 csúcs van, a 2 mélységben több mint egymillió levél található. Minden x csúcsban $n[x]$, azaz az x-ben levő kulcsok száma is látható.

veletet használjuk arra, hogy egy megváltozott mezőjű x objektumot a mágneslemezre mentsünk el. Egy objektummal kapcsolatos művelet tipikus mintája tehát a következő:

```
1  x ← az objektum mutatója
2  LEMEZRŐL-OLVAS(x)
3  azok a műveletek, amelyek az x mezőit olvassák vagy módosítják
4  LEMEZRE-ÍR(x)                ▷ Kimarad, ha az x egyik mezője sem változott meg.
5  további műveletek, amelyek az x mezőit olvassák, de nem módosítják
```

A rendszer egyidejűleg csak korlátozott darabszámú lapot tud tárolni a főmemóriában. Feltételezzük, hogy azokat a lapokat, amelyekre a továbbiakban nincs szükség, a főmemóriából a rendszer távolítja el; a B-fa algoritmusok ezzel a feladattal nem foglalkoznak.

Mivel a legtöbb rendszerben egy B-fa algoritmus futási idejét elsősorban a végrehajtott LEMEZRŐL-OLVAS és LEMEZRE-ÍR műveletek száma határozza meg, fontos, hogy ezeket a műveleteket hatékonyan használjuk, azaz egy művelet olyan sok információt olvasson vagy írjon, amennyi csak lehetséges. Ezért a B-fa egy csúcsának a nagysága rendszerint a mágneslemez egy lapjának a nagyságával egyezik meg, és így a B-fa egy csúcsában a gyerekek számát a mágneslemez lapjának mérete korlátozhatja.

Egy mágneslemezen tárolt B-fára az elágazási tényező gyakran 50 és 2000 között van, attól függően, hogy mi az arány egy kulcs mérete és egy lap mérete között. A nagy elágazási tényező miatt mind a fa magassága, mind egy adott kulcs megkereséséhez szükséges lemezelérések száma jelentősen csökken. A 18.3. ábrán egy olyan 2 magasságú B-fa látható, amelynek az elágazási tényezője 1001, és a B-fa több mint egymilliárd kulcsot tartalmaz; és, mivel a fa gyökércsúcsát állandóan a főmemóriában tárolhatjuk, a fa bármelyik kulcsának eléréséhez legfeljebb csak *két* lemezművelet szükséges.

18.1. A B-fa definíciója

Az egyszerűség kedvéért feltesszük, ahogyan a bináris keresőfáknál és a piros-fekete fáknál is feltettük, hogy a kulcshoz tartozó minden „kísérő információt” ugyanabban a csúcsban tárolunk, ahol a kulcsot. Ez a gyakorlatban azt jelenti, hogy mindegyik kulcshoz elegendő egy másik lapra mutató pointert tárolni, amelyen a kulcshoz tartozó kísérő információt he-

lyezhetjük el. Az ebben a fejezetben levő algoritmusokban feltesszük, hogy ha a kulcsot egyik csúcsból egy másikba visszük, a kulcshoz tartozó kísérő információ, vagy az ilyen információra mutató pointer a kulccsal együtt mozog. Egy másik általánosan használt B-fa szerkezet, amit ***B*$^+$*-fának*** nevezünk, olyan, hogy minden kísérő információt a levelekben tárolunk, és csak a kulcsokat, valamint a gyerekekre mutató pointereket tartjuk a közbülső csúcsokban, ezzel maximalizálva a belső csúcsok elágazási tényezőjét.

A T ***B-fa*** egy olyan gyökeres fa (a gyökércsúcsa *gyökér*$[T]$), amelyre a következő tulajdonságok teljesülnek.

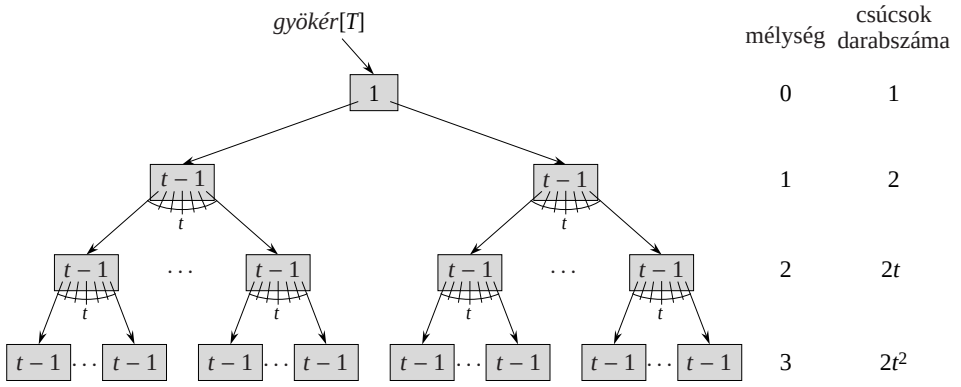
1. Minden x csúcsnak a következő mezői vannak:
   a. $n[x]$, az x csúcsban tárolt kulcsok darabszáma,
   b. az $n[x]$ darab kulcs, a kulcsokat nemcsökkenő sorrendben tároljuk, úgy, hogy $kulcs_1[x] \leq kulcs_2[x] \leq \cdots \leq kulcs_{n[x]}[x]$,
   c. *levél*$[x]$, egy logikai változó, amelynek az értéke IGAZ, ha x levél, és HAMIS, ha x egy belső csúcs.
2. Minden belső x csúcs ezenkívül tartalmaz $n[x]+1$ mutatót, a $c_1[x]$, $c_2[x]$, $\ldots$, $c_{n[x]+1}[x]$ mutatókat, amelyek az x gyerekeire mutatnak. A levélcsúcsoknak nincsenek gyerekeik, ezért a levelek c_i mezői definiálatlanok.
3. A $kulcs_i[x]$ értékek meghatározzák a kulcsértékeknek azokat a tartományait, amelyekbe a részfák kulcsai esnek. Ha k_i egy olyan kulcs, amelyik a $c_i[x]$ gyökerű részfában van, akkor
$$k_1 \leq kulcs_1[x] \leq k_2 \leq kulcs_2[x] \leq \cdots \leq kulcs_{n[x]}[x] \leq k_{n[x]+1}.$$
4. Minden levélnek azonos a mélysége, ez az érték a fa h magassága.
5. A csúcsokban tárolható kulcsok darabszámára adott egy alsó és egy felső korlát. Ezeket a korlátokat egy rögzített t egész számmal ($t \geq 2$) lehet kifejezni, és ezt a számot a B-fa ***minimális fokszámának*** nevezzük:
   a. Minden nemgyökér csúcsnak legalább $t-1$ kulcsa van. Minden belső csúcsnak így legalább t gyereke van. Ha a fa nem üres, akkor a gyökércsúcsnak legalább egy kulcsának kell lennie.
   b. Minden csúcsnak legfeljebb $2t-1$ kulcsa lehet. Tehát egy belső csúcsnak legfeljebb $2t$ gyereke lehet. Azt mondjuk, hogy egy csúcs ***telített,*** ha pontosan $2t-1$ kulcsa van.[1]

A legegyszerűbb B-fára $t = 2$. Ekkor minden belső csúcsnak 2, 3 vagy 4 gyereke van, ezt a fát ***2-3-4 fának*** nevezzük. A gyakorlatban azonban ennél sokkal nagyobb t értékű fákat használnak.

A B-fa magassága

A B-fákra vonatkozó műveletek lemezhozzáféréseinek száma arányos a B-fa magasságával. Most a B-fák magasságainak legrosszabb eseteit vizsgáljuk.

[1] A B-fák egy másik, ***B***-fának* nevezett változatában minden belső csúcs legalább 2/3 részben telített, szemben a B-fák legalább 1/2-es telítettségével.



18.4. ábra. Egy 3 magasságú B-fa, amelyik a lehető legkevesebb kulcsot tartalmazza. Minden x csúcsban megadjuk az x-ben tárolt kulcsok $n[x]$ számát.

18.1. tétel. *Ha $n \geq 1$, akkor minden olyan T n-kulcsos B-fára, amelynek magassága h és minimális fokszáma $t \geq 2$, teljesül, hogy*

$$h \leq \log_t \frac{n+1}{2}.$$

Bizonyítás. Ha egy B-fa magassága h, a gyökércsúcsnak egy kulcsa, és minden más csúcsnak legalább $t-1$ kulcsa van. Így van legalább 2 darab 1 mélységű, legalább $2t$ darab 2 mélységű, legalább $2t^2$ darab 3 mélységű, ..., legalább $2t^{h-1}$ darab h mélységű csúcs. A 18.4. ábrán egy ilyen fa látható $h = 3$-ra. Így a kulcsok n darabszámára teljesül a következő egyenlőtlenség:

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1. \end{aligned}$$

Ebből nyilvánvalóan $t^h \leq (n+1)/2$. Mindkét oldal t-alapú logaritmusát véve a tétel állítását kapjuk meg. ■

Összehasonlítva a piros-fekete fákkal, most már látszik a B-fák hatékonysága. Bár mindkét fára a fa magassága $O(\lg n)$ (hangsúlyozzuk, hogy t állandó), a B-fára a logaritmus alapja sokszorosan nagyobb lehet. Így a legtöbb műveletre a B-fák legalább $(\lg t)$-szer kevesebb csúcsot vizsgálnak meg, mint a piros-fekete fák. Mivel egy csúcs vizsgálata rendszerint egy lemezhozzáférést igényel, a lemezhozzáférések száma jelentősen csökken.

Gyakorlatok

18.1-1. Miért nem engedtük meg azt, hogy a minimális fokszám $t = 1$ legyen?

18.1-2. A 18.1. ábrán látható fa milyen t értékre lesz valóban B-fa?

18.1-3. Adjuk meg az összes olyan helyes B-fát, amelynek a minimális fokszáma 2, és amelyek az {1,2,3,4,5}-t ábrázolják.
18.1-4. Fejezzük ki a h magasságú B-fában tárolható kulcsok maximális számát a t minimális fokszám függvényében.
18.1-5. Írjuk le azt az adatszerkezetet, amelyet akkor kapnánk, ha egy piros-fekete fában mindegyik fekete csúcs lekötné a fa összes piros gyerekét, beleértve a sajátjait is.

18.2. A B-fák alapműveletei

Ebben az alfejezetben részletesen foglalkozunk a B-FÁBAN-KERES, a B-FÁT-LÉTREHOZ és a B-FÁBA-BESZÚR műveletekkel. Ezekben az eljárásokban mindig feltesszük, hogy teljesül a következő két feltétel:

- A B-fa gyökere mindig a főmemóriában van, ezért a gyökércsúcsra a LEMEZRŐL-OLVAS műveletet sohasem kell végrehajtani; a LEMEZRE-ÍR műveletet azonban végre kell hajtani, ha a gyökércsúcs megváltozott.

- Minden olyan csúcsra, amelyet paraméterként adunk át, már végrehajtottunk egy LEMEZRŐL-OLVAS műveletet.

A következő algoritmusok mindegyike „egymenetes”, a fában a gyökércsúcsból kiindulva felülről lefelé, visszalépés nélkül halad.

Keresés a B-fában

A B-fában történő keresés nagyon hasonló a bináris keresőfában történő kereséshez, abban különbözik tőle, hogy a fa minden csúcsában a bináris vagy „kétutas” elágaztatások helyett itt a csúcs gyerekeinek számától függő többutas elágaztató döntéseket kell hoznunk. Pontosabban, minden x belső csúcsban egy $(n[x] + 1)$-utas elágazásban kell döntést hoznunk.

A B-FÁBAN-KERES eljárás a bináris keresőfákra definiált FÁBAN-KERES eljárás általánosítása. A B-FÁBAN-KERES eljárás bemenő adata egy részfa x gyökércsúcsára mutató pointer és egy k kulcs, amelyet ebben a részfában kell megkeresni. A legfelső szintre történő hívás tehát B-FÁBAN-KERES($gyökér[T], k$) alakú. Ha a k kulcs a B-fában benne van, akkor a B-FÁBAN-KERES eredményül az (y, i) rendezett párt adja, ahol az y csúcsra és az i indexre $kulcs_i[y] = k$. Egyébként az eredmény a NIL lesz.

B-FÁBAN-KERES(x, k)

```
1  i ← 1
2  while i ≤ n[x] és k > kulcs_i[x]
3        do i ← i + 1
4  if i ≤ n[x] és k = kulcs_i[x]
5    then return (x, i)
6  if levél[x]
7    then return NIL
8    else LEMEZRŐL-OLVAS(c_i[x])
9         return B-FÁBAN-KERES(c_i[x], k)
```

A lineáris keresést használva, az 1–3. sorokban megkeressük a legkisebb olyan i indexet, amelyre $k \leq kulcs_i[x]$, vagy ha ilyen i nincs, akkor az i-be az $n[x] + 1$ kerül. A 4–5. sorokban azt ellenőrizzük, hogy vajon a kulcsot találtuk-e meg, és ha igen, akkor ezt adjuk vissza eredményül. A 6–9. sorokban vagy sikertelen lesz a keresés (ha x egy levél), vagy rekurzívan tovább folytatjuk a keresést x megfelelő részfájában, miután erre a gyerekre végrehajtjuk a szükséges LEMEZRŐL-OLVAS műveletet.

A 18.1. ábrán a B-FÁBAN-KERES eljárás látható; a világosszürke csúcsokat vizsgáljuk meg az R kulcs keresése során.

Ahogyan az a bináris keresőfákra alkalmazott FÁBAN-KERES eljárásban is volt, a rekurzióban szereplő csúcsok a fa gyökércsúcsából kiinduló, lefelé haladó útnak a csúcsai. A B-FÁBAN-KERES által végrehajtott lemezműveletek száma ezért $O(h) = O(\log_t n)$, ahol h a B-fa magassága, és n a B-fában lévő kulcsok száma. Mivel $n[x] < 2t$, a 2–3. sorokban lévő **while** ciklus ideje minden csúcsra $O(t)$, és a központi egység összes műveleti ideje $O(th) = O(t \log_t n)$.

Egy üres B-fa készítése

Egy T B-fa felépítéséhez először a B-FÁT-LÉTREHOZ műveletet használjuk, ez egy üres gyökércsúcsot fog adni, ezután az új kulcsok beírására a B-FÁBA-BESZÚR eljárást hívjuk meg. Mindkét eljárás egy CSÚCSOT-ELHELYEZ segédeljárást használ, amelyik $O(1)$ idő alatt lefoglalja az új csúcsnak a lemez egy lapját. Feltételezhetjük, hogy a CSÚCSOT-ELHELYEZ eljárásnak nincs szüksége a LEMEZRŐL-OLVAS meghívására, mivel eddig ehhez a csúcshoz semmilyen információt nem tároltunk még a lemezen.

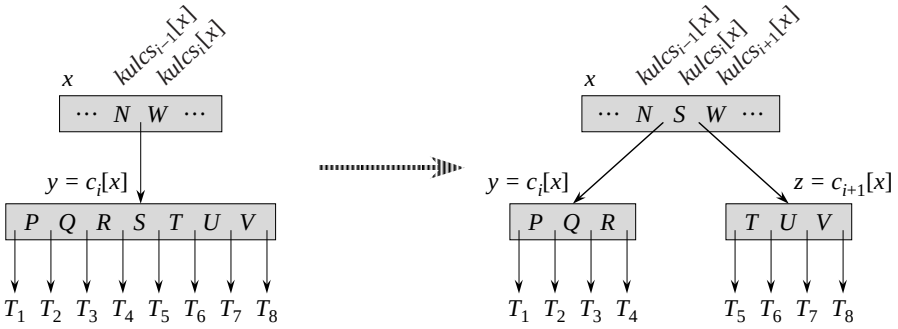
B-FÁT-LÉTREHOZ(T)

1 $x \leftarrow$ CSÚCSOT-ELHELYEZ()
2 *levél*$[x] \leftarrow$ IGAZ
3 $n[x] \leftarrow 0$
4 LEMEZRE-ÍR(x)
5 *gyökér*$[T] \leftarrow x$

A B-FÁT-LÉTREHOZ eljáráshoz $O(1)$ lemezművelet és $O(1)$ központi egység idő kell.

Egy kulcs beszúrása a B-fába

Egy B-fába kulcsot beszúrni sokkal bonyolultabb, mint egy bináris keresőfába. Mint a bináris keresőfáknál, megkeressük annak a levélnek a helyét, ahová az új kulcsot beszúrjuk. B-fáknál azonban nem tudunk egyszerűen csak létrehozni és beszúrni egy levélcsúcsot, mivel előfordulhat, hogy az így kapott fa nem lesz érvényes B-fa. Ehelyett az új kulcsot egy már létező levélcsúcsba szúrjuk be. Mivel nem tudunk beszúrni egy kulcsot egy olyan levélcsúcsba, ami már tele van, bevezetünk egy műveletet, amely ***szétvágja*** a $2t - 1$ darab kulcsot tartalmazó telített y csúcsot a $kulcs_t[y]$ ***középső kulcsa*** körül két $t - 1$ kulcsú csúcsra. A középső kulcs az y szülőjébe megy át, és ez a középső csúcs adja a vágási pontot a két új fa között. Ha az y szülője telített, szét kell vágni, még mielőtt az új kulcsot ide behelyeznénk, és így lehet, hogy a telített kulcsok szétvágását meg kell ismételnünk a fában felfelé haladva több csúcson keresztül.



18.5. ábra. Egy olyan csúcs szétvágása, amelyre $t = 4$. Az y csúcsot az y és a z csúcsokra vágtuk szét, és az y csúcs középső S kulcsát az y szülőjébe vittük át.

Ahogyan a bináris kereső fáknál is tettük, egy menetben, a gyökértől egy levél felé lefelé haladva, be tudunk szúrni egy kulcsot a B-fába. Azért, hogy ezt megtehessük, nem várunk arra, hogy vajon szét kell-e vágnunk a telített csúcsokat azért, hogy a beszúrás végrehajtható legyen. Ahogyan haladunk lefelé a fában, keresve azt a pozíciót, ahová az új kulcs tartozik, minden olyan telített csúcsot szétvágunk, ami az úton van (még a levélcsúcsot is). Így bármikor, amikor egy telített y csúcsot szét akarunk vágni, feltehetjük, hogy a szülője nem telített.

Egy csúcs szétvágása a B-fában

A B-FA-VÁGÁS-GYEREK eljárás bemenete egy belső x csúcs, amelyik *nem telített* (feltesszük, hogy ez a csúcs a központi memóriában van), egy i index és egy olyan y csúcs (feltesszük, hogy ez a csúcs is a központi memóriában van), melyre $y = c_i[x]$, és y az x-nek egy olyan gyereke, amelyik *telített*. Az eljárás ekkor az y-t két részre vágja, és az x-et úgy alakítja át, hogy eggyel több gyereke lesz. (Egy telített gyökércsúcs szétvágásához először a gyökércsúcsot egy új üres gyökércsúcs gyerekévé tesszük, erre a B-FA-VÁGÁS-GYEREK algoritmust használhatjuk. Ekkor a fa magassága eggyel nő; a vágás az egyetlen olyan művelet, amely a magasságát növeli.)

Ez a folyamat a 18.5. ábrán látható. Az y csúcsot, amelyik telített, az S középső kulcsnál vágjuk el, az S az y szülőjéhez, az x-hez megy át. Az y-nak azok a kulcsai, amelyek nagyobbak a középső kulcsnál, a z csúcsot alkotják, és ez a csúcs az x egy új gyereke lesz.

B-FA-VÁGÁS-GYEREK(x, i, y)

```
1  z ← CSÚCSOT-ELHELYEZ()
2  levél[z] ← levél[y]
3  n[z] ← t – 1
4  for j ← 1 to t – 1
5       do kulcs_j[z] ← kulcs_{j+t}[y]
6  if nem levél[y]
7     then for j ← 1 to t
8               do c_j[z] ← c_{j+t}[y]
9  n[y] ← t – 1
```

```
10  for j ← n[x] + 1 downto i + 1
11      do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14      do kulcs_{j+1}[x] ← kulcs_j[x]
15  kulcs_i[x] ← kulcs_t[y]
16  n[x] ← n[x] + 1
17  LEMEZRE-ÍR(y)
18  LEMEZRE-ÍR(z)
19  LEMEZRE-ÍR(x)
```

A B-FA-VÁGÁS-GYEREK eljárás a „szétvágás és összeillesztés” módszert használja. Az eljárásban y az x-nek az i-edik gyereke, és az y csúcsot vágjuk szét. Az y csúcsnak eredetileg $2t$ gyereke ($2t - 1$ kulcsa) volt, de a szétvágás után már csak t gyereke ($t - 1$ kulcsa) lesz. A z csúcs az y csúcs t legnagyobb gyerekeit ($t - 1$ kulcsát) „adoptálja”, és z az x új gyereke lesz, az x gyerekeinek táblázatában a z közvetlenül az y után áll. Az y csúcs középső kulcsa az x-nek egy olyan új kulcsa lesz, amelyik az y és a z között áll.

Az 1–8. sorokban a z csúcsot építjük fel, megadjuk hozzá a „nagyobb” $t - 1$ darab kulcsot és az y-nak a kulcsokhoz tartozó t gyerekét. A 9. sorban módosítjuk az y kulcsainak darabszámát. Végül, a 10–16. sorokban, behelyezzük z-t az x gyerekei közé, átvisszük az y középső csúcsát az x-be az y és a z közé, majd beállítjuk az x kulcsainak számát. A 17–19. sorokban a módosított lemezlapokat felírjuk a lemezre. A B-FA-VÁGÁS-GYEREK eljárás központi egység ideje $\Theta(t)$, a 4–5. és a 7–8. sorokban levő ciklusok miatt. (A többi ciklus ismétléseinek száma legfeljebb $O(t)$.) Az eljárás $O(1)$ lemezműveletet hajt végre.

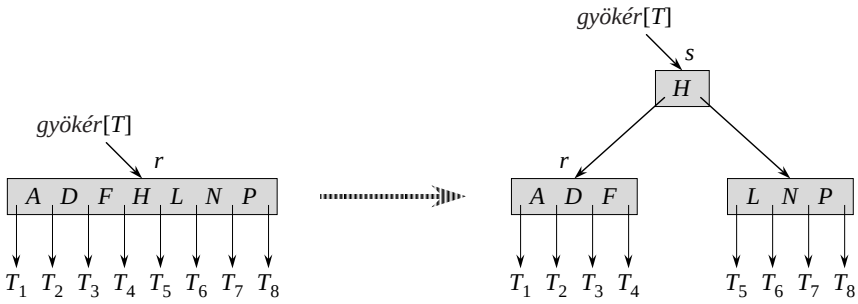
Kulcs beszúrása a B-fába, egy menetben, felülről lefelé haladva

Egy menetben beszúrunk egy k kulcsot egy h magasságú T B-fába, a fában lefelé haladva úgy, hogy az algoritmus végrehajtásához $O(h)$ lemezhozzáférés kell. A szükséges központi egység időigény $O(th) = O(t \log_t n)$. A B-FÁBA-BESZÚR eljárás a B-FA-VÁGÁS-GYEREK eljárást használja, ez biztosítja, hogy a rekurzió sohasem száll le olyan csúcsra, amelyik telített.

B-FÁBA-BESZÚR(T, k)

```
 1  r ← gyökér[T]
 2  if n[r] = 2t − 1
 3    then s ← CSÚCSOT-ELHELYEZ()
 4         gyökér[T] ← s
 5         levél[s] ← HAMIS
 6         n[s] ← 0
 7         c_1[s] ← r
 8         B-FA-VÁGÁS-GYEREK(s, 1, r)
 9         NEM-TELÍTETT-B-FÁBA-BESZÚR(s, k)
10    else NEM-TELÍTETT-B-FÁBA-BESZÚR(r, k)
```

A 3–9. sorokban az az eset áll fenn, hogy az r gyökércsúcs telített: a gyökércsúcs két részre válik, és egy új s csúcs (amelynek két gyereke van) lesz a gyökércsúcs. A gyökércsúcs kettévágása az egyetlen olyan művelet, amely egy B-fa magasságát növeli. A 18.6. ábrán ez az eset látható. A bináris keresőfával ellentétben, a B-fa magassága a gyökérnél nő, és nem



18.6. ábra. Egy olyan gyökércsúcs szétvágása, amelyre $t = 4$. Az r gyökércsúcs két részre válik, és egy új s gyökércsúcs keletkezik. Az új gyökércsúcs r középső kulcsát tartalmazza, és két gyereke van, ezek r félbevágásából származnak. A gyökércsúcs szétvágásával a B-fa magassága eggyel nő.

a fa alján. Az eljárás a NEM-TELÍTETT-B-FÁBA-BESZÚR meghívásával fejeződik be, ez az eljárás beszúrja a k kulcsot abba a fába, amelynek a gyökércsúcsa nem telített. A NEM-TELÍTETT-B-FÁBA-BESZÚR eljárás, ha szükséges, rekurzívan lefelé halad a fában, a B-FA-VÁGÁS-GYEREK meghívásával mindig biztosítva azt, hogy az a csúcs, amivel foglalkozik, nem telített.

A NEM-TELÍTETT-B-FÁBA-BESZÚR rekurzív segédeljárás beszúrja a k kulcsot az x csúcsba, amelyről feltesszük, hogy az eljárás hívásakor nem telített. A B-FÁBA-BESZÚR és a NEM-TELÍTETT-B-FÁBA-BESZÚR eljárás rekurzív működése ezt a feltételt biztosítja.

```
NEM-TELÍTETT-B-FÁBA-BESZÚR(x, k)
 1  i ← n[x]
 2  if levél[x]
 3    then while i ≥ 1 és k < kulcs_i[x]
 4                 do kulcs_{i+1}[x] ← kulcs_i[x]
 5                    i ← i − 1
 6         kulcs_{i+1}[x] ← k
 7         n[x] ← n[x] + 1
 8         LEMEZRE-ÍR(x)
 9    else while i ≥ 1 és k < kulcs_i[x]
10                 do i ← i − 1
11         i ← i + 1
12         LEMEZRŐL-OLVAS(c_i[x])
13         if n[c_i[x]] = 2t − 1
14           then B-FA-VÁGÁS-GYEREK(x, i, c_i[x])
15                if k > kulcs_i[x]
16                  then i ← i + 1
17         NEM-TELÍTETT-B-FÁBA-BESZÚR(c_i[x], k)
```

A NEM-TELÍTETT-B-FÁBA-BESZÚR eljárás a következőképpen működik. A 3–8. sorokban az az eset van, amikor x egy levél, és a k kulcsot az x-be kell beszúrni. Ha x nem levél, akkor a k kulcsot egy olyan részfának egy megfelelő levelébe kell beszúrni, amelynek az x belső csúcs a gyökércsúcsa. Ebben az esetben, a 9–11. sorokban meghatározzuk az x-nek azt a gyerekét, amelyik a rekurzív hívásnak majd paramétere lesz. A 13. sorban azt vizsgáljuk,

hogy ez a gyerek telített-e; ha igen, akkor a 14. sorban a B-FA-VÁGÁS-GYEREK eljárással ezt a csúcsot két olyan gyerekre bontjuk, amelyek egyike sem telített. A 15–16. sorokban azt határozzuk meg, hogy a két gyerek közül melyik az, amelyikre a rekurziót alkalmaznunk kell. (Megjegyezzük, hogy a 16. sorban, az i növelése után nincs szükség a LEMEZRŐL-OLVAS($c_i[x]$) műveletre, mivel a rekurzív hívás ebben az esetben arra a gyerekre vonatkozik, amelyet a B-FA-VÁGÁS-GYEREK eljárás éppen most készített.) A 13–16. sorok programja tehát azt biztosítja, hogy a rekurzív hívást sohasem adjuk ki egy telített csúcsra. A 17. sorban rekurzívan meghívjuk a k beszúrását végző eljárást az így meghatározott részfára. A 18.7. ábrán egy B-fába történő beszúrás különböző eseteit láthatjuk.

Egy h magasságú B-fára a B-FÁBA-BESZÚR eljárás $O(h)$ lemezhozzáférést hajt végre, mivel a NEM-TELÍTETT-B-FÁBA-BESZÚR eljáráshívások között csak $O(1)$ LEMEZRŐL-OLVAS és LEMEZRE-ÍR műveletre van szükség. A központi egység teljes időigénye $O(th) = O(t \log_t n)$. Mivel a NEM-TELÍTETT-B-FÁBA-BESZÚR eljárás vég-rekurzív, az eljárás egy **while** ciklussal is megvalósítható, ami azt mutatja, hogy a központi memóriában egyidejűleg csak $O(1)$ lapnak kell lennie.

Gyakorlatok

18.2-1. Mutassuk meg, mi az eredménye annak, ha egy 2 minimális fokszámú, üres B-fába rendre beszúrjuk az

$$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$$

kulcsokat. Rajzoljuk fel azokat a B-fákat, amelyeknek egy csúcsát a soron következő beszúrás felvágja. Adjuk meg az utolsó beszúrás utánit is.

18.2-2. Vizsgáljuk meg, hogy a B-FÁBA-BESZÚR eljáráshívás végrehajtása milyen körülmények között hajt végre felesleges LEMEZRŐL-OLVAS vagy LEMEZRE-ÍR műveleteket, ha egyáltalán van ilyen körülmény. (Egy LEMEZRŐL-OLVAS műveletet feleslegesnek nevezünk, ha egy olyan lapra hajtjuk végre, amelyik már a memóriában van. Egy LEMEZRE-ÍR művelet felesleges, ha egy olyan lapot írunk a lemezre, amelynek tartalma azonos azzal, amit már a lemezen tárolunk.)

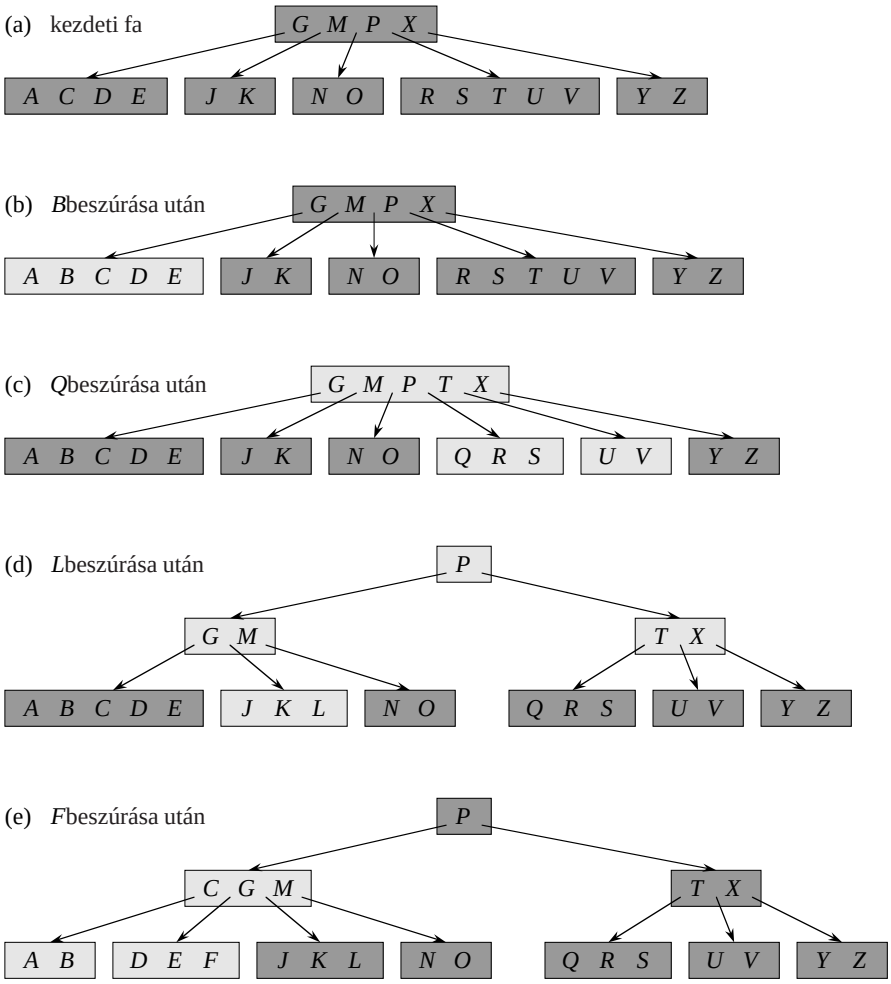
18.2-3. Vizsgáljuk meg, hogyan lehet egy B-fában a minimális kulcsot megkeresni, és hogyan lehet egy adott kulcsot megelőző kulcsot megtalálni.

18.2-4.⋆ Tegyük fel, hogy az $\{1, 2, \ldots, n\}$ kulcsokat egy üres B-fába beszúrtuk, és a minimális fokszám 2. Hány csúcsa van az eredményül kapott B-fának?

18.2-5. Mivel a levelekben nincs gyerekekre mutató pointer, elképzelhető, hogy ugyanarra a mágneslemez lapméretre a leveleknek más (nagyobb) t értékük van, mint a belső csúcsoknak. Mutassuk meg, hogyan kellene módosítani a B-fát létrehozó és a B-fába beszúró eljárásokat úgy, hogy ezt a lehetőséget is kihasználják.

18.2-6. Tegyük fel, hogy a B-FÁBAN-KERES eljárást úgy valósítjuk meg, hogy az eljárás a B-fa mindegyik csúcsában a lineáris keresés helyett a bináris keresést használja. Mutassuk meg, hogy ezzel a központi egység időigénye $O(\lg n)$ lesz, függetlenül attól, hogy a t-t, mint az n függvényét, mekkorának választjuk.

18.2-7. Tegyük fel, hogy a hardver lehetővé teszi azt, hogy a lemez lapméretét tetszőlegesen megválaszthassuk, de egy lap olvasásának ideje mindig $a + bt$, ahol a és b előre megadott állandók, t a B-fa minimális fokszáma, és a B-fa műveletek a megválasztott lapméretet használják. Adjuk meg, hogy mekkorának kell választani a t-t, ha a B-fa keresési idejét (közelítőleg) minimalizálni akarjuk. Javasoljunk egy optimális t értéket arra az esetre, amikor $a = 5$ ms és $b = 10\ \mu$s.



18.7. ábra. Egy B-fába kulcsokat szúrunk be. A B-fa t minimális fokszáma 3, ezért egy csúcsnak legfeljebb 5 kulcsa lehet. Azokat a csúcsokat, amelyeket a beszúrás alatt módosítottuk, világosszürke színnel jelöltük. **(a)** A példában szereplő fa kezdeti állapota. **(b)** A B beszúrásának az eredménye; ez egy egyszerű eset, egy levélbe kellett beszúrni. **(c)** Az előző fába egy Q kulcsot szúrtunk be. Az $RSTUV$ csúcs két részre bomlott, az RS és az UV csúcsokra, a T a gyökércsúcsba ment, és a Q a bal oldali új csúcsba (RS-be) került. **(d)** Az előző fába egy L kulcsot szúrtunk be. A gyökércsúcsot fel kellett bontani, mivel már telített volt, így a B-fa magassága eggyel nőtt. Ezután az L kulcsot a JK-t tartalmazó levélbe szúrtuk be. **(e)** Az előző fába az F kulcsot szúrtuk be. Az $ABCDE$ csúcsot szétvágtuk, és ezután az F kulcsot a jobb oldali új csúcsba (DE-be) szúrtuk be.

18.3. Egy kulcs törlése a B-fából

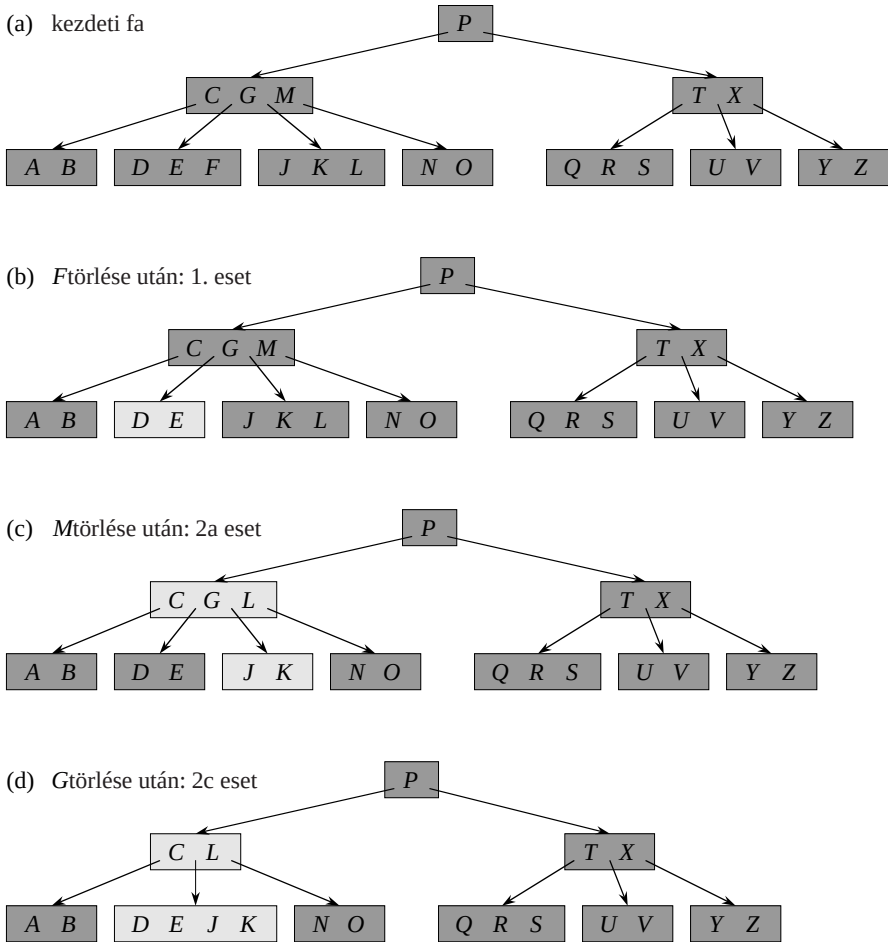
Egy kulcs törlésének algoritmusa hasonló egy kulcs beszúrásának algoritmusához, bár bonyolultabb, mivel egy kulcsot nemcsak egy levélből, hanem tetszőleges csúcsból lehet törölni, és egy belső csúcsból való törlés a csúcs gyerekeinek átrendezésével jár. Mint a beszúrásnál is tettük, nem szabad végrehajtanunk az olyan törlést, amelyik a B-fa tulajdonságoknak ellentmondó fát ad eredményül. A beszúrásnál azt kellett biztosítanunk, hogy a

csúcs ne legyen túlzottan nagy, most a törlésnél arra kell ügyelnünk, hogy a csúcs ne váljon nagyon kicsivé (kivéve azt az esetet, hogy a gyökércsúcsban lehet a minimális $(t - 1)$ kulcsnál kevesebb, de az már nem engedélyezett, hogy a gyökércsúcsnak a maximális $2t - 1$ kulcsnál több kulcsa legyen). Ahogyan a beszúrás algoritmusa visszalépett akkor, amikor azon az úton, ahová egy kulcsot akart beszúrni, egy csúcs telített volt, egy egyszerű törlő algoritmusnak vissza kell lépnie, ha azon az úton, ahonnan egy kulcsot akar törölni, egy (a gyökércsúcstól különböző) olyan csúcs van, aminek minimális számú kulcsa van.

Tegyük fel, hogy a B-FÁBÓL-TÖRÖL eljárás egy B-fa x gyökércsúcsú részfájából törli a k kulcsot. Az eljárást úgy szerkesztettük meg, hogy minden olyan esetben, amikor a B-FÁBÓL-TÖRÖL eljárást rekurzívan meghívjuk egy x csúcsra, az x csúcs kulcsainak darabszáma nem kisebb, mint a t minimális fokszám. Megjegyezzük, hogy ez a feltétel eggyel több kulcsot követel meg, mint amit a B-fa definíciója előír, éppen ezért néha egy kulcsot be kell vinni egy gyerekcsúcsba, mielőtt a rekurzióval erre a gyerekcsúcsra lépnénk. Ez az erős feltétel lehetővé teszi, hogy a fából egy kulcsot töröljünk egy lefelé haladó menetben, visszalépésre nincs szükség (egy kivétel van, erre majd visszatérünk). A B-fából való törlés következő specifikációja úgy értelmezendő, hogy figyelembe vesszük: ha az x gyökércsúcs egy olyan belső csúccsá válik, amelyiknek nincs kulcsa (ez fordul elő az alábbi 2c és 3b esetekben), akkor az x törlődik, és a $c_1[x]$ gyerek lesz a fa új gyökércsúcsa, ezzel a fa magassága eggyel csökken, és megmarad az a tulajdonság, hogy a fa gyökerének legalább egy kulcsa van (feltéve, hogy a fa nem üres).

Az algoritmus megadása helyett csak azt vázoljuk, hogy a törlés hogyan működik. Egy B-fából történő kulcstörlések különböző esetei a 18.8. ábrán láthatók.

1. Ha a k kulcs az x csúcsban van, és x egy levél, akkor az x kulcsot töröljük az x-ből.
2. Ha a k kulcs az x csúcsban van, és x a fa egy belső csúcsa, akkor a következőket kell végrehajtani:
   a. Ha x-ben a k-t megelőző gyereknek legalább t kulcsa van, akkor keressük meg az y gyökércsúcsú részfában a k-t közvetlenül megelőző k' kulcsot. Rekurzívan töröljük k'-t, és helyettesítsük k-t k'-vel az x-ben. (A k' megkereséséhez és törléséhez egy lefelé haladó menet elegendő.)
   b. Szimmetrikusan, ha a z gyerek következik az x-beli k után, és z-nek legalább t kulcsa van, akkor keressük meg a z gyökércsúcsú részfában a k-t közvetlenül követő k' kulcsot. Rekurzívan töröljük k'-t, és helyettesítsük k-t k'-vel az x-ben. (A k' megkereséséhez és törléséhez egy lefelé haladó menet elegendő.)
   c. Egyébként, ha mind y-nak, mind z-nek csak $t - 1$ kulcsa van, akkor egyesítsük k-t és a z kulcsait y-ba, úgy, hogy x-ből töröljük a k-t és a z-re mutató pointert. Ekkor y-nak $2t - 1$ kulcsa lesz. Ezután szabadítsuk fel z-t és rekurzívan töröljük k-t az y-ból.
3. Ha k nincs benne az x belső csúcsban, akkor határozzuk meg annak a megfelelő részfának a $c_i[x]$ gyökércsúcsát, amelyben a k biztosan benne van, ha egyáltalán benne van k a fában. Ha $c_i[x]$-nek csak $t - 1$ kulcsa van, akkor hajtsuk végre a 3a vagy a 3b pontban leírtakat, mivel biztosítani kell azt, hogy annak a csúcsnak, amelyre lelépünk, legalább t kulcsa legyen. Ezután a műveletet az x megfelelő gyerekén rekurzióval befejezhetjük.
   a. Ha $c_i[x]$-nek csak $t - 1$ kulcsa van, de van egy közvetlen testvére, melynek legalább t kulcsa van, akkor vigyünk le $c_i[x]$-be egy kulcsot x-ből, a $c_i[x]$ közvetlen bal oldali vagy jobb oldali testvérétől pedig vigyünk fel egy kulcsot x-be, és vigyük át

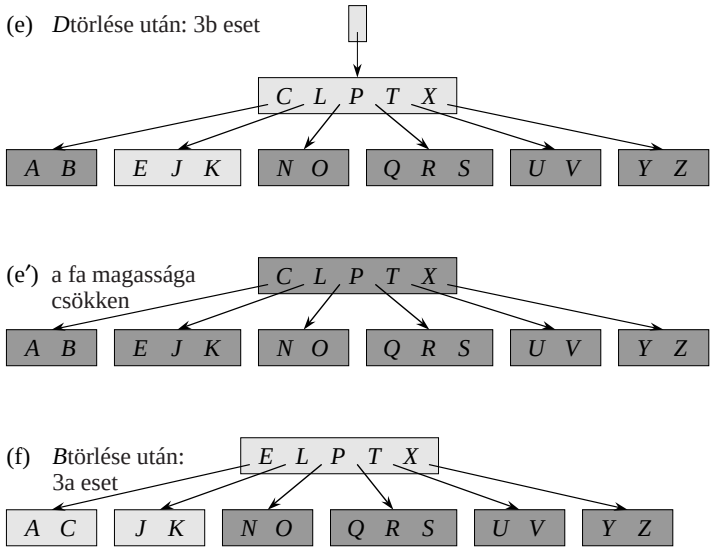


18.8. ábra. Kulcsok törlése egy B-fából. A B-fa minimális fokszáma $t = 3$, így a csúcsoknak (a gyökércsúcsot kivéve) nem lehet kettőnél kevesebb kulcsa. A módosított csúcsokat világosszürkére színeztük. **(a)** A 18.7(e) ábrán látható B-fa. **(b)** Az F törlése. Ez az 1. eset: egy egyszerű törlés a levélből. **(c)** Az M törlése. Ez a 2a eset: L-t, ami az M megelőzője, felvisszük az M helyére. **(d)** A G törlése. Ez a 2c eset: A G-t levisszük, és elkészítjük a $DEGJK$ csúcsot, majd a G-t kitöröljük ebből a csúcsból. **(e)** A D törlése. Ez a 3b eset: a rekurziót nem lehet levinni a CL csúcsba, mivel annak csak 2 kulcsa van, ezért a P-t kell levinni, és egyesíteni a CL-lel és TX-szel a $CLPTX$ csúcsba; ezután a D törölhető a levélből (1. eset). **(e′)** A (d) után a fa gyökércsúcsát töröljük, ezzel a fa magassága eggyel csökken. **(f)** A B törlése. Ez a 3a eset: A C elfoglalja a B helyét, E pedig a C pozícióját.

a megfelelő gyerek mutatóját a testvértől a $c_i[x]$-be.

b. Ha $c_i[x]$-nek és a $c_i[x]$ mindkét közvetlen testvérének $t - 1$ kulcsa van, akkor egyesítsük $c_i[x]$-t az egyik testvérével, és utána vigyünk le egy kulcsot x-ből ebbe az egyesített csúcsba, ez a kulcs lesz az egyesített csúcs középső kulcsa.

Mivel egy B-fában a kulcsok többsége levelekben van, valószínű, hogy a törlés műveletek nagy része levelekből töröl kulcsot. Ekkor a B-FÁBÓL-TÖRÖL eljárás a fában lefelé haladó, egymenetes, visszalépés nélküli eljárás. Ha azonban egy belső csúcsból kell egy



kulcsot törölni, az eljárás egy lefelé haladó menetben megy le a fában, de utána visszatér abba a csúcsba, ahonnan egy kulcsot törölt, azért, hogy ezt a megelőző vagy követő kulccsal helyettesítse (2a és 2b eset).

Bár ez az eljárás bonyolultnak tűnik, egy h magasságú B-fára a lemezműveletek száma $O(h)$, mivel csak $O(1)$ LEMEZRŐL-OLVAS és LEMEZRE-ÍR műveletet hajt végre az eljárás rekurzív hívásai között. A szükséges központi egység időigény $O(th) = O(t \log_t n)$.

Gyakorlatok

18.3-1. A 18.8(f) ábrán látható B-fából rendre a C, P és V kulcsokat töröljük. Adjuk meg a törlések eredményét.

18.3-2. Írjuk meg a B-FÁBÓL-TÖRÖL eljárás algoritmusát.

Feladatok

18-1. Verem a másodlagos tárolón

Vizsgáljuk meg a verem megvalósítását egy olyan számítógépen, amelynek viszonylag kicsi az elsődleges memóriája, de nagyméretű lassú lemeztárolóval rendelkezik. A VEREMBE és a VEREMBŐL műveletek egyszavas értékeket mozgatnak. A verem, amit mi használunk, olyan nagyra is megnőhet, hogy már nem fér el a memóriában, ezért egy részét a lemezen kell tárolni.

Egy egyszerű, de kevésbé hatékony megvalósítás a teljes vermet a lemezen tárolja. A memóriában csak a veremmutató van, amelyben a verem legfelső elemének a címe található. Ha a mutató értéke p, akkor a legfelső elem a lemez $\lfloor p/m \rfloor$-edik lapján a $(p \mod m)$-edik szóban van, ahol m a lapokon található szavak száma.

A VEREMBE művelet végrehajtásakor először növeljük a veremmutatót, beolvassuk a megfelelő lapot a lemezről a memóriába, beírjuk a tárolandó értéket a lap megfelelő szavába, és a lapot visszaírjuk a lemezre. A VEREMBŐL művelet ehhez hasonló. Beolvassuk a megfelelő lapot a lemezről, kiolvassuk belőle a verem tetején levő szót, majd eggyel csökkentjük a veremmutatót. A lapot nem kell visszaírni a lemezre, mert tartalmát nem módosítottuk.

Mivel a lemezműveletek viszonylag lassúak, minden megvalósításnál két költséget veszünk figyelembe: a lemezhozzáférések teljes számát és a teljes központi egység időt. Minden m szóból álló lap lemezművelete egy lemezhozzáférés költségébe és $\Theta(m)$ központi egység időbe kerül.

a. Ezt az egyszerű megvalósítást használva, a legrosszabb esetben aszimptotikusan mekkora az n veremművelet lemezhozzáféréseinek a száma? Mekkora az n veremművelet központi egység ideje? (Az eredményt erre és a további kérdésekre is, n és m függvényével adjuk meg.)

Most tekintsük a veremnek azt a megvalósítását, amelyben a verem egy lapját mindig a memóriában tartjuk. (Arra is elhasználunk egy kevés memóriát, hogy nyilvántartsuk, melyik lap van pillanatnyilag a memóriában.) Ekkor egy veremműveletet csak akkor hajthatunk végre, ha a szükséges lap a memóriában van. Ezért, ha kell, a memóriában levő lapot a lemezre írjuk, és a szükséges lapot a lemezről a memóriába olvassuk. Ha a lap már a memóriában van, akkor természetesen nem hajtunk végre lemezműveleteket.

b. A legrosszabb esetben mennyi lemezművelet szükséges n darab VEREMBE művelet végrehajtásához?

c. A legrosszabb esetben mennyi lemezművelet szükséges n darab veremművelet végrehajtásához? Mennyi a központi egység ideje?

Most tegyük fel, hogy a vermet úgy valósítjuk meg, hogy mindig kettő lap van a memóriában (és még néhány szó a lapok nyilvántartására).

d. Írjuk le, hogyan lehetne kezelni a verem lapjait úgy, hogy egy veremművelet lemezhozzáféréseinek amortizált száma $O(1/m)$, és egy veremművelet amortizált központi egység ideje $O(1)$ legyen.

18-2. 2-3-4 fák egyesítése és szétvágása

Az ***egyesítés*** művelet az S' és az S'' dinamikus halmazokat az x elemmel összekapcsolja, úgy, hogy ha $x' \in S'$ és $x'' \in S''$, akkor $kulcs[x'] < kulcs[x] < kulcs[x'']$. Az eredmény az $S = S' \cup \{x\} \cup S''$ lesz. A ***vágás*** művelet az összekapcsolás „inverze”, ha adott egy S dinamikus halmaz és egy $x \in S$ elem, akkor a vágás művelet eredménye két halmaz lesz, egy olyan S' halmaz, melyben az $S - \{x\}$ halmaz azon elemei lesznek, amelyeknek a kulcsa kisebb, mint $kulcs[x]$, továbbá egy olyan S'' halmaz, melyben az $S - \{x\}$ halmaz azon elemei lesznek, amelyeknek a kulcsa nagyobb $kulcs[x]$-nél. Ebben a feladatban azt vizsgáljuk, hogy hogyan lehet alkalmazni ezeket a műveleteket a 2-3-4 fákra. Az egyszerűség kedvéért feltesszük, hogy az elemek csak kulcsokból állnak, és minden kulcs értéke különböző.

a. Mutassuk meg, hogy hogyan lehet kezelni az x gyökércsúcsú részfa magasságát, ha a 2-3-4 fa minden x csúcsára a magasságot a $magasság[x]$ mezőben tároljuk.

b. Hogyan lehet megvalósítani az összekapcsolás műveletet? Legyen adott két 2-3-4 fa, T' és T'', és a k kulcs, az összekapcsolás műveletnek $O(1 + |h' - h''|)$ időben kell lefutnia, ahol h' és h'' a T', illetve a T'' fa magassága.

c. Tekintsük egy 2-3-4 fa gyökéreleméből egy adott k kulcsához vezető p utat, a T fa k-nál kisebb kulcsainak S', és k-nál nagyobb kulcsainak S'' halmazát. Mutassuk meg, hogy p az S'-t a fák $\{T'_0, T'_1, \ldots, T'_m\}$ és a kulcsok $\{k'_0, k'_1, \ldots, k'_m\}$ halmazára bontja, ahol $i = 1, 2, \ldots, m$-re és minden $y \in T'_{i-1}$ és $z \in T'_i$ kulcsra $y < k'_i < z$. Mi a kapcsolat a T'_{i-1} és a T'_i fák magassága között? Írjuk le, p hogyan osztja fel S''-t fák és kulcsok halmazára.

d. Mutassuk meg, hogyan lehet megvalósítani a T-re vonatkozó vágás műveletet. Használjuk az összekapcsolás műveletet arra, hogy az S' kulcsai egy T' 2-3-4 fába, és az S'' kulcsai a T'' fába kerüljenek. A vágás művelet futási ideje $O(\lg n)$ legyen, ahol n a T kulcsainak száma. (*Útmutatás.* Az összekapcsolás költségei egymásba vihetők).

Megjegyzések a fejezethez

Knuth [185], Aho, Hopcroft és Ullman [5], valamint Sedgewick [269] foglalkozott a kiegyensúlyozott fastruktúrák és B-fák további vizsgálatával. Comer [66] egy átfogó tanulmányt írt a B-fákról. Guibas és Sedgewick [135] vizsgálta a kiegyensúlyozott fastruktúrák különböző típusai közötti kapcsolatokat, beleértve a piros-fekete fákat és a 2-3-4 fákat is.

A 2-3 fákat 1970-ben J. E. Hopcroft írta le, ezekben a fákban a csúcsoknak kettő vagy három gyerekük lehet. A 2-3 fákból alakultak ki a B-fák és a 2-3-4 fák. A B-fákat Bayer és McCreight vezette be 1972-ben [32], de nem adták meg, hogy az elnevezés honnan származik.

Bender, Demaine és Farach-Colton [37] azt vizsgálták, hogy hogyan lehet a B-fákat jól megvalósítani akkor, amikor a memória hierarchikus hatásai is jelen vannak. A ***cache-t nem használó*** algoritmusuk hatékonyan működik annak ellenére, hogy pontosan nem tudjuk, milyen az adatátvitelek mérete a hierarchikus memóriában.

19. Binomiális kupacok

Ebben és a 20. fejezetben az ***összefésülhető kupacoknak*** nevezett adatszerkezetekkel foglalkozunk, és ezekre az adatszerkezetekre a következő öt műveletet alkalmazzuk:

KUPACOT-KÉSZÍT(), egy új üres kupacot készít és ad vissza.

BESZÚR(H,x),] a H kupacba beszúr egy olyan x csúcsot, amelynek a *kulcs* mezője már ki van töltve.

MINIMUM(H),] visszaad egy olyan mutatót, amely a H kupacnak a minimális kulcsú csúcsára mutat.

MINIMUMOT-KIVÁG(H),] a H kupacból kiveszi azt a csúcsot, amelynek a kulcsa minimális, és visszaad egy erre a csúcsra mutató pointert.

EGYESÍT(H_1,H_2),] egy olyan új kupacot készít és ad vissza, amelyik a H_1 és H_2 minden csúcsát tartalmazza. A H_1 és a H_2 kupacok a művelet során „megsemmisülnek”.

Továbbá, az ezekben a fejezetekben szereplő adatszerkezetekre a következő két műveletet is használjuk:

KULCSOT-CSÖKKENT(H,x,k), amelyik a H kupac x csúcsához az új k kulcsértéket rendeli hozzá, feltéve, hogy ez az érték nem nagyobb, mint az eredeti kulcsérték.[1]

TÖRÖL(H,x), a H kupacból az x csúcsot törli.

Mint a 19.1. ábrán is látható, ha nincs szükség az EGYESÍT műveletre, mint például a kupac rendezésben (6. fejezet), akkor az egyszerű bináris kupacok is jól használhatók. Az EGYESÍT művelet kivételével, a műveletek bináris kupacokon legrosszabb esetben $O(\lg n)$, vagy ennél jobb időben futnak le. Ha az EGYESÍT műveletre is szükség van, akkor a bináris kupacok már kevésbé hatékonyak. Az összefésülendő két bináris kupacot tartalmazó két tömböt konkatenálva és lefuttatva a MIN-KUPACOL (lásd a 6.2-2. gyakorlatot) műveletet, az EGYESÍT műveletre a legrosszabb esetben $\Theta(n)$ időt kapunk.

Ebben a fejezetben a „binomiális kupacokat” vizsgáljuk, melyeknek legrosszabb esetre vonatkozó időkorlátai ugyancsak megtalálhatók a 19.1. ábrán levő táblázatban. Speciálisan,

[1] Mint az V. fejezet bevezetőjében említettük, összefésülhető kupacon összefésülhető min-kupacot értünk, és így a MINIMUM, MINIMUMOT-KIVÁG és a KULCSOT-CSÖKKENT műveletek alkalmazhatók. Hasonlóképpen, az ***összefésülhető max-kupacokra*** meghatározhatjuk a MAXIMUM, MAXIMUMOT-KIVÁG és a KULCSOT-NÖVEL műveleteket is.

Eljárás	Bináris kupac (legrosszabb eset)	Binomiális kupac (legrosszabb eset)	Fibonacci-kupac (amortizált)
KUPACOT-KÉSZÍT	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
BESZÚR	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
MINIMUMOT-KIVÁG	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
EGYESÍT	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
KULCSOT-CSÖKKENT	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
TÖRÖL	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

19.1. ábra. Az összefésülhető kupacok három megvalósításában használt műveletek végrehajtási idő adatai. n-nel a kupac (vagy kupacok) művelet alatti elemszámát jelöltük.

az EGYESÍT művelet csak $O(\lg n)$ ideig tart, amíg az összesen n elemből álló két binomiális kupacot egyesíti.

A 20. fejezetben a Fibonacci-kupacokat tanulmányozzuk, amelyek bizonyos műveletekre jobb időkorlátokat tesznek lehetővé. Megjegyezzük azonban, hogy a 19.1. ábrán a Fibonacci-kupacokra megadott futási idők amortizált időkorlátok, és nem legrosszabb esetű időkorlátok.

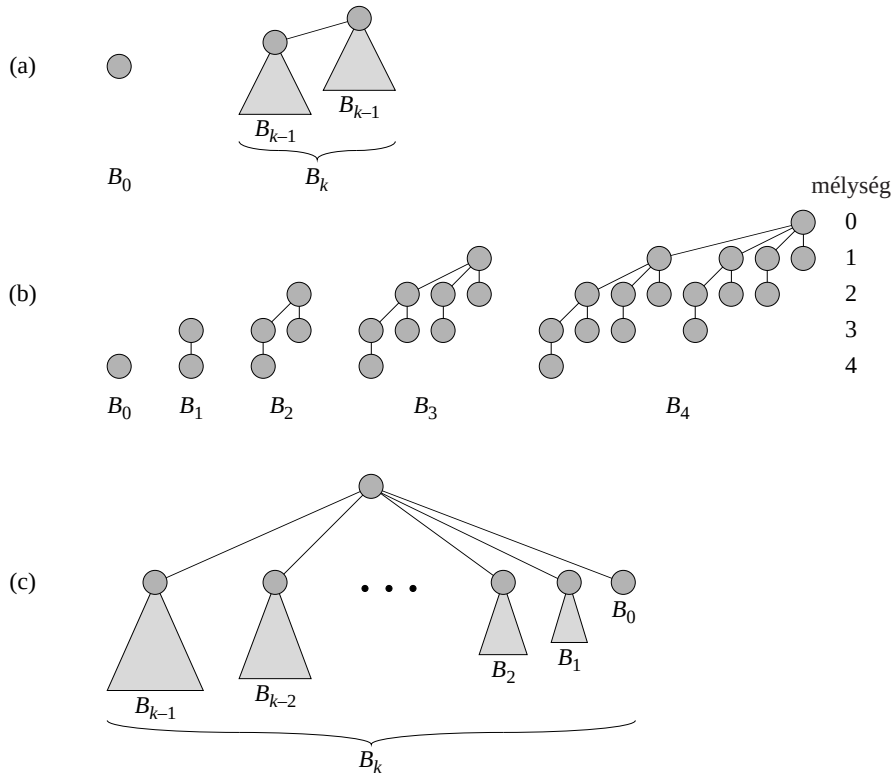
Ebben a fejezetben nem foglalkozunk a beszúrt csúcs helyének meghatározásával, és törlés után a csúcs helyének felszabadításával. Feltesszük, hogy ezekkel a részletekkel a kupac műveleteit meghívó programok foglalkoznak.

A bináris, binomiális és Fibonacci-kupacok egyikében sem lehet hatékonyan végrehajtani a KERES műveletet; ez a művelet egy adott kulcsú csúcs megkeresésével sok időt tölthet el. Ezért az egy megadott csúcsra vonatkozó műveletek, mint például a KULCSOT-CSÖKKENT és a TÖRÖL művelet számára a megadott csúcsra mutató pointert a bemenő paraméterek között kell megadni. Mint a 6.5. szakaszban, amikor az elsőbbségi sorokat vizsgáltuk és egy alkalmazásban egy összefésülhető kupacot használtunk, gyakran tárolunk egy, a megfelelő alkalmazás objektumra mutató nyelet minden összefésülhető kupac elemben, valamint egy, a megfelelő összefésülhető kupac elemre mutató nyelet minden alkalmazás objektumban. Ezeknek a nyeleknek a pontos tulajdonságai az alkalmazásoktól és az alkalmazások megvalósításaitól függnek.

A 19.1. alfejezetben a binomiális kupacokat definiáljuk, miután megadtuk az őket alkotó binomiális fák definícióját. Itt a binomiális kupacok egy speciális ábrázolásával is foglalkozunk. A 19.2. alfejezetben megmutatjuk, hogyan lehet megvalósítani binomiális kupacokra azokat a műveleteket, amelyeknek időkorlátait a 19.1. ábrán adtuk meg.

19.1. Binomiális fák és binomiális kupacok

A binomiális kupacok binomiális fákból állnak, ezért ebben az alfejezetben először a binomiális fákat definiáljuk, és bebizonyítjuk néhány alaptulajdonságukat. Ezután definiáljuk a binomiális kupacokat, és megvizsgáljuk ábrázolásuk módszereit.



19.2. ábra. (a) A B_k binomiális fa rekurzív definíciója. A háromszögek gyökeres részfákat jelölnek. **(b)** Binomiális fák B_0-tól B_4-ig. A B_4 fa mélységét is jelöljük. **(c)** A B_k binomiális fa egy más stílusú ábrázolása.

19.1.1. Binomiális fák

A B_k ***binomiális fa*** egy rekurzív módon definiált rendezett fa (lásd a B.5.2. pontot). A 19.2(a) ábrán látható B_0 binomiális fa egy csúcsból áll. A B_k binomiális fa két ***összekapcsolt*** B_{k-1} binomiális fából áll; az egyik fa gyökércsúcsa a másik fa gyökércsúcsának legbaloldalibb gyereke. A 19.2(b) ábrán a B_0–B_4 binomiális fák láthatók.

A binomiális fák néhány tulajdonságát a következő lemma mondja ki.

19.1. lemma (a binomiális fák tulajdonságai). *Ha B_k binomiális fa, akkor*

1. *2^k csúcsa van,*
2. *a magassága k,*
3. *az i-edik mélységben pontosan $\binom{k}{i}$ csúcs van ($i = 0, 1, \ldots, k$), és*
4. *a gyökércsúcs fokszáma k, ami nagyobb, mint bármely másik csúcs fokszáma; továbbá, ha a gyökércsúcs gyerekeit balról jobbra haladva megszámozzuk $k-1, k-2, \ldots, 0$-val, akkor az i gyerek a B_i részfa gyökércsúcsa.*

Bizonyítás. Az állításokat k szerinti teljes indukcióval bizonyítjuk. Mind a négy állítás triviális a B_0 binomiális fára.

Az indukciós feltétel az, hogy az állítások igazak a B_{k-1} fára.

1. A B_k binomiális fa két B_{k-1} binomiális fából áll, és így B_k-nak $2^{k-1} + 2^{k-1} = 2^k$ csúcsa van.
2. A B_k konstruálása szerint a B_k maximális mélysége eggyel nagyobb, mint a B_{k-1} maximális mélysége. Így az indukciós feltétel szerint a B_k maximális mélysége $(k-1)+1 = k$.
3. Jelöljük $D(k, i)$-vel a B_k binomiális fa i mélységben levő csúcsainak számát. Mivel B_k két B_{k-1} kompozíciója, a B_{k-1} egy i mélységben levő csúcsa a B_k-ban kétszer szerepel, egyszer az i és egyszer az $i + 1$ mélységben. Azaz, a B_k i mélységű csúcsainak száma megegyezik a B_{k-1} i és $i - 1$ mélységű csúcsszámának összegével. Így

$$\begin{aligned} D(k, i) &= D(k-1, i) + D(k-1, i-1) && \text{(az indukciós feltevésből)} \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} && \text{(a C.1-7. gyakorlat alapján)} \\ &= \binom{k}{i}. \end{aligned}$$

4. Csak egy olyan csúcs van a B_k-ban, amelyiknek a fokszáma nagyobb, mint a B_{k-1}-ben, ez a B_k gyökércsúcsa, amelynek eggyel több gyereke van, mint a B_{k-1} gyökércsúcsának. Mivel a B_{k-1} gyökércsúcsának fokszáma $k - 1$, a B_k gyökerének fokszáma k. Az indukciós feltétel szerint a B_{k-1} gyökerének gyerekei balról jobbra haladva a $B_{k-2}, B_{k-3}, \ldots, B_0$ gyökerei, mint az a 19.2(c) ábrán is látható. Amikor a két B_{k-1}-et összekapcsoltuk, az eredményül kapott fa gyökerének gyerekei éppen a $B_{k-1}, B_{k-2}, \ldots, B_0$ gyökércsúcsai. ■

19.2. következmény. *Egy n csúcsú binomiális fa minden csúcsának fokszáma legfeljebb* $\lg n$.

Bizonyítás. A következmény állítása a 19.1. lemma 1. és 4. állításából azonnal adódik. ■

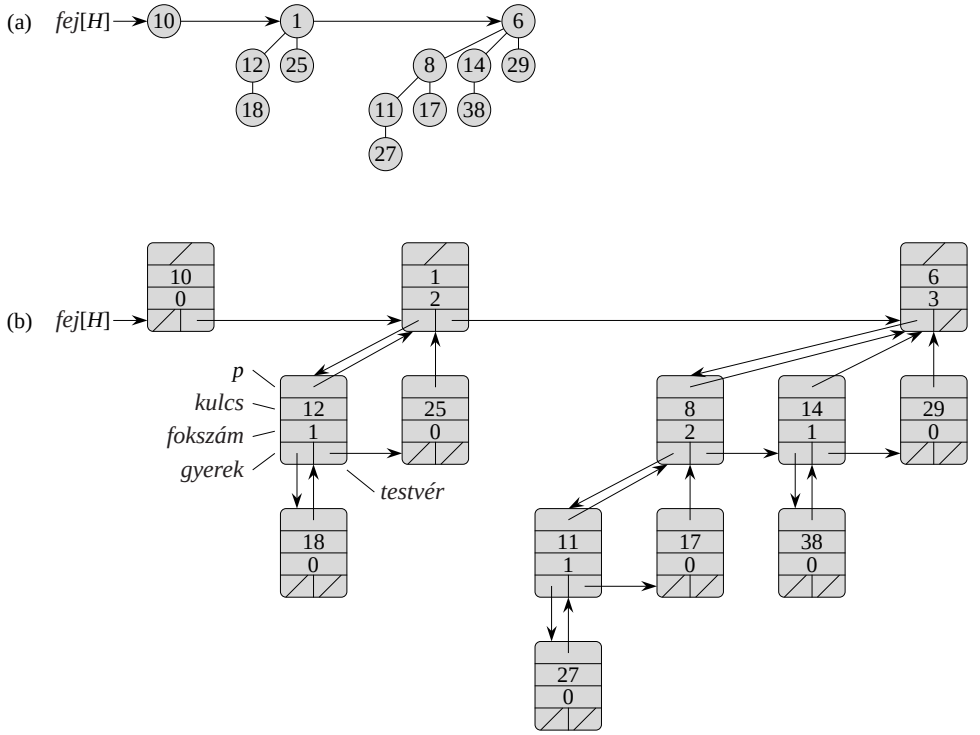
A „binomiális fa” elnevezés a 19.1. lemma 3. állításából származik, mivel a $\binom{k}{i}$ kifejezések a binomiális együtthatók. A 19.1-3. gyakorlatban az elnevezés további indokait is megtalálhatjuk.

19.1.2. Binomiális kupacok

Egy H ***binomiális kupac*** binomiális fák olyan halmaza, amely kielégíti a következő ***binomiális-kupac tulajdonságokat***.

1. H minden binomiális fája rendelkezik a ***min-kupac tulajdonsággal:*** egy csúcs kulcsa nagyobb vagy egyenlő, mint a szülőjének a kulcsa. Ekkor azt mondjuk, hogy ezek a fák ***min-kupac-rendezett*** fák.
2. H-ban legfeljebb egy olyan binomiális fa van, amelyikben a gyökércsúcsnak egy meghatározott fokszáma van.

Az első tulajdonság azt mondja ki, hogy egy min-kupac-rendezett fa gyökércsúcsában van a fa legkisebb kulcsa.



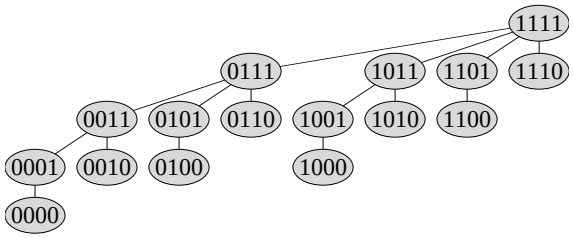
19.3. ábra. Egy H binomiális kupac, amelynek $n = 13$ csúcsa van. **(a)** A kupac a B_0, B_2 és B_3 binomiális fákból áll, a fáknak rendre 1, 4 és 8 csúcsuk van, a teljes csúcsszám 13. Mivel minden binomiális fa min-kupac-rendezett, egy csúcs kulcsa nem kisebb, mint a csúcs szülőjének a kulcsa. A gyökérlista is látható, a gyökérlista a gyökércsúcsok láncolt listája, amely a gyökércsúcsokat fokszámuk növekvő sorrendjében tartalmazza. **(b)** A H binomiális kupac részletesebb ábrázolása. Mindegyik binomiális fát balgyerek-, jobbtestvér-ábrázolással adjuk meg, és mindegyik csúcsban a csúcs fokszámát is tároljuk.

A második tulajdonságból az következik, hogy egy n csúcsból álló H binomiális kupac legfeljebb $\lfloor \lg n \rfloor + 1$ binomiális fából áll. Ez könnyen látható, hiszen az n bináris ábrázolása $\lfloor \lg n \rfloor + 1$ bitből áll, a bináris szám legyen $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \rangle$, ekkor $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. A 19.1. lemma első állítása szerint a B_i binomiális fa akkor és csak akkor van benne H-ban, ha a b_i bitre $b_i = 1$. Így a H binomiális kupac legalább $\lfloor \lg n \rfloor + 1$ binomiális fából áll.

A 19.3(a) ábrán egy 13 csúcsból álló H binomiális kupac látható. A 13 bináris formája $\langle 1101 \rangle$, és a H a B_3, B_2 és a B_0 min-kupac-rendezett binomiális fákat tartalmazza. A fáknak rendre 8, 4 és 1 csúcsuk van.

Binomiális kupacok ábrázolása

Mint a 19.3(b) ábrán is látható, egy binomiális kupacban levő binomiális fákat a 10.4. alfejezetben szereplő baloldaligyerek-, jobboldalitestvér-ábrázolással adjuk meg. Mindegyik csúcsnak van egy *kulcs* mezője, és további, az alkalmazástól függő adatokat tartalmazó mezői. Ezenkívül minden x csúcsban van egy $p[x]$ mutató, amelyik a csúcs szülőjére, egy *gyerek*$[x]$ mutató, amelyik a legbaloldalibb gyerekére, és egy *testvér*$[x]$ mutató, ame-



19.4. ábra. A B_4 binomiális fa, amelynek csúcsait posztorder bejárással binárisan megcímkéztük.

lyik az x jobb oldalán álló első testvérére mutat. Ha az x csúcsnak nincs gyereke, akkor *gyerek*$[x]$ = NIL, és ha az x a szülőjének a legjobboldalibb gyereke, akkor *testvér*$[x]$ = NIL. Mindegyik x csúcsnak van egy *fokszám*$[x]$ mezője is, amelyik az x gyerekeinek számát tartalmazza.

Mint a 19.3. ábrán is látható, egy binomiális kupac binomiális fáinak a gyökércsúcsai egy láncolt listába vannak fűzve, ezt a listát ***gyökérlistának*** nevezzük. A gyökérlista bejárásakor a gyökércsúcsok fokszámai szigorúan növekednek. A binomiális kupacok második tulajdonsága alapján, ha egy binomiális kupacnak n csúcsa van, akkor a gyökércsúcsok fokszámainak halmaza a $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ halmaz részhalmaza. A gyökércsúcsokban a *testvér* mezőnek más a jelentése, mint a nemgyökér csúcsokban. Ha x egy gyökércsúcs, akkor a *testvér*$[x]$ a gyökérlista következő elemére mutat. (Ha x a gyökérlista utolsó eleme, akkor, természetesen, *testvér*$[x]$ = NIL.)

Egy adott H binomiális kupac a *fejelem*$[H]$ mezővel címezhető meg, ez a H gyökérlistájának első gyökércsúcsára mutató pointer. Ha egy H binomiális kupacnak nincs egy eleme sem, akkor *fejelem*$[H]$ = NIL.

Gyakorlatok

19.1-1. Tegyük fel, hogy x egy binomiális kupac egyik binomiális fájának a csúcsa, és tegyük fel, hogy *testvér*$[x] \neq$ NIL. Ha x nem gyökércsúcs, akkor milyen kapcsolat van a *fokszám*[*testvér*$[x]$] és a *fokszám*$[x]$ között? Mit lehet mondani akkor, ha x gyökércsúcs?

19.1-2. Ha x egy binomiális kupac egyik binomiális fájának nem gyökércsúcsa, akkor milyen kapcsolat van a *fokszám*$[x]$ és a *fokszám*$[p[x]]$ között?

19.1-3. Tegyük fel, hogy egy B_k binomiális fa csúcsait posztorder bejárás szerint binárisan megcímkéztük (lásd 19.4. ábra). Tekintsük az i mélységű l címkéjű x csúcsot, amelyre legyen $j = k - i$. Mutassuk meg, hogy ekkor az x címkéjében j darab egyes van. Hány darab olyan k hosszúságú címke van, amelyben pontosan j darab egyes van? Mutassuk meg, hogy x fokszáma megegyezik az l bináris ábrázolásában balról jobbra haladva a legjobboldalibb nulláig található egyesek darabszámával.

19.2. A binomiális kupacokon értelmezett műveletek

Ebben az alfejezetben megmutatjuk, hogy hogyan lehet végrehajtani a binomiális kupacon értelmezett műveleteket a 19.1. ábrán megadott időkorlátokon belül. Csak a felső korlátot mutatjuk meg, az alsó korlátokat a 19.2-10. gyakorlatban tűzzük ki.

Új binomiális kupac készítése

Egy üres binomiális kupac készítésére a BINOMIÁLIS-KUPACOT-LÉTREHOZ eljárás egyszerűen allokáljon és adjon vissza egy olyan H objektumot, amelyre $fejelem[H] = \text{NIL}$. A futási idő $\Theta(1)$.

A minimális kulcs megkeresése

A BINOMIÁLIS-KUPACBAN-MIN eljárás visszaad egy mutatót, amelyik az n csúcsból álló H binomiális kupac minimális kulcsú csúcsára mutat. Ez a megvalósítás feltételezi, hogy nincs olyan kulcs, amelynek az értéke ∞. (Lásd a 19.2-5. gyakorlatot.)

BINOMIÁLIS-KUPACBAN-MIN(H)

```
1  y ← NIL
2  x ← fejelem[H]
3  min ← ∞
4  while x ≠ NIL
5        do if kulcs[x] < min
6              then min ← kulcs[x]
7                   y ← x
8              x ← testvér[x]
9  return y
```

Mivel a binomiális kupac min-kupac-rendezett, a minimális kulcsnak a gyökércsúcsban kell lennie. A BINOMIÁLIS-KUPACBAN-MIN eljárás az összes gyökércsúcsot megvizsgálja, ez legfeljebb $\lfloor \lg n \rfloor + 1$ darab vizsgálat. Az aktuális minimum értéket a *min*-ben tárolja, és az aktuális minimumra az y pointer mutat. Ha a BINOMIÁLIS-KUPACBAN-MIN eljárást a 19.3. ábrán látható binomiális kupacra alkalmazzuk, akkor az 1 kulcsot tartalmazó csúcsra mutató pointert kapjuk eredményül.

Mivel legfeljebb $\lfloor \lg n \rfloor + 1$ gyökércsúcsot kell megvizsgálni, a BINOMIÁLIS-KUPACBAN-MIN eljárás futási ideje $O(\lg n)$.

Két binomiális kupac egyesítése

A további műveletek a két binomiális kupac egyesítésének műveletét szubrutinként fogják alkalmazni. A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás a ciklusában olyan binomiális fákat kapcsol össze, amelyeknek gyökércsúcsa azonos fokszámú. A következő eljárás az y gyökércsúcsú B_{k-1} fát a z gyökércsúcsú B_{k-1} fához kapcsolja hozzá; azaz z az y szülőjévé válik. Így a B_k fa gyökércsúcsa a z csúcs lesz.

BINOMIÁLIS-ÖSSZEKAPCSOLÁS(y, z)

```
1  p[y] ← z
2  testvér[y] ← gyerek[z]
3  gyerek[z] ← y
4  fokszám[z] ← fokszám[z] + 1
```

A BINOMIÁLIS-ÖSSZEKAPCSOLÁS eljárásban $O(1)$ idő alatt lesz az y csúcs a z gyerekeit tartalmazó láncolt lista fejeleme. Ez azért ilyen egyszerű, mivel a binomiális fák balgyerek-, jobbtestvér-ábrázolása éppen a fa rendezettségi tulajdonságának felel meg: a B_k fában a gyökércsúcs legbaloldalibb gyereke egy B_{k-1} fa gyökércsúcsa.

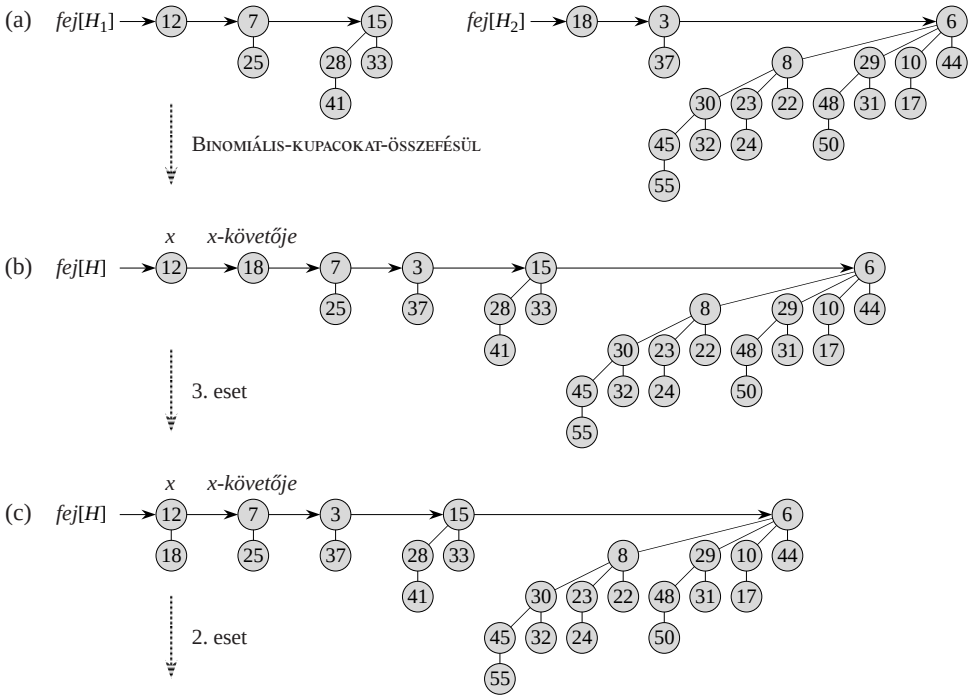
A következő eljárás a H_1 és H_2 binomiális kupacokat egyesíti, és az eredményül kapott kupacot adja vissza. Az egyesítés alatt a H_1 és a H_2 megsemmisül. Az eljárás a BINOMIÁLIS-ÖSSZEKAPCSOLÁS eljáráson kívül használ egy BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárást is, amely összefésüli a H_1 és H_2 gyökérlistáját egy olyan láncolt listába, amelyben az elemek fokszámuk növekvő sorrendjében helyezkednek el. A BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás programja hasonló a 2.3.1. pontban szereplő ÖSSZEFÉSÜL eljárás programjához, az eljárás kódjának megírását a 19.2-1. gyakorlatban tűzzük ki.

BINOMIÁLIS-KUPACOKAT-EGYESÍT(H_1, H_2)

```
 1  H ← BINOMIÁLIS-KUPACOT-LÉTREHOZ()
 2  fejelem[H] ← BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL(H1, H2)
 3  a H1 és H2 objektumok felszabadítása, de a kupacokra mutató listák
    megmaradnak
 4  if fejelem[H] = NIL
 5     then return H
 6  x-megelőzője ← NIL
 7  x ← fejelem[H]
 8  x-követője ← testvér[x]
 9  while x-követője ≠ NIL
10        do if (fokszám[x] ≠ fokszám[x-követője]) vagy
                (testvér[x-követője] ≠ NIL és fokszám[testvér[x-követője]] = fokszám[x])
11              then x-megelőzője ← x                                  ▷ 1. és 2. eset
12                   x ← x-követője                                    ▷ 1. és 2. eset
13              else if kulcs[x] ≤ kulcs[x-követője]
14                      then testvér[x] ← testvér[x-követője]          ▷ 3. eset
15                           BINOMIÁLIS-ÖSSZEKAPCSOLÁS(x-követője,x)   ▷ 3. eset
16                      else if x-megelőzője = NIL                     ▷ 4. eset
17                              then fejelem[H] ← x-követője           ▷ 4. eset
18                              else testvér[x-megelőzője] ← x-követője ▷ 4. eset
19                           BINOMIÁLIS-ÖSSZEKAPCSOLÁS(x,x-követője)   ▷ 4. eset
20                           x ← x-követője                            ▷ 4. eset
21           x-követője ← testvér[x]
22  return H
```

A 19.5. ábrán a BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárásra egy olyan példát láthatunk, amelyben a fenti programnak mind a négy esete előfordul.

A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás két részből áll. Az első rész, amely a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás meghívásakor hajtódik végre, a H_1 és H_2 binomiális kupacok gyökérlistáját egyesíti egy H láncolt listába úgy, hogy benne az elemek monoton növekvő sorrendben lesznek. Minden fokszámhoz kettő (de nem több) gyökércsúcs tartozhat, az eljárás második része az azonos fokszámú gyökércsúcsok átláncolásaival foglalkozik, addig, amíg egy fokszámhoz már csak legfeljebb egy gyökércsúcs tartozik. Mivel



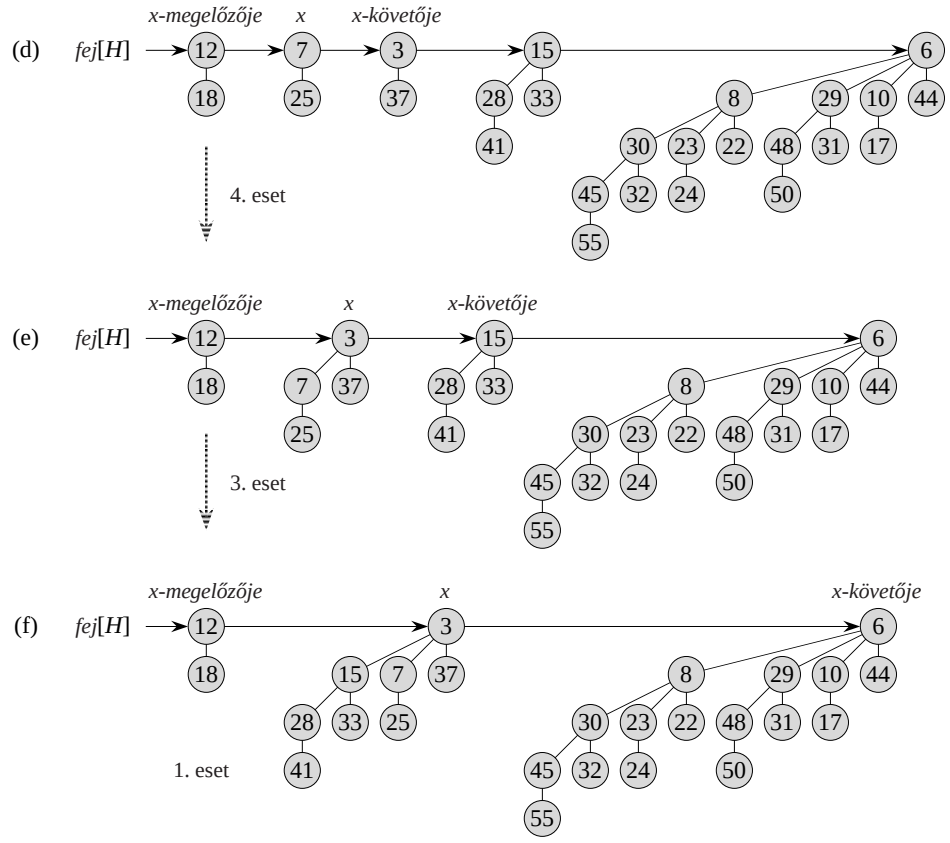
19.5. ábra. A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás végrehajtása. **(a)** A H_1 és H_2 binomiális kupacok. **(b)** A H binomiális kupac a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL(H_1, H_2) végrehajtásának eredménye. Első lépésben az x a H gyökérlista első gyökércsúcsa. Mivel mind az x-nek, mind az x-*követőjé*-nek a fokszáma nulla, és $kulcs[x] < kulcs[x\text{-követője}]$, a 3. esetet kell alkalmazni. **(c)** Az átláncolás végrehajtása után az első három gyökércsúcsnak azonos a fokszáma, ezért a 2. esetet kell alkalmazni. **(d)** Miután mindegyik mutató továbblépett egy pozíciót a listában, a 4. eset alkalmazható, mivel az x egy kettő hosszúságú azonos fokszámú elemeket tartalmazó sorozat első tagja. **(e)** Miután az átláncolást végrehajtottuk, a 3. eset alkalmazható. **(f)** Az átláncolás után az 1. eset alkalmazható, mivel x-nek a fokszáma 3, és az x-*követőjé*-nek a fokszáma 4. Ez a **while** ciklus utolsó iterációja, mivel, miután az x mutató eggyel továbblép a listában, x-*követője* = NIL.

a H láncolt lista fokszám szerint van rendezve, ezeket az átláncolási műveleteket gyorsan végre tudjuk hajtani.

Részletesen az eljárás a következőképpen működik. Az 1–3. sorban a H_1 és H_2 binomiális kupacok gyökérlistáit egy H láncolt listába fűzzük össze. A H_1 és H_2 gyökérlisták fokszám szerint szigorúan növekvő sorrendben vannak rendezve, és a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás visszaad egy fokszám szerint monoton növekvő H gyökérlistát. Ha a H_1 és H_2 gyökérlistának összesen m gyökércsúcsa van, akkor a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás $O(m)$ idő alatt lefut. Az eljárás egy ciklusa a két gyökérlista fejelemét vizsgálja meg és hasonlítja össze, a kisebb fokszámú listaelemet a kimeneti listához fűzi, majd kiveszi ezt az elemet a bemeneti listából.

A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás ezután a H gyökérlistára mutató pointereket állítja be. Először is, az eljárás befejeződik a 4–5. sorban, ha két üres binomiális kupacot kell egyesíteni. Így, a 6. sortól kezdve már biztosan tudjuk, hogy H-nak legalább egy eleme van. Az eljárás végrehajtása alatt három gyökérlistára mutató pointert használunk:

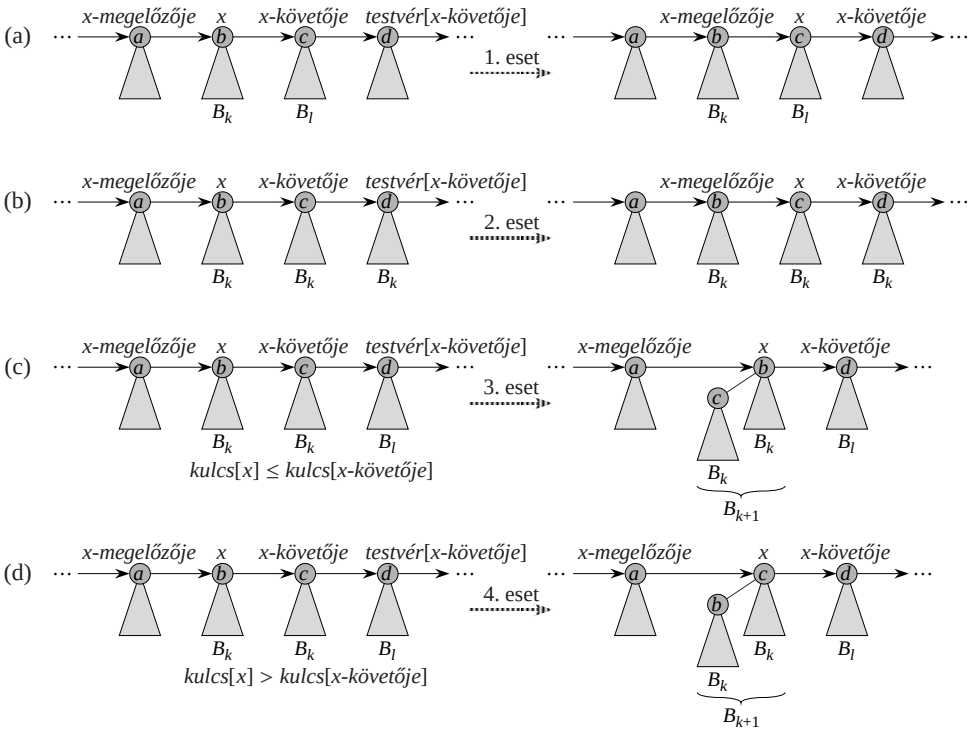
- az x az éppen vizsgált gyökércsúcsra mutat



- *x-megelőzője* az *x*-et a gyökérlistában közvetlenül megelőző gyökércsúcsra mutat, *testvér*[*x-megelőzője*] = *x* (mivel kezdetben az *x*-nek nincs megelőzője, kezdhetjük azzal, hogy a *x-megelőzője*-jét NIL-re állítjuk), és
- *x-követője* az *x*-et a gyökérlistában közvetlenül követő gyökércsúcsra mutat, *testvér*[*x*] = *x-követője*.

Először a H gyökérlistában csak legfeljebb két azonos fokszámú gyökércsúcs lehet: mivel H_1 és H_2 binomiális kupacok, mindegyikben legfeljebb csak egy darab adott fokszámú gyökércsúcs lehet. A BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás biztosítja azt, hogy ha két gyökércsúcsnak azonos a fokszáma, akkor ezek a gyökércsúcsok a H gyökérlistában egymás melletti csúcsok.

A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás végrehajtása alatt azonban előfordulhat, hogy egyidejűleg három azonos fokszámú gyökércsúcs van a H gyökérlistában. Mindjárt megnézzük, hogy ez az eset hogyan fordulhat elő. A 9–21. sorokban levő **while** ciklus minden menetében ezért meg kell határozni, hogy az *x*-nek és a *x-követőjé*-nek, esetleg még a *testvér*[*x-követőjé*]-nek is, azonos-e a fokszáma, ezért vannak-e ebben a sorrendben a listában. A ciklusinvariánsa az, hogy a ciklus törzsébe történő minden belépéskor igaz, hogy sem az *x*, sem a *x-követője* nem NIL. (A pontos ciklusinvariánshoz lásd a 19.2-4. gyakorlatot.)



19.6. ábra. A BINOMIÁLIS-KUPACOKAT-EGYESÍT négy esete. Az a, b, c és d címkék csak a gyökérelemek azonosítására szolgálnak, és nem jelzik az elemek fokszámát vagy kulcsát. Mindegyik esetben x a B_k fa gyökere, és $l > k$. **(a)** 1. eset: *fokszám*[x] ≠ *fokszám*[*x-követője*]. A mutató egy pozícióval továbblép a gyökérlistában. **(b)** 2. eset: *fokszám*[x] = *fokszám*[*x-követője*] = *fokszám*[*testvér*[*x-követője*]]. Az előzőhöz hasonlóan, a mutató egy pozícióval továbblép a listában, és a következő iterációban a 3. vagy a 4. eset következik. **(c)** 3. eset: *fokszám*[x] = *fokszám*[*x-követője*] ≠ *fokszám*[*testvér*[*x-követője*]], és *kulcs*[x] ≤ *kulcs*[*x-követője*]. A gyökérlistából az *x-követője* elemet töröljük és az x-hez kapcsoljuk, ezzel egy B_{k+1} fát készítünk. **(d)** 4. eset: *fokszám*[x] = *fokszám*[*x-követője*] ≠ *fokszám*[*testvér*[*x-követője*]], és *kulcs*[*x-követője*] ≤ *kulcs*[x]. A gyökérlistából az x elemet töröljük és az *x-követője* elemhez kapcsoljuk, ezzel ismét egy B_{k+1} fát készítünk.

Az 1. eset (lásd 19.6(a) ábra) akkor fordul elő, ha *fokszám*[x] ≠ *fokszám*[$x-$ *követője*], azaz amikor x a B_k fa gyökere és *x-követője* egy B_l fa gyökércsúcsa valamilyen $l > k$-ra. Ennek az esetnek a programja a 11–12. sorokban van. Az x és *x-követője* csúcsokat nem kapcsoljuk össze, így csak a lista mutatóját léptetjük tovább egy pozícióval. Az új x csúcsot követő elemre mutató *x-követője* pointert a 21. sorban módosítjuk, ezt a sort már mindegyik esetben végrehajtjuk.

A 2. eset (lásd a 19.6(b) ábrán) akkor fordul elő, ha x első tagja egy három azonos fokszámú elemet tartalmazó sorozatnak, azaz amikor

$$\textit{fokszám}[x] = \textit{fokszám}[\textit{x-követője}] = \textit{fokszám}[\textit{testvér}[\textit{x-követője}]].$$

Ezt az esetet az 1. esethez hasonló módon kezeljük: csak a mutatót léptetjük egy pozícióval tovább. A következő iteráció a 3. vagy a 4. esetet fogja végrehajtani, és a három egyenlő

fokszámú gyökér közül a másodikat és a harmadikat fogja egyesíteni. A 10. sorban mind az 1., mind a 2. esetet vizsgáljuk, és a 11–12. sorban van mindkét eset programja.

A 3. és 4. eset akkor fordul elő, amikor az x két egymás utáni azonos fokszámú gyökércsúcs első tagja, azaz

$$\mathit{fokszám}[x] = \mathit{fokszám}[\mathit{x\text{-követője}}] \neq \mathit{fokszám}[\mathit{testvér}[\mathit{x\text{-követője}}]].$$

Ezek az esetek bármelyik iterációban előfordulhatnak, de a 2. esetet követően e két eset egyike biztosan bekövetkezik. A 3. és 4. esetben az x és az *x-követője* csúcsokat kapcsoljuk össze. A két eset abban különbözik, hogy az x-nek vagy a *x-követőjé*-nek van-e kisebb kulcsa, ugyanis ez az érték határozza meg azt, hogy az összeláncolás után melyik lesz a gyökércsúcs.

A 19.6(c) ábrán látható 3. esetben, amelyik $\mathit{kulcs}[x] \leq \mathit{kulcs}[\mathit{x\text{-követője}}]$, ezért az *x-követőjé*-t láncoljuk az x-hez. A 14. sorban lévő utasítás törli *x-követőjé*-t a gyökérlistából, és a 15. sorban lesz az *x-követője* az x legbaloldalibb gyereke.

A 4. esetben, amely a 19.6(d) ábrán látható, *x-követőjé*-nek van a legkisebb kulcsa, ezért az x-et láncoljuk a *x-követőjé*-hez. A 16–18. sorok programja kiveszi x-et a gyökérlistából. A program egy elágazást hajt végre, ha x az első gyökér a listában, akkor a 17. sor, ellenkező esetben a 18. sor hajtódik végre. A 19. sorban lesz az x az *x-követőjé*-nek a legbaloldalibb gyereke, és a 20. sor módosítja az x-et a következő iterációhoz.

Mind a 3., mind a 4. eset után a **while** ciklus következő iterációjának előkészítése azonos. Ezekben az esetekben két B_k fából egy B_{k+1} fát hoztunk létre, és az x most erre mutat. A gyökérlistában vagy ilyen fa még nem volt, vagy már volt egy vagy kettő B_{k+1} fa, mint a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL művelet eredménye, így x most az első tagja egy egy-, két- vagy háromelemű, B_{k+1} fákból álló sorozatnak. Ha x az egyedüli ilyen gyökércsúcs, akkor a következő iterációban az 1. eset következik: $\mathit{fokszám}[x] \neq \mathit{fokszám}[\mathit{x\text{-követője}}]$. Ha x egy kételemű sorozat első tagja, akkor a következő iterációban a 3. vagy 4. eset következik. Ha az x egy háromelemű sorozat első tagja, akkor a következő iterációban a 2. esetet hajtjuk végre.

A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás futási ideje $O(\lg n)$, ahol n a H_1 és a H_2 binomiális kupacok csúcsainak együttes száma. Ez a következőkből adódik. Legyen H_1-nek n_1, a H_2-nek n_2 csúcsa, és legyen $n = n_1 + n_2$. Ekkor H_1-nek legfeljebb $\lfloor \lg n_1 \rfloor + 1$, H_2-nek legfeljebb $\lfloor \lg n_2 \rfloor + 1$ gyökércsúcsa van, így H-nak legfeljebb $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2\lfloor \lg n \rfloor + 2 = O(\lg n)$ gyökércsúcsa van a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás végrehajtása után. Így a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL eljárás végrehajtásának ideje $O(\lg n)$. A **while** ciklus minden iterációja $O(1)$ idejű, és legfeljebb $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ iteráció van, mivel minden egyes iterációban vagy az x mutató lép tovább egy pozícióval a H gyökérlistájában, vagy egy gyökércsúcsot törlünk a gyökérlistából. A teljes végrehajtási idő tehát $O(\lg n)$.

Egy csúcs beszúrása

A következő eljárás a H binomiális kupacba beszúr egy x csúcsot, feltesszük, hogy az x már allokálva van, és hogy a csúcs *kulcs*$[x]$ mezője már ki van töltve.

BINOMIÁLIS-KUPACBA-BESZÚR(H, x)

1 $H' \leftarrow$ BINOMIÁLIS-KUPACOT-LÉTREHOZ()
2 $p[x] \leftarrow$ NIL
3 *gyerek*$[x] \leftarrow$ NIL
4 *testvér*$[x] \leftarrow$ NIL
5 *fokszám*$[x] \leftarrow 0$
6 *fejelem*$[H'] \leftarrow x$
7 $H \leftarrow$ BINOMIÁLIS-KUPACOKAT-EGYESÍT(H, H')

Az eljárás egyszerűen létrehoz egy egy csúcsból álló H' binomiális kupacot, és ezt egyesíti a H binomiális kupaccal $O(\lg n)$ idő alatt. A BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárás végzi el az ideiglenes H' binomiális kupac felszabadítását. (A 19.2-8. gyakorlatban kitűzzük egy olyan megoldás elkészítését, amelyik nem hívja meg a BINOMIÁLIS-KUPACOKAT-EGYESÍT eljárást.)

A minimális kulcsú csúcs kivágása

A következő eljárás egy H binomiális kupacból kiveszi a minimális kulcsú csúcsot, és eredményül visszaad egy mutatót, amely erre a kivágott csúcsra mutat.

BINOMIÁLIS-KUPACBÓL-MINIMUMOT-KIVÁG(H)

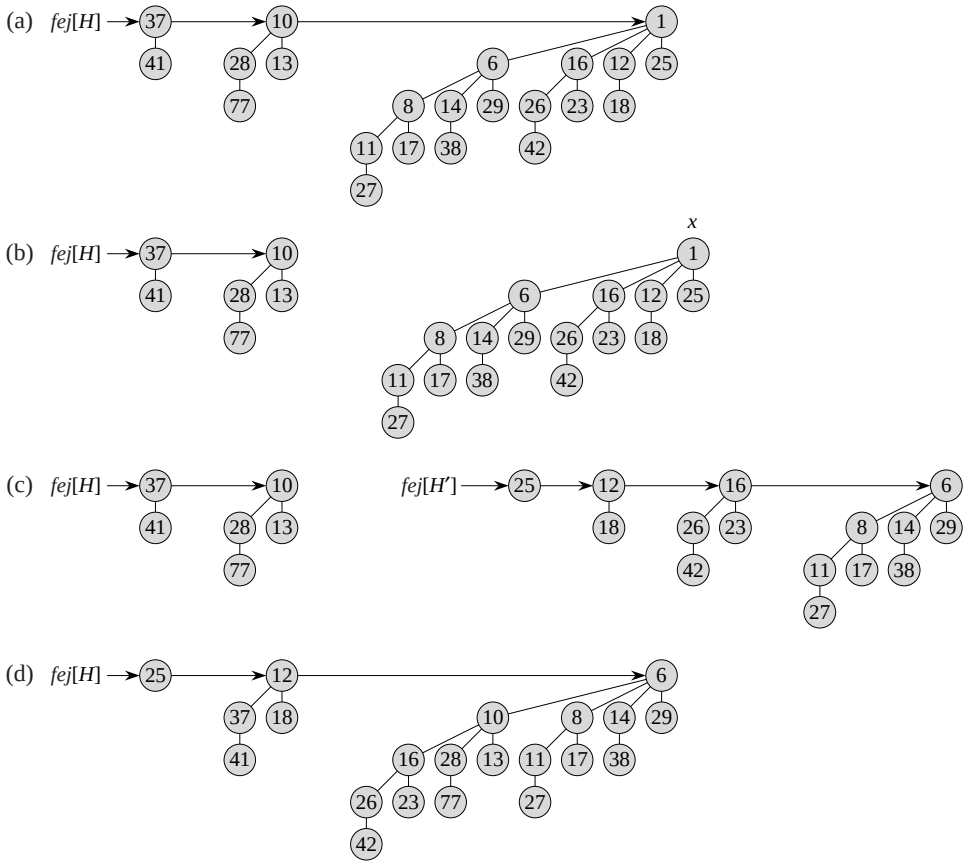
1 a H gyökérlistájában a minimális kulcsú gyökérelem megkeresése, és ennek az elemnek a H gyökérlistájából való kiláncolása
2 $H' \leftarrow$ BINOMIÁLIS-KUPACOT-LÉTREHOZ()
3 az x gyerekeit tartalmazó egyirányú listában a láncolás sorrendjét fordítsuk meg, és mutasson a *fejelem*$[H']$ az így kapott lista fejelemére
4 $H \leftarrow$ BINOMIÁLIS-KUPACOKAT-EGYESÍT(H, H')
5 **return** x

Ennek az eljárásnak a működését a 19.7. ábrán láthatjuk. A bemenő H binomiális kupacot a 19.7(a) ábra tartalmazza. Az 1. sor végrehajtása utáni állapot a 19.7(b) ábrán látható: a minimális kulcsú x gyökércsúcsot kiláncoltuk a H gyökérlistájából. Ha az x egy B_k fa gyökércsúcsa, akkor a 19.1. lemma 4. állítása szerint az x gyerekei, balról jobbra haladva, a $B_{k-1}, B_{k-2}, \ldots, B_0$ fák. A 19.7(c) ábra azt mutatja, hogy az x gyerekeinek láncolását a 3. sorban leírtak szerint megfordítva egy olyan H' binomiális kupacot kapunk, amelyik az x kivételével az x-hez tartozott fa összes csúcsát tartalmazza. Mivel az x-hez tartozó fát az 1. sorban már kivettük a H-ból, a H és H' egyesítésével adódó binomiális kupac az x kivételével a H összes csúcsát tartalmazza (19.7(d) ábra).

Mivel az 1–4. sorok mindegyike, ha a H-nak n eleme van, $O(\lg n)$ idejű, a BINOMIÁLIS-KUPACBÓL-MINIMUMOT-KIVÁG eljárás időigénye $O(\lg n)$.

Egy kulcs csökkentése

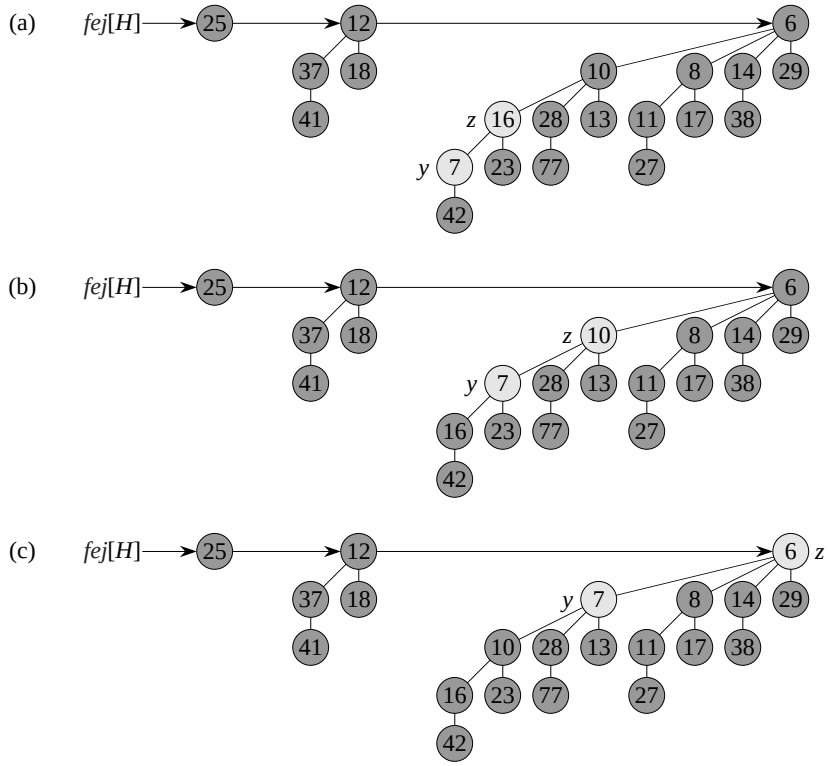
A következő eljárás egy H binomiális kupac x csúcsának kulcsát egy adott k értékre csökkenti. Az eljárás hibajelzést ad, ha k nagyobb, mint az x kulcsának pillanatnyi értéke.



19.7. ábra. A BINOMIÁLIS-KUPACBÓL-MINIMUMOT-KIVÁG eljárás működése. **(a)** A H binomiális kupac. **(b)** A minimális kulcsú x gyökércsúcsot kiláncoljuk a H gyökérlistájából. **(c)** Az x gyerekeinek láncolását megfordítjuk, így kapjuk a H' binomiális kupacot. **(d)** A H és H' egyesítésének eredménye.

BINOMIÁLIS-KUPACBAN-KULCSOT-CSÖKKENT(H, x, k)

```
1  if k > kulcs[x]
2     then error „az új kulcs nagyobb, mint az aktuális kulcs”
3  kulcs[x] ← k
4  y ← x
5  z ← p[y]
6  while z ≠ NIL és kulcs[y] < kulcs[z]
7        do csere kulcs[y] ↔ kulcs[z]
8        ▷ Ha van y-hoz és z-hez tartozó további mező, akkor azokat is cseréljük meg.
9           y ← z
10          z ← p[y]
```



19.8. ábra. A BINOMIÁLIS-KUPACBAN-KULCSOT-CSÖKKENT eljárás működése. **(a)** A **while** ciklus első iterációjában az 6. sor végrehajtása előtti állapot. Az *y* csúcs kulcsát 7-re kell csökkenteni, ez az érték kisebb, mint az *y* szülőjének, *z*-nek a kulcsa. **(b)** A két csúcs kulcsát felcseréljük, a második iterációnak az 6. sor végrehajtása előtti állapota látható az ábrán. Az *y* és a *z* mutatók a fában egy szinttel magasabban vannak, de a min-kupac rendezettsége még mindig nem teljesül. **(c)** A következő csere és a pointerek még egy szinttel magasabbra kerülésekor azt kapjuk, hogy a min-kupac rendezett lett, ezért a **while** ciklus befejeződik.

Mint a 19.8. ábrán is látható, ez az eljárás ugyanolyan módszerrel csökkenti a kulcsot, mint amilyen módszerrel egy kulcs „felfelé buborékol” a bináris min-kupacokban. Miután ellenőriztük, hogy az új kulcs valóban nem nagyobb, mint az aktuális kulcs, az új kulcsot beírjuk az *x*-be, majd az eljárás felfelé halad a fában az *y* mutatót használva, az *y* az első lépésben az *x*-re mutat. A ***while*** ciklus minden iterációjában (6–10. sor) a *kulcs*[*y*]-t és az *y* csúcs *z* szülőjének kulcsértékét hasonlítjuk össze. Ha *y* a gyökércsúcs vagy *kulcs*[*y*] $\geq$ *kulcs*[*z*], akkor a binomiális fa már min-kupac-rendezett. Egyébként az *y* csúcs nem elégíti ki a min-kupac-rendezettség feltételét, és ezért a kulcsát a többi hozzátartozó információval együtt a *z* adataival fel kell cserélni. Az eljárás ezután az *y* mutatót a *z*-re állítja, ezzel a fában egy szinttel feljebb lépünk, és az eljárás a következő iteráció végrehajtásával folytatódik.

A BINOMIÁLIS-KUPACBAN-KULCSOT-CSÖKKENT eljárás végrehajtásához $O(\lg n)$ idő kell. A 19.1. lemma 2. állítása szerint az *x* maximális mélysége $\lfloor \lg n \rfloor$, így a 6–10. sorokban levő **while** ciklus legfeljebb $\lfloor \lg n \rfloor$-szer hajtódik végre.

Egy kulcs törlése

Egy H binomiális kupacban az x csúcs kulcsának és a csúcshoz tartozó információnak a törlése nem nehéz feladat, időigénye $O(\lg n)$. A következő megvalósításban feltesszük, hogy nincs a binomiális kupacban olyan csúcs, amelynek a kulcsa $-\infty$.

BINOMIÁLIS-KUPACBÓL-TÖRÖL(H, x)

1 BINOMIÁLIS-KUPACBAN-KULCSOT-CSÖKKENT(H, x, $-\infty$)
2 BINOMIÁLIS-KUPACBÓL-MINIMUMOT-KIVÁG(H)

A BINOMIÁLIS-KUPACBÓL-TÖRÖL eljárás az x csúcs kulcsát a $-\infty$ értékre állítja, így ez lesz az egész binomiális kupac minimális kulcsú csúcsa. (A 19.2-6. gyakorlat foglalkozik azzal az esettel, amikor a $-\infty$ érték még időlegesen sem lehet egy kulcs értéke.) A BINOMIÁLIS-KUPACBAN-KULCSOT-CSÖKKENT végrehajtása után ez a csúcs felkerül a gyökércsúcsba. Ezt a csúcsot a H-ból a BINOMIÁLIS-KUPACBÓL-MINIMUMOT-KIVÁG eljárás veszi ki.

A BINOMIÁLIS-KUPACBÓL-TÖRÖL eljárás időigénye $O(\lg n)$.

Gyakorlatok

19.2-1. Írjuk meg a BINOMIÁLIS-KUPACOKAT-ÖSSZEFÉSÜL algoritmusát.

19.2-2. A 19.7(d) ábrán látható binomiális kupacba beszúrunk egy olyan csúcsot, amelynek a kulcsa 24. Adjuk meg az eredményül kapott binomiális kupacot.

19.2-3. A 19.8(c) ábrán látható binomiális kupacból töröljük a 28 kulcsú csúcsot. Mutassuk meg az eredményül kapott binomiális kupacot.

19.2-4. Mutassuk meg a BINOMIÁLIS-KUPACOKAT-EGYESÍT algoritmus helyességét úgy, hogy a következő ciklusinvariánst használjuk:

> A 9–21. sorokban levő ***while*** ciklus mindegyik iterációjának kezdetén x a gyökérpontra mutat, ami a következők egyike lehet:
>
> - a fokszámának megfelelő gyökérpont,
> - a fokszámának megfelelő két gyökérpont közül az első,
> - a fokszámának megfelelő három gyökérpont közül az első vagy a második.
>
> Ezenkívül, a gyökérlistában az x megelőzőjét megelőző minden gyökérelemnek egyedi fokszáma van, és ha x megelőzőjének x fokszámától különböző fokszáma van, x megelőzőjének is egyedi fokszáma van a gyökérlistában. Végül, a csúcsok fokszámai monoton növekednek, ahogy a gyökérlistát bejárjuk.

19.2-5. Magyarázzuk meg, hogy a BINOMIÁLIS-KUPACBAN-MIN eljárás miért nem működik helyesen, ha a kulcsok felvehetik a ∞ értéket. Írjuk át az algoritmust úgy, hogy ilyen esetben is jól működjön.

19.2-6. Tegyük fel, hogy a $-\infty$ kulcs nem ábrázolható. Írjuk újra a BINOMIÁLIS-KUPACBÓL-TÖRÖL eljárást úgy, hogy ebben az esetben is jól működjön. Az eljárás időigénye maradjon $O(\lg n)$.

19.2-7. Vizsgáljuk meg a binomiális kupacba történő beszúrás és egy bináris szám eggyel történő növelése közötti kapcsolatot, valamint a két binomiális kupac és két bináris szám összeadása közötti kapcsolatot.

19.2-8. A 19.2-7. gyakorlat eredményét felhasználva írjuk újra a BINOMIÁLIS-KUPACBA-BESZÚR eljárást úgy, hogy az eljárás egy elem beszúrását közvetlenül, a BINOMIÁLIS-KUPACOKAT-EGYESÍT meghívása nélkül végezze el.

19.2-9. Mutassuk meg, hogy ha egy gyökérlista a fokszámok szigorúan csökkenő sorrendjében rendezett (a szigorúan növekvő sorrend helyett), akkor mindegyik binomiális kupac műveletet meg lehet valósítani úgy, hogy az aszimptotikus időigény nem változik meg.

19.2-10. Keressünk olyan bemenő adatokat, amelyekre BINOMIÁLIS-KUPACBÓL MINIMUMOT-KIVÁG, BINOMIÁLIS-KUPACBAN-KULCSOT-CSÖKKENT és BINOMIÁLIS-KUPACBÓL-TÖRÖL $\Omega(\lg n)$ időben fut le. Magyarázzuk meg, hogy BINOMIÁLIS-KUPACBA-BESZÚR, BINOMIÁLIS-KUPACBAN-MIN és BINOMIÁLIS-KUPACOKAT-EGYESÍT legrosszabb futási ideje $\overset{\infty}{\Omega}(\lg n)$, és nem $\Omega(\lg n)$. (Lásd a 3-5. feladatot.)

Feladatok

19-1. 2-3-4 kupacok

A 19. fejezetben foglalkoztunk a 2-3-4 fákkal, amelyekben minden belső (nem gyökér) csúcsnak kettő, három vagy négy gyereke van, és minden levél azonos mélységben van. Ebben a feladatban a ***2-3-4 kupacokat*** valósítjuk meg, és ezekre a kupacokra is alkalmazzuk az összefésülhető kupac műveleteket.

A 2-3-4 kupacok abban különböznek a 2-3-4 fáktól, hogy a 2-3-4 kupacokban csak a leveleknek van kulcs mezőjük, mindegyik x levélnek csak pontosan egy kulcsa van (ez a $kulcs[x]$ mezőben található), nincs speciális rendezés a levelek kulcsai között (azaz balról jobbra haladva, a kulcsok tetszőleges sorrendben lehetnek), mindegyik x belső csúcsnak van egy $kicsi[x]$ értéke, amelyik az x gyökerű részfa levelein levő kulcsok minimuma, az r gyökércsúcsnak van egy $magasság[r]$ mezője, amelyik a fa magasságát tartalmazza. Végül, a 2-3-4 kupacokat a központi memóriában célszerű tárolni azért, hogy ne legyen szükség lemezírás és -olvasás műveletekre.

Valósítsuk meg a következő 2-3-4 kupac műveleteket. Egy n elemű 2-3-4 kupacon az (a)–(e) műveletek mindegyikének le kell futnia $O(\lg n)$ időben. Az (f)-ben leírt EGYESÍT műveletnek le kell futnia $O(n)$ időben, ahol n a két bemenő kupac elemszámainak összege.

a. MIN, amelyik a legkisebb kulcsú levélre mutató pointert adja vissza.

b. KULCSOT-CSÖKKENT, amelyik egy megadott x levél kulcsát egy megadott $k \leq kulcs[x]$ értékre csökkenti.

c. BESZÚR, amelyik a k kulcsú x csúcsot beszúrja.

d. TÖRÖL, amelyik egy adott x levelet kitöröl.

e. MINIMUMOT-KIVÁG, amelyik kiveszi a legkisebb kulcsú levelet.

f. EGYESÍT, amelyik egyesít két 2-3-4 kupacot, visszaad egy 2-3-4 kupacot, és a bemeneti kupacokat megszünteti.

19-2. Összefésülhető kupacokat használó minimális feszítőfa algoritmus

Egy irányítatlan gráf minimális feszítőfájának megkeresésére a 23. fejezetben két algoritmust adtunk meg. Most megmutatjuk, hogyan használhatók a binomiális kupacok egy másik minimális feszítőfa algoritmus kidolgozásában.

Legyen adott egy összefüggő, irányítatlan $G = (V, E)$ gráf, amelynek a súlyfüggvénye $w : E \to \mathbf{R}$. A $w(u, v)$-t az (u, v) él súlyának nevezzük. A G minimális feszítőfáját akarjuk megkeresni: egy olyan körmentes $T \subseteq E$ részhalmazt, amelyik összeköti a V minden csúcsát, és amelyre a

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

teljes súly minimális.

A következő algoritmus, amelyiknek a helyességét a 23.1. alfejezetben használt módszerrel be lehet bizonyítani, a T minimális feszítőfát adja meg. Az algoritmus a V csúcshalmaz $\{V_i\}$ részhalmazát használja, ahol mindegyik V_i-hez tartozik egy

$$E_i \subseteq \{(u, v) : u \in V_i \text{ vagy } v \in V_i\}$$

halmaz, amely olyan élekből áll, amelyeknek legalább az egyik végpontja V_i-be esik.

```
MFF(G)
 1  T ← ∅
 2  for minden v_i ∈ V[G] csúcsra
 3      do V_i ← {v_i}
 4         E_i ← {(v_i, v) ∈ E[G]}
 5  while egynél több V_i halmaz van
 6      do válasszunk egy V_i halmazt
 7         vegyük ki E_i-ből a minimális súlyú (u, v) élt,
 8         az általánosság megszorítása nélkül tegyük fel, hogy u ∈ V_i és v ∈ V_j
 9         if i ≠ j
10           then T ← T ∪ {(u, v)}
11                V_i ← V_i ∪ V_j,  V_j megszűnik
12                E_i ← E_i ∪ E_j
```

Írjuk le, hogy hogyan lehet megvalósítani a fenti algoritmust a 19.1. ábrán megadott binomiális kupac műveletekkel. Meg kell változtatnunk a binomiális kupac ábrázolását? A 19.1. ábrán lévő összefésülhető kupac műveleteken kívül szükségesek újabb műveletek? Adjuk meg a megvalósítás futási idejét.

Megjegyzések a fejezethez

A binomiális kupacokat 1978-ban Vuillemin [307] definiálta. Tulajdonságaikat részletesen Brown [49, 50] tanulmányozta.

20. Fibonacci-kupacok

A 19. fejezetben láttuk, hogy az összefésülhető-kupacokra alkalmazott BESZÚR, MIN, MINIMUMOT-KIVÁG és EGYESÍT műveletek, valamint a KULCSOT-CSÖKKENT és a TÖRÖL műveletek binomiális kupacokon $O(\lg n)$ legrosszabb futási időt adnak. Ebben a fejezetben a Fibonacci-kupacokat vizsgáljuk, és látni fogjuk, hogy ha nem kell elemeket törölni, akkor ugyanezekre a műveletekre a sokkal jobb $O(1)$ amortizált futási időt kapjuk.

Elméleti szempontból a Fibonacci-kupacok különösen jól alkalmazhatók, ha a MINIMUMOT-KIVÁG és a TÖRÖL műveleteket kevesebbszer kell végrehajtani, mint a többi műveletet. Ez az eset sok alkalmazásban előfordul. Például, egyes gráfproblémákat megoldó algoritmusok a gráfok minden élére csak egyszer hívják meg a KULCSOT-CSÖKKENT műveletet. Sűrű gráfokra, amelyeknek sok élük van, az $O(1)$-es amortizált idejű KULCSOT-CSÖKKENT hívások jelentősen javítják a bináris vagy binomiális kupacokra kapott $O(\lg n)$ legrosszabb futási időt. A Fibonacci-kupacok fő alkalmazásait azok a gyors algoritmusok adják, amelyek olyan problémák megoldására szolgálnak, mint a minimális feszítőfák számítása (23. fejezet) és az egy csúcsból induló legrövidebb utak megkeresése (24. fejezet).

Gyakorlati szempontból a legtöbb alkalmazásban a Fibonacci-kupacok – kedvezőtlen állandóik és programozási bonyolultságuk miatt – kevésbé előnyösek, mint a közönséges bináris (vagy k-s) kupacok. Így a Fibonacci-kupacok elsősorban elméleti szempontból érdekesek. Ha azonban több, a Fibonacci-kupacokkal azonos amortizált időkorlátú egyszerűbb adatszerkezetet fejlesztünk, akkor gyakorlati hasznuk is van.

A binomiális kupachoz hasonlóan a Fibonacci-kupac is fákból áll. Valójában azonban a Fibonacci-kupacok csak lazán kapcsolódnak a binomiális kupacokhoz. Ha a KULCSOT-CSÖKKENT és a TÖRÖL műveleteket egy Fibonacci-kupacra egyszer sem alkalmazzuk, akkor a kupac minden fája olyan, mint egy binomiális fa. A Fibonacci-kupacoknak szabadabb szerkezetük van, mint a binomiális kupacoknak, és mindemellett aszimptotikus időkorlátokat tesznek lehetővé. A szerkezet karbantartását el lehet halasztani egészen addig, amikor az már kényelmesen elvégezhető.

A 17.4. alfejezetben szereplő dinamikus táblázatokhoz hasonlóan, a Fibonacci-kupacok is jó példák olyan adatszerkezetekre, amelyeket az amortizált elemzés figyelembevételével terveznek. A fejezet hátralévő részében levő, a Fibonacci-kupacokra vonatkozó műveletek ötlete és elemzése igen erősen támaszkodik a 17.3. alfejezetben szereplő potenciál módszerre.

Ebben a fejezetben feltételezzük, hogy a binomiális kupacokról szóló 19. fejezet anyaga már ismert. A műveletek specifikációi is abban a fejezetben találhatók, a 19.1. ábra táblázatában a bináris kupacok, a binomiális kupacok és a Fibonacci-kupacok műveleteinek id ő-korlátait már megmutattuk. A Fibonacci-kupacok szerkezetének bemutatása a binomiális kupacok szerkezetére épül, és a Fibonacci-kupacokra alkalmazott néhány művelet nagyon hasonló a binomiális kupacokra alkalmazott művelethez.

A binomiális kupacokhoz hasonlóan, a Fibonacci-kupacokat sem úgy tervezték meg, hogy bennük a KERES művelet hatékonyan elvégezhet ő legyen; egy adott csúcsra vonatkozó művelethez ezért az adott csúcsra mutató pointer szükséges, és ez a pointer lesz a művelet bemenete. Amikor egy alkalmazásban Fibonacci-kupacot használunk, gyakran tárolunk egy, a megfelelő alkalmazás-objektumra mutató nyelet (pointert) mindegyik Fibonacci-kupac elemben, valamint a megfelel ő Fibonacci-kupac elemre mutató pointert mindegyik alkalmazás-objektumban.

A 20.1. alfejezetben definiáljuk a Fibonacci-kupacokat, foglalkozunk ábrázolásukkal, és megadjuk azt a potenciálfüggvényt, amelyet az amortizált elemzésükben használunk. A 20.2. alfejezetben megmutatjuk, hogy hogyan lehet megvalósítani az összefésülhet ő-kupac műveleteket, és azt, hogy hogyan lehet elérni a 19.1. ábrán megadott amortizált időkorlátokat. A másik két műveletet, a KULCSOT-CSÖKKENT és a TÖRÖL műveletet a 20.3. alfejezetben tárgyaljuk. Végül, a 20.4. alfejezet az elemzés egyik kulcsfontosságú részével fejeződik be, és megmagyarázza az adatszerkezet különös elnevezését is.

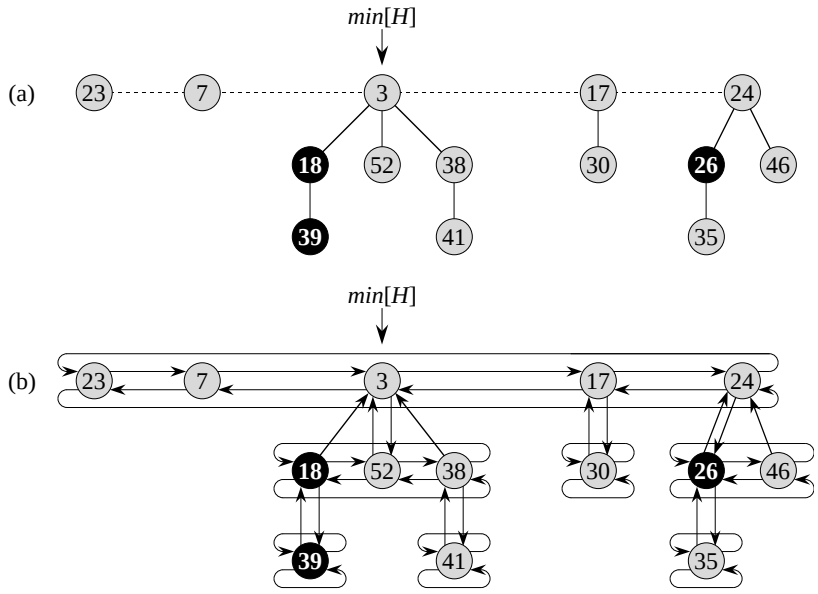
20.1. A Fibonacci-kupacok szerkezete

A binomiális kupacokhoz hasonlóan, a ***Fibonacci-kupac*** min-kupac-rendezett fák gyűjteménye. Azonban a Fibonacci-kupacban lev ő fák nem feltétlenül binomiális fák. Fibonacci-kupacra egy példa a 20.1(a) ábrán látható.

A binomiális kupacokban levő fák rendezettek, ezzel ellentétben a Fibonacci-kupacokban levő fáknak van gyökérelemük, de nem rendezettek. Mint a 20.1(b) ábrán látható, mindegyik x csúcsnak van egy $p[x]$ pointere a szül őjére, és egy $gyerek[x]$ pointere az egyik gyerekére. Az x csúcs gyerekei egy kétirányú ciklikus listával vannak összekapcsolva, ezt a listát az x ***gyerek-listájának*** nevezzük. A gyerek-lista minden y csúcsának van egy $bal[y]$ és egy $jobb[y]$ pointere, ezek az y bal oldali és jobb oldali testvérére mutatnak. Ha az y egyedüli gyerek, akkor $bal[y] = jobb[y] = y$. A gyerekek sorrendje a gyerek-listában tetszőleges.

A Fibonacci-kupacokban a kétszeresen láncolt listák (lásd a 10.2. alfejezetet) alkalmazásának két előnye is van. Az első az, hogy egy kétszeresen láncolt listából $O(1)$ id ő alatt lehet elemet törölni. A második pedig az, hogy két ilyen listát egy kétszeresen láncolt listába konkatenálni (azaz egymáshoz kapcsolni) szintén $O(1)$ id őben lehet. A továbbiakban Fibonacci-kupacok műveleteinek leírásaiban erre a két műveletre már csak informálisan hivatkozunk, és az olvasóra bízzuk a megvalósítás részleteinek kidolgozását.

Mindegyik csúcshoz még további két mez őt rendelünk. Az x csúcs gyerek-listájában levő gyerekek számát a $fokszám[x]$ mez őben tároljuk. A logikai $megjelöl[x]$ mez ő pedig azt jelzi, hogy az x elvesztette-e gyerekét azóta, mióta x egy másik csúcs gyerekévé vált. Az újonnan létrehozott csúcsok nincsenek megjelölve, és egy x csúcs megjelöletlenné válik, amikor egy másik csúcs gyereke lesz. A 20.3. alfejezetig, ahol majd a KULCSOT-CSÖKKENT műveletet megismerjük, mindegyik $megjelöl$ mez őt a HAMIS értékkel töltjük fel.



20.1. ábra. (a) A Fibonacci-kupacnak öt min-kupac-rendezett fája és 14 csúcsa van. A szaggatott vonal a gyökérlistát jelöli. A kupac minimum-csúcsa a 3-as kulcsot tartalmazó csúcs. A fekete szín a megjelölt csúcsokat jelzi. Ennek a speciális Fibonacci-kupacnak a potenciálja $5 + 2 \cdot 3 = 11$. **(b)** Egy teljesebb reprezentáció tartalmazza a *p* (felfelé mutató nyilak), a *gyerek* (lefelé mutató nyilak) és a *bal* és *jobb* (oldalirányba mutató nyilak) pointereket is. Ezeket a részleteket a fejezet további ábráiból már kihagyjuk, mivel ezek az **(a)** ábrából is meghatározhatók.

Egy adott Fibonacci-kupac a minimális kulcsot tartalmazó fájának gyökérelemére mutató $min[H]$ pointerrel címezhető meg, a minimális kulcsot tartalmazó csúcsot a Fibonacci-kupac ***minimumcsúcsának*** nevezzük. Ha egy H Fibonacci-kupac üres, akkor $min[H] = \text{NIL}$.

A Fibonacci-kupac fáinak gyökércsúcsai *bal* és *jobb pointerekkel* vannak összekapcsolva, ezt a ciklikus kétirányú listát a Fibonacci-kupac ***gyökérlistájának*** nevezzük. A $min[H]$ pointer a gyökérlista minimális kulcsú csúcsára mutat. A fák sorrendje a gyökérlistában tetszőleges.

A H Fibonacci-kupac egy másik adatát is fogjuk majd használni: a H csúcsainak számát $n[H]$-val jelöljük.

A potenciálfüggvény

Mint már említettük, a Fibonacci-kupacon végrehajtható műveletek hatékonyságának vizsgálatában a 17.3. alfejezetben megismert potenciál módszert fogjuk használni. Egy adott H Fibonacci-kupacra jelöljük $t(H)$-val a H gyökérlistájában levő fák számát, és $m(H)$-val a H-beli megjelölt csúcsok számát. Ekkor a H Fibonacci-kupac potenciálját a következőképpen definiáljuk:

$$\Phi(H) = t(H) + 2m(H). \tag{20.1}$$

(Ehhez a potenciálfüggvényhez bizonyos intuíciókat fogunk kapni a 20.3. alfejezetben.) Például, a 20.1. ábrán látható Fibonacci-kupac potenciálja $5 + 2 \cdot 3 = 11$. Fibonacci-kupacokból álló halmaz potenciálja a halmazt alkotó Fibonacci-kupacok potenciáljának összege. Fel-

tesszük, hogy egy egységnyi potenciál egyenlő egy konstans nagyságú munka költségével, ahol a konstans elegendő nagy arra, hogy az elvégzendő munka bármelyik egyedi konstansidejű részének költségét fedezze.

Feltesszük, hogy egy Fibonacci-kupac alkalmazásának kezdetén még nincs kupac, így a kezdeti potenciál 0, és minden további időpontban a (20.1) egyenlet alapján a potenciál nemnegatív. Ezért a (17.3) egyenlet szerint a teljes amortizált költség felső korlátja a műveletsorozat teljes aktuális költségének felső korlátja lesz.

Maximális fokszám

A fejezet további részében, amikor amortizált elemzéseket végzünk, feltesszük, hogy adott egy $D(n)$ felső korlát függvény, amelyik az n csúcsból álló Fibonacci-kupac csúcsainak maximális fokszámát adja meg. A 20.2-3. gyakorlatban majd látni fogjuk, hogy ha csak összefésülhető-kupac műveleteket végzünk, akkor $D(n) \leq \lfloor \lg n \rfloor$. A 20.3. alfejezetben megmutatjuk, hogy a KULCSOT-CSÖKKENT és a TÖRÖL műveletek alkalmazása esetén $D(n) = O(\lg n)$.

20.2. Összefésülhető-kupac műveletek

Ebben az alpontban leírjuk és elemezzük a Fibonacci-kupacokra megvalósított összefésülhető-kupac műveleteket. Ha csak ezeket a műveleteket – KUPACOT-LÉTREHOZ, BESZÚR, MIN, MINIMUMOT-KIVÁG és EGYESÍT – használjuk, akkor a Fibonacci-kupacok egyszerűen „rendezetlen” binomiális fák halmazának tekinthetők. A binomiális fához hasonlón, a ***rendezetlen binomiális fa*** definíciója is rekurzióval adható meg. Az U_0 rendezetlen binomiális fa egy csúcsból áll, és az U_k rendezetlen binomiális fa két U_{k-1} rendezetlen binomiális fából áll, ahol az egyik fa gyökércsúcsa a másik fa gyökércsúcsának *tetszőleges* gyereke. A 19.1. lemma, amelyik a binomiális fák tulajdonságait mondta ki, érvényes a rendezetlen binomiális fákra is, csupán a 4. tulajdonságot kell módosítani (lásd a 20.2-2. gyakorlatot):

4′. Az U_K rendezetlen binomiális fára a gyökércsúcs fokszáma k, ami nagyobb, mint bármelyik más csúcsának a fokszáma. A gyökércsúcs gyerekei az $U_0, U_1, \ldots, U_{k-1}$ részfák gyökerei, valamilyen sorrendben.

Így ha egy n csúcsból álló Fibonacci-kupac rendezetlen binomiális fák halmaza, akkor $D(n) = \lg n$.

A Fibonacci-kupacon értelmezett összefésülhető-kupac műveletek alapelve az, hogy a munkát el kell halasztani addig, amíg csak lehetséges. A különböző műveletek megvalósításaiban egyes előnyök kihasználását feladjuk jobbak alkalmazásáért. Ha egy Fibonacci-kupacban a fák száma kicsi, akkor a MINIMUMOT-KIVÁG műveletben gyorsan meg tudjuk határozni, hogy a megmaradt csúcsok közül melyik lett az új minimumcsúcs. Azonban, amint a binomiális kupacokra a 19.2-10. gyakorlatban is láttuk, a fák számának minimális értéken tartásáért fizetnünk kell: egy csúcsnak a binomiális kupacba történő beszúrásáért, vagy két binomiális kupac egyesítéséért legalább $\Omega(\lg n)$ időt használunk el. Amint majd látni fogjuk, ha egy új csúcsot beszúrunk, vagy két kupacot egyesítünk, nem törekszünk a Fibonacci-kupacok fáinak összevonására. Az összevonást csak a MINIMUMOT-KIVÁG eljárásban használjuk, ahol ténylegesen szükség van a minimumcsúcs meghatározására.

Egy új Fibonacci-kupac létrehozása

Egy új Fibonacci-kupac készítésére a FIB-KUPACOT-LÉTREHOZ eljárás allokál és visszaad egy H Fibonacci-kupac objektumot, amelyre $n[H] = 0$, és $min[H] =$ NIL; a H-ban egyetlen egy fa sincs. Mivel $t(H) = 0$ és $m(H) = 0$, az üres Fibonacci-kupac potenciálja $\Phi(H) = 0$. A FIB-KUPACOT-LÉTREHOZ amortizált költsége így megegyezik az $O(1)$ aktuális költséggel.

Egy csúcs beszúrása

A következő eljárás beszúrja a H Fibonacci-kupacba az x csúcsot, feltéve, hogy az x csúcs már allokálva van, és hogy a *kulcs*[x] mező már ki van töltve.

FIB-KUPACBA-BESZÚR(H, x)

```
 1  fokszám ← 0
 2  p[x] ← NIL
 3  gyerek[x] ← NIL
 4  bal[x] ← x
 5  jobb[x] ← x
 6  megjelöl[x] ← HAMIS
 7  összefűzi az x-et tartalmazó gyökérlistát a H gyökérlistájával
 8  if min[H] = NIL vagy kulcs[x] < kulcs[min[H]]
 9     then min[H] ← x
10  n[H] ← n[H] + 1
```

Miután az 1–6. sorokban feltöltöttük az x csúcs szerkezetet leíró mezőit úgy, hogy készítettünk egy saját magában ciklikus, kétirányú listát, a 7. sorban hozzáadjuk x-et a H gyökérlistájához, ennek a műveletnek az aktuális ideje $O(1)$. Ezzel az x egy egycsúcsos, min-kupac-rendezett fa lett, és ezzel a Fibonacci-kupac egyik nem rendezett binomiális fájává vált. Az x-nek nincsenek gyerekei, és nincs megjelölve. A 8–9. sorokban, ha szükséges, a H Fibonacci-kupac minimumcsúcsára mutató pointert állítjuk be. Végül, a 10. sorban növeljük az $n[H]$ értékét, mivel a Fibonacci-kupac egy új csúccsal bővült. A 20.1. ábrán látható Fibonacci-kupacba beszúrtuk a 21-es kulcsú csúcsot, az eredmény a 20.2. ábrán látható.

A BINOMIÁLIS-KUPACBA-BESZÚR eljárással ellentétben, a FIB-KUPACBA-BESZÚR eljárás nem kísérli meg a Fibonacci-kupac fáinak összevonását. Ha egymásután k-szor hajtjuk végre a FIB-KUPACBA-BESZÚR eljárást, akkor k darab egycsúcsos fát adunk hozzá a gyökérlistához.

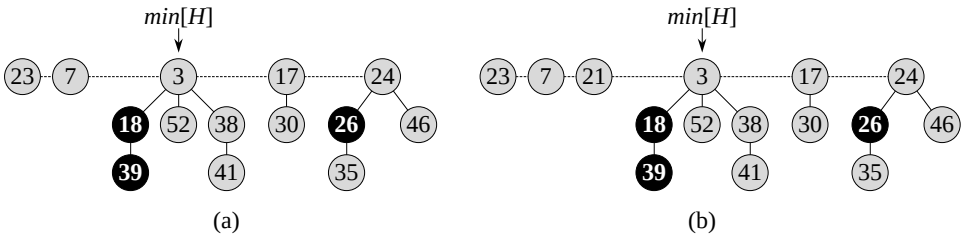
A FIB-KUPACBA-BESZÚR eljárás amortizált költségének meghatározásához legyen H a bemenő adat, és H' az eredményül kapott Fibonacci-kupac. Ekkor $t(H') = t(H) + 1$ és $m(H') = m(H)$, és a potenciál növekedése:

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1.$$

Mivel az aktuális költség $O(1)$, az amortizált költség $O(1) + 1 = O(1)$.

A minimumcsúcs megkeresése

Egy H Fibonacci-kupac minimumcsúcsára a $min[H]$ pointer mutat, így a minimumcsúcsot $O(1)$ aktuális idő alatt találhatjuk meg. Mivel a H potenciálja nem változik, a művelet amortizált költsége megegyezik az $O(1)$ aktuális költséggel.



20.2. ábra. Egy Fibonacci-kupacba egy csúcsot beszúrunk. *(a)* A H Fibonacci-kupac. *(b)* A H Fibonacci-kupac azután, hogy a 21 kulcsú csúcsot beszúrtuk. A csúcs maga is egy min-kupac-rendezett fává vált, a gyökérlistába került, a gyökércsúcs bal oldali testvére lett.

Két Fibonacci-kupac egyesítése

A következő eljárás a H_1 és H_2 Fibonacci-kupacokat egyesíti, a művelet közben a H_1 és H_2 megszűnik. Az eljárás konkatenálja a H_1 és H_2 gyökérlistáját, majd ezután meghatározza az új minimális csúcsot.

FIB-KUPACOKAT-EGYESÍT(H_1, H_2)

```
1 H ← FIB-KUPACOT-LÉTREHOZ()
2 min[H] ← min[H1]
3 összefűzi a H2 gyökérlistáját a H gyökérlistájával
4 if (min[H1] = NIL) vagy (min[H2] ≠ NIL és kulcs[min[H2]] < kulcs[min[H1]])
5    then min[H] ← min[H2]
6 n[H] ← n[H1] + n[H2]
7 szabadítsuk fel a H1 és H2 objektumokat
8 return H
```

Az 1–3. sorokban a H_1 és a H_2 gyökérlistáit fűzzük össze a H új gyökérlistájába. A 2., 4. és 5. sorokban meghatározzuk a H minimumcsúcsát, a 6. sorban betöltjük $n[H]$-ba a H csúcsainak számát. A H_1 és H_2 Fibonacci-kupacokat a 7. sorban felszabadítjuk, és a 8. sorban eredményül visszaadjuk a H Fibonacci-kupacot. A FIB-KUPACBA-BESZÚR algoritmushoz hasonlóan itt sem végezzük el a fák összevonását.

A potenciál változása a következő:

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) + (t(H_2) + 2\,m(H_2))) \\ &= 0,\end{aligned}$$

mivel $t(H) = t(H_1) + t(H_2)$ és $m(H) = m(H_1) + m(H_2)$. Ezért a FIB-KUPACOKAT-EGYESÍT amortizált költsége az $O(1)$ aktuális költséggel egyezik meg.

A minimumcsúcs kivágása

A minimumcsúcs kivágásának módszere az alfejezet eljárásai között a legbonyolultabb. Ez az a módszer, amelyben most már el kell végezni a gyökérlistában levő fák eddig elhalasztott összevonását. A következő algoritmus kivágja a minimumcsúcsot. A program az egyszerűség kedvéért feltételezi, hogy amikor egy csúcsot kiveszünk egy láncolt listából, a listában

maradó pointereket módosítjuk, de a kiemelt csúcsra mutató pointerek nem változnak meg. Az eljárás használja a KIEGYENLÍT eljárást, amit majd röviden ismertetünk.

FIB-KUPACBÓL-MINIMUMOT-KIVÁG(H)

```
 1 z ← min[H]
 2 if z ≠ NIL
 3   then for a z minden x gyerekére
 4             do az x-et tegyük be a H gyökérlistájába
 5                p[x] ← NIL
 6        vegyük ki z-t a H gyökérlistájából
 7        if z = jobb[z]
 8          then min[H] ← NIL
 9          else min[H] ← jobb[z]
10               KIEGYENLÍT(H)
11        n[H] ← n[H] – 1
12 return z
```

Mint a 20.3. ábrán is látható, a FIB-KUPACBÓL-MINIMUMOT-KIVÁG eljárás úgy működik, hogy először átirányítja a minimumcsúcs gyerekeit a gyökérlistába, majd a gyökérlistából a minimumcsúcsot kiveszi. Ezután végrehajtja a kiegyenlítés műveletet, azaz összevonja a megegyező fokszámú gyökércsúcsokat addig, amíg mindegyik fokszámú csúcsból már csak egy marad.

Az eljárás azzal kezdődik, hogy elmentjük z-be a minimumcsúcs mutatóját; ez a pointer lesz majd az eljárás eredménye. Ha z = NIL, akkor a Fibonacci-kupac üressé vált, és készen vagyunk. Ellenkező esetben, a BINOMIALIS-KUPACBÓL-MINIMUMOT-KIVÁG eljáráshoz hasonlóan, miután a z gyerekeit a 3–5. sorokban a H gyökérlistájába láncoltuk, a 6. sorban kivesszük z-t a gyökérlistából, azaz töröljük z-t a H-ból. Ha a 6. sor végrehajtása után $z = jobb[z]$, akkor z az egyetlen csúcs volt a gyökérlistában, és nem voltak gyerekei, ezért ekkor csak azt kell csinálni, hogy a Fibonacci-kupacot üresre állítjuk (8. sor), és ezután eredményül adhatjuk a z-t. Egyébként a $min[H]$ mutatót a gyökérlista egy z-től különböző csúcsára állítjuk (ebben az esetben $jobb[z]$-re), ami nem feltétlenül lesz a minimális csúcs a FIB-KUPACBÓL-MINIMUMOT-KIVÁG befejezésekor. A 20.3(b) ábrán az az állapot látható, amikor a 20.3(a) ábrán levő Fibonacci-kupacra a 9. sort már végrehajtottuk.

A következő lépést, amelyben a Fibonacci-kupac fáinak darabszámát csökkentjük, a H gyökérlista ***kiegyenlítésének*** nevezzük; ezt a műveletet a KIEGYENLÍT(H) hívással hajtjuk végre. A gyökérlista kiegyenlítése a következő lépések ismételt végrehajtását jelenti, mindaddig, amíg a gyökérlistában levő valamennyi gyökércsúcs *fokszáma* különböző nem lesz.

1. Keressünk a gyökérlistában két azonos fokszámú x és y gyökércsúcsot, amelyre $kulcs[x] \leq kulcs[y]$.
2. ***Kapcsoljuk*** az y-t az x-hez: vegyük ki y-t a gyökérlistából, és legyen y az x-nek egy gyereke. Ezt a műveletet a FIB-KUPACOT-SZERKESZT eljárás meghívásával hajtjuk végre. A $fokszám[x]$ mezőt eggyel növeljük, és ha szükséges, az y megjelölését töröljük.

A KIEGYENLÍT eljárás az $A[0\,..\,D(n[H])]$ tömböt használja; ha $A[i] = y$, akkor y az a gyökércsúcs, amelyre $fokszám[y] = i$.

```
KIEGYENLÍT(H)
 1  for i ← 0 to D(n[H])
 2      do A[i] ← NIL
 3  for a H gyökérlistájának minden w csúcsára
 4      do x ← w
 5         d ← fokszám[x]
 6         while A[d] ≠ NIL
 7               do y ← A[d]
                                  ▷ Egy másik, az x fokszámával azonos fokszámú csúcs
 8                  if kulcs[x] > kulcs[y]
 9                    then csere x ↔ y
10                  FIB-KUPACOT-SZERKESZT(H, y, k)
11                  A[d] ← NIL
12                  d ← d + 1
13         A[d] ← x
14  min[H] ← NIL
15  for i ← 0 to D(n[H])
16      do if A[i] ≠ NIL
17           then tegyük be A[i]-t a H gyökérlistájába
18                if min[H] = NIL vagy kulcs[A[i]] < kulcs[min[H]]
19                  then min[H] ← A[i]
```

```
FIB-KUPACOT-SZERKESZT(H, y, x)
1  kivesszük y-t a H gyökérlistájából
2  legyen y az x-nek egy gyereke, növeljük eggyel fokszám[x]-et
3  megjelöl[y] ← HAMIS
```

Részletesen, a KIEGYENLÍT eljárás a következőképpen működik. Az 1–2. sorok beállítják az A kezdeti állapotát, minden elemébe NIL-t töltünk. A **for** ciklus 3–13. soraiban a gyökérlista mindegyik w elemét dolgozzuk fel. Mindegyik w gyökércsúcs feldolgozásának eredménye egy x gyökércsúcsú fa, az x nem feltétlenül egyezik meg a w-vel. A feldolgozott gyökércsúcsok között nincs másik, amelynek ugyanaz lenne a fokszáma, mint x-nek, ezért az $A[fokszám[x]]$ mezőbe betöltjük az x-re mutató pointert. Amikor ez a **for** ciklus befejező-dik, mindegyik fokszámhoz legfeljebb egy gyökércsúcs marad, és az A tömb a megmaradt gyökérpontokra mutató pointereket tartalmazza.

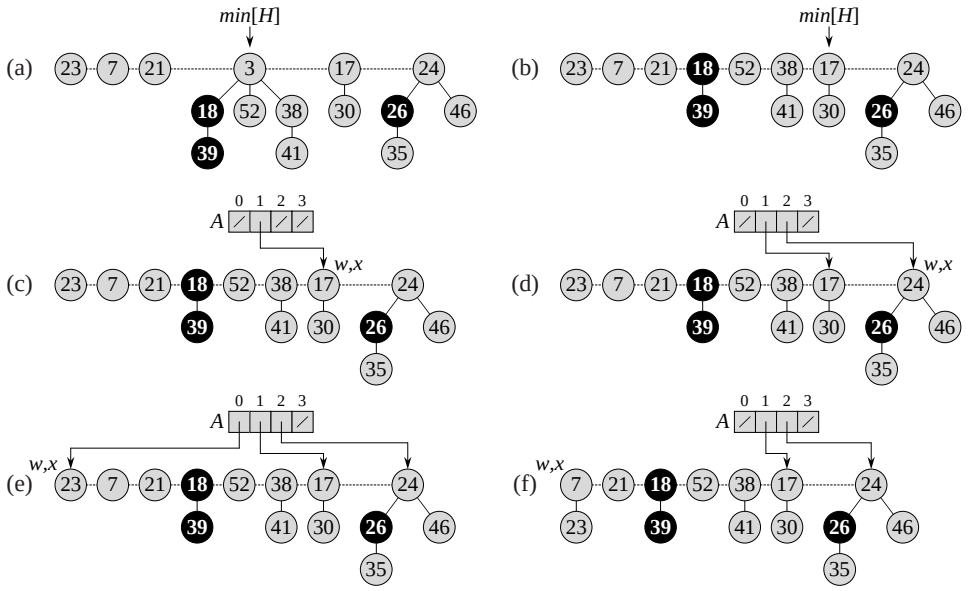
A 6–12. sorokban levő **while** ciklus a w csúcsot tartalmazó fa x csúcsát összekapcsolja egy másik olyan fával, amelyben a gyökércsúcs fokszáma szintén x, és ezt addig ismétli, amíg ilyen fokszámú gyökércsúcs van. A **while** ciklusinvariánsa a következő:

A **while** ciklus mindegyik iterációjának kezdetén $d = fokszám[x]$.

Ezt a ciklusinvariánst a következőképpen használjuk:

Teljesül: Az 5. sor biztosítja azt, hogy a ciklusinvariáns fennáll a ciklusba való első belépéskor.

Megmarad: A **while** ciklus mindegyik iterációjában az $A[d]$ valamelyik y gyökércsúcsra mutat. Mivel $d = fokszám[x] = fokszám[y]$, az x-et és az y-t össze szeretnénk kapcsolni.



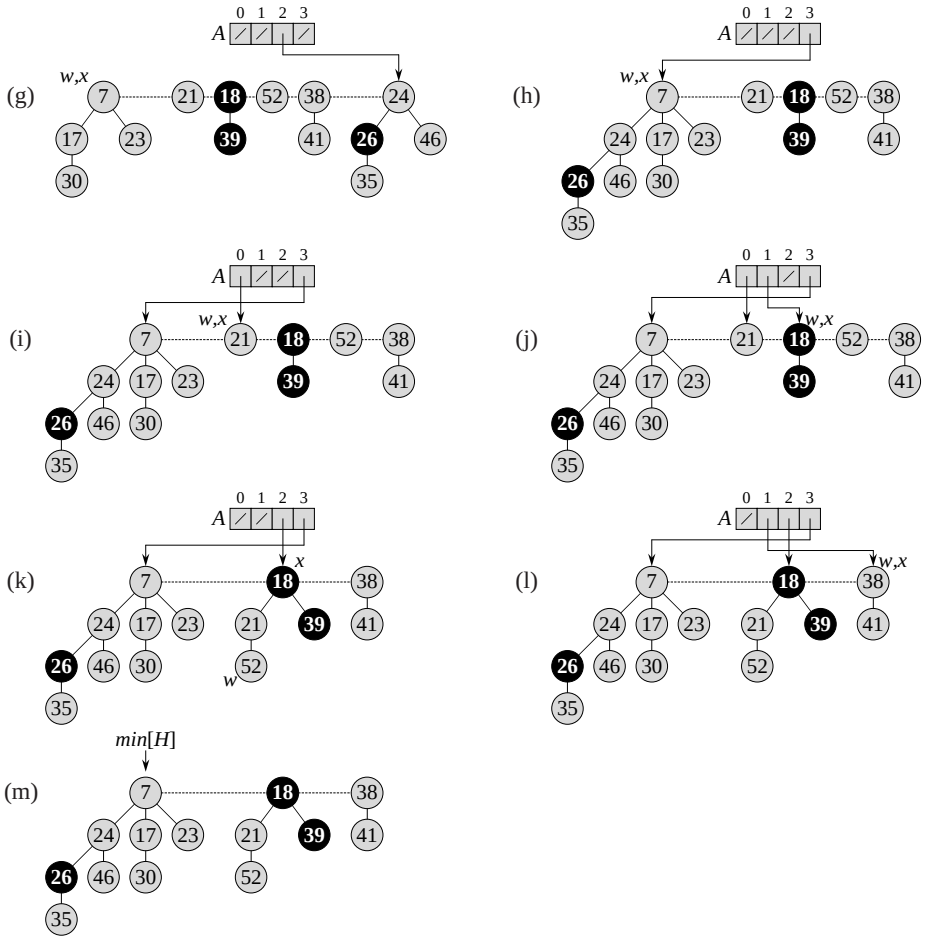
20.3. ábra. A FIB-KUPACBÓL-MINIMUMOT-KIVÁG működése. **(a)** A H Fibonacci-kupac. **(b)** Az az állapot, amikor az gyökérlistából a z csúcsot kivettük, és a minimumcsúcs gyerekeit a gyökérlistába tettük. **(c)–(e)** Az A tömb és a fák, a KIEGYENLÍT eljárás 3–13. sorában levő **for** ciklus első, második és harmadik iterációja után. A művelet a gyökérlista minimumcsúcsánál kezdődik, majd a következő csúcsokat a *jobb* pointer határozza meg. Mindegyik képen a w és az x iteráció utáni értékét láthatjuk. **(f)–(h)** A **for** ciklus következő iterációja, az ábrán látható w és x értékek a 6–12. sorokban levő **while** ciklus lefutása utáni értékek. Az (f) a **while** első lefutása utáni állapotot mutatja. A 23-as kulcsú csúcs a 7-es kulcshoz lett kapcsolva, most erre a csúcsra mutat az x. A (g) képen, a 17-es kulcsú csúcs szintén a 7-es kulcsú csúcshoz lett kapcsolva, az x még mindig erre a csúcsra mutat. A (h) képen látható, hogy a 24-es kulcsú csúcs szintén a 7-eshez került. Mivel az $A[3]$ pointer nem mutat sehová, a **for** ciklus végén $A[3]$-ba az eredményül kapott fára mutató pointer kerül. **(i)–(l)** A **while** ciklus első négy lépése utáni állapotok. **(m)** A H Fibonacci-kupac állapota, miután az A tömbből a gyökérlista rendezése befejeződött, és a $min[H]$ pointert is meghatároztuk.

Az x és y közül az összekapcsolás eredményeként a kisebbik kulcsú a másiknak szülőjévé válik, és ha szükséges, az x-re és az y-ra mutató pointereket felcseréljük. Ezután a 10. sorban az y-t az x-hez hozzákapcsoljuk a FIB-KUPACOT-SZERKESZT(H, y, x) eljárás meghívásával. Ez az eljárás eggyel növeli a $fokszám[x]$-et, de a $fokszám[y]$ d értékét nem változtatja meg. Mivel az y csúcs többé már nem gyökércsúcs, a reá mutató pointert az A-ból a 11. sorban kivesszük. Mivel a FIB-KUPACOT-SZERKESZT(H, y, x) eljárás a $fokszám[x]$ értékét eggyel növeli, a 12. sorban visszaállítjuk a $d = fokszám[x]$ invariánst.

Befejeződik: A **while** ciklust az $A[d] =$ NIL-ig ismételjük, ami azt jelenti, hogy nincs több ugyanekkora x fokszámú gyökércsúcs.

Miután a **while** ciklus befejeződött, a 13. sorban az $A[d]$-t az x-re állítjuk, és ezután a **for** ciklus következő iterációja következik.

A 20.3(c)–(e) ábrák a 3–13. sorban levő **for** ciklus első, második és harmadik iterációja utáni A tömböt és az eredményül kapott fákat mutatják. A **for** ciklus következő iterációjában három szerkesztés lesz, ezeknek az eredményei a 20.3(f)–(h) ábrákon láthatók. A 20.3(i)–(l) ábrák a **for** ciklus következő négy iterációjának eredményét mutatják.



Ezután már csak a „megtisztítás” marad hátra. Ha a **for** ciklus befejeződött, akkor a 14. sorban a gyökérlistát üresre állítjuk, és a 15–19. sorokban újra felépítjük az A tömbből. Az eredményül kapott Fibonacci-kupacot a 20.3(m) ábrán láthatjuk. A gyökérlista kiegyenlítése után a FIB-KUPACBÓL-MINIMUMOT-KIVÁG eljárás a 11. sorban eggyel csökkenti $n[H]$-t, majd a 12. sorban eredményül adja a törölt z csúcsra mutató pointert.

Megfigyelhetjük, hogy ha a FIB-KUPACBÓL-MINIMUMOT-KIVÁG algoritmus végrehajtása előtt a Fibonacci-kupac minden fája nem rendezett binomiális fa, akkor mindegyik fa nem rendezett binomiális fa is marad. Két olyan hely van, ahol fák megváltozhatnak. Az első: a FIB-KUPACBÓL-MINIMUMOT-KIVÁG algoritmus 3–5. sorában a z gyökércsúcs mindegyik x gyereke gyökércsúcs lesz. A 20.2-2. gyakorlat alapján, mindegyik új fa egy nem rendezett binomiális fa. A második: a FIB-KUPACOT-SZERKESZT eljárás két fát összekapcsol, ha a fák fokszáma azonos. Mivel a szerkesztés előtt minden fa nem rendezett binomiális fa, az a két fa, amelyekben a gyökércsúcsoknak k gyerekük van, biztosan U_k szerkezetű. Az eredményül kapott fa így biztosan U_{k+1}.

Most már eljutottunk oda, hogy megmutassuk: egy n-csúcsos Fibonacci-kupacból a minimumcsúcs kivágásának amortizált költsége $O(D(n))$. Jelölje H a Fibonacci-kupacot a FIB-KUPACBÓL-MINIMUMOT-KIVÁG algoritmus végrehajtása előtt.

A minimumcsúcs kivágásának aktuális költségét a következőképpen határozzuk meg. Egy $O(D(n))$ összetevőt kapunk abból, hogy a FIB-KUPACBÓL-MINIMUMOT-KIVÁG algoritmusban a minimumcsúcsnak legfeljebb $D(n)$ gyerekével kell foglalkozni, és ekkora költséget kapunk a KIEGYENLÍT algoritmus 1–2. és 14–19. soraiból is. Ezután már csak a **for** ciklus 3–13. sorait kell elemezni. A KIEGYENLÍT eljárás meghívásakor a gyökérlista mérete legfeljebb $D(n) + t(H) - 1$, mivel a gyökérlista tartalmazza az eredeti $t(H)$ gyökércsúcsokat, nincs benne az egy darab kivágott gyökércsúcs, de benne vannak a kivágott csúcs gyerekei, ezeknek a száma legfeljebb $D(n)$. A **while** ciklus 6–12. sorának minden egyes végrehajtásakor egy gyökércsúcsot egy másikhoz kapcsolunk, így a **for** ciklus által elvégzett munka legfeljebb $D(n) + t(H)$-val arányos. Így a minimális csúcs kivágásának teljes aktuális munkája $O(D(n) + t(H))$.

A potenciál a minimumcsúcs kivágása előtt $t(H) + 2m(H)$, a kivágás után $(D(n) + 1) + 2m(H)$, mivel a művelet után legfeljebb $D(n) + 1$ gyökér lesz, és a művelet végrehajtása alatt egy csúcs sem válik megjelöltté. Ezért az amortizált költség legfeljebb

$$\begin{aligned} O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H) &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)), \end{aligned}$$

mivel a potenciál egységeket megnövelhetjük úgy, hogy dominálják az $O(t(H))$-ban elrejtett konstanst. Egyszerűen, a potenciál csökkentéséért végzett mindegyik szerkesztés költsége megfelel annak a költségnek, amibe a gyökércsúcsok számának eggyel történő csökkentése kerül.

Gyakorlatok

20.2-1. Adjuk meg azt a Fibonacci-kupacot, amelyiket akkor kapnak, ha a 20.3(m) ábrán látható Fibonacci-kupacra végrehajtják a FIB-KUPACBÓL-MINIMUMOT-KIVÁG műveletet.

20.2-2. Bizonyítsuk be a 19.1. lemma állításait nem rendezett binomiális fákra, a 4. állítást a $4'$-vel kell helyettesíteni.

20.2-3. Mutassuk meg, hogy ha csak az összefésülhető-kupac műveleteket használjuk, akkor egy n-csúcsos Fibonacci-kupac $D(n)$ maximális fokszáma legfeljebb $\lfloor \lg n \rfloor$.

20.2-4. McGee professzor definiált egy Fibonacci-kupacokon alapuló új adatszerkezetet. A McGee-kupacnak ugyanolyan szerkezete van, mint a Fibonacci-kupacnak, és a McGee-kupacokra ugyanazokat az összefésülhető-kupac műveleteket lehet végrehajtani, mint amit a Fibonacci-kupacokra. A műveletek megvalósítása is ugyanaz, mint a Fibonacci-kupacokra, kivéve azt, hogy a beszúrás és az egyesítés művelet mindegyike az utolsó lépésében kiegyenlítést hajt végre. Mi a McGee-kupacok műveleteinek legrosszabb futási ideje? Mi a szokatlan a professzor adatszerkezeteiben?

20.2-5. Indokoljuk meg, hogy ha a kulcsokra csak két kulcs összehasonlításának a műveletét tesszük lehetővé (mint ahogy ez ebben a fejezetben az összes megvalósításban történt), akkor nem mindegyik összefésülhető-kupac művelet fog lefutni $O(1)$ amortizált idő alatt.

20.3. Egy kulcs csökkentése és egy csúcs törlése

Ebben az alfejezetben megmutatjuk, hogy hogyan lehet a Fibonacci-kupac egy csúcsának kulcsát $O(1)$ amortizált időben csökkenteni, és egy n-csúcsos Fibonacci-kupacból egy csúcsot $O(D(n))$ amortizált időben kitörölni. Ezek a műveletek nem őrzik meg azt a tulajdonságot, hogy a Fibonacci-kupacban minden fa nem rendezett binomiális fa. Azonban eléggé zártak ahhoz, hogy a $D(n)$ maximális fokszám korlátját $O(\lg n)$-nel lehessen megadni. Ennek a korlátnak a bizonyításában, amit majd a 20.4. alfejezetben fogunk elvégezni, benne lesz az is, hogy a FIB-KUPACBÓL-MINIMUMOT-KIVÁG és a FIB-KUPACBÓL-TÖRÖL eljárások $O(\lg n)$ amortizált időben futnak le.

Egy kulcs csökkentése

A FIB-KUPACBAN-KULCSOT-CSÖKKENT művelet következő programjában, mint a korábbiakban is, feltételezzük, hogy ha egy csúcsot egy láncolt listából kiveszünk, akkor a csúcs szerkezeti mezőinek egyike sem változik meg.

```
FIB-KUPACBAN-KULCSOT-CSÖKKENT(H, x, k)

1  if k > kulcs[x]
2     then error „az új kulcs nagyobb, mint az aktuális kulcs”
3  kulcs[x] ← k
4  y ← p[x]
5  if y ≠ NIL és kulcs[x] < kulcs[y]
6     then KIVÁG(H, x, y)
7          KASZKÁD-VÁGÁS(H, y)
8  if kulcs[x] < kulcs[min[H]]
9     then min[H] ← x
```

```
KIVÁG(H, x, y)

1  vegyük ki x-et az y gyereklistájából, és csökkentsük eggyel fokszám[y]-t
2  tegyük bele x-et a H gyökérlistájába
3  p[x] ←NIL
4  megjelöl[x] ← HAMIS
```

```
KASZKÁD-VÁGÁS(H, y)

1  z ← p[y]
2  if z ≠ NIL
3     then if megjelöl[y] = HAMIS
4             then megjelöl[y] ← IGAZ
5             else KIVÁG(H, y, z)
6                  KASZKÁD-VÁGÁS(H, z)
```

A FIB-KUPACBAN-KULCSOT-CSÖKKENT eljárás a következőképpen működik. A 1–3. sorok programja azt biztosítja, hogy az új kulcs nem nagyobb, mint az x aktuális kulcsa, és ebben az esetben a k-t az x csúcshoz hozzárendeljük. Ha $kulcs[x] \geq kulcs[y]$, ahol y az x csúcs

szülője, vagy ha x egy gyökércsúcs, akkor a Fibonacci-kupac szerkezete nem változik meg, mivel a min-kupac rendezettsége megmarad. Ezt a feltételt a 4–5. sorokban vizsgáljuk meg.

Ha a min-kupac rendezettsége megsérül, több módosítást kell végeznünk. A 6. sorban levő ***kivágással*** kezdjük. A KIVÁG eljárás „elvágja” az x és a szülője, y közötti kapcsolatot, és az x egy gyökércsúcs lesz.

A *megjelöl* mezőket arra a célra használjuk, hogy a megadott időkorlátokat megkapjuk. Ezek a mezők mindegyik csúcs történetének egy kis darabját rögzítik. Tegyük fel, hogy x egy olyan csúcs, amelyikkel a következő események történtek:

1. egyszer régebben az x egy gyökércsúcs volt,
2. azután az x-et egy másik csúcshoz kapcsolták,
3. majd az x két gyermekét a vágással elvették.

Mihelyt az x második gyerekét is elvették, az x-et elvágjuk a szülőjétől, és az x egy új gyökércsúcs lesz. A *megjelöl*[x] mező IGAZ lesz, ha az 1. és a 2. lépést végrehajtottuk, és utána az x egyik gyerekét kivágtuk. A KIVÁG eljárás törli a *megjelöl*[x]-et a 4. sorban, mivel az 1. lépést hajtja végre. (Most láthatjuk, hogy a FIB-KUPACOT-SZERKESZT eljárás a 3. sorban miért törli a *megjelöl*[y]-t: az y csúcs egy másik csúcshoz kapcsolódik, és így a 2. lépést hajtjuk végre. A következő lépésben az y gyerekét vágjuk ki, és ekkor a *megjelöl*[y] értéke IGAZ lesz.)

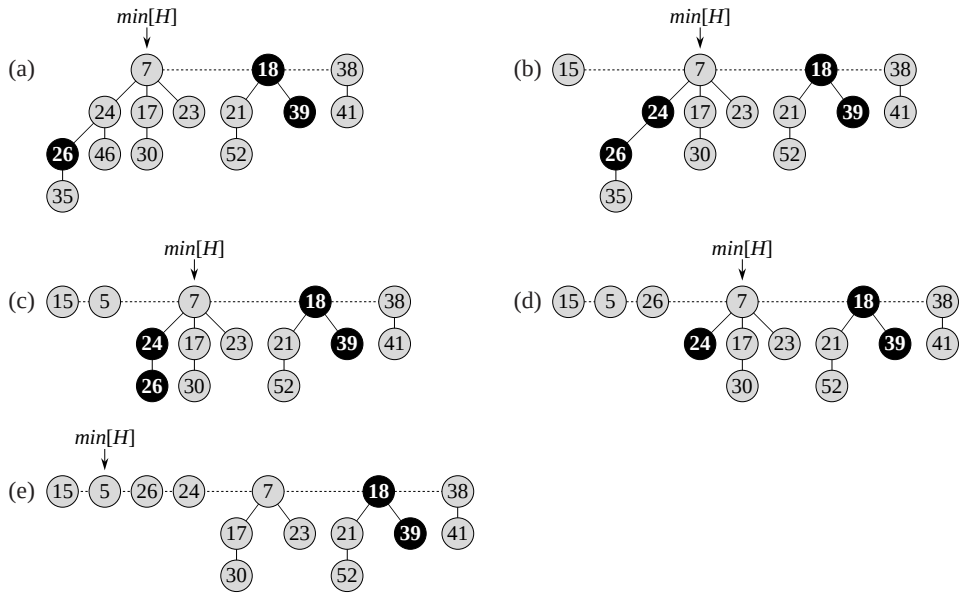
Mi most nem ezt csináljuk, mivel x a második gyerek is lehet, amit az y szülőjétől elvágtak azóta, amióta az y csúcs egy másik csúcshoz lett kapcsolva. Ezért a FIB-KUPACBAN-KULCSOT-CSÖKKENT a 7. sorban az y csúcsra meghívja a ***kaszkád vágás*** műveletet. Ha y a gyökércsúcs, akkor a KASZKÁD-VÁGÁS eljárás a 2. sorban végrehajtott vizsgálat után befejeződik. Ha y nincs megjelölve, akkor az eljárás 4. sorában az y megjelöltté válik. Ellenkező esetben, ha y megjelölt volt, akkor y nemrégen vesztette el második gyerekét; az y-t kivágjuk az 5. sorban, és a 6. sorban rekurzívan meghívjuk z-re, az y csúcs szülőjére a KASZKÁD-VÁGÁS műveletet. A KASZKÁD-VÁGÁS eljárás a rekurzív hívásokkal vagy eljut a fa gyökércsúcsába, vagy egy nem megjelölt csúcsot talál.

Ha már az összes kaszkád vágást végrehajtotta, akkor a 8–9. sorban FIB-KUPACBAN-KULCSOT-CSÖKKENT befejeződik, és ha szükséges, a *min*[H]-t is beállítja. Az egyetlen csúcs, amelynek a kulcsa megváltozott, az x csúcs volt, és az x kulcsa csökkent. Így az új minimális csúcs vagy az eredeti minimális csúcs, vagy az x csúcs.

A 20.4. ábrán két FIB-KUPACBAN-KULCSOT-CSÖKKENT eljárás hívásának végrehajtása látható, a kezdeti Fibonacci-kupacot a 20.4(a) ábra tartalmazza. Az első híváshoz (20.4(b) ábra) nem kell kaszkád vágás. A második procedúra hívásban (20.4(c)–(e) ábra) két kaszkád vágás szükséges.

Most megmutatjuk, hogy a FIB-KUPACBAN-KULCSOT-CSÖKKENT amortizált költsége csak $O(1)$. Először az aktuális költséget határozzuk meg. A FIB-KUPACBAN-KULCSOT-CSÖKKENT eljárás $O(1)$ ideig tart, és ehhez hozzá kell még adni a kaszkád vágások idejét. Tegyük fel, hogy a FIB-KUPACBAN-KULCSOT-CSÖKKENT egy adott meghívásakor a KASZKÁD-VÁGÁS eljárás c-szer hajtódik végre. Mindegyik KASZKÁD-VÁGÁS $O(1)$ ideig tart, ha nem számítjuk a rekurzív hívások idejét. Így a FIB-KUPACBAN-KULCSOT-CSÖKKENT aktuális költsége, beleszámítva a rekurzív hívásokat is, $O(c)$.

Most kiszámítjuk a potenciál változását. Jelölje H a Fibonacci-kupacot a FIB-KUPACBAN-KULCSOT-CSÖKKENT hívása előtt. A KASZKÁD-VÁGÁS mindegyik rekurzív hívása, kivéve az utolsót, kivágja a megjelölt csúcsot és törli a megjelölés bitet. A KASZKÁD-VÁGÁS eljárások vég-



20.4. ábra. A FIB-KUPACBAN-KULCSOT-CSÖKKENT két hívása. **(a)** A kezdeti Fibonacci-kupac. **(b)** 15-re csökkentjük annak a csúcsnak a kulcsát, amelyiknek a kulcsa 46 volt. A csúcs gyökércsúccsá válik, és a szülője (kulcsa 24), ami korábban nem megjelölt csúcs volt, most megjelöltté válik. **(c)–(e)** 5-re csökkentjük annak a csúcsnak a kulcsát, amelyiknek a kulcsa 35 volt. A (c) ábrán az 5-ös kulcsú csúcs gyökércsúccsá válik. Szülője, a 26-os kulcsú csúcs megjelölt csúcs lesz, ezért kaszkád vágásra van szükség. A 26-os kulcsú csúcsot elvágjuk a szülőjétől, és ez a csúcs egy nem megjelölt gyökércsúcs lesz a (d) ábrán. Még egy kaszkád vágás kell, mivel a 24-es kulcsú csúcs is megjelölt csúcs. Ezt a csúcsot is elvágjuk a szülőjétől, ez a csúcs is egy nem megjelölt gyökércsúcs lesz az (e) ábrán. A kaszkád vágások itt befejeződnek, mivel a 7-es kulcsú csúcs gyökércsúcs. (Ha ez a csúcs nem lenne gyökércsúcs, a sorozat akkor is befejeződne, mivel ez a csúcs nincs megjelölve.) A FIB-KUPACBAN-KULCSOT-CSÖKKENT művelet végeredménye az (e) ábrán látható, a $min[H]$ az új minimumcsúcsra mutat.

rehajtása után a Fibonacci-kupac $t(H)+c$ fából áll (eredetileg volt $t(H)$ fa, $c-1$ új fa készült a kaszkád kivágásokkal, és van egy x gyökércsúcsú fa), és kaptunk legfeljebb $m(H)-c+2$ megjelölt csúcsot (a kaszkád vágásokkal $c-1$ csúcs vált nem megjelöltté, és a KASZKÁD-VÁGÁS eljárás utolsó hívása egy csúcsot esetleg megjelölt). A potenciál változása tehát

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c.$$

Így a FIB-KUPACBAN-KULCSOT-CSÖKKENT amortizált költsége legfeljebb

$$O(c)+4-c=O(1),$$

mivel a potenciál egységeket megnövelhetjük úgy, hogy dominálják az $O(c)$-ben elrejtett konstanst.

Most már látszik, hogy a potenciálfüggvényt miért definiáltuk úgy, hogy a megjelölt csúcsok egy kettes szorzóval szerepeljenek benne. Ha egy megjelölt y csúcsot kivágunk a kaszkád vágás művelettel, akkor a megjelölés bit törlődik, és ezért a potenciál értéke kettővel csökken. Egy potenciál egység csökkenés lesz a vágásért és a megjelölés bit törléséért, a második egység pedig ellensúlyozza azt a plusz egységet, ami abból adódik, hogy y egy új gyökércsúccsá válik.

Egy csúcs törlése

Egy n-csúcsos Fibonacci-kupacból egy csúcsot kitörölni $O(D(n))$ amortizált idő alatt nem nehéz feladat, a következő programmal ezt mutatjuk meg. Feltételezzük, hogy a Fibonacci-kupacban jelenleg nincs $-\infty$ kulcsérték.

FIB-KUPACBÓL-TÖRÖL(H, x)

1 FIB-KUPACBAN-KULCSOT-CSÖKKENT(H, x, $-\infty$)
2 FIB-KUPACBÓL-MINIMUMOT-KIVÁG(H)

A FIB-KUPACBÓL-TÖRÖL eljárás a BINOMIÁLIS-KUPACOT-TÖRÖL eljáráshoz hasonló. Az x csúcs a Fibonacci-kupac minimumcsúcsa lesz, a kulcsa felveszi a $-\infty$ értéket, ez lesz az egyetlen ilyen kulcsértékű csúcs. Ezt a csúcsot ezután a FIB-KUPACBÓL-MINIMUMOT-KIVÁG eljárással kivesszük a Fibonacci-kupacból. A FIB-KUPACBÓL-TÖRÖL amortizált ideje a FIB-KUPACBAN-KULCSOT-CSÖKKENT $O(1)$ és a FIB-KUPACBÓL-MINIMUMOT-KIVÁG $O(D(n))$ amortizált időinek az összege. A 20.4. alfejezetben majd látni fogjuk, hogy $D(n) = O(\lg n)$, ezért a FIB-KUPACBÓL-TÖRÖL amortizált ideje $O(\lg n)$.

Gyakorlatok

20.3-1. Tegyük fel, hogy egy Fibonacci-kupacban az x gyökércsúcs meg van jelölve. Mutassuk meg, hogy hogyan válhatott az x megjelölt csúccsá. Bizonyítsuk be, hogy az elemzés szempontjából közömbös, hogy az x meg van-e jelölve, még akkor is, ha nem is gyökércsúcs az a csúcs, amelyiket először hozzákapcsoltunk egy másik csúcshoz, és amelyik ezután elvesztett egy gyereket.

20.3-2. Az összekapcsoló elemzés módszerét alkalmazva igazoljuk a FIB-KUPACBAN-KULCSOT-CSÖKKENT $O(1)$ amortizált idejét, mint az egy műveletre eső átlagos költséget.

20.4. A maximális fokszám korlátja

A FIB-KUPACBÓL-MINIMUMOT-KIVÁG és a FIB-KUPACBÓL-TÖRÖL $O(\lg n)$ amortizált idejének bebizonyításához be kell látnunk, hogy egy n csúcsból álló Fibonacci-kupac tetszőleges csúcsának $D(n)$ fokszámára a felső korlát $O(\lg n)$. A 20.2-3. gyakorlatban láttuk, hogy ha egy Fibonacci-kupac minden fája nem rendezett binomiális fa, akkor $D(n) = \lfloor \lg n \rfloor$. A FIB-KUPACBÓL-TÖRÖL eljárásban előforduló vágás azonban átalakíthatja a Fibonacci-kupac fáját úgy, hogy az már nem teljesíti a nem rendezett binomiális fa feltételt. Ebben az alfejezetben látni fogjuk, hogy mivel mi elvágunk egy csúcsot a szülőjétől már akkor, amikor a csúcs elveszít két gyereket, ezért a $D(n)$ nagyságrendje $O(\lg n)$. Pontosabban, látni fogjuk, hogy $D(n) \leq \lfloor \log_\phi n \rfloor$, ahol $\phi = (1 + \sqrt{5})/2$.

Az elemzés kulcsa a következő. A Fibonacci-kupac minden x csúcsához adjunk meg egy méret(x) mezőt, ez az x gyökércsúcsú részfában levő csúcsok számát tartalmazza, a csúcsok számába az x-et is beleszámoljuk. (Megjegyezzük, hogy x-nek nem kell feltétlenül a gyökérlistában lennie, x egy tetszőleges csúcs.) Majd látni fogjuk, hogy a méret(x) a *fokszám*[x] exponenciális függvénye lesz. Ne felejtsük el, hogy a *fokszám*[x] az x csúcs fokszámának mindig az aktuális, pontos értékét tartalmazza.

20.1. lemma. *Legyen x a Fibonacci-kupac egy tetszőleges csúcsa, és tegyük fel, hogy fokszám$[x] = k$. Jelölje $y_1, y_2, \ldots, y_k$ az x gyerekeit abban a sorrendben, ahogyan az x-hez kapcsolódtak, legkorábban az y_1 és utoljára az y_k. Ekkor fokszám$[y_1] \geq 0$, és fokszám$[y_i] \geq i - 2$ $i = 2, 3, \ldots, k$-ra.*

Bizonyítás. A *fokszám*$[y_1] \geq 0$ állítás nyilvánvaló.

$i \geq 2$ esetén vegyük észre, hogy amikor az y_i-t az x-hez kapcsoltuk, akkor már $y_1, y_2, \ldots, y_{i-1}$ az x gyerekei voltak, és így *fokszám*$[x] = i - 1$ fennállt. Az y_i csúcsot csak akkor kapcsolhatjuk az x-hez, ha *fokszám*$[x]$ = *fokszám*$[y_i]$, ezért akkor a *fokszám*$[y_i] = i - 1$-nek is teljesülnie kellett. Azóta az y_i legfeljebb egy gyereket vesztett el, mivel ha két gyereket vesztett volna el, akkor az x-ből kivágták volna. Így tehát *fokszám*$[y_i] \geq i - 2$. ■

Végül eljutottunk az elemzésnek arra a pontjára, ahol megadhatjuk a „Fibonacci-kupac" elnevezés indokait. Hivatkozunk a 3.2. alfejezetre, ahol a k-adik ($k = 0, 1, 2, \ldots$) Fibonacci-számot a következőképpen definiáltuk:

$$F_k = \begin{cases} 0, & \text{ha } k = 0, \\ 1, & \text{ha } k = 1, \\ F_{k-1} + F_{k-2}, & \text{ha } k \geq 2. \end{cases}$$

A következő lemma egy másik módszert ad az F_k kifejezésére.

20.2. lemma. *Minden $k \geq 0$ egész számra*

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Bizonyítás. Az állítást k szerinti teljes indukcióval bizonyítjuk. $k = 0$-ra

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2. \end{aligned}$$

Az indukciós feltétel szerint $F_{k+1} = 1 + \sum_{i=o}^{k-1} F_i$, ekkor

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i. \end{aligned}$$ ■

A következő lemma és a lemma következménye teljessé teszi az elemzést. Ezek a (3.2-7. gyakorlatban bizonyított)

$$F_{k+2} \geq \phi^k$$

egyenlőtlenséget használják, ahol ϕ a (3.22) egyenletben definiált aranymetszés aránya, azaz $\phi = (1 + \sqrt{5})/2 = 1,61803\ldots$.

20.3. lemma. *Legyen x egy Fibonacci-kupac tetszőleges csúcsa, és legyen $k = fokszám[x]$. Ekkor méret$(x) \geq F_{k+2} \geq \phi^k$, ahol $\phi = (1 + \sqrt{5})/2$.*

Bizonyítás. Jelölje s_k az olyan z csúcsok lehetséges méret(z) értékeinek minimumát, amelyekre $fokszám[z] = k$. Nyilvánvaló, hogy $s_0 = 1$, $s_1 = 2$ és $s_2 = 3$. Az s_k értéke legfeljebb méret(x) lehet, és nyilvánvaló, hogy s_k értéke k szerint monoton nő. Mint a 20.1. lemmában, most is jelölje $y_1, y_2, \ldots, y_k$ az x gyerekeit abban a sorrendben, amelyben az x-hez kapcsolódtak. Most meghatározzuk a méret(x) alsó korlátját úgy, hogy összeszámoljuk a csúcsokat: az első az x csúcs önmaga, a második az y_1, azaz az első gyerek (méret$(y_1) \geq 1$), így

$$\begin{aligned} \text{méret}(x) &\geq s_k \\ &= \sum_{i=2}^{k} s_{fokszám[y_i]} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2}, \end{aligned}$$

ahol az utolsó sor a 20.1. lemma (így $fokszám[y_i] \geq i - 2$) és az s_k monotonitása (így $s_{fokszám[y_i]} \geq s_{i-2}$) következménye.

Most k szerinti teljes indukcióval megmutatjuk, hogy $s_k \geq F_{k+2}$, ahol k nemnegatív egész szám. $k = 0$-ra és $k = 1$-re az állítás triviális. Az indukciós feltétel legyen az, hogy $k \geq 2$ esetén $i = 0, 1, \ldots, k - 1$-re $s_i \geq F_{i+2}$. Ekkor

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} \qquad \text{(a 20.2. lemma alapján).} \end{aligned}$$

Így tehát beláttuk, hogy méret$(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

20.4. következmény. *Egy n csúcsból álló Fibonacci-kupac tetszőleges csúcsának $D(n)$ maximális fokszáma $O(\lg n)$.*

Bizonyítás. Legyen x egy n csúcsból álló Fibonacci-kupac tetszőleges csúcsa, és legyen $k = fokszám[x]$. A 20.3. lemma szerint $n \geq$ méret$(x) \geq \phi^k$. ϕ-alapú logaritmusukat véve azt kapjuk, hogy $k \leq \log_\phi n$. (Pontosabban, mivel k egy egész szám, $k \leq \lfloor \log_\phi n \rfloor$.) Így bármelyik csúcs $D(n)$ maximális fokszáma $O(\lg n)$. ■

Gyakorlatok

20.4-1. Pinocchio professzor azt állítja, hogy egy n csúcsból álló Fibonacci-kupac magassága $O(\lg n)$. Mutassuk meg, hogy a professzor állítása hibás, bármely pozitív egész n-re adjunk egy olyan Fibonacci-kupac műveletsorozatot, amelyik egy olyan Fibonacci-kupacot készít, amely csak egy fából áll, és ez a fa az n csúcsból álló lineáris lánc.

20.4-2. Tegyük fel, hogy általánosítottuk a kaszkád vágás műveletet úgy, hogy elvágja az x-et a szülőjétől, mihelyt az elveszti a k-adik gyerekét, valamilyen k egész konstansra. (A 21.3. alfejezetben szereplő műveletre $k = 2$.) Milyen k értékre lesz $D(n) = O(\lg n)$?

Feladatok

20-1. A törlés egy másik megvalósítása

Pisano professzor javasolta a FIB-KUPACBÓL-TÖRÖL eljárás következő változatát, azt állítva, hogy ez gyorsabban fut le, ha a törlendő csúcs nem az a csúcs, amelyikre a $min[H]$ mutat.

```
PISANO-TÖRLÉS(H, x)

1  if x = min[H]
2    then FIB-KUPACBÓL-MINIMUMOT-KIVÁG(H)
3    else y ← p[x]
4         if y ≠ NIL
5           then VÁGÁS(H, x, y)
6                KASZKÁD-VÁGÁS(H, y)
7         tegyük be x gyerekeinek listáját a H gyökérlistájába
8         vegyük ki x-et a H gyökérlistájából
```

a. A professzornak az az állítása, hogy ez az eljárás gyorsabban fut le, azon a feltevésen alapul, hogy a 7. sort $O(1)$ aktuális időben végre lehet hajtani. Miért hibás ez a feltevés?

b. Adjunk egy jó felső becslést a PISANO-TÖRLÉS aktuális idejére, arra az esetre, ha x nem $min[H]$. A felső korlátot a $fokszám[x]$ és a KASZKÁD-VÁGÁS eljáráshívások c száma függvényében adjuk meg.

c. Tegyük fel, hogy meghívjuk a PISANO-TÖRLÉS(H, x) eljárást, és legyen a H' Fibonacci-kupac az eljárás eredménye. Feltéve, hogy x nem gyökér, adjuk meg H' potenciálját a $fokszám[x]$, c, $t(H)$ és $m(H)$ függvényeként.

d. Mutassuk meg, hogy a PISANO-TÖRLÉS amortizált ideje aszimptotikusan nem jobb, mint a FIB-KUPACBÓL-TÖRÖL ideje, még akkor sem, ha $x \neq min[H]$.

20-2. További Fibonacci-kupac műveletek

A Fibonacci-kupacokra be szeretnénk vezetni két új műveletet úgy, hogy amortizált futási idejük megegyezzen a korábbi műveletek amortizált futási idejével.

a. A FIB-KUPACBAN-KULCSOT-MÓDOSÍT(H, x, k) művelet az x csúcs kulcsát k-ra változtatja meg. Adjunk egy hatékony megvalósítást a FIB-KUPACBAN-KULCSOT-MÓDOSÍT műveletre, és elemezzük az algoritmus futási idejét azokra az esetekre, amikor k és $kulcs[x]$ között a nagyobb, egyenlő és a kisebb reláció áll fenn.

b. Adjunk egy hatékony algoritmust a FIB-KUPACOT-RITKÍT(H, r) műveletre, amelyik kitöröl $min(r, n[H])$ csúcsot a H-ból. A kitörlendő csúcsokat tetszőlegesen kiválaszthatjuk. Elemezzük az algoritmus amortizált futási idejét. (*Útmutatás.* Az adatszerkezetet és a potenciálfüggvényt is lehet módosítani.)

Megjegyzések a fejezethez

A Fibonacci-kupacokat Fredman és Tarjan vezette be [98]. Publikációjukban leírják a Fibonacci-kupacok alkalmazását a következő problémákra: adott csúcsból induló legrövidebb utak, minden párra számított legrövidebb utak, súlyozott kétoldalú illesztések és minimális feszítőfák.

Később Driscoll, Gabow, Shrairman és Tarjan [81] bevezette a „laza kupacokat”mint a Fibonacci-kupacok egyik módosítását. A laza kupacoknak két változata van. Az egyik ugyanazokat az amortizált időket adja, mint a Fibonacci-kupacok. A másikban a KULCSOT-CSÖKKENT eljárás legrosszabb esetben $O(1)$ (nem amortizált) időben fut, és a MINIMUMOT-KIVÁG és TÖRÖL legrosszabb futási ideje $O(\lg n)$. A laza kupacok a párhuzamos algoritmusokban a Fibonacci-kupacoknál is előnyösebben alkalmazhatók.

A 6. fejezet Megjegyzések a fejezethez részében olyan további adatszerkezeteket találhatunk, amelyek gyors KULCSOT-CSÖKKENT műveletet tesznek lehetővé, ha a KULCSOT-CSÖKKENT hívás által adott értéksorozat időben monoton növekszik és az adatok egy adott intervallumba eső egész számok.

21. Adatszerkezetek diszjunkt halmazokra

Vannak olyan alkalmazások, ahol szükség van arra, hogy n különálló elemet diszjunkt halmazokba csoportosítsunk. Ekkor két fontos műveletről beszélhetünk. Az egyik: meg kell találnunk, hogy egy adott elem melyik halmazban van benne, és a másik: két halmaz egyesítése. Ebben a fejezetben olyan adatstruktúrák kezeléséről lesz szó, amelyekben e két művelet könnyen elvégezhető.

A 21.1. alfejezetben leírjuk azokat a műveleteket, amelyek a diszjunkt-halmaz adatszerkezetekre vonatkoznak, és egyszerű alkalmazásokat mutatunk be. A 21.2. alfejezetben megnézzük a diszjunkt halmazok egyszerű irányított listával történő ábrázolását. A 21.3. alfejezetben egy olyan hatékonyabb ábrázolást adunk meg, amely gyökércsúccsal rendelkező fákat használ. A fákat használó ábrázolásban a futási idő gyakorlati szempontból lineáris, de elméletileg a lineárisnál több időt használ. A 21.4. alfejezetben definiálunk és megvizsgálunk egy nagyon gyorsan növekedő függvényt és a függvény nagyon lassan növekedő inverzét, amelyek a fákat használó megvalósításban a műveletek futási idejében szerepelnek, majd ezután az amortizált elemzést használjuk arra, hogy a futási időre egy olyan felső korlátot bizonyítsunk, amely éppen hogy csak szuperlineáris.

21.1. Diszjunkt-halmaz műveletek

A ***diszjunkt-halmaz adatszerkezet*** diszjunkt dinamikus halmazok $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ együttese. Mindegyik halmazt egy ***képviselője*** azonosít, ez a halmaznak egy eleme. Bizonyos alkalmazásokban közömbös, hogy a halmaz melyik eleme a halmaz képviselője; az a lényeges, hogy ha kétszer megkérdezzük, hogy melyik elem a dinamikus halmaz képviselője, akkor mindkét esetben ugyanazt a választ kell kapnunk, feltéve, hogy közben a halmazt nem változtattuk meg. Más alkalmazásokban egy előre megadott szabály van a képviselő meghatározására. Ilyen szabály például az, hogy a halmaz legkisebb eleme a halmaz képviselője (feltéve természetesen, hogy az elemeket lehet rendezni).

Mint a már korábban tanulmányozott dinamikus-halmaz megvalósításokban, itt is objektumokkal ábrázoljuk egy halmaz elemeit. A következő műveleteket használjuk (az x egy objektumot jelöl):

HALMAZT-KÉSZÍT(x) egy olyan új halmazt hoz létre, amelynek csak ez az egy x eleme van, és ez az elem a halmaz képviselője is. Mivel a halmazoknak diszjunktaknak kell lenniük, az x nem lehet a már meglévő halmazok egyikének sem eleme.

EGYESÍT(x, y) az x-et tartalmazó S_x és az y-t tartalmazó S_y halmazokat egyesíti egy új halmazba, ez az új halmaz a két halmaz uniója. Feltesszük, hogy a művelet végrehajtása előtt a két halmaz diszjunkt. Az eredményül kapott halmaz képviselője az $S_x \cup S_y$ egy tetszőleges eleme, bár sok megvalósításban az EGYESÍT művelet speciálisan vagy az S_x, vagy az S_y képviselőjét választja új képviselőnek. Mivel feltételezzük, hogy a halmazok diszjunkt halmazok, a művelet az S_x és az S_y halmazokat „lerombolja”, azaz kiveszi őket az $\mathcal{S}$-ből.

HALMAZT-KERES(x) visszaad egy mutatót, amelyik annak a halmaznak a képviselőjére mutat, amelyben az x benne van.

Ebben a fejezetben a diszjunkt-halmaz adatszerkezetek futási idő it kétváltozós kifejezésekkel adjuk meg, az egyik változó az n, a HALMAZT-KÉSZÍT műveletek száma, a másik pedig m, a HALMAZT-KÉSZÍT, EGYESÍT és HALMAZT-KERES műveletek együttes száma. Mivel a halmazok diszjunktak, mindegyik EGYESÍT művelet eggyel csökkenti a halmazok számát. Ezért $n - 1$ EGYESÍT művelet után csak egy halmaz marad. Így az EGYESÍT műveletek száma legfeljebb $n - 1$ lehet. Megjegyezzük, hogy mivel a HALMAZT-KÉSZÍT benne van m-ben, a műveletek összes számában, fennáll az $m \geq n$ összefüggés. Feltesszük, hogy az elsőnek n HALMAZT-KÉSZÍT műveletet hajtunk végre.

Diszjunkt-halmaz adatszerkezetek alkalmazása

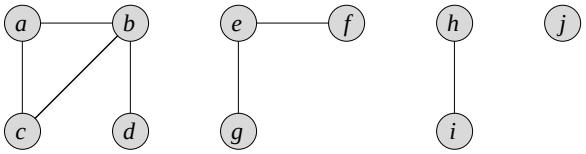
Diszjunkt-halmaz adatszerkezetet alkalmazunk akkor, amikor egy irányítatlan gráf összefüggő komponenseit határozzuk meg (lásd a B.4. alfejezetet). Például, a 21.1(a) ábrán látható egy gráf, amelynek négy összefüggő komponense van.

Az ÖSSZEFÜGGŐ-KOMPONENSEK eljárás diszjunkt-halmaz műveleteket használ arra, hogy meghatározza egy gráf összefüggő komponenseit. Ha az ÖSSZEFÜGGŐ-KOMPONENSEK eljárást már lefuttattuk, mint egy előfeldolgozó programot, akkor az UGYANAZ-A-KOMPONENS függvény meg tudja határozni, hogy két él ugyanabban az összefüggő komponensben van-e.[1] (Egy G gráf csúcsainak halmazát $V[G]$-vel, a gráf éleinek halmazát $E[G]$-vel jelöljük.)

```
ÖSSZEFÜGGŐ-KOMPONENSEK(G)
1  for minden v ∈ V[G] csúcsra
2      do HALMAZT-KÉSZÍT(v)
3  for minden (u, v) ∈ E[G] élre
4      do if HALMAZT-KERES(u) ≠ HALMAZT-KERES(v)
5             then EGYESÍT(u, v)
```

```
UGYANAZ-A-KOMPONENS(u, v)
1  if HALMAZT-KERES(u) = HALMAZT-KERES(v)
2    then return IGAZ
3    else  return HAMIS
```

[1] Ha egy gráf élei „statikusak” – időben nem változnak –, akkor az összefüggő komponenseket gyorsabban meg lehet határozni a mélységi kereséssel (22.3-11. gyakorlat). Néha azonban a gráfot új élekkel kell „dinamikusan” bővíteni, és ekkor újra meg kell határozni az új összefüggő komponenseket. Ebben az esetben a fenti megvalósítás az új élek keresésére sokkal hatékonyabb, mint a mélységi keresés.



(a)

Feldolgozott él	Diszjunkt halmazok összessége									
kezdőhalmazok	{*a*}	{*b*}	{*c*}	{*d*}	{*e*}	{*f*}	{*g*}	{*h*}	{*i*}	{*j*}
(*b*,*d*)	{*a*}	{*b*,*d*}	{*c*}		{*e*}	{*f*}	{*g*}	{*h*}	{*i*}	{*j*}
(*e*,*g*)	{*a*}	{*b*,*d*}	{*c*}		{*e*,*g*}	{*f*}		{*h*}	{*i*}	{*j*}
(*a*,*c*)	{*a*,*c*}	{*b*,*d*}			{*e*,*g*}	{*f*}		{*h*}	{*i*}	{*j*}
(*h*,*i*)	{*a*,*c*}	{*b*,*d*}			{*e*,*g*}	{*f*}		{*h*,*i*}		{*j*}
(*a*,*b*)	{*a*,*b*,*c*,*d*}				{*e*,*g*}	{*f*}		{*h*,*i*}		{*j*}
(*e*, *f*)	{*a*,*b*,*c*,*d*}				{*e*, *f*,*g*}			{*h*,*i*}		{*j*}
(*b*,*c*)	{*a*,*b*,*c*,*d*}				{*e*, *f*,*g*}			{*h*,*i*}		{*j*}

(b)

21.1. ábra. (a) Gráf négy összefüggő komponenssel: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$ és {j}. **(b)** A diszjunkt halmazok összessége az élek feldolgozása után.

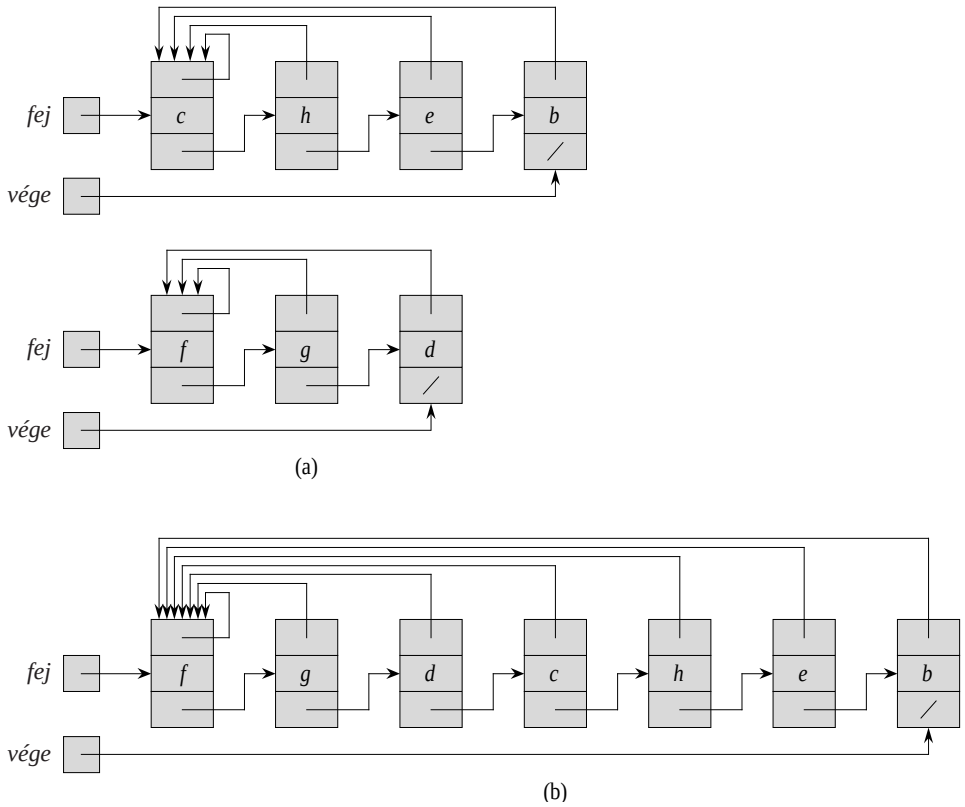
Az ÖSSZEFÜGGŐ-KOMPONENSEK eljárás először minden v csúcsot elhelyez a saját halmazában. Ezután minden (u, v) élre egyesíti az u-t és a v-t tartalmazó halmazokat. A 21.1-2. gyakorlat alapján, az összes él feldolgozása után két csúcs akkor és csak akkor van egy összefüggő komponensben, ha a megfelelő objektumok ugyanabban a halmazban vannak. Az ÖSSZEFÜGGŐ-KOMPONENSEK eljárás a halmazokat úgy adja meg, hogy az UGYANAZ-A-KOMPONENS függvény meg tudja határozni, hogy két csúcs ugyanabban az összefüggő komponensben van-e. A 21.1(b) ábrán látható, hogyan határozza meg az ÖSSZEFÜGGŐ-KOMPONENSEK a diszjunkt halmazokat.

Ennek az összefüggő-komponensek eljárásnak a konkrét megvalósításában a gráf és a diszjunk-halmaz adatszerkezet ábrázolásainak hivatkozniuk kell egymásra. Egy csúcsot ábrázoló objektumnak egy olyan pointert kell tartalmaznia, amelyik a megfelelő diszjunkt-halmaz objektumra mutat, és fordítva, a diszjunkt-halmazt ábrázoló objektumból is kell egy mutató a csúcs objektumára. Ezek a programozási részletek a megvalósítás programnyelvétől függnek, és ezekkel a továbbiakban nem foglalkozunk.

Gyakorlatok

21.1-1. Tegyük fel, hogy az ÖSSZEFÜGGŐ-KOMPONENSEK eljárás a $G = (V, E)$ irányítatlan gráfot kapja bemenetként, ahol $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ és az E-beli éleket a következő sorrendben dolgozza fel: $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f)$, $(g, j), (a, e), (i, d)$. Határozzuk meg az összefüggő komponensekben levő csúcsokat, az eljárás 3–5. soraiban levő iterációk végrehajtása után.

21.1-2. Mutassuk meg, hogy miután az ÖSSZEFÜGGŐ-KOMPONENSEK eljárás az összes élt feldolgozta, két csúcs akkor és csak akkor van ugyanabban az összefüggő komponensben, ha ugyanabban a halmazban vannak.



21.2. ábra. (a) Két halmaz láncolt listás ábrázolása. Az egyik halmaz elemei b, c, e és h, a képviselőjük c; a másik halmaz elemei d, f és g, képviselőjük f. A lista mindegyik eleme tartalmazza a halmaz egy elemét, egy mutatót a lista következő elemére, és egy mutatót, amelyik a lista első elemére, a halmaz képviselőjére mutat. Mindegyik listának van egy *fej* és egy *vége* pointere, amelyik a lista első, illetve a lista utolsó elemére mutat. **(b)** Az EGYESÍT(e, g) művelet eredménye. Az eredményül kapott halmaz képviselője f.

21.1-3. Egy k összefüggő komponensű $G = (V, E)$ irányítatlan gráfra az ÖSSZEFÜGGŐ-KOMPONENSEK eljárás hányszor hívja meg a HALMAZT-KERES eljárást? Hányszor hívja meg az EGYESÍT eljárást? Fejezzük ki a válaszokat $|V|, |E|$ és k függvényeként.

21.2. Diszjunkt halmazok láncolt listás ábrázolása

A diszjunkt-halmaz adatszerkezetek megvalósításának egy egyszerű módja az, ha a halmazokat láncolt listával ábrázoljuk. Mindegyik láncolt lista első eleme a halmaz képviselője. A láncolt lista mindegyik eleme tartalmazza a halmaz egy elemét, egy mutatót, amelyik a halmaz következő elemére mutat, és egy másik mutatót, amelyik a halmaz képviselőjére mutat. Mindegyik listának van egy, a képviselőre mutató *fej* pointere és a lista utolsó elemére mutató *vége* pointere. A 21.2(a) ábrán két halmaz látható. A láncolt listákon belül az elemek sorrendje tetszőleges (azzal a megkötéssel, hogy mindegyik listában az első elem a halmaz képviselője).

Művelet	Megváltoztatott elemek száma
HALMAZT-KÉSZÍT(x_1)	1
HALMAZT-KÉSZÍT(x_2)	1
⋮	⋮
HALMAZT-KÉSZÍT(x_n)	1
EGYESÍT(x_1, x_2)	1
EGYESÍT(x_2, x_3)	2
EGYESÍT(x_3, x_4)	3
⋮	⋮
EGYESÍT(x_{n-1}, x_n)	$n-1$

21.3. ábra. Egy n elemen végzett $2n-1$ műveletből álló sorozat, amelyik $\Theta(n^2)$ ideig vagy átlagosan műveletenként Θn ideig tart, ha a halmazok láncolt listás ábrázolását és az EGYESÍT egyszerű megvalósítását használjuk.

Ebben a láncolt listás ábrázolásban mind a HALMAZT-KÉSZÍT, mind a HALMAZT-KERES könnyű, $O(1)$ időt igényel. A HALMAZT-KÉSZÍT(x) végrehajtásakor egy olyan új láncolt listát készítünk, amelynek csak egy eleme van, ez az elem az x. A HALMAZT-KERES(x) eljárásban az x-re mutató pointert a halmaz képviselőjére állítjuk át.

Az egyesítés egyszerű megvalósítása

A láncolt listás halmaz ábrázolására megadható EGYESÍT művelet legegyszerűbb megvalósítása sokkal több futási időt igényel, mint a HALMAZT-KÉSZÍT, vagy a HALMAZT-KERES. Mint a 21.2(b) ábrán is látható, az EGYESÍT(x, y) végrehajtásakor az x-et tartalmazó listát az y-t tartalmazó lista végéhez fűzzük. Az y listájának *vége* pointerét használjuk arra, hogy gyorsan megtaláljuk, hová kell az x listáját illeszteni. Az eredményül kapott halmaz képviselője az y-t tartalmazó halmaz képviselője lesz. Sajnos, ekkor az x-et tartalmazó lista minden elemében módosítani kell a képviselőre mutató pointert, ez az x-et tartalmazó lista hosszával egyenesen arányos időt igényel.

Valóban, nem nehéz találni n elemen végzett m műveletből álló olyan sorozatot, amely $\Theta(n^2)$ ideig tart. Tegyük fel, hogy az elemeink $x_1, x_2, \ldots, x_n$. Hajtsunk végre – a 21.3. ábrán is látható – n HALMAZT-KÉSZÍT, majd ezután $n-1$ EGYESÍT műveletet, így $m = 2n-1$. Eltöltünk $\Theta(n)$ időt az n darab HALMAZT-KÉSZÍT művelet végrehajtásával. Mivel az i-edik EGYESÍT művelet i elemet változtat meg, az $n-1$ EGYESÍT művelet által megváltoztatott elemek darabszáma

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

lesz. A műveletek száma $2n-1$, és ezért átlagosan mindegyik művelethez $\Theta(n)$ idő kell. Azaz egy művelet amortizált ideje $\Theta(n)$.

Egy súlyozott-egyesítés heurisztika

A legrosszabb esetben, az EGYESÍT eljárás fenti megvalósítása hívásonként átlagosan $\Theta(n)$ időt igényel, mivel egy hosszabb listát is hozzáfűzhetünk egy rövidebb listához; a hosszú

lista mindegyik elemét módosítani kell, a képviselőre mutató pointert a hosszú lista mindegyik elemében meg kell változtatni. Tegyük fel azonban, hogy mindegyik listában van egy lista hossza mező (amit könnyen lehet kezelni), és hogy mindig a rövidebb listát fűzzük a hosszabbikhoz, ha a listák egyforma hosszúak, akkor tetszőlegesen választhatunk, hogy melyik legyen az első. Ezzel a ***súlyozott-egyesítés heurisztikával*** egy EGYESÍT művelet $\Omega(n)$ ideig tart, ha mindkét halmaznak $\Omega(n)$ eleme van. Azonban, amint a következő tételből is látni fogjuk, egy m hosszúságú HALMAZT-KÉSZÍT, EGYESÍT és HALMAZT-KERES műveletekből álló olyan sorozat, amelyben n HALMAZT-KÉSZÍT művelet van, $O(m + n \lg n)$ idő alatt végrehajtható.

21.1. tétel. *A diszjunkt halmazok láncolt listás ábrázolását és a súlyozott-egyesítés heurisztikát használva az m hosszúságú,* HALMAZT-KÉSZÍT, EGYESÍT *és* HALMAZT-KERES *műveletekből álló és n* HALMAZT-KÉSZÍT *műveletet tartalmazó sorozat $O(m + n \lg n)$ idő alatt végrehajtható.*

Bizonyítás. Azzal kezdjük, hogy egy n elemű halmaz minden elemére egy felső korlátot adunk arra, hányszor kell módosítani az elemekben a képviselőre mutató pointereket. Legyen x egy rögzített elem. Tudjuk, hogy mindig, amikor az x-nek a képviselőre mutató pointerét módosítani kell, x a kisebbik halmazban van. Az első olyan esetben, amikor a mutatót módosítani kell, az eredményül kapott halmaznak legalább 2 eleme lett. Hasonlóan, az x képviselőre mutató pointerének második módosításakor az eredményhalmaznak legalább 4 eleme lett. Tovább folytatva, megfigyelhetjük, hogy minden $k \leq n$-re, az x képviselőre mutató pointerének $\lceil \lg k \rceil$-adik módosítása után az eredményül kapott halmaz legalább k elemű. Mivel a legnagyobb halmaznak legfeljebb n eleme van, az EGYESÍT művelet végrehajtása alatt mindegyik képviselőre mutató pointert legfeljebb $\lceil \lg n \rceil$-szer módosítjuk. Figyelembe kell vennünk a *fej* és *vége* pointereknek, valamint a lista hossza mezőknek a módosításait is, ezeknek az időigénye azonban csak $\Theta(1)$ minden EGYESÍT műveletre. Így az n elem módosításához szükséges idő $O(n \lg n)$.

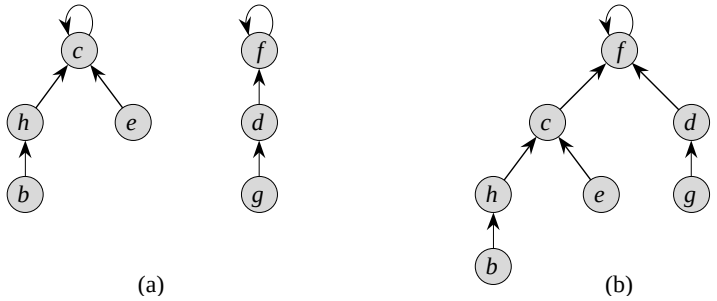
Az m műveletből álló sorozat teljes időigényét ezután már könnyű meghatározni. Mindegyik HALMAZT-KÉSZÍT és HALMAZT-KERES művelet $O(1)$ ideig tart, és így együttes idejük $O(m)$. A teljes sorozat időigénye tehát $O(m + n \lg n)$. ■

Gyakorlatok

21.2-1. Írjuk meg a láncolt listás ábrázolást és a súlyozott-egyesítés heurisztikát használó HALMAZT-KÉSZÍT, HALMAZT-KERES és EGYESÍT pszeudokódokat. Legyen mindegyik x elemnek egy *képv*$[x]$ mezője, amelyik az x-et tartalmazó halmaz képviselőjére mutat, és mindegyik S halmaznak legyen egy *fej*$[S]$, *vége*$[S]$ és egy *méret*$[S]$ attribútuma, ez utóbbi a lista hosszával legyen egyenlő.

21.2-2. A 21.1. tétel tömör bizonyítását felhasználva bizonyítsuk be, hogy a láncolt listás ábrázolást és a súlyozott-egyesítés heurisztikát használva a HALMAZT-KÉSZÍT, HALMAZT-KERES műveletekre $O(1)$, az EGYESÍT műveletre $O(\lg n)$ amortizált időkorlátot kapunk.

21.2-3. Feltételezve, hogy a láncolt listás ábrázolást és a súlyozott-egyesítés heurisztikát használjuk, adjunk egy erősebb aszimptotikus korlátot a 21.3. ábrán látható műveletsorozat végrehajtási idejére.



21.4. ábra. Egy diszjunkt-halmaz erdő. **(a)** Két fa, amely a 21.2. ábrán látható két halmaznak felel meg. A bal oldali fa a $\{b, c, e, h\}$ halmaznak felel meg, képviselője c, a jobb oldali fa a $\{d, f, g\}$ halmazt ábrázolja, ennek a képviselője f. **(b)** Az EGYESÍT(e, g) művelet eredménye.

21.2-4. Mutassuk meg azt az adatszerkezetet, amit a következő program eredményez, és azokat az adatszerkezeteket, amiket a programban levő HALMAZT-KERES műveletekre eredményül kapunk. Használjuk a láncolt listás ábrázolást és a súlyozott-egyesítés heurisztikát.

```
 1  for i ← 1 to 16
 2      do HALMAZT-KÉSZÍT(x_i)
 3  for i ← 1 to 15 by 2
 4      do EGYESÍT(x_i, x_{i+2})
 5  for i ← 1 to 13 by 4
 6      do EGYESÍT(x_i, x_{i+2})
 7  EGYESÍT(x_1, x_5)
 8  EGYESÍT(x_11, x_13)
 9  EGYESÍT(x_1, x_10)
10  HALMAZT-KERES(x_2)
11  HALMAZT-KERES(x_9)
```

Tegyük fel, hogy ha az x_i-t és az x_j-t tartalmazó halmazok azonos méretűek, akkor az EGYESÍT(x_i, x_j) művelet az x_j listáját fűzi az x_i listájához.

21.2-5. A láncolt listás ábrázolás esetére adjunk meg az EGYESÍT eljárásra egy olyan módosítást, hogy ne használjuk a listákban az utolsó elemre mutató *vége* pointert. Akár használjuk a súlyozott-egyesítés heurisztikát, akár nem, ez a módosítás ne változtassa meg az EGYESÍT eljárás aszimptotikus futási időigényét. (*Útmutatás.* A listák egymáshoz fűzése helyett a listákat kapcsoljuk össze.)

21.3. Diszjunkt-halmaz erdők

A diszjunkt halmazok egy gyorsabb megvalósításában a halmazokat gyökeres fákkal ábrázoljuk úgy, hogy a fa minden csúcsa a halmaz egy elemének, és mindegyik fa egy halmaznak felel meg. A 21.4(a) ábrán is látható ***diszjunkt-halmaz erdőben*** mindegyik elem csak a szülőjére mutat. Mindegyik fának a gyökércsúcsa a halmaz képviselőjét tartalmazza, és a gyökércsúcs saját magának a szülője. Mint majd látni fogjuk, azok az algoritmusok, amelyek ezeket az ábrázolásokat használják, nem olyan gyorsak, mint a láncolt listás

ábrázolások, de bevezetve két új heurisztikát – a „rang szerinti egyesítést” és az „úttömörítést” –, az aszimptotikusan leggyorsabb ismert diszjunkt-halmaz adatszerkezetet kapjuk meg.

Három diszjunkt-halmaz műveletet adunk meg. A HALMAZT-KÉSZÍT művelet egyszerűen egy olyan fát készít, amelynek csak egy csúcsa van. A HALMAZT-KERES műveletet a szülő mutatókat járja be, addig, amíg a fa gyökércsúcsát meg nem találja. Azok a csúcsok, amelyeket a gyökércsúcs felé haladva ezen az úton a művelet bejár, a ***keresési utat*** alkotják. Az EGYESÍT művelet, amelyet a 21.4(b) ábrán láthatunk, azt eredményezi, hogy az egyik fa gyökércsúcsából a másik fa gyökerébe mutat egy pointer.

A futási időt javító heurisztikák

Eddig még nem tökéletesítettük a láncolt listás megvalósítást. Az $n - 1$ EGYESÍT műveletből álló sorozat készíthet egy olyan fát is, ami olyan, mint egy n csúcsból álló lineáris lánc. Két heurisztikát használva azonban olyan futási időt érhetünk el, ami már a műveletek m számának lineáris függvénye.

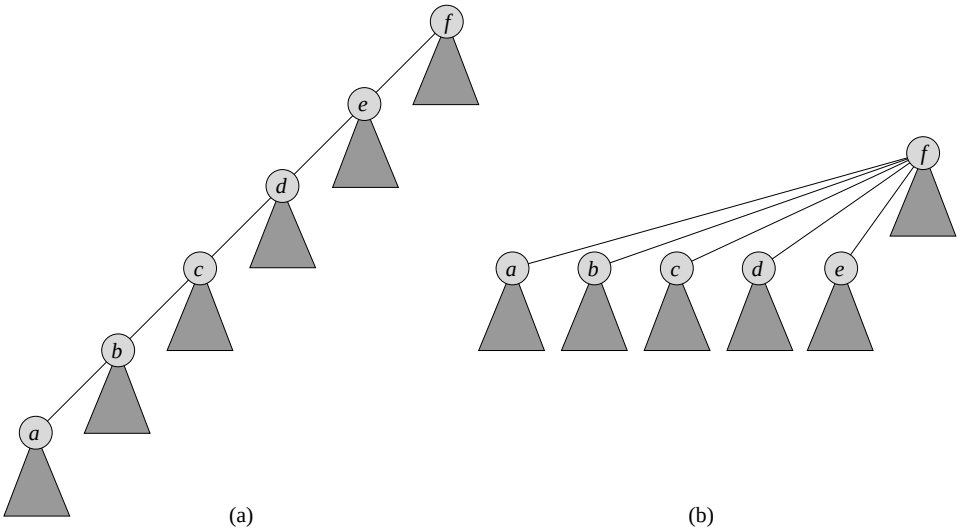
Az első heurisztika, a ***rang szerinti egyesítés*** hasonló ahhoz a súlyozott-egyesítés heurisztikához, amit a láncolt lista ábrázolásban használtunk. Az elv az, hogy a kevesebb csúcsból álló fát kapcsoljuk egy mutatóval a több csúcsból álló fához. Ahelyett, hogy minden csúcsra pontosan nyilvántartanánk a csúcshoz tartozó részfa méretét, egy olyan közelítést használunk, ami megkönnyíti az elemzést. Minden csúcsra megadjuk a csúcs ***rangját,*** ami a csúcs magasságának felső korlátja. A rang szerinti egyesítést használva az EGYESÍT művelet a kisebb rangú gyökércsúcsot fogja egy mutatóval a nagyobb rangú gyökérhez kapcsolni, a mutató a kisebb rangú gyökérből a nagyobb rangú gyökérbe mutat.

A másik heurisztika, az ***úttömörítés,*** szintén egyszerű és nagyon hatékony. Ahogyan a 21.5. ábrán is látható, a HALMAZT-KERES művelet alatt használjuk, az úttömörítés a keresési út minden csúcsát közvetlenül a gyökércsúcshoz kapcsolja. Az út tömörítése nem változtatja meg a csúcsok rangját.

Programok diszjunkt-halmaz erdőkre

Ahhoz, hogy a rang szerinti egyesítés heurisztikát használó diszjunkt-halmaz erdőket megvalósíthassuk, először a ranggal kell foglalkoznunk. Minden x csúcshoz hozzárendelünk egy $rang[x]$ egész értéket, ez az érték az x magasságának a felső korlátja (ahol az x magassága az x és az x gyökércsúcsú részfa levelei közötti leghosszabb út éleinek a száma). Amikor a HALMAZT-KÉSZÍT művelettel egyetlen halmazt készítünk, a létrehozott fa egyetlen csúcsának a rangja 0. A HALMAZT-KERES műveletek a rangokat nem változtatják meg. Amikor az EGYESÍT műveletet alkalmazzuk két fára, két eset lehetséges, attól függően, hogy a gyökércsúcsok rangja megegyezik-e vagy sem. Ha a gyökércsúcsok rangja nem egyezik meg, akkor a nagyobb rangú fa gyökércsúcsa lesz a kisebb rangú fa gyökércsúcsának a szülője. Ha a rangok egyenlőek, akkor tetszőlegesen választhatjuk meg, hogy melyik gyökércsúcs legyen a szülő, és a szülő rangját eggyel növeljük.

Adjuk meg ennek az eljárásnak a programját. Az x csúcs szülőjére a $p[x]$ mutat. Az EGYESÍT által meghívott ÖSSZEKAPCSOL eljárás bemenete két gyökércsúcsra mutató pointer.



21.5. ábra. Úttömörítés a HALMAZT-KERES művelet végrehajtása alatt. A nyilakat és a gyökércsúcs önmagára mutató pointerét elhagytuk. **(a)** Egy fa, amely egy halmazt ábrázol, a HALMAZT-KERES(a) végrehajtása előtt. A háromszögek részfákat jelölnek, csak a részfák gyökércsúcsait ábrázoljuk. Minden csúcsból egy pointer mutat a csúcs szülőjére. **(b)** Ugyanaz a halmaz a HALMAZT-KERES(a) művelet végrehajtása után. A keresési út csúcsai most közvetlenül a gyökércsúcsra mutatnak.

HALMAZT-KÉSZÍT(x)

```
1  p[x] ← x
2  rang[x] ← 0
```

EGYESÍT(x, y)

```
1  ÖSSZEKAPCSOL(HALMAZT-KERES(x),HALMAZT-KERES(y))
```

ÖSSZEKAPCSOL(x, y)

```
1  if rang[x] > rang[y]
2     then p[x] ← x
3     else p[x] ← y
4          if rang[x] = rang[y]
5             then rang[y] ← rang[y] + 1
```

Az úttömörítést használó HALMAZT-KERES eljárás is egyszerű:

HALMAZT-KERES(x)

```
1  if x ≠ p[x]
2     then p[x] ← HALMAZT-KERES(p[x])
3  return p[x]
```

A HALMAZT-KERES eljárás ***kétmenetes:*** az első menetben a fában felfelé haladva felépíti a gyökércsúcsba vezető keresési utat, a második menetben, lefelé haladva a keresési úton, az út mindegyik csúcsát közvetlenül a gyökérhez fűzi hozzá. A HALMAZT-KERES(x) mindegyik hívása a 3. sorban $p[x]$-et adja vissza eredményül. Ha x a gyökércsúcs, akkor a 2. sor nem hajtódik végre, és $p[x] = x$ lesz az eredmény. Ez az az eset, amikor a rekurzió befejeződik. Ellenkező esetben a 2. sor végrehajtódik, és a $p[x]$ paraméterű rekurzív hívás egy gyökérre mutató pointert ad vissza. A 2. sorban módosítjuk az x csúcsot úgy, hogy a pointere a gyökérre mutasson, és ezt a pointert adjuk vissza a 3. sorban.

A heurisztikák hatása a futási időre

Külön-külön mind a rang szerinti egyesítés, mind az úttömörítés javítja a diszjunkt-halmaz erdőkre megadott műveletek futási idejét, de a javítás még nagyobb, ha ezt a két heurisztikát együtt alkalmazzuk. Ha a rang szerinti egyesítést önmagában alkalmazzuk, a futási időigény $O(m \lg n)$ (lásd a 21.4-4. gyakorlatot), és ez éles korlát (lásd a 21.3-3. gyakorlatot). Bár itt nem bizonyítjuk, igaz a következő állítás: ha van n HALMAZT-KÉSZÍT művelet (és így legfeljebb $n - 1$ EGYESÍT művelet), és f HALMAZT-KERES művelet, akkor egyedül az úttömörítés heurisztika a legrosszabb esetben $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$ futási időigényt ad.

Ha a rang szerinti egyesítés és az úttömörítés mindegyikét alkalmazzuk, a legrosszabb futási idő $O(m\,\alpha(n))$, ahol $\alpha(n)$ egy *nagyon* lassan növekedő függvény, ezt a függvényt a 21.4. alfejezetben definiáljuk. A diszjunkt adatszerkezetek minden elképzelhető alkalmazásában $\alpha(n) \leq 4$; így látható, hogy a futási idő m-re nézve lineáris minden gyakorlati alkalmazásban. A 21.4. alfejezetben ezt a felső korlátot be fogjuk bizonyítani.

Gyakorlatok

21.3-1. Oldjuk meg a 21.2-4. gyakorlatot diszjunk-halmaz erdőkre, használva a rang szerinti egyesítést és az úttömörítést.

21.3-2. Írjuk meg az úttömörítést használó HALMAZT-KERES rekurziómentes változatát.

21.3-3. Adjunk meg egy olyan m HALMAZT-KÉSZÍT, EGYESÍT és HALMAZT-KERES műveletből álló sorozatot (a műveletek között n HALMAZT-KÉSZÍT legyen), melynek végrehajtása $\Omega(m \lg n))$ ideig tart, ha a rang szerinti egyesítést használjuk.

21.3-4.⋆ Mutassuk meg, hogy egy m HALMAZT-KÉSZÍT, HALMAZT-KERES és ÖSSZEKAPCSOL műveletből álló sorozat, amelyben mindegyik ÖSSZEKAPCSOL művelet megelőzi az összes HALMAZT-KERES műveletet, időigénye $O(m)$, ha a rang szerinti egyesítés és az úttömörítés heurisztika mindegyikét alkalmazzuk. Mi történik, ha az ugyanilyen sorozatra csak az úttömörítés heurisztikát használjuk?

⋆ 21.4. A rang szerinti egyesítés és az úttömörítés együttes használatának elemzése

Mint a 21.3. alfejezetben említettük, egy n elemű diszjunkt halmazon egy m diszjunkt-halmaz műveletből álló sorozat végrehajtási időigénye, együtt használva a rang szerinti egyesítés és az úttömörítés heurisztikát, $O(m\,\alpha(n))$. Ebben az alfejezetben megvizsgáljuk az α függvényt, hogy lássuk, milyen lassan nő. Ezután az amortizált elemzés potenciálmódszerét alkalmazva bebizonyítjuk ezt a végrehajtási időigényt.

Egy nagyon gyorsan növekedő függvény és nagyon lassan növekedő inverze

Definiáljuk a $k \geq 0$ és $j \geq 1$ egész számokra az $A_k(j)$ függvényt:

$$A_k(j) = \begin{cases} j+1, & \text{ha} \quad k = 0, \\ A_{k-1}^{(j+1)}(j), & \text{ha} \quad k \geq 1, \end{cases}$$

ahol az $A_{k-1}^{(j+1)}(j)$-ben a 3.2. alfejezetben megadott függvényiteráció jelölést használjuk. Speciálisan, $A_{k-1}^{(0)}(j) = j$ és $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ minden $i \geq 1$-re. A k paramétert az A függvény ***szintjének*** nevezzük.

Az $A_k(j)$ függvény mind j-re, mind k-ra szigorúan növekszik. Azért, hogy lássuk, ez a függvény milyen gyorsan nő, először az $A_1(j)$ és $A_2(j)$ kifejezések zárt formáját adjuk meg.

21.2. lemma. *Minden $j \geq 1$ egész számra $A_1(j) = 2j + 1$.*

Bizonyítás. Először i szerinti teljes indukcióval megmutatjuk, hogy $A_0^{(i)}(j) = j + i$. Az alapesetben, $i = 0$-ra, $A_0^{(0)}(j) = j = j+0$. Az indukciós feltevés az, hogy $A_0^{(i-1)}(j) = j+(i-1)$. Ekkor $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j+(i-1))+1 = j+i$. Végül, $A_1(j) = A_0^{(j+1)}(j) = j+(j+1) = 2j + 1$. ■

21.3. lemma. *Minden $j \geq 1$ egész számra $A_2(j) = 2^{j+1}(j + 1) - 1$.*

Bizonyítás. Először i szerinti teljes indukcióval megmutatjuk, hogy $A_1^{(i)}(j) = 2^i(j + i) - 1$. Az alapesetben, $i = 0$-ra, $A_1^{(0)}(j) = j = 2^0(j + 1) - 1$. Az indukciós feltevés az, hogy $A_1^{(i-1)}(j) = 2^{i-1}(j + 1) - 1$. Ekkor $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j + 1) - 1) = 2 \cdot (2^{i-1}(j + 1) - 1) + 1 = 2^i(j + 1) - 2 + 1 = 2^i(j + 1) - 1$. Végül, $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j + 1) - 1$. ■

Ezek után most már megnézhetjük, hogy az $A_k(j)$ milyen gyorsan növekszik, egyszerűen meghatározva az $A_k(1)$-t a $k = 0, 1, 2, 3, 4$ szintszámokra. Az $A_0(k)$ definíciójából és a fenti lemmákból kapjuk, hogy $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$ és $A_2(1) = 2^{1+1} \cdot (1 + 1) - 1 = 7$. Hasonlóan,

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047, \end{aligned}$$

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \\ &\gg A_2(2047) \end{aligned}$$

$$\begin{aligned} &= 2^{2048} \cdot 2048 - 1 \\ &> 2^{2048} \\ &= (2^4)^{512} \\ &= 16^{512} \\ &\gg 10^{80}, \end{aligned}$$

és ez a szám az ismert univerzum atomjainak becsült darabszáma.

Most az $A_k(n)$ függvény inverzét definiáljuk, legyen az $n \geq 0$ egész számokra

$$\alpha(n) = \min\{k : A_k(1) \geq n\}.$$

Szavakkal kifejezve, $\alpha(n)$ a legkisebb olyan szintszám, amelyre $A_k(1)$ legalább n. Az $A_k(1)$ fenti értékeiből láthatjuk, hogy

$$\alpha(n) = \begin{cases} 0, & \text{ha} \quad 0 \leq n \leq 2, \\ 1, & \text{ha} \quad n = 3, \\ 2, & \text{ha} \quad 4 \leq n \leq 7, \\ 3, & \text{ha} \quad 8 \leq n \leq 2047, \\ 4, & \text{ha} \quad 2048 \leq n \leq A_4(1). \end{cases}$$

Csak a gyakorlatban nem használt nagy n értékekre (azaz ha n nagyobb az $A_4(1)$ óriási számnál) igaz, hogy $\alpha(n) > 4$, és így $\alpha(n) \leq 4$ teljesül minden gyakorlati esetben.

A rang tulajdonságai

Az alfejezet hátralevő részében bebizonyítjuk az $O(m\,\alpha(n))$ futási időkorlátot, amit a diszjunkt-halmaz műveletekre kapunk, ha a rang szerinti egyesítést és az úttömörítést használjuk. A korlát igazolásához először a rang néhány egyszerű tulajdonságát bizonyítjuk.

21.4. lemma. *Minden x csúcsra $rang[x] \leq rang[p[x]]$, és szigorú egyenlőtlenség áll fenn, ha $x \neq p[x]$. A $rang[x]$ kezdetben 0, és addig növekszik, amíg $x \neq p[x]$; ettől kezdve a $rang[x]$ nem változik. A $rang[p[x]]$ függvény az idő függvényében monoton növekedő függvény.*

Bizonyítás. A bizonyítás a 21.3. alfejezetben megadott HALMAZT-KÉSZÍT, EGYESÍT és HALMAZT-KERES műveletek számára vonatkozó teljes indukcióval végezhető. A bizonyítás elvégzését a 21.4-1. gyakorlatban tűzzük ki. ■

21.5. következmény. *Egy tetszőleges csúcsból a gyökércsúcs felé haladó úton a csúcsok rangjai határozottan nőnek.*

21.6. lemma. *Minden csúcs rangja legfeljebb $n - 1$.*

Bizonyítás. Kezdetben minden csúcs rangja 0, és a rangot csak az ÖSSZEKAPCSOL művelet növeli. Mivel legfeljebb $n - 1$ EGYESÍT művelet van, legfeljebb $n - 1$ ÖSSZEKAPCSOL művelet lehet. Mivel minden egyes ÖSSZEKAPCSOL művelet vagy változatlanul hagyja a csúcsok rangját, vagy egyes csúcsok rangját eggyel növeli, a csúcsok rangja legfeljebb $n - 1$. ■

A 21.6. lemma a rangokra egy gyenge korlátot ad. Valóban, minden csúcsnak a rangja legfeljebb $\lfloor \lg n \rfloor$ (lásd a 21.4-2. gyakorlatot). A 21.6. lemmában szereplő gyengébb korlát azonban számunkra elegendő lesz.

Az időkorlát bizonyítása

Az $O(m\alpha(n))$ időkorlát bizonyítására az amortizált elemzés potenciálmódszerét (lásd a 17.1. alfejezetet) használjuk. Az amortizált elemzésben kényelmesebb, ha feltesszük, hogy az EGYESÍT művelet helyett az ÖSSZEKAPCSOL műveletet használjuk. Azaz, mivel az ÖSSZEKAPCSOL művelet paramétere két gyökércsúcsra mutató pointer, feltesszük, hogy a megfelelő HALMAZT-KERES műveleteket külön-külön végrehajtjuk. A következő lemma azt mutatja, hogy ha még az EGYESÍT műveletek által meghívott HALMAZT-KERES műveleteket is figyelembe vesszük, az aszimptotikus végrehajtási időigény még akkor is változatlan marad.

21.7. lemma. *Tegyük fel, hogy egy m'* HALMAZT-KÉSZÍT, EGYESÍT *és* HALMAZT-KERES *műveletből álló S' sorozatot egy m* HALMAZT-KÉSZÍT, ÖSSZEKAPCSOL *és* HALMAZT-KERES *műveletből álló S sorozatra alakítunk át úgy, hogy mindegyik* EGYESÍT *műveletet két* HALMAZT-KERES *műveletre és az ezeket követő* ÖSSZEKAPCSOL *műveletre cseréljük ki. Ekkor, ha az S sorozat $O(m\,\alpha(n))$ időigény alatt fut le, akkor az S' sorozat végrehajtási időigénye $O(m'\,\alpha(n))$.*

Bizonyítás. Mivel S'-ben minden EGYESÍT műveletnek az S-ben három művelet felel meg, $m' \le m \le 3m'$. Mivel $m = O(m')$, az átalakított S sorozat $O(m\,\alpha(n))$ időkorlátjából következik az eredeti S' sorozat $O(m'\,\alpha(n))$ időkorlátja. ■

Az alfejezet hátralevő részében feltesszük, hogy az m' HALMAZT-KÉSZÍT, EGYESÍT és HALMAZT-KERES műveletből álló kezdeti sorozatot egy m HALMAZT-KÉSZÍT, ÖSSZEKAPCSOL és HALMAZT-KERES műveletből álló sorozatra konvertáltuk. Az $O(m\alpha(n))$ időkorlátot erre a konvertált sorozatra bizonyítjuk be, és a 21.7. lemmát alkalmazzuk az m' műveletből álló eredeti sorozat $O(m'\,\alpha(n))$ futási idejének a bizonyításához.

A potenciálfüggvény

A potenciálfüggvényt arra használjuk, hogy q művelet végrehajtása után a diszjunkt-halmaz erdő minden x csúcsához hozzárendeljünk egy $\phi_q(x)$ potenciált. A csúcsok potenciálját az egész erdőre nézve összeadhatjuk, $\Phi_q = \sum_x \phi_q(x)$), ahol Φ_q jelöli az erdő potenciálját q művelet végrehajtása után. Az első művelet végrehajtása előtt az erdő üres, ezért azt mondhatjuk, hogy $\Phi_0 = 0$. A Φ_q potenciál sohasem lehet negatív.

A $\phi_q(x)$ értéke attól függ, hogy vajon x egy fának a gyökércsúcsa-e a q-adik művelet végrehajtása után. Ha igen, vagy ha $rang[x] = 0$, akkor $\phi_q(x) = \alpha(n) \cdot rang[x]$.

Tegyük fel, hogy a q-adik művelet végrehajtása után x nem gyökércsúcs, és hogy $rang[x] \ge 1$. Az x-re két segédfüggvényt kell megadnunk, miel őtt $\phi_q(x)$-et definiáljuk. Először legyen

$$\text{szint}(x) = \max\{k : rang[p[x]] \ge A_k(rang[x])\}.$$

Tehát szint(x) a legnagyobb olyan szintszám, amelyre az x rangjára alkalmazott A_k értéke nem nagyobb, mint az x szülőjének a rangja.

Azt állítjuk, hogy

$$0 \le \text{szint}(x) < \alpha(n), \tag{21.1}$$

amit a következőkből láthatunk be. Tudjuk, hogy

$$\begin{aligned} rang[p[x]] &\geq rang[x] + 1 && \text{(a 21.4. lemma alapján)} \\ &= A_0(rang[x]) && \text{(az } A_0(j) \text{ definíciója alapján),} \end{aligned}$$

amiből az következik, hogy szint$(x) \geq 0$, és így

$$\begin{aligned} A_{\alpha(n)}(rang[x]) &\geq A_{\alpha(n)}(1) && \text{(mivel } A_k(j) \text{ szigorúan növekszik)} \\ &\geq n && \text{(az } \alpha(n) \text{ definíciója alapján)} \\ &> rang[p[x]] && \text{(a 21.6. lemma alapján),} \end{aligned}$$

amiből következik, hogy szint$(x) < \alpha(n)$. Megjegyezzük, hogy mivel $rang[p[x]]$ az id ő szerint monoton növekszik, ez igaz szint(x)-re is.

A második segédfüggvény a következ ő:

$$\text{iter}(x) = \max\{i : rang[p[x]] \geq A^{(i)}_{\text{szint}(x)}(rang[x])\}.$$

Tehát iter(x) az iteráció legnagyobb olyan száma, ahányszor iterálva az x rangjára alkalmazott $A_{\text{szint}(x)}$-t, az érték még éppen kisebb, mint az x szül őjének a rangja.

Azt állítjuk, hogy

$$1 \leq \text{iter}(x) < rang[x], \tag{21.2}$$

amit a következőkből láthatunk be. Tudjuk, hogy

$$\begin{aligned} rang[p[x]] &\geq A_{\text{szint}(x)}(rang[x]) && \text{(a szint}(x) \text{ definíciója alapján)} \\ &= A^{(1)}_{\text{szint}(x)}(rang[x]) && \text{(a függvény iteráció alapján),} \end{aligned}$$

amiből az következik, hogy iter$(x) \geq 1$, és így

$$\begin{aligned} A^{(rang[x]+1)}_{\text{szint}(x)}(rang[x]) &= A_{\text{szint}(x)+1}(rang[x]) && \text{(az } A_k(j) \text{ definíciója alapján)} \\ &> rang[p[x]] && \text{(a szint}(x) \text{ definíciója alapján),} \end{aligned}$$

amiből következik, hogy iter$(x) \leq rang[x]$. Megjegyezzük, hogy mivel $rang[p[x]]$ az id ő szerint monoton növekszik, azért, hogy az iter(x) csökkenjen, a szint(x)-nek növekednie kell. Addig, amíg a szint(x) változatlan, az iter(x) vagy növekszik, vagy változatlan marad.

Ezekkel a segédfüggvényekkel már definiálni tudjuk az x csúcs potenciálját q művelet végrehajtása után:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot rang[x], & \text{ha } x \text{ gyökércsúcs vagy } rang[x] = 0, \\ (\alpha(n) - \text{szint}(x)) \cdot rang[x] - \text{iter}(x), & \text{ha } x \text{ nem gyökércsúcs és } rang[x] \geq 1. \end{cases}$$

A következő két lemma a csúcs potenciáljának hasznos tulajdonságait adja meg.

21.8. lemma. *Minden x csúcsra és minden q műveletszámra*

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot rang[x].$$

Bizonyítás. Ha x gyökércsúcs vagy $rang[x] = 0$, akkor definíció szerint $\phi_q(x) = \alpha(n) \cdot rang[x]$. Tegyük fel, hogy x nem gyökércsúcs és $rang[x] \geq 1$. $\phi_q(x)$-re egy alsó korlátot a szint(x) és iter(x) maximalizálásával kapunk. A (21.1) korlát szerint szint$(x) \leq \alpha(n) - 1$, és a (21.2) szerint iter$(x) \leq$ rang$[x]$. Így

$$\begin{aligned}\phi_q(x) &\leq (\alpha(n) - (\alpha(n) - 1)) \cdot rang[x] - rang[x] \\ &= rang[x] - rang[x] \\ &= 0.\end{aligned}$$

Hasonlóan, a szint(x) és az iter(x) minimalizálásával a $\phi_q(x)$-re egy felső korlátot kapunk. A (21.1) korlát szerint szint(x) ≥ 0, és a (21.2) korlát alapján iter(x) ≥ 1. Így

$$\begin{aligned}\phi_q(x) &\leq (\alpha(n) - 0) \cdot rang[x] - 1 \\ &= \alpha(n) \cdot rang[x] - 1 \\ &< \alpha(n) \cdot rang[x].\end{aligned}$$

■

A potenciál megváltozása és a műveletek amortizált költsége

Most már meg tudjuk vizsgálni azt, hogyan hatnak a diszjunkt-halmaz műveletek a csúcsok potenciáljára. Azért, hogy a műveletek által okozott potenciálváltozásokat megértsük, meghatározzuk a műveletek amortizált költségigényét.

21.9. lemma. *Legyen x egy nemgyökércsúcs, és tegyük fel, hogy a q-adik művelet egy* Összekapcsol *vagy egy* Halmazt-keres *művelet. Ekkor a q-adik művelet után $\phi_q(x) \leq \phi_{q-1}(x)$. Sőt mi több, ha $rang[x] \geq 1$, és ha a q-adik művelet végrehajtásakor* szint(x) *vagy* iter(x) *megváltozik, akkor $\phi_q(x) \leq \phi_{q-1}(x) - 1$. Azaz x potenciálja nem nő, és ha pozitív rangja van és* szint(x) *vagy* iter(x) *megváltozik, akkor az x potenciálja legalább eggyel csökken.*

Bizonyítás. Mivel x nem gyökércsúcs, a q-adik művelet nem változtatja meg a $rang[x]$-et, és mivel n nem változik az első n Halmazt-készít művelet után, az $\alpha(n)$ is változatlan marad. Így az x potenciáljának képletében ezek a komponensek a q-adik művelet végrehajtása után nem változnak meg. Ha $rang[x] = 0$, akkor $\phi_q(x) = \phi_{q-1}(x) = 0$. Ezek után tegyük fel, hogy $rang[x] \geq 1$.

Megismételjük, hogy szint(x) az idő függvényében monoton növekvő. Ha a q-adik művelet szint(x)-et változatlanul hagyja, akkor az iter(x) vagy növekszik, vagy változatlan marad. Ha mind a szint(x), mind az iter(x) változatlan, akkor $\phi_q(x) = \phi_{q-1}(x)$. Ha szint($x$) változatlan és iter($x$) növekszik, akkor legalább eggyel növekedik, és így $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

Végül, ha a q-adik művelet növeli a szint(x)-et, akkor legalább eggyel növeli, és így a $(\alpha(n) - \text{szint}(x)) \cdot rang[x]$ legalább $rang[x]$-szel csökken. Mivel a szint(x) növekedett, az iter(x)-nek csökkennie kellett, de a (21.2.) korlát miatt a csökkenés legfeljebb $rang[x] - 1$. Ezért az iter(x) megváltozása által okozott potenciálnövekedés kevesebb, mint a szint(x) változása miatti potenciálcsökkenés, és ezért $\phi_q(x) = \phi_{q-1}(x) - 1$. ■

Az utolsó három lemmánk megmutatja, hogy a Halmazt-készít, Összekapcsol és Halmazt-keres művelet mindegyikének az amortizált költsége $O(\alpha(n))$. Hivatkozunk a (17.2.) egyenletre, amelyben láttuk, hogy mindegyik művelet amortizált költsége az aktuális költségének és a művelet által okozott potenciálnövekedésnek az összege.

21.10. lemma. *Mindegyik* Halmazt-készít *művelet amortizált költsége $O(1)$.*

Bizonyítás. Tegyük fel, hogy a q-adik művelet Halmazt-készít(x). Ez a művelet egy nulla rangú x csúcsot hoz létre, melyre $\phi_q(x) = 0$. Semmilyen más rang vagy potenciál nem

változik, és így $\phi_q = \phi_{q-1}$. A HALMAZT-KÉSZÍT művelet aktuális költsége $O(1)$, és ezzel a bizonyítást befejeztük. ■

21.11. lemma. *Mindegyik* ÖSSZEKAPCSOL *művelet amortizált költsége* $O(\alpha(n))$.

Bizonyítás. Tegyük fel, hogy a q-adik művelet ÖSSZEKAPCSOL(x, y). Az ÖSSZEKAPCSOL művelet aktuális költsége $O(1)$. Az általánosság megsértése nélkül, tegyük fel, hogy az ÖSSZEKAPCSOL művelet y-t teszi az x szülőjének.

Az ÖSSZEKAPCSOL által okozott potenciálváltozás meghatározásához megjegyezzük, hogy csak az x és y csúcsok potenciálja, és csak azon csúcsok potenciálja változhat meg, amelyek közvetlenül a művelet előtt y gyerekei voltak. Meg fogjuk mutatni, hogy az ÖSSZEKAPCSOL művelet végrehajtásakor csak az y csúcs potenciálja növekedhet, és hogy a növekedés legfeljebb $\alpha(n)$:

- A 21.9. lemma alapján azoknak a csúcsoknak a potenciálja, amelyek az ÖSSZEKAPCSOL művelet előtt közvetlenül az y gyerekei voltak, az ÖSSZEKAPCSOL művelet hatására nem növekedhet.
- A $\phi_q(x)$ definíciójából látható, hogy mivel x a q-adik művelet előtt egy gyökércsúcs volt, $\phi_{q-1}(x) = \alpha(n) \cdot rang[x]$. Ha $rang[x] = 0$, akkor $\phi_q(x) = \phi_{q-1}(x)$. Egyébként

$$\begin{aligned} \phi_q(x) &= (\alpha(n) - \text{szint}(x)) \cdot rang[x] - \text{iter}(x) \\ &< \alpha(n) \cdot rang[x] \qquad \text{(a (21.1.) és (21.2.) egyenlőtlenségek alapján)} \end{aligned}$$

  Mivel az utolsó érték $\phi_{q-1}(x)$, láthatjuk, hogy az x potenciálja csökken.
- Mivel az ÖSSZEKAPCSOL művelet előtt y gyökércsúcs volt, $\phi_{q-1}(y) = \alpha(n) \cdot rang[y]$. Az ÖSSZEKAPCSOL művelet az y-t gyökércsúcsnak hagyja, és y rangja vagy változatlan marad, vagy az y rangját eggyel növeli. Így $\phi_q(y) = \phi_{q-1}(y)$, vagy $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

Az ÖSSZEKAPCSOL művelet által okozott potenciálnövekedés tehát legfeljebb $\alpha(n)$. Az ÖSSZEKAPCSOL művelet amortizált költsége $O(1) + \alpha(n) = O(\alpha(n))$. ■

21.12. lemma. *Mindegyik* HALMAZT-KERES *művelet amortizált költsége* $O(\alpha(n))$.

Bizonyítás. Tegyük fel, hogy a q-adik művelet HALMAZT-KERES, és hogy a keresési úton s csúcs van. A HALMAZT-KERES művelet aktuális költsége $O(s)$. Megmutatjuk, hogy a HALMAZT-KERES művelet egyetlen csúcsnak sem növeli a potenciálját, és a keresési úton legalább $\max(0, s - (\alpha(n) + 2))$ csúcsnak csökken a potenciálja legalább eggyel.

Egyetlen nemgyökércsúcsnak sem növekszik a potenciálja a 21.9. lemma alapján. Ha x gyökércsúcs, akkor potenciálja $\alpha(n) \cdot rang[x]$, és ez nem változik.

Most megmutatjuk, hogy legalább $\max(0, s-(\alpha(n)+2))$ csúcsnak csökken a potenciálja legalább eggyel. Legyen x a keresési út egy olyan csúcsa, amelyre $rang(x) > 0$, és x-et valahol a keresési úton egy olyan y csúcs követi, amelyik nem gyökércsúcs, és $\text{szint}(y) = \text{szint}(x)$ a HALMAZT-KERES művelet előtt. (Az y nem feltétlenül *közvetlen* követője az x-nek a keresési úton.) A keresési úton legfeljebb $\alpha(n) + 2$ csúcs kivételével a csúcsok kielégítik ezeket az x-re vonatkozó feltételeket. Azok a csúcsok, amelyek nem elégítik ki a feltételeket, a keresési út első csúcsa (ha a rangja 0), az út utolsó csúcsa (azaz a gyökércsúcs), és az út olyan utolsó w csúcsai, amelyekre $\text{szint}(w) = k$, $k = 0, 1, 2, \ldots, \alpha(n) - 1$-re.

Rögzítsünk egy ilyen x csúcsot, és megmutatjuk, hogy x potenciálja legalább eggyel csökken. Legyen $k = \text{szint}(x) = \text{szint}(y)$. A HALMAZT-KERES úttömörítése miatt

$$\begin{aligned}
rang[p[x]] &\geq A_k^{(\text{iter}(x))}(rang[x]) && \text{(az iter}(x)\text{ definíciója alapján),}\\
rang[p[y]] &\geq A_k(rang[y]) && \text{(a szint}(y)\text{ definíciója alapján),}\\
rang[y] &\geq rang[p[x]] && \text{(a 21.5. következmény alapján és mivel}\\
& && \quad y\text{ követi }x\text{-et a keresési úton).}
\end{aligned}$$

Ezeket az egyenlőtlenségeket összefogva és i-vel jelölve az iter(x) úttömörítés előtti értékét, azt kapjuk, hogy

$$\begin{aligned}
rang[p[x]] &\geq A_k(rang[y])\\
&\geq A_k(rang[p[x]] && \text{(mivel } A_k(j) \text{ szigorúan növekszik)}\\
&\geq A_k(A_k^{(\text{iter}(x))}(rang[x]))\\
&= A_k^{(i+1)}(rang[x]).
\end{aligned}$$

Mivel az úttömörítés eredményeképpen az x-nek és az y-nak ugyanaz a szülője lesz, tudjuk, hogy az úttömörítés után $rang[p[x]] = rang[p[y]]$ és azt, hogy az úttömörítés nem csökkenti $rang[p[y]]$-t. Mivel $rang[x]$ nem változik, az úttömörítés után $rang[p[x]] \geq A_k^{(i+1)}(rang[x])$. Így az úttömörítés vagy az iter(x)-et növeli (legalább $i + 1$-gyel), vagy a szint(x)-et növeli (ami akkor fordul elő, ha iter(x) növekedik legalább $rang[x] + 1$-gyel). Mindkét esetben, a 21.9. lemma alapján, azt kapjuk, hogy $\phi_q(x) \leq \phi_{q-1}(x) - 1$. Így tehát x potenciálja legalább eggyel csökken.

A HALMAZT-KERES művelet amortizált költsége az aktuális költség és a potenciál változásának összege. Az aktuális költség $O(s)$, és beláttuk, hogy a teljes potenciál legalább $\max(0, s - (\alpha(n) + 2))$-vel csökken. Ezért az amortizált költség legfeljebb $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, mivel a potenciál összetevőit az $O(s)$-ben rejtett konstanssal tudjuk jellemezni. ■

A fenti lemmákat összekapcsolva a következő tételt kapjuk:

21.13. tétel. *Egy m* HALMAZT-KÉSZÍT, EGYESÍT *és* HALMAZT-KERES *műveletből álló sorozat, amelyben n* HALMAZT-KÉSZÍT *művelet van, diszjunkt-halmaz erdő adatszerkezeten a rang szerinti egyesítést és az úttömörítést használva legrosszabb esetben $O(m\,\alpha(n))$ idő alatt végrehajtható.*

Bizonyítás. Az állítás közvetlenül adódik a 21.7., 21.10., 21.11. és 21.12. lemmákból. ■

Gyakorlatok

21.4-1. Bizonyítsuk be a 21.4. lemmát.

21.4-2. Bizonyítsuk be, hogy minden csúcsnak a rangja legfeljebb $\lfloor \lg n \rfloor$.

21.4-3. A 21.4-2. gyakorlat alapján, hány bit szükséges az x csúcsok $rang(x)$ adatának tárolásához?

21.4-4. A 21.2-3. gyakorlatot felhasználva adjunk egy egyszerű bizonyítást arra, hogy a diszjunkt-halmaz erdőkön a rang szerinti egyesítést használva, de az úttömörítést nem használva, a műveletek végrehajtási időigénye $O(m \lg n)$.

21.4-5. Dante professzor azt állítja, hogy mivel a gyökércsúcsba vezető úton a csúcsok rangja szigorúan növekszik, az úton a csúcsok szintjeinek is monoton növekedniük kell.

Másképpen fogalmazva, ha $rang(x) > 0$ és $p[x]$ nem gyökércsúcs, akkor szint$(x) \leq$ szint$(p[x])$. Helyes a professzor állítása?

21.4-6.$\star$ Tekintsük az $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n+1)\}$ függvényt. Mutassuk meg, hogy $\alpha'(n) \leq 3$ az n minden gyakorlatban használatos értékére. Felhasználva a 21.4-2. gyakorlatot, hogyan módosíthatjuk a potenciálfüggvény argumentumát úgy, hogy bebizonyítsuk, egy m HALMAZT-KÉSZÍT, EGYESÍT és HALMAZT-KERES műveletből álló sorozat, amelyben n HALMAZT-KÉSZÍT művelet van, diszjunkt-halmaz erdő adatszerkezeten a rang szerinti egyesítést és az úttömörítést használva legrosszabb esetben $O(m\,\alpha'(n))$ idő alatt végrehajtható.

Feladatok

21-1. Off-line minimum

Az ***off-line minimum problémában*** egy olyan T dinamikus halmazzal foglalkozunk, amelynek elemei az $\{1, 2, \ldots, n\}$ halmazból valók, és a halmazra a BESZÚR és a MINIMUMOT-KIVÁG műveleteket alkalmazzuk. Legyen adott egy n BESZÚR és m MINIMUMOT-KIVÁG eljáráshívást tartalmazó S sorozat, amelyben mindegyik $\{1, 2, \ldots, n\}$-beli kulcsot csak egyszer szúrjuk be. A feladat az, hogy meghatározzuk, melyik kulcsot adják eredményül a MINIMUMOT-KIVÁG műveletek. Pontosabban, egy *kivágott*$[1..m]$ tömböt kell kitöltenünk, ahol $i = 1, 2, \ldots, m$-re *kivágott*$[i]$ az i-edik MINIMUMOT-KIVÁG hívás által visszaadott kulcs. A probléma „off-line" abban az értelemben, hogy az egész S sorozat feldolgozását elvégezhetjük azel őtt, hogy egy visszaadandó kulcsot is meghatároznánk.

a. Az off-line minimum problémára tekintsük a következ ő példát. A BESZÚR műveleteket számjegyekkel, a MINIMUMOT-KIVÁG műveleteket E betűkkel jelöltük:

$$4, 8, \mathrm{E}, 3, \mathrm{E}, 9, 2, 6, \mathrm{E}, \mathrm{E}, \mathrm{E}, 1, 7, \mathrm{E}, 5.$$

Töltsük ki a *kivágott* tömb elemeit a helyes értékekkel.

Ha erre a problémára egy algoritmust készítünk, akkor bontsuk az S sorozatot homogén részsorozatokra. Legyen S a

$$\mathrm{I}_1, \mathrm{E}, \mathrm{I}_2, \mathrm{E}, \mathrm{I}_3, \ldots, \mathrm{I}_m, \mathrm{E}, \mathrm{I}_{m+1},$$

ahol mindegyik E egy MINIMUMOT-KIVÁG hívást, és mindegyik I_j BESZÚR hívások (esetleg üres) sorozatát jelöli. Mindegyik I_j részsorozatra először helyezzük el a részsorozat által beszúrandó kulcsokat egy K_j halmazba, amelyik üres, ha I_j üres. Ezután hajtsuk végre a következőket.

OFF-LINE-MINIMUM(m, n)

```
1  for i ← 1 to n
2      do határozzuk meg azt a j-t, amelyre i ∈ K_j
3         if j ≠ m + 1
4            then kivágott[j] ← i
5                 legyen l a legkisebb j-nél nagyobb olyan érték, amelyre a K_l halmaz létezik
6                 K_l ← K_j ∪ K_l, a K_j törlése
7  return kivágott
```

b. Bizonyítsuk be, hogy az OFF-LINE-MINIMUM által eredményül adott *kivágott* tömb helyes.

c. Írjuk le, hogyan lehet az OFF-LINE-MINIMUM algoritmust diszjunkt-halmaz adatszerkezetekre hatékonyan megvalósítani. Adjunk a megvalósításunk legrosszabb futási idő igényére egy pontos korlátot.

21-2. A mélység meghatározása

A ***mélység meghatározásának problémájában*** gyökeres fák $\mathcal{F} = \{T_i\}$ erdejére a következő műveleteket definiáljuk:

FÁT-KÉSZÍT(v) egy olyan fát készít, amelynek csak a v csúcsa van.

MÉLYSÉGET-KERES(v) a v-t tartalmazó fában meghatározza a v csúcs mélységét.

BEÜLTET(r, v) létrehoz egy r csúcsot, ami egy olyan fának lesz a gyökércsúcsa, amelyik egy másik fában levő v csúcsnak lesz a gyereke. A v-nek nem feltétlenül kell gyökércsúcsnak lennie.

a. Tegyük fel, hogy a diszjunkt-halmaz erdőhöz hasonló faszerkezeteket használunk: $p[v]$ a v csúcs szülője, kivéve, ha v a gyökércsúcs, ekkor $p[v] = v$. Ha a BEÜLTET(r, v)-t úgy valósítjuk meg, hogy $p[r] \leftarrow v$, a MÉLYSÉGET-KERES(y)-t úgy, hogy a gyökércsúcsba vezető keresési utat követi, és visszaadja a v-t nem számítva azon csúcsok számát, amelyeken keresztülment, akkor mutassák meg, hogy az m FÁT-KÉSZÍT, MÉLYSÉGET-KERES és BEÜLTET műveletből álló sorozat legrosszabb végrehajtási ideje $\Theta(m^2)$.

A rang szerinti egyesítés és az úttömörítés heurisztikát használva csökkenteni tudjuk a legrosszabb futási időket. Az $\mathcal{S} = \{S_i\}$ diszjunkt-halmaz erdőt használjuk, ahol minden egyes S_i halmaz (ami maga is egy fa,) megfelel az $\mathcal{F}$ erdő T_i fájának. Egy S_i-beli faszerkezet azonban nem feltétlenül felel meg a T_i szerkezetének. Ez azt jelenti, hogy az S_i megvalósítása nem tartalmazza a konkrét szülő-gyerek kapcsolatot, és így egyáltalán nem tudjuk meghatározni a csúcsok mélységét.

Az ötlet az, hogy minden v csúcsra meghatározunk egy $d[v]$ „pszeudotávolságot”, amit így definiálunk: a v csúcs T_i-beli mélysége legyen a v csúcsból a v-t tartalmazó S_i halmaz gyökércsúcsába vezető út csúcsai pszeudotávolságainak összege. Azaz, ha a v-ből az S_i gyökércsúcsába vezető út csúcsai $v_0, v_1, \ldots, v_k$, ahol $v_0 = v$ és v_k az S_i gyökere, akkor a v mélysége T_i-ben $\sum_{j=0}^{k} d[v_j]$.

b. Adjuk meg a FÁT-KÉSZÍT eljárás megvalósítását.

c. Mutassuk meg, hogyan kell módosítani HALMAZT-KERES eljárást a MÉLYSÉGET-KERES megvalósításához. A megvalósítás hajtson végre úttömörítést, és a futási ideje legyen lineáris a keresési út hosszának függvényében. Biztosítsuk azt, hogy a megvalósítás a pszeudotávolságokat jól kezelje.

d. Mutassuk meg, hogy hogyan kell módosítani az EGYESÍT és az ÖSSZEKAPCSOL eljárásokat a BEÜLTET(r, v) megvalósításához, ahol a BEÜLTET(r, v) összekapcsolja az r-et és a v-t tartalmazó halmazokat. Biztosítsuk azt, hogy a megvalósítás a pszeudotávolságokat jól kezeli. Megjegyezzük, hogy egy S_i halmaz gyökere nem feltétlenül a neki megfelelő T_i fa gyökércsúcsa.

e. Adjunk meg egy pontosabb futási időkorlátot a legrosszabb esetre, ha egy olyan m FÁT-KÉSZÍT, MÉLYSÉGET-KERES és BEÜLTET műveletből álló sorozatot hajtunk végre, amelyben n FÁT-KÉSZÍT művelet van.

21-3. Tarjan off-line legközelebbi közös ősök algoritmusa

A gyökeres T fában az u és a v csúcsok ***legközelebbi közös őse*** w, ha w a legnagyobb mélységű olyan csúcs, amelyre w mind az u-nak, mind a v-nek őse. Az ***off-line legközelebbi közös ősök problémája*** a legyen adott egy gyökeres T fa, és a T-beli csúcsokat tartalmazó rendezetlen párok egy $P = \{\{u, v\}\}$ tetszőleges halmaza. A feladat az, hogy a P minden elemére határozzuk meg a legközelebbi közös őst.

Az off-line legközelebbi közös ősök problémának a megoldására a következő eljárás szolgál. Az első LKŐ(*gyökér*[T]) eljáráshívás a T egy fabejárását hajtja végre. Feltesszük, hogy a bejárás előtt a fa minden csúcsa FEHÉR színűre van színezve.

```
LKŐ(u)
 1  HALMAZT-KÉSZÍT(u)
 2  ős[HALMAZT-KERES(u)] ← u
 3  for az u minden T-beli v gyerekére
 4      do LKŐ(v)
 5         EGYESÍT(u, v)
 6         ős[HALMAZT-KERES(u)] ← u
 7  szín[u] ← FEKETE
 8  for minden olyan v csúcsra, amelyre {u, v} ∈ P
 9      do if szín[v] = FEKETE
10            then kiír „Az” u „és a” v „legközelebbi közös ős csúcsa”
                   ős[HALMAZT-KERES(v)]
```

a. Bizonyítsuk be, hogy a 10. sor mindegyik $\{u, v\} \in P$ párra pontosan egyszer hajtódik végre.

b. Bizonyítsuk be, hogy az LKŐ(u) meghívása időpillanatában a diszjunkt-halmaz adatszerkezetben levő halmazok száma megegyezik az u csúcs T-beli mélységével.

c. Bizonyítsuk be, hogy az LKŐ eljárás minden $\{u, v\} \in P$ párra helyesen írja ki az u és a v legközelebbi közös ősét.

d. Elemezzük az LKŐ eljárás futási idejét, feltételezve, hogy a 21.3. alfejezetben megadott diszjunkt-halmaz adatszerkezet megvalósítását használjuk.

Megjegyzések a fejezethez

A diszjunkt-halmaz adatszerkezetekre vonatkozó sok fontos eredmény, legalábbis részben, R. E. Tarjantól származik. Összesítési elemzést használva Tarjan [290, 292] adta meg az első pontos felső korlátot az Ackermann-függvény nagyon lassan növekvő $\hat{\alpha}(m, n)$ inverzének felhasználával. (A 21.4. alfejezetben megadott $A_k(j)$ függvény hasonló az Ackermann-függvényhez, és az $\alpha(n)$ hasonló az inverzéhez. Mind az $\alpha(n)$, mind az $\hat{\alpha}(m, n)$ legfeljebb 4 az összes értelmes m és n értékekre. Egy $O(m \lg^* n)$ felső korlátot Hopcroft és Ullman [5, 155] korábban már bizonyított. A 21.4. alfejezetben alkalmazott módszer Tarjan egy későbbi elemzése [294] alapján készült, ami viszont Kozen elemzésén [193] alapult. Harfst és Reingold [139] adta meg Tarján korábbi korlátjának potenciálalapú változatát.

Tarjan és van Leeuwen [295] vizsgálta az úttömörítés heurisztika különböző változatait, beleértve az „egymenetes módszereket” is, amelyek a hatékonyságban gyakran jobb tényezőt adnak, mint a kétmenetes módszerek. Míg Tarjan korábbi elemzése az úttömörítő heurisztikát használja, Tarjan és Leeuwen elemzése összesítő. Harfst és Reingold [139] később megmutatta, hogyan kell a potenciálfüggvényt egy kicsit módosítani ahhoz, hogy alkalmazható legyen az úttömörítő elemzés egymenetes változatához is. Gabow és Tarjan [103] megmutatta, hogy bizonyos alkalmazásokban a diszjunkt-halmaz műveletek végrehajtási időigénye $O(m)$ is lehet.

Tarjan [291] megmutatta, hogy az $\Omega(m\,\hat{\alpha}(m,n))$ alsó korlát szükséges a diszjunkt-halmaz adatszerkezetek azon műveleteihez, amelyek bizonyos technikai feltételeket teljesítenek. Ezt az alsó korlátot később Fredman és Saks [97] általánosította, megmutatva, hogy a legrosszabb esetben a memóriaméret $\Omega(m\,\hat{\alpha}(m,n))$ számú $(\lg n)$-bit hosszúságú szó is lehet.

VI. GRÁFALGORITMUSOK

Bevezetés

A gráfok rendkívül gyakran használt struktúrák, és a gráfokkal foglalkozó algoritmusok alapvető szerepet játszanak a számítástudományban. Számos érdekes probléma fogalmazható és oldható meg gráfok segítségével. Ebben a részben néhány jelentősebb problémát vizsgálunk.

A 22. fejezet elején gráfok számítógépes ábrázolásával foglalkozunk, majd szélességi, illetve mélységi keresésen alapuló algoritmusokat mutatunk be. A mélységi keresés két alkalmazását adjuk meg: irányított, körmentes gráf topologikus rendezését, és irányított gráf erősen összefüggő komponenseinek meghatározását.

A 23. fejezetben bemutatjuk, hogyan számíthatjuk ki egy gráf minimális súlyú feszítőfáját. Ezt a fát úgy definiáljuk, mint a gráf csúcsainak legkisebb összsúlyú összekötését, ha minden élhez hozzárendeltünk egy súlyt. A minimális feszítőfát előállító algoritmusok jó példák mohó algoritmusokra (lásd 16. fejezet).

A 24. és 25. fejezetekben a gráf csúcsait összekötő legrövidebb utak meghatározásának problémájával foglalkozunk. Ekkor minden élnek adott a hossza, illetve „súlya". A 24. fejezet az egy csúcsból az összes többi csúcsba vezető legrövidebb utak előállításáról, a 25. pedig az összes csúcspár közötti legrövidebb utak kiszámításáról szól.

Végül, a 26. fejezet a maximális folyamok problémáját tárgyalja. Ekkor a feladat a maximális folyam meghatározása egy hálózatban (irányított gráfban), ha adott a kiinduló csúcs (forrás), a célcsúcs (nyelő) és a folyam maximális nagysága az egyes irányított élek mentén. Ez az általános probléma számos alakban felbukkanhat, és ha ismert egy jó algoritmus a maximális folyam meghatározására, akkor azt felhasználva sok hasonló problémát hatékonyan oldhatunk meg.

A gráfalgoritmusok futási idejét adott $G = (V, E)$ gráf esetén a csúcsok $|V|$ és az élek $|E|$ számával fejezzük ki. Vagyis ebben az esetben nem egy, hanem két paraméter határozza meg az időt. A futási idők megadásakor ezeket a paramétereket a szokásos módon tüntetjük fel, azaz az aszimptotikus jelölésekben (mint O vagy Θ), és *csak* ezekben, V jelentése $|V|$, és E $|E|$-nek felel meg. Például azt mondjuk, hogy „az algoritmus futási ideje $O(VE)$", ami azt jelenti, hogy a futási idő $O(|V||E|)$. Így ezek a képletek könnyebben áttekinthetőek, és félreértés sem származhat ebből.

Az algoritmusok leírásakor egy másik jelölési konvenciót is alkalmazunk. A G gráf csúcshalmazára $V[G]$-vel, élhalmazára pedig $E[G]$-vel hivatkozunk. Így az algoritmusok a csúcshalmazt és az élhalmazt a gráf attribútumaiként kezelik.

22. Elemi gráfalgoritmusok

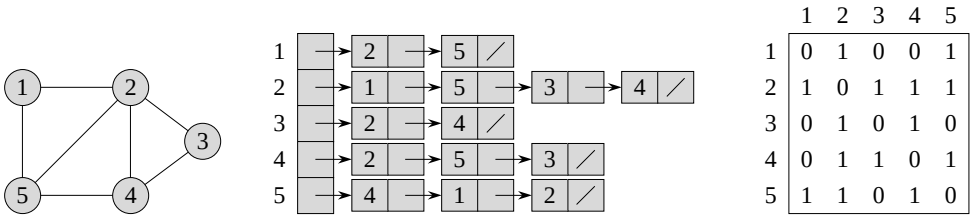
Ebben a fejezetben gráfok ábrázolási és bejárási módszereivel foglalkozunk. A gráfbejárás során módszeresen követjük a gráf éleit úgy, hogy érintsük a gráf összes csúcsát. Gráfbejáró algoritmus segítségével sokat megtudhatunk egy gráf szerkezetéről. Sok algoritmus gráfbejárással kezdődik, hogy felderítse a bemeneti gráf szerkezetét. Más algoritmusok az egyszerű gráfkereső algoritmusok továbbfejlesztéseként állnak elő. Ezért a gráfbejárási módszerek alapvetőek a gráfalgoritmusok szempontjából.

A 22.1. alfejezetben a két leggyakoribb ábrázolási módot mutatjuk be: a szomszédsági listákat és a szomszédsági mátrixokat. A 22.2. alfejezetben leírunk egy egyszerű gráfbejáró algoritmust, a szélességi keresést, és módszert adunk a szélességi fa meghatározására. A 22.3. alfejezetben bemutatjuk a mélységi keresést, és megvizsgáljuk, hogy ennek során milyen sorrendben érintjük a csúcsokat. A mélységi keresés első alkalmazását, egy körmentes, irányított gráf topologikus rendezését tartalmazza a 22.4. alfejezet. Végül a 22.5. alfejezetben a mélységi keresés egy másik felhasználását adjuk meg, amelyben egy irányított gráf erősen összefüggő komponenseit határozzuk meg.

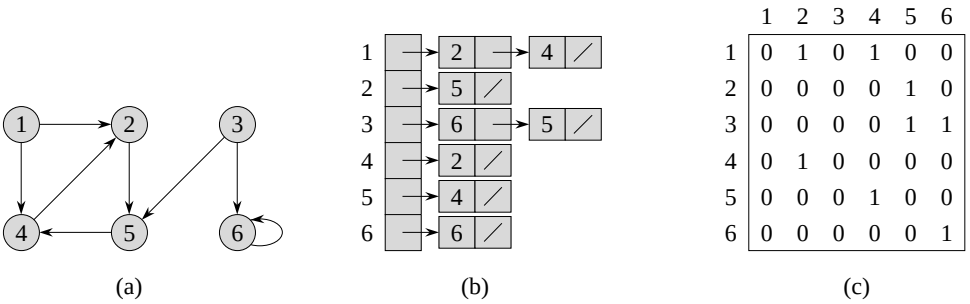
22.1. Gráfok ábrázolási módjai

Két módszert szokás használni egy $G = (V, E)$ gráf ábrázolására: az egyikben szomszédsági listákkal, a másikban szomszédsági mátrixszal adjuk meg a gráfot. Rendszerint a szomszédsági listákon alapuló ábrázolást választják, mert ezzel ***ritka*** gráfok tömören ábrázolhatók. Egy gráfot ritkának nevezünk, ha $|E|$ sokkal kisebb, mint $|V|^2$. A könyvben található algoritmusok többségében feltesszük, hogy a bemeneti gráfot szomszédsági listákkal ábrázoljuk. Ugyanakkor a csúcsmátrixos ábrázolás előnyösebb lehet ***sűrű*** gráfok – $|E|$ megközelíti $|V|^2$-et – esetén, vagy ha gyorsan kell eldönteni, hogy két csúcsot összeköt-e él. Például, a 25. fejezetben található legrövidebb utakat meghatározó algoritmusok közül kettőben csúcsmátrixszal ábrázoljuk a bemeneti gráfot.

$G = (V, E)$ gráf ***szomszédsági listás ábrázolása*** során egy Adj tömböt használunk. Ez $|V|$ darab listából áll, és az Adj tömbben minden csúcshoz egy lista tartozik. Minden $u \in V$ csúcs esetén az $Adj[u]$ szomszédsági lista tartalmazza az összes olyan v csúcsot, amelyre létezik $(u, v) \in E$ él. Azaz: $Adj[u]$ elemei az u csúcs G-beli szomszédjai. (Sokszor nem csúcsokat, hanem megfelelő mutatókat tartalmaz a lista.) A szomszédsági listákban a csúcsok



22.1. ábra. Irányítatlan gráf kétféle ábrázolása. **(a)** Egy G irányítatlan gráf 5 csúccsal és 7 éllel. **(b)** G ábrázolása szomszédsági listákkal. **(c)** G ábrázolása csúcsmátrixszal.



22.2. ábra. Irányított gráf kétféle ábrázolása. **(a)** Egy G irányított gráf 6 csúccsal és 8 éllel. **(b)** G ábrázolása szomszédsági listákkal. **(c)** G ábrázolása csúcsmátrixszal.

sorrendje rendszerint tetszőleges. A 22.1. ábra (b) része az (a) részen található irányítatlan gráf szomszédsági listás ábrázolását szemlélteti. Hasonlóan, a 22.2. ábra (b) része tartalmazza az (a) részen látható irányított gráf szomszédsági listás ábrázolását.

Ha G irányított gráf, akkor a szomszédsági listák hosszainak összege $|E|$, hiszen egy (u, v) élt úgy ábrázolunk, hogy v-t felvesszük az $Adj[u]$ listába. Ha G irányítatlan gráf, akkor az összeg $2|E|$, mert (u, v) irányítatlan él ábrázolása során u-t betesszük v szomszédsági listájába, és fordítva. Akár irányított, akár irányítatlan a gráf, a szomszédsági listás ábrázolás azzal a kedvező tulajdonsággal rendelkezik, hogy az ábrázoláshoz szükséges tárterület mérete $\Theta(V + E)$.

Egy gráfot ***súlyozott gráfnak*** nevezünk, ha minden élhez ***súlyt*** rendelünk, amit rendszerint egy $w : E \to \mathbf{R}$ ***súlyfüggvény*** segítségével adunk meg. A szomszédsági listákat könnyen módosíthatjuk úgy, hogy azokkal súlyozott gráfokat ábrázolhassunk. Például, legyen $G = (V, E)$ súlyozott gráf w súlyfüggvénnyel. Ekkor az $(u, v) \in E$ él $w(u, v)$ súlyát egyszerűen a v csúcs mellett tároljuk u szomszédsági listájában. A szomszédsági listás ábrázolás könnyen alkalmassá tehető sok gráfváltozat reprezentálására.

A szomszédsági listás ábrázolás hátránya, hogy nehéz eldönteni, szerepel-e egy (u, v) él a gráfban, hiszen ehhez az $Adj[u]$ szomszédsági listában kell v-t keresni. Ez a hátrány kiküszöbölhető csúcsmátrix használatával, ez azonban aszimptotikusan növeli a szükséges tárterület méretét. (Lásd 22.1-8. gyakorlatot, amelyben a szomszédsági lista olyan változatait vázoljuk, amelyek az élek gyorsabb elérését teszik lehetővé.)

Ha egy $G = (V, E)$ gráfot ***szomszédsági mátrixszal*** (vagy más néven ***csúcsmátrixszal***) ábrázolunk, feltesszük, hogy a csúcsokat tetszőleges módon megszámozzuk az $1, 2, \ldots, |V|$

értékekkel. A G ábrázolásához használt $A = (a_{ij})$ csúcsmátrix mérete $|V| \times |V|$, és

$$a_{ij} = \begin{cases} 1, & \text{ha } (i, j) \in E, \\ 0, & \text{különben.} \end{cases}$$

A 22.1. ábra, illetve a 22.2. ábra (c) része tartalmazza az (a) részeken látható irányított, illetve irányítatlan gráfokhoz tartozó csúcsmátrixokat. A csúcsmátrix $\Theta(V^2)$ tárterületet foglal le, függetlenül a gráf éleinek számától.

Látható, hogy a 22.1. ábra (c) részén bemutatott csúcsmátrix szimmetrikus a főátlójára. Az $A = (a_{ij})$ mátrix ***transzponáltja*** az $A^{\mathrm{T}} = (a_{ij}^{\mathrm{T}})$ mátrix, ha $a_{ij}^{\mathrm{T}} = a_{ji}$. Irányítatlan gráf esetén (u, v) és (v, u) ugyanaz az él, így a gráfhoz tartozó A csúcsmátrix megegyezik önmaga transzponáltjával, azaz $A = A^{\mathrm{T}}$. Gyakran kifizetődő a csúcsmátrixból csak a főátlóban és az efölött szereplő elemeket tárolni, ezzel majdnem felére csökkenthetjük az ábrázoláshoz szükséges tárterület méretét.

A szomszédsági listás ábrázoláshoz hasonlóan csúcsmátrixokkal is reprezentálhatunk súlyozott gráfokat. Ha $G = (V, E)$ súlyozott gráf w súlyfüggvénnyel, akkor az $(u, v) \in E$ él $w(u, v)$ súlyát a csúcsmátrix u sorában és v oszlopában tároljuk. Nem létező él esetén a mátrix megfelelő elemét NIL-nek választhatjuk, noha sokszor célszerű ehelyett 0 vagy ∞ értékeket használni.

A szomszédsági listák együttesen aszimptotikusan kevesebb tárterületet igényelnek, mint a csúcsmátrix, azonban a használat során hatékonyságban ugyanennyivel elmaradnak attól, így ha a gráf mérete nem túl nagy, akkor kedvezőbb a hatékonyabb és egyszerűbb csúcsmátrixos ábrázolást használni. Ha a gráf nem súlyozott, akkor a csúcsmátrixos ábrázolás tovább javítható. Ebben az esetben a mátrix elemei lehetnek bitek, így jelentősen csökkenthetjük a szükséges tárterület méretét.

Gyakorlatok

22.1-1. Tegyük fel, hogy egy irányított gráfot szomszédsági listákkal ábrázolunk. Mennyi idő alatt határozható meg az összes csúcs kimeneti fokszáma? Mennyi idő alatt számíthatók ki a bemeneti fokszámok?

22.1-2. Tegyük fel, hogy adott egy 7 csúcsú teljes bináris fa szomszédsági listás ábrázolása. Adjuk meg az ekvivalens csúcsmátrixot. Tegyük fel, hogy a csúcsok 1-től 7-ig számozottak, mint egy bináris kupac esetén.

22.1-3. A $G = (V, E)$ irányított gráf ***transzponáltja*** a $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$ gráf, ahol $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$. Azaz, G^{T}-t úgy kapjuk G-ből, hogy minden élt megfordítunk. Adjunk hatékony algoritmust G^{T} meghatározására, G szomszédsági listás, illetve csúcsmátrixos ábrázolása esetén. Elemezzük az algoritmus futási idejét.

22.1-4. Adjunk $O(V + E)$ idejű algoritmust egy szomszédsági listákkal ábrázolt $G = (V, E)$ párhuzamos éleket is tartalmazó gráffal „ekvivalens” irányítatlan $G' = (V, E')$ gráf szomszédsági listás ábrázolásának meghatározására, ahol E'-ben egy éllel helyettesítjük az E-beli két csúcsot összekötő párhuzamos éleket, és a hurokéleket elhagyjuk.

22.1-5. $G = (V, E)$ irányított gráf ***négyzete*** a $G^2 = (V, E^2)$ gráf, ahol $(u, w) \in E^2$ akkor és csak akkor, ha valamely $v \in V$ csúcsra $(u, v) \in E$ és $(v, w) \in E$. Azaz G^2-ben pontosan akkor vezet él u-ból v-be, ha G-ben van pontosan két élből álló út u-ból v-be. Adjunk hatékony algoritmust G^2 meghatározására, G szomszédsági listás, illetve csúcsmátrixos ábrázolása esetén. Elemezzük az algoritmus futási idejét.

22.1-6. Csúcsmátrixos ábrázolás esetén a legtöbb gráfalgoritmus futási ideje $\Omega(V^2)$, azonban van néhány kivétel. Egy irányított gráf csúcsa ***nyelő,*** ha bemeneti foka $|V|-1$ és kimeneti foka 0. Mutassuk meg, eldönthető $O(V)$ idő alatt, hogy egy csúcsmátrixszal ábrázolt irányított gráfban van-e nyelő csúcs.

22.1-7. $G = (V, E)$ irányított gráf ***illeszkedési mátrixa*** a $|V| \times |E|$ méretű $B = (b_{ij})$ mátrix, ahol

$$b_{ij} = \begin{cases} -1, & \text{ha } j \text{ él kivezet } i \text{ csúcsból,} \\ 1, & \text{ha } j \text{ él } i \text{ csúcsba vezet,} \\ 0, & \text{különben.} \end{cases}$$

Mondjuk meg, mit adnak meg a BB^{T} mátrixszorzat elemei, ahol B^{T} B transzponáltja.

22.1-8. Tegyük fel, hogy az $Adj[u]$ tömb minden eleme láncolt lista helyett egy hasító táblázatot tartalmaz, ahol a táblázat elemei azok a v csúcsok, amelyekre $(u, v) \in E$. Mennyi idő alatt lehet várhatóan eldönteni, hogy egy él szerepel-e a gráfban, ha az élek előfordulási valószínűsége egyenletes? Milyen hátrányai vannak ennek az ábrázolásnak? Javasoljunk olyan adatszerkezetet az éllisták ábrázolásához, amely kiküszöböli ezeket a problémákat. Vizsgáljuk meg, hogy vannak-e ennek az ábrázolásnak hátrányai a hasító táblázattal szemben.

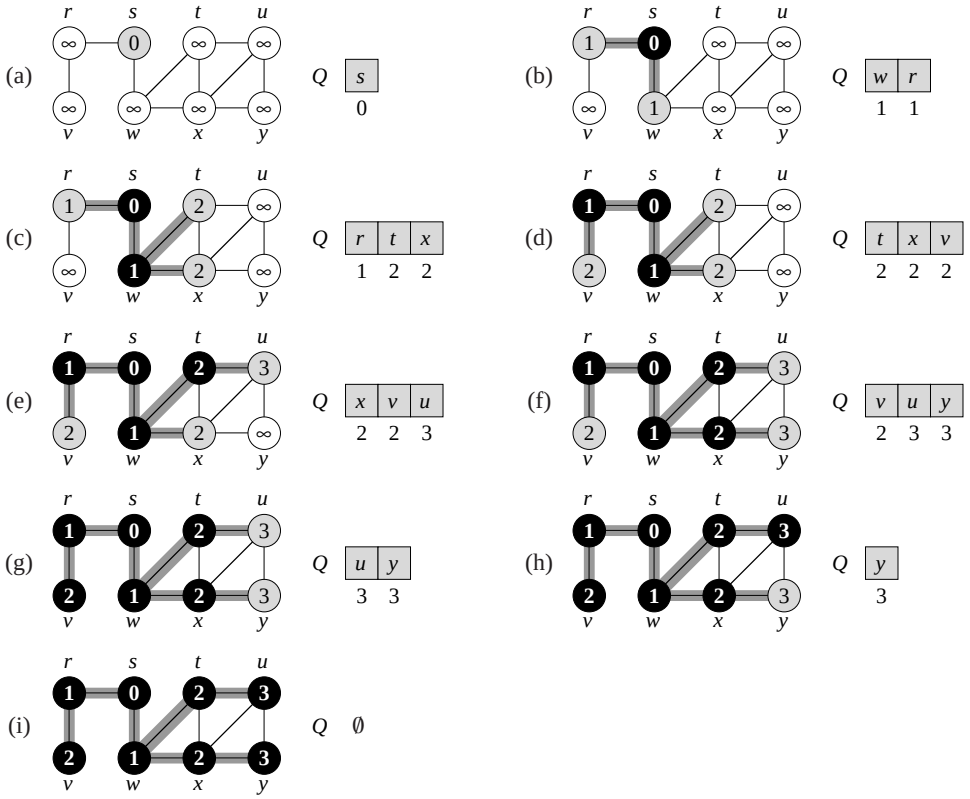
22.2. Szélességi keresés

A ***szélességi keresés*** az egyik legegyszerűbb gráfbejáró algoritmus, és ezen alapul sok fontos gráfalgoritmus. Prim minimális feszítőfákra adott algoritmusa (23.2. alfejezet) és Dijkstra legrövidebb utat meghatározó algoritmusa (24.3. alfejezet) a szélességi kereséshez hasonló gondolatmenetet használ.

Adott $G = (V, E)$ irányított vagy irányítatlan gráf és egy kitüntetett s ***kezdő*** csúcs esetén a szélességi keresés módszeresen megvizsgálja G éleit , és így „rátalál" minden s-ből elérhető csúcsra. Kiszámítja az elérhető csúcsok távolságát (legkevesebb él) s-től. Létrehoz egy s gyökerű „szélességi fát", amely tartalmazza az összes elérhető csúcsot. Bármely s-ből elérhető v csúcsra, A szélességi fában s-ből v-be vezető út a „legrövidebb" s-ből v-be vezető útnak felel meg G-ben – bármely s-ből elérhető v csúcsra. Legrövidebb útnak most a legkevesebb élből álló utat nevezzük. Az algoritmus egyaránt alkalmazható irányított vagy irányítatlan gráfok esetén.

A szélességi keresés elnevezés onnan ered, hogy az algoritmus a már elért és a még felfedezetlen csúcsok közötti határvonalat egyenletesen terjeszti ki a határ teljes széltében. Az algoritmus eljut az összes olyan csúcsba, amely s-től k távolságra van, mielőtt egy $k + 1$ távolságra levő csúcsot elérne.

Az algoritmus a bejárás pillanatnyi állapotát a csúcsok fehér, szürke, illetve fekete színezésével tartja számon. Kezdetben minden csúcs fehér, és később szürkére, majd feketére változhat. Egy csúcs ***elértté*** válik, amikor először rátalálunk a keresés során, és ezután a színe nem lehet fehér. Így a szürke és fekete csúcsok már elért csúcsok, de a szélességi keresés megkülönbözteti ezeket is, hogy a keresés jellege szélességi maradjon. Ha $(u, v) \in E$ és u fekete, akkor v fekete vagy szürke lehet; azaz, egy fekete csúcs összes szomszédja elért csúcs. A szürke csúcsoknak lehetnek fehér szomszédjaik; ezek alkotják az elért és még felfedezetlen csúcsok közötti határt.



22.3. ábra. SzK működésének szemléltetése egy irányítatlan gráfon. Az algoritmus által előállított aktuális fa éleit megvastagítottuk az egyes részeken. A csúcsokon belül tüntettük fel a d távolságértékeket. A 10–18. sorokban szereplő **while** ciklus magjának megkezdésekor érvényes Q sor a gráf mellett látható. A sorban szereplő csúcsok távolságait a csúcs alá írtuk.

A szélességi keresés létrehoz egy szélességi fát, amely kezdetben csak a gyökeret tartalmazza. A gyökér az s kezdő csúcs. Ha egy fehér v csúcsot elérünk egy már elért u csúcshoz tartozó szomszédsági lista vizsgálata során, akkor a fát kiegészítjük a v csúccsal és az (u, v) éllel. Azt mondjuk, hogy a szélességi fában u a v csúcs ***előd*je** vagy ***szülője.*** Egy csúcsot legfeljebb egyszer érhetünk el, így legfeljebb egy szülője lehet. Az ős és leszármazott relációkat, a szokásos módon, az s gyökérhez viszonyítva definiáljuk: ha u az s-ből v-be vezető úton helyezkedik el a fában, akkor u a v őse és v az u leszármazottja.

A következő SzK szélességi kereső eljárásban feltesszük, hogy a $G = (V, E)$ gráfot szomszédsági listákkal ábrázoljuk. Az algoritmus a gráf csúcsainak több tulajdonságát is meghatározza. Egy $u \in V$ csúcs színét a $szín[u]$ érték adja meg, u elődjét pedig $\pi[u]$ változóban tároljuk. Ha u-nak nincs elődje (például $u = s$, vagy még nem értük el u-t), akkor $\pi[u] = \text{NIL}$. $d[u]$ tartalmazza u távolságát az s kezdő csúcstól. Az algoritmus a Q sor (lásd 10.1. alfejezet) segítségével kezeli a szürke csúcsokat.

```
SzK(G, s)
 1  for minden u ∈ V[G] − {s} csúcsra
 2      do szín[u] ← FEHÉR
 3         d[u] ← ∞
 4         π[u] ← NIL
 5  szín[s] ← SZÜRKE
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  SORBA(Q, s)
10  while Q ≠ ∅
11        do u ← SORBÓL(Q)
12           for minden v ∈ Adj[u]-re
13               do if szín[v] = FEHÉR
14                     then szín[v] ← SZÜRKE
15                          d[v] ← d[u] + 1
16                          π[v] ← u
17                          SORBA(Q, v)
18           szín[u] ← FEKETE
```

A 22.3. ábra szemlélteti az SzK algoritmus működését.

Az SzK algoritmus első 4 sora minden s-től különböző u csúcs színét fehérre, $d[u]$ távolságát végtelenre, és szülőjét NIL-re állítja. Az 5. sorban a kiinduló s csúcs színét szürkére változtatjuk, mert ezt elértnek tekintjük az eljárás kezdetekor. A 6. sorban $d[s]$ értékét nullázzuk, és a 7. sorban a kezdő csúcs előd jét NIL-nek választjuk. A 8–9. sorban a Q sort inicializáljuk úgy, hogy abban csak s legyen.

A 10–18. sorok tartalmazzák a program fő ciklusát. A ciklus akkor fejeződik be, ha elfogynak a szürke csúcsok, azaz addig fut, amíg van olyan elért csúcs, amelynek szomszédsági listáját még nem dolgoztuk fel teljesen. Ennek a **while** ciklusnak az invariánsa:

A 10. sor feltételének vizsgálatakor Q elemei a szürke csúcsok.

Ezt az invariánst nem használjuk az algoritmus helyességének bizonyításához, de ennek ellenére látható, hogy fennáll az első iteráció végrehajtása előtt, és minden egyes iteráció fenntartja az invariánst. A ciklus megkezdése előtt az s kezdő csúcs az egyetlen szürke csúcs, és csak ez szerepel Q-ban. A 11. sorban kiválasztjuk a Q sorból az első szürke csúcsot, u-t, és el is távolítjuk a sorból. A 12–17. sor **for** ciklusa megvizsgálja az u szomszédsági listájában található összes v csúcsot. Ha v fehér, azaz még nem értük el, akkor az algoritmus eléri a 14–17. sorokban. Először átszínezi a csúcsot szürkére, majd a $d[v]$ távolságot állítja $d[u] + 1$-re, aztán feljegyzi, hogy u a csúcs szülője, végül a Q sor végéhez fűzi a csúcsot. Ha u szomszédsági listájában szereplő összes csúcsot megvizsgáltuk, akkor u színét feketére változtatjuk (18. sor). A ciklusinvariáns ezután is fennáll, mert ha egy csúcsot szürkére színezünk (14. sor), akkor a sorba is betesszük (17. sor), illetve ha egy csúcsot kiveszünk a sorból (11. sor), akkor feketére színezzük (18. sor).

A szélességi keresés eredménye függhet attól, hogy az egyes csúcsok szomszédjait milyen sorrendben vesszük figyelembe a 12. sorban. Ez azt jelenti, hogy a szélességi fa változhat, azonban az algoritmus által meghatározott d távolságértékek ugyanazok lesznek minden esetben. (Lásd 22.2-4. gyakorlat.)

Elemzés

A szélességi keresés különféle tulajdonságainak bizonyítása előtt egy egyszerűbb feladattal foglalkozunk, amelyben az algoritmus futási idejét vizsgáljuk meg egy $G = (V, E)$ bemeneti gráf esetén. Ehhez a 17.1. alfejezetben megismert aggregációs elemzést használjuk. A kezdeti értékadások után egyetlen csúcs színét sem állítjuk fehérre, ezért a 13. sor feltétele miatt bármely csúcsot legfeljebb egyszer tehetünk a sorba, és így legfeljebb egyszer vehetjük abból ki. A behelyezés és a kivétel műveletek ideje $O(1)$, így a sorműveletek összideje $O(V)$. A csúcsokhoz tartozó szomszédsági listákat legfeljebb egyszer vizsgáljuk, mert erre csak akkor kerül sor, amikor a csúcsot kivesszük a sorból. A szomszédsági listák együttes hossza $\Theta(E)$, ezért összesen legfeljebb $O(E)$ időt fordítunk a szomszédsági listák vizsgálatára. A kezdeti értékadásokhoz $O(V)$ időt használunk, tehát SzK teljes futási ideje $O(V + E)$. Azt kaptuk, hogy a szélességi keresés futási ideje a szomszédsági listás ábrázolás méretének lineáris függvénye.

Legrövidebb utak

Az alfejezet elején azt állítottuk, hogy a szélességi keresés a $G = (V, E)$ gráfban kiszámítja az adott $s \in V$ kezdő csúcsból elérhető összes csúcs s-től mért távolságát. A v csúcs ***legrövidebb út távolsága*** s-től legyen $\delta(s, v)$, amelyet az s-ből v-be vezető egyes utak élszámainak minimumaként definiálunk, ha van ilyen út, és ∞, ha nem vezet út s-ből v-be. Egy $\delta(s, v)$ hosszúságú s-ből v-be vezető utat s és v közötti ***legrövidebb útnak***[1] nevezünk. Mielőtt megmutatnánk, hogy a szélességi keresés a legrövidebb út távolságokat határozza meg, a legrövidebb út távolságok egy fontos tulajdonságát vizsgáljuk meg.

22.1. lemma. *Legyen $G = (V, E)$ irányított vagy irányítatlan gráf és $s \in V$ tetszőleges csúcs. Ekkor, bármely $(u, v) \in E$ élre:*

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Bizonyítás. Ha u elérhető s-ből, akkor v is. Ebben az esetben, az s-ből v-be vezető legrövidebb út nem lehet hosszabb, mint ha az s-ből u-ba vezető legrövidebb utat kiegészítjük az (u, v) éllel, így az egyenlőtlenség teljesül. Ha u nem érhető el s-ből, akkor $\delta(s, u) = \infty$, ezért az egyenlőtlenség biztosan igaz. ■

Be akarjuk látni, hogy SzK által kiszámított d értékekre igaz, hogy minden $v \in V$ csúcsra $d[v] = \delta(s, v)$. Először megmutatjuk, hogy $d[v]$ felső korlátja $\delta(s, v)$-nek.

22.2. lemma. *Legyen $G = (V, E)$ irányított vagy irányítatlan gráf, és tegyük fel, hogy az* SzK *algoritmust futtatjuk G-n egy adott $s \in V$ kezdő csúcsból kiindulva. Ekkor az algoritmus befejeződésekor* SzK *által kiszámított d értékek minden $v \in V$ csúcsra kielégítik a $d[v] \geq \delta(s, v)$ egyenlőtlenséget.*

Bizonyítás. Az állítást teljes indukcióval bizonyítjuk. Az indukció alapja, hogy hányszor tettünk csúcsot a Q sorba (Sorba műveletek száma). Az indukciós feltevés, hogy minden $v \in V$ csúcsra fennáll $d[v] \geq \delta(s, v)$.

[1] A 24. és 25. fejezetekben a legrövidebb utakat általánosabban, súlyozott gráfokban vizsgáljuk. Ekkor minden élhez egy valós értéket rendelünk súlyként, és az út súlyát a benne szereplő élek súlyainak összegeként definiáljuk. Ebben a fejezetben súlyozatlan gráfokkal foglalkozunk.

Az indukció alapja az a helyzet, amely SzK 9. sorának végrehajtása után áll fenn, közvetlen azután, hogy s-t Q-ba tettük. Ekkor az indukciós feltevés teljesül, mert $d[s] = 0 = \delta(s, s)$ és $\forall v \in V - \{s\} : d[v] = \infty \geq \delta(s, v)$.

Az indukciós lépésben v legyen egy fehér csúcs, amelyet u csúcs szomszédsági listájának vizsgálatakor értünk el. Az indukciós feltevés miatt $d[u] \geq \delta(s, u)$. A 15. sorban végrehajtott értékadás és a 22.1. lemma alapján:

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v). \end{aligned}$$

Ezután a v csúcsot betesszük a Q sorba, és többet nem fogjuk betenni, mert a színe szürke lesz, és a 14–17. sorokat csak fehér csúcsok esetén hajtjuk végre. Így $d[v]$ értéke nem változik a továbbiakban, és az indukciós feltevésünk továbbra is fennáll. ■

$d[v] = \delta(s, v)$ bizonyításához pontosabban meg kell vizsgálnunk, miként működik a Q sor SzK végrehajtása alatt. A következő lemma kimondja, hogy a sorban legfeljebb két különböző d érték lehet.

22.3. lemma. *Tegyük fel, hogy az* SzK *algoritmust a $G = (V, E)$ gráfra alkalmazzuk, és a futás során a Q sor a $\langle v_1, v_2, \ldots, v_r \rangle$ csúcsokat tartalmazza, ahol v_1 a sor első, v_r pedig az utolsó eleme. Ekkor $d[v_r] \leq d[v_1] + 1$ és $d[v_i] \leq d[v_{i+1}]$ bármely $i = 1, 2, \ldots, r - 1$ értékre.*

Bizonyítás. A lemmát a sorműveletek száma szerinti indukcióval bizonyítjuk. Kezdetben, amikor a sor csak az s csúcsot tartalmazza, a lemma nyilvánvalóan igaz.

Az indukciós lépésben be kell látnunk, hogy az állítás fennáll akkor is, ha kiveszünk a sorból egy csúcsot, és akkor is, amikor beteszünk egy csúcsot. Ha a sor első elemét, v_1-et, kivesszük, az új fejelem v_2 lesz. (Ha a kivétel után a sor üres lesz, akkor az állítás teljesül, hiszen a második rész nyilvánvalóan igaz, az első résznek pedig nincs értelme.) Ekkor az indukciós feltétel miatt $d[v_1] \leq d[v_2]$, így $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, és a többi egyenlőtlenséget ez a változás nem érinti. Így a lemma fennáll v_2 új fejelemre.

Egy csúcs sorba helyezése az algoritmus alaposabb vizsgálatát igényli. SzK 17. sorában, amikor a v csúcsot betesszük a sorba és az lesz v_{r+1}, a Q sorból kivettük az u csúcsot, amelynek éppen ekkor vizsgáljuk a szomszédsági listáját. Így az új v_1 fejelemre $d[v_1] \geq d[u]$ fennáll, ezért $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. Az indukciós feltétel miatt $d[v_r] \leq d[u] + 1$, tehát $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$ is teljesül, a többi egyenlőtlenséget a művelet nem érinti. Ezért a lemma a v csúcs sorba helyezése után is fennáll. ■

Az alábbi következményben megfogalmazzuk azt a tényt, hogy a csúcsokat a d értékek szerint monoton növekvő sorrendben helyezzük be a sorba az algoritmus végrehajtása során.

22.4. következmény. *Tegyük fel, hogy v_i és v_j csúcsokat betettük a sorba* SzK *végrehajtása során, és v_i-t előbb tettük be, mint v_j-t. Amikor v_j csúcsot a sorba tesszük, akkor $d[v_i] \leq d[v_j]$ fennáll.*

Bizonyítás. Az állítás következik a 22.3. lemmából és abból a tényből, hogy minden csúcshoz egyszer rendelünk d értéket SzK végrehajtása során. ■

Az eddigiek felhasználásával bebizonyíthatjuk, hogy a szélességi keresés a legrövidebb út távolságokat határozza meg.

22.5. tétel (a szélességi keresés helyessége). *Legyen $G = (V, E)$ irányított vagy irányítatlan gráf, és tegyük fel, hogy* SzK *algoritmust a G gráfra alkalmazzuk egy adott $s \in V$ kezdő csúcsból kiindulva. Ekkor* SzK *működése során minden s-ből elérhető $v \in V$ csúcsot elér, és a befejezéskor $d[v] = \delta(s, v)$ minden $v \in V$ csúcsra. Továbbá, bármely s-ből elérhető $v \neq s$ csúcsra, az s-ből v-be vezető legrövidebb utak egyikét megkapjuk, ha az s-ből $\pi[v]$-be vezető legrövidebb utat kiegészítjük a $(\pi[v], v)$ éllel.*

Bizonyítás. Tegyük fel, hogy a tétel állításával szemben létezik olyan csúcs, amelyhez tartozó d érték nem a legrövidebb út távolsággal egyezik meg. Legyen v ezen csúcsok közül az, amelynek a legkisebb a $\delta(s, v)$ értéke. Nyilvánvalóan $v \neq s$. A 22.2. lemma miatt $d[v] \geq \delta(s, v)$, így a feltevésünk alapján $d[v] > \delta(s, v)$. A v csúcs elérhető s-ből, mert ha nem lenne az, akkor $\delta(s, v) = \infty \geq d[v]$ teljesülne. Legyen u egy s-ből v-be vezető legrövidebb út v-t közvetlenül megelőző csúcsa, azaz $\delta(s, v) = \delta(s, u) + 1$. $\delta(s, u) < \delta(s, v)$ fennállása és v választása miatt $d[u] = \delta(s, u)$. Az eddigiek alapján:

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1. \tag{22.1}$$

Tekintsük azt a pillanatot, amikor SzK kiveszi a Q sorból az u csúcsot a 11. sorban. Ekkor v csúcs fehér, szürke vagy fekete. Belátjuk, hogy egyik esetben sem állhat fenn a (22.1) egyenlőtlenség. Ha v fehér, akkor a 15. sort hajtjuk végre, és ezért $d[v] = d[u] + 1$, így nem teljesül a (22.1) egyenlőtlenség. Ha v fekete, akkor már kivettük a sorból, és a 22.4. következmény miatt $d[v] \leq d[u]$, ami szintén ellentmond a (22.1) egyenlőtlenségnek. Ha v szürke, akkor v-t akkor színeztük szürkére, amikor egy w csúcsot vettünk ki a sorból, méghozzá az u csúcs kivétele előtt. Ezért $d[v] = d[w] + 1$, ugyanakkor a 22.4. következmény miatt $d[w] \leq d[u]$, így $d[v] \leq d[u] + 1$, ami szintén ellentmond a (22.1) egyenlőtlenségnek.

A feltevésünk nem állhat fenn, tehát minden $v \in V$ csúcsra $d[v] = \delta(s, v)$. Minden s-ből elérhető csúcsot érintenünk kell, mert ha nem érintenénk, a csúcs d értéke végtelen lenne. Végül, vegyük észre, hogy ha $\pi[v] = u$, akkor $d[v] = d[u] + 1$, azaz egy s-ből v-be vezető legrövidebb utat kapunk, ha az s-ből $\pi[v]$-be vezető legrövidebb utat kiegészítjük a $(\pi[v], v)$ éllel. ■

Szélességi fák

Az SzK eljárás a gráf keresése során felépít egy szélességi fát, amint az a 22.3. ábrán látható. A fát az egyes csúcsok π értékeivel ábrázoljuk. Ezt pontosítjuk a következőkben. Egy s kezdőcsúcsú $G = (V, E)$ gráf esetén definiáljuk a $G_\pi = (V_\pi, E_\pi)$ ***előd részfát*** úgy, hogy

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

és

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}.$$

A G_π előd részfa ***szélességi fa,*** ha V_π elemei az s-ből elérhető csúcsok, és bármely $v \in V_\pi$ csúcsra egyetlen egyszerű út vezet s-ből v-be G_π-ben. Ez az út egyike az s-ből v-be vezető

legrövidebb G-beli utaknak. A szélességi fa valóban fa, mivel összefüggő és $|E_\pi| = |V_\pi| - 1$ (lásd B.2. tétel). E_π éleit ***fa éleknek*** nevezzük.

A következő lemma kimondja, hogy SzK futtatása után előálló előd részfa egy szélességi fa.

22.6. lemma. *Ha az* SzK *algoritmust egy irányított vagy irányítatlan $G = (V, E)$ gráfra alkalmazzuk, akkor az olyan π értékeket határoz meg, amelyekre a $G_\pi = (V_\pi, E_\pi)$ előd részfa egy szélességi fa.*

Bizonyítás. SzK 16. sorában akkor és csak akkor lesz $\pi[v] = u$, ha $(u, v) \in E$ és $\delta(s, v) < \infty$ – azaz v elérhető s-ből –, így V_π olyan csúcsokból áll, amelyek elérhetők s-ből. Miután G_π fa, a benne s-ből V_π csúcsaiba vezető utak egyediek a B.2. tétel miatt. A 22.5. tétel indukciós alkalmazásával azt kapjuk, hogy minden ilyen út legrövidebb út. ■

A következő eljárás kiírja az s-ből v-be vezető legrövidebb úton található csúcsokat, feltéve, hogy előtte már SzK eljárás futtatásával meghatároztuk a π értékeket.

ÚT-KIÍR(G, s, v)

```
1  if v = s
2     then print s
3     else if π[v] = NIL
4             then print „Nincs út” s „és” v „között”
5             else ÚT-KIÍR(G, s, π[v])
6                  print v
```

Az eljárás futási ideje az úton szereplő csúcsok számával arányos, mert minden rekurzív híváskor az új út egy csúccsal rövidebb.

Gyakorlatok

22.2-1. Határozzuk meg a szélességi keresés során előállított d és π értékeket a 22.2. ábra (a) részén látható irányított gráfra, ha a kezdő csúcs a 3-as címkéjű csúcs.

22.2-2. Határozzuk meg a szélességi keresés során előállított d és π értékeket a 22.3. ábrán látható irányítatlan gráfra, ha u a kezdő csúcs.

22.2-3. Mi az SzK algoritmus futási ideje, ha a bemeneti gráfot csúcsmátrixszal ábrázoljuk, és az algoritmust ennek megfelelően módosítjuk?

22.2-4. Lássuk be, hogy a szélességi keresés által u csúcshoz rendelt $d[u]$ érték független attól, hogy milyen sorrendben szerepelnek a csúcsok az egyes szomszédsági listákban. A 22.3. ábrát példaként felhasználva, mutassuk meg, hogy az SzK által meghatározott szélességi fa függhet attól, hogy a csúcsok milyen sorrendben helyezkednek el a szomszédsági listákban.

22.2-5. Adjunk meg olyan $G = (V, E)$ irányított gráfot, $s \in V$ kezdő csúcsot és $E_\pi \subseteq E$ élhalmazt, hogy minden $v \in V$ csúcsra az E_π-beli s-ből v-be vezető egyedüli út legrövidebb út G-ben, és ennek ellenére az E_π élhalmazt nem állíthatjuk elő, ha SzK-t G-re alkalmazzuk, akármilyen sorrendben is szerepelnek a csúcsok a szomszédsági listákban.

22.2-6. Kétféle profi ökölvívót különböztetünk meg: „jó fiúkat” és „rossz fiúkat”. Bármely két ökölvívó vagy rivalizál egymással, vagy nem. Tegyük fel, hogy adott n profi ökölvívó, amelyek közül r pár rivalizál. A rivalizáló párokat egy lista tartalmazza. Adjunk $O(n + r)$

futási idejű algoritmust annak eldöntésére, hogy ki lehet-e jelölni jó fiúkat, a maradékot rossz fiúnak tekintve, úgy hogy csak jó fiúk és rossz fiúk rivalizáljanak egymással. Ha van ilyen kijelölés, az algoritmus állítsa is azt elő.

22.2-7.⋆ Egy $T = (V, E)$ fa ***átmérője*** a fában előforduló legrövidebb út távolságok legnagyobbika, azaz

$$\max_{u,v \in V} \delta(u, v).$$

Adjunk hatékony algoritmust egy fa átmérőjének kiszámítására, és elemezzük az algoritmus futási idejét.

22.2-8. Legyen $G = (V, E)$ összefüggő, irányítatlan gráf. Adjunk $O(V+E)$ idejű algoritmust egy olyan G-beli út meghatározására, amely minden E-beli élen pontosan egyszer halad át az egyik és a másik irányban is. Adjunk módszert arra, hogy miként lehet egy útveszt őből kijutni, ha kellően sok pénzérmével rendelkezünk.

22.3. Mélységi keresés

A mélységi keresés stratégiája, amint azt a neve is mutatja, hogy a gráfban a lehető „legmélyebben” keressünk. A mélységi keresés során az utoljára elért, új kivezető élekkel rendelkező v csúcsból kivezető, még nem vizsgált éleket derítjük fel. Ha a v-hez tartozó összes élt megvizsgáltuk, akkor a keresés „visszalép”, és megvizsgálja annak a csúcsnak a kivezető éleit, amelyből v-t elértük. Ezt addig folytatja, amíg el nem éri az összes csúcsot, amely elérhető az eredeti kezdő csúcsból. Ha marad olyan csúcs, amelyet nem értünk el, akkor ezek közül valamelyiket kiválasztjuk mint új kezdő csúcsot, és az eljárást ebből kiindulva megismételjük. Ezt egészen addig folytatjuk, amíg az összes csúcsot el nem érjük.

A szélességi kereséshez hasonlóan, ha egy már elért u csúcs szomszédsági listájának vizsgálata során elérünk egy v csúcsot, akkor a mélységi keresés u-t feljegyzi mint v elődjét, azaz $\pi[v]$ értékét u-ra állítja. A szélességi kereséssel ellentétben, amikor az előd részgráf egy fa, a mélységi keresés által előállított előd részgráf több fából állhat, hiszen a keresést többször hajthatjuk végre különböző kezdőcsúcsokból kiindulva.[2] Ezért mélységi keresés esetén az ***előd részgráf*** definíciója kicsit eltér a szélességi keresésnél megadottól. Ekkor ez legyen $G_\pi = (V, E_\pi)$, ahol

$$E_\pi = \{(\pi[v], v) : v \in V \text{ és } \pi[v] \neq \text{NIL}\}.$$

Mélységi keresés esetén az előd részgráf egy ***mélységi erdő,*** amely több ***mélységi fát*** tartalmaz. E_π éleit ***fa éleknek*** nevezzük.

A szélességi kereséshez hasonlóan a csúcsok állapotait ebben az esetben is színekkel különböztetjük meg. Kezdetben minden csúcs fehér, amikor ***elérünk*** egy csúcsot, akkor szürkére színezzük azt, és befeketítjük, ha ***elhagytuk,*** azaz amikor a szomszédsági listájának minden elemét megvizsgáltuk. Ez a módszer biztosítja, hogy minden csúcs pontosan egy mélységi fába kerüljön, azaz, hogy ezek a fák diszjunktak legyenek.

[2] Önkényesnek tűnhet, hogy a szélességi keresés csak egy kezdő csúcsot használ, míg a mélységi keresés több kezdőcsúcsból indulhat ki. Noha elvileg semmi akadálya sincs annak, hogy a szélességi keresés induljon ki több kezdőcsúcsból, és a mélységi keresés használjon egyet, a bemutatott módszer illeszkedik ezen keresések tipikus felhasználásához. A szélességi keresést rendszerint egy kezdőcsúcsból kiinduló legrövidebb út távolságok (és a megfelelő előd részfa) meghatározására használjuk. A mélységi keresés gyakran egy másik algoritmus eljárása, amint azt látni fogjuk később a fejezetben.

A mélységi erdő létrehozásán kívül a mélységi keresés minden csúcshoz ***időpontot*** rendel. Minden v csúcshoz két időpont tartozik: az első $d[v]$ időpontot akkor rögzítjük, amikor v-t elérjük (és szürkére színezzük); a második $f[v]$ időpontot akkor jegyezzük fel, amikor befejezzük v szomszédsági listájának vizsgálatát (és v-t befeketítjük). Ezeket az időpontokat több gráfalgoritmus felhasználja, és általában hasznosak lesznek a mélységi keresés tulajdonságainak vizsgálatakor.

A következő MK eljárás egy u csúcs ***elérési időpontját*** a $d[u]$, ***elhagyási időpontját*** pedig az $f[u]$ változóban tárolja. Ezek az időpontok 1 és $2|V|$ közötti egész számok, hiszen a $|V|$ csúcs mindegyikét pontosan egyszer érjük el, illetve hagyjuk el. Továbbá bármely u csúcsra

$$d[u] < f[u]. \tag{22.2}$$

$d[u]$ időpont előtt az u csúcs FEHÉR, SZÜRKE $d[u]$ és $f[u]$ között, és FEKETE azután.

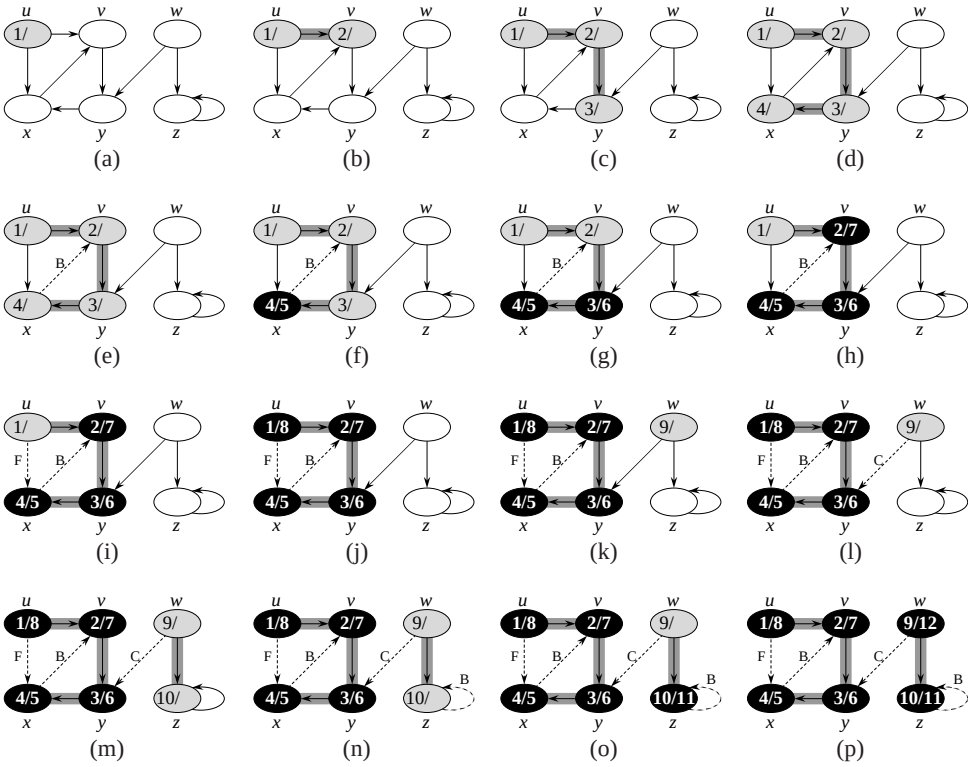
A következő eljárás adja meg a mélységi keresés algoritmusát. A bemeneti gráf lehet irányított vagy irányítatlan. Az *idő* globális változót használjuk az időpontok feljegyzéséhez.

```
MK(G)
1  for minden u ∈ V[G] csúcsra
2      do szín[u] ← FEHÉR
3         π[u] ← NIL
4  idő ← 0
5  for minden u ∈ V[G] csúcsra
6      do if szín[u] = FEHÉR
7            then MK-BEJÁR(u)
```

```
MK-BEJÁR(u)
1  szín[u] ← SZÜRKE                ▷ Most érjük el a fehér u csúcsot.
2  idő ← idő +1
3  d[u] ← idő
4  for minden v ∈ Adj[u] csúcsra   ▷ (u, v) él vizsgálata.
5      do if szín[v] = FEHÉR
6            then π[v] ← u
7                 MK-BEJÁR(v)
8  szín[u] ← FEKETE                ▷ u fekete, mert elhagytuk.
9  f[u] ← idő ← idő +1
```

A 22.4. ábra szemlélteti MK működését a 22.2. ábra gráfján.

MK első három sorában minden csúcs színét fehérre állítjuk, és a π értékek kezdetben NIL-ek lesznek. A 4. sor az időt inicializálja. Az 5–7. sorokban V csúcsain haladunk végig, és ha fehér csúcshoz jutunk, az abból elérhető csúcsokat bejárjuk MK-BEJÁR segítségével. Valahányszor MK-BEJÁR(u)-t meghívjuk a 7. sorban, az u csúcs a mélységi erdő egy új fájának a gyökere lesz. MK befejeződésekor minden u csúcsnak ismert a $d[u]$ ***elérési*** és $f[u]$ elhagyási időpontja.



22.4. ábra. MK algoritmus működésének szemléltetése egy irányított gráfon. Az algoritmusban az adott pontig megvizsgált éleket megvastagítottuk, ha fa élek, egyébként szaggatott vonallal jelöljük ezeket. Azokat az éleket, amelyek nem tartoznak fához, B, C vagy F címkékkel láttuk el, annak megfelelően, hogy visszamutató, kereszt vagy előremutató élek. A csúcsokban feltüntettük az elérési/elhagyási időpontokat.

MK-BEJÁR(u) hívásakor az u csúcs fehér. Az első sorban u színét szürkére változtatjuk, a 2. sorban eggyel növeljük a globális *idő* változó értékét, és a 3. sorban feljegyezzük $d[u]$-ban az elérési időt. A 4–7. sorokban megvizsgáljuk u összes v szomszédját, és rekurzív módon bejárjuk a v alatti részt, ha v fehér. Azt mondjuk, hogy a mélységi keresés ***megvizsgálja*** az (u, v) élt, amikor a $v \in Adj[u]$ csúcs kerül sorra a 4. sorban. Végül, miután az u-ból kivezető összes élt megvizsgáltuk, a 8–9. sorokban u színét feketére állítjuk, és feljegyezzük $f[u]$-ban az elhagyás időpontját.

Vegyük észre, hogy a mélységi keresés eredményét befolyásolhatja, hogy milyen sorrendben vizsgáljuk meg a csúcsokat MK-BEJÁR 5. sorában, és az is, hogy milyen sorrendben járjuk be egy csúcs szomszédjait MK-BEJÁR 4. sorában. Az eltérő sorrendek rendszerint nem okoznak gondot a gyakorlatban, mert általában *bármelyik* eredmény hatékonyan felhasználható, és alapvetően ekvivalens végeredményt adnak.

Vizsgáljuk meg MK futási idejét. Az algoritmus 1–3. és 5–7. sorainak futási ideje $\Theta(V)$, ha eltekintünk a meghívott MK-BEJÁR eljárás végrehajtásához szükséges időtől. Aggregációs elemzést használunk, ahogy azt a szélességi keresés esetén is tettük. Az MK-BEJÁR eljárást pontosan egyszer hívjuk meg minden egyes $v \in V$ csúcsra, hiszen csak fehér csúcsra hívható meg, és az eljárás elsőként szürkére festi a csúcsot. MK-BEJÁR(v) egyszeri végrehajtása során

a 4–7. sorokat alkotó ciklus iterációszáma: $|Adj[v]|$. Ugyanakkor

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

így MK-BEJÁR 4–7. sorai végrehajtásához felhasznált összidő $\Theta(E)$. Ezért MK futási ideje $\Theta(V + E)$.

A mélységi keresés tulajdonságai

A mélységi keresés a gráf szerkezetére vonatkozó sok hasznos ismeretet felderít. A mélységi keresés talán legalapvetőbb tulajdonsága, hogy a G_π előd részgráf valóban erdő, hiszen a mélységi fák szerkezete pontosan tükrözi az MK-BEJÁR rekurzív hívásainak mikéntjét. Azaz, $u = \pi[v]$ akkor és csak akkor, ha MK-BEJÁR(v)-t u szomszédsági listájának vizsgálata során hívtuk meg. Továbbá v csúcs az u csúcs leszármazottja a mélységi erdőben akkor és csak akkor, ha v csúcsot az alatt az idő alatt fedeztük fel, amíg u szürke volt.

A mélységi keresés másik fontos tulajdonsága, hogy az elérési és elhagyási időpontok ***zárójelszerkezetűek.*** Ha az u csúcs elérését nyitó zárójellel, „(u”, ábrázoljuk, elhagyását pedig csukó zárójellel, „u)”, akkor az elérések és elhagyások története egy jól formázott kifejezést alkot, azaz a kapott zárójelezés helyes. Például a 22.5. ábra (a) részén látható mélységi keresés megfelel az ábra (b) részén található zárójelezésnek. A zárójelszerkezetre vonatkozó feltételt mondja ki más formában a következő tétel.

22.7. tétel (zárójelezés tétele). *Mélységi keresést alkalmazva egy* (*irányított vagy irányítatlan*) $G = (V, E)$ *gráfra, a következő három feltétel közül pontosan egy teljesül bármely u és v csúcsra:*

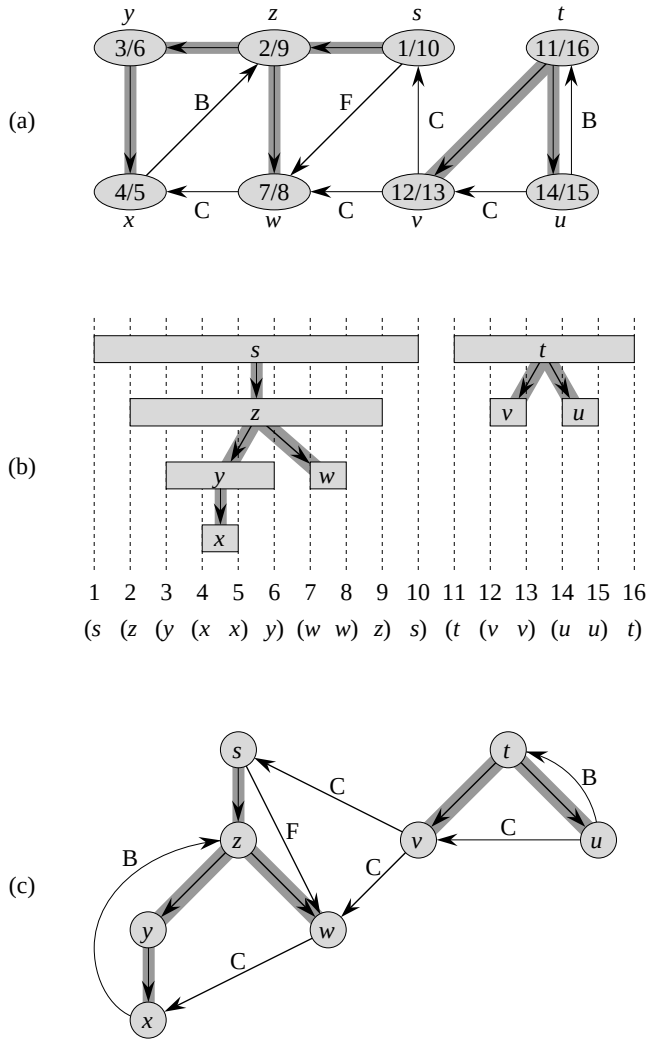
- *a* $[d[u], f[u]]$ *és* $[d[v], f[v]]$ *intervallumok diszjunktak, és u és v csúcsok közül egyik sem leszármazottja a másiknak a mélységi erdőben,*
- *a* $[d[v], f[v]]$ *intervallum tartalmazza a* $[d[u], f[u]]$ *intervallumot, és u csúcs v leszármazottja a mélységi fában, vagy*
- $[d[u], f[u]]$ *intervallum tartalmazza a* $[d[v], f[v]]$ *intervallumot, és v csúcs u leszármazottja a mélységi fában.*

Bizonyítás. Először a $d[u] < d[v]$ esetet vizsgáljuk meg. Ekkor két alesetet kell megkülönböztetni aszerint, hogy $d[v] < f[u]$, vagy sem. Az első alesetben $d[v] < f[u]$, így v elérésekor u szürke volt. Ebből az következik, hogy v csúcs u leszármazottja. Továbbá, mivel v-t később értük el, mint u-t, ezért minden kivezető élt megvizsgálunk, és így v-t elhagyjuk, mielőtt visszatérnénk u-hoz és elhagynánk azt. Tehát ebben az esetben a $[d[v], f[v]]$ intervallumot tartalmazza a $[d[u], f[u]]$ intervallum. A másik alesetben, amikor $f[u] < d[v]$, a (22.2) egyenlőtlenség felhasználásával azt kapjuk, hogy a $[d[u], f[u]]$ és $[d[v], f[v]]$ intervallumok diszjunktak. Miután az intervallumok diszjunktak, egyik csúcsot sem akkor értük fel, amikor a másik szürke volt, ezért egyik csúcs sem leszármazottja a másiknak.

A $d[v] < d[u]$ eset – u és v szerepét felcserélve – hasonlóan bizonyítható. ■

22.8. következmény (leszármazottak intervallumainak beágyazása). *v csúcs akkor és csak akkor leszármazottja az u csúcsnak az* (*irányított vagy irányítatlan*) *G gráf mélységi erdejében, ha* $d[u] < d[v] < f[v] < f[u]$.

Bizonyítás. Adódik a 22.7. tételből. ■



22.5. ábra. A mélységi keresés tulajdonságai. **(a)** Egy irányított gráfon végrehajtott mélységi keresés eredménye. A csúcsokban feltüntettük az elérési/elhagyási időpontokat. Az éleket a 22.4. ábrának megfelelően ábrázoltuk. **(b)** A csúcsok elérési és elhagyási időintervallumai megfelelnek a megadott zárójelezésnek. Az egyes téglalapok a megfelelő csúcs elérési és elhagyási időpontja által meghatározott intervallumokat szemléltetik. Feltüntettük a fa éleket is. Két intervallumnak csak akkor lehet közös része, ha az egyik tartalmazza a másikat. Ekkor a kisebb intervallumnak megfelelő csúcs a nagyobb intervallumhoz tartozó csúcs leszármazottja. **(c)** Az (a) részen látható gráfot újrarajzoltuk. Az összes fa él és előremutató él lefelé halad a mélységi fában, az összes visszamutató él a leszármazottról mutat fel az ősre.

A következő tétel egy újabb fontos jellemzését adja annak, hogy egy csúcs mikor leszármazottja egy másiknak a mélységi erdőben.

22.9. tétel (fehér út tétel). *Egy* (*irányított vagy irányítatlan*) $G = (V, E)$ *gráfhoz tartozó mélységi erdőben a v csúcs akkor és csak akkor leszármazottja az u csúcsnak, ha u elérésekor a* $d[u]$ *időpontban v elérhető u-ból olyan úton, amely csak fehér csúcsokat tartalmaz.*

Bizonyítás. $\Rightarrow$: Tegyük fel, hogy v u leszármazottja. Legyen w a mélységi fában u-ból v-be vezető út tetszőleges csúcsa, és w legyen u leszármazottja. A 22.8. következmény miatt $d[u] < d[w]$, ezért w fehér $d[u]$ időpontban.

$\Leftarrow$: Tegyük fel, hogy v elérhető u-ból olyan úton, amelynek csúcsai fehérek $d[u]$ időpontban, de v nem válik u leszármazottjává a mélységi fában. Ekkor feltehetjük, hogy az út minden más pontja u leszármazottja lesz. (Ha ez nem teljesülne, akkor válasszuk ki v-t úgy az útból, hogy az legyen legközelebb u-hoz a nem leszármazott csúcsok közül.) Legyen w csúcs v elődje az úton (w és u lehet ugyanaz a csúcs). Ekkor w u leszármazottja, és a 22.8. következmény miatt $f[w] \leq f[u]$. Vegyük észre, hogy v-t csak u után érhetjük el, még azelőtt, hogy w-t elhagynánk. Így $d[u] < d[v] < f[w] \leq f[u]$. A 22.7. tétel miatt a $[d[u], f[u]]$ intervallum tartalmazza a $[d[v], f[v]]$ intervallumot. A 22.8. következmény miatt v-nek u leszármazottjának kell lennie. ■

Élek osztályozása

A mélységi keresés további érdekes tulajdonsága, hogy annak segítségével a $G = (V, E)$ bemeneti gráf éleit osztályokba sorolhatjuk. (Az osztályozás természetesen függ a mélységi kereséstől.) Az élek osztályozását felhasználhatjuk a gráfra vonatkozó fontos információk meghatározására. Például a következő alfejezetben megmutatjuk, hogy egy irányított gráf akkor és csak akkor körmentes, ha a mélységi keresés során nem találunk „visszamutató" éleket (22.11. lemma).

Négy éltípust különböztethetünk meg G_π mélységi erdő segítségével, amelyet a G gráf mélységi keresése során kaptunk.

1. ***Fa élek*** a G_π mélységi erdő élei. Az (u, v) él fa él, ha v-t az (u, v) él vizsgálata során értük el először.
2. ***Visszamutató él*** az (u, v) él, ha v őse u-nak egy mélységi fában. A hurokéleket, amelyek előfordulhatnak irányított gráfokban, visszamutató éleknek tekintjük.
3. ***Előremutató él*** az (u, v) él, ha v leszármazottja u-nak egy mélységi fában, és (u, v) nem éle a mélységi fának.
4. ***Kereszt él*** az összes többi él. Ezek ugyanazon mélységi fa csúcsait köthetik össze, ha azok közül egyik sem őse a másiknak, illetve végpontjaik különböző mélységi fákhoz tartozó csúcsok lehetnek.

A 22.4. és 22.5. ábrákon az éleket a típusok alapján címkéztük meg. A 22.5. ábra (c) részén az is látható, miként lehet az (a) rész gráfját újrarajzolni úgy, hogy minden fa él és előremutató él lefele irányuljon a mélységi fában, és az összes visszamutató él felfele irányuljon. Tetszőleges gráfot újra lehet rajzolni ilyen módon.

Az MK algoritmust módosíthatjuk úgy, hogy az éleket osztályozza a vizsgálatuk során. Az alapötlet, hogy az (u, v) él osztálya meghatározható a v csúcs színe alapján, amelyet az él első vizsgálatakor érünk el (eltekintve az előremutató és kereszt élek megkülönböztetésétől):

1. FEHÉR esetén az él fa él,
2. SZÜRKE esetén az él visszamutató él, és
3. FEKETE esetén az él előremutató vagy kereszt él.

Az első eset nyilvánvaló az algoritmus specifikációja miatt. A második esetben vegyük észre, hogy a szürke csúcsok a leszármazottak lineáris láncát alkotják, amely megfelel az aktív MK-BEJÁR hívások vermének. A szürke csúcsok száma eggyel nagyobb annál, mint amilyen mélyen vagyunk az éppen elért csúcshoz tartozó mélységi fában. A keresést mindig a legmélyebben található szürke csúcsból folytatjuk, így egy másik szürke csúcsba vezető él egy ősre mutat. A harmadik eset a fennmaradt lehetőségeknek felel meg. Belátható, hogy egy ilyen (u, v) él előremutató, ha $d[u] < d[v]$, illetve kereszt él, ha $d[u] > d[v]$. (Lásd 22.3-4. gyakorlatot.)

Irányítatlan gráf esetén előfordulhat, hogy az osztályozás nem egyértelmű, ugyanis (u, v) és (v, u) valójában ugyanaz az él. Ebben az esetben az él osztálya az *első* olyan lesz az osztályozási listából, amelybe besorolható. Ezzel ekvivalens (lásd 22.3-5. gyakorlat), hogy az élt annak alapján osztályozzuk, hogy (u, v) vagy (v, u) közül melyiket vizsgáljuk el előbb az algoritmusban.

Most belátjuk, hogy irányítatlan gráfok mélységi keresése során nem találunk el őremutató, illetve kereszt élt.

22.10. tétel. *Egy irányítatlan G gráf mélységi keresésekor bármely él vagy fa él, vagy visszamutató él.*

Bizonyítás. Legyen (u, v) tetszőleges éle G-nek, és tegyük fel, hogy $d[u] < d[v]$. (Ezt nyugodtan megtehetjük.) Ekkor v-t el kell érnünk és el kell hagynunk, miel őtt elhagynánk u-t (amíg u szürke), mivel v szerepel u szomszédsági listájában. Ha (u, v) élt u-ból v felé haladva vizsgáljuk először, akkor v-t megelőzően nem értük még el (fehér), különben már megvizsgáltuk volna ezt az élt v-ből u felé haladva. Ezért (u, v) fa él lesz. Ha (u, v)-t v-ből u felé haladva vizsgáljuk először, akkor az visszamutató él, hiszen u még szürke az él első vizsgálatakor. ■

A következő alfejezetekben a kimondott tételek számos alkalmazását mutatjuk be.

Gyakorlatok

22.3-1. Készítsünk egy háromszor hármas táblázatot, amelynek sorait is és oszlopait is a FEHÉR, SZÜRKE és FEKETE értékekkel címkézzük. A táblázat (i, j) eleme jelölje azt, hogy egy irányított gráf mélységi keresése során valamikor vezet i színű csúcsból j színű csúcsba él. Minden lehetséges él esetén tüntessük fel, hogy az milyen típusú lehet. Készítsünk ugyanilyen táblázatot irányítatlan gráf esetén is.

22.3-2. Szemléltessük a mélységi keresés működését a 22.6. ábra gráfján. Tegyük fel, hogy az MK eljárás 5–7. sorában szereplő **for** ciklus a csúcsokat ábécé szerinti sorrendben vizsgálja, továbbá, hogy a szomszédsági listák is ábécé szerint rendezettek. Adjuk meg a csúcsok elérési és elhagyási időpontjait, és az élek osztályozását.

22.3-3. Adjuk meg a 22.4. ábrán látható mélységi kereséshez tartozó zárójelezést.

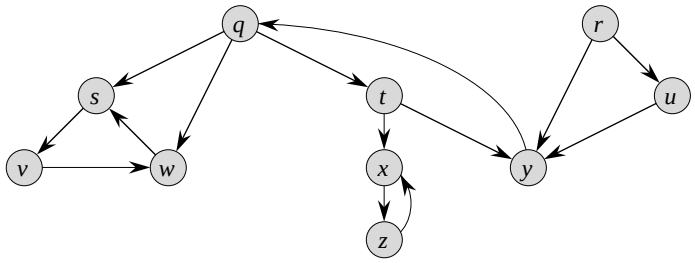
22.3-4. Lássuk be, hogy (u, v) él

a. fa vagy előremutató él akkor és csak akkor, ha $d[u] < d[v] < f[v] < f[u]$,

b. visszamutató él akkor és csak akkor, ha $d[v] < d[u] < f[u] < f[v]$, és

c. kereszt él akkor és csak akkor, ha $d[v] < f[v] < d[u] < f[u]$.

22.3-5. Mutassuk meg, hogy ha egy irányítatlan gráfban annak alapján határozzuk meg, hogy (u, v) él fa él vagy visszamutató él, hogy a mélységi keresésben (u, v)-ként vagy (v, u)-



22.6. ábra. Irányított gráf a 22.3-2. és 22.5-2. gyakorlatokhoz.

ként vizsgáljuk előbb, az ekvivalens azzal, hogy ha az osztályt az élosztályok definiálásakor megadott prioritások figyelembevételével adjuk meg.

22.3-6. Írjuk át az MK eljárást úgy, hogy egy verem használatával küszöböljük ki a rekurziót.

22.3-7. Adjunk ellenpéldát a következő sejtésre: ha egy irányított G gráfban vezet út u-ból v-be, és $d[u] < d[v]$ a mélységi keresésben, akkor v csúcs u leszármazottja az előállított mélységi erdőben.

22.3-8. Adjunk ellenpéldát a következő sejtésre: ha egy irányított G gráfban vezet út az u csúcsból a v csúcsba, akkor bármely mélységi keresés eredménye szükségszerűen $d[v] \leq f[u]$.

22.3-9. Módosítsuk a mélységi keresés algoritmusát úgy, hogy a G irányított gráf összes élét kiírja a típusukkal együtt. Adjuk meg a szükséges módosításokat (ha egyáltalán kell változtatás), ha G irányítatlan.

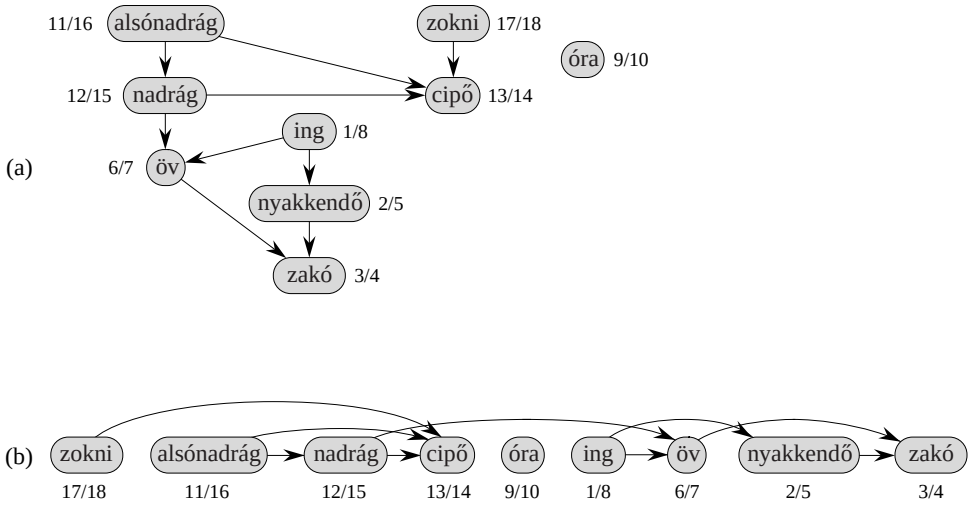
22.3-10. Magyarázzuk meg miként fordulhat elő, hogy egy irányított gráf u csúcsa a keresés után egy olyan mélységi fába kerül, amely csak az u csúcsból áll, annak ellenére, hogy G-ben vannak u-ba bevezető és u-ból kivezető élek.

22.3-11. Mutassuk meg, hogy egy irányítatlan G gráf mélységi keresése felhasználható G összefüggő komponenseinek meghatározására, és a mélységi erdő annyi fából áll, amennyi összefüggő komponens van G-ben. Pontosabban, mutassuk meg miként kell a mélységi keresést módosítani, hogy minden v csúcshoz egy 1 és k közötti $ök[v]$ értéket rendeljünk úgy, hogy $ök[u] = ök[v]$ akkor és csak akkor álljon fenn, ha u és v ugyanahhoz az összefüggő komponenshez tartozik. (k az összefüggő komponensek száma.)

22.3-12.* Az irányított $G = (V, E)$ gráf ***egyszeresen összefüggő***, ha $u \rightsquigarrow v$-ből következik, hogy legfeljebb egy egyszerű út vezet u-ból v-be bármely $u, v \in V$ esetén. Adjunk hatékony algoritmust annak eldöntésére, hogy egy irányított gráf egyszeresen összefüggő-e.

22.4. Topologikus rendezés

Ebben az alfejezetben megmutatjuk miként használható a mélységi keresés irányított, körmentes gráfok topologikus rendezésére. Egy $G = (V, E)$ irányított gráf ***topologikus rendezése*** a csúcsainak sorba rendezése úgy, hogy ha G-ben szerepel az (u, v) él, akkor u előzze meg v-t a sorban. (Ha a gráf tartalmaz irányított kört, akkor nincs ilyen sorba rendezés.) Egy gráf topologikus rendezését elképzelhetjük úgy is, hogy a gráf csúcsait egy vízszintes vo-



22.7. ábra. (a) A ruhadarabok topologikus rendezése. (u, v) irányított él jelentése: u ruhadarabot előbb kell felvenni, mint v-t. A mélységi keresés elérési/elhagyási időpontjait az egyes csúcsok mellé írtuk. **(b)** Ugyanaz a gráf topologikusan rendezve. A csúcsokat balról jobbra haladva rendeztük az elhagyási időpontok szerinti csökkenő sorrendbe. Vegyük észre, hogy minden irányított él balról jobbra mutat.

nal mentén helyezzük sorba, és az irányított élek balról jobbra mutatnak. Így a topologikus rendezés különbözik a II. részben bemutatott szokásos rendezésektől.

Irányított, körmentes gráfokat számos esetben alkalmaznak események precedenciájának megadására. A 22.7. ábra a reggeli öltözködéskor felmerülő problémát szemlélteti. Bizonyos ruhadarabokat előbb kell felvenni, mint másokat (pl.: zoknit a cipő előtt). Más ruhaneműk sorrendje tetszőleges lehet (pl.: zokni és nadrág). A 22.7. ábra (a) részén egy irányított (u, v) él jelöli, hogy u ruhadarabot v előtt kell felvenni. A gráf topologikus rendezése megad egy lehetséges öltözködési sorrendet. A 22.7. ábra (b) része tartalmazza a topologikusan rendezett gráfot. A csúcsokat egy vízszintes vonal mentén állítottuk sorba, és az összes irányított él balról jobbra mutat.

A következő egyszerű algoritmus topologikusan rendez irányított, körmentes gráfokat.

TOPOLOGIKUS-RENDEZÉS(G)

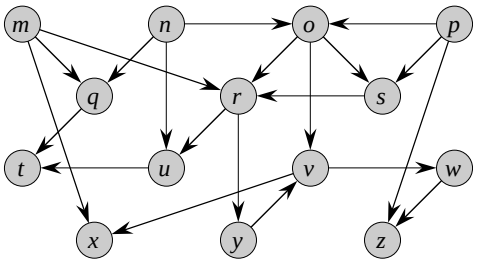
1 MK(G) hívása; minden v csúcsra meghatározzuk az $f[v]$ elhagyási időt
2 az egyes csúcsok elhagyásakor szúrjuk be azokat egy láncolt lista elejére
3 **return** a csúcsok láncolt listája

A 22.7. ábra (b) részén látható, hogy a topologikusan rendezett csúcsok az elhagyási időpontok fordított sorrendjében szerepelnek.

A topologikus rendezés elvégezhető $\Theta(V + E)$ időben, hiszen a mélységi keresés ideje $\Theta(V + E)$, és a $|V|$ csúcs mindegyike $O(1)$ idő alatt beszúrható a láncolt lista elejére.

Az algoritmus helyességét a következő lemma segítségével látjuk be, amellyel irányított, körmentes gráfokat jellemezhetünk.

22.11. lemma. *Egy G irányított gráf akkor és csak akkor körmentes, ha G mélységi keresése során nem találunk visszamutató élt.*



22.8. ábra. Topologikusan rendezendő irányított gráf.

Bizonyítás. $\Rightarrow$: Tegyük fel, hogy létezik (u, v) visszamutató él. Ekkor a v csúcs őse u-nak a mélységi erdőben, azaz létezik G-ben v-ből u-ba vezető út. Ezt kiegészítve az (u, v) éllel egy kört kapunk.

$\Leftarrow$: Tegyük fel, hogy G tartalmaz egy c kört. Belátjuk, hogy ekkor a mélységi keresésben találunk visszamutató élt. Legyen v a c kör elsőnek elért csúcsa, és (u, v) él a megelőző él c-ben. A $d[v]$ időpontban c csúcsai egy v-ből u-ba vezető út fehér csúcsai. A fehér út tétel miatt, az u csúcs v leszármazottja lesz a mélységi erdőben. Így (u, v) visszamutató él. ■

22.12. tétel. TOPOLOGIKUS-RENDEZÉS(G) *egy irányított, körmentes G gráf topologikus rendezését állítja elő.*

Bizonyítás. Tegyük fel, hogy az MK algoritmus segítségével határozzuk meg egy irányított, körmentes $G = (V, E)$ gráf csúcsainak elhagyási időpontjait. Elegendő belátni, hogy bármely $u, v \in V$ $(u \neq v)$ csúcspárra teljesül: ha vezet u-ból v-be él G-ben, akkor $f[v] < f[u]$. Tekintsünk egy tetszőleges MK(G) által megvizsgált (u, v) élt. Amikor az algoritmus az élt megvizsgálja, akkor v nem lehet szürke, mert abban az esetben v őse lenne u-nak és (u, v) visszamutató él lenne, ami ellentmond a 22.11. lemmának. Tehát v vagy fehér, vagy fekete. Ha v fehér, akkor u leszármazottja lesz, ezért $f[v] < f[u]$. Ha v fekete, v-t már elhagytuk, és $f[v]$ értékét beállítottuk. Ugyanakkor u-t még nem hagytuk el, így még nem adtunk értéket $f[u]$-nak, ezért amikor ezt megtesszük $f[v] < f[u]$ szintén teljesülni fog. Ezért a G gráf bármely (u, v) élére $f[v] < f[u]$, így a tétel állítása fennáll. ■

Gyakorlatok

22.4-1. Adjuk meg a csúcsok sorrendjét, amelyet a TOPOLOGIKUS-RENDEZÉS határoz meg, ha a 22.8. ábrán látható gráfra alkalmazzuk. (Az alkalmazás módja ugyanaz, mint a 22.3-2. gyakorlatban.)

22.4-2. Adjunk lineáris futási idejű algoritmust, amelynek bemenete egy irányított, körmentes $G = (V, E)$ gráf és annak két csúcsa, s és t, és az algoritmus meghatározza az s-ből t-be vezető utak számát G-ben. Például a 22.8. ábrán látható irányított, körmentes gráfban pontosan négy út vezet a p csúcsból a v-be: *pov*, *poryv*, *posryv* és *psryv*. (Az algoritmusnak csak az utak számát kell meghatároznia, az utakat nem kell kiírnia.)

22.4-3. Adjunk algoritmust annak eldöntésére, hogy egy irányítatlan $G = (V, E)$ gráf tartalmaz-e kört. Az algoritmus futási ideje $O(V)$ legyen, függetlenül $(|E|)$-től.

22.4-4. Bizonyítsuk vagy cáfoljuk: Ha G irányított gráfban van kör, akkor TOPOLOGIKUS-RENDEZÉS(G) a csúcsok olyan sorrendjét adja meg, amelyben a lehető legkevesebb számú „rossz” él szerepel. Rossz él az, amelyik ellentmond az előállított sorrendnek.
22.4-5. Egy irányított, körmentes $G = (V, E)$ gráf topologikus rendezése más módszerrel is meghatározható: Keressünk egy 0 bemeneti fokú csúcsot, írjuk ezt ki, és töröljük a gráfból az összes kivezető élével együtt; és ezt ismételjük, amíg lehet. Hogyan valósíthatjuk meg ezt a módszert úgy, hogy az előállított algoritmus futási ideje $O(V + E)$ legyen? Miként viselkedik az algoritmus, ha G-ben van kör?

22.5. Erősen összefüggő komponensek

Ebben az alfejezetben a mélységi keresés egyik klasszikus alkalmazásával foglalkozunk, egy irányított gráf erősen összefüggő komponenseinek meghatározásával. Megmutatjuk miként tehetjük ezt meg két mélységi keresés felhasználásával. Sok algoritmus a bemeneti gráfot első lépésben erősen összefüggő komponenseire bontja; így gyakran az eredeti feladat részfeladatokra osztható az erősen összefüggő komponenseknek megfelelően. Az erősen összefüggő komponensek közötti kapcsolatok szerkezete adja meg, hogy miként állíthatjuk elő az egyes részfeladatok megoldásaiból az eredeti feladat megoldását.

Amint azt a B függelékben leírtuk, $G = (V, E)$ irányított gráf egy erősen összefüggő komponense a csúcsok egy maximális $C \subseteq V$ halmaza, amelynek bármely $u, v \in C$ csúcsára $u \rightsquigarrow v$ és $v \rightsquigarrow u$ fennáll, azaz u és v kölcsönösen elérhetők egymásból. A 22.9. ábrán egy példát láthatunk.

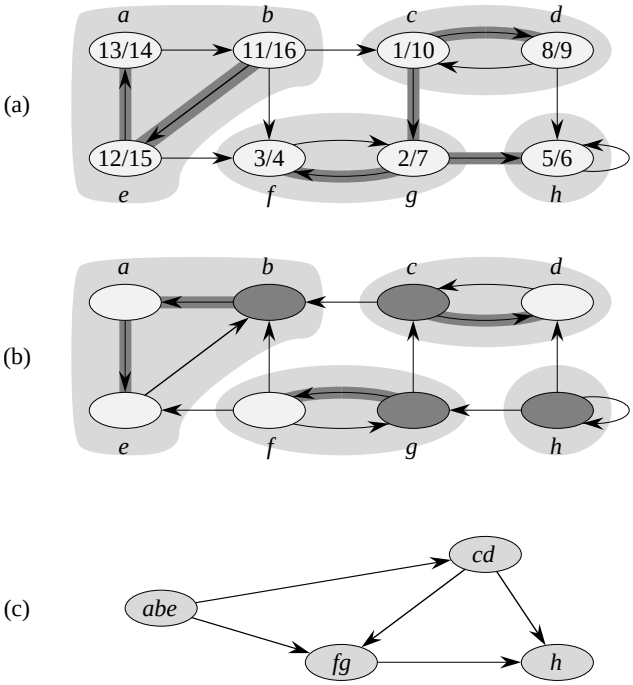
$G = (V, E)$ gráfot erősen összefüggő komponenseire bontó algoritmusunk használja G transzponáltját, G^T-t, amelyet a 22.1-3. gyakorlatban definiáltunk. Eszerint $G^T = (V, E^T)$, ahol $E^T = \{(u, v) : (v, u) \in E\}$, azaz E^T a G éleit tartalmazza az irányítás felcserélésével. G szomszédsági listás ábrázolása esetén $O(V + E)$ idő alatt meghatározhatjuk G^T-t. Egyszerű észrevétel, hogy G és G^T erősen összefüggő komponensei megegyeznek, hiszen u és v csúcsok akkor és csak akkor érhetőek el egymásból G-ben, ha elérhetők egymásból G^T-ben. A 22.9. ábra (b) része tartalmazza az (a) rész gráfjának transzponáltját. Az erősen összefüggő komponenseket beszürkítettük.

A következő lineáris (azaz $\Theta(V + E)$) idejű algoritmus meghatározza egy $G = (V, E)$ irányított gráf erősen összefüggő komponenseit két mélységi keresés segítségével, amelyek közül az egyiket G-re, a másikat pedig G^T-re alkalmazzuk.

ERŐSEN-ÖSSZEFÜGGŐ-KOMPONENSEK(G)

1 MK(G) hívása; minden u csúcsra kiszámítjuk az $f[u]$ elhagyási időpontot
2 G^T meghatározása
3 MK(G^T) hívása, de MK fő ciklusában a csúcsokat $f[u]$ szerint csökkenő sorrendben vizsgáljuk (ahogy az első sorban kiszámítjuk)
4 a 3. lépésben kapott mélységi erdő egyes fáinak csúcsait írjuk ki, mint erősen összefüggő komponenseket

Az algoritmus a $G^{EÖK} = (V^{EÖK}, E^{EÖK})$ ***komponens gráf,*** amelyet a következőkben definiálunk, alapvető tulajdonságán alapul. Tegyük fel, hogy $C_1, C_2, \ldots, C_k$ a G gráf erősen



22.9. ábra. (a) G irányított gráf, amelynek erősen összefüggő komponenseit szürkével jelöltük. Minden csúcsba beírtuk az elérési/elhagyási időpontokat. A fa éleket megvastagítottuk. **(b)** G gráf transzponáltja: G^T. ERŐSEN-ÖSSZEFÜGGŐ-KOMPONENSEK 3. sorában meghatározott mélységi erdőt a fa élek megvastagításával szemléltetjük. Minden erősen összefüggő komponens megfelel egy mélységi fának. A sötétre színezett b, c, g és h csúcsok a gyökerei a G^T mélységi keresése során előállított mélységi fáknak. **(c)** A körmentes $G^{EÖK}$ komponens gráfot megkapjuk, ha az egyes erősen összefüggő komponenseket összevonjuk egyetlen csúcsba.

összefüggő komponensei. A $V^{EÖK}$ csúcshalmaz a $\{v_1, v_2, \ldots, v_k\}$ halmaz, amelyben minden G-beli C_i erősen összefüggő komponenshez pontosan egy v_i csúcs tartozik. A $(v_i, v_j) \in E^{EÖK}$ élt behúzzuk, ha G-ben szerepel (x, y) irányított él valamelyik $x \in C_i$ és valamelyik $y \in C_j$ csúcsra. Másképp fogalmazva, ha elhagyjuk azokat a G-beli éleket, amelyek végpontjai ugyanabba az erősen összefüggő komponensbe tartoznak, akkor az eredményül kapott gráf $G^{EÖK}$. A 22.9. ábra (c) részén látható a 22.9. ábra (a) részén szereplő gráf komponens gráfja.

A komponens gráf alapvető tulajdonsága, hogy körmentes gráf, ami a következő lemmából adódik.

22.13. lemma. *Legyen C és C' a $G = (V, E)$ irányított gráf két különböző, erősen összefüggő komponense. Legyen továbbá $u, v \in C$ és $u', v' \in C'$, és tegyük fel, hogy létezik $u \rightsquigarrow u'$ út G-ben. Ekkor nem létezik G-ben $v' \rightsquigarrow v$ út.*

Bizonyítás. Ha létezik G-ben $v \rightsquigarrow v'$ út, akkor létezik $u \rightsquigarrow u' \rightsquigarrow v'$ és $v' \rightsquigarrow v \rightsquigarrow u$ út is G-ben. Ennélfogva u és v' egymásból kölcsönösen elérhető, ami ellentmond annak a feltételnek, hogy C és C' két különböző, erősen összefüggő komponens. ■

Belátjuk, hogy ha a második mélységi keresésben a csúcsokat az első mélységi keresés által meghatározott elhagyási időpontok csökkenő sorrendjében vizsgáljuk, akkor lényegében a komponens gráf csúcsait (amelyek mindegyike egy G-beli erősen összefüggő komponensnek felel meg) a topologikus rendezés sorrendjében járjuk be.

Erősen-összefüggő-komponensek két mélységi keresést használ, ezért fennáll a kétértelműség veszélye, amikor $d[u]$ és $f[u]$ időpontokról beszélünk. Ebben az alfejezetben ezekkel az értékekkel mindig az 1. sorban szereplő, első MK-hívás által meghatározott elérési és elhagyási időpontokra hivatkozunk.

Kiterjesztjük az elérési és elhagyási időpontok értelmezését csúcshalmazokra. Ha $U \subseteq V$, akkor legyen $d(U) = \min_{u \in U}\{d[u]\}$ és $f(U) = \max_{u \in U}\{f[u]\}$, azaz $d(U)$, illetve $f(U)$ az U-beli csúcsokra vonatkozó legkorábbi elérési, illetve legkésőbbi elhagyási időpont.

A következő lemma és következménye megadja az erősen összefüggő komponensek és az első mélységi keresés elhagyási időpontjai közötti kapcsolat alapvető tulajdonságát.

22.14. lemma. *Legyen C és C' a $G = (V, E)$ irányított gráf két különböző, erősen összefüggő komponense. Tegyük fel, hogy létezik olyan $(u, v) \in E$ él, amelyre $u \in C$, és $v \in C'$. Ekkor $f(C) > f(C')$.*

Bizonyítás. Két esetet különböztethetünk meg annak alapján, hogy C és C' erősen összefüggő komponensek közül melyik tartalmazza a mélységi keresés által legkorábban elért csúcsot.

Ha $d(C) < d(C')$, akkor legyen x az elsőnek elért csúcs C-ből. A $d[x]$ időpontban C és C' összes csúcsa fehér. Az x csúcsból C minden csúcsába vezet csak fehér csúcsokat tartalmazó út G-ben. Mivel $(u, v) \in E$, bármely $w \in C'$ csúcsra létezik olyan G-beli út x-ből w-be, $x \rightsquigarrow u \to v \rightsquigarrow w$, amelynek minden csúcsa fehér a $d[x]$ időpontban. A fehér út tétel miatt C és C' összes csúcsa x leszármazottja lesz a mélységi fában. A 22.8 következmény miatt $f(x) = f(C) > f(C')$.

Ha $d(C') < d(C)$, akkor legyen y az elsőnek elért C'-beli csúcs. A $d[y]$ időpontban C' összes csúcsa fehér, és vezet G-ben y-ból minden C'-beli csúcsba csak fehér csúcsokat tartalmazó út. A fehér út tétel miatt C' minden csúcsa y leszármazottja lesz a mélységi fában, továbbá a 22.8. következmény miatt $f[y] = f[C']$. A $d[y]$ időpontban C összes csúcsa fehér. Miután létezik C-ből C'-be vezető (u, v) él, a 22.13. lemmából következik, hogy nem vezethet út C'-ből C-be. Ezért egyetlen C-beli csúcs sem érhető el y-ból. Így $f[y]$ időpontban C minden csúcsa fehér. Tehát bármely $w \in C$ csúcsra $f[w] > f[y]$, amiből adódik, hogy $f(C) > f(C')$. ■

Az alábbi következmény rávilágít arra, hogy G^T minden olyan éle, amely különböző, erősen összefüggő komponenseket köt össze, a korábbi elhagyási időponttal rendelkező komponensből vezet a későbbi elhagyási időpontú komponensbe. (Az időpontok az első mélységi keresés eredményei.)

22.15. következmény. *Legyen C és C' a $G = (V, E)$ irányított gráf két különböző erősen összefüggő komponense. Tegyük fel, hogy létezik olyan $(u, v) \in E^T$ él, hogy $u \in C$ és $v \in C'$. Ekkor $f(C) < f(C')$.*

Bizonyítás. Mivel $(u, v) \in E^T$, ezért $(v, u) \in E$. Ugyanakkor G és G^T erősen összefüggő komponensei megegyeznek, így a 22.14. lemma miatt $f(C) < f(C')$. ■

A 22.15. következmény adja a kulcsot az ERŐSEN-ÖSSZEFÜGGŐ-KOMPONENSEK eljárás működésének megértéséhez. Vizsgáljuk meg, mi történik a második mélységi keresés során, amikor G^T gráfot járjuk be. A legnagyobb $f(C)$ elhagyási időponttal rendelkező C erősen összefüggő komponensből indulunk. A keresés valamilyen $x \in C$ csúcsból indul, és bejárjuk C összes csúcsát. A 22.15. következmény miatt nincs olyan él G^T-ben, amely C-ből egy másik erősen összefüggő komponensbe vezet, ezért az x-ből kiinduló keresés egyetlen más komponensbe tartozó csúcsot sem érint. Így az x gyökerű fa pontosan C csúcsait tartalmazza. A C-be tartozó csúcsok bejárása után, a keresés a 3. sorban egy másik C' erősen összefüggő komponens csúcsát választja gyökérnek. A kiválasztott C' erősen összefüggő komponens $f(C')$ elhagyási időpontja a legnagyobb a maradék, C-n kívüli komponensek közül. A keresés bejárja C' csúcsait, ugyanakkor a 22.15. következmény miatt G^T-ben csak olyan él vezethet ki C'-ből, amely C-beli csúcsba mutat, de ezeket a csúcsokat már előzőleg bejártuk. Általában, amikor a 3. sorban G^T mélységi keresésével egy erősen összefüggő komponens csúcsait járjuk be, akkor a komponensből csak olyan él vezethet ki, amelynek végpontja egy már bejárt komponens csúcsa. Ennélfogva minden mélységi fa egy erősen összefüggő komponensnek felel meg. A következő tételben ezt az okfejtést pontosítjuk.

22.16. tétel. ERŐSEN-ÖSSZEFÜGGŐ-KOMPONENSEK(G) *helyesen állítja elő egy G irányított gráf erősen összefüggő komponenseit.*

Bizonyítás. Az algoritmus 3. sorában G^T mélységi keresése során előállított mélységi fák száma szerinti indukcióval bizonyítjuk, hogy minden egyes fa csúcsai egy-egy erősen összefüggő komponenst alkotnak. Az indukciós feltétel, hogy a 3. sorban előállított első k fa mindegyike erősen összefüggő komponens. Az indukciós feltétel kezdetben, amikor $k = 0$ nyilvánvalóan fennáll.

Az indukciós lépésben feltesszük, hogy a 3. sorban előállított első k mélységi fa mindegyike erősen összefüggő komponens, és a $k+1$.-ként létrehozott fát vizsgáljuk. Legyen a fa gyökere az u csúcs, és az u csúcs tartozzon a C erősen összefüggő komponenshez. A 3. sorban szereplő mélységi keresés gyökerének kiválasztási módja miatt $f[u] = f(C) > f(C')$ fennáll minden olyan C-től különböző C' erősen összefüggő komponensre, amelyet még nem jártunk be. Az indukciós feltétel miatt u elérésének pillanatában C minden csúcsa fehér. A fehér út tétel miatt C összes u-tól különböző csúcsa leszármazottja lesz u-nak az u-hoz tartozó mélységi fában. Továbbá az indukciós feltétel és a 22.15. következmény miatt minden G^T-beli C-ből kivezető él egy már bejárt erősen összefüggő komponensbe vezet. Ezért csak a C erősen összefüggő komponensbe tartozó csúcsok lesznek u leszármazottai G^T mélységi keresése során. Így a G^T-beli, u gyökerű mélységi fa csúcsai pontosan egy erősen összefüggő komponens csúcsainak felelnek meg. Ezzel beláttuk, hogy az indukciós lépés teljesül, és így teljes a bizonyítás. ■

Egy másik lehetséges szemléltetése a második mélységi keresés működésének a következő. Tekintsük a G^T gráf $(G^T)^{EÖK}$ komponens gráfját. Ha a második mélységi keresésben bejárt összes erősen összefüggő komponenshez $(G^T)^{EÖK}$ megfelelő csúcsát rendeljük, akkor $(G^T)^{EÖK}$ csúcsait a topologikus rendezés szerint fordított sorrendben járjuk be. Ha $(G^T)^{EÖK}$ éleinek irányítását megfordítjuk, a $((G^T)^{EÖK})^T$ gráfhoz jutunk. Mivel $((G^T)^{EÖK})^T = G^{EÖK}$ (lásd a 22.5-4. gyakorlatot), a második mélységi keresés $G^{EÖK}$ csúcsait a topologikus rendezésnek megfelelő sorrendben járja be.

Gyakorlatok

22.5-1. Hogyan változhat egy gráf erősen összefüggő komponenseinek száma, ha egy új élt hozzáveszünk a gráfhoz?

22.5-2. Szemléltessük az ERŐSEN-ÖSSZEFÜGGŐ-KOMPONENSEK algoritmus működését a 22.6. ábrán látható gráfon. Pontosabban, adjuk meg az első sorban kiszámított elhagyási időpontokat, és a 3. sorban előállított erdőt. Tegyük fel, hogy MK-BEJÁR 5–7. sorában szereplő ciklus a csúcsokat ábécé sorrendben vizsgálja, és a szomszédsági listák is ábécé szerint rendezettek.

22.5-3. Igaz-e, hogy az erősen összefüggő komponenseket előállító algoritmus egyszerűsíthető, ha az eredeti gráfot használjuk (a transzponált helyett) a második mélységi keresésben, és a csúcsokat az elhagyási időpontok szerint *növekvően* vizsgáljuk?

22.5-4. Lássuk be, hogy tetszőleges G irányított gráf esetén $((G^T)^{EÖK})^T = G^{EÖK}$, azaz G^T komponens gráfjának transzponáltja megegyezik G komponens gráfjával.

22.5-5. Adjunk $O(V + E)$ futási idejű algoritmust egy irányított $G = (V, E)$ gráf komponens gráfjának meghatározására. Ügyeljünk arra, hogy legfeljebb egy él kösse össze az algoritmus által előállított komponens gráf csúcsait.

22.5-6. Adjunk módszert arra, hogy miként lehet egy adott $G = (V, E)$ irányított gráf esetén előállítani azt a $G' = (V, E')$ gráfot, amelyre teljesül, hogy (a) G'-nek ugyanazok az erősen összefüggő komponensei, mint G-nek, (b) G' és G komponens gráfja megegyezik, és (c) E' a lehető legkevesebb élt tartalmazza. Adjunk gyors algoritmust G' meghatározására.

22.5-7. Azt mondjuk, hogy a $G = (V, E)$ irányított gráf ***félig összefüggő,*** ha bármely $u, v \in V$ csúcspár esetén $u \rightsquigarrow v$ vagy $v \rightsquigarrow u$. Adjunk hatékony algoritmust annak eldöntésére, hogy G félig összefüggő-e. Lássuk be az algoritmus helyességét, és elemezzük a futási idejét.

Feladatok

22-1. Élek osztályozása szélességi kereséssel

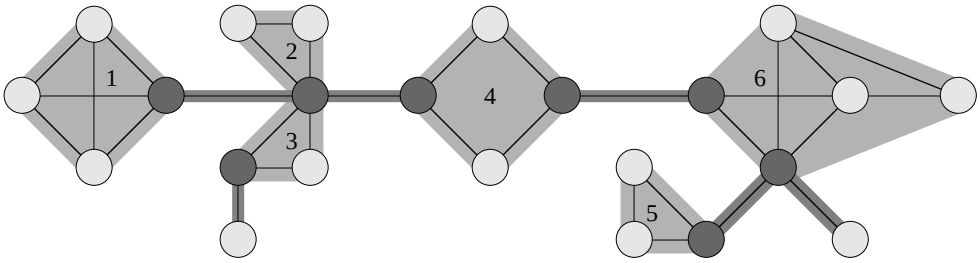
Egy mélységi erdő az élekhez a fa él, visszamutató él, előremutató él és kereszt él osztályokat rendeli. A kezdő csúcsból elérhető élek szélességi fa felhasználásával is ugyanebbe a négy osztályba sorolhatók.

a. Bizonyítsuk be, hogy egy irányítatlan gráf szélességi keresése esetén:

1. Nincs sem visszamutató, sem előremutató él.
2. Minden (u, v) fa élre: $d[v] = d[u] + 1$.
3. Minden (u, v) kereszt élre: $d[v] = d[u]$ vagy $d[v] = d[u] + 1$.

b. Bizonyítsuk be, hogy egy irányított gráf szélességi keresése esetén:

1. Nincs előremutató él.
2. Minden (u, v) fa élre: $d[v] = d[u] + 1$.
3. Minden (u, v) kereszt élre: $d[v] \leq d[u] + 1$.
4. Minden visszamutató élre: $0 \leq d[v] < d[u]$.



22.10. ábra. A 22-2. feladatban használt összefüggő, irányítatlan gráf elvágó pontjai, hídjai és kétszeresen összefüggő komponensei. Az elvágó pontokat sötétszürkére festettük, a hidak a sötétszürke élek, és a kétszeresen összefüggő komponensek éleit a világosabb szürke részek tartalmazzák. A komponensek számozását is megadtuk.

22-2. Elvágó pontok, hidak, blokkok, kétszeresen összefüggő komponensek

Legyen $G = (V, E)$ összefüggő, irányítatlan gráf. G ***elvágó pontja*** egy olyan csúcs, amelyet elhagyva a gráf csúcsai közül a kapott gráf már nem összefüggő. G ***hídja*** egy olyan él, amelyet elhagyva a gráf élei közül a kapott gráf nem lesz összefüggő. G ***blokkja*** egy olyan nem bővíthető élhalmaz, amelynek bármely két eleme egy körön van. (Mivel egy híd nem tartozhat körhöz, ezért minden híd egyelemű blokkot alkot.) A nem egyelemű blokkokat ***kétszeresen összefüggő komponenseknek*** nevezzük. A 22.10. ábrán szemléltetjük ezeket a fogalmakat. Mélységi keresés segítségével meghatározhatjuk az elvágó pontokat, hidakat és kétszeresen összefüggő komponenseket. Legyen $G_\pi = (V, E_\pi)$ a G gráf mélységi fája.

a. Lássuk be, hogy G_π gyökere akkor és csak akkor elvágó pont G-ben, ha legalább két gyereke van G_π-ben.

b. Legyen v egy gyökértől különböző csúcsa G-nek. Bizonyítsuk, hogy v akkor és csak akkor elvágó pont G-ben, ha v-nek van olyan s gyereke, hogy nincs s-ből kiinduló visszamutató él, vagy s bármely leszármazottja v megfelelő őse.

c. Legyen

$$\textit{mély}[v] = \min \begin{cases} d[v], \\ \{d[w] : \quad (u, w) \text{ visszamutató él és } u \text{ leszármazottja } v\text{-nek}\}. \end{cases}$$

Adjunk módszert, amelynek segítségével a *mély*$[v]$ értékek az összes $v \in V$ csúcsra kiszámíthatók $O(E)$ idő alatt.

d. Adjunk módszert az összes elvágó pont $O(E)$ idejű meghatározására.

e. Lássuk be, hogy G egy éle akkor és csak akkor híd, ha nem éle egyetlen egyszerű körnek sem G-ben.

f. Adjunk módszert, amellyel $O(E)$ időben meghatározhatjuk G összes hídját.

g. Mutassuk meg, hogy G blokkjai az E élhalmaz partícióját alkotják.

h. Adjunk $O(E)$ futási idejű algoritmust, amelyben minden G-beli e élhez egy *blokk*$[e]$ pozitív egész számot rendelünk úgy, hogy *blokk*$[e]$ = *blokk*$[e']$ akkor és csak akkor teljesüljön, ha e és e' ugyanahhoz a blokkhoz tartozik. Tegyük meg ugyanezt kétszeresen összefüggő komponensek esetén is.

22-3. Euler-kör

A $G = (V, E)$ összefüggő, irányított gráf ***Euler-köre*** egy olyan kör, amelyben pontosan egyszer szerepel G minden éle. (Egy csúcsot többször is érinthetünk.)

a. Lássuk be, hogy G-ben akkor és csak akkor létezik Euler-kör, ha be-fok(v) = ki-fok(v) minden $v \in V$ csúcsra.

b. Adjunk $O(E)$ futási idejű algoritmust Euler-kör meghatározására, ha létezik. (*Útmutatás.* Vonjuk össze a diszjunkt élhalmazú köröket.)

22-4. Elérhetőség

Legyen $G = (V, E)$ egy irányított gráf, amelyben minden $u \in V$ csúcsot egy $L(u)$ egész számmal címkézünk meg az $\{1, 2, \ldots, |V|\}$ halmazból. Minden $u \in V$ csúcs esetén $R(u) = \{v \in V : u \rightsquigarrow v\}$ legyen az u-ból elérhető csúcsok halmaza. Definiáljuk min(u) csúcsot úgy, hogy az legyen az $R(u)$ halmaz legkisebb címkéjű csúcsa. Azaz min(u) az a v csúcs, amelyre $L(v) = \min\{L(w) : w \in R(u)\}$. Adjunk $O(V + E)$ futási idejű algoritmust, amelyben meghatározzuk a min(u) értékeket minden $u \in V$ csúcsra.

Megjegyzések a fejezethez

Even [87] és Tarjan [292] kitűnő hivatkozási alap a gráfalgoritmusokkal kapcsolatban.

Moore [226] fejlesztette ki a mélységi keresést, miközben az útvesztőkbeli bolyongást tanulmányozta. Ettől függetlenül Lee [198] is felfedezte ugyanezt az algoritmust áramköri kapcsolások vizsgálata során.

Hopcroft és Tarjan [154] javasolta ritka gráfok esetén a szomszédsági listás ábrázolást a csúcsmátrix helyett, és ők ismerték fel először a mélységi keresés jelentőségét. A mélységi keresést széles körben használják az 1950-es évek második felétől kezdődően, különösen a mesterséges intelligencia területén.

Tarjan [289] lineáris idejű algoritmust adott erősen összefüggő komponensek meghatározására. A 22.5. alfejezetben bemutatott algoritmus Aho, Hopcroft és Ullman [6] megoldásán alapul. Ők S. R. Kosajaru (publikálatlan) és Sharir [276] munkáját adják meg forrásként. Gabow [101] is kidolgozott egy algoritmust erősen összefüggő komponensek megkeresésére, ami körök összevonásán alapul, és két verem használatával éri el a lineáris futási időt. Knuth [182] adott először lineáris idejű algoritmust a topologikus rendezés megoldására.

23. Minimális feszítőfák

Elektromos áramkörök tervezésénél gyakori feladat az, amikor elektromosan ekvivalenssé kell tennünk az áramkör bizonyos pontjait. Ahhoz, hogy n darab ilyen pont között kapcsolatot teremtsünk, elég, ha $n-1$ darab alkalmasan megválasztott pontpárt egy-egy vezetékszállal összekötünk. Mivel a pontoknak többféle ilyen összekapcsolása is lehet, kívánatos ezek közül azt megtalálnunk, amelyikhez összességében a legkevesebb (legrövidebb) vezetékre van szükség.

Ezt a huzalozási problémát egy összefüggő, irányítatlan $G = (V, E)$ gráf segítségével modellezhetjük, ahol V-beli csúcsok az áramkör pontjait, E élei az elvileg összeköthető pontpárokat jelölik, továbbá minden $(u, v) \in E$ él rendelkezik egy $w(u, v)$ súllyal, amely az u és v összekötésének költségét (az összekötéshez szükséges vezetékszál hosszát) adja meg. Célunk az, hogy megtaláljuk azt a körmentes $T \subseteq E$ részhalmazt, amelynek segítségével az összes pont összekapcsolható, és amelynek a

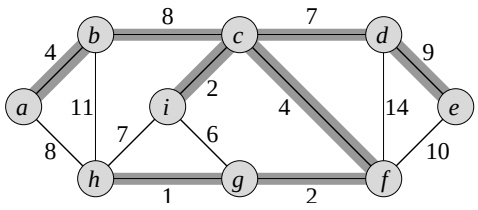
$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

teljes súlya a lehető legkisebb. A körmentes és minden csúcspontot összekapcsoló T nyilvánvalóan egy fa, amelyik „átfogja", „kifeszíti" a G gráfot, ezért ***feszítőfának*** nevezzük. A T fa meghatározásának problémáját ***minimális feszítőfa problémának***[1] hívjuk. A 23.1. ábra egy összefüggő gráfra és annak minimális feszítőfájára mutat példát.

Ebben a fejezetben két, a minimális feszítőfa problémát megoldó algoritmust mutatunk be: Kruskal és Prim algoritmusát. Mindkettő futási ideje közönséges bináris kupacok használata mellett $O(E \lg V)$. Fibonacci-kupacok alkalmazásával a Prim-algoritmus futási ideje $O(E + V \lg V)$-re csökkenthető, amely azonban csak abban az esetben jelent javulást, ha $|V|$ sokkal kisebb, mint $|E|$.

Ez a két algoritmus a 16. fejezetben vázolt mohó algoritmusok közé tartozik. Általában egy algoritmus minden lépésben több lehetőség közül választ. A mohó stratégia ezek közül azt részesíti előnyben, amelyik az adott pillanatban a legjobbnak látszik. Az optimális megoldás megtalálását ez a stratégia általában nem garantálja. A minimális feszítőfa problémánál azonban bebizonyítható, hogy bizonyos mohó stratégiák minimális súlyú feszítőfához vezetnek. A most bemutatásra kerülő mohó módszerek a 16. fejezetben ismertetett

[1] A minimális feszítőfa kifejezés egy rövidített formája a minimális súlyú feszítőfa megnevezésnek. A T-beli élek számát például nem is minimalizálhatnánk, hiszen a B.2. tétel miatt minden feszítőfa pontosan $|V| - 1$ élből áll.



23.1. ábra. Egy összefüggő gráf minimális feszítőfája. Az élek súlyait az ábrán feltüntettük. A minimális feszítőfa éleit vastag vonalak jelölik. A fa teljes súlya 37. Nem ez az egyetlen minimális feszítőfa: cseréljük ki a (b, c) élt az (a, h) élre, és egy másik 37 súlyú feszítőfát kapunk.

elmélet klasszikus alkalmazásai, amelyeket azonban az említett fejezet elolvasása nélkül is megérthetünk.

A 23.1. alfejezetben egy olyan „általános” minimális feszítőfa algoritmust mutatunk be, amelyik fokozatosan, újabb és újabb élek hozzáadásával állítja elő a feszítőfát. A 23.2. alfejezetben ennek az általános algoritmusnak kétféle megvalósítását adjuk meg. Az első, Kruskaltól származó algoritmus, a 21.1. alfejezet összefüggő-komponensek algoritmusához hasonlít. A Prim nevével fémjelzett második algoritmus Dijkstra legrövidebb-utak algoritmusával (24.3. alfejezet) mutat közös vonásokat.

23.1. Minimális feszítőfa növelése

Egy minimális feszítőfát szeretnénk találni egy összefüggő, irányítatlan, $w : E \to \mathbf{R}$ súlyfüggvénnyel élsúlyozott $G = (V, E)$ gráfban. Az a két, mohó stratégiára épülő algoritmus, amelyet e probléma megoldására alkalmazunk ebben a fejezetben, abban különbözik egymástól, hogy eltérő módon alkalmazza ezt a stratégiát.

Ez a mohó stratégia azon az „általános” algoritmuson keresztül érthető meg, amely fokozatosan, újabb és újabb élek hozzáadásával növeli a minimális feszítőfát. Az algoritmus az éleknek azt az A-val jelölt halmazát kezeli, amelyre az alábbi invariáns állítás teljesül:

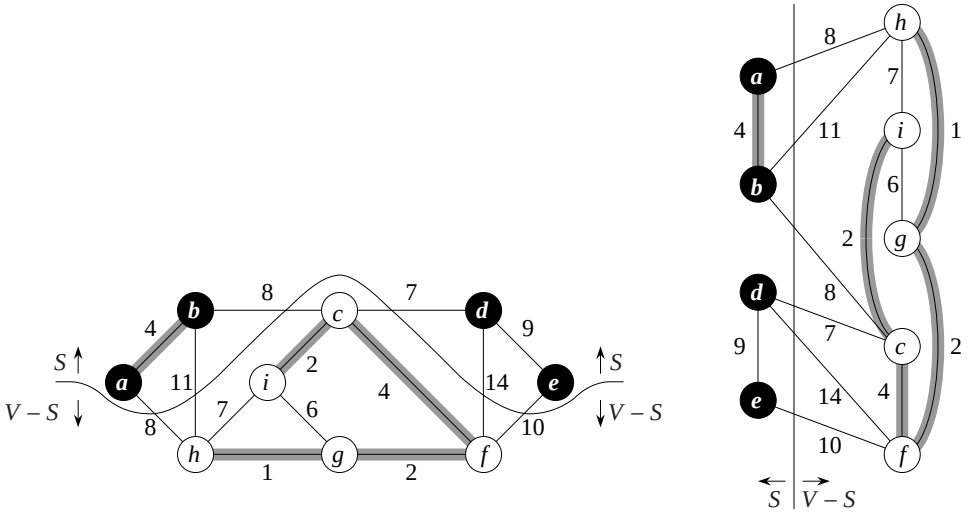
Az iterációk előtt az A valamelyik minimális feszítőfának a részhalmaza.

Az algoritmus minden lépésben azt az (u, v) élt határozza meg, amelyiket az A-hoz téve, továbbra is fennáll az előbbi invariáns állítás, miszerint $A \cup \{(u, v)\}$ is egy részhalmaza az egyik minimális feszítőfának. Egy ilyen élt A-ra nézve ***biztonságos élnek*** hívunk, mivel A-hoz történő hozzávétele biztosan nem rontja el az A-ra vonatkozó invariáns állítást.

```
MFF(G, w)
1  A ← ∅
2  while az A nem feszítőfa
3        do keresünk az A-ra nézve egy (u, v) biztonságos élt
4           A ← A ∪ {(u, v)}
5  return A
```

A ciklusinvariánst az alábbiak szerint használjuk:

Teljesül: Az 1. sor után az A magától értetődően elégíti ki a ciklusinvariánst.



23.2. ábra. Kétféleképpen mutatjuk be a 23.1. ábra gráfjának egy $(S, V-S)$ vágását. **(a)** Fekete színnel jelöltük az S halmazbeli, és fehérrel a $V-S$ halmazbeli csúcsokat. A vágást keresztező élek fehér csúcsokat kötnek össze feketékkel. A (d,c) él a vágást keresztező egyetlen könnyű él. Az élek egy A részhalmazát megvastagított vonalakkal jelöltük; a $(S, V-S)$ vágás elkerüli az A-t, mivel A egyik éle sem keresztezi a vágást. **(b)** Az előbbi gráfnak S-beli csúcsait a bal oldalon, $V-S$-beli csúcsait a jobb oldalon helyeztük el. Mindegyik vágást keresztező él egy bal oldali csúcsot egy jobb oldali csúccsal köt össze.

Megmarad: A 2–4. sorok ciklusa kizárólag biztonságos élekkel bővíti az A-t, így az invariáns továbbra is fenn áll.

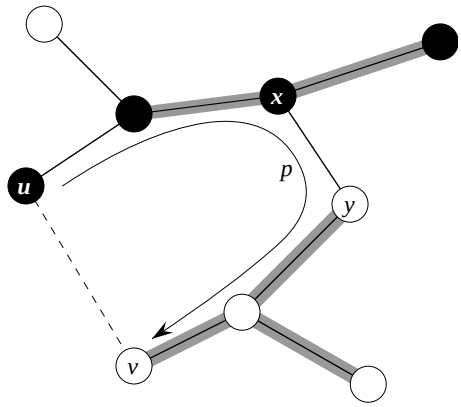
Befejeződik: Minden A-hoz hozzávett él egy minimális feszítőfa része, ezért az 5. sorban visszaadott A halmaz egy minimális feszítőfa.

A trükkös rész természetesen a biztonságos él 3. sorban történő kiválasztása. Ilyen él biztosan létezik, hiszen a 3. sor végrehajtásakor az invariáns szerint van olyan T feszítőfa, hogy $A \subseteq T$. A **while** magjában az A egy valódi részhalmaza a T-nek, és ezért van olyan $(u,v) \in T$ él, amelyre $(u,v) \notin A$ és az (u,v) él biztonságos A-ra nézve.

A továbbiakban egy szabályt (23.1. tétel) adunk a biztonságos élek felismerésére. A következő alfejezetben azt a két algoritmust mutatjuk be, amelyek ezt a szabályt hatékonyan alkalmazzák.

Először néhány fogalmat vezetünk be. A $G = (V,E)$ irányítatlan gráf egy $(S, V-S)$ ***vágásának*** a V-nek egy kettéosztását nevezzük. A 23.2. ábra erre mutat egy példát. Azt mondjuk, hogy egy $(u,v) \in E$ él ***keresztezi*** az $(S, V-S)$ vágást, ha annak egyik végpontja S-ben, a másik $V-S$-ben található. Egy vágás ***elkerül*** egy A halmazt, ha az A egyetlen éle sem keresztezi a vágást. Egy vágást keresztező él ***könnyű él,*** ha a vágást keresztező élek közül neki van a legkisebb súlya. Megjegyezzük, hogy egynél több vágást keresztező könnyű él is lehet egyszerre. Általánosabban úgy is fogalmazhatunk, hogy egy él egy adott tulajdonságot kielégítő ***könnyű él,*** ha az adott tulajdonságot kielégítő élek között neki van a legkisebb súlya.

A biztonságos élt felismerő szabályt a következő tételből nyerjük.



23.3. ábra. A 23.1. tétel bizonyítása. Az S-beli csúcsok feketék, a $(V - S)$-beliek fehérek. Az ábrán csak a T minimális feszítőfa élei láthatók, a G gráf élei nem. Az A-beli éleket vastagított vonalak jelölik, és (u, v) egy könnyű él az $(S, V - S)$ vágásban. Az (x, y) él az egyetlen u-t v-vel összekötő T-beli p úton található. Az (u, v) élt tartalmazó T' minimális feszítőfát a T-ből kapjuk úgy, hogy eltávolítjuk belőle az (x, y) élt, és hozzávesszük az (u, v) élt.

23.1. tétel. *Legyen $G = (V, E)$ egy összefüggő, irányítatlan, a $w : E \to \mathbf{R}$ függvénnyel élsúlyozott gráf. Legyen A egy olyan részhalmaza E-nek, amelyik G valamelyik minimális feszítőfájának is része. Legyen $(S, V - S)$ tetszőleges A-t elkerülő vágása a G-nek. Legyen (u, v) az $(S, V - S)$ vágást keresztező könnyű él. Ekkor az (u, v) él biztonságos az A-ra nézve.*

Bizonyítás. Legyen T egy A-t tartalmazó minimális feszítőfa. Feltehetjük, hogy az (u, v) könnyű él nincs benne a T-ben, mert ha benne van, akkor máris készen vagyunk a bizonyítással. Kivág-beilleszt technikával meg fogjuk szerkeszteni azt a T' minimális feszítőfát, amelyik az $A \cup \{(u, v)\}$-t tartalmazza, ami azt mutatja, hogy az (u, v) él az A-ra nézve biztonságos.

Az (u, v) él az u és v csúcsokat T-ben összekötő p úttal együtt egy kört alkot, mint azt a 23.3. ábrán láthatjuk. Mivel u és v az $(S, V - S)$ vágás ellentétes oldalán helyezkedik el, kell lenni a p úton legalább egy olyan élnek, ami szintén keresztezi a vágást. Legyen egy ilyen él az (x, y). Az (x, y) nincs A-ban, mert A-t elkerüli a vágás. Mivel (x, y) az u-t és v-t összekötő egyetlen T-beli úton van, az (x, y)-t eltávolítva a T két komponensre esik szét. Az (u, v) hozzáadása azonban újra összekapcsolja ezt a két komponenst, és a $T' = T - \{(x, y)\} \cup \{(u, v)\}$ egy új feszítőfa lesz.

Most megmutatjuk, hogy T' egy minimális feszítőfa. Mivel (u, v) egy könnyű él az $(S, V - S)$ vágást keresztező élek között, és (x, y) szintén keresztezi ezt a vágást, a $w(u, v) \leq w(x, y)$ fennáll. Ennek megfelelően

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T).$$

De T egy minimális feszítőfa, azaz $w(T) \leq w(T')$; így T'-nek is egy minimális feszítőfának kell lennie.

Már csak az maradt hátra, hogy megmutassuk, az (u, v) él biztonságos A-ra nézve. Tudjuk, hogy $A \subseteq T'$, hiszen $A \subseteq T$ és $(x, y) \notin A$; ezért $A \cup \{(u, v)\} \subseteq T'$. Következésképpen, mivel T' egy minimális feszítőfa, az (u, v) él biztonságos A-ra nézve. ■

A 23.1. tétel az általános MFF algoritmus jobb megértését segíti. Ennek végrehajtása során a $G = (V, E)$ összefüggő gráf éleiből felépített A mindig körmentes; máskülönben az A-t tartalmazó minimális feszítőfa is kört tartalmazna, ami nem lehetséges. A végrehajtás bármelyik pillanatában a $G_A = (V, A)$ gráf egy erdő, és a G_A minden összefüggő komponense fa. (Lehet, hogy némelyik fa csak egyetlen csúcsot tartalmaz, mint például abban az esetben, amikor az algoritmus elindul. Ekkor A üres, és az erdő $|V|$ darab egyetlen csúcsot tartalmazó fából áll.) Ezenkívül bármelyik A-ra nézve biztonságos (u, v) él G_A-nak két különböző komponensét köti össze, mivel $A \cup \{(u, v)\}$-nek körmentesnek kell lennie.

Az általános MFF algoritmus 2–4. sorainak ciklusa $|V| - 1$-szer hajtódik végre; minden lépésben egy újabb élét határozza meg a $|V| - 1$ élt tartalmazó minimális feszítőfának. Kezdetben, amikor $A = \emptyset$, $|V|$ darab fa van G_A-ban, és minden iteráció eggyel csökkenti ezek számát. Amikor az erdő már csak egyetlen fából áll, az algoritmus leáll.

A 23.2. alfejezetben szereplő két algoritmus a 23.1. tétel alábbi következményére támaszkodik.

23.2. következmény. *Legyen $G = (V, E)$ egy összefüggő, irányítatlan, a $w : E \to \mathbf{R}$ függvénnyel élsúlyozott gráf. Legyen A olyan részhalmaza E-nek, amelyik G valamelyik minimális feszítőfájának is része. Legyen C egy összefüggő komponens a $G_A = (V, A)$ erdőben. Ha (u, v) a C-t és a G_A valamely másik komponensét összekötő könnyű él, akkor (u, v) biztonságos az A-ra nézve.*

Bizonyítás. A $(C, V - C)$ elkerüli az A-t, és (u, v) egy könnyű él ebben a vágásban. Ezért az (u, v) biztonságos az A-ra nézve. ■

Gyakorlatok

23.1-1. Legyen (u, v) minimális súlyú él a G gráfban. Mutassuk meg, hogy (u, v) hozzátartozik a G valamelyik minimális feszítőfájához.

23.1-2. Sabatier professzor feltételezi a 23.1. tétel alábbi megfordítását. Legyen $G = (V, E)$ egy összefüggő, irányítatlan, a $w : E \to \mathbf{R}$ függvénnyel súlyozott gráf. Legyen A egy olyan részhalmaza E-nek, amelyik G valamelyik minimális feszítőfájának is része. Legyen $(S, V - S)$ egy tetszőleges A-t elkerülő vágása a G-nek. Legyen (u, v) egy biztonságos él az A-ra nézve az $(S, V - S)$ vágásban. Ekkor az (u, v) él könnyű az $(S, V - S)$ vágásban. Mutassuk meg egy számpéldán keresztül, hogy a professzor feltételezése helytelen.

23.1-3. Mutassuk meg, hogy ha az (u, v) él valamelyik minimális feszítőfához tartozik, akkor van olyan vágás, amelyben ez könnyű él.

23.1-4. Adjunk egyszerű példát olyan gráfra, amelyben azon élek halmaza, amelyek könnyűek valamely vágásban, nem alkot minimális feszítőfát.

23.1-5. Legyen e maximális súlyú él egy összefüggő $G = (V, E)$ gráf valamelyik körén. Bizonyítsuk be, hogy a $G' = (V, E - \{e\})$ gráfnak van olyan minimális feszítőfája, ami G-nek is minimális feszítőfája.

23.1-6. Mutassuk meg, hogy ha egy gráfnak bármelyik vágásában csak egyetlen könnyű él található, akkor a gráfnak egyetlen minimális feszítőfája van. Adjunk ellenpéldát arra, hogy az állítás visszafelé nem igaz.

23.1-7. Igaz-e, hogy ha egy gráf összes élsúlya pozitív, akkor az éleknek bármelyik olyan részhalmaza, amelyik az összes csúcsot összekapcsolja, és minimális összsúlyú, biztosan fát alkot? Adjunk példát, ahol az állítás nem teljesül nempozitív súlyok előfordulása esetén.

23.1-8. Legyen T minimális feszítőfa G-ben, és legyen L a T-beli élsúlyok rendezett listája. Mutassuk meg, hogy a G-nek bármely másik T' minimális feszítőfájára a T' élsúlyainak rendezett listája megegyezik az L listával.

23.1-9. Legyen T minimális feszítőfa G-ben, és legyen V' a V részhalmaza. Legyen T' a T-nek V' által meghatározott részgráfja, és legyen G' a G-nek V' által meghatározott részgráfja. Mutassuk meg, hogy ha T' összefüggő, akkor T' minimális feszítőfa G'-ben.

23.1-10. Legyen T minimális feszítőfa G-ben, és tegyük fel, hogy egy T-beli élnek csökkentjük a súlyát. Mutassuk meg, hogy T még ezután is minimális feszítőfa lesz G-ben. Formálisan: legyen T egy minimális feszítőfa abban a G-ben, amelynek súlyfüggvénye w. Válasszunk egy $(x, y) \in T$ élt és egy pozitív k számot, majd definiáljuk az alábbi w' súlyfüggvényt:

$$w'(u,v) = \begin{cases} w(u,v), & \text{ha } (u,v) \neq (x,y), \\ w(x,y) - k, & \text{ha } (u,v) = (x,y). \end{cases}$$

Mutassuk meg, hogy T egy minimális feszítőfa abban a G gráfban, amelynek súlyfüggvénye a w'.

23.1-11.⋆ Legyen T minimális feszítőfa G-ben, és tegyük fel, hogy egy T-n kívül eső élnek csökkentjük a súlyát. Adjunk algoritmust, amely az így módosított gráfban megtalálja a minimális feszítőfát.

23.2. Kruskal és Prim algoritmusai

Ebben az alfejezetben a minimális feszítőfát előállító általános algoritmus két speciális változatát mutatjuk be. Mindkettő sajátos szabályt alkalmaz a biztonságos él meghatározására az általános MFF 3. sorában. Kruskal algoritmusában az A halmaz egy erdő. Az A-hoz hozzávett biztonságos él mindig az erdő két különböző komponensét összekötő legkisebb súlyú él. Prim algoritmusában az A egyetlen fa. Az A-hoz hozzáadott biztonságos él mindig a legkisebb súlyú olyan él, amelyik egy A-beli és egy A-n kívüli csúcsot köt össze.

Kruskal-algoritmus

Kruskal algoritmusa közvetlenül a 23.1. alfejezetben megadott általános MFF-algoritmuson alapul. Minden lépésben megkeresi a $G_A = (V, A)$ erdő két tetszőleges komponensét összekötő élek közül a legkisebb súlyú (u, v) élt, és ezt veszi hozzá az egyre bővülő erdőhöz. Legyen C_1 és C_2 az erdőnek az a két fája, amit az (u, v) él összeköt. Mivel (u, v) egyben a legkisebb súlyú olyan él is, amelyik C_1-et valamelyik másik komponenssel összeköti, a 23.2. következmény miatt (u, v) biztonságos C_1-re nézve. A Kruskal-algoritmus egy mohó algoritmus, mert minden lépésben a lehetséges legkisebb súlyú élt adja hozzá az erdőhöz.

Az általunk megvalósított Kruskal-algoritmus a 21.1. alfejezet összefüggő komponenseket előállító algoritmusához hasonlít. Az elemek diszjunkt halmazainak kezelésére egy diszjunkt-halmaz adatszerkezetet használ. Mindegyik halmaz az aktuális erdő egy fájának csúcsait tartalmazza. A HALMAZT-KERES(u) az u csúcsot tartalmazó halmazt jelölő elemet ad meg. Az, hogy az u és a v csúcsok ugyanahhoz a fához tartoznak-e, úgy határozható meg, hogy ellenőrizzük a HALMAZT-KERES(u) és HALMAZT-KERES(v) egyenlőségét. Két fa összekapcsolását az EGYESÍT eljárás végzi.

MFF-KRUSKAL(G, w)

```
1  A ← ∅
2  for minden v ∈ V[G]-re
3      do HALMAZT-KÉSZÍT(v)
4  rendezzük az E éleit a w súly szerint növekvő sorrendben
5  for minden (u, v) ∈ E élre, az élek súly szerint növekvő sorrendjében
6      do if HALMAZT-KERES(u) ≠ HALMAZT-KERES(v)
7           then A ← A ∪ {(u, v)}
8                EGYESÍT(u, v)
9  return A
```

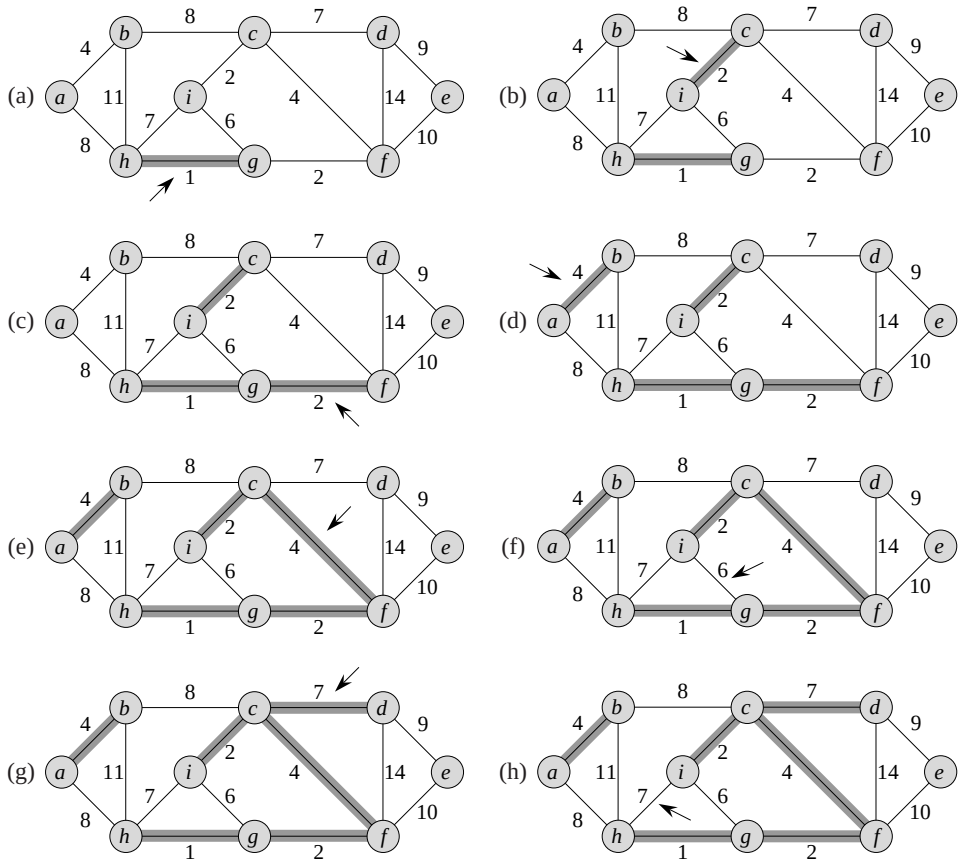
Kruskal algoritmusának működését a 23.4. ábra mutatja be. Az 1–3. sorban üres kezdőértéket kap az A halmazt, létrejön a gráf csúcsait külön-külön tartalmazó $|V|$ darab fa. A 4. sor súlyuk szerint növekvő sorba rendezi az E éleit. Az 5–8. sorok **for** ciklusa minden (u, v) élre megvizsgálja, hogy annak végpontjai ugyanahhoz a fához tartoznak-e. Ha igen, akkor ezt az élt nem lehet az erdőhöz hozzávenni anélkül, hogy kör alakulna ki, ezért nem kell vele foglalkozni. Ellenkező esetben a két csúcs különböző fákhoz tartozik. Ekkor az (u, v) élt a 7. sorban hozzávesszük az A-hoz, majd a két fa csúcsait a 8. sorban összevonjuk.

A $G = (V, E)$ gráfon működő Kruskal-algoritmus futási ideje a diszjunkt-halmaz megvalósításától függ. Vegyük a 21.3. alfejezetben megvalósított diszjunkt-halmaz erdőt a „rang szerinti egyesítés” és „úttömörítés” heurisztikákkal, mivel ez az aszimptotikusan leggyorsabbnak ismert megvalósítás. Ekkor az 1. sor kezdeti értékadása $O(1)$ ideig tart, a 4. sor rendezése pedig $O(E \lg E)$ idejű. (Mindjárt kitérünk a 2–3. sorbeli **for** ciklus $|E|$ darab HALMAZT-KÉSZÍT műveletének futási idejére is.) Az 5–8. sorok **for** ciklusa $O(E)$ HALMAZT-KERES és EGYESÍT műveletet tartalmaz a diszjunkthalmaz erdőre. Ezek teljes futási ideje a $|E|$ HALMAZT-KÉSZÍT művelettel együtt $O((V + E)\alpha(V))$, ahol α a 21.4. alfejezetben definiált nagyon lassan növekedő függvény. Mivel a G-ről feltettük, hogy összefüggő, fennáll, $|E| \geq |V|1$, és így a diszjunkthalmaz műveletek $O((E)\alpha(V))$ idejűek. Továbbá, mivel $\alpha(|V|) = O(\lg E)$, a Kruskal-algoritmus teljes futási ideje $O(E \lg E)$. Az $|E| < |V|^2$ mellett azt kapjuk, hogy $\lg |E| = O(\lg V)$, és ekkor a Kruskal-algoritmus teljes futási ideje $O(E \lg V)$ lesz.

Prim-algoritmus

Prim algoritmusa, akárcsak Kruskalé, a 23.1. alfejezet általános MFF algoritmusának speciális változata. A Prim-algoritmus működése nagyon hasonlít Dijkstra legrövidebb utakat kereső algoritmusáéhoz (lásd 24.3. alfejezet). Prim algoritmusában az A halmaz élei mindig egyetlen fát formáznak. A 23.5. ábrán is látható, hogy a fa építése egy tetszőlegesen kiválasztott r gyökérpontból indul, és addig növekszik, amíg a V összes csúcsa nem kerül be a fába. Minden lépésben azt a könnyű élt vesszük hozzá az A fához, amelyik egy A-beli csúcsot köt össze a $G_A = (V, A)$ egy izolált csúcsával. A 23.2. következmény szerint ez a szabály biztonságos élt ad A-ra nézve; így amikor az algoritmus befejeződik az A-beli élek minimális feszítőfát alkotnak. Ez a stratégia mohó, mivel minden lépésben egy olyan éllel bővíti a fát, amely a lehető legkisebb mértékben növeli a fa teljes súlyát.

A Prim-algoritmus hatékony megvalósításának kulcsa az A-hoz hozzáadandó új él kiválasztásának ügyes megvalósításán múlik. Az alább megadott pszeudokód bemenő adatai



23.4. ábra. Kruskal algoritmusának működése a 23.01. ábra gráfján. A vastagított vonalakkal jelölt élek az egyre bővülő A erdőhöz tartoznak. Az algoritmus az éleket súlyuk szerint rendezett sorrendben vizsgálja meg. Egy nyíl mutat arra az élre, amellyel az adott lépésben az algoritmus dolgozik. Ha ez két különböző fát köt össze az erdőben, akkor ezt hozzávesszük az erdőhöz, miközben a két fát összevonjuk.

az összefüggő G gráf és az előállítandó minimális feszítőfa r gyökere. A fában még nem szereplő csúcsok a végrehajtás közben egy *kulcs* értéken alapuló Q elsőbbségi sorban helyezkednek el. A *kulcs*[v] a vt valamelyik fabeli csúccsal összekötő minimális súlyú él súlya; amennyiben nincs ilyen él, akkor megállapodás szerint *kulcs*[v] = ∞. A $\pi[v]$ érték a v csúcs fabeli szülőjét adja meg. Az általános MFF A halmazát a Prim-algoritmus implicit módon az

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$$

képlettel írja le. Amikor az algoritmus befejeződik, a Q elsőbbségi sor üres; a G-beli A minimális feszítőfa az

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}$$

halmaz lesz.

MFF-PRIM(G, w, r)

```
 1  for minden v ∈ V[G]re
 2      do kulcs[v] ← ∞
 3         π[u] ← NIL
 4  kulcs[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7        do u ← KIVESZ-MIN(Q)
 8           for minden v ∈ Szomszéd[u]-ra
 9               do if v ∈ Q and w(u, v) <kulcs[v]
10                     then π[v] ← u
11                          kulcs[v] ← w(u, v)
```

A Prim-algoritmus működését a 23.5. ábrán követhetjük nyomon. Az 1–5. sorok beállítják a csúcsoknál *kulcs* értéket ∞-re (kivéve az r gyökérpontot, amelyik *kulcs* értéke 0 lesz, és ezáltal ő lesz az elsőként feldolgozott csúcs), a szülő értéket NIL-re, és elhelyezik az összes csúcsot a Q minimalizált elsőbbségi sorban. Az algoritmus által karbantartott invariáns állítás a következő három részből tevődik össze:

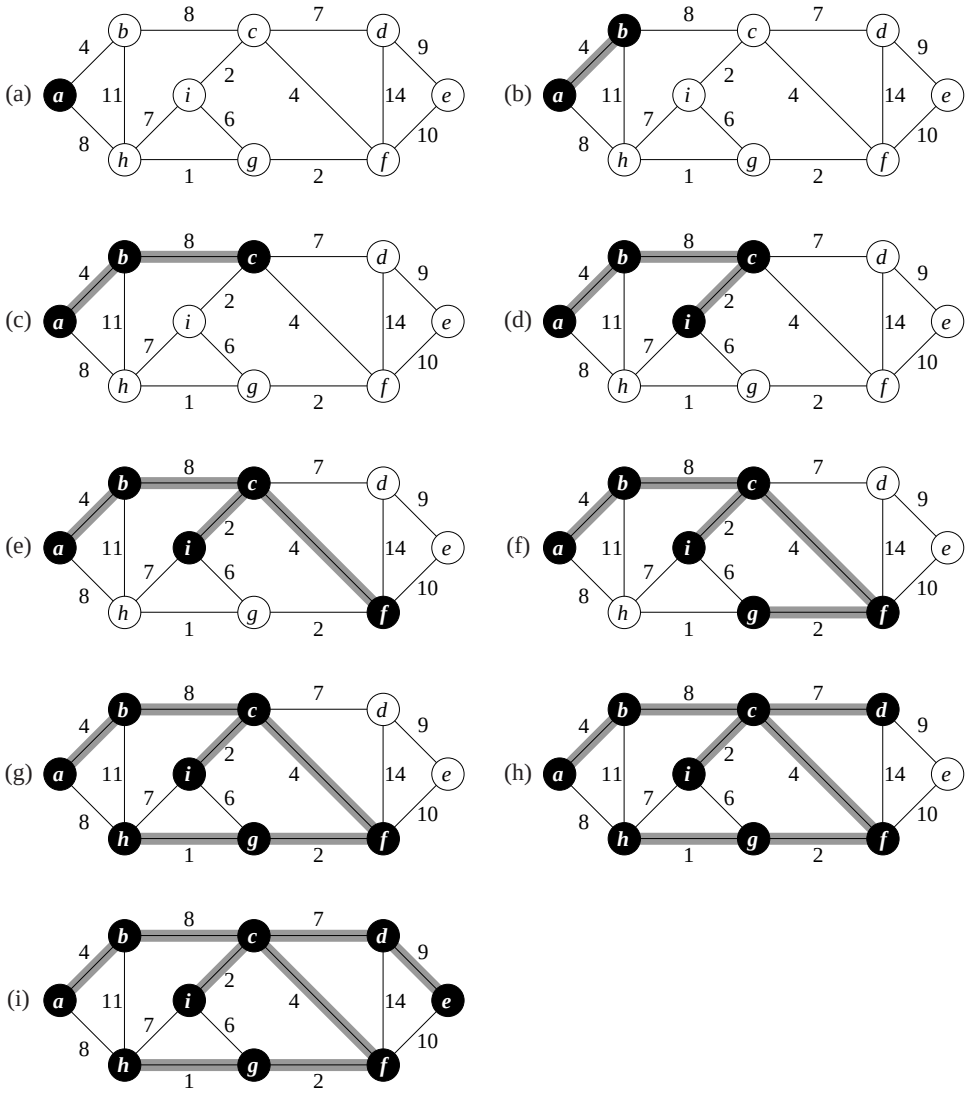
A 6–11. sorok **while** ciklus minden iterációja előtt

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
2. A minimális feszítőfában elhelyezett csúcsok a $V - Q$ halmazban vannak.
3. Minden $v \in Q$ csúcsra, ha $\pi[v] \neq$ NIL, akkor $key[v] < \infty$ és a $key[v]$ egy olyan $(u, \pi[v])$ könnyű él súlya, amelyik a v-t a minimális feszítőfában már elhelyezett valamelyik u csúccsal köti össze.

A 7. sor meghatározza azt az $u \in Q$ csúcsot, amely a $(V - Q, Q)$ vágás egyik könnyű élének végpontja (kivéve az első iterációt, ahol a 4. sornak megfelelően $u = r$). Eltávolítva az u-t a Q halmazból, az átkerül a fa csúcsait tartalmazó $V - Q$ halmazba, ezáltal az $(u, \pi[u])$ éllel módosul az A. A ciklusinvariáns harmadik részét a 8–11. sorok **for** ciklusa biztosítja úgy, hogy minden fán kívüli u-val szomszédos csúcsra időszerűsíti a *kulcs* és π értékeket.

A Prim-algoritmus hatékonysága a Q elsőbbségi sor megvalósításán múlik. Ha Q-t bináris minimum-kupacként (lásd 6. fejezet) valósítjuk meg, akkor a KUPACOT-ÉPÍT eljárással az 1–5. sorok kezdeti értékadásait $O(V)$ időben elvégezhetjük. A **while** ciklus magja $|V|$-szer kerül végrehajtásra, és mivel a KIVESZ-MIN művelet $O(\lg V)$ idejű, a KIVESZ-MIN művelet összes meghívása $O(V \lg V)$ ideig tart. A 8–11. sorok **for** ciklusának végrehajtási ideje $O(E)$, mivel a szomszédsági listák hosszainak összege $2|E|$. A **for** cikluson belül a 9. sorban egy elem Q-hoz tartozásának vizsgálata konstans időben történik, ha minden csúcsnál fenntartunk egy bitet annak jelzésére, hogy a csúcs Q-beli-e, vagy sem. Ezt akkor kell megváltoztatnunk, amikor a csúcs kikerül a Q-ból. A 11. sor értékadása magában hordozza a KULCSOT-CSÖKKENT művelet végrehajtását a kupacra, amelyet bináris kupac esetén $O(\lg V)$ időben lehet megvalósítani. Így a Prim-algoritmus teljes futási ideje $O(V \lg V + E \lg V) = O(E \lg V)$, amelyik aszimptotikusan egyenlő a Kruskal-algoritmusra adott megvalósítás futási idejével.

A Prim-algoritmus aszimptotikus futási ideje azonban Fibonacci-kupacok segítségével javítható. A 20. fejezetben láthatjuk, hogy ha $|V|$ darab elemet Fibonacci-kupacba szervezünk, akkor egy KIVESZ-MIN művelet futási ideje $O(\lg V)$-re csökkenthető, és a (11. sor meg-



23.5. ábra. Prim algoritmusának működése a 23.1. ábra gráfján. A gyökérpont a. A vastagított vonalakkal jelölt élek és a feketére színezett csúcsok az egyre bővülő fát mutatják. A fa csúcsai az algoritmus minden lépésében egy vágást határoznak meg, és ezt a vágást keresztező könnyű élt vesszük hozzá a fához. Például a második lépésben az algoritmus a (b, c) vagy az (a, h) élek valamelyikét adja hozzá a fához, mivel mindkettő könnyű éle a vágásnak.

valósításában szereplő) KULCSOT-CSÖKKENT művelet ideje $O(1)$ lesz. Mindent egybevetve a Fibonacci-kupaccal megvalósított elsőbbségi sort használó Prim-algoritmus futási ideje $O(E + V \lg V)$.

Gyakorlatok

23.2-1. Kruskal algoritmusa ugyanazon G gráf esetén eltérő feszítőfákat eredményezhet attól függően, hogy az azonos súlyú éleket hogyan rendeztük sorba. Mutassuk meg, hogy

minden T minimális feszítőfához megadható olyan rendezés, hogy az algoritmus éppen a T-t állítja elő.

23.2-2. Tegyük fel, hogy a $G = (V, E)$ gráfot egy szomszédsági mátrixban ábrázoljuk. Adjunk ekkor $O(V^2)$ futási idejű megvalósítást Prim algoritmusára.

23.2-3. Vajon aszimptotikusan gyorsabb-e a Prim-algoritmus Fibonacci-kupacos megvalósítása a bináris kupacos változatnál ritka $G = (V, E)$ gráf esetén, ahol $|E| = \Theta(V)$? Mit mondhatunk a sűrű gráfokról, ahol $|E| = \Theta(V^2)$? Milyen kapcsolatnak kell lenni a $|E|$ és $|V|$ között ahhoz, hogy a Fibonacci-kupacos megvalósítás aszimptotikusan gyorsabb legyen a bináris kupacos változatnál?

23.2-4. Tegyük fel, hogy egy gráf súlyai 1 és $|V|$ közé eső egész számok. Milyen gyorssá tehető a Kruskal-algoritmus? Mit mondhatunk arról az esetről, amikor a súlyok 1 és W közé esnek, ahol W állandó?

23.2-5. Tegyük fel, hogy egy gráf súlyai 1 és $|V|$ közé eső egész számok. Milyen gyorssá tehető a Prim-algoritmus? Mit mondhatunk arról az esetről, amikor a súlyok 1 és W közé esnek, ahol W állandó?

23.2-6.⋆ Tegyük fel, hogy egy gráf élsúlyai egyenletesen helyezkednek el a $[0, 1)$ félig nyílt intervallumon. Kruskal vagy Prim algoritmusa a gyorsabb ilyenkor?

23.2-7.⋆ Tegyük fel, hogy a G gráf minimális feszítőfáját már előállítottuk. Milyen gyorsan lehet módosítani ezt a fát, ha G-hez egy új csúcsot és ahhoz kapcsolódó éleket veszünk hozzá?

23.2-8. Toole professzor az alábbi új oszd-meg-és-uralkodj stratégiájú algoritmust tervezi minimális feszítőfák előállítására. Adva van egy $G = (V, E)$ gráf, amelyben a csúcsok V halmazát két részre, V_1 és V_2 halmazokra vágjuk úgy, hogy a $|V_1|$ és $|V_2|$ különbsége legfeljebb egy. Legyen E_1 azon élek halmaza, amelyek V_1-beli csúcsokat kötnek össze, E_2 pedig azon élek halmaza, amelyek végpontjai V_2-beli csúcsok. Oldjuk meg rekurzív módon a minimális feszítőfa problémát mindegyik $G_1 = (V_1, E_1)$ és $G_2 = (V_2, E_2)$ részgráfra. Ezután keressük meg azt a legkisebb súlyú E-beli élet, amelyik keresztezi a V_1, V_2 vágást, és ezzel az éllel kössük össze az előbb talált minimális feszítőfákat.

Vagy mutassuk meg, hogy az így kapott feszítőfa minimális a G-ben, vagy adjunk ellenpéldát arra, amikor nem az.

Feladatok

23-1. Második legjobb feszítőfa

Legyen $G = (V, E)$ irányítatlan, összefüggő gráf a $w : E \to \mathbf{R}$ súlyfüggvénnyel, és tegyük fel, hogy $|E| \geq |V|$, és az élsúlyok mind különböznek.

A második legjobb feszítőfát az alábbiak szerint definiáljuk. Legyen $\mathcal{T}$ a G összes feszítőfájának halmaza, és T' egy minimális feszítőfa G-ben. A második legjobb feszítőfa egy olyan feszítőfa, amelyre $w(T) = min_{T'' \in \mathcal{T} - \{T'\}}\{w(T'')\}$.

a. Mutassuk meg, hogy a minimális feszítőfa egyértelmű, de a második legjobb feszítőfa nem szükségképpen az.

b. Legyen T egy minimális feszítőfa G-ben. Bizonyítsuk be, hogy léteznek olyan $(u, v) \in T$ és $(x, y) \notin T$ élek, hogy a $T - \{(u, v)\} \cup \{(x, y)\}$ egy második legjobb feszítőfája a G-nek.

c. Legyen T feszítőfája a G-nek, és tetszőleges $u, v \in V$ csúcsok esetén a $max[u, v]$ legyen a T-ben az u-t v-vel összekötő egyetlen út legnagyobb súlyú élének a súlya. Adjunk olyan $O(V^2)$ idejű algoritmust, amelyik adott T-re minden $u, v \in V$ csúcs pár esetén kiszámolja a $max[u, v]$-t.

d. Adjunk egy hatékony algoritmust a második legjobb feszítőfa előállítására.

23-2. Minimális feszítőfa ritka gráfokban

Egy nagyon ritka összefüggő $G = (V, E)$ gráf esetén tovább javíthatjuk a Fibonacci-kupacokat használó Prim-algoritmus $O(E + V \lg V)$ futási idejét, ha a Prim-algoritmus előtt a G egy alkalmas előfeldolgozásával csökkentjük a csúcsok számát. Nevezetesen minden u csúcshoz választunk egy hozzákapcsolódó minimális súlyú (u, v) élet, amelyet beteszünk a fokozatosan felépülő minimális feszítőfába. Ezután összevonunk minden kiválasztott élt (lásd B.4. alfejezet). Ahelyett, hogy ezeket az éleket egyesével vonnánk össze, először meghatározzuk azoknak a csúcsoknak a halmazait, amelyeket egyazon új csúccsá egyesítünk. Utána létrehozzuk azt a gráfot, amelyet az élek egyesével történő összevonásával kapnánk, de az éleket a végpontjaikat tartalmazó halmazok alapján átnevezzük. Így az eredeti gráf számos élét ugyanúgy fogjuk hívni. Az azonos nevű élek közül annak a neve marad meg, amelyiknek a súlya a megfelelő eredeti élek súlyai közül a legkisebb.

MFF-VÁZ(G, *kapocs*, c, T)

```
 1  for minden v ∈ V[G]-re
 2      do volt[v] ← HAMIS
 3         HALMAZT-KÉSZÍT(v)
 4  for minden u ∈ V[G]-re
 5      do if volt[u] = HAMIS
 6            then legyen v ∈ Szomszéd[u] a legkisebb c[u, v] értékű csúcs
 7                 EGYESÍT(u, v)
 8                 T ← T ∪ {kapocs[u, v]}
 9                 volt[u] ← volt[v] ← IGAZ
10  V[G'] ←{HALMAZT-KERES(v): v ∈ V[G]}
11  E[G'] ← ∅
12  for minden (x, y) ∈ E[G]-re
13      do u ← HALMAZT-KERES(x)
14         v ← HALMAZT-KERES(y)
15         if (u, v) ∉ E[G']
16            then E[G'] ← E[G'] ∪ {(u, v)}
17                 kapocs'[u, v] ← kapocs[x, y]
18                 c'[u, v] ← c[x, y]
19            else if c[x, y] < c'[u, v]
20                    then kapocs'[u, v] ← kapocs[x, y]
21                         c'[u, v] ← c[x, y]
22  Felépítjük a G' Szomszéd szomszédsági listáját
23  return G', kapocs', c' és T
```

Kezdetben a T minimális feszítőfa üres, és minden $(u, v) \in E$ élre a $kapocs[u, v] = (u, v)$, és $c[u, v] = w(u, v)$. A *kapocs* tulajdonság a kezdeti gráf élei és az összevont gráf élei közötti

kapcsolatot adja meg. A c tulajdonság egy él súlyát jelzi, és amikor összevonunk éleket, c-t az élsúlyok választására vonatkozó fenti séma szerint kell módosítani. Az MFF-VÁZ eljárás bemenő adatként a G, *kapocs*, c és T értékeket kapja meg, és visszaadja az összevont G' gráfot, valamint a G' gráfban a módosított *kapocs'* és c' tulajdonságokat. Az eljárás ezenkívül összegyűjti azokat a G-beli éleket, amelyek a minimális feszítőfához tartoznak.

a. Legyen T az MFF-VÁZ által visszaadott élek halmaza, és legyen A a G' gráf minimális feszítőfája, amelyet a MFF-PRIM(G', c', t) állít elő, ahol r egy tetszőleges $V[G']$-beli csúcs. Bizonyítsuk be, hogy a $T \cup \{kapocs[x, y] : (x, y) \in A\}$ egy minimális feszítőfája a G-nek.

b. Igaz-e, hogy $|V[G']| \leq |V|/2$?

c. Mutassuk meg, hogyan valósítható meg az MFF-VÁZ úgy, hogy a futási ideje $O(E)$ legyen. (*Útmutatás.* Használjon egyszerű adatszerkezetet.)

d. Tegyük fel, hogy k-szor egymás után lefuttatjuk az MFF-VÁZ eljárást úgy, hogy az egyik menet eredményeként előállt $G', kapocs'$ és c' a következő menetnek a $G, kapocs$ és c bemenete legyen, miközben a kezdetben üres T-ben felhalmozzuk az éleket. Igaz-e, hogy a k menet teljes futási ideje $O(kE)$?

e. Tegyük fel, hogy a (d) részben látott k menetes MFF-VÁZ után a Prim-algoritmust a MFF-PRIM$(G', kapocs', c')$ meghívásával hajtjuk végre, ahol a G' és c' az utolsó menetben kapott eredmények, az r pedig egy tetszőleges $V[G']$-beli csúcs. Mutassuk meg, hogyan lehet úgy megválasztani k-t, hogy a teljes futási idő $O(E \lg \lg V)$ legyen. Igaz-e, hogy a k értékének ez a megválasztása a minimális teljes aszimptotikus futási időt eredményezi?

f. Az $|E|$ milyen ($|V|$ segítségével kifejezett) értékére lesz aszimptotikusan jobb az előfeldolgozásos Prim-algoritmus az előfeldolgozás nélkülinél?

23-3. Üvegnyakú feszítőfa

Egy G irányítatlan gráf T ***üvegnyakú feszítőfája*** egy olyan feszítőfája a G-nek, amelynek maximális élsúlya a legkisebb a G összes feszítőfája között. Azt mondjuk, hogy az üvegnyakú feszítőfa értéke a T-beli legnagyobb élsúly.

a. Igaz-e, hogy egy minimális feszítőfa egyben üvegnyakú is.

Az (a) rész azt mutatja, hogy egy üvegnyakú feszítőfát nem nehezebb megtalálni, mint egy minimális feszítőfát. A további részekben azt is megmutatjuk, hogy ehhez lineáris idő elég.

b. Adjunk egy olyan lineáris idejű algoritmust, amely adott G gráf és b egész szám esetén meghatározza, hogy van-e legfeljebb b értékű üvegnyakú feszítőfa.

c. Használjuk fel a (b) rész algoritmusát alprogramként az üvegnyakú feszítőfa probléma megoldó algoritmusában. (*Útmutatás.* Használhat egy olyan alprogramot, amelyik a 23-2. feladatbeli MFF-VÁZ eljáráshoz hasonlóan összevonja az élek halmazait.)

23-4. Alternatív minimális feszítőfa algoritmus

Ebben a feladatban három különböző algoritmus pszeudo kódját adjuk meg. Mindegyik egy gráfot kap bemenetként és egy T élhalmazt ad vissza. Mindegyik algoritmusnál el kell dönteni, hogy a T minimális feszítőfa vagy nem. Mindegyik algoritmusnál – akár talál minimális feszítőfát, akár nem – vázolni kell a leghatékonyabb megvalósítást.

a. LEHET-MFF-A(G, w)

```
1 az élek nemnövekvő sorrendű rendezése a w súlyaik alapján
2 T ← E
3 for minden súlyaik szerinti nemnövekvő sorrendben vett e élre
4     do if T − {e} egy összefüggő gráf
5           then T ← T − {e}
6 return T
```

b. LEHET-MFF-B(G, w)

```
1 T ← ∅
2 for minden tetszőleges sorrendben vett e élre
3     do if T ∪ {e} nem tartalmaz kört
4           then T ← T − {e}
5 return T
```

c. LEHET-MFF-C(G, w)

```
1 T ← ∅
2 for minden tetszőleges sorrendben vett e élre
3     do if T ∪ {e} tartalmaz egy c kört
4           then legyen e′ egy maximális súlyú él a c-n
5                T ← T − {e′}
6 return T
```

Megjegyzések a fejezethez

Tarjan [292] kitűnő értekezésben tekinti át a minimális feszítőfa problémát. A minimális feszítőfa probléma történetét Graham és Hell [131] írta meg.

Tarjan az első minimális feszítőfa algoritmust O. Borůvka 1926-ban megjelent cikkének tulajdonítja. Borůvka algoritmusa a 23-2. feladatban leírt MFF-VÁZ eljárás $O(\lg V)$ számú iterációból áll. A Kruskal-algoritmust 1956-ban Kruskal [195] adta meg. A Prim-algoritmusként ismert algoritmust valóban Prim [250] találta ki, de korábban, 1930-ban már V. Jarník is felfedezte.

Annak az oka, hogy a mohó algoritmus hatékonyan keres minimális feszítőfát az, hogy egy gráf erdőinek halmaza egy matroidot alkot. (Lásd 16.4. alfejezet.)

Amikor $|E| = \Omega(V \lg V)$, a Fibonacci-kupacok segítségével megvalósított Prim-algoritmus $O(E)$ futási idejű. Ritka gráfokra, a Prim-algoritmus, a Kruskal-algoritmus és Borůvka algoritmusának ötleteit, valamint fejlett adatszerkezeteket felhasználva, Fredman és Tarjan [98] adott egy $O(E \lg^\star V)$ futási idejű algoritmust. Gabow, Galil, Spencer és Tarjan [102] ezt az algoritmust javította $O(E \lg\lg^\star V)$ idejűre. Chazelle [53] adott egy $O(E\,\widehat{\alpha}(E, V))$ futási idejű algoritmust, ahol $\widehat{\alpha}(E, V)$ az Ackermann-függvény inverz függvénye. (Lásd a 21. fejezet megjegyzéseit az Ackermann függvény és annak inverzének elemzésénél.) Eltér ően az előző minimális feszítőfa algoritmusoktól, Chazelle algoritmusa nem a mohó stratégiát követi.

Egy kapcsolódó probléma a ***feszítőfa ellenőrzés,*** amelyben adott egy $G = (V, E)$ gráf és egy $T \subset E$ fa, és azt kell eldönteni, hogy T egy minimális feszítőfa-e a G-ben. King [177] ad

egy olyan lineáris idejű algoritmust a feszítőfa ellenőrzésre, amely Komlós [188] és Dixon, Rauch és Tarjan [77] korábbi munkáira épül.

A fenti algoritmusok mind determinisztikusak és a 8. fejezetben leírt összehasonlítás alapú modellek közé tartoznak. Karger, Klein és Tarjan [169] ad egy véletlenített minimális feszítőfa algoritmust, amelyik várható futási ideje $O(V + E)$. Ez az algoritmus, hasonlóan a 9.3 alfejezet lineáris idejű kiválasztás algoritmushoz, rekurziót alkalmaz: egy rekurzív hívás azt a segédproblémát oldja meg, amely tetszőleges minimális feszítőfához nem tartozó élek E' halmazát határozza meg. Ezután az $E - E'$-re tett másik rekurzív hívással meghatározza a minimális feszítőfát. Ez az algoritmus Borùvka algoritmusának és King feszítőfa ellenőrző algoritmusának ötleteit is felhasználja.

Fredman és Willard [100] megmutatta, hogyan lehet $O(E + V)$ idő alatt minimális feszítőfát találni egy nem összehasonlítás alapú determinisztikus algoritmussal. Algoritmusuk feltételezi, hogy az adatok b bites egész számok, és a számítógép memóriája b bites szavanként címezhető.

24. Adott csúcsból induló legrövidebb utak

Egy gépkocsivezető a lehető legrövidebb úton szeretne eljutni Chicagóból Bostonba. Hogyan határozhatjuk meg ezt a legrövidebb utat, ha rendelkezünk az Egyesült Államok autós térképével, amelyen minden, két szomszédos útkeresztező dés közötti távolságot bejelöltek?

Egyik lehetséges megoldás az, ha módszeresen elő állítjuk az összes, Chicagóból Bostonba vezető utat azok hosszával együtt, és kiválasztjuk a legrövidebbet. Könnyű azonban azt belátni, hogy még ha a kört tartalmazó utakkal nem is foglalkozunk, akkor is több millió olyan lehetőség marad, amelyek többsége egyáltalán nem érdemel figyelmet. Például, egy Houstonon keresztül vezető Chicago és Boston közötti út nyilvánvalóan szerencsétlen útvonalválasztás lenne, hiszen Houston érintése csaknem ezer mérföldes kitér őt jelent.

Ebben és a 25. fejezetben azt mutatjuk meg, hogyan lehet hatékonyan megoldani az ilyen és ehhez hasonló problémákat. A ***legrövidebb utak problémában*** adott egy élsúlyozott, irányított $G = (V, E)$ gráf, ahol a $w : E \to \mathbf{R}$ súlyfüggvény rendel az élekhez valós értékeket. A $p = \langle v_0, v_1, \ldots, v_k \rangle$ út ***súlya*** az utat alkotó élek súlyainak összege:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Definiáljuk az u-ból v-be vezető ***legrövidebb út súlyát*** az alábbi módon:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{ha vezet út } u\text{-ból } v\text{-be,} \\ \infty, & \text{különben.} \end{cases}$$

Egy az u csúcsból a v csúcsba vezető ***legrövidebb úton*** az olyan p utat értjük, amelyre $w(p) = \delta(u, v)$ teljesül.

A Chicago–Boston példában az autós térképet egy gráf segítségével modellezhetjük: a csúcsok jelentik a kereszteződéseket, az élek szimbolizálják a kereszteződések közötti útszakaszokat, az élek súlyai pedig az útszakaszok hosszaira utalnak. A célunk egy olyan legrövidebb útnak a megtalálása, amely Chicago egy adott (mondjuk a Szent Clark és az Addison sugárút) kereszteződéséből indul, és Boston egy adott (például a Brookline sugárút és Yawkey Way) kereszteződésébe érkezik.

Az élsúlyok a távolságokétól eltérő metrikákat is kifejezhetnek. Gyakran használják idő, költség, büntetés, veszteség vagy más olyan mennyiség megjelenítésére, amely egy út mentén lineárisan halmozódik, és amelyet minimalizálni szeretnénk.

A 22.2. alfejezet szélességi keresése egy olyan legrövidebb utak algoritmus, amely súlyozatlan gráfokon működik, azaz olyan gráfokon, amelyben minden élt egységnyi súlyúnak tekintünk. Javasoljuk az Olvasónak a 22.2. alfejezet ismételt áttekintését, mivel a legrövidebb utak súlyozott gráfokban történő tanulmányozása során a szélességi keresés számos ötletét felhasználjuk.

Változatok

Ebben a fejezetben az ***adott csúcsból induló legrövidebb utak problémájával*** foglalkozunk. Egy $G = (V, E)$ gráfban meg szeretnénk találni egy adott $s \in V$ ***kezdőcsúcsból*** az összes $v \in V$ csúcshoz vezető legrövidebb utat. Az ilyen problémát megoldó algoritmussal számos más, az alábbi változatokat is magában foglaló, problémát lehet megoldani.

Adott csúcsba érkező legrövidebb utak problémája: Keressük meg az összes v csúcsból egy adott t ***célcsúcsba*** vezető legrövidebb utakat. Ennek a problémának a megoldását könnyen visszavezethetjük az adott kezdőcsúcsból induló legrövidebb utak problémájára, ha megfordítjuk a gráfbeli élek irányítását.

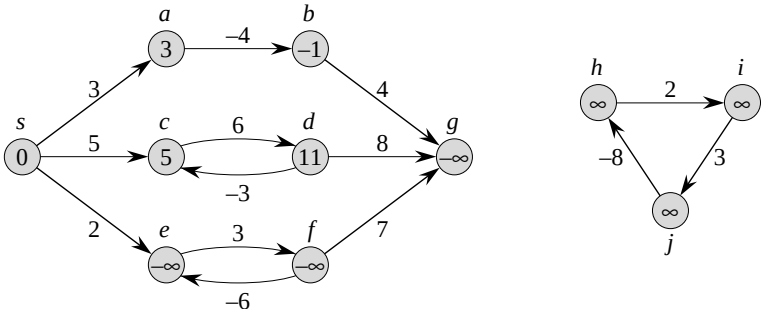
Adott csúcspár közötti legrövidebb út problémája: Keressük meg egy adott u csúcsból egy adott v csúcsba vezető egyik legrövidebb utat. Ha megoldjuk az adott kezdőcsúcsból induló utak problémáját, akkor ezt a problémát is megoldottuk. S őt, nem is ismerünk erre a problémára olyan megoldó algoritmust, amelyik aszimptotikusan gyorsabb lenne az adott kezdőcsúcsból induló legrövidebb utakat találó legjobb algoritmusnál.

Összes csúcs pár közti legrövidebb utak problémája: Keressük meg az összes u és v csúcspárra az u-ból a v-be vezető legrövidebb utat. Igaz ugyan, hogy ez a probléma megoldható az adott kezdőcsúcsból induló legrövidebb utakat kereső algoritmus segítségével, amennyiben azt minden lehetséges kezdőcsúcsra végrehajtjuk, de található ennél gyorsabb megoldás is. A 25. fejezet részletesen foglalkozik az összes csúcs pár problémájával.

Legrövidebb út optimális részstruktúrája

A legrövidebb utak algoritmusai jellegzetesen azt a tulajdonságot aknázzák ki, hogy egy két csúcs között vezető legrövidebb út más legrövidebb utakat is tartalmaz. (A 26. fejezet Edmonds–Karp maximális folyam algoritmusa ugyancsak erre a tulajdonságra épül.) Mind a dinamikus programozás (15. fejezet), mind a mohó módszerek (16. fejezet) alkalmazhatóságát ez az optimális-részstruktúra tulajdonság garantálja. A Dijkstra-algoritmus, amelyet a 24.3. alfejezetben vizsgálunk majd meg, egy mohó algoritmus, a Floyd–Warshall-algoritmus pedig, amelyik az összes csúcs pár között keres legrövidebb utakat (lásd 25. fejezet), egy dinamikus-programozási algoritmus. A következő lemma pontosabban fogalmazza meg az optimális-részstruktúra tulajdonságot.

24.1. lemma (legrövidebb út részútja is legrövidebb út). *Adott egy $w : E \to \mathbf{R}$ súlyfüggvénnyel élsúlyozott, irányított $G = (V, E)$ gráf, legyen egy v_1-ből v_k-ba vezető legrövidebb út $p = \langle v_1, v_2, \ldots, v_k \rangle$, és bármely olyan i és j-re, amelyre $1 \le i \le j \le k$, legyen $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ egy v_i-ből v_j-be vezető részútja p-nek. Ekkor p_{ij} egy legrövidebb v_i-ből v_j-be vezető út.*



24.1. ábra. Negatív súlyú élek egy irányított gráfban. Az egyes csúcsokba írt számok az s kezdőcsúcsból odavezető legrövidebb út súlyát mutatják. Mivel az e és az f az s-ből elérhető negatív kört alkot, ezek legrövidebb-út súlya $-\infty$. Amiért a g elérhető egy $-\infty$ legrövidebb-út súlyú csúcsból, ugyancsak $-\infty$ legrövidebb-út súllyal rendelkezik. A h, i és a j nem érhetők el az s-ből, ezért a legrövidebb-út súlyuk ∞, annak ellenére, hogy egy negatív súlyú körön fekszenek.

Bizonyítás. Ha felbontjuk a p utat $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$-ra, akkor $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Tegyük fel, hogy van egy p'_{ij} v_i-ből v_j-be vezető út, amelynek súlya $w(p'_{ij}) < w(p_{ij})$. Ekkor a $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ egy v_i-ből v_j-be vezető olyan út, amelynek súlya $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ kisebb, mint $w(p)$, ami ellentmond annak a feltételnek, hogy p egy legrövidebb v_1-ből v_k-be vezető út. ■

Negatív súlyú élek

Találkozhatunk olyan adott kezdőcsúcsból induló legrövidebb utak problémával, ahol előfordulhatnak negatív súlyú élek. Ha a $G = (V, E)$ irányított gráf nem tartalmaz az s kezdőcsúcsból elérhető negatív összsúlyú köröket (röviden negatív köröket), akkor bármelyik $v \in V$ csúcs esetében a $\delta(s, v)$ legrövidebb út súlya jól definiált marad még akkor is, ha az egy negatív érték. Ha azonban vannak s-ből elérhető negatív körök, akkor a legrövidebb-út súlyok definiálatlanok. Nincs ugyanis s-ből egy ilyen körön fekvő csúcsba vezető legrövidebb út – kisebb súlyú utat mindig lehet találni a feltételezett legrövidebb útnál, ha ahhoz még hozzávesszük a negatív kör egy bejárását. Ha egy negatív kör található valamelyik s-ből v-be vezető úton, akkor definíció szerint $\delta(s, v) = -\infty$.

A 24.1. ábra bemutatja a negatív súlyok hatását a legrövidebb utak súlyára. Mivel csak egy s-ből a-ba vezető út ($\langle s, a\rangle$) van, a $\delta(s, a) = w(s, a) = 3$. Ugyancsak egy út vezet az s-ből a b-be, így $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Végtelen sok út vezet s-ből c-be: $\langle s, c\rangle, \langle s, c, d, c\rangle \langle s, c, d, c, d, c\rangle$ és így tovább. A $\langle c, d, c\rangle$ kör súlya $6 + (-3) = 3 > 0$, ezért a legrövidebb s-ből c-be vezető út az $\langle s, c\rangle$, amelynek a súlya $\delta(s, c) = 5$. Hasonló megfontolásból az s-ből a d-be vezető legrövidebb út az $\langle s, c, d\rangle$ a $\delta(s, d) = w(s, c) + w(c, d) = 11$ súllyal. Ugyancsak végtelen sok út vezet az s-ből az e-be: $\langle s, e\rangle, \langle s, e, f, e\rangle, \langle s, e, f, e, f, e\rangle$ és így tovább. Mivel azonban az $\langle e, f, e\rangle$ kör súlya $3 + (-6) = -3 < 0$, ezért nincs legrövidebb út s-ből e-be. A negatív súlyú $\langle e, f, e\rangle$ kört tetszőleges sokszor bejárva, tetszőlegesen kicsi súlyú s-ből e-be vezető utat találhatunk, azaz $\delta(s, e) = -\infty$. Ugyanígy $\delta(s, f) = -\infty$. Amiért g elérhető az f-ből, tetszőlegesen kis súlyú utakat találhatunk az s-ből a g-be is, és $\delta(s, g) = -\infty$. A h, i és j csúcsok szintén negatív kört alkotnak, de nem érhetők el az s csúcsból, ezért $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Néhány legrövidebb-utak algoritmus, mint például a Dijkstra-algoritmus is, feltételezik, hogy a vizsgált gráf élei között nincs negatív súlyú. Ilyen gráffal találkoztunk az autótérképes példában. Más algoritmusok, mint például a Bellman–Ford-algoritmus, negatív súlyú éleket tartalmazó gráfban is működnek, és helyes választ adnak mind addig, amíg nincs a kezdőcsúcsból elérhető negatív kör. Ha viszont van ilyen negatív súlyú kör, akkor az algoritmus felfedi és jelzi ezeket.

Körök

Tartalmazhat-e kört egy legrövidebb út? Amint azt az előbb éppen láthattuk, negatív súlyú kört nem. Pozitív súlyú kör sem lehet benne, hiszen a kört eltávolítva belőle egy ugyanolyan kezdő- és végpontú, de kisebb súlyú utat kapunk. Azaz, ha $p = \langle v_0, v_1, \ldots, v_k \rangle$ egy olyan út, amelyen $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ egy pozitív súlyú kör ($v_i = v_j$ és $w(c) \rangle 0$)), akkor a $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$ út súlya $w(p') = w(p) - w(c) \langle w(p)$ lesz, és így p nem lehet v_0-ból v_k-ba vezető legrövidebb út.

Hátra vannak még a nulla súlyú körök. A nulla súlyú köröket bármelyik útból eltávolítva ugyanolyan súlyú utakat kapunk. Így ha van az s kezdőcsúcsból a v végcsúcsba vezető nulla súlyú kört tartalmazó út, akkor van ezt a kört nem tartalmazó másik legrövidebb út is. Egy nulla súlyú köröket tartalmazó legrövidebb útról egymás után eltávolíthatjuk ezeket a köröket egészen addig, amíg körmentes legrövidebb utat nem kapunk. Így az általánosság rovása nélkül feltehetjük, hogy amikor legrövidebb utakat találunk, azok nem tartalmaznak köröket. Mivel egy körmentes $G = (V, E)$ gráfban bármelyik körmentes út legfeljebb $|V|$ különböző csúcsot tartalmazhat, az út éleinek száma legfeljebb $|V| - 1$. Ezzel leszűkíthetjük a vizsgálatunkat a legfeljebb $|V| - 1$ élt tartalmazó legrövidebb utakra.

Legrövidebb utak ábrázolása

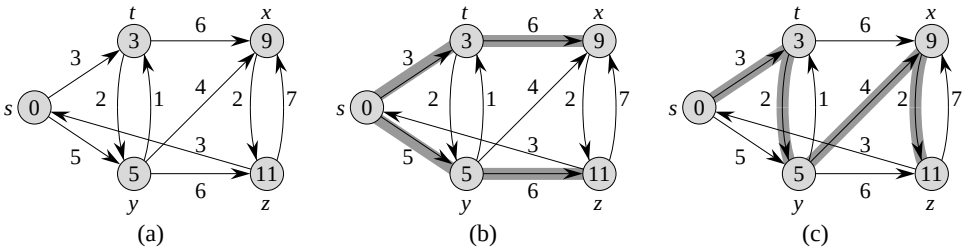
Gyakori kívánság, hogy ne csak a legrövidebb-út súlyokat számoljuk ki, hanem adjuk meg azokat a csúcsokat is, amelyek a legrövidebb utak mentén helyezkednek el. A legrövidebb utak általunk használt reprezentációja hasonlít arra, amit a 22.2. alfejezetben a szélességi keresésnél alkalmaztunk. Egy adott $G = (V, E)$ gráf minden $v \in V$ csúcsához hozzárendelünk egy $\pi[v]$ ***szülő*** értéket, amely vagy a csúcsnak az egyik szülője, vagy a NIL érték. Az ebben a fejezetben szereplő legrövidebb-utak algoritmusai úgy állítják be a π értékeket, hogy azok mentén egy v csúcsból visszafelé haladva megkapjuk az egyik s-ből v-be vezető legrövidebb utat. Egy olyan v csúcsra, amelyikre $\pi[v] \neq$ NIL, a 22.2. alfejezet UTAT-NYOMTAT(G, s, v) eljárása kiír egy s-ből v-be vezető legrövidebb utat.

Egy legrövidebb-utak algoritmus működése alatt azonban a π értékek nem szükségszerűen a legrövidebb utakat jelölik. Akárcsak a szélességi keresésnél, bennünket a π értékek által meghatározott $G_\pi = (V_\pi, E_\pi)$ ***szülő részgráf*** érdekel. Itt a V_π a nem NIL szülőjű G-beli csúcsokat, valamint az s csúcsot tartalmazza:

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}.$$

Az E_π-beli irányított élek a π értékek által kijelölt csúcsokból vezetnek a V_π-beli csúcsokba:

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}.$$



24.2. ábra. (a) Élsúlyozott, irányított gráf az s kezdőcsúcsból induló legrövidebb-út súlyokkal. **(b)** A vastagított élek egy s gyökerű legrövidebb-utak fát jelölnek. **(c)** Egy másik ugyanolyan gyökerű legrövidebb-utak fa.

Be fogjuk bizonyítani, hogy e fejezet algoritmusainak G_π gráfja befejeződéskor egy „legrövidebb-utak fája" lesz – azaz egy olyan s gyökerű fa, amely az s-ből elérhető bármelyik G-beli csúcsba egy s-ből kiinduló legrövidebb utat tartalmaz. Egy legrövidebb-utak fája hasonló a 22.2. alfejezet szélességi keresés fájához, csak itt a kezdőcsúcsból kiinduló legrövidebb utakat az élek súlya szerint, nem pedig az élek száma szerint értjük. Pontosan fogalmazva, legyen $G = (V, E)$ egy irányított, $w : E \to \mathbf{R}$ súlyfüggvénnyel élsúlyozott gráf, és feltesszük, hogy G nem tartalmaz az s kezdőcsúcsból elérhető negatív köröket, azaz a legrövidebb utak jól definiáltak. Egy s gyökerű ***legrövidebb-utak fa*** egy olyan $G' = (V', E')$ részgráf, ahol $V' \subseteq V$ és $E' \subseteq E$ úgy, hogy

1. V' az s-ből elérhető G-beli csúcsok halmaza,
2. G' egy s gyökerű fa, és
3. minden $v \in V'$-re az egyetlen G'-beli s-ből vezető út egy legrövidebb s-ből v-be vezető út a G-ben.

A legrövidebb utak nem szükségszerűen egyértelműek, akárcsak a legrövidebb-utak fái. Példaként a 24.2. ábra mutat egy élsúlyozott, irányított gráfot, és abban ugyanazzal a gyökérrel két legrövidebb-utak fát.

Fokozatos közelítés

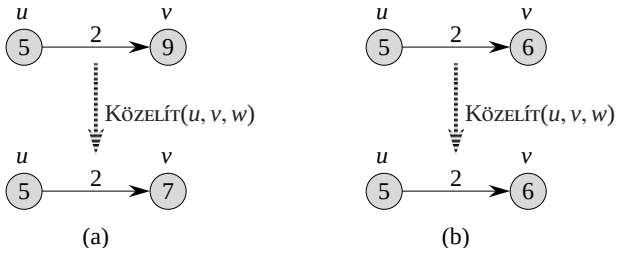
Ennek a fejezetnek az algoritmusai a ***fokozatos közelítés*** módszerét alkalmazzák. Minden $v \in V$ csúcsnál nyilvántartunk egy $d[v]$ értéket, amely egy felső korlátot ad az s kezdőcsúcsból a v-be vezető legrövidebb út súlyára. A $d[v]$-t egy ***legrövidebb-út becslésnek*** nevezzük.

Kezdetben a legrövidebb-út becsléseket és a szülőkre mutató π értékeket a következő $\Theta(V)$ futási idejű eljárás állítja be.

EGY-FORRÁS-KEZDŐÉRTÉK(G, s)

```
1  for minden v ∈ V[G]-re
2      do d[v] ← ∞
3         π[v] ← NIL
4  d[s] ← 0
```

A kezdeti értékek beállítása után minden $v \in V$-re $\pi[v] =$ NIL, és minden $v \in V - \{s\}$-re $d[v] = \infty$ áll fenn.



24.3. ábra. Egy (u, v) éllel történő közelítő lépés. A csúcsokban feltüntettük az aktuális legrövidebb-út becsléseket, az él súlya $w(u, v) = 2$. **(a)** A közelítést megelőzően $d[v] > d[u] + w(u, v)$, ezért a $d[v]$ értéke csökken. **(b)** Itt a közelítés előtt $d[v] > d[u] + w(u, v)$, így a $d[v]$-t nem változtatja meg a közelítés.

Egy (u, v) él segítségével történő ***közelítés***[1] technikáját alkalmazzák. Minden $v \in V$ csúcsnál nyilvántartunk egy $d[v]$ értéket, amely egy felső korlátot ad az s kezdőcsúcsból a v-be vezető legrövidebb út súlyára. A $d[v]$-t egy ***legrövidebb-út becslésnek*** nevezzük. Egy ellenőrzésből áll, amelyik összeveti a v csúcshoz ez idáig legrövidebbnek talált utat az u csúcson keresztül vezető úttal, és ha ez utóbbi rövidebb, akkor módosítja a $d[v]$ és $\pi[v]$ értékeket. A közelítő lépés csökkentheti a $d[v]$ legrövidebb-út becslés értékét, és átállíthatja a $\pi[v]$ mezőt az u csúcsra. Az alábbi kód az (u, v) él közelítő lépését írja le.

Közelít(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

A 24.3. ábra két példát mutat egy él segítségével történő közelítésre; egyikben a legrövidebb-út becslés csökken, a másikban a becslés nem változik.

Ennek a fejezetnek mindegyik algoritmusa meghívja az Egy-forrás-kezdőérték eljárást, majd az élekkel egymás után végez közelítéseket. Sőt a közelítés az egyetlen olyan lépés, amely megváltoztatja a legrövidebb-út becsléseket és a szülő értékeket. A fejezet algoritmusai abban különböznek egymástól, hogy hányszor és milyen sorrendben végzik el az élekkel a Közelít műveletet. Dijkstra algoritmusa és a körmentes irányított gráfban működő legrövidebb-utak algoritmus minden éllel pontosan egyszer közelít. A Bellman–Ford-algoritmus az egyes élekkel többször végez közelítést.

A legrövidebb utak és a fokozatos közelítés tulajdonságai

Ahhoz, hogy e fejezet algoritmusainak helyességét belássuk, fel kell használnunk a legrövidebb utak és a fokozatos közelítés néhány tulajdonságát. Itt most csak kimondjuk ezeket a tulajdonságokat, és majd a 24.5. alfejezetben bizonyítjuk be őket formálisan. A helyes hivatkozás érdekében minden itt kimondott tulajdonság tartalmazza a 24.5. alfejezet megfelelő lemmájának vagy következményének sorszámát. Az alábbi öt tulajdonság, amelyek a legrö-

[1]Furcsának tűnhet, hogy egy felső határ szigorítását végző műveletre a „relaxál” kifejezést használjuk. Ennek történeti okai vannak. Egy relaxáló lépés eredményére úgy is tekinthetünk, mint a $d[v] \leq d[u] + w(u, v)$ feltétel pihentetésére, amely a háromszög-egyenlőtlenség (24.10. lemma) miatt a $d[u] = \delta(s, u)$ és $d[v] = \delta(s, v)$ fennállása esetén biztosan teljesül. Azaz, ha $d[v] \leq d[u] + w(u, v)$, akkor nincs olyan kényszer, hogy ezt a feltételt kielégítsük, a feltétel tehát relaxál.

videbb utak becslésével vagy a szülő részgráffal kapcsolatosak, implicit módon feltételezik azt, hogy a gráfot már inicializáltuk az EGY-FORRÁS-KEZDŐÉRTÉK(G, s) eljárás meghívásával, és a legrövidebb utak becsléseinek, valamint a szülő részgráfnak a megváltoztatására az egyetlen mód a fokozatos közelítés sorozatos alkalmazása.

Háromszög-egyenlőtlenség (24.10. lemma)
Bármely $(u, v) \in E$ élre fennáll, hogy $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Felső-korlát tulajdonság (24.11. lemma)
Minden $v \in V$ csúcsra fennáll, hogy $d[v] \geq \delta(s, v)$, és ha $d[v]$ egyszer eléri a $\delta(s, v)$ értéket, attól fogva sohasem változik.

Nincs-út tulajdonság (24.12. következmény)
Ha nincs út s-ből v-be, akkor a $d[v] = \delta(s, v) = \infty$ áll fenn.

Konvergencia tulajdonság (24.14. lemma)
Ha valamely $u, v \in V$ csúcsokra az $s \rightsquigarrow u \rightarrow v$ egy legrövidebb út a G-ben, és az (u, v) éllel történő fokozatos közelítése kezdetén a $d[u] = \delta(s, u)$ fennáll, akkor azt követően teljesülni fog a $d[v] = \delta(s, v)$.

Út-közelítés tulajdonság (24.15. lemma)
Ha $p = \langle v_0, v_1, \ldots, v_k \rangle$ legrövidebb út $s = v_0$-ból v_k-ba, és p éleit a $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ sorrendben fokozatosan közelítjük, akkor $d[v_k] = \delta(s, v_k)$ teljesülni fog. Ez a tulajdonság akkor is igaz, ha más élekkel is végzünk közelítést, legyenek azok akár a p éleivel való közelítések közé keverve.

Szülő részgráf tulajdonság (24.17. lemma)
Amikor az összes $v \in V$ csúcsra fennáll, hogy $d[v] = \delta(s, v)$, akkor a szülő részgráf egy s gyökerű legrövidebb utakat tartalmazó fa.

A fejezet tartalma

A 24.1. alfejezetben a Bellman–Ford-algoritmust mutatjuk be, amelyik abban az általános esetben oldja meg az adott kezdőcsúcsból induló legrövidebb utak problémáját, amikor a gráfban negatív súlyú élek is lehetnek. Érdemes felfigyelni a Bellman–Ford-algoritmus egyszerűségére, valamint arra a tulajdonságára, hogy észreveszi a kezdőcsúcsból elérhető negatív köröket. A 24.2. alfejezet egy lineáris idejű algoritmust ad az adott kezdőcsúcsból induló legrövidebb utak megtalálására irányított körmentes gráfokban. A 24.3. alfejezet a Dijkstra-algoritmust mutatja be, amelyik kisebb futási idejű, mint a Bellman–Ford-algoritmus, de csak nemnegatív súlyú éleket tartalmazó gráfokban alkalmazható. A 24.4. alfejezet megmutatja, hogyan használható fel a Bellman–Ford-algoritmus a „lineáris programozás” egyik speciális esetének megoldására. Végül a 24.5. alfejezetben bebizonyítjuk a legrövidebb utak és fokozatos közelítés feljebb vázolt tulajdonságait.

A későbbiekben a végtelennel végzett aritmetika gyakran megjelenik elemzésünkben, ami néhány konvenció bevezetését teszi szükségessé. Feltesszük, hogy bármely $a \neq -\infty$ valós számra $a + \infty = \infty + a = \infty$. Ahhoz, hogy bizonyításaink érvényesek legyenek negatív körök megjelenése esetén is, feltesszük, hogy bármely $a \neq \infty$ valós számra $a + (-\infty) = (-\infty) + a = -\infty$.

Ennek a fejezetnek minden algoritmusa feltételezi, hogy a G irányított gráfot szomszédsági listákkal reprezentáljuk. Továbbá minden élt a súlyával együtt tárolunk úgy, hogy minden szomszédsági lista bejárásakor minden él súlya $O(1)$ idő alatt elérhető legyen.

24.1. Bellman–Ford-algoritmus

A ***Bellman–Ford-algoritmus*** az adott kezdőcsúcsból induló legrövidebb utak problémáját abban az általánosabb esetben oldja meg, amikor az élek között negatív súlyúakat is találhatunk. Adott egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott irányított $G = (V, E)$ gráf, ahol a kezdőcsúcs az s. A Bellman–Ford-algoritmus egy logikai értéket ad vissza annak jelölésére, hogy van vagy nincs a kezdőcsúcsból elérhető negatív kör. Ha van ilyen kör, az algoritmus jelzi, hogy nem létezik megoldás. Ha nincs ilyen kör, akkor az algoritmus előállítja a legrövidebb utakat és azok súlyait.

A Bellman–Ford-algoritmus a fokozatos közelítés technikáját alkalmazza, bármelyik $v \in V$ csúcsnál az s kezdőcsúcsból odavezető legrövidebb út súlyára adott $d[v]$ becslését ismételten csökkenti mindaddig, amíg az eléri annak tényleges $\delta(s, v)$ értékét. Az algoritmus akkor és csak akkor tér vissza IGAZ értékkel, ha a gráf nem tartalmaz a kezdőcsúcsból elérhető negatív köröket.

BELLMAN–FORD(G, w, s)

```
1  EGY-FORRÁS-KEZDŐÉRTÉK(G, s)
2  for i ← 1 to |V[G]| − 1
3      do for minden (u, v) ∈ E[G]
4             KÖZELÍT(u, v, w)
5  for minden (u, v) ∈ E[G]
6      do if d[v] ⟩ d[u] + w(u, v)
7            then return HAMIS
8  return IGAZ
```

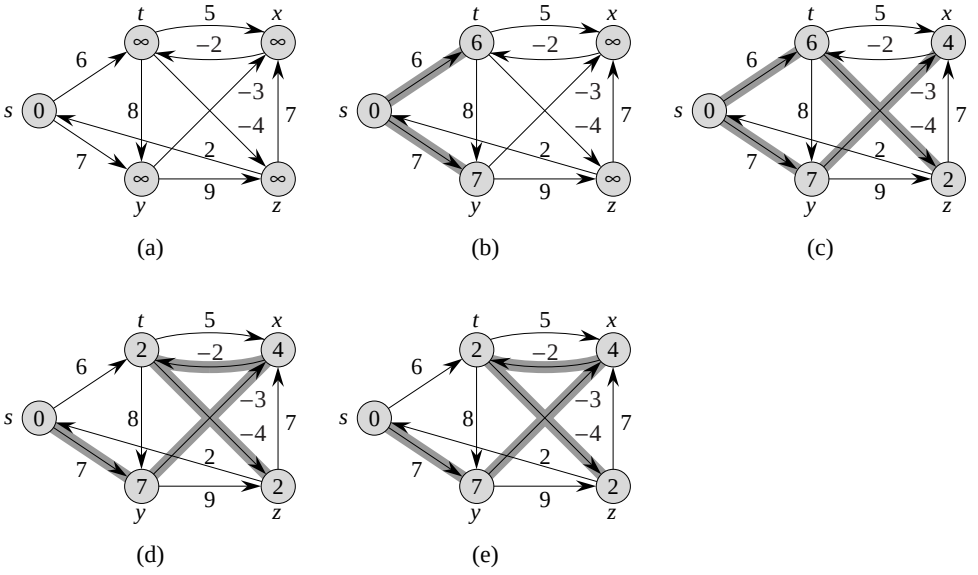
A 24.4. ábra a Bellman–Ford-algoritmus működését egy öt csúcsból álló gráfon mutatja be. A d és π kezdeti beállítása után az algoritmus $|V| - 1$ menetet végez a gráf éleivel. Minden menet a 2–4. sorok **for** ciklusának egy iterációja, amely a gráf minden élével egyszer végez közelítést. A 24.4(b)–(e) ábrái az algoritmus állapotait mutatják az élekkel végzett négy menet mindegyike után. $|V| - 1$ menet után az 5–8. sorok negatív kört keresnek, és a megfelelő logikai értéket adják vissza. (Kicsit később majd látni fogjuk, hogyan működik ez az ellenőrzés.)

A Bellman–Ford-algoritmus futási ideje $O(VE)$, mivel az 1. sor előkészítése $\Theta(V)$ idejű, a 2–4. sorokban az élekkel végzett $|V| - 1$ menet mindegyike $\Theta(E)$ ideig tart, és az 5–7. sorok **for** ciklusa $O(E)$ idejű.

A Bellman–Ford-algoritmus helyességét úgy bizonyítjuk, hogy először azt mutatjuk meg, ha nincsenek negatív körök, akkor az algoritmus a helyes legrövidebb-út súlyokat állítja elő bármelyik kezdőcsúcsból elérhető csúcsra.

24.2. lemma. *Legyen $G = (V, E)$ a $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol s a kezdőcsúcs, és tegyük fel, hogy a G nem tartalmaz s-ből elérhető negatív köröket. Ekkor* BELLMAN-FORD *2–4. sorai* ***for*** *ciklusának $|V| - 1$ darab iterációja után minden s-ből elérhető v csúcsra fennáll, hogy $d[v] = \delta(s, v)$.*

Bizonyítás. A lemma bizonyításához az út-közelítés tulajdonságot használjuk fel. Legyen a v egy s-ből elérhető csúcs, a $p = \langle v_0, v_1, \ldots, v_k \rangle$ pedig egy s-ből v-be vezető körmentes legrövidebb út, ahol $v_0 = s$ és $v_k = v$. A p út legfeljebb $|V| - 1$ élből áll, így $k \leq |V| - 1$.



24.4. ábra. A Bellman–Ford-algoritmus működése. A kezdőcsúcs az s csúcs. A d értékeket beleírtuk a csúcsokba, és a vastagított élek jelzik a szülő értékeket: ha (u, v) él vastagított, akkor $\pi[v] = u$. Ebben a sajátos példában mindegyik menetben az alábbi sorrendben végzünk fokozatos közelítést az élekkel: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** Az első menet előtti helyzet. **(b)–(e)** Az egymást követő menetek után kialakult helyzetek. Az (e) rész a végleges d és π értékeket mutatja. A Bellman–Ford-algoritmus ebben a példában IGAZ értékkel tér vissza.

A 2–4. sorok **for** ciklusának $|V| - 1$ darab iterációja az összes $|E|$ éllel végez közelítést. Az első iterációban sor kerül a (v_0, v_1) élre, a másodikban a (v_1, v_2) élre, végül a k-adik iterációban a (v_{k-1}, v_k) élre. Az út-közelítés tulajdonság miatt ekkor $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. ■

24.3. következmény. *Legyen $G = (V, E)$ a $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol s a kezdőcsúcs. Egy tetszőleges $v \in V$ csúcshoz akkor és csak akkor vezet az s-ből út, ha a G-re futtatott* BELLMAN-FORD *a $d[v] < \infty$ értékkel áll meg.*

Bizonyítás. A bizonyítást a 24.1-2. gyakorlatra hagyjuk. ■

24.4. tétel (Bellman–Ford-algoritmus helyessége). *Egy $G = (V, E)$ súlyozott, irányított gráfon futtassuk a* BELLMAN-FORD*-ot, ahol a súlyfüggvény a $w : E \to \mathbf{R}$, és a kezdőcsúcs az s. Ha G nem tartalmaz az s-ből elérhető negatív köröket, akkor az algoritmus* IGAZ *értéket ad vissza, továbbá minden $v \in V$ csúcsra $d[v] = \delta(s, v)$, és a G_π szülő részgráf egy s gyökerű legrövidebb-utak fa lesz. Ha G-ben van egy s-ből elérhető negatív kör, akkor az algoritmus a* HAMIS *értékkel tér vissza.*

Bizonyítás. Tegyük fel, hogy a G gráf nem tartalmaz az s-ből elérhető negatív súlyú köröket. Először azt az állítást bizonyítjuk be, hogy megálláskor minden $v \in V$ csúcsra a $d[v] = \delta(s, v)$ fennáll. Ha a v csúcs az s-ből elérhető, akkor a 24.2. lemma igazolja az állításunkat. Ha v nem érhető el az s-ből, akkor az állítás a nincs-út tulajdonságból adódik. Az állítás

tehát igaz. Az állítást felhasználva a szülő részgráf tulajdonság miatt a G_π egy legrövidebb-utak fa. Most azt mutatjuk meg az állítás segítségével, hogy a BELLMAN-FORD IGAZ értéket ad vissza. Befejeződésekor minden $(u, v) \in E$ élre

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \qquad \text{(a háromszög-egyenlőtlenség miatt)} \\ &= d[u] + w(u, v), \end{aligned}$$

és így a 6. sor egyik ellenőrzése során sem fog a BELLMAN-FORD HAMIS értéket visszaadni. Ezért mindig IGAZ értéket ad vissza.

Tegyük fel most azt, hogy a G gráf egy $c = \langle v_0, v_1, \ldots, v_k \rangle$ s-ből elérhető negatív kört tartalmaz, ahol $v_0 = v_k$. Ekkor

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0. \tag{24.1}$$

Tegyük fel indirekt módon azt, hogy a Bellman–Ford-algoritmus IGAZ értéket ad vissza. Ennek megfelelően $d[v_i] \leq d[vi - 1] + w(v_{i-1}, v_i)$ minden $i = 1, 2, \ldots, k$ indexre. A c kör mentén összegezve az egyenlőtlenségeket azt kapjuk, hogy

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k} d[v_{i-1}] + w(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

Mivel $v_0 = v_k$, a c kör mindegyik csúcsa pontosan egyszer szerepel a $\sum_{i=1}^{k} d[v_i]$ és a $\sum_{i=1}^{k} d[v_{i-1}]$ összegzésekben, és így

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}].$$

Továbbá, a 24.3. következmény miatt, $d[v_i]$ véges minden $i = 1, 2, \ldots, k$-ra. Ezért

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

ami ellentmond a (24.1) egyenlőtlenségnek. Beláttuk, hogy a Bellman–Ford-algoritmus IGAZ értéket ad vissza, ha a G gráf nem tartalmaz a kezdőcsúcsból elérhető negatív köröket, és HAMIS értékkel tér vissza különben. ■

Gyakorlatok

24.1-1. Futtassuk a Bellman–Ford-algoritmust a 24.4. ábra irányított gráfján, ahol a z csúcsot választjuk kezdőcsúcsnak. Mindegyik menetben lexikografikus sorrendben végezzünk közelítést az élekkel, és mutassuk be a d és π értékeket az egyes menetek után. Majd változtassuk meg az (z, x) él súlyát 4-re, és futtassuk le újra az algoritmust úgy, hogy a s-t használjuk kezdőcsúcsnak.

24.1-2. Bizonyítsuk be a 24.3. következményt.

24.1-3. Adott a $G = (V, E)$ élsúlyozott, irányított negatív köröket nem tartalmazó gráf. Legyen m az u-ból v-be vezető legrövidebb utak élszámának minimuma az összes $u, v \in V$ csúcspár esetén. (Itt a legrövidebb utat a súlya, nem pedig az éleinek száma szerint értjük.) Változtassuk meg a Bellman–Ford-algoritmust úgy, hogy $m + 1$ menetben befejező djön.

24.1-4. Módosítsuk a Bellman–Ford-algoritmust úgy, hogy az minden olyan v csúcsra, amelyet egy negatív kört tartalmazó úton lehet a kezdőcsúcsból elérni, a $d[v]$ értékét $-\infty$-re állítsa be.

24.1-5.⋆ Legyen $G = (V, E)$ a $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf. Adjunk egy olyan $O(V E)$ futási idejű algoritmust, amelyik minden $v \in V$ csúcsra megtalálja a $\delta^*(v) = \min_{u \in V}\{\delta(u, v)\}$.

24.1-6.⋆ Tegyük fel, hogy egy súlyozott irányított $G = (V, E)$ gráfban van negatív kör. Adjunk hatékony algoritmust egy ilyen körön elhelyezkedő csúcsok kilistázására. Bizonyítsuk be, hogy az algoritmus helyes.

24.2. Adott kezdőcsúcsból induló legrövidebb utak irányított körmentes gráfokban

Ha egy $G = (V, E)$ élsúlyozott KIG (körmentes irányított gráf) éleivel úgy végzünk fokozatos közelítést, hogy az éleket a csúcsok egy topologikus rendezésének megfelelő sorrendjében vizsgáljuk, akkor $\Theta(V + E)$ időben kiszámolhatjuk az adott kezdőcsúcsból induló legrövidebb utakat. Egy KIG-ben a legrövidebb utak mindig jól definiáltak, mivel még ha vannak is negatív súlyú élek, akkor sincs negatív kör.

Az algoritmus a KIG topologikus rendezésével indul (lásd a 22.4. alfejezetet), amely a csúcsokat sorba állítja. Ha van egy út az u csúcsból a v csúcsba, akkor a topologikus sorban u megelőzi a v-t. Végezzünk egy menetet a topologikusan rendezett csúcsok felett. Amint egy csúcsot feldolgozunk, végezzünk közelítést az összes belőle kiinduló éllel.

KIG-BAN-LEGRÖVIDEBB-ÚT(G, w, s)

```
1  A G csúcsainak topologikus rendezése
2  EGY-FORRÁS-KEZDŐÉRTÉK(G, s)
3  for minden u csúcsra azok topologikus sorrendjében
4      do for minden v ∈ Szomszéd[u] csúcsra
5             do KÖZELÍT(u, v, w)
```

Az algoritmus működésére a 24.5. ábra mutat egy példát. Az algoritmus futási idejét könnyű elemezni. Mint azt a 22.4. alfejezetben megmutattuk, az 1. sorbeli topologikus rendezést $\Theta(V + E)$ időben elvégezhetjük. A 2. sorbeli EGY-FORRÁS-KEZDŐÉRTÉK hívása $\Theta(V)$ ideig tart. A 3–5. sorok **for** ciklusa minden csúcsra egyszer hajtódik végre. Mindegyik csúcsra az abból kivezető éleket pontosan egyszer vizsgáljuk meg. Így megkapjuk a 4–5. sorok belső **for** ciklusa $|E|$ iterációjának teljes idejét. (Itt egy összesített elemzést használtunk.) Mivel a belső **for** ciklus mindegyik iterációja $\Theta(1)$ ideig tart, a teljes futási idő $\Theta(V + E)$, amelyik a gráfot reprezentáló szomszédsági lista méretének lineáris függvénye.

A következő tétel azt mutatja meg, hogy a KIG-BAN-LEGRÖVIDEBB-ÚT eljárás helyesen számolja ki a legrövidebb utakat.

24.5. tétel. *Ha egy $G = (V, E)$ élsúlyozott irányított gráfban s a kezdőcsúcs, és nincsenek körök, akkor a* KIG-BAN-LEGRÖVIDEBB-ÚT *eljárás befejeződésekor minden $v \in V$ csúcsra $d[v] = \delta(s, v)$, és a G_π szülő részgráf egy legrövidebb-utak fa.*

Bizonyítás. Először megmutatjuk, hogy minden $v \in V$ csúcsra a $d[v] = \delta(s, v)$ fennáll a befejeződéskor. Ha v nem érhető el az s-ből, akkor $d[v] = \delta(s, v) = \infty$ a nincs-út tulajdonság miatt. Most tegyük fel, hogy v elérhető az s-ből, így van egy $p = \langle v_0, v_1, \ldots, v_k \rangle$ legrövidebb út, ahol $v_0 = s$ és $v_k = v$. Mivel a csúcsokat topologikusan rendezett sorrendben dolgozzuk fel, a p út éleivel történő közelítések a $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ sorrendben történnek. Az út-közelítés tulajdonság következtében minden $i = 0, 1, \ldots, k$-ra a befejeződéskor a $d[v_i] = \delta(s, v_i)$ fennáll. Végül a szülő részgráf tulajdonság alapján látható, hogy a G_π egy legrövidebb-utak fa. ■

A ***PERT diagram*** [2] elemzésének keretében, amikor a kritikus utat kell meghatároznunk, ennek az algoritmusnak egy érdekes alkalmazásával találkozhatunk. Az élek elvégzendő munkafolyamatokat reprezentálnak, és a súlyok az egyes munkák végrehajtásához szükséges időt fejezik ki. Ha egy (u, v) él vezet be a v csúcsba, egy (v, x) pedig kivezet belőle, akkor az (u, v) munkafolyamatnak meg kell előznie a (v, x) munkafolyamatot. Egy út ebben a KIG-ben egy tevékenységsorozatot reprezentál, amelyben az egyes munkákat egy sajátos sorrendben kell elvégezni. A ***kritikus út*** egy leghosszabb KIG-en keresztül vezető út, amely a leghosszabb idejű elvégzendő tevékenység sorozatnak felel meg. Egy kritikus út súlya az alsó határa annak a teljes időnek, amely alatt az összes munkafolyamat elvégezhető. Egy kritikus utat úgy találhatunk meg, hogy vagy

- vesszük az élek súlyainak ellentettjét, és így futtatjuk le a KIG-BAN-LEGRÖVIDEBB-ÚT eljárást, vagy
- lefuttatjuk a KIG-BAN-LEGRÖVIDEBB-ÚT eljárást, de a 2. sorának EGY-FORRÁS-KEZDŐÉRTÉK eljárásában helyettesítjük a ∞ értékeket a $-\infty$ értékkel, és a KÖZELÍT eljárásban a „$>$” jeleket „$<$” jellel.

Gyakorlatok

24.2-1. Futtassuk a KIG-BAN-LEGRÖVIDEBB-ÚT eljárást a 24.5. ábra irányított gráfján, ahol az r csúcs legyen a kezdőcsúcs.

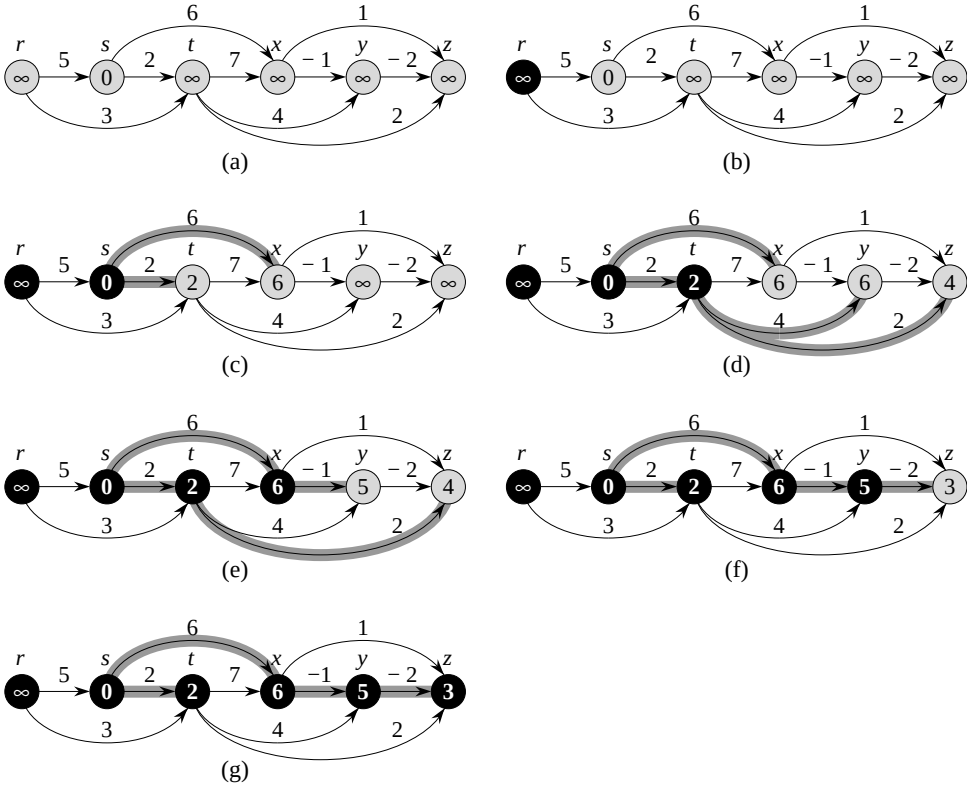
24.2-2. Változtassuk meg a KIG-BAN-LEGRÖVIDEBB-ÚT eljárás 3. sorát úgy, hogy az a

3 **for** az első $|V| - 1$ csúcsra, azok topologikusan rendezett sorrendjében

legyen. Mutassuk meg, hogy az eljárás helyes marad.

24.2-3. A feljebb leírt PERT diagram kicsit mesterkéltnek tűnik. Természetesebb lenne, ha a csúcsok ábrázolnák a munkafolyamatokat, és az élek azok rákövetkezési feltételeit; (u, v) él azt jelezné, hogy az u munka végrehajtásának meg kell előznie a v munkát. A súlyokat a csúcsokhoz rendelnénk, nem pedig az élekhez. Módosítsuk a KIG-BAN-LEGRÖVIDEBB-ÚT eljárást úgy, hogy az a lineáris időben találja meg a leghosszabb utat egy körmentes, irányított gráfban, ahol a csúcsok súlyozottak.

[2] A „PERT” a „program evaluation and review technique” elnevezés mozaikszava.



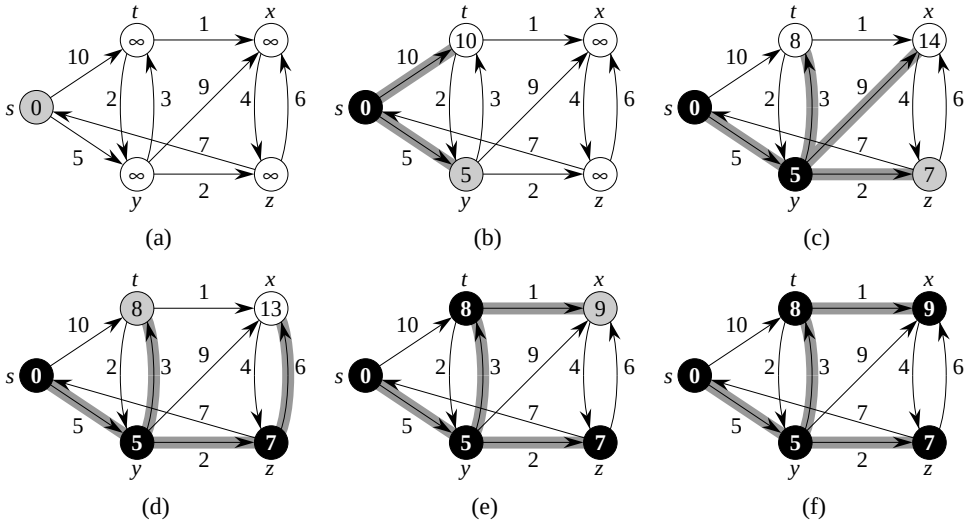
24.5. ábra. Az irányított körmentes gráfok legrövidebb-utak algoritmusának működése. A csúcsok balról jobbra topologikusan rendezettek. A kezdőcsúcs az s. A d értékeket a csúcsokon tüntettük fel, és a vastagított élek jelzik a π értékeket. **(a)** A 3–5. sorok **for** ciklusa előtti helyzet. **(b)–(g)** A 3–5. sorok **for** ciklusának mindegyik iterációja után kialakult állapotok. Az egyes iterációkban befeketített csúcsokat u csúcsként használta az iteráció. A végső értékek a (g) részben láthatók.

24.2-4. Adjunk hatékony algoritmust egy körmentes, irányított gráfbeli utak számának meghatározására. Elemezzük az algoritmust, és jellemezzük sajátosságait.

24.3. Dijkstra algoritmusa

Dijkstra algoritmusa az adott kezdőcsúcsból induló legrövidebb utak problémáját egy élsúlyozott, irányított $G = (V, E)$ gráfban abban az esetben oldja meg, ha egyik élnek sem negatív a súlya. Ebben az alfejezetben ennek megfelelően feltesszük, hogy minden $(u, v) \in E$ élre $w(u, v) \geq 0$. Mint azt látni fogjuk, Dijkstra algoritmusának futási ideje, egy jó megvalósítás mellett, gyorsabb, mint a Bellman–Ford-algoritmusé.

A Dijkstra-algoritmus azoknak a csúcsoknak az S halmazát tartja nyilván, amelyekhez már meghatározta az s kezdőcsúcsból odavezető legrövidebb-út súlyát. Az algoritmus minden lépésben a legkisebb legrövidebb-út becslésű $u \in V - S$ csúcsot választja ki, beteszi az u-t az S-be, és minden u-ból kivezető éllel egy-egy közelítést végez. Az alábbi megvalósí-



24.6. ábra. Dijkstra algoritmusának működése. A kezdőcsúcs a bal oldali csúcs. A legrövidebb-út becsléseket a csúcsok belsejében tüntettük fel, és a vastagított élek jelzik a szülő csúcsokat: ha (u, v) vastag, akkor $\pi[v] = u$. A fekete színű csúcsok az S halmazban vannak, és a fehér színűek a $Q = V - S$ prioritásos sorban. **(a)** Ez a 4–8. sorok **while** ciklusának első iterációja előtti helyzet. A sötét színű csúcs a legkisebb d értékű csúcs, és az 5. sor ezt választja ki u csúcsként. **(b)–(f)** A **while** ciklus soron következő iterációi utáni helyzetek. Minden részben a sötétebb színnel jelölt csúcsot mint u csúcsot választja ki a következő iteráció 5. sora. Az (f) rész a végleges d és π értékeket mutatja.

tásban egy Q minimum-elsőbbségi sort alkalmazunk a $V - S$-beli csúcsok nyilvántartására, amelyeket azok d értékeivel indexelünk. Az algoritmus feltételezi, hogy a G gráf egy szomszédsági listával van megadva.

DIJKSTRA(G, s)

```
1  EGY-FORRÁS-KEZDŐÉRTÉK(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← KIVESZ-MIN(Q)
6           S ← S ∪ {u}
7           for minden v ∈ Szomszéd[u]-ra
8                 do KÖZELÍT(u, v, w)
```

A Dijkstra-algoritmus a 24.6. ábrán bemutatott módon végzi az élekkel a fokozatos közelítést. Az 1. sor a d és a π értékeinek szokásos kezdeti beállítását végzi, majd a 2. sor az S halmazt teszi üressé. A 4–8. sorok **while** ciklusának minden iterációja előtt fennáll a $Q = V - S$ invariáns állítás. A 3. sor a Q minimum-elsőbbségi sort készíti elő úgy, hogy az kezdetben minden V-beli csúcsot tartalmazzon; mivel ekkor az S még üres, az invariáns állítás a 3. sor után teljesül. A 4–8. sorok **while** ciklusában egy u csúcsot veszünk ki az $Q = V - S$ halmazból (legelőször $u = s$), és hozzáadjuk az S halmazhoz, tehát az invariáns állítás továbbra is igaz marad. Az u csúcs tehát a legkisebb legrövidebb-út becslésű csúcs a

$V - S$-ben. Ezután a 7–8. sorok közelítést végeznek az u-ból kivezető (u, v) élekkel, ezáltal módosítják a $d[v]$ becslést és a $\pi[v]$ szülőt, feltéve, hogy a v-hez az u-n keresztül most talált út rövidebb, mint az ott eddig nyilvántartott legrövidebb út. Figyelembe véve azt, hogy a 3. sor után már egyetlen csúcsot sem teszünk bele a Q-ba, valamint azt, hogy mindegyik csúcsot egyetlenegyszer veszünk ki a Q-ból és tesszük át az S-be, a 4–8. sorok **while** ciklusa pontosan $|V|$-szer hajtódik végre.

A Dijkstra-algoritmus mohó stratégiát alkalmaz, hiszen mindig a „legkönnyebb", a „legközelebbi" csúcsot választja ki a $V - S$-ből, hogy azután betegye az S halmazba. A mohó stratégiákat a 16. fejezetben mutattuk be részletesen, de nem szükséges azt a fejezetet elolvasni ahhoz, hogy megértsük Dijkstra algoritmusát. A mohó stratégiák általában nem mindig adnak optimális eredményt, de amint a következő tétel és annak következménye mutatja, a Dijkstra-algoritmus szükségszerűen a legrövidebb utakat állítja elő. A bizonyítás annak a ténynek a bemutatásán múlik, hogy egy S halmazba betett u csúcsra a legrövidebb-út becslés már a $d[u] = \delta(s, u)$ legrövidebb-út súly lesz.

24.6. tétel (Dijkstra-algoritmus helyessége). *Ha Dijkstra algoritmusát egy nemnegatív w súlyfüggvénnyel súlyozott, s kezdőcsúcsú irányított $G = (V, E)$ gráfban futtatjuk, akkor annak a befejeződésekor minden $u \in V$ csúcsra teljesül, hogy $d[u] = \delta(s, u)$.*

Bizonyítás. Az alábbi ciklusinvariánst fogjuk használni:

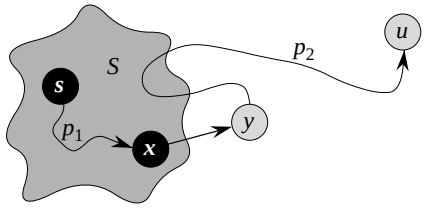
> A 4–8. sorok **while** ciklusának minden iterációja előtt bármely $v \in S$ csúcsra fennáll a $d[u] = \delta(s, u)$.

Elegendő megmutatni, hogy minden $u \in V$ csúcsra, amikor az u bekerül az S halmazba, teljesül a $d[u] = \delta(s, u)$. Ha ugyanis a $d[u] = \delta(s, u)$ megvalósul, akkor a felső-korlát tulajdonság miatt ez az egyenlőség mindvégig meg is marad.

Teljesül: Kezdetben $S = \emptyset$, és így a invariáns magától értetődően teljesül.

Megmarad: Megmutatjuk, hogy az egyes iterációkban az S halmazba betett u csúcsra fennáll a $d[u] = \delta(s, u)$ egyenlőség. Tegyük fel indirekt módon, hogy u az első olyan csúcs, amelyre az S halmazba történő beillesztésekor $d[u] \neq \delta(s, u)$. Vegyük szemügyre a **while** ciklus azon iterációját, amelyben az u csúcs az S halmazba kerül. Vezessük le a $d[u] = \delta(s, u)$ ellentmondást úgy, hogy megmutatjuk, hogy ekkor egy s-ből u-ba vezető legrövidebb út vizsgálatára kerül sor. Az $u \neq s$, mert s-t elsőként illesztjük be az S halmazba, és akkor $d[s] = \delta(s, s) = 0$. Mivel $u \neq s$, feltehetjük, hogy közvetlenül az u beillesztése előtt az $S \neq \emptyset$. Kell lenni továbbá egy s-ből u-ba vezető útnak, máskülönben a nincs-út tulajdonság miatt $d[u] = \delta(s, u) = \infty$, ami ellentmondana a $d[u] \neq \delta(s, u)$ feltételezésünknek. Mivel van legalább egy út, van egy p legrövidebb út is az s-ből az u-ba. Az u csúcs S halmazba helyezése előtt a p út összeköt egy S-beli csúcsot, nevezetesen az s-t, egy $V - S$-beli csúccsal, nevezetesen az u-val. Tekintsük az első olyan csúcsot a p úton, amelyik már $V - S$ halmazban van, jelöljük ezt y-nal, és legyen az y-nak a p úton fekvő szülője az x csúcs. Ekkor, mint azt a 24.7. ábra is mutatja, a p utat felbonthatjuk $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$-ra. (A p_1 és a p_2 utak egyike lehet üres is.)

Azt állítjuk, hogy az u S-be helyezésekor $d[y] = \delta(s, y)$. Ahhoz, hogy ezt bizonyítsuk, vessünk egy pillantást az $x \in S$ csúcsra. Mivel u az első olyan csúcs, amely $d[u] \neq \delta(s, u)$ érték mellett kerül be az S-be, a $d[x] = \delta(s, x)$ biztos fennállt, amikor az x az



24.7. ábra. A 24.6. tétel bizonyítása. Az S halmaz közvetlenül az u csúcs beillesztése előtt nem üres. Egy legrövidebb s kezdőcsúcsból a v csúcsba vezető p út felbontható az $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$-ra, ahol y az első olyan csúcs ezen az úton, amelyik nincs az S-ben, és az $x \in S$ közvetlenül megelőzi y-t. Az x és y csúcsok különböznek, de lehetséges, hogy $s = x$ vagy $y = u$. A p_2 út lehet, hogy érinti, de lehet, hogy elkerüli az S halmazt.

S-be került. Ekkor került sor az (x, y) éllel történő közelítésre, így tehát az állításunk a konvergencia tulajdonság következménye.

A tétel bizonyításához szükséges ellentmondáshoz mutassuk meg, hogy $d[u] = \delta(s, u)$. Mivel az y egy s-ből u-ba vezető legrövidebb úton az u előtt fordul elő, és az összes él súlya nemnegatív (különös tekintettel a p_2 úton lévőkre), továbbá a $\delta(s, y) \leq \delta(s, u)$ teljesül, így

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \qquad \text{(felső-korlát tulajdonság miatt).} \end{aligned} \tag{24.2}$$

De mivel az u is és az y is a $V - S$-ben van, ahonnan az 5. sor az u-t választja ki, fennáll, hogy $d[u] \leq d[y]$. Így a (24.2) két egyenlőtlensége valójában egyenlőség

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

Következésképpen, $d[u] = \delta(s, u)$, amely ellentmond az általunk választott u-nak. Tehát amikor egy u csúcsot beteszünk az S halmazba, akkor már $d[u] = \delta(s, u)$, és ez az egyenlőség a későbbiek során sem változik meg.

Befejeződik: Befejeződéskor $Q =$, és ez a korábbi $Q = V - S$ invariánst figyelembe véve azt vonja maga után, hogy $S = V$. Így minden $u \in V$ csúcsra $d[u] = \delta(s, u)$. ■

24.7. következmény. *Ha Dijkstra algoritmusát egy nemnegatív w súlyfüggvénnyel súlyozott, irányított s kezdőcsúcsú $G = (V, E)$ gráfban futtatjuk, akkor annak befejeződésekor a G_π szülő részgráf egy s gyökerű legrövidebb-utak fa lesz.*

Bizonyítás. Közvetlenül adódik a 24.6. tételből és a szülő részgráf tulajdonságból. ■

Elemzés

Milyen gyors a Dijkstra-algoritmus? Tudjuk róla, hogy egy Q minimum-elsőbbségi sort használ, amelyre háromféle műveletet hív meg: a BESZÚR (a 3. sor utasításában van elrejtve), a KIVESZ-MIN (az 5. sorban) és a KULCSOT-CSÖKKENT (a 8. sorban meghívott KÖZELÍT eljárásban van elrejtve) műveleteket. A BESZÚR, akárcsak a KIVESZ-MIN csúcsonként egyszer kerül

meghívásra. Amiért az algoritmus működése alatt minden $v \in V$ csúcs pontosan egyszer kerül át az S halmazba, ezért a *Szomszéd*[v] szomszédsági lista mindegyik élét is pontosan egyszer vizsgálja meg 7–8. sorok **for** ciklusa. Mivel az összes szomszédsági lista együttesen $|E|$ élt tartalmaz, ez a **for** ciklus $|E|$ iterációt végez, és így a KULCSOT-CSÖKKENT művelet legfeljebb $|E|$-szer hívódik meg. (Felhívjuk újra a figyelmet, hogy most összesített elemzést alkalmazunk.)

A Dijkstra-algoritmus futási ideje attól függ, hogyan valósítjuk meg a minimum-elsőbbségi sort. Vegyük először azt az esetet, amikor a minimum-elsőbbségi sor kezelésénél kihasználjuk, hogy a csúcsokat meg lehet számozni 1-től $|V|$-ig. Ekkor a $d[v]$ értéket egyszerűen tömb v-edik pozícióján tároljuk. Mindegyik BESZÚR és KULCSOT-CSÖKKENT művelet $O(1)$ ideig tart, és minden KIVESZ-MIN művelet $O(V)$ ideig (mivel az egész tömböt végig kell nézni). A teljes futási idő tehát $O(V^2 + E) = O(V^2)$.

Ha azonban eléggé ritka a gráf – nevezetesen, $E = o(V^2/\lg V)$ –, akkor a minimum-elsőbbségi sort előnyösebb bináris kupaccal megvalósítani. (Amint azt a 6.5. alfejezetben megvizsgáltuk, egy fontos megvalósítási részlet az, hogy a csúcsokat és a megfelelő indexeket együtt kell kezelni.) Ekkor mindegyik KIVESZ-MIN művelet futási ideje $O(\lg V)$. Akárcsak előbb, $|V|$ ilyen műveletre kerül sor. A bináris kupac felépítésének ideje $O(V)$. Mindegyik KULCSOT-CSÖKKENT művelet $O(\lg V)$ idejű, és $|E|$ ilyen művelet van. A teljes futási idő ezért $O((V + E) \lg V)$, ami $O(E \lg V)$, ha mindegyik csúcs elérhető a kezdőcsúcsból. Ez a futási idő $E = o(V^2/\lg V)$ esetén jobb, mint az előbb látott $O(V^2)$.

Valójában a $O(V \lg V + E)$ futási időt is elérhetjük, ha a minimum-elsőbbségi sort Fibonacci-kupac segítségével implementáljuk (lásd a 20. fejezetet). A $|V|$ KIVESZ-MIN művelet mindegyikének amortizációs ideje $O(\lg V)$, és a legfeljebb $|E|$ darab KULCSOT-CSÖKKENT művelet mindegyike pedig $O(1)$ amortizációs idejű. A Fibonacci-kupacok felfedezését történetileg az motiválta, hogy megfigyelték, hogy a Dijkstra-algoritmusban potenciálisan sokkal több KULCSOT-CSÖKKENT művelet van, mint KIVESZ-MIN, így bármelyik olyan módszer, amely minden KULCSOT-CSÖKKENT műveletet $o(\lg V)$ amortizációs idejűre redukál, miközben az KIVESZ-MIN amortizációs ideje nem növekszik, aszimptotikusan gyorsabb implementációt eredményez.

A Dijkstra-algoritmus bizonyos hasonlóságot mutat mind a szélességi kereséssel (lásd a 22.2. alfejezetet), mind a minimális feszítőfát előállító Prim-algoritmussal (lásd a 23.2. alfejezetet). A szélességi kereséshez annyiban, amennyiben az S halmaz a szélességi keresés fekete csúcsai halmazának felel meg; az S-ben levő csúcsoknak már ismert a legrövidebb-út súlya, akárcsak a szélességi keresés fekete csúcsainál azok helyes szélességikeresés-távolsága. A Dijkstra-algoritmus ugyanakkor hasonlít a Prim-algoritmusra, amennyiben mindkét algoritmus egy elsőbbségi sort használ azoknak a csúcsoknak a tárolására, amelyek egy adott halmazon kívül (a Dijkstra-algoritmusban az S halmazon kívül, a Prim-algoritmusban a fokozatosan növelt fán kívül) helyezkednek el, ahonnét a „legkönnyebb” csúcsot kiválasztják, majd beillesztik a halmazba, és a halmazon kívül maradt csúcsok súlyait a kialakult helyzetnek megfelelően kijavítják.

Gyakorlatok

24.3-1. Futtassuk Dijkstra algoritmusát a 24.2. ábra irányított gráfján, először az s csúcsot használva kezdőcsúcsként, aztán pedig az y csúcsot. Mutassuk be a 24.2. ábrán látott módon azt, hogyan változnak a d és π értékek, valamint az S halmaz csúcsai a **while** ciklus egyes iterációi nyomán.

24.3-2. Adjunk egy egyszerű példát egy negatív súlyú éleket is tartalmazó olyan irányított gráfra, amelyen a Dijkstra-algoritmus helytelen választ ad. Miért nem vihető át a 24.10. tétel bizonyítása a negatív súlyú élek esetére?

24.3-3. Tegyük fel, hogy a Dijkstra-algoritmus 4. sorát az alábbi módon megváltoztatjuk.

4 **while** $|Q| > 1$

Ez a változtatás eggyel csökkenti a ciklus iterációinak számát. Helyes-e az így javított algoritmus?

24.3-4. Adott egy irányított $G = (V, E)$ gráf, amelynek minden $(u, v) \in E$ éle rendelkezik egy valós $0 \leq r(u, v) \leq 1$ kapcsolat értékkel, amely az u csúcsból a v csúcsba vezető kommunikációs csatorna megbízhatóságát fejezi ki. Az $r(u, v)$-t annak a valószínűségeként értelmezhetjük, hogy az u-ból a v-be vezető csatorna nem sérül meg, és feltesszük, hogy ezek a valószínűségek függetlenek. Adjunk hatékony algoritmust két adott csúcs közötti legmegbízhatóbb út előállítására.

24.3-5. Legyen $G = (V, E)$ egy $w : E \to \{0, 1, \ldots, W - 1\}$ súlyfüggvénnyel súlyozott irányított gráf, ahol W egy pozitív egész szám, és tegyük fel, hogy bármelyik két csúcsnak az s kezdőcsúcsból számított legrövidebb-út súlya különbözik. Most definiáljunk egy $G' = (V \cup V', E')$ súlyozatlan irányított gráfot úgy, hogy mindegyik $(u, v) \in E$ élt lecserélünk egy $w(u, v)$ darab egységnyi súlyú élből álló úttal. Hány csúcsa van a G'-nek? Most futtassuk le a szélességi keresést a G' gráfra. Mutassuk meg, hogy a G' feletti szélességi keresés ugyanabban a sorrendben színezi feketére a V-beli csúcsokat, mint ahogy a G felett működő Dijkstra-algoritmus az 5. sorban kiveszi a V-beli csúcsokat az elsőbbségi sorból.

24.3-6. Legyen $G = (V, E)$ egy $w : E \to \{0, 1, \ldots, W - 1\}$ súlyfüggvénnyel súlyozott irányított gráf, ahol W nemnegatív egész szám. Módosítsuk Dijkstra algoritmusát úgy, hogy az $O(WV + E)$ idő alatt állítsuk elő egy adott s kezdőcsúcsból induló legrövidebb utakat.

24.3-7. Módosítsuk a 24.2-5. gyakorlatban kapott algoritmust $O((V + E) \lg W)$ futási idejűre. (*Útmutatás.* Hány különböző legrövidebb-út becslés lehetséges a $V - S$-ben az algoritmus egy adott pillanatában?)

24.3-8. Tegyük fel, hogy egy élsúlyozott irányított $G = (V, E)$ gráfban az s kezdőcsúcsból kivezető élek lehetnek negatív súlyúak, de minden más él nemnegatív súlyú, és nincsenek a gráfban negatív súlyú körök. Igaz-e, hogy a Dijkstra-algoritmus ebben a gráfban is megtalálja az s-ből kivezető legrövidebb utakat?

24.4. Különbségi korlátok és legrövidebb utak

A 29. fejezet foglalkozik az általános lineáris programozási problémával, amelynek a megoldásához egy lineáris függvényt kell optimalizálni úgy, hogy közben lineáris egyenlőtlenségek egy halmazát is ki kell elégíteni. Ebben az alfejezetben a lineáris programozás egy speciális esetét vizsgáljuk meg, amelyet vissza lehet vezetni egy adott kezdőcsúcsból induló legrövidebb utak problémára. A Bellman–Ford-algoritmus, amelyik egy adott kezdőcsúcsból induló legrövidebb utak problémát old meg, megoldja ezt a lineáris programozási problémát is.

Lineáris programozás

Az általános ***lineáris programozási problémában*** adott egy $m \times n$-es A mátrix, egy m elemű b vektor és egy n elemű c vektor. Egy olyan n elemű x vektor megtalálása a célunk, amelyik maximalizálja a $\sum_{i=1}^{n} c_i x_i$ ***célfüggvényt,*** és közben megfelel az $Ax \leq b$ által kijelölt m darab korlátnak.

Habár a szimplex algoritmus, amelyet a 29. fejezetben vizsgálunk majd meg, nem mindig fut le a bemenő adatainak méretéhez viszonyított polinomiális idő alatt, vannak más lineáris programozási algoritmusok, amelyek polinomiális futási idejűek. Több okból is fontos, hogy megértsük a lineáris programozási feladatokat. Egyrészt, ha egy adott problémát polinomiális idejű lineáris programozási feladatként tudunk modellezni, akkor abból közvetlenül adódik, hogy a probléma megoldására van polinomiális idejű algoritmus. Másrészt a lineáris programozásnak vannak olyan speciális esetei, amelyekre gyorsabb megoldó algoritmusok is léteznek. Például, mint azt ebben az alfejezetben is meg fogjuk mutatni, az adott kezdőcsúcsból induló legrövidebb utak probléma egy speciális esete a lineáris programozásnak. Más, lineáris programozási feladatként megfogalmazható probléma még az adott csúcspárok közötti legrövidebb út probléma (24.4-4. gyakorlat) vagy a maximális folyam probléma (26.1-8. gyakorlat).

Néha egyáltalán nem foglalkozunk a célfüggvénnyel; csak egy ***megengedett megoldást*** szeretnénk találni, azaz egy olyan x vektort, amely kielégíti az $Ax \leq b$ korlátokat, vagy belátni azt, hogy nem létezik megengedett megoldás. Egy ilyen ***megengedett feladattal*** foglalkozunk majd.

Különbségi korlátok rendszere

Egy ***különbségi korlátok rendszerében*** a lineáris programozás A mátrixának mindegyik sora egyetlen 1-es, egyetlen -1-es érték kivételével csupa 0 értéket tartalmaz. Ezért a megadott $Ax \leq b$ korlátok m darab ***különbségi korlátot*** alkotnak, amelyekben n számú ismeretlen van, és mindegyik korlát egy

$$x_j - x_i \leq b_k$$

formájú egyszerű lineáris egyenlőtlenség, ahol $1 \leq i, j \leq n$ és $1 \leq k \leq m$.

Példaként tekintsük annak az 5 elemű $x = (x_i)$ vektornak a keresését, amelyik kielégíti az

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

korlát rendszert. Ez a feladat ekvivalens azzal, amikor minden $i = 1, 2, \ldots, 5$-re azokat az x_i ismeretleneket keressük, amelyek kielégítik az

$$x_1 - x_2 \leq 0, \tag{24.3}$$

$$x_1 - x_5 \leq -1, \tag{24.4}$$

$$x_2 - x_5 \leq 1, \tag{24.5}$$
$$x_3 - x_1 \leq 5, \tag{24.6}$$
$$x_4 - x_1 \leq 4, \tag{24.7}$$
$$x_4 - x_3 \leq -1, \tag{24.8}$$
$$x_5 - x_3 \leq -3, \tag{24.9}$$
$$x_5 - x_4 \leq -3 \tag{24.10}$$

különbségi korlátokat. Ennek a feladatnak az $x = (-5, -3, 0, -1, -4)$ az egyik megoldása, amit az egyes egyenlőtlenségekbe történő közvetlen behelyettesítéssel ellenőrizhetünk. Valójában több megoldása is van ennek a feladatnak. Egy másik az $x' = (0, 2, 5, 4, 1)$. Ezek a megoldások összefüggnek: Az x' mindegyik komponense 5-tel nagyobb az x megfelelő komponensénél. Ez nem a puszta véletlen műve.

24.8. lemma. *Legyen $x = (x_1, x_2, \ldots, x_n)$ egy megoldása egy $Ax \leq b$ különbségi korlátrendszernek, és legyen d egy tetszőleges konstans. Ekkor $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ ugyancsak megoldása az $Ax \leq b$-nek.*

Bizonyítás. Mindegyik x_i és x_j-re az $(x_j + d) - (x_i + d) = x_j - x_i$. Ennek megfelelően, ha x kielégíti az $Ax \leq b$-t, akkor $x + d$ is kielégíti azt. ■

A különbségi korlátok rendszerei számos különféle alkalmazásban fordulnak elő. Például az x_i ismeretlenek lehetnek olyan időpillanatok, amikor valamilyen eseménynek be kell következni. Az egyes korlátokat úgy lehet értelmezni, mint amelyek azt fejezik ki, hogy két esemény közötti időnek legalább, illetve legfeljebb mennyinek kell lenni. Lehetnek az események olyan munkafolyamatok, amelyeket egy termék összeállításánál kell végrehajtani. Ha az x_1 időpillanatban egy 2 óra alatt megkötő ragasztóanyagot alkalmazunk, és az x_2 időpillanatban az összeragasztott elemeket már fel kell használnunk, akkor az $x_2 \leq x_1 + 2$, vagy ami ezzel ekvivalens a $x_1 - x_2 \leq -2$ korláttal kell számolnunk. Vagy megkövetelhetjük azt, hogy az összeragasztott elemeket a ragasztás után akarjuk felhasználni, de még a ragasztó száradási ideje felének letelte előtt. Ebben az esetben az $x_2 \geq x_1$ és az $x_2 \leq x_1 + 1$ korlát párt kapjuk, vagy az ezzel egyenértékű $x_1 - x_2 \leq 0$ és $x_1 - x_2 \leq 1$ korlátokat. Előnyös, ha a különbségi korlátok rendszereit gráfelméleti nézőpontból szemléljük. Az ötlet az, hogy egy $Ax \leq b$ különbségi korlátrendszerben az $n \times m$-es A lineáris programozási mátrixot egy n csúcspontú és m élű gráf szomszédsági mátrixának (lásd 22.1-7. gyakorlat) tekintjük. A gráf minden v_i csúcsát ($i = 1, 2, \ldots, n$) megfeleltetjük az n darab x_i ismeretlen egyikének. A gráf mindegyik irányított éle az m darab két ismeretlent tartalmazó egyenlőtlenség valamelyikét jelöli.

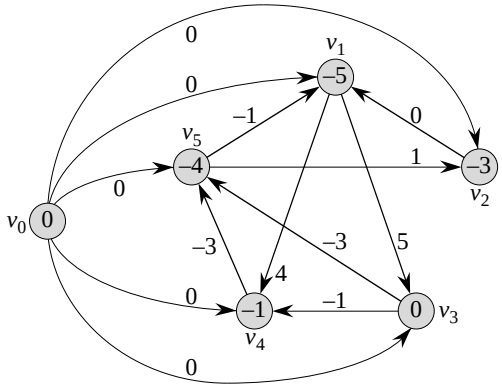
Formálisan fogalmazva, egy $Ax \leq b$ különbségi korlátrendszernek megfeleltetett ***korlátgráf*** egy $G = (V, E)$ élsúlyozott irányított gráf, ahol

$$V = \{v_0, v_1, \ldots, v_n\}$$

és

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ egy korlát}\} \cup \{(v_0, v_1), (v_0, v_2)(v_0, v_3), \ldots, (v_0, v_n)\}.$$

A v_0 csúcsot azért vettük fel, mint azt rövidesen látni fogjuk, hogy minden más csúcs elérhető legyen belőle. Ennek megfelelően a V halmaz minden x_i ismeretlenre egy v_i csúcsot, és



24.8. ábra. A (24.3)–(24.10) különbségi korlátok rendszerének megfelelő korlátgráf. A $\delta(v_0, v_i)$ értékeket a v_i csúcsokban tüntettük fel. A rendszer egy megoldása az $x = (-5, -3, 0, -1, -4)$.

egy további v_0 csúcsot tartalmaz. Az E élhalmaz mindegyik különbségi korláthoz tartalmaz egy élt, és az összes x_i ismeretlenre egy (v_0, v_i) élt. Ha $x_j - x_i \leq b_k$ különbségi korlát, akkor a (v_i, v_j) él költsége $w(v_i, v_j) = b_k$. A v_0-ból kivezető összes él költsége 0. A 24.8. ábra a (24.3)–(24.10) különbségi korlátrendszer korlátgráfját mutatja be.

A következő tétel megmutatja, hogy egy különbségi korlátrendszer egy megoldását a megfelelő korlátgráf legrövidebb-út súlyainak megtalálásával kaphatjuk meg.

24.9. tétel. *Adott egy $Ax \leq b$ különbségi korlátrendszer, legyen $G = (V, E)$ a megfelelő korlátgráf. Ha G nem tartalmaz negatív kört, akkor az*

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \tag{24.11}$$

egy megoldása a rendszernek. Ha G tartalmaz negatív kört, akkor nincs megoldás.

Bizonyítás. Először azt mutatjuk meg, hogy ha a gráf nem tartalmaz negatív köröket, akkor a (24.11) egyenlőség egy megoldást ad. Vegyünk egy tetszőleges $(v_i, v_j) \in E$ élt. A háromszög-egyenlőtlenség tulajdonság miatt $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$, azaz másként $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Ekkor az $x_i = \delta(v_0, v_i)$ és az $x_j = \delta(v_0, v_j)$ választás kielégíti a (v_i, v_j) élnek megfeleltetett $x_j - x_i \leq w(v_i, v_j)$ különbségi korlátot.

Most megmutatjuk, hogy ha a gráf negatív köröket tartalmaz, akkor a különbségi korlátrendszernek nincs megoldása. Az általánosság megsértése nélkül feltehetjük, hogy a $c = (v_1, v_2, \ldots, v_k)$ egy kör, ahol $v_1 = v_k$. (A v_0 csúcs nem szerepelhet a c körön, mert nincs hozzá vezető él.) A c kör az alábbi különbségi korlátoknak felel meg:

$$\begin{aligned}
x_2 - x_1 &\leq w(v_1, v_2), \\
x_3 - x_2 &\leq w(v_2, v_3), \\
&\vdots \\
x_{k-1} - x_{k-2} &\leq w(v_{k-2}, v_{k-1}), \\
x_k - x_{k-1} &\leq w(v_{k-1}, v_k).
\end{aligned}$$

Tegyük fel, hogy van egy x megoldás, amely kielégíti ezt a k darab egyenlőtlenséget csakúgy, mint ezen egyenlőtlenségek összegét. Ha összeadjuk a bal oldalakat, minden x_i ismeretlen egyszer fog pozitív, és egyszer negatív előjellel szerepelni (ne felejtsük el, hogy $v_1 = v_k$

miatt $x_1 = x_k$ adódik), így a bal oldal összege 0 lesz. A jobb oldalak összege a $w(c)$, tehát azt kapjuk, hogy $0 \leq w(c)$. De a c egy negatív súlyú kör, $w(c)\langle 0$, és ezáltal a $0 \leq w(c)\langle 0$ ellentmondásra jutottunk. ■

Különbségi korlátrendszerek megoldása

A 24.9. tétel azt sugallja, hogy egy különbségi korlátrendszer megoldására alkalmazzuk a Bellman–Ford-algoritmust. Mivel a v_0 kezdőcsúcsból az összes többi csúcsba vezet él a korlátgráfban, bármelyik negatív súlyú kör elérhető a v_0-ból. Ha a Bellman–Ford-algoritmus IGAZ értékkel tér vissza, akkor a legrövidebb-út súlyok egy megoldását adják a rendszernek. A 24.8. ábrán például a legrövidebb-út súlyok biztosítják az $x = (-5, -3, 0, -1, -4)$ megoldást, és a 24.8. lemma miatt bármely d konstansra az $x = (d - 5, d - 3, d, d - 1, d - 4)$ is egy megoldás lesz. Ha a Bellman–Ford-algoritmus HAMIS értékkel tér vissza, akkor nincs megoldása a különbségi korlátrendszernek.

Egy m korlátot és n ismeretlent tartalmazó különbségi korlátrendszernek egy $n + 1$ csúcsot és $n + m$ élt tartalmazó korlátgráf felel meg. Ennek alapján a Bellman–Ford-algoritmus $O((n + 1)(n + m)) = O(n^2 + nm)$ idő alatt oldhatja meg a rendszert. A 24.4-5. gyakorlat azt kéri, hogy mutassuk meg, hogy az algoritmus valójában $O(nm)$ futási idejű, feltéve, hogy az m sokkal kisebb, mint az n.

Gyakorlatok

24.4-1. Döntsük el, hogy létezik-e megoldása az alábbi különbségi korlátrendszernek, és ha igen, keressünk egyet.

$$\begin{aligned}
x_1 - x_2 &\leq 1,\\
x_1 - x_4 &\leq -4,\\
x_2 - x_3 &\leq 2,\\
x_2 - x_5 &\leq 7,\\
x_2 - x_6 &\leq 5,\\
x_3 - x_6 &\leq 10,\\
x_4 - x_2 &\leq 2,\\
x_5 - x_1 &\leq -1,\\
x_5 - x_4 &\leq 3,\\
x_6 - x_3 &\leq -8.
\end{aligned}$$

24.4-2. Döntsük el, hogy létezik-e megoldása az alábbi különbségi korlátrendszernek, és ha igen, keressünk egyet.

$$\begin{aligned}
x_1 - x_2 &\leq 4,\\
x_1 - x_5 &\leq 5,\\
x_2 - x_4 &\leq -6,\\
x_3 - x_2 &\leq 1,\\
x_4 - x_1 &\leq 3,\\
x_4 - x_3 &\leq 5,\\
x_4 - x_5 &\leq 10,\\
x_5 - x_3 &\leq -4,\\
x_5 - x_4 &\leq -8.
\end{aligned}$$

24.4-3. Lehet-e a v_0 új csúcsból induló bármelyik legrövidebb út súlya pozitív? Indokoljuk.

24.4-4. Fejezzük ki az adott csúcs pár között vezető legrövidebb út problémát lineáris programozási feladatként.
24.4-5. Mutassuk meg, hogyan módosítható egy kissé a Bellman–Ford-algoritmus úgy, hogy amikor egy m egyenlőtlenséget és n ismeretlent tartalmazó különbségi korlátrendszert oldunk meg, akkor a futási ideje $O(nm)$ legyen.
24.4-6. Tegyük fel, hogy egy különbségi korlátrendszer mellett $x_i = x_j + b_k$ alakú ***egyenlőségi korlátokat*** is kezelni akarunk. Mutassuk meg, hogyan adaptálhatjuk a Bellman–Ford-algoritmust a korlátrendszer ilyen változatának megoldására.
24.4-7. Mutassuk meg, hogyan lehet egy különbségi korlátrendszert egy olyan Bellman–Ford-szerű algoritmussal megoldani, amelyik az extra v_0 csúcsot nem tartalmazó korlátgráfon működik.
24.4-8.⋆ Legyen $Ax \leq b$ egy m különbségi korlátot és n ismeretlent tartalmazó rendszer. Mutassuk meg, hogy amikor a Bellman–Ford-algoritmus a megfelelő korlátgráfon fut, akkor maximalizálja a $\sum_{i=1}^{n} x_i$-t az $Ax \leq b$ és minden x_i-re az $x_i \leq 0$ feltételek fenntartása mellett.
24.4-9.⋆ Mutassuk meg, hogy amikor a Bellman–Ford-algoritmus egy $Ax \leq b$ különbségi korlátrendszernek megfelelő korlátgráfon fut, akkor minimalizálja a $(max\{x_i\} - min\{x_i\})$ mennyiséget az $Ax \leq b$ feltétel fenntartása mellett. Magyarázzuk meg, miért jön ez jól, ha az algoritmust a munkafolyamatok beosztásához használjuk.
24.4-10. Tegyük fel, hogy egy $Ax \leq b$ lineáris programozási feladat A mátrixának minden sora egy olyan különbségi korlátnak felel meg, amelyik egy $x_i \leq b_k$ formájú vagy egy $-x_i \leq b_k$ formájú egyváltozós korlát. Mutassuk meg, hogyan adaptálható a Bellman–Ford-algoritmus az ilyen korlátokat tartalmazó rendszer megoldására.
24.4-11. Adjunk hatékony algoritmust egy $Ax \leq b$ különbségi korlátrendszer megoldására, amikor a b minden eleme valós értékű, és az összes x_i ismeretlennek egésznek kell lenni.
24.4-12.⋆ Adjunk hatékony algoritmust egy $Ax \leq b$ különbségi korlátrendszer megoldására, amikor a b minden eleme valós értékű, és valamelyik, de nem szükségképpen az összes x_i ismeretlennek kell egésznek lenni.

24.5. A legrövidebb utak tulajdonságainak bizonyítása

Ennek a fejezetnek a helyesség bizonyításainál mindenütt támaszkodtunk a háromszög-egyenlőtlenségre, a felső-korlát tulajdonságra, a nincs-út tulajdonságra, a konvergencia tulajdonságra, az út-közelítés tulajdonságra és a szülő részgráf tulajdonságra. Ezeket a tulajdonságokat a fejezet elején bizonyítás nélkül fogalmaztuk meg. Ebben az alfejezetben bebizonyítjuk őket.

Háromszög-egyenlőtlenség

A szélességi keresés tanulmányozásánál (22.2. alfejezet) a legrövidebb távolságok egyszerű tulajdonságát bizonyítottuk be súlyozatlan gráfokra. A háromszög-egyenlőtlenség ennek a tulajdonságnak súlyozott gráfokra vett általánosítása.

24.10. lemma (háromszög-egyenlőtlenség). *Legyen $G = (V, E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, és s a kezdőcsúcs. Ekkor minden $(u, v) \in E$ élre fennáll,*

hogy

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Bizonyítás. Egy az s kezdőcsúcsból v csúcsba tartó legrövidebb p útnak nem nagyobb a súlya, mint bármely más s-ből v-be vezető útnak. Speciálisan a p útnak nem nagyobb a súlya, mint annak az útnak, amelyiknek egy s kezdőcsúcsból u csúcsba vezető legrövidebb útból és az (u, v) élből áll.

Azt az esetet, amikor nincsen s-ből v-be vezető legrövidebb út, a 24.5-3. gyakorlat keretében tárgyaljuk. ■

Legrövidebb-út becslések fokozatos közelítésének hatásai

A lemmák következő csoportja azt mutatja meg, hogy hogyan befolyásolja a legrövidebb-út becsléseket az, amikor közelítések sorozatát végezzük el az EGY-FORRÁS-KEZDŐÉRTÉK eljárás által előkészített élsúlyozott irányított gráf éleivel.

24.11. lemma (felső-korlát tulajdonság). *Legyen $G = (V, E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol $s \in V$ a kezdőcsúcs. Legyen a gráf az* EGY-FORRÁS-KEZDŐÉRTÉK(G, s) *eljárással előkészítve. Ekkor minden $v \in V$ csúcsra fennáll a $d[v] \geq \delta(s, v)$, és ez az egyenlőtlenség bármilyen G-beli élekkel történő közelítő lépéssorozatra fennmarad. Továbbá, ha a $d[v]$ egyszer eléri a $\delta(s, v)$ alsó határt, akkor már sohasem változik meg az értéke.*

Bizonyítás. A $d[v] \geq \delta(s, v)$ invariáns tulajdonságot az összes $v \in V - \{s\}$ csúcsra a közelítő lépések száma szerinti indukcióval látjuk be.

Kezdetben, a kezdeti értékadások után a $d[v] \geq \delta(s, v)$ nyilván teljesül, mert $d[s] = 0 \geq \delta(s, s)$ (megjegyezzük, hogy $\delta(s, s) = -\infty$, ha s egy negatív súlyú körön van, és 0 különben), és a $d[v] = \infty$ miatt minden $v \in V - \{s\}$ csúcsra a $d[v] \geq \delta(s, v)$ is fennáll.

Az indukciós lépésnél tekintünk egy (u, v) éllel való közelítést. A közelítés előtt az indukciós feltevés miatt minden $v \in V$ csúcsra $d[x] \geq \delta(s, x)$. Az egyetlen d érték, amely megváltozhat a közelítés során, a $d[v]$. Ha ez megváltozik, akkor

$$\begin{aligned} d[v] &= d[u] + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{(indukciós feltevés miatt)} \\ &\geq \delta(s, v) && \text{(háromszög-egyenlőtlenség miatt),} \end{aligned}$$

azaz az invariáns állítás továbbra is igaz.

Ahhoz, hogy azt is lássuk, hogy a $d[v]$ érték sohasem változik meg azután, ha egyszer $d[v] = \delta(s, v)$ bekövetkezik, megjegyezzük, hogy a $d[v]$ az alsó korlátot elérve tovább nem csökkenhet, hiszen megmutattuk, hogy $d[v] \geq \delta(s, v)$, de nem is nőhet, mert a közelítő lépés nem növeli meg a d értékeket. ■

24.12. következmény (nincs-út tulajdonság). *Tegyük fel, hogy egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított $G = (V, E)$ gráfban nem vezet út egy s kezdőcsúcsból egy adott $v \in V$ csúcsba. Ekkor, a gráf* EGY-FORRÁS-KEZDŐÉRTÉK(G, s) *eljárással történő előkészítése után $d[v] = \delta(s, v) = \infty$, és ez az egyenlőség bármilyen G-beli élekkel történő közelítő lépéssorozatra mint invariáns fennmarad.*

Bizonyítás. A felső-korlát tulajdonság szerint mindig fennáll, hogy $\infty = \delta(s,v) \leq d[v]$, így $d[v] = \infty = \delta(s,v)$. ■

24.13. lemma. *Legyen $G = (V,E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, és legyen $(u,v) \in E$. Ekkor közvetlenül a* KÖZELÍT(u,v,w) *által végrehajtott (u,v) éllel történő közelítő lépés után fennáll a $d[v] \leq d[u] + w(u,v)$ egyenlőtlenség.*

Bizonyítás. Ha közvetlenül az (u,v) éllel történő közelítés előtt $d[v] > d[u] + w(u,v)$, akkor utána $d[v] = d[u] + w(u,v)$ lesz. Ha viszont a közelítést megelőzően $d[v] \leq d[u] + w(u,v)$, akkor sem $d[u]$, sem $d[v]$ nem változik, így $d[v] \leq d[u] + w(u,v)$ megmarad. ■

24.14. lemma (konvergencia tulajdonság). *Legyen $G = (V,E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol $s \in V$ egy kezdőcsúcs, és legyen $s \rightsquigarrow u \to v$ egy legrövidebb G-beli út valamely $u, v \in V$ csúcsokra. Tegyük fel, hogy a gráfot az* EGY-FORR ÁS-KEZDŐÉRTÉK(G,s) *eljárással előkészítettük, majd végrehajtunk a G éleivel egy olyan közelítő lépéssorozatot, amely tartalmazza a* KÖZELÍT(u,v,w) *hívását is. Ha $d[u] = \delta(s,u)$ fennáll közvetlenül az eljárás meghívása előtt, akkor utána már mindig teljesül, hogy $d[v] = \delta(s,v)$.*

Bizonyítás. A felső-korlát tulajdonság szerint ha $d[u] = \delta(s,u)$ fennáll az (u,v) élű közelítés előtt közvetlenül, akkor ez fennmarad azután is. Részletesebben, az (u,v) élű közelítés után fennáll, hogy

$$\begin{aligned} d[v] &\leq d[u] + w(u,v) && \text{(24.13. lemma miatt)} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) && \text{(24.1. következmény miatt).} \end{aligned}$$

A felső-korlát tulajdonság szerint $d[v] \geq \delta(s,v)$, amelyből az következik, hogy $d[v] = \delta(s,v)$, és ez az egyenlőség később is fennmarad. ■

24.15. lemma (út-közelítés tulajdonság). *Legyen $G = (V,E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol $s \in V$ egy kezdőcsúcs. Tekintsünk egy tetszőleges $p = \langle v_0, v_1, \ldots, v_k \rangle$ $s = v_0$-ból v_k-ba vezető legrövidebb utat. Ha a gráfot az* EGY-FORR ÁS-KEZDŐÉRTÉK(G,s) *eljárással előkészítettük, és aztán olyan közelítő lépéssorozat hajtódott végre, amely magában foglalta a $(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1},v_k)$ élek közelítéseit az itt megadott sorrendben, akkor ezen közelítések után bekövetkezik $d[v_k] = \delta(s,v_k)$ egyenlőség, és ez később is fennmarad. Ez a tulajdonság nem függ attól, hogy milyen más élekkel történik közben közelítés, legyenek azok tetszőlegesen összekeverve a p út éleivel történő közelítésekkel.*

Bizonyítás. Indukcióval megmutatjuk, hogy a p út i-edik élével való közelítés után a $d[v_i] = \delta(s,v_i)$ bekövetkezik. Kezdetben, $i = 0$ esetén, mielőtt a p bármelyik élével közelítést végeztünk volna, a kezdeti értékadások miatt $d[v_0] = d[s] = \delta(s,s)$ teljesül. A felső-korlát tulajdonság miatt a $d[s]$ soha sem változik meg az inicializáló lépés után.

Az indukciós lépésnél feltesszük, hogy $d[v_{i-1}] = \delta(s,v_{i-1})$, és megvizsgáljuk a $(v_{[}i-1], v_i)$ éllel történő közelítést. Ezen közelítés után, a konvergencia tulajdonság miatt, fennáll a $d[v_i] = \delta(s,v_i)$, és ez a későbbiekben is megmarad. ■

Fokozatos közelítés és a legrövidebb utak fái

Most azt mutatjuk meg, hogy amikor egy közelítő lépéssorozat kiszámolja a tényleges legrövidebb-út súlyokat, akkor a π értékek eredményeként előállt G_π szülő részgráf egy legrövidebb-utak fa lesz a G-ben. A soron következő lemma arról szól, hogy a szülő részgráf mindig egy olyan fa, amelyiknek a gyökere a kezdőcsúcs.

24.16. lemma. *Legyen $G = (V, E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol $s \in V$ egy kezdőcsúcs, és tegyük fel, hogy a G nem tartalmaz s-ből elérhető negatív köröket. Ekkor az* Egy-forrás-kezdőérték(G, s) *eljárással történő előkészítése után, a G_π szülő részgráf egy s gyökerű fa lesz, és bármelyik G-beli élekkel történő közelítő lépéssorozat fenntartja ezt az invariánst.*

Bizonyítás. Kezdetben az egyetlen G_π-beli csúcs a kezdőcsúcs, így a lemma triviálisan teljesül. Tekintsünk egy közelítő lépéssorozat után kialakult G_π szülő részgráfot. Először belátjuk, hogy G_π körmentes. Tegyük fel indirekt módon, hogy valamely közelítő lépés egy kört hozott létre a G_π gráfban. Legyen $c = \langle v_0, v_1, \ldots, v_k \rangle$ a kör, ahol $v_k = v_0$. Ekkor $\pi[v_i] = v_{i-1}$ minden $i = 1, 2, \ldots, k$-ra, és az általánosság megsértése nélkül feltehetjük, hogy a (v_{k-1}, v_k) éllel történő közelítés hozta létre a G_π-ben a kört.

Azt állítjuk, hogy a c körön elhelyezkedő összes csúcs elérhető az s kezdőcsúcsból. Miért? A c körön levő csúcsok szülő értékei nem nil-ek, és ezért a c körön levő csúcsoknak egy véges legrövidebb-út becslése van, amit akkor kaptak, amikor a nem nil értéket. A felső-korlát tulajdonságot figyelembe véve a c körön levő összes csúcs egy véges legrövidebb-út súllyal rendelkezik, amiből az következik, hogy ezek a csúcsok elérhetők az s-ből.

Most megvizsgáljuk közvetlenül a Közelít(v_{k-1}, v_k, w) hívása előtt a c kör legrövidebb-út becsléseit, és megmutatjuk, hogy c egy negatív kör, ami ellentmond annak a feltételnek, hogy a G nem tartalmaz kezdőcsúcsból elérhető negatív köröket. Az eljárás meghívása előtt közvetlenül $\pi[v_i] = v_{i-1}$ minden $i = 1, 2, \ldots, k-1$-re. Így minden $i = 1, 2, \ldots, k-1$-re a $d[v_i]$ utolsó módosítása a $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$ értékadás volt. Ha $d[v_{i-1}]$ ezután megváltozott, az csak csökkent. Ezért a Közelít(v_{k-1}, v_k, w) hívása előtt minden $i = 1, 2, \ldots, k-1$-re

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i). \tag{24.12}$$

Amiért a $\pi[v_k]$-t megváltoztatta az eljárás, közvetlenül azelőtt fennállt, hogy

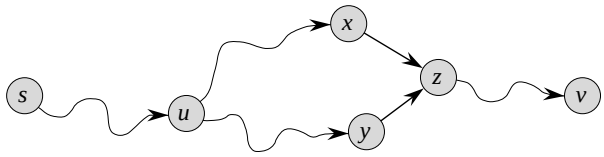
$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k).$$

Összegezve ezt a szigorú egyenlőtlenséget a (24.12)-ben lévő $k-1$ darab egyenlőtlenséggel megkapjuk a c kör csúcsaira a legrövidebb-út becslések összegét:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

De

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}],$$



24.9. ábra. Annak bizonyítása, hogy csak egyetlen G_π-beli s-ből v csúcsba vezető út van. Ha lenne két út, a $p_1(s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v)$ és a $p_2(s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v)$, ahol $x \neq y$, akkor a $\pi[z] = x$ és a $\pi[z] = y$ egyenlőségeknek egyszerre fenn kellene állniuk, ami ellentmondás.

mivel a c kör minden csúcsa pontosan egyszer szerepel az egyes összegekben. Ebből következik, hogy

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Így a c kör súlyainak összege negatív, ezáltal eljutottunk a kívánt ellentmondáshoz.

Most bebizonyítottuk, hogy a G_π irányított gráf körmentes. Ahhoz, hogy igazoljuk, hogy ez egy s gyökerű fa, elég (lásd B.5-2. gyakorlat) belátni, hogy minden $v \in V_\pi$ csúcshoz egyetlen G_π-beli út vezet s-ből.

Először meg kell mutatnunk, hogy léteznek s-ből V_π csúcsaiba vezető utak. A V_π csúcsai a nem NIL π értékű csúcsok plusz az s. A bizonyítás ötlete az, hogy indukcióval igazoljuk az s-ből V_π csúcsaiba vezető utak létezését. A részleteket a 24.5-6. gyakorlatra hagyjuk.

A lemma bizonyításának befejezéséhez meg kell most mutatnunk, hogy bármely $v \in V_\pi$ csúcshoz legfeljebb egy s-ből odavezető út van. Tegyük fel az ellenkezőjét. Tegyük fel, hogy két út vezet s-ből valamelyik v csúcsba: p_1 felbontható $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$-re, a p_2 pedig $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$-re, ahol $x \neq y$. (Lásd a 24.9. ábrát.) De ekkor $\pi[z] = x$ és $\pi[z] = y$, ami csak $x = y$ esetén lehetséges, és ez ellentmondás. Megmutattuk tehát, hogy egyetlen út vezet G_π-ben az s-ből a v-be, azaz G_π egy s gyökerű fa. ■

Most megmutatjuk, hogy ha egy olyan közelítő lépéssorozatot hajtunk végre, amelyik minden csúcshoz annak tényleges legrövidebb-út súlyát rendeli, akkor a G_π szülő részgráf egy legrövidebb-utak fája lesz.

24.17. lemma (szülő részgráf tulajdonság). *Legyen $G = (V, E)$ egy $w : E \rightarrow \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, ahol a kezdőcsúcs az $s \in V$, és tegyük fel, hogy a G nem tartalmaz s-ből elérhető negatív köröket. Legyen a gráf az* EGY-FORRÁS-KEZDŐÉRTÉK(G, s) *eljárással előkészítve, és ezután hajtsunk végre egy olyan tetszőleges G-beli élekkel történő közelítő lépéssorozatot, amely minden $v \in V$-re előállítja a $d[v] = \delta(s, v)$ értékeket. Ekkor a G_π szülő részgráf egy legrövidebb-utak fája lesz.*

Bizonyítás. Be kell bizonyítanunk, hogy a legrövidebb-utak korábban felírt három tulajdonsága teljesül a G_π-re. Az első tulajdonság igazolásához meg kell mutatnunk, hogy a V_π az s-ből elérhető csúcsokat tartalmazza. Definíció szerint egy $\delta(s, v)$ legrövidebb-út súly akkor és csak akkor véges, ha v elérhető az s-ből, és így az s-ből elérhető csúcsok pontosan azok, amelyeknek véges d értéke van. De egy $v \in V - \{s\}$ csúcs akkor és csak akkor kap véges értéket, ha $\pi[v] \neq$ NIL. Így pontosan a V_π-beli csúcsok érhetők el s-ből.

A második tulajdonság közvetlenül a 24.16. lemmából adódik.

Hátravan még a legrövidebb-utak fa utolsó tulajdonságnak a bizonyítása: minden $v \in V_\pi$-re az egyetlen $s \overset{p}{\rightsquigarrow} v$ G_π-beli út egy legrövidebb s-ből v-be vezető G-beli út. Legyen $p = \langle v_0, v_1, \ldots, v_k \rangle$, ahol $v_0 = s$ és $v_k = v$. Minden $i = 1, 2, \ldots, k$-ra fennáll a $d[v_i] = \delta(s, v_i)$ és a $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, amiből következik, hogy $w(vi - 1, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Összegezve a p út mentén a súlyokat

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) && \text{(teleszkóp szabály miatt)} \\ &= \delta(s, v_k) && \text{(mert } \delta(s, v_0 = \delta(s, s) = 0)\text{).} \end{aligned}$$

Tehát $w(p) \leq \delta(s, v_k)$. Mivel $\delta(s, v_k)$ egy alsó korlátja bármelyik s-ből v_k-ba vezető út súlyának, ebből az következik, hogy $w(p) = \delta(s, v_k)$, és ezért p egy legrövidebb s-ből $v = v_k$-ba vezető út. ■

Gyakorlatok

24.5-1. Adjunk még két másik legrövidebb-utak fát a 24.2. ábrán látható irányított gráfhoz.

24.5-2. Adjunk példát olyan $G = (V, E)$ $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráfra és az s kezdőcsúcsra, hogy az eleget tegyen a következő tulajdonságnak: Minden $(u, v) \in V$ élre van egy s gyökerű legrövidebb-utak fa, amelyik tartalmazza az (u, v) élt, és van egy másik s gyökerű legrövidebb-utak fa, amelyik viszont nem tartalmazza azt.

24.5-3. Egészítsük ki a 24.10. lemma bizonyítását azon esetek vizsgálatával, amelyekben a legrövidebb-út súlyok ∞ vagy $-\infty$ értékűek.

24.5-4. Legyen $G = (V, E)$ élsúlyozott, irányított gráf az s kezdőcsúccsal, és legyen G az EGY-FORRÁS-KEZDŐÉRTÉK(G, s) eljárással előkészítve. Bizonyítsuk be, hogy ha egy közelítő lépéssorozat nem NIL értékűre állítja be $\pi[s]$-t, akkor G tartalmaz negatív kört.

24.5-5. Legyen a $G = (V, E)$ egy negatív köröket nem tartalmazó élsúlyozott, irányított gráf. Legyen $s \in V$ a kezdőcsúcs, és definiáljuk a $\pi[v]$-t a szokásos módon: $\pi[v]$ a szülője a v csúcsnak valamelyik s-ből v-be tartó út mentén, ha $v \in V - \{s\}$ elérhető az s-ből, egyébként pedig NIL. Adjunk példát olyan G gráfra és π-re vonatkozó értékadásra, hogy a G_π-ben egy kör jelenjen meg. (A 24.16. lemma miatt a közelítő lépések egyik sorozata sem tud ilyen értékeket előállítani.)

24.5-6. Legyen $G = (V, E)$ egy $w : E \to \mathbf{R}$ súlyfüggvénnyel súlyozott, irányított gráf, amelyikben nincsenek negatív körök. Legyen $s \in V$ a kezdőcsúcs, és legyen a G az EGY-FORRÁS-KEZDŐÉRTÉK(G, s) eljárással előkészítve. Bizonyítsuk be, hogy minden $v \in V_\pi$ csúcsra létezik egy G_π-beli s-ből v-be vezető út, és ez a tulajdonság bármelyik közelítés sorozat esetén invariánsként fennmarad.

24.5-7. Legyen $G = (V, E)$ egy élsúlyozott, irányított gráf, amelyikben nincsenek negatív körök. Legyen $s \in V$ a kezdőcsúcs, és legyen a G az EGY-FORRÁS-KEZDŐÉRTÉK(G, s) eljárással előkészítve. Bizonyítsuk be, hogy van a közelítő lépéseknek olyan $|V| - 1$ hosszú sorozata, amelyik minden $v \in V$-re beállítja a $d[v] = \delta(s, v)$ értékeket.

24.5-8. Legyen $G = (V, E)$ egy élsúlyozott, irányított gráf az s-ből elérhető negatív körökkel. Mutassunk a G gráf éleivel történő közelítéseknek egy olyan végtelen lépéssorozatát, amelyik minden közelítése egy legrövidebb-út becslés megváltozását okozza.

Feladatok

24-1. Yen javítása a Bellman–Ford-algoritmushoz

Tegyük fel, hogy a következőképpen rendezzük a Bellman–Ford-algoritmus minden menetében a közelítést végző éleket. Az első menet előtt egy tetszőleges $v_1, v_2, \ldots, v_{|V|}$ lineáris rendezését vesszük a $G = (V, E)$ bemenő gráf csúcsainak. Aztán két részre $E_f \cup E_b$-re osztjuk az E halmaz éleit, ahol $E_f = \{(v_i, v_j) \in E : i\langle j\}$ és $E_b = \{(v_i, v_j) \in E : i\rangle j\}$. Definiáljuk a $G_f = (V, E_f)$ és a $G_b = (V, E_b)$ gráfokat.

a. Bizonyítsuk be, hogy a G_f a $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ topologikus rendezést felhasználva lesz körmentes, míg a G_b a $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1\rangle$ topologikus rendezés mellett lesz körmentes.

Tegyük fel, hogy a következő módon valósítjuk meg a Bellman–Ford-algoritmus egyes meneteit. A $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ sorrendben vizsgáljuk meg a csúcsokat, és a belőlük kivezető E_f-beli élekkel végzünk közelítéseket. Aztán a $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1\rangle$ sorrendnek megfelelően vesszük a csúcsokat, és a belőlük kivezető E_b-beli élekkel végzünk közelítéseket.

b. Bizonyítsuk be, hogy ezzel a sémával, ha a G nem tartalmaz az s kezdőcsúcsból elérhető negatív köröket, akkor már $\lceil |V|/2 \rceil$ menet után teljesül, hogy minden $v \in V$ csúcsra $d[v] = \delta(s, v)$.

c. Hogyan hat ez a séma a Bellman–Ford-algoritmus futási idejére?

24-2. Egymásba illeszthető dobozok

Egy $(x_1, x_2, \ldots, x_d)$ méretű d dimenziós doboz ***beleilleszthető*** egy másik $(y_1, y_2, \ldots, y_d)$ méretű d dimenziós dobozba, ha létezik az $(1, 2, \ldots, d)$-nek egy olyan π permutációja, hogy $x_{\pi(1)}\langle y_1, x_{\pi(2)}\langle y_2, \ldots, x_{\pi(d)}\langle y_d$.

a. Igaz-e, hogy a beleilleszthetőségi reláció tranzitív?

b. Írjuk le azt a hatékony módszert, amely meghatározza, hogy egy d dimenziós doboz beleilleszthető-e vagy sem egy másikba.

c. Tegyük fel, hogy van n darab $\{B_1, B_2, \ldots, B_n\}$ d dimenziós dobozunk. Adjunk olyan hatékony algoritmust, amely meghatározza azt a leghosszabb $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k}\rangle$ dobozokból álló sorozatot, amelyben minden $j = 1, 2, \ldots, k-1$-re a B_{i_j} beleilleszthető a $B_{i_{j+1}}$-be. Fejezzük ki ennek az algoritmusnak a futási idejét az n és a d függvényében.

24-3. Arbitrázs

Az ***arbitrázs*** a valutaátváltások során az árfolyamok egyenetlenségeinek olyan hasznosítását jelenti, amikor egy valuta egy egységét ugyanazon valuta egy egységnél nagyobb értékére váltjuk át. Például tegyük fel, hogy 1 dollárért 46, 4 indiai rúpiát, 1 indiai rúpiáért 2, 5 japán jent, és 1 japán jenért 0, 0091 dollárt vehetünk. Ekkor az a pénzváltó, aki 1 dollárt fektet be, és a valuta konverzió során $46, 4 \times 2, 5 \times 0, 0091 = 1, 0556$ dollárt vásárol, 5, 56 százalék haszonra tesz szert.

Tegyük fel, hogy adott n különböző $(c_1, c_2, \ldots, c_n)$ valutanem, és a valutaárfolyamoknak egy $n \times n$-es R táblázata, amely azt mutatja, hogy a c_i valuta egy egységéért $R(i, j)$ egységnyi c_j valuta vásárolható.

a. Adjunk hatékony algoritmust, amely meghatározza, hogy létezik-e a valutanemeknek egy olyan $(c_{i_1}, c_{i_2}, \ldots, c_{i_k})$ sorozata, amelyre

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Elemezzük az algoritmus futási idejét.

b. Adjunk hatékony algoritmust egy ilyen sorozat kiírására feltéve, hogy létezik. Elemezzük az algoritmus a futási idejét.

24-4. Gabow skálázó algoritmusa adott csúcsból induló legrövidebb utak problémájára

Egy ***skálázó*** algoritmus úgy old meg egy problémát, hogy kezdetben a lényeges bemenő adatok értékének (mint például egy élsúly) csak a legmagasabb helyiértékű bitjét veszi figyelembe. Azután a kezdeti fázisban kapott megoldást a második helyiértékű bitek alapján finomítja. Ezt tovább folytatva egyre több és több helyiértéket vesz figyelembe, addig finomítja a megoldást, amíg az összes bitet felhasználva a helyes megoldáshoz jut.

Ebben a feladatban egy olyan algoritmust vizsgálunk meg, amelyik az élsúlyok skálázásával keres adott kezdőcsúcsból induló legrövidebb utakat. Adott egy $G = (V, E)$ gráf, amely éleit egy nemnegatív egész értékű w függvénnyel súlyoztunk. Egy olyan algoritmus kifejlesztése a célunk, melynek futási ideje $O(E \lg W)$, ahol $W = max_{(u,v)\in E}\{w(u, v)\}$. Feltesszük, hogy minden csúcs elérhető a kezdőcsúcsból.

Az algoritmus az élek súlyának bináris reprezentációjában szereplő biteket egyesével tárja fel a legfontosabbtól a legkevésbé fontosig. Részletezve, legyen $k = \lceil \lg(W + 1) \rceil$ a W bináris reprezentációjában a bitek száma, és minden $i = 1, 2, \ldots, k$-ra legyen $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. Másként fogalmazva, a $w_i(u, v)$ a $w(u, v)$-nek a legjellemzőbb i bit szerinti „leskálázott” változata. (Így, minden $(u, v) \in E$-re $w_k(u, v) = w(u, v)$.) Például ha $k = 5$ és $w(u, v) = 25$, amelynek bináris reprezentációja a $\langle 11001 \rangle$, akkor $w_3(u, v) = \langle 110 \rangle = 6$. Egy másik példában ha $k = 5$ és $w(u, v) = \langle 00100 \rangle = 4$, akkor $w_3(u, v) = \langle 001 \rangle = 1$. Definiáljuk a $\delta_i(u, v)$-t úgy, mint az u csúcsból a v csúcsba vezető w_i súlyfüggvény mellett számolt legrövidebb-út költségét. Ekkor minden $u, v \in V$-re $\delta_k(u, v) = \delta(u, v)$. Egy adott s kezdőcsúcsból indulva a skálázó algoritmus először minden $v \in E$-re a $\delta_1(s, v)$ legrövidebb-út súlyokat állítja elő, aztán minden $v \in V$-re a $\delta_2(s, v)$ értékeket, és ezt addig folytatja, amíg minden $v \in V$-re a $\delta_k(s, v)$ értékeket is kiszámolja. Mindvégig feltesszük, hogy $|E| \geq |V| - 1$, és látni fogjuk, hogy a δ_i-nek a δ_{i-1} alapján történő kiszámítása $O(kE) = O(E \lg W)$ ideig tart.

a. Tegyük fel, hogy minden $v \in V$ csúcsra a $\delta(s, v) \leq |E|$ fennáll. Mutassuk meg, hogy minden $v \in V$-re a $\delta(s, v)$ kiszámolása $O(E)$ ideig tart.

b. Mutassuk meg, hogy minden $v \in V$-re a $\delta(s, v)$ kiszámítása $O(E)$ ideig tart.

Koncentráljunk most arra, hogyan számolható ki a δ_i a δ_{i-1}-ből.

c. Bizonyítsuk be, hogy minden $i = 2, 3, \ldots, k$-ra vagy $w_i(u, v) = 2w_{i-1}(u, v)$, vagy $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Bizonyítsuk be azután azt is, hogy

$$2\delta_{i-1}(s,v) \le \delta_i(s,v) \le 2\delta_{i-1}(s,v) + |V| - 1$$

minden $v \in V$-re.

d. Definiáljuk minden $i = 2, 3, \ldots, k$-ra és minden $(u,v) \in E$-re a

$$\widehat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v)$$

súlyt. Bizonyítsuk be, hogy minden $i = 2, 3, \ldots, k$-ra és minden $(u,v) \in E$-re az (u,v) él „átsúlyozott" $\widehat{w}_i(u,v)$ értéke nemnegatív egész.

e. Definiáljuk most a $\widehat{\delta}_i(s,v)$ értéket úgy, mint az s-ből v-be vezető $\widehat{w}_i$ súlyfüggvény alapján számolt legrövidebb út súlyát. Bizonyítsuk be, hogy minden $v \in V$-re

$$\delta(s,v) = \widehat{\delta}_i(s,v) + 2\delta_{i-1}(s,v),$$

és hogy $\widehat{\delta}_i(s,v) \le |E|$.

f. Mutassuk meg, hogyan számolható ki minden $v \in V$-re $O(E)$ időben a $\delta_i(s,v)$ a $\delta_{i-1}(s,v)$ értékéből, és igazoljuk, hogy minden $v \in V$-re a $\delta(s,v)$ kiszámítása $O(E \lg W)$ ideig tart.

24-5. Karp minimális átlagsúlyú kör algoritmusa

Legyen $G = (V,E)$ egy irányított gráf a $w : E \to \mathbf{R}$ súlyfüggvénnyel, és legyen $n = |V|$. Legyen definíció szerint az E-beli élek egy $c = \langle e_1, e_2, \ldots, e_k \rangle$ körének ***átlagsúlya***

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i).$$

Legyen $\mu^* = \min_c \mu(c)$, ahol c befutja az összes G-beli irányított kört. Egy c kört, amelyre teljesül a $\mu(c) = \mu^*$, ***minimális átlagsúlyú körnek*** nevezzük. Ez a feladat a μ^* kiszámításához keres hatékony algoritmust.

Tegyük fel az általánosság megsértése nélkül, hogy az $s \in V$ kezdőcsúcsból elérhető az összes $v \in V$ csúcs. Legyen $\delta(s,v)$ egy legrövidebb s-ből v-be vezető út súlya, és legyen a $\delta_k(s,v)$ a pontosan k élt tartalmazó s-ből v-be vezető legrövidebb út súlya. Ha nem létezik pontosan k élű s-ből v-be vezető út, akkor $\delta_k(s,v) = \infty$.

a. Mutassuk meg, hogy ha $\mu^* = 0$, akkor a G nem tartalmaz negatív súlyú köröket, és minden $v \in V$ csúcsra a $\delta(s,v) = \min_{0 \le k \le n-1} \delta_k(s,v)$.

b. Mutassuk meg, hogy ha $\mu^* = 0$, akkor minden $v \in V$ csúcsra a

$$\max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \ge 0.$$

(*Útmutatás.* Használjuk ki az (a.) rész mindkét tulajdonságát.)

c. Legyen c egy 0 súlyú kör, és legyen u és v két csúcs a c körön. Tegyük fel, hogy az u-ból v-be a c kör mentén vezető út súlya x. Bizonyítsuk be, hogy $\delta(s,v) = \delta(s,u) + x$. (*Útmutatás.* A v-ből u-ba a c kör mentén vezető út súlya $-x$.)

d. Mutassuk meg, hogy ha $\mu^* = 0$, akkor létezik egy v csúcs a minimális átlagsúlyú körön úgy, hogy a

$$\max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

(*Útmutatás.* Mutassuk meg, hogy a minimális átlagsúlyú körön egy tetszőleges csúcshoz vezető legrövidebb út kiegészíthető a kör mentén úgy, hogy egy legrövidebb utat kapjunk a kör következő csúcsához.)

e. Mutassuk meg, hogy ha $\mu^* = 0$, akkor

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

f. Mutassuk meg, hogy ha a G mindegyik élének súlyához hozzáadunk egy t konstanst, akkor a μ^* értéke t-vel nő. Ezt felhasználva mutassuk meg, hogy

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}.$$

g. Adjunk $O(V\,E)$ idejű algoritmust a μ^* kiszámolására.

24-6. Kéttónusú legrövidebb utak

Egy sorozat ***kéttónusú,*** ha ciklikusan összekapcsolva egy monoton növekedő és egy monoton csökkenő szakaszból áll. Például az $\langle 1, 4, 6, 8, 3, -2\rangle$, a $\langle 9, 2, -4, -10, -5\rangle$ és az $\langle 1, 2, 3, 4\rangle$ kéttónusú, de az $\langle 1, 3, 12, 4, 2, 10\rangle$ nem. (Lásd a 27. fejezetbeli kéttónusú rendezést és a 15-1. feladatban szereplő kéttónusú euklideszi utazóügynök problémát.)

Tegyük fel, hogy adva van egy $G = (V, E)$ irányított gráf a $w : E \to \mathbf{R}$ súlyfüggvénnyel, és meg szeretnénk találni az s kezdőcsúcsból kiinduló legrövidebb utakat. Rendelkezünk többletinformációval: minden $v \in V$ csúcsra az s-ből v-be vezető bármelyik legrövidebb út súlyai kéttónusú sorozatot alkotnak, és az összes súly egyedi.

Adjuk meg a lehető leghatékonyabb algoritmust erre a feladatra, és elemezzük a futási idejét.

Megjegyzések a fejezethez

A Dijkstra-algoritmus [75] 1959-ben jelent meg, de nem tartalmazta az elsőbbségi sor ötletét. A Bellman–Ford-algoritmus Bellman [35] és Ford [93] egymástól független algoritmusaira épül. Bellman adta meg a legrövidebb utak és a különbségi korlátok kapcsolatát. Lawler [196] írta le a lineáris idejű körmentes irányított gráfokon működő legrövidebb-utak algoritmust, amelyet ő a szóhagyomány részének tart.

Amikor az élsúlyok hozzávetőlegesen kicsi egészek, hatékonyabb algoritmusokat használhatunk az adott kezdőcsúcsból induló legrövidebb utak probléma megoldásához. A Dijkstra-algoritmus által meghívott KIVESZ-MIN eljárás az értékeket monoton növekvő sorrendben adja vissza. Mint azt a 6. fejezet megjegyzései között megtárgyaltuk, ebben az esetben több olyan adatszerkezet is van, amely segítségével az elsőbbségi sor műveleteit hatékonyabban implementálhatjuk, mint ha bináris vagy Fibonacci-kupacot használnánk. Ahuja, Mehlhorn, Orlin és Tarjan [8] egy olyan algoritmust mutattak, amelynek futási ideje $O(E + V\sqrt{\lg W})$ olyan nemnegatív élsúlyú gráfokon, ahol a gráfban előforduló legnagyobb élsúly a W. A legjobb korlátok egyrészt Thoruptól [299] származnak, aki egy $O(E \lg\lg V)$ idejű algoritmust adott, másrészt Ramantól [256], akinek algoritmusa $O(E + V \min\{(\lg V)^{1/3+\epsilon}, (\lg V)^{1/4+\epsilon}\})$. Ennek a két algoritmusnak a memóriaigénye a számítógép szóméretétől függ. Habár a felhasznált memória mérete korlátlan nagy lehet, csökkenthető úgy, hogy az a bemenő adatok méretétől lineárisan függjön.

Egész számokkal súlyozott irányítatlan gráfokra Thorup [298] adott egy $O(V+E)$ idejű, adott kezdőcsúcsból induló legrövidebb utakat kereső algoritmust. Ellentétben az előző bekezdésben vázolt algoritmussal, ez nem a Dijkstra-algoritmus megvalósítása, mivel az általa meghívott KIVESZ-MIN eljárás az értékeket nem monoton növekvő sorrendben adja vissza.

Negatív élsúlyú gráfokra egy Gabownak és Tarjannak [104] tulajdonított algoritmus $O(\sqrt{V}E \lg(VW))$, egy Goldbergtől [118] származó pedig $O(\sqrt{V}E \lg W)$ idejű, ahol a $W = \max_{(u,v)\in E}\{|w(u,v)|\}$.

Cherkassky, Goldberg és Radzik [57] széles körű kísérleteket folytatott a különféle legrövidebb utak algoritmusainak összehasonlítására.

25. Legrövidebb utak minden csúcspárra

Ebben a fejezetben célunk egy gráf valamennyi rendezett csúcspárjára a két csúcs közti legrövidebb út megkeresése. Ha például egy autóstérképhez elkészítjük a városok egymástól mért távolságainak táblázatát, éppen ezt a feladatot oldjuk meg. Csakúgy, mint a 24. fejezetben, $G = (V, E)$ egy súlyozott irányított gráfot jelöl, amelynek élhalmazán egy $w : E \to \mathbf{R}$ valós értékű súlyfüggvény van megadva. A gráf minden $u, v \in V$ csúcspárjára keressük az u-ból v-be vezető legrövidebb (legkisebb súlyú) utat, ahol egy út súlya az úthoz tartozó élek súlyának összege. Az eredményt általában táblázatos formában keressük; a táblázat u-hoz tartozó sorában és v-hez tartozó oszlopában álló elem az u-ból v-be vezető legrövidebb út hossza.

A csúcspárok közti legrövidebb utakat természetesen megkereshetjük úgy, hogy az előző fejezetben látott valamelyik egy kezdőcsúcsból kiinduló legrövidebb utakat keresőalgoritmust egyenként végrehajtunk a lehetséges $|V|$ különböző gyökércsúcsot választva. Ha a távolságok nemnegatívok, Dijkstra algoritmusát alkalmazhatjuk; ha elsőbbségi sorként lineáris tömböt használunk, a teljes futási idő $O(V^3 + VE) = O(V^3)$ lesz. Bináris min-kupacot használva e futási idő $O(VE \lg V)$, ami az előzőnél jobb ritka gráfok esetén. Alkalmazhatjuk továbbá a Fibonacci-kupac adatszerkezetet is; ekkor a futási idő $O(V^2 \lg V + VE)$.

Ha negatív élsúlyokat is megengedünk a gráfban, Dijkstra algoritmusa nem alkalmazható, helyette a lassabb Bellman–Ford-algoritmust kell minden csúcsra egyszer végrehajtanunk. Így a futási idő $O(V^2E)$ lesz, amely sűrű gráfokon csak $O(V^4)$-t ad. Célunk tehát, hogy a negatív élsúlyok esetére hatékonyabb algoritmusokat adjunk. Eközben megismerjük az összes csúcspár közti legrövidebb út probléma és a mátrixszorzás kapcsolatát is.

Az egy csúcsból kiinduló legrövidebb utakat megadó algoritmusok általában a gráf éllistás megadását igénylik. Ezzel szemben a fejezet legtöbb algoritmusa a szomszédsági mátrixos megadást használja. (Johnson algoritmusa az egyetlen kivétel, mivel ritka gráfok esetén éllistákat használ.) A bemenő adat tehát egy W $n \times n$-es mátrix lesz. A mátrix egy n csúcsú irányított $G = (V, E)$ gráf élsúlyait tartalmazza: $W = (w_{ij})$, ahol

$$w_{ij} = \begin{cases} 0, & \text{ha } i = j, \\ \text{az irányított } (i, j) \text{ él hossza}, & \text{ha } i \neq j \text{ és } (i, j) \in E, \\ \infty, & \text{ha } i \neq j \text{ és } (i, j) \notin E. \end{cases} \tag{25.1}$$

A gráfban megengedünk negatív súlyú éleket, azonban egyelőre feltesszük, hogy negatív összsúlyú körök nincsenek.

A fejezetbeli algoritmusok kimenete az összes párra adott legrövidebb úthosszakat tartalmazó táblázat, egy D $n \times n$-es mátrix lesz. A mátrix d_{ij} eleme az i csúcsból a j csúcsba vezető legrövidebb út súlyát tartalmazza. A végeredményként kapott d_{ij} értékek tehát megegyeznek a 24. fejezetbeli $\delta(i, j)$-vel, az i csúcsból a j csúcsba vezető legrövidebb út súlyával.

Csak akkor mondhatjuk, hogy a kitűzött feladatot megoldottuk, ha a legrövidebb utak súlya mellett magukat az utakat is meg tudjuk adni. E célból egy $\Pi = (\pi_{ij})$ ***megelőzési mátrixot*** is meg kell adnunk, amelyben $\pi_{ij} = \text{NIL}$, ha $i = j$ vagy ha nem vezet i és j között út; ellenkező esetben π_{ij} a j-t megelőző csúcs az egyik i-ből j-be vezető legrövidebb úton. Ahogyan a 24. fejezet G_π megelőzési részgráfja a legrövidebb utak fája egy adott gyökérpont mellett, a Π mátrix i-edik sora által indukált részgráf egy i gyökerű legrövidebb utak fája lesz. Minden $i \in V$ csúcsra a G gráf következő $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$ ***megelőzési részgráfját*** definiálhatjuk:

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

és

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ és } - \{i\}\}.$$

Ha $G_{\pi,i}$ legrövidebb utak fája, akkor a 22. fejezetbeli UTAT-NYOMTAT eljárás következő módosított változata megadja a fa által definiált, i csúcstól a j-ig vezető legrövidebb utat:

```
MINDEN-PÁRHOZ-UTAT-NYOMTAT(Π, i, j)
1  if i = j
2    then print i
3    else if π_ij = NIL
4           then print i„-ből” j„-be nem vezet út”
5           else MINDEN-PÁRHOZ-UTAT-NYOMTAT(Π, i, π_ij)
6                print j
```

Mivel ebben a fejezetben az összes csúcspárt kezelő algoritmusok alaptulajdonságait kívánjuk hangsúlyozni, nem fogjuk a megelőzési mátrixok létrehozását és tulajdonságait olyan részletességgel ismertetni, mint azt a 24. fejezet megelőzési részgráfjaival tettük. Néhány alaptulajdonság igazolása a gyakorlatok között fog szerepelni.

A fejezet tartalma

A 25.1. alfejezetben mátrixszorzáson alapuló dinamikus programozással oldjuk meg az összes csúcspárra a legrövidebb utak keresését. Az „ismételt négyzetre emelések” módszerével elérhető, hogy az algoritmus futási ideje $\Theta(V^3 \lg V)$ legyen. A 25.2. alfejezetben egy másik dinamikus programozási módszerrel, a $\Theta(V^3)$ futási idejű Floyd–Warshall-algoritmussal ismerkedünk meg. Ugyanebben az alfejezetben az összes párra vonatkozó legrövidebb út problémához kapcsolódóan az irányított gráfok tranzitív lezártjának megkeresésével is foglalkozunk. Végezetül a 25.3. alfejezetben található Johnson algoritmusa. A fejezet többi algoritmusától eltérően ez az algoritmus a gráf éllistás megadását használja; az összes párra vonatkozó legrövidebb út problémát $O(V^2 \lg V + VE)$ időben oldja meg, így nagyméretű ritka gráfok esetén hatékony.

Mielőtt az algoritmusokat ismertetnénk, állapodjunk meg néhány, szomszédsági mátrixokkal kapcsolatos, jelölésben. Először is a $G = (V, E)$ gráfnak általában n csúcsa lesz, azaz $n = |V|$. Másodszor a mátrixokat nagybetűvel, egyes elemeiket pedig alulindexelt kisbetűvel fogjuk jelölni, tehát például W, D, illetve L elemei w_{ij}, d_{ij}, illetve ℓ_{ij} lesznek. Bizonyos esetekben iterációk jelölésére a mátrixokat zárójelezett felsőindexszel látjuk el, úgy mint $D^{(m)} = (d_{ij}^{(m)})$ vagy $L^{(m)} = (\ell_{ij}^{(m)})$. Végül egy adott A $n \times n$-es mátrix esetén *sorok-száma*$[A]$ tartalmazza n értékét.

25.1. Egy mátrixszorzás típusú módszer legrövidebb utak keresésére

Ebben az alfejezetben dinamikus programozással fogjuk egy irányított $G = (V, E)$ gráfon az összes csúcspárra vonatkozó legrövidebb út problémát megoldani. A dinamikus programozás minden fő lépése a mátrixszorzáshoz nagyon hasonló műveletet végez; az algoritmus maga mátrixok ismételt szorzására fog hasonlítani. Először egy $\Theta(V^4)$ idejű algoritmust fogunk adni, melynek futási idejét azután $\Theta(V^3 \lg V)$-re javítjuk.

Emlékeztetőül a dinamikus programozás 15. fejezetben látott fő lépései az optimális megoldások

1. szerkezetének vizsgálata;
2. értékeinek rekurzív definíciója; és
3. értékeinek kiszámítása alulról felfelé haladva.

(A 4. lépés, az optimális megoldás megadása a kiszámított adatok alapján, a gyakorlatok között fog szerepelni.)

A legrövidebb utak szerkezete

Először is megadjuk az optimális megoldás szerkezetét. Egy $G = (V, E)$ irányított gráf összes csúcspárjára vonatkozó legrövidebb utak esetére bebizonyítottuk, hogy a legrövidebb utak minden részútja is legrövidebb út (24.1. lemma). Tételezzük fel, hogy a gráfot a $W = (w_{ij})$ szomszédsági mátrix adja meg. Legyen p egy i-ből j-be vezető legrövidebb út, álljon p m darab élből. Feltéve, hogy nincsenek negatív összsúlyú körök, m véges. Ha $i = j$, akkor p súlya 0 és nem tartalmaz élt. Ha $i \neq j$, akkor felbonthatjuk p-t $i \overset{p'}{\rightsquigarrow} k \rightarrow j$ alakban, ahol p' legfeljebb $m - 1$ élt tartalmaz, továbbá a 24.1. lemma szerint p' egy i és k közti legrövidebb út, így $\delta(i, j) = \delta(i, k) + w_{kj}$.

Az összes csúcspár közti legrövidebb utak rekurzív megadása

Legyen $\ell_{ij}^{(m)}$ a legrövidebb olyan út súlya, mely i-ből j-be vezet és legfeljebb m élből áll. Ha $m = 0$, akkor és csak akkor van i-ből j-be vezető élt nem tartalmazó legrövidebb út, ha $i = j$:

$$\ell_{ij}^{(0)} = \begin{cases} 0, & \text{ha } i = j, \\ \infty, & \text{ha } i \neq j. \end{cases}$$

Ha $m \geq 1$, akkor $\ell_{ij}^{(m)}$ a minimuma $\ell_{ij}^{(m-1)}$-nek (a legrövidebb legfeljebb $m-1$ élből álló i-ből j-be vezető út súlyának) és minden k csúcsra az olyan legrövidebb i-ből j-be vezető m élű út súlyának, melyen j-t közvetlenül a k csúcs előzi meg. Így rekurzívan definiálható

$$\begin{aligned} \ell_{ij}^{(m)} &= \min(\ell_{ij}^{(m-1)}, \min_{1\leq k\leq n}\{\ell_{ik}^{(m-1)} + w_{kj}\}) \\ &= \min_{1\leq k\leq n}\{\ell_{ik}^{(m-1)} + w_{kj}\} . \end{aligned} \tag{25.2}$$

A második egyenlőség abból következik, hogy $w_{jj} = 0$ minden j esetén.

Mi lesz tehát a legrövidebb utak $\delta(i, j)$ súlya? Ha nincsen negatív súlyú kör a gráfban, akkor minden legrövidebb út egyszerű és így legfeljebb $n-1$ élből áll. Egy i-ből j-be vezető több mint $n-1$ élű út súlya pedig legfeljebb akkora, mint a legrövidebb ilyen úté. A legrövidebb úthossz tehát úgy adható meg, mint

$$\delta(i, j) = \ell_{ij}^{(n-1)} = \ell_{ij}^{(n)} = \ell_{ij}^{(n+1)} = \cdots . \tag{25.3}$$

Az úthosszak kiszámítása alulról felfelé haladva

A most következő lépések során a bemenő $W = (w_{ij})$ mátrixból mátrixok egy $L^{(1)}$, $L^{(2)}$, …, $L^{(n-1)}$ sorozatát számítjuk ki, ahol az $m = 1, 2, \ldots, n-1$ értékek esetén $L^{(m)} = (\ell_{ij}^{(m)})$. Az utolsó $L^{(n-1)}$ mátrix a legrövidebb utak hosszát tartalmazza. A kezdőlépésben $L^{(1)} = W$, hiszen minden $i, j \in V$ csúcspárra $d_{ij}^{(1)} = w_{ij}$.

Az algoritmus magja $L^{(m-1)}$ és W ismeretében $L^{(m)}$ értékét számítja, azaz az eddig számított legrövidebb utakat egy éllel bővíti.

LEGRÖVIDEBB-ÚT-BŐVÍTÉS(L, W)

```
1  n ← sorok-száma[L]
2  legyen L' = (ℓ'_ij) egy n × n-es mátrix
3  for i ← 1 to n
4      do for j ← 1 to n
5             do ℓ'_ij ← ∞
6                for k ← 1 to n
7                    do ℓ'_ij ← min(ℓ'_ij, ℓ_ik + w_kj)
8  return L'
```

Az eljárás végeredménye a számított $L' = (\ell'_{ij})$ mátrix. A számítás lépései a (25.2) egyenletet alkalmazzák minden i és j értékre, $L^{(m-1)}$-t L-vel és $L^{(m)}$-t L'-vel helyettesítve. (A felsőindexeket azért célszerű elhagyni a fenti eljárásban, hogy a be- és kimenő mátrixok m értékétől ne függjenek.) Az eljárás futási ideje a háromszorosan egymásba ágyazott ciklusok miatt $\Theta(n^3)$ lesz.

A fenti lépések és a mátrixszorzás hasonlósága most már jól látható: ha az A és B $n \times n$ mátrixok $C = A \cdot B$ szorzatát számítjuk, akkor $i, j = 1, 2, \ldots, n$ esetén a

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{25.4}$$

műveleteket végezzük el. Ha a (25.2) egyenlőségben a

$$\begin{aligned} d^{(m-1)} &\to a, \\ w &\to b, \\ d^{(m)} &\to c, \\ \min &\to +, \\ + &\to \cdot \end{aligned}$$

helyettesítést alkalmazzuk, a fenti (25.4) egyenlőséghez jutunk. Tehát ha a LEGRÖVIDEBB-ÚT-BŐVÍTÉS eljárásban is alkalmazzuk a helyettesítéseket és a ∞ értéket (a min művelet egységelemét) 0-ra (a + egységelemére) cseréljük, a közismert $\Theta(n^3)$ lépésű mátrixszorzó eljárást kapjuk.

MÁTRIXSZORZÁS(A, B)

```
1  n ← sorok-száma[A]
2  legyen C = (c_ij) egy n × n mátrix
3  for i ← 1 to n
4      do for j ← 1 to n
5             do c_ij ← 0
6                for k ← 1 to n
7                    do c_ij ← c_ij + a_ik + b_kj
8  return C
```

Most térjünk vissza a legrövidebb utak kérdésére. A legrövidebb utak súlyát úgy keressük, hogy élek egyenkénti hozzáadásával építünk fel utakat. Jelölje $A \cdot B$ a LEGRÖVIDEBB-ÚT-BŐVÍTÉS(A, B) által kapott „mátrixszorzatot". Így a következő $n-1$ mátrixból álló sorozatot számítjuk ki:

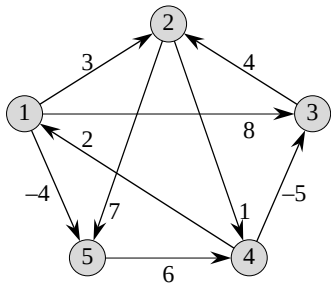
$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W &&= W, \\ L^{(2)} &= L^{(1)} \cdot W &&= W^2, \\ L^{(3)} &= L^{(2)} \cdot W &&= W^3, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}. \end{aligned}$$

Mint korábban bizonyítottuk, a $L^{(n-1)} = W^{n-1}$ mátrix a legrövidebb utak súlyát tartalmazza. A következő algoritmus a fenti mátrixsorozatot $\Theta(n^4)$ lépésben számítja:

LASSÚ-MINDEN-LEGRÖVIDEBB-ÚT(W)

```
1  n ← sorok-száma[W]
2  L^(1) ← W
3  for m ← 2 to n − 1
4      do L^(m) ← LEGRÖVIDEBB-ÚT-BŐVÍTÉS(L^(m−1), W)
5  return L^(n−1)
```

A 25.1. ábrán egy gráfon a LASSÚ-MINDEN-LEGRÖVIDEBB-ÚT eljárás által számított $L^{(m)}$ mátrixok láthatók.



25.1. ábra. Egy irányított gráf és a LASSÚ-MINDEN-LEGRÖVIDEBB-ÚT eljárás által számított $L^{(m)}$ mátrixok sorozata $m = 1, 2, 3, 4$ esetén (az Olvasóra bízzuk annak ellenőrzését, hogy $L^{(5)} = L^{(4)} \cdot W$ megegyezik $L^{(4)}$-gyel és így $L^{(m)} = L^{(4)}$ minden $m \geq 4$ esetén):

$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

A futási idő csökkentése

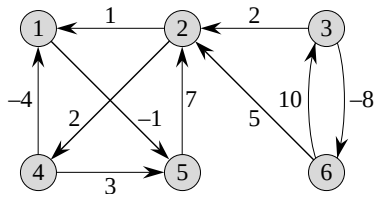
Míg a LASSÚ-MINDEN-LEGRÖVIDEBB-ÚT algoritmus a $L^{(m)}$ mátrixok *teljes* sorozatát előállítja, a mi célunk csak a sorozat utolsó $L^{(n-1)}$ elemének kiszámítása. Amennyiben gráf nem tartalmaz negatív összsúlyú köröket, a (25.3) egyenlőség szerint $L^{(m)} = L^{(n-1)}$ minden $m \geq n - 1$ érték esetén. A hagyományos mátrixszorzáshoz hasonlóan a LEGRÖVIDEBB-ÚT-BŐVÍTÉS eljárásbeli szorzás is asszociatív (lásd 25.1-4. gyakorlat). A keresett $L^{(n-1)}$ mátrix ezért $\lceil \lg(n-1) \rceil$ darab mátrixszorzással a következő sorozatban számítható:

$$\begin{aligned} L^{(1)} &= W, \\ L^{(2)} &= W^2 &&= W \cdot W, \\ L^{(4)} &= W^4 &&= W^2 \cdot W^2, \\ L^{(8)} &= W^8 &&= W^4 \cdot W^4, \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil}-1} \cdot W^{2^{\lceil \lg(n-1) \rceil}-1}. \end{aligned}$$

Mivel $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, a végső $L^{(2^{\lceil \lg(n-1) \rceil})}$ szorzat a keresett $L^{(n-1)}$-gyel megegyezik.

A fenti mátrixsorozatot az ***ismételt négyzetreemelések*** módszerével számíthatjuk ki.

Az alábbi kódban a 4–6. sorokban levő **while** ciklus $m = 1$-től kezdve minden végrehajtása során az $L^{(2m)} = (L^{(m)})^2$ mátrixokat számítja; m értéke ezek után megkétszereződik. Az utolsó iterációban az $L^{(n-1)}$ mátrixot kapjuk úgy, hogy valamely $n - 1 \leq 2m \leq 2n - 2$ értékre $L^{(2m)}$-t számítjuk. A (25.3) egyenlőség szerint $L^{(2m)} = L^{(n-1)}$. Amikor a 4. sorban a kilépési feltételt ellenőrizzük, m értéke megkétszereződött, így $m \geq n - 1$, a ciklus végetér, és az algoritmus végeredményként az utolsóként számított mátrixot szolgáltatja.



25.2. ábra. A 25.1-1., 25.2-1. és 25.3-1. gyakorlatokban szereplő súlyozott irányított gráf.

GYORSABB-MINDEN-LEGRÖVIDEBB-ÚT(W)

1 $n \leftarrow$ *sorok-száma*$[W]$
2 $L^{(1)} \leftarrow W$
3 $m \leftarrow 1$
4 **while** $n - 1 > m$
5 **do** $L^{(2m)} \leftarrow$ LEGRÖVIDEBB-ÚT-BŐVÍTÉS$(L^{(m)}, L^{(m)})$
6 $m \leftarrow 2m$
7 **return** $L^{(m)}$

GYORSABB-MINDEN-LEGRÖVIDEBB-ÚT futási ideje $\Theta(n^3 \lg n)$, hiszen az $\lceil \lg(n-1) \rceil$ darab mátrixszorzat mindegyikét $\Theta(n^3)$ időben kapjuk. A fenti programkód tömör és kizárólag egyszerű adatszerkezeteket használ, így a Θ-jelölésben rejlő állandó értéke kicsi.

Gyakorlatok

25.1-1. Adjuk meg az egyes iterációk során keletkező mátrixokat, ha a LASSÚ-MINDEN-LEGRÖVIDEBB-ÚT algoritmust a 25.2. ábrán szereplő gráfon futtatjuk. Adjuk meg ugyanezeket a mátrixokat a GYORSABB-MINDEN-LEGRÖVIDEBB-ÚT algoritmus esetén is.

25.1-2. Miért szükséges feltenni, hogy $w_{ii} = 0$ minden $1 \leq i \leq n$ értékre?

25.1-3. A közönséges mátrixszorzás esetében melyik mátrixnak felel meg a legrövidebb utak keresésekor használt

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \dots & \infty \\ \infty & 0 & \infty & \dots & \infty \\ \infty & \infty & 0 & \dots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \infty & \infty & \dots & \infty \end{pmatrix}$$

mátrix?

25.1-4. Igazoljuk, hogy a LEGRÖVIDEBB-ÚT-BŐVÍTÉS algoritmusban definiált mátrixszorzás asszociatív.

25.1-5. Mutassunk kapcsolatot az egy csúcsból vett legrövidebb utak keresése és a mátrixok egy vektorral vett szorzatai között. Mutassuk meg, hogy a szorzat kiértékelése egy Bellman–Ford típusú algoritmusnak felel meg (lásd 24.1. alfejezet).

25.1-6. Ha az alfejezetbeli algoritmusok segítségével a legrövidebb utak csúcsait is meg akarjuk adni, a Π megelőzési mátrixot is ki kell számítanunk. Adjunk $O(n^3)$ lépésű eljárást, amely a legrövidebb útsúlyok L mátrixából a Π mátrixot számítja.

25.1-7. A legrövidebb utak csúcsait súlyaikkal egy időben is megkaphatjuk a következőképpen. Legyen $\pi_{ij}^{(m)}$ a j csúcsot megelőző csúcs valamely i-ből j-be vezető legfeljebb m élből álló legrövidebb úton. Módosítsuk a LEGRÖVIDEBB-ÚT-BŐVÍTÉS és LASSÚ-MINDEN-LEGRÖVI-

DEBB-ÚT algoritmusokat úgy, hogy a $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ mátrixok számítása mellett a $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ mátrixokat is megadják.

25.1-8. Az alfejezetben megadott GYORSABB-MINDEN-LEGRÖVIDEBB-ÚT algoritmus $\lceil \lg(n-1) \rceil$ darab, egyenként n^2 elemű mátrixot tárol, teljes tárigénye tehát $\Theta(n^2 \lg n)$. Módosítsuk az algoritmust úgy, hogy csak két $n \times n$ mátrixot tároljon és így tárigénye csak $\Theta(n^2)$ legyen.

25.1-9. Módosítsuk a GYORSABB-MINDEN-LEGRÖVIDEBB-ÚT algoritmust úgy, hogy felismerje, ha a gráf tartalmaz negatív összsúlyú kört.

25.1-10. Készítsünk hatékony algoritmust, mely megadja a gráf élei közül legkevesebbet tartalmazó negatív összsúlyú kör hosszát (éleinek számát).

25.2. A Floyd–Warshall-algoritmus

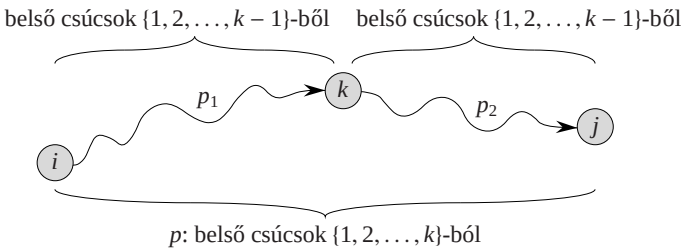
A következőkben az előző alfejezetétől eltérő dinamikus programozási feladatként értelmezzük a legrövidebb utak keresését. Egy $G = (V, E)$ irányított gráfon a keletkező ún. ***Floyd–Warshall-algoritmus*** futási ideje $\Theta(n^3)$ lesz. Az előző algoritmusokhoz hasonlóan a bemenő gráfban megengedünk negatív élsúlyokat, negatív összsúlyú köröket azonban nem. Az algoritmust a dinamikus programozás 25.1. alfejezetben összefoglalt fő lépései alapján fogjuk tervezni. A kapott algoritmus vizsgálata után egy hasonló algoritmust ismertetünk, mely egy irányított gráf tranzitív lezártját adja meg.

A legrövidebb utak szerkezete

A Floyd–Warshall-algoritmus kiindulópontja a legrövidebb utak mátrixszorzáson alapuló algoritmusétól eltérő jellemzése. Dinamikus programozásunk a legrövidebb utak „belső" csúcsait tekinti, ahol egy egyszerű $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ út ***belső*** csúcsa p minden v_1-től és v_ℓ-től különböző csúcsa, azaz a $\{v_2, v_3, \ldots, v_{\ell-1}\}$ halmaz minden eleme.

A legrövidebb utak szerkezetének ígért jellemzése a következő észrevételen alapul. Legyen a G gráf csúcshalmaza $V = \{1, 2, \ldots, n\}$, és tekintsük valamely k-ra az $\{1, 2, \ldots, k\}$ részhalmazt. Legyen p a legrövidebb i-ből j-be vezető olyan út, melynek belső csúcsait az $\{1, 2, \ldots, k\}$ részhalmazból választhatjuk. (A p út egyszerű, hiszen G nem tartalmaz negatív összsúlyú köröket.) A Floyd–Warshall-algoritmus a p út és az olyan legrövidebb utak kapcsolatát alkalmazza, melynek belső csúcsait az $\{1, 2, \ldots, k-1\}$ részhalmazból választjuk. E kapcsolat két esetre osztható attól függően, hogy k belső csúcsa-e p-nek vagy sem:

- Ha k a p útnak nem belső csúcsa, akkor a p út minden belső csúcsa az $\{1, 2, \ldots, k-1\}$ halmaz eleme. Így a legrövidebb i-ből j-be vezető és belső csúcsként csak az $\{1, 2, \ldots, k-1\}$ halmaz elemeit használó út szintén legrövidebb út lesz, ha a belső csúcsok az $\{1, 2, \ldots, k\}$ halmazból kerülhetnek ki.
- Ha k belső csúcs a p úton, akkor felbontjuk a p utat a 25.3. ábrán látható módon két $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ útra. A 24.1. lemma szerint p_1 egy olyan legrövidebb út i és k között, melynek belső csúcsai az $\{1, 2, \ldots, k\}$ halmaz elemei. Sőt k nem is lehet a p_1 út belső csúcsa, így p_1 egyben olyan legrövidebb út is, melynek belső csúcsai $\{1, 2, \ldots, k-1\}$-beliek. Hasonlóképpen p_2 egy legrövidebb, belső csúcsként csak $\{1, 2, \ldots, k-1\}$-et használó k-ból j-be vezető út.



25.3. ábra. A p út egy i-ből j-be vezető legrövidebb út. A p úton k a legnagyobb sorszámú belső csúcs. A p út i-ből k-ba vezető p_1 részútja belső csúcsként csak az $\{1, 2, \ldots, k-1\}$ halmaz elemeit használja; ugyanez igaz a p_2 k-ból j-be vezető részútra.

Az összes csúcspár közti legrövidebb utak rekurzív megadása

A fenti megfigyelés alapján a 25.1. alfejezettől eltérő rekurzióval adhatunk egyre javuló becsléseket a legrövidebb utak súlyára. Legyen $d_{ij}^{(k)}$ a legrövidebb olyan i-ből j-be vezető út hossza, melynek minden belső csúcsa az $\{1, 2, \ldots, k\}$ halmaz eleme. A $k = 0$ érték esetén egy olyan i-ből j-be vezető útnak, melyen a belső csúcsok sorszáma legfeljebb 0 lehet, egyáltalán nem lehet belső csúcsa. Egy ilyen útnak tehát legfeljebb egyetlen éle lehet és így $d_{ij}^{(0)} = w_{ij}$. A további értékeket a következő rekurzió szolgáltatja:

$$d_{ij}^{(k)} = \begin{cases} w_{ij,} & \text{ha } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), & \text{ha } k \geq 1. \end{cases} \tag{25.5}$$

A végeredményt a $D^{(n)} = (d_{ij}^{(n)})$ mátrix tartalmazza, hiszen az egyes legrövidebb utak belső csúcsai az $\{1, 2, \ldots, n\}$ halmaz elemei, és így $d_{ij}^{(n)} = \delta(i, j)$ minden $i, j \in V$ esetén.

Az úthosszak kiszámítása alulról felfelé haladva

A (25.5) rekurzió segítségével a $d_{ij}^{(k)}$ értékeket alulról felfelé, k értéke szerinti növekvő sorrendben számíthatjuk. Az algoritmus bemenő adata a (25.1) egyenlőség által definiált W $n \times n$ mátrix lesz, eredményül pedig a legrövidebb úthosszak $D^{(n)}$ mátrixát adja.

FLOYD–WARSHALL(W)

```
1  n ← sorok-száma[W]
2  D^(0) ← W
3  for k ← 1 to n
4      do for i ← 1 to n
5             do for j ← 1 to n
6                    d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

A 25.4. ábrán a Floyd–Warshall-algoritmus által számított $D^{(k)}$ mátrixok követhetők végig egy példaként választott irányított gráf esetén.

A Floyd–Warshall-algoritmus futási idejét a 3–6. sorok háromszorosan egymásba ágyazott **for** ciklusa határozza meg. A 6. sor minden egyes alkalommal $O(1)$ időben végrehajtható, így a teljes futási idő $\Theta(n^3)$. Csakúgy, mint a 25.1. alfejezet végső algoritmusa esetén,

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

25.4. ábra. A $D^{(k)}$ és $\Pi^{(k)}$ mátrixok, ha a Floyd–Warshall-algoritmust a 25.1. ábrán látható gráfon futtatjuk.

a megadott programkód tömör és csak egyszerű adatszerkezeteket igényel. A Θ-jelölésbeli állandó tehát kicsi, és a Floyd–Warshall-algoritmus még közepesen nagy méretű gráfok esetén is hatékony.

A legrövidebb utak megadása

A Floyd–Warshall-algoritmusban több módszer is kínálkozik arra, hogy a legrövidebb utak éleit is megkapjuk. Megtehető, hogy a legrövidebb úthosszak D mátrixának ismeretében $O(n^3)$ idő alatt megkonstruáljuk a Π megelőzési mátrixot (25.1-6. gyakorlat). A Π megelőzési mátrix ismeretében pedig a MINDEN-PÁRHOZ-UTAT-NYOMTAT eljárás megadja a kívánt két csúcs közti legrövidebb út éleit.

A Π megelőzési mátrixot számíthatjuk a Floyd–Warshall-algoritmus menetében a $D^{(k)}$ mátrixokkal egyidejűleg is. Pontosabban mátrixok egy $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$ sorozatát számíthatunk, melyre $\Pi = \Pi^{(n)}$. E sorozatban $\pi_{ij}^{(k)}$-t úgy definiáljuk, mint egy olyan legrövidebb i-ből j-be vezető úton a j-t megelőző csúcsot, melynek belső csúcsai az $\{1, 2, \ldots, k\}$ halmaz elemei.

Megadjuk a $\pi_{ij}^{(k)}$ értékeit definiáló rekurziót. (A $\Pi^{(k)}$ mátrixok számítása a 25.4. ábra példáján követhető.) Ha $k = 0$, akkor az i-től j-ig tartó úton nincs közbülső csúcs és így

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL}, & \text{ha } i = j \text{ vagy } w_{ij} = \infty, \\ i, & \text{ha } i \neq j \text{ és } w_{ij} < \infty. \end{cases} \tag{25.6}$$

A $k \geq 1$ esetben pedig egy $i \rightsquigarrow k \rightsquigarrow j$ úton a j-t megelőző csúcsnak választható ugyanaz a csúcs, amely j-t egy legrövidebb k-ból induló és belső csúcsként csak az $\{1, 2, \ldots, k-1\}$ halmaz elemeit használó úton előzi meg. Ha pedig a legrövidebb i-ből j-be vezető és az $\{1, 2, \ldots, k\}$ halmaz elemeit használó út k-t nem tartalmazza, akkor a j-t megelőző csúcs megegyezik a $k-1$ érték esetén választott megelőző csúccsal. Tehát $k \geq 1$ mellett a rekurzív egyenlet

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{ha } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{ha } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \tag{25.7}$$

A legrövidebb utak megszerkesztésével kapcsolatban több további kérdés a gyakorlatok között szerepel. Először is a $\Pi^{(k)}$ mátrix fenti rekurzív számítási lépéseit a FLOYD–WARSHALL eljárás lépései közé illeszthetjük (25.2-3. gyakorlat). Ugyanebben a gyakorlatban egy nehezebb kérdést is kitűzünk: annak igazolását, hogy a $G_{\Pi,i}$ megelőzési gráf egy i gyökerű legrövidebb utak fája lesz. Végül a 25.2-7. gyakorlat a legrövidebb utak megszerkesztésének egy további lehetőségét mutatja meg.

Irányított gráfok tranzitív lezártja

Egy adott $G = (V, E)$, $V = \{1, 2, \ldots, n\}$ irányított gráf minden egyes $i, j \in V$ csúcspárja esetén szükségünk lehet annak ismeretére, hogy létezik-e i-ből j-be vezető út a gráfban. A G gráf ***tranzitív lezártja*** az a $G^* = (V, E^*)$ gráf, melyre

$$E^* = \{(i, j) : \text{létezik } G\text{-ben } i\text{-ből } j\text{-be út}\}.$$

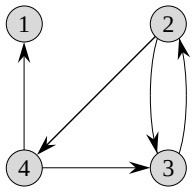
Egy lehetséges mód a tranzitív lezárt kiszámítására, ha E minden élsúlyát egységnyinek választjuk, és az így kapott gráfra alkalmazzuk a Floyd–Warshall-algoritmust. Ha létezik i-től j-ig út G-ben, akkor $d_{ij} < n$; különben $d_{ij} = \infty$. Az algoritmus $\Theta(n^3)$ lépést tesz.

A gyakorlatban időt és tárkapacitást takaríthatunk meg a Floyd–Warshall-algoritmus egy apró módosításával: az algoritmusban előforduló két aritmetikai műveletet, a min-t és a +-t két logikai műveletre, $\vee$-re (logikai VAGY) és $\wedge$-ra (logikai ÉS) cseréljük. A módosított algoritmus lépésszáma változatlanul $\Theta(n^3)$ lesz, a Θ jelölésben elrejtett állandó azonban csökken. Az $i, j, k = 1, 2, \ldots, n$ értékekre legyen $t_{ij}^{(k)} = 1$, ha van G-ben olyan út i-től j-ig, mely belső csúcsként csak az $\{1, 2, \ldots, k\}$ halmaz elemeit tartalmazza. Ellenkező esetben legyen $t_{ij}^{(k)} = 0$. A $G^* = (V, E^*)$ tranzitív lezártban $(i, j) \in E^*$ pontosan akkor, ha $t_{ij}^{(n)} = 1$. A $t_{ij}^{(k)}$ értékeit a (25.5) rekurzióhoz hasonlóan definiálhatjuk:

$$t_{ij}^{(0)} = \begin{cases} 0, & \text{ha } i \neq j \text{ és } (i, j) \notin E, \\ 1, & \text{ha } i = j \text{ vagy } (i, j) \in E, \end{cases}$$

$k \geq 1$ esetén pedig

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}). \tag{25.8}$$



25.5. ábra. Egy irányított gráf és a TRANZITÍV-LEZÁRT algoritmus által számított $T^{(k)}$ mátrixok:

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

A Floyd–Warshall-algoritmus lépéseinek megfelelően a $T^{(k)} = (t_{ij}^{(k)})$ mátrixokat k szerinti növekvő sorrendben számíthatjuk:

TRANZITÍV-LEZÁRT(G)

```
1  n ← |V[G]|
2  for i ← 1 to n
3      do for j ← 1 to n
4             do if i = j vagy (i, j) ∈ E[G]
5                    then t_ij^(0) ← 1
6                    else t_ij^(0) ← 0
7  for k ← 1 to n
8      do for i ← 1 to n
9             do for j ← 1 to n
10                   do t_ij^(k) ← t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
11 return T^(n)
```

A TRANZITÍV-LEZÁRT eljárás egy példagráfon futtatva a 25.5. ábrán látható $T^{(k)}$ mátrixokat adja. A Floyd–Warshall-algoritmussal megegyezően a futási idő $\Theta(n^3)$ lesz. A gyakorlatban azonban sok esetben gyorsabban hajtódnak végre az egyetlen bites adatokon vett logikai műveletek, mint az egész számokon végzett aritmetikai műveletek. További előny a tranzitív lezárt közvetlen keresésekor, hogy az egész értékek helyett csak logikai értékeket kell tárolni. Így a tárigény a Floyd–Warshall-algoritmushoz képest annyiszor kisebb, amennyi bitet számítógépünk egy egész szám tárolására használ.

Gyakorlatok

25.2-1. Adjuk meg a külső **for** ciklus egyes iterációi során keletkező $D^{(k)}$ mátrixokat, ha a Floyd–Warshall-algoritmust a 25.2. ábrán szereplő súlyozott irányított gráfon futtatjuk.

25.2-2. A 25.1. alfejezet módszerével adjunk algoritmust a tranzitív lezárt számítására.

25.2-3. Egészítsük ki a FLOYD–WARSHALL eljárást úgy, hogy a $\Pi^{(k)}$ mátrixokat a (25.6) és (25.7) egyenlőségek alapján számítsa. Bizonyítsuk be, hogy minden $i \in V$ csúcsra a $G_{\pi,i}$

megelőzési gráf egy i gyökerű legrövidebb utak fája. (*Útmutatás.* Annak igazolására, hogy $G_{\pi,i}$ körmentes, lássuk be, hogy ha $\pi_{ij}^{(k)} = \ell$, akkor $d_{ij}^{(k)} \geq d_{i\ell}^{(k)} + w_{\ell j}$. Ezután kövessük a 24.16. lemma bizonyításának lépéseit.)

25.2-4. Az alfejezetben megadott Floyd–Warshall-algoritmus tárigénye $\Theta(n^3)$, hiszen minden $i, j, k = 1, 2, \ldots, n$ esetén kiszámítja a $d_{ij}^{(k)}$ értékét. Igazoljuk, hogy helyes az alábbi algoritmus, amely a felső indexeket elhagyja és így tárigénye csak $\Theta(n^2)$.

```
FLOYD–WARSHALL′(W)
1  n ← sorok-száma[W]
2  D ← W
3  for k ← 1 to n
4      do for i ← 1 to n
5             do for j ← 1 to n
6                    d_ij ← min(d_ij, d_ik + d_kj)
7  return D
```

25.2-5. Helyes-e a Π megelőzési mátrixokat adó (25.7) rekurzió következő módosított változata:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{ha } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{ha } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}? \end{cases}$$

25.2-6. Hogyan ismerhető fel a Floyd–Warshall-algoritmus kimenetének ismeretében, ha a gráf negatív összsúlyú kört tartalmazott?

25.2-7. A Floyd–Warshall-algoritmus segítségével a legrövidebb utak egy további módon is megtalálhatók. Legyen $i, j, k = 1, 2, \ldots, n$ esetén $\phi_{ij}^{(k)}$ a legnagyobb sorszámú belső csúcs valamely i-ből j-be vezető legrövidebb és belső csúcsként az $\{1, 2, \ldots, k\}$ elemeit használó úton. Adjuk $\phi_{ij}^{(k)}$-t meg rekurzívan; módosítsuk a Floyd–Warshall-algoritmust úgy, hogy $\phi_{ij}^{(k)}$ értékeit is számítsa; végül alakítsuk a MINDEN-PÁRHOZ-UTAT-NYOMTAT eljárást át úgy, hogy bemenő adata a $\Phi = (\phi_{ij}^{(n)})$ mátrix legyen. Mi a hasonlóság a Φ mátrix és a 15.2. alfejezetbeli mátrixlánc szorzásra használt s táblázat között?

25.2-8. Adjunk algoritmust, amely egy irányított $G = (V, E)$ gráf tranzitív lezártját $O(VE)$ időben megadja.

25.2-9. Tételezzük fel, hogy a tranzitív lezárt irányított *körmentes* gráfon $f(V, E)$ időben számítható, ahol az $f(V, E)$ mindkét változójában monoton növő függvény. Mutassuk meg, hogy tetszőleges $G = (V, E)$ irányított gráf $G^*(V, E^*)$ tranzitív lezártja megtalálható $f(V, E) + O(V + E^*)$ időben.

25.3. Johnson ritka gráfokon hatékony algoritmusa

Johnson módszerével az összes csúcspár közti legrövidebb utakat $O(V^2 \lg V + VE)$ időben találhatjuk meg, ritka gráfokon tehát aszimptotikusan gyorsabban, mint akár ismételt négyzetre emelésekkel, akár a Floyd–Warshall-algoritmussal. Johnson algoritmusa vagy a legrövidebb útsúlyok mátrixának megadásával ér véget, vagy annak felismerésével, hogy a

kérdéses gráf negatív összsúlyú kört tartalmaz. Szubrutinként a 24. fejezetben ismertetett Dijkstra és Bellman–Ford-algoritmusokat fogjuk alkalmazni.

A módszer alapötlete a következő ***átsúlyozó*** technika. Ha a $G = (V, E)$ gráf minden w élsúlya nemnegatív, akkor az összes csúcspár közti legrövidebb utakat megtalálhatjuk Dijkstra algoritmusát minden lehetséges gyökérpontra alkalmazva. Ha tehát a szükséges elsőbbségi sorokat Fibonacci-kupacokkal valósítjuk meg, a futási idő az ígért $O(V^2 \lg V + VE)$ lesz. Ha G nem tartalmaz negatív kört, azonban negatív élsúlyok is előfordulnak, akkor olyan olyan új $\hat{w}$ nemnegatív értékeket kell találnunk, melyeken azután az előző megoldás alkalmazható. A $\hat{w}$ súlyok két fontos feltételnek fognak eleget tenni:

1. Minden $u, v \in V$ csúcspárra a w súlyozás szerint vett bármely legrövidebb u-ból v-be vezető út legrövidebb út a $\hat{w}$ súlyozás esetén is.
2. Minden $(u, v) \in E$ élre az új $\hat{w}(u, v)$ súly nemnegatív.

A következőkben megmutatjuk, hogy a fentieknek eleget tevő $\hat{w}$ súlyozás $O(VE)$ időben számítható.

A legrövidebb utakat megőrző átsúlyozás

A következő lemma alapján igen egyszerű a fenti első tulajdonságnak eleget tevő módon átsúlyozni egy gráfot. Jelölje $\delta(u, v)$ a w, $\hat{\delta}(u, v)$ a $\hat{w}$ súlyozás esetén keletkező legrövidebb u-ból v-be vezető út súlyát.

25.1. lemma (átsúlyozás). *Legyen $h : V \mapsto \mathbf{R}$ tetszőleges valós értékű függvény egy irányított $G = (V, E)$ gráf csúcsain. Legyen $w : E \mapsto \mathbf{R}$ ugyanennek a gráfnak élsúlyozása. Minden $(u, v) \in E$ élre legyen*

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v). \tag{25.9}$$

Ekkor minden v_0, v_k csúcspárra és minden $p = \langle v_0, v_1, \ldots, v_k \rangle$ útra pontosan akkor lesz $w(p) = \delta(v_0, v_k)$, ha $\hat{w}(p) = \hat{\delta}(v_0, v_k)$; továbbá pontosan akkor van a G gráfban negatív összsúlyú kör a w súlyozás esetén, ha a $\hat{w}$ súlyozás esetén is van.

Bizonyítás. Először belátjuk, hogy

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k). \tag{25.10}$$

Teljesül, hogy

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_i) - h(v_{i-1})) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + \sum_{i=1}^{k} h(v_i) - \sum_{i=1}^{k} h(v_{i-1}) \qquad \text{(mivel teleszkóp összegek)} \\
&= w(p) + h(v_0) - h(v_k).
\end{aligned}$$

A fentiek szerint minden egyes v_0-ból v_k-ba vezető p út hosszára $\hat{w}(p) = w(p)+h(v_0)-h(v_k)$ teljesül. Ha tehát egy v_0-ból v_k-ba vezető út a w súlyozás szerint rövidebb egy másiknál, akkor ugyanez teljesül a $\hat{w}$ súlyozás szerint is, és fordítva. Így tehát $w(p) = \delta(v_0, v_k)$ akkor és csak akkor, ha $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.

Végül bebizonyítjuk, hogy pontosan akkor van a G gráfban negatív összsúlyú kör a w súlyozás esetén, ha a $\hat{w}$ súlyozás esetén is van. Legyen $c\langle v_0, v_1, \ldots, v_k\rangle$ egy tetszőleges kör, ahol $v_0 = v_k$. Ekkor a (25.10) egyenlőség szerint

$$\begin{aligned}\hat{w}(c) &= w(c) + h(v_0) - h(v_k)\\ &= w(c),\end{aligned}$$

így c súlya pontosan akkor negatív a w súlyozásra, ha negatív a $\hat{w}$ súlyozásra. ■

Nemnegatív értékű átsúlyozás

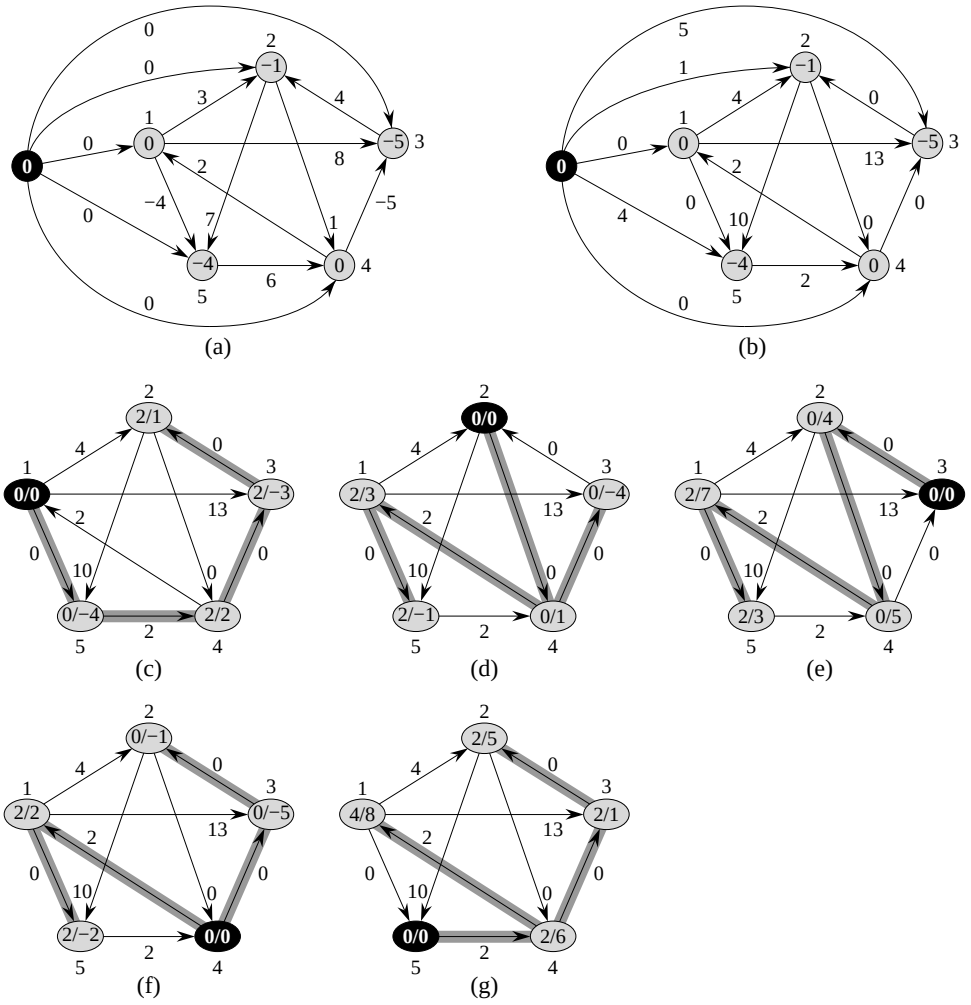
A $\hat{w}$ súlyozás első tulajdonságának biztosítása után térjünk a másodikra: célunk az, hogy minden $(u, v) \in E$ élre az új $\hat{w}(u, v)$ súly nemnegatív legyen. Egy $w : E \mapsto \mathbf{R}$ súlyozással ellátott irányított $G = (V, E)$ gráfot egészítsük ki egyetlen $s \notin V$ csúcs hozzáadásával egy $G' = (V', E')$ gráffá, ahol tehát $V' = V \cup \{s\}$ és $E' = E \cup \{(s, v) : v \in V\}$. A w súlyozást kiterjesztjük G' éleire úgy, hogy $w(s, v) = 0$ minden $v \in V$ esetén. Fontos észrevétel, hogy s-be él nem lép, így kizárólag azok a G'-beli legrövidebb utak tartalmazzák s-t, melyek s-ből indulnak; ugyanezért G'-ben pontosan akkor van negatív összsúlyú kör, ha G-ben. Ha a 25.1. ábra G gráfját a fentiek szerint kibővítjük egy G' gráffá, a keletkező gráfot a 25.6(a) ábrán láthatjuk.

Tételezzük fel, hogy G és G' nem tartalmaz negatív összsúlyú kört és legyen $h(v) = \delta(s, v)$ minden $v \in V'$ csúcsra. A háromszög-egyenlőtlenség (24.10. lemma) szerint $h(v) \leq h(u) + w(u, v)$ minden $(u, v) \in E'$ élre. Ha tehát az új $\hat{w}$ súlyozást a (25.9) egyenlet szerint definiáljuk, $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$ és a nemnegativitást megkövetelő második tulajdonság teljesül. A 25.6(a) ábra gráfját átsúlyozva a 25.6(b) ábra gráfjához jutunk.

Az úthosszak kiszámítása

Johnson most következő algoritmusa a Bellman–Ford- (24.1. alfejezet) és Dijkstra-algoritmusokat (24.3. alfejezet) szubrutinként használja. A bemenet a gráf éllistája; a kimenet a szokásos módon vagy egy $D = (d_{ij})$ $|V| \times |V|$ méretű mátrix a $d_{ij} = \delta(i, j)$ súlyokkal, vagy annak jelzése, hogy a gráf negatív összsúlyú kört tartalmaz. A D mátrix indexelése céljából feltesszük, hogy a gráf csúcshalmaza $\{v_1, v_2, \ldots, v_n\}$, ahol $n = |V|$.

Az alábbi program sorai a korábban ismertetett lépéseket hajtják végre. Az 1. sorban G'-t építjük fel. A 2. sorban a Bellman–Ford-algoritmust hívjuk meg a w súlyokkal ellátott G' gráfra. Ha G' és így G negatív súlyú kört tartalmaz, ezt a 3. sorban jelezzük; a 4–11. sorokban feltesszük, hogy nem ez az eset áll fenn. A 4–5. sorokban $h(v)$ a Bellman–Ford-algoritmus által számított $\delta(s, v)$ legrövidebb úthossz lesz minden $v \in V'$ csúcsra. Az új $\hat{w}$ súlyokat a 6–7. sorokban számítjuk. Dijkstra algoritmusa megadja minden $u, v \in V$ csúcspárra a $\hat{\delta}(u, v)$ legrövidebb úthosszakat a 9–11. sorokban; az algoritmust a 8. sor **for** ciklusa minden egyes $u \in V$ csúcsra meghívja. A 11. sorban a (25.10) egyenlőség szerint számítjuk



25.6. ábra. Johnson algoritmusa a 25.1. ábra G gráfján a következő lépésekben találja meg az összes csúcspár közti legrövidebb utakat. **(a)** A kiegészített G' gráf az eredeti w súlyokkal. Az új s csúcs fekete. Minden egyes v csúcs belsejében $h(v) = \delta(s, v)$ áll. **(b)** Az (u, v) élek új súlya $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(c)–(g)** Dijkstra algoritmusa által szolgáltatott értékek a $\hat{w}$ súlyokkal, az öt lehetséges forráspont választásával. Az u forráspontot feketével jelöltük; az egyes v csúcsok belsejében törtvonallal elválasztva $\hat{\delta}(u, v)$ és $\delta(u, v)$ áll. $d_{uv} = \delta(u, v)$ megegyezik a $\hat{\delta}(u, v) + h(v) - h(u)$ értékkel.

$d_{ij} = \delta(v_i, v_j)$ legrövidebb úthosszakat. Végül a 12. sorban a végeredmény az így kapott D mátrix. Johnson algoritmusának lépései a 25.6. ábrán követhetők.

Johnson algoritmusának futási ideje könnyen láthatóan $O(V^2 \lg V + VE)$, ha a Dijkstra algoritmusában használt minimumot szolgáltató elsőbbségi sorokat Fibonacci-kupacokkal valósítjuk meg. Az egyszerűbb bináris min-kupac adatszerkezettel kapott $O(VE \lg V)$ futási idő is aszimptotikusan jobb a Floyd–Warshall-algoritmusénál ritka gráfok esetén.

JOHNSON(G)

```
 1  Legyen G' = (V', E'), ahol V[G'] = V[G] ∪ {s}
                               E[G'] = E[G] ∪ {(s, v) : v ∈ V[G]}
                               w(s, v) = 0 minden v ∈ V[G] esetén.
 2  if BELLMAN–FORD(G', w, s) = HAMIS
 3    then print „A megadott gráf negatív összsúlyú kört tartalmaz”
 4    else for i ← 1 to n + 1
 5              do h(v_i) ← a Bellmann–Ford-algoritmus által adott δ(s, v_i) érték
 6         for (u, v) ← az E[G'] éllista elemei
 7              do ŵ(u, v) ← w(u, v) + h(u) – h(v)
 8         for i ← 1 to n
 9              do minden v ∈ V δ̂(v_i, v) ← DIJKSTRA(G, ŵ, v_i)
10                 for j ← 1 to n
11                      do d_ij ← δ̂(v_i, v_j) + h(v_j) – h(v_i)
12  return D
```

Gyakorlatok

25.3-1. Johnson algoritmusának segítségével keressük meg a legrövidebb utakat a 25.2. ábra minden csúcspárja között. Adjuk meg az algoritmus által számított h és $\hat{w}$ értékeket.

25.3-2. Mi a célja annak, hogy a G gráfot egy új s csúcs hozzáadásával kibővítjük?

25.3-3. Feltéve, hogy $w(u, v) \geq 0$ minden $(u, v) \in E$ élre, mi lesz a w és a $\hat{w}$ súlyozások kapcsolata?

25.3-4. Zöldi Professzor szerint Johnson algoritmusánál egyszerűbben is átsúlyozhatjuk egy gráf éleit: legyen $w^* = \min_{(u,v)\in E}\{w(u, v)\}$, és ekkor egyszerűen használjuk a $\hat{w}(u, v) = w(u, v) - w^*$ értékeket. Mi a hiba a professzor módszerében?

25.3-5. Futtassuk Johnson algoritmusát egy G irányított, w súlyfüggvénnyel ellátott gráfon. Mutassuk meg, hogy ha G tartalmaz egy c nulla összhosszú kört, akkor $\hat{w}(u, v) = 0$ a c kör minden (u, v) élére.

25.3-6. Hamari professzor szerint felesleges Johnson algoritmusának első sorában az új s forráscsúcs választása. Azt állítja, hogy elegendő a $G = G'$ gráffal dolgozni és s-ként a G egy tetszőleges csúcsát választani. Adjunk meg olyan súlyozott irányított G gráfot, amelyen Hamari professzor módosítása hibás eredményre vezet. Mutassuk ezután meg, hogy ha G erősen összefüggő (minden csúcsából minden másik irányított úton elérhető), akkor Johnson algoritmusa a professzor módosításával is helyes marad.

Feladatok

25-1. Dinamikus tranzitív lezárt algoritmus

Tekintsük a következő algoritmikus feladatot: egy irányított $G = (V, E)$ gráf tranzitív lezártját új élek beszúrása esetén fel kell újítanunk. Pontosabban az E élhalmazhoz egyenként új éleket adunk, és minden ilyen lépés után meg kell határoznunk a pillanatnyi élhalmaz tranzitív lezártját. Tételezzük fel, hogy a kiinduló gráf üres (él nélküli). A tranzitív lezártat logikai értékekből álló mátrix alakjában keressük.

a. Bizonyítsuk be, hogy egy ismert tranzitív lezártú $G = (V, E)$ gráf $G^* = (V, E^*)$ tranzitív lezártja új él beszúrása után $O(V^2)$ időben meghatározható.

b. Adjunk meg egy G gráfot és egy olyan e élt, melynek G gráfhoz adása után a tranzitív lezárt meghatározása $\Omega(V^2)$ időt igényel – függetlenül attól, hogy a tranzitív lezártat melyik algoritmussal határozzuk meg.

c. Adjunk hatékony algoritmust, mely ismételt élbeszúrások esetén meghatározza egy irányított gráf tranzitív lezártját. Tetszőleges k él beszúrásából álló sorozat esetén az algoritmus teljes futási idejére legyen igaz $\sum_{i=1}^{k} t_i = O(V^3)$, ahol t_i az i-edik él beszúrása utáni tranzitív lezárt számítási ideje.

25-2. Legrövidebb utak ϵ-sűrű gráfokban

Egy $G = (V, E)$ gráf ***ϵ-sűrű***, ha $|E| = \Theta(V^{1+\epsilon})$ egy adott $0 < \epsilon \leq 1$ értékre. A 6-2. feladatban szereplő d-rendű kupacok segítségével most ϵ-sűrű gráfok legrövidebb úthosszait fogjuk megkeresni. Célunk a Fibonacci-kupacon alapuló algoritmusok aszimptotikus futási idejét bonyolult adatszerkezetek alkalmazása nélkül elérni.

a. Mi a BESZÚR, KIVESZ-MIN, KULCSOT-CSÖKKENT lépések futási ideje d és n függvényében, ahol n a d-rendű kupac elemszáma? Mi a helyzet a $d = \Theta(n^\alpha)$ választása mellett, ahol $0 < \alpha \leq 1$ értéke állandó? Hogyan viszonyulnak e futási idők a Fibonacci-kupacra kapott amortizált időkhöz?

b. Mutassuk meg, hogy egy ϵ-sűrű irányított $G = (V, E)$ gráf adott csúcsától vett legrövidebb úthosszak $O(E)$ időben számíthatók, feltéve, hogy a gráf nem tartalmaz negatív élsúlyokat. (*Útmutatás.* Válasszuk d értékét ϵ egy alkalmas függvényeként.)

c. Mutassuk meg, hogy a **b.** kérdés feltételei mellett minden csúcspárra megkaphatjuk a legrövidebb úthosszakat $O(VE)$ időben.

d. Mutassuk meg, hogy minden csúcspárra megkaphatjuk a legrövidebb úthosszakat $O(VE)$ időben akkor is, ha az élhosszak lehetnek negatívak, de a gráfunk nem tartalmaz negatív összsúlyú kört.

Megjegyzések a fejezethez

Lawler [196] részletesen ismerteti az összes párra vonatkozó legrövidebb út problémát, azonban az algoritmusok hatékonyságát ritka gráfok esetén nem vizsgálja. A mátrixszorzáson alapuló algoritmus eredete Lawler szerint ismeretlen. Warshall [308] ad egy logikai értékeket tartalmazó mátrix tranzitív lezártját számító algoritmust. Warshall e tételére alapozva a Floyd–Warshall-algoritmust először Floyd [89] írja le. Johnson [166] tartalmazza Johnson algoritmusát.

A mátrixszorzáson alapuló legrövidebb út algoritmus javítására számos kutatási eredmény született. Fredman [95] megmutatja, hogy a minden csúcspár közti legrövidebb út probléma megoldható élsúlyok összegeinek $O(V^{5/2})$ darab összehasonlításával, és ezzel a Floyd–Warshall-algoritmusénál aszimptotikusan kicsit jobb $O(V^3(\log\log V/\log V)^{1/3})$ futási idejű algoritmus adható. A futási idő javításának egy másik lehetséges iránya gyors mátrixszorzó algoritmusok (lásd a megjegyzéseket a 28. fejezethez) alkalmazása. Legyen az $n \times n$ mátrixokat szorzó leggyorsabb algoritmus futási ideje $O(n^\omega)$ – a jelenlegi legjobb eredmény Coppersmith és Winograd [70] nevéhez fűződik: $\omega < 2,376$. Galil és Margalit [105, 106], valamint Seidel [270] megmutatják, hogy irányítatlan és súlyozatlan gráfokban a

minden csúcspár közti legrövidebb út probléma megoldható $O(V^{\omega}p(V))$ időben, ahol $p(V)$ egy bizonyos polilogaritmikusan kicsi függvény. Sűrű gráfokon ez a módszer aszimptotikusan gyorsabb, mint a V darab szélességi keresés $O(VE)$ ideje. A gyors mátrixszorzáson alapuló módszer alkalmazhatóságát többen megmutatták olyan esetekben is, amikor az irányítatlan gráf éleinek súlya az $\{1, 2, \ldots, W\}$ tartománybeli egész. A legjobb ilyen algoritmus Shoshantól és Zwicktől [278] származik és futási ideje $O(WV^{\omega}p(VW))$.

Jelölje E^* azokat az E-beli éleket, amelyeket legalább egy legrövidebb út tartalmaz. Karger, Koller és Phillips [170], és tőlük függetlenül McGeoch [215] olyan legrövidebb út algoritmust adtak, amelynek futási ideje csak E^*-tól függ. Amennyiben az élsúlyok nemnegatívak, a futási idő $O(VE^* + V^2 \log V)$, amely az $|E^*| = o(E)$ esetben jobb, mint a Dijkstra-algoritmus $|V|$-szeri alkalmazása.

Aho, Hopcroft és Ullman [5] definiálják az úgynevezett „zárt félgyűrű” algebrai struktúrát, amely általános keretet ad irányított gráfok utakkal kapcsolatos problémáinak megoldására. A 25.2. szakaszbeli Floyd–Warshall és tranzitív lezárt algoritmusok például egyazon, a zárt félgyűrűkön alapuló minden csúcspár között utat keresős algoritmus speciális esetei. Maggs és Plotkin [208] zárt félgyűrűk segítségével megmutatják, hogy ugyanez az algoritmus minimális feszítőfák keresésére is alkalmazható.

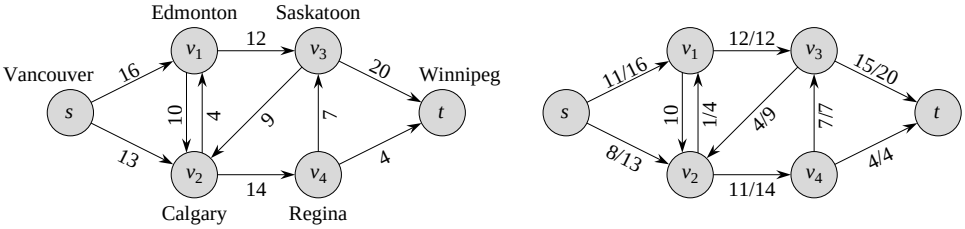
26. Maximális folyamok

Amint egy autóstérképet modellezhetünk irányított gráffal azért, hogy megtaláljuk a legrövidebb utat egy adott pontból egy másikba, úgy az irányított gráfokat elképzelhetjük „folyamhálózatként” is, hogy anyagok áramlására vonatkozó kérdéseket meg tudjunk válaszolni. Képzeljük el, hogy valamilyen anyag áramlik egy olyan rendszeren keresztül, ami egy „termelőből”, amelyben az anyag készül, egy „fogyasztóba” vezet, ahol is az anyag felhasználásra kerül. A termelő állandó ütemben előállítja a szóban forgó anyagot, a fogyasztó pedig ugyanebben az ütemben felhasználja azt. A „folyam” a rendszer egy pontjában az a sebesség vagy ütem, amellyel az anyag mozog. A hálózati folyamokat alkalmazhatjuk cseppfolyós anyagok csőhálózaton keresztül való áramlásának a modellezésére, munkadarabok futószalagon való haladásának, villamos áram elektromos hálózatban vagy információ kommunikációs hálózatban való áramlásának a modellezésére és még sok egyébre.

A hálózat minden irányított élét vezetéknek képzeljük el, amin keresztül anyag áramolhat. Minden vezetéknek meghatározott kapacitása van, ami az a lehető legnagyobb anyagmennyiség, amely a vezetéken képes átáramlani, például egy vízvezeték maximálisan 2000 liter vizet tud percenként átengedni, vagy egy villamos vezeték legfeljebb 20 amperes áramot bír el. A gráf csúcsai a vezetékek összeillesztési és elágazási pontjai, amelyekben – ha nem a termelőről vagy a fogyasztóról van szó – az anyag nem gyűlhet fel az áramlás közben. Vagyis a beáramló anyag mennyiségének egy pontban meg kell egyeznie a kiáramló anyag mennyiségével. A hálózati folyam ezen tulajdonságát „megmaradási szabálynak” nevezzük, ami villamos áram esetén megegyezik Kirchhoff csomóponti törvényével.

A „maximális folyam probléma” azt kérdezi, hogy mekkora az a maximális anyagmennyiség, amit a kapacitási előírások megsértése nélkül a termelőtől el tudunk juttatni a hálózaton keresztül a fogyasztóhoz. Amint látni fogjuk, ez az egyik legegyszerűbb probléma, amelyet hálózati folyamokkal kapcsolatban felvethetünk, és meg is adunk néhány hatékony algoritmust a kérdés megválaszolására. Ráadásul ezen algoritmusok segítségével más folyamokkal kapcsolatos feladatokat is meg tudunk oldani.

Ebben a fejezetben két általános módszert fogunk bemutatni a maximális folyam probléma megoldására. A 26.1. alfejezetben formálisan definiáljuk a hálózatokra és folyamokra vonatkozó fogalmainkat. A 26.2. alfejezetben ismertetjük Ford és Fulkerson immáron klasszikusnak nevezhető algoritmusát a maximális folyam megkeresésére. Ennek a módszernek egy alkalmazását mutatjuk meg a 26.3. alfejezetben páros gráfok maximális párosításának a megkeresésére. A 26.4. alfejezetben megismerkedünk az úgynevezett el őfolyam-



(a) (b)

26.1. ábra. (a) A Hokipakk Vállalat szállítási problémájának hálózata. A vancouveri gyár az s termelőben van, a winnipegi raktár pedig a t fogyasztóban. A jégkorongokat a közbeeső városokon keresztül szállítják, de az u városból a v városba legfeljebb csak $c(u, v)$ ládával naponta. A gráf minden élén az él kapacitása szerepel, a be nem húzott élek kapacitása 0. **(b)** Egy 19 nagyságú f folyam. Csak a pozitív folyamértékeket írtuk be az ábrába. Ha $f(u, v) > 0$, akkor az (u, v) élen $f(u, v)/c(u, v)$ szerepel, vagyis a folyam értéke az (u, v) élen, majd egy perjel, amit az él kapacitása követ. Ha $f(u, v) \leq 0$, akkor az (u, v) élre csak a kapacitását írtuk.

módszerrel, ami a hálózati folyamokra vonatkozó mai leggyorsabb algoritmusok többségének alapjául szolgál. A 26.5. alfejezetben egy $O(V^3)$ futási idejű előfolyam-algoritmust írunk le. Bár ez az algoritmus nem a leggyorsabb ismert változat, mégis jól példázza azokat a technikákat, amelyeket az aszimptotikusan leggyorsabb algoritmusok használnak, és még a gyakorlatban is elfogadhatóan hatékonynak bizonyul.

26.1. Hálózati folyamok

Ebben a fejezetben megadjuk a hálózatok és folyamok pontos gráfelméleti definícióit, majd megismerkedünk néhány alapvető összefüggéssel, végül bevezetünk egy hasznos jelölést.

Hálózatok és folyamok

Ha a $G = (V, E)$ irányított gráf minden $(u, v) \in E$ élének adott egy nemnegatív ***kapacitása*** – $c(u, v) \geq 0$ –, akkor a gráfot ***hálózatnak*** nevezzük. Ha $(u, v) \notin E$, akkor feltesszük, hogy $c(u, v) = 0$. A hálózatnak kitüntetjük két pontját: az s ***termelőt*** és a t ***fogyasztót.*** (Szokás a termelőt ***forrásnak,*** a fogyasztót pedig ***nyelőnek*** nevezni.) A könnyebbség kedvéért feltehetjük, hogy minden $v \in V$ pont rajta van valamely s-ből t-be vezető úton. Következésképpen a gráf összefüggő, ezért $|E| \geq |V| - 1$. A 26.1. ábrán egy hálózatot láthatunk.

Most már rátérhetünk a folyamok formális definíciójára. Legyen $G = (V, E)$ egy hálózat az éleken a c kapacitásfüggvénnyel. Legyen s a hálózat termelője, t a fogyasztója. ***Hálózati folyamnak*** (vagy csak ***folyamnak***) nevezzük a következő három tulajdonsággal rendelkező $f : V \times V \to \mathbf{R}$ valósértékű függvényt:

Kapacitási megszorítás: $f(u, v) \leq c(u, v)$ teljesül minden $u, v \in V$-re.

Ferde szimmetria: $f(u, v) = -f(v, u)$ teljesül minden $u, v \in V$-re.

Megmaradási szabály: $\sum_{v \in V} f(u, v) = 0$ teljesül minden $u \in V - \{s, t\}$-re.

Az $f(u, v)$ értéket, ami egyaránt lehet pozitív, negatív vagy 0, az u-ból v-be vezető élen a ***folyam értékének*** nevezzük. Az egész folyamra vonatkozó ***folyamnagyságot*** a következőképp definiáljuk:

$$\|f\| = \sum_{v \in V} f(s, v), \tag{26.1}$$

azaz az s-ből kiinduló összes folyam. ***Maximális folyam problémának*** azt nevezzük, amikor egy adott G hálózatban az s-ből t-be vezető maximális nagyságú folyamot keressük.

Mielőtt szemügyre vennénk egy konkrét hálózati folyamot, vizsgáljuk meg a fenti három tulajdonságot. A kapacitási megszorítás szerint a folyam értéke nem haladhatja meg egy élen az adott él kapacitását. A ferde szimmetria mindössze egy technikai eszköz, amely azt mondja, hogy az u pontból a v pontba a folyam nagysága ellentettje a v-ből u-ba menő folyamnak. A megmaradási szabály szerint a termelő és a fogyasztó kivételével semelyik pontban nem keletkezhet vagy tűnhet el folyam. A ferde szimmetriából kifolyólag a megmaradási szabályt átírhatjuk a következő alakba:

$$\sum_{u \in V} f(u, v) = 0,$$

minden $v \in V - \{s, t\}$-re, azaz az egy pontban keletkező folyam nagysága 0.

Amennyiben (u, v) és (v, u) egyike sem éle a hálózatnak, akkor nem mehet folyam a két pont között, azaz $f(u, v) = f(v, u) = 0$ (lásd 26.1-1. gyakorlat).

A folyamok elemi tulajdonságaira vonatkozó utolsó megfigyelésünk azokkal a folyamértékekkel foglalkozik, amelyek pozitívak. A v pontba ***bemenő folyam nagysága*** definíció szerint a következő:

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v). \tag{26.2}$$

Az egy pontból kimenő folyam nagyságát szimmetrikusan definiáljuk. Egy csúcs ***teljes folyamát*** úgy definiáljuk, mint a csúcsból kimenő teljes folyam és a csúcsba bemenő teljes folyam különbségét. A megmaradási szabály szemléletes megfogalmazása, amikor azt mondjuk, hogy minden – a forrástól és a nyelőtől különböző – pontba bemenő teljes folyam nagyságának meg kell egyeznie a pontból kimenő folyam nagyságával. Erre a tulajdonságra, amely szerint a csúcs teljes folyamának nullának kell lennie, informálisan gyakran úgy hivatkoznak, mint „a bemenő folyam egyenlő a kimenő folyammal."

Egy példa

Hálózati folyammal modellezni tudjuk a 26.1. ábra szállítási problémáját. A Hokipakk Vállalatnak van egy jéghokikorongokat előállító gyára Vancouverben és egy raktára Winnipegben. A vállalat egy másik cégtől bérli a teherautókat, amelyekkel a gyárból a raktárba szállítják a korongokat. Mivel a teherautók két város (pont) között meghatározott útvonalon (élen) közlekednek, és a rakterük korlátozott, ezért a Hokipakk Vállalat legfeljebb egy meghatározott $c(u, v)$ darab ládányi korongot tud naponta az u városból a v városba szállíttatni, ahogyan az a 26.1(a) ábrán látható. A Hokipakk Vállalat sem a szállítási útvonalakat nem tudja megváltoztatni, sem a teherautók rakterének befogadóképességét, ezért a 26.1(a) ábrán szereplő hálózat sem módosítható. Céljuk a lehető legnagyobb p ládaszám meghatározása, amelyet naponta át tudnak szállíttatni a gyárból a raktárba. Ezt követően majd napi p láda jégkorongot fognak termelni, hiszen nem tudnának nagyobb mennyiséggel mit kezdeni. Az, hogy mennyi ideig szállítják a korongokat, nem fontos a vállalat

szempontjából, jelentősége csak annak van, hogy p ládányi mennyiséget el tudnak minden nap indítani a gyárból, és p láda pedig meg is érkezik a raktárba.

Első ránézésre úgy tűnik, hogy a jégkorongokat tartalmazó ládák szállítása folyamot alkot a megfelelő hálózatban, hiszen egyik városból egy másik városba szállított ládák mennyisége nem lehet több az előre meghatározott $c(u, v)$ értékeknél. Továbbá a folyam megmaradási szabályának is fenn kell állnia, hiszen egy közbeeső városban a beáramló ládák számának meg kell egyeznie az onnan továbbszállított ládák számával, különben a ládák felgyülemlenének a közbeeső városokban.

Van azonban egy árnyalatnyi különbség a folyamok és a szállítások között. Előfordulhat, hogy a Hokipakk ládák szállítását tervezi Edmontonból Calgaryba, és Calgaryból Edmontonba is. Tegyük fel, hogy naponta 8 ládát szállítanak Edmontonból (v_1-gyel jelöltük a 26.1. ábrán) Calgaryba (v_2), és 3 ládát szállítanak Calgaryból Edmontonba. Úgy tűnik, hogy ezeket a szállításokat azon nyomban megfogalmazhatjuk folyamként, ám ez közvetlenül nem tehető meg. $f(v_1, v_2) = 8$, $f(v_2, v_1) = 3$ mellett nem teljesül ugyanis a ferde szimmetria, azaz az $f(v_1, v_2) = -f(v_2, v_1)$ feltétel.

A Hokipakk esetleg észleli, hogy nincs sok értelme 8 láda jégkorongot szállítani Edmontonból Calgaryba, ugyanakkor pedig 3 ládával Calgaryból Edmontonba, hiszen ugyanazt a mennyiséget el tudják juttatni a raktárba, ha Edmontonból Calgaryba csak 5 ládával szállítanának, Calgaryból Edmontonba pedig nem szállítanának semennyit (feltehetően ez a megoldás még olcsóbb is lenne). Ezt az utóbbi szállítást már a ferde szimmetriának megfelelően az $f(v_1, v_2) = 5$, $f(v_2, v_1) = -5$ értékekkel kifejezhetjük. Összességében tehát a v_1-ből v_2-be szállított 8 ládából 3-at ***kitörölt*** a v_2-ből v_1-be szállított 3 láda.

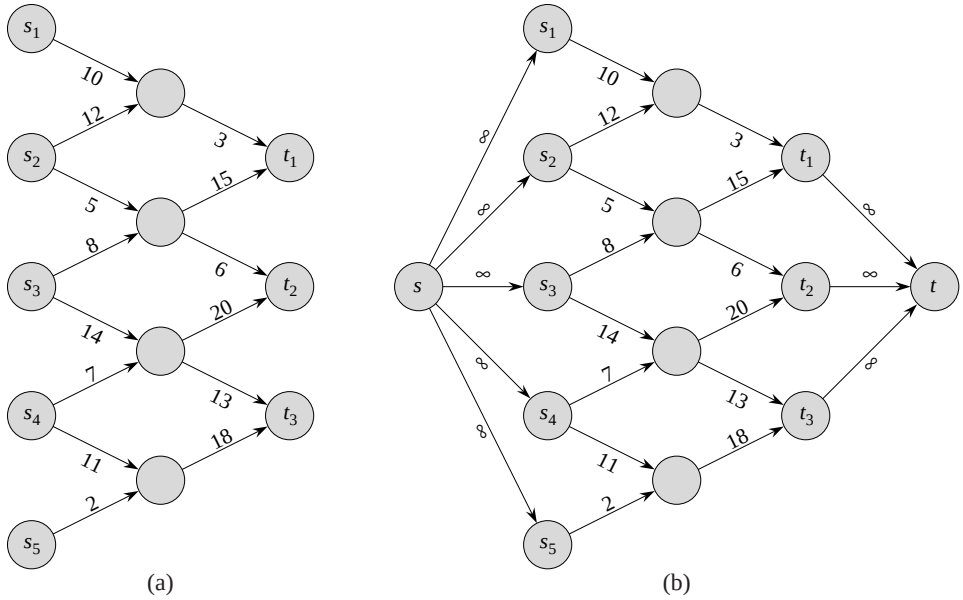
Általában a *kitörlés* lehetőséget ad arra, hogy a városok közötti szállításokat folyamként fogalmazzuk meg, ahol két pont között legfeljebb az egyik irányba pozitív a folyam. Tehát egy olyan helyzet, amelyben két város között mind a két irányba szállítunk jégkorongokat, megfogalmazható egy lényegét tekintve ekvivalens szállítási szituációval, amelyben tényleges szállítás csak az egyik irányba történik: a pozitív folyamérték irányába.

Ha adott egy f folyam, amely tényleges fizikai szállítások által keletkező folyam, akkor a konkrét szállításokat nem tudjuk kitalálni. Ha tudjuk, hogy $f(u, v) = 5$, akkor az jelentheti azt, hogy 5 egységnyi árut szállítottunk u-ból v-be, de azt is, hogy 8 egységet szállítottunk u-ból v-be, és 3-at v-ből u-ba. Általában nem tulajdonítunk jelentőséget annak, hogy maguk a szállítások hogyan történnek, csak a pontok közötti folyamértékeket fogjuk fontosnak tekinteni. Ha a szállítások miként is fontos, akkor egy más modellt kell használnunk, amely nyilvántartja mind a két irányba történő szállításokat.

A kitörlés implicit módon szerepet játszik a fejezetben megfogalmazott algoritmusokban. Tegyük fel, hogy az (u, v) élen a folyam értéke $f(u, v)$. Az algoritmus futása során megtörténhet, hogy a folyam értékét a (v, u) élen egy d mennyiséggel növeljük. Matematikailag ez azt jelenti, hogy $f(u, v)$ d-vel csökken, amit d egységnyi anyag *kitörlésével* nyugtázhatunk az (u, v) élen már folyó anyagból.

Több termelő és több fogyasztó

Előfordulhat, hogy egy hálózatban nemcsak egy, hanem több termelő és fogyasztó van. Például a Hokipakk Vállalatnak lehet m darab gyára: $\{s_1, s_2, \ldots, s_m\}$, és n darab raktára: $\{t_1, t_2, \ldots, t_n\}$, ahogyan az a 26.2(a) ábrán látható. Szerencsére az így adódó probléma nem nehezebb a közönséges maximális folyam feladatnál.



26.2. ábra. Többtermelős, többfogyasztós maximális folyam probléma visszavezetése egytermelős, egyfogyasztós maximális folyam problémára. **(a)** Egy hálózat, melynek öt termelője $S = \{s_1, s_2, s_3, s_4, s_5\}$ és három fogyasztója $T = \{t_1, t_2, t_3\}$ van. **(b)** Az ekvivalens egytermelős, egyfogyasztós hálózat. Egy s szuperfogyasztót veszünk fel, és minden eredeti termelőbe vezetünk belőle egy végtelen kapacitású új élt. Továbbá felveszünk egy t szuperfogyasztót, és minden eredeti fogyasztóból vezetünk bele egy végtelen kapacitású új élt.

A többtermelős, többfogyasztós maximális folyam problémát vissza tudjuk vezetni a közönséges maximális folyam problémára. Adjunk ugyanis a hálózatunkhoz egy s ***szupertermelőt,*** és válasszuk a $c(s, s_i)$ kapacitásokat ∞-nek minden $i = 1, 2, \ldots, m$ értékre, továbbá vegyünk fel egy t ***szuperfogyasztót*** is, és a $c(t_j, t)$ kapacitásokat válasszuk szintén ∞-nek minden $j = 1, 2, \ldots, n$ értékre. A 26.2(b) ábra szemlélteti az így előálló új hálózatot. Könnyen látható, hogy az (a)-beli folyamoknak egyértelműen megfeleltethetők a (b)-beli folyamok, és fordítva. Az s termelő éppen annyit termel, amennyi az s_i pontokból továbbfolyik, a t fogyasztó pedig pontosan annyit fogyaszt, amennyi folyam a t_i-kbe beérkezik. A 26.1-3. gyakorlat ezen ekvivalencia pontos bizonyítására vonatkozik.

Néhány egyenlőség

Számos olyan függvénnyel fogunk foglalkozni, amelyek pontpárokon vannak értelmezve. Vezessük be a következő jelölést, ahol az X és Y két ponthalmaz:

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y).$$

Eszerint a megmaradási szabály a következő alakba írható: $f(u, V) = 0$ minden $u \in V - \{s, t\}$ pontra. A rövidség kedvéért el fogjuk hagyni a kapcsos zárójeleket, amikor csak egy elemet tartalmazó halmaz jelölésére szolgálnának, mint azt az előbb is tettük: $\{u\}$ helyett u-t írtunk. (Így például az $f(s, V - s) = f(s, V)$ egyenlőségben $V - s$ a $V - \{s\}$ halmazt jelenti.)

A következő lemma néhány alapvető, folyamokra vonatkozó azonosságot tartalmaz. A bizonyításokat a 26.1-4. gyakorlatban az Olvasóra bízzuk.

26.1. lemma. *Legyen $G = (V, E)$ egy hálózat, f pedig egy G-beli folyam. Ekkor a következők teljesülnek:*

1. *Minden $X \subseteq V$ részhalmazra, $f(X, X) = 0$.*
2. *Minden $X, Y \subseteq V$ részhalmazra, $f(X, Y) = -f(Y, X)$.*
3. *Minden $X, Y, Z \subseteq V$ részhalmazra, amelyre $X \cap Y = \emptyset$, teljesül $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ és $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.*

A most bevezetett jelölés hasznosságának érzékeltetésére gyakorlásképp bebizonyítjuk, hogy a folyam nagysága megegyezik a fogyasztóba beáramló folyammal, azaz

$$\|f\| = f(V, t). \tag{26.3}$$

Ennek nyilván igaznak kell lennie, hiszen intuitívan világos, hogy a megmaradási szabály szerint a termelőtől és a fogyasztótól különböző pontokban a termelőből kiindult folyam nem tud elnyelődni, így kénytelen a fogyasztó levezetni a teljes folyamot. A formális bizonyítás a következő:

$$\begin{aligned}
\|f\| &= f(s, V) && \text{(a definíció szerint)}\\
&= f(V, V) - f(V - s, V) && \text{(a 26.1. lemma 3. része szerint)}\\
&= -f(V - s, V) && \text{(a 26.1. lemma 1. része szerint)}\\
&= f(V, V - s) && \text{(a 26.1. lemma 2. része szerint)}\\
&= f(V, t) + f(V, V - s - t) && \text{(a 26.1. lemma 3. része szerint)}\\
&= f(V, t) && \text{(a megmaradási szabály szerint)}
\end{aligned}$$

Ezt az eredményt később általánosabban is bebizonyítjuk (26.5. lemma).

Gyakorlatok

26.1-1. A folyam definícióját használva bizonyítsuk be, hogy ha $(u, v) \notin E$ és $(v, u) \notin E$, akkor $f(u, v) = f(v, u) = 0$.

26.1-2. Bizonyítsuk be, hogy tetszőleges, a termelőtől és a fogyasztótól különböző v pont esetén a v-be bemenő folyam nagysága megegyezik a v-ből kimenő folyam nagyságával.

26.1-3. Fogalmazzuk meg a hálózat és a folyam definícióját a többtermelős, többfogyasztós esetben. Mutassuk meg, hogy egy többtermelős, többfogyasztós hálózati folyam egyértelműen megfeleltethető azonos folyamértékekkel a szupertermelővel és szuperfogyasztóval kiegészített egytermelős és egyfogyasztós hálózat egy folyamának és fordítva.

26.1-4. Bizonyítsuk be a 26.1. lemmát.

26.1-5. A 26.1(b) ábra hálózati folyamában keressünk olyan $X, Y \subseteq V$ ponthalmazokat, amelyekre $f(X, Y) = -f(V - X, Y)$. Majd keressünk olyan $X, Y \subseteq V$ ponthalmazokat, amelyekre $f(X, Y) \neq -f(V - X, Y)$.

26.1-6. Adott a $G = (V, E)$ hálózat, és az f_1 és f_2 $V \times V$-ből **R**-be képező függvény. Az $f_1 + f_2 : V \times V \to \mathbf{R}$ ***összeg*** a következő:

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v), \tag{26.4}$$

minden $u, v \in V$-re. Ha f_1 és f_2 két G-beli folyam, akkor a három folyamtulajdonság közül melyek teljesülnek az $f_1 + f_2$ függvényre, és melyik sérülhet meg esetleg?

26.1-7. Legyen f egy hálózati folyam valamely hálózatban, α pedig egy valós szám. Az f folyam α ***skalárral való szorzatán*** a következő $V \times V$-ből **R**-be képező függvényt értjük:

$$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$

Bizonyítsuk be, hogy egy hálózat folyamai ***konvex halmazt*** alkotnak, azaz ha f_1 és f_2 folyam, akkor minden $0 \leq \alpha \leq 1$ esetén $\alpha f_1 + (1 - \alpha) f_2$ is folyam.

26.1-8. Fogalmazzuk meg a maximális folyam problémát lineáris programozási feladatként.

26.1-9. Adam professzornak két gyermeke van, akik sajnálatos módon ki nem állhatják egymást, ha az iskolába menetről van szó. A probléma olyan súlyossá fajult, hogy nemcsak nem hajlandóak együtt elsétálni az iskolába, hanem azt is megtagadják, hogy azon háztömb mellett elhaladjanak, amely mellett a másikuk aznap elhaladt. A gyermekek azonban nem látnak abban kivetnivalót, ha útjuk egy sarkon keresztezi a másikét. A sors kegyessége a professzorék házát és az iskolát is egy-egy háztömb sarkára helyezte. A professzor azonban nem tudja, hogy vajon járathatja-e két gyermekét egyazon iskolába. Fogalmazzuk meg a professzor útkeresési problémáját maximális folyam feladatként, ha azt akarja eldönteni, hogy mégis járathatja-e közös iskolába két elvetemedett gyermekét.

26.2. Ford és Fulkerson algoritmusa

Ebben az alfejezetben megismerkedünk Fordnak és Fulkersonnak a maximális folyam problémát megoldó algoritmusával. (Jogos lenne „algoritmus” helyett a „módszer” elnevezés, mivel az alapötletnek különféle megvalósításai ismertek más és más futási időt eredményezve.) A Ford–Fulkerson-algoritmus három fő fogalmat használ, amelyek más folyamalgoritmusoknál is döntő szerepet játszanak, ez a három fogalom a reziduális hálózat, a javítóút és a vágás. Ezek segítségével nyer majd bizonyítást a folyamok elméletének alapvető tétele, a maximális folyam minimális vágás tétel (elterjedt elnevezése az angolból származó max flow – min cut vagy MFMC tétel), ami megadja egy hálózat maximális folyamának a nagyságát a hálózat vágásainak értékében kifejezve. Az alfejezet lezárásaként a Ford–Fulkerson-algoritmus egy módosított változatát közöljük, és meghatározzuk a futási idejét.

Ford és Fulkerson algoritmusa iteratív lépésekből áll. Kezdőlépésként kiindulunk az azonosan 0 folyamból, azaz $f(u, v) = 0$ minden $u, v \in V$-re. Az iteráció minden egyes lépésében egy „javítóút” mentén növeljük a folyam értékét. A javítóút egyszerűen egy, az s-ből a t-be vezető út, amely mentén növelni tudjuk a folyam értékét. Amikor már nem találunk több javítóutat, az algoritmusunk leáll.

A 26.7. tétel biztosítja majd, hogy ez az eljárás a leállásakor tényleg maximális folyamot ad.

FORD–FULKERSON-MÓDSZER(G, s, t)

1 f legyen az azonosan 0 folyam
2 **while** létezik p javítóút
3 **do** növeljük az f folyamot a p mentén
4 **return** f

A reziduális hálózat

A reziduális hálózat egy adott hálózatra és annak egy folyamára vonatkozóan a hálózat azon éleiből áll, amelyeken a folyam értéke növelhető. Formálisan: legyen adott a $G = (V, E)$ hálózat az s termelővel és t fogyasztóval. Legyen f egy G-beli folyam. Tekintsük az $u, v \in V$ pontpárokat, az (u, v) él ***reziduális kapacitása*** legyen a következő:

$$c_f(u, v) = c(u, v) - f(u, v), \tag{26.5}$$

vagyis az az érték, amellyel még megnövelhető a folyam az adott élen a kapacitási feltétel megsértése nélkül. Például ha $c(u, v) = 16$, $f(u, v) = 11$, akkor a $c_f(u, v) = 5$ értékkel még növelhetnénk u-ból v-be a folyamot. Amikor az $f(u, v)$ folyamérték negatív, a $c_f(u, v)$ reziduális kapacitás nagyobb, mint a $c(u, v)$ kapacitás. Például ha $c(u, v) = 16$, $f(u, v) = -4$, akkor a $c_f(u, v)$ reziduális kapacitás 20. A következőképpen képzelhetjük ezt el: a v-ből u-ba 4 egység folyam halad, amit kitörölhetünk 4 egység folyam u-ból v-be küldésével, és ezután az (u, v) él kapacitásának megfelelően 16 egységet még mindig küldhetünk u-ból v-be, vagyis összesen még tényleg 20 egységgel növelhetünk a kapacitási megszorítás megsértése nélkül.

Adott $G = (V, E)$ hálózat és f folyam esetén az f-hez tartozó ***reziduális hálózat*** az a $G_f = (V, E_f)$ hálózat, ahol

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\},$$

és c_f az élekhez tartozó kapacitásfüggvény. Eszerint a reziduális hálózat minden egyes éle (a továbbiakban ezeket ***reziduális éleknek*** nevezzük) pozitív folyamnövelést enged meg a neki megfelelő élen. A 26.3(a) ábra a 26.1(b) ábra f folyamát ismétli meg, a 26.3(b) ábra pedig a megfelelő G_f reziduális hálózatot mutat.

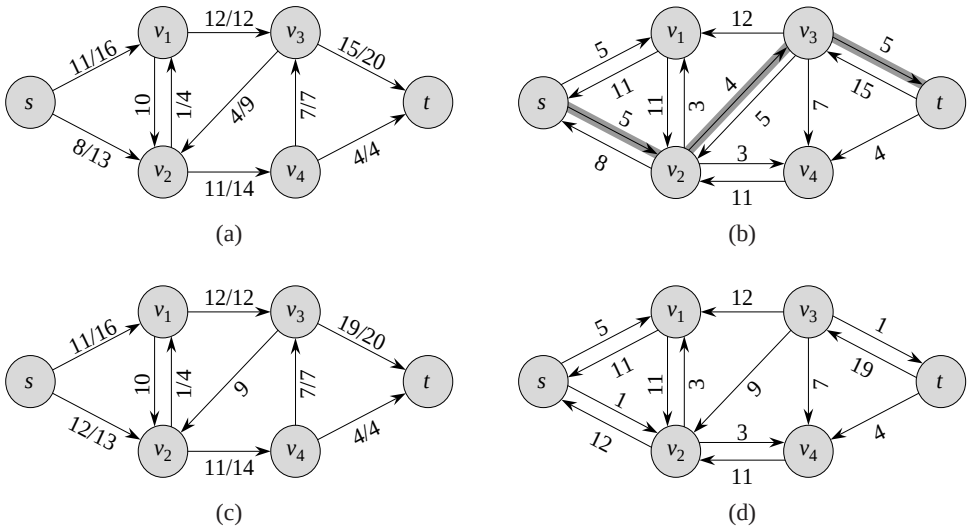
Az E_f egy tetszőleges éle vagy annak fordítottja az E-nek is éle. Ha $f(u, v) < c(u, v)$ teljesül az $(u, v) \in E$ élre, akkor $c_f(u, v) = c(u, v) - f(u, v) > 0$ és $(u, v) \in E_f$. Ha $f(u, v) > 0$ teljesül az $(u, v) \in E$ élre, akkor $f(v, u) < 0$. Ekkor $c_f(v, u) = c(v, u) - f(v, u) > 0$, és így $(v, u) \in E_f$. Amennyiben sem (u, v), sem (v, u) nem éle az eredeti hálózatnak, akkor $c(u, v) = c(v, u) = 0$, $f(u, v) = f(v, u) = 0$ (a 26.1-1. gyakorlat szerint), és $c_f(u, v) = c_f(v, u) = 0$. Tehát az (u, v) él akkor és csak akkor szerepelhet a reziduális hálózatban, ha legalább az (u, v) és (v, u) élek egyike az eredeti hálózatnak is éle, ami szerint

$$|E_f| \leq 2|E|.$$

Figyeljük meg, hogy a G_f reziduális hálózat maga is egy hálózat a c_f reziduális kapacitásokkal. A következő lemma azt mutatja meg, hogy a reziduális hálózat egy folyama miként viszonyul az eredeti hálózat egy folyamához.

26.2. lemma. *Legyen $G = (V, E)$ egy hálózat az s termelővel és a t fogyasztóval, és legyen f egy G-beli folyam. Legyen továbbá G_f az f-hez tartozó reziduális hálózat, és f' a G_f egy folyama. Ekkor az $f + f'$ összeg G-beli folyam, amelynek a nagysága: $\|f + f'\| = \|f\| + \|f'\|$.*

Bizonyítás. Ellenőriznünk kell, hogy teljesül-e az összegfüggvényre a ferde szimmetria, a kapacitási megszorítás és a megmaradási szabály. A ferde szimmetria $u, v \in V$-re:



26.3. ábra. (a) A 26.1(b) ábra G hálózata és f folyama. **(b)** A G_f reziduális hálózat. A p javítóutat szürkével jelöltük. p reziduális kapacitása: $c_f(p) = c_f(v_2, v_3) = 4$. **(c)** Ezt a folyamot kapjuk, ha p mentén 4 egységnyivel növeljük a folyamértéket. **(d)** A (c)-beli folyam reziduális hálózata.

$$\begin{aligned}(f + f')(u,v) &= f(u,v) + f'(u,v) \\ &= -f(v,u) - f'(v,u) \\ &= -(f(v,u) + f'(v,u)) \\ &= -(f + f')(v,u).\end{aligned}$$

A kapacitási megszorítás (felhasználjuk, hogy minden $u, v \in V$-re $f'(u,v) \leq c_f(u,v)$, és még a (26.5) egyenlőséget is):

$$\begin{aligned}(f + f')(u,v) &= f(u,v) + f'(u,v) \\ &\leq f(u,v) + (c(u,v) - f(u,v)) \\ &= c(u,v).\end{aligned}$$

A megmaradási szabály minden $u \in V - \{s,t\}$-re:

$$\begin{aligned}\sum_{v \in V}(f + f')(u,v) &= \sum_{v \in V}(f(u,v) + f'(u,v)) \\ &= \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) \\ &= 0 + 0 \\ &= 0.\end{aligned}$$

Végül az új folyam nagysága:

$$\|f + f'\| = \sum_{v \in V}(f + f')(s,v)$$

$$\begin{aligned} &= \sum_{v\in V}(f(s,v)+f'(s,v)) \\ &= \sum_{v\in V} f(s,v)+\sum_{v\in V} f'(s,v) \\ &= \|f\|+\|f'\|. \end{aligned}$$

■

Javítóutak

Adott $G=(V,E)$ hálózatra és f folyamra a G_f-beli s-ből t-be vezető utakat ***javítóútnak*** nevezzük. A reziduális hálózat definíciója szerint egy javítóút minden éle megenged valamekkora folyamnövelést az adott élen.

A 26.3(b) ábra szürkével jelzett útja javítóút. Ezen út mentén a reziduális hálózatban el tudunk 4 egység folyamot juttatni s-ből t-be, mivel a javítóút mentén a legkisebb reziduális kapacitás: $c_f(v_2,v_3)=4$. A p javítóút ***reziduális kapacitásának*** nevezzük azt a maximális értéket, amekkora folyam rajta átvezethető a kapacitási megszorítások megsértése nélkül, azaz

$$c_f(p)=\min\{c_f(u,v):(u,v)\ p\text{-beli él}\}.$$

A következő lemma, melynek a bizonyítása a 26.2-7. feladatban az Olvasóra marad, a fenti okoskodás matematikai megfogalmazása.

26.3. lemma. *Legyen $G=(V,E)$ egy hálózat, és f egy G-beli folyam, p pedig egy javítóút. Az $f_p: V\times V\to\mathbf{R}$ függvény legyen a következő:*

$$f_p(u,v)=\begin{cases} c_f(p), & \text{ha } (u,v) \text{ rajta van } p\text{-n},\\ -c_f(p), & \text{ha } (v,u) \text{ rajta van } p\text{-n},\\ 0 & \text{különben.}\end{cases} \tag{26.6}$$

Ekkor f_p egy G_f-beli folyam, melynek nagysága $\|f_p\|=c_f(p)>0$.

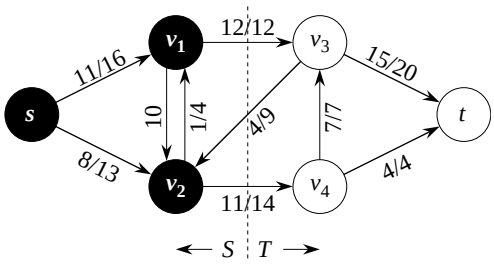
Az alábbi következmény szerint az f_p folyamot f-hez hozzáadva egy f-nél nagyobb nagyságú G-beli folyamot kapunk. A 26.4(c) ábra mutatja a 26.4(b)-beli f_p folyam és a 26.4(a)-beli f folyam összeadásával keletkező folyamot.

26.4. következmény. *Legyen $G=(V,E)$ egy hálózat, és f egy G-beli folyam, p pedig egy javítóút. Az f_p függvény legyen a* (26.6) *egyenlőség szerint definiálva. Legyen $f': V\times V\to\mathbf{R}$ a következő: $f'=f+f_p$. Ekkor f' egy G-beli folyam, amelynek nagysága: $\|f'\|=\|f\|+\|f_p\|>\|f\|$.*

Bizonyítás. A 26.2. és a 26.3. lemmából azonnal adódik. ■

Hálózati folyamok vágásai

A Ford–Fulkerson-módszer minden egyes lépésében növeli a folyamértéket a javítóutak mentén mindaddig, amíg a maximális folyamot el nem éri. A maximális folyam minimális vágás tétel szerint, amit nemsokára bizonyítani is fogunk, egy folyam akkor és csak akkor maximális, ha a reziduális hálózat nem tartalmaz javítóutat. A tétel bizonyítása előtt meg kell ismerkednünk a hálózati folyamokra vonatkozó vágás fogalmával.



26.4. ábra. A 26.1(b) ábra hálózatának (S,T) vágása, ahol $S = \{s, v_1, v_2\}$, $T = \{v_3, v_4, t\}$. S pontjait feketével jelöltük, T pontjait fehérrel. A vágáson átáramló folyam nagysága $f(S,T) = 19$, a vágás értéke $c(S,T) = 26$.

A $G = (V,E)$ hálózat (S,T) ***vágásán*** a V halmaz kétrészes partícióját értjük az S és $T = V - S$ halmazba, amelyekre $s \in S$ és $t \in T$ teljesül. (Ez a definíció hasonlít a 23. fejezetben a minimális feszítőfák esetében használt vágás fogalmához, de itt főleg irányított gráfokról beszélünk, míg ott irányítatlanokról volt szó, és itt még megköveteljük az $s \in S$, $t \in T$ feltételek teljesülését is.) Ha f egy folyam, akkor az (S,T) vágáson keresztülfolyó ***folyam nagysága*** $f(S,T)$. A ***vágás értéke*** legyen $c(S,T)$. Egy vágás ***minimális,*** ha nincs nála kisebb értékű vágás.

A 26.1(b) ábra hálózati folyamának $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ vágását mutatja a 26.4. ábra. A vágáson keresztülfolyó folyam:

$$\begin{aligned} f(v_1,v_3) + f(v_2,v_3) + f(v_2,v_4) &= 12 + (-4) + 11 \\ &= 19, \end{aligned}$$

a vágás kapacitása pedig:

$$\begin{aligned} c(v_1,v_3) + c(v_2,v_4) &= 12 + 14 \\ &= 26. \end{aligned}$$

Figyeljük meg, hogy a vágáson átfolyó folyam értékében negatív folyamértékek is szerepet játszanak, azaz a visszafolyó folyam is, míg a vágás kapacitása csak nemnegatív számok összege. Más szóval az (S,T) vágáson átfolyó folyam nagysága az S-ből T-be átmenő pozitív folyamértékek összegéből kivonva a T-ből S-be átmenő pozitív folyamértékeket. Az (S,T) vágás értékét viszont csak az S-ből T-be mutató élek kapacitásösszege adja, a T-ből S-be mutató élek kapacitásai nem játszanak szerepet.

A következő lemma szerint minden vágáson annyi folyam folyik át, amekkora a folyam nagysága.

26.5. lemma. *Legyen f a G hálózat egy folyama, és legyen (S,T) a G egy vágása. Ekkor az (S,T) vágáson átfolyó folyam nagysága: $f(S,T) = \|f\|$.*

Bizonyítás. Mivel a folyammegmaradás miatt ($f(S - s, V) = 0$), így

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) && \text{(a 26.1. lemma 3. része szerint)} \\ &= f(S,V) && \text{(a 26.1. lemma 1. része szerint)} \\ &= f(s,V) + f(S - s,V) && \text{(a 26.1. lemma 3. része szerint)} \\ &= f(s,V) && \text{(mivel } f(S - s,V) = 0\text{)} \\ &= \|f\|. \end{aligned}$$

■

A 26.5. lemma azonnali következménye a (26.3) egyenlőség, amit már korábban is bebizonyítottunk, miszerint a folyam nagysága megegyezik a fogyasztóba befolyó folyammal.

A 26.5. lemma egy másik következménye az alábbi állítás.

26.6. következmény. *A G hálózat tetszőleges folyamának a nagysága nem lehet nagyobb, mint bármely vágásának az értéke.*

Bizonyítás. Legyen (S, T) a G egy vágása, f pedig egy G-beli folyam. A 26.5. lemma és a kapacitási megszorítás szerint

$$\begin{aligned} \|f\| &= f(S,T) \\ &= \sum_{u\in S}\sum_{v\in T} f(u,v) \\ &\le \sum_{u\in S}\sum_{v\in T} c(u,v) \\ &= c(S,T). \end{aligned}$$

■

A 26.6. lemma közvetlen következménye, hogy hálózat maximális folyama legfeljebb akkora, mint a hálózat minimális vágásának a kapacitása. A fontos maximális folyam-minimális vágás tétel szerint, amelyet most kimondunk és bizonyítunk, a maximális folyam értéke egyenlő a minimális vágás kapacitásával.

26.7. tétel (maximális folyam minimális vágás tétel). *Legyen a $G = (V, E)$ hálózat termelője s, fogyasztója t. Továbbá legyen f egy G-beli folyam. Ekkor a következők ekvivalensek.*

1. *f a G egy maximális folyama.*
2. *A G_f reziduális hálózat nem tartalmaz javítóutat.*
3. *$\|f\| = c(S, T)$ teljesül G valamely (S, T) vágására.*

Bizonyítás. (1) $\Rightarrow$ (2): Tegyük fel indirekten, hogy f a G egy maximális folyama, de G_f mégis tartalmaz egy p javítóutat. Ekkor a 26.4. következmény szerint az $f + f_p$ összeg, ahol f_p a (26.6) szerint értendő, határozottan nagyobb nagyságú folyam, mint f, ami ellentmond f maximalitásának.

(2) $\Rightarrow$ (3): Tegyük fel, hogy G_f-ben nincs javítóút, azaz G_f-ben nincs s-ből t-be vezető út. Legyen S a következő halmaz

$$S = \{v \in V : G_f\text{-ben létezik } s\text{-ből } v\text{-be út}\},$$

és $T = V - S$. (S, T) nyilván vágás, mivel G_f-ben nem vezet út s-ből t-be. Minden $u \in S$, $v \in T$ párra $f(u, v) = c(u, v)$, különben $(u, v) \in E_f$ teljesülne, amiből $v \in S$ következne. A 26.5. lemma szerint tehát $\|f\| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): A 26.6. következmény szerint $\|f\| \le c(S, T)$ igaz minden (S, T) vágásra. $\|f\| = c(S, T)$ szerint tehát f egy maximális folyam. ■

A Ford–Fulkerson-algoritmus

A Ford–Fulkerson-algoritmus iteratív lépésében egy tetszőleges p javítóutat keresünk, és annak mentén növeljük az f folyamot a $c_f(p)$ reziduális kapacitással. A módszer alább

következő megvalósítása meghatározza a maximális folyamot a $G = (V, E)$ hálózatban az u és v pont közötti $f[u, v]$ aktuális folyamértékeket változtatgatva.[1] Ha u és v között nincs él egyik irányba sem, akkor $f[u, v] = 0$. A $c(u, v)$ kapacitásokat a gráffal együtt adottnak tekintjük, és $c(u, v) = 0$, ha $(u, v) \notin E$. A $c_f(u, v)$ reziduális kapacitás a (26.5) képlet szerint számolható ki. A $c_f(p)$ változó az aktuális p javítóút reziduális kapacitásának a tárolására szolgál.

FORD-FULKERSON(G, s, t)

```
1  for minden (u, v) ∈ E[G]-re
2      do f[u, v] ← 0
3         f[v, u] ← 0
4  while létezik G_f-ben s-ből t-be egy p út
5      do c_f(p) ← min{c_f(u, v) : (u, v) rajta van p-n}
6      for minden p-n lévő (u, v) élre
7      do f[u, v] ← f[u, v] + c_f(p)
8         f[v, u] ← −f[u, v]
```

A FORD-FULKERSON a korábban megadott FORD-FULKERSON-MÓDSZER részletesebben kidolgozott változata. A 26.5. ábrán az algoritmus egy futását szemléltetjük. Az algoritmus 1–3. sora az induló f folyam azonosan 0-ra való beállítására szolgál. A 4–8. sor **while** ciklusa minden egyes lépésében egy p javítóutat keres a G_f gráfban, és annak $c_f(p)$ reziduális kapacitásával növeli p mentén az f folyamot. Amikor nem talál javítóutat, akkor – amint azt láttuk – a megtalált f folyam maximális.

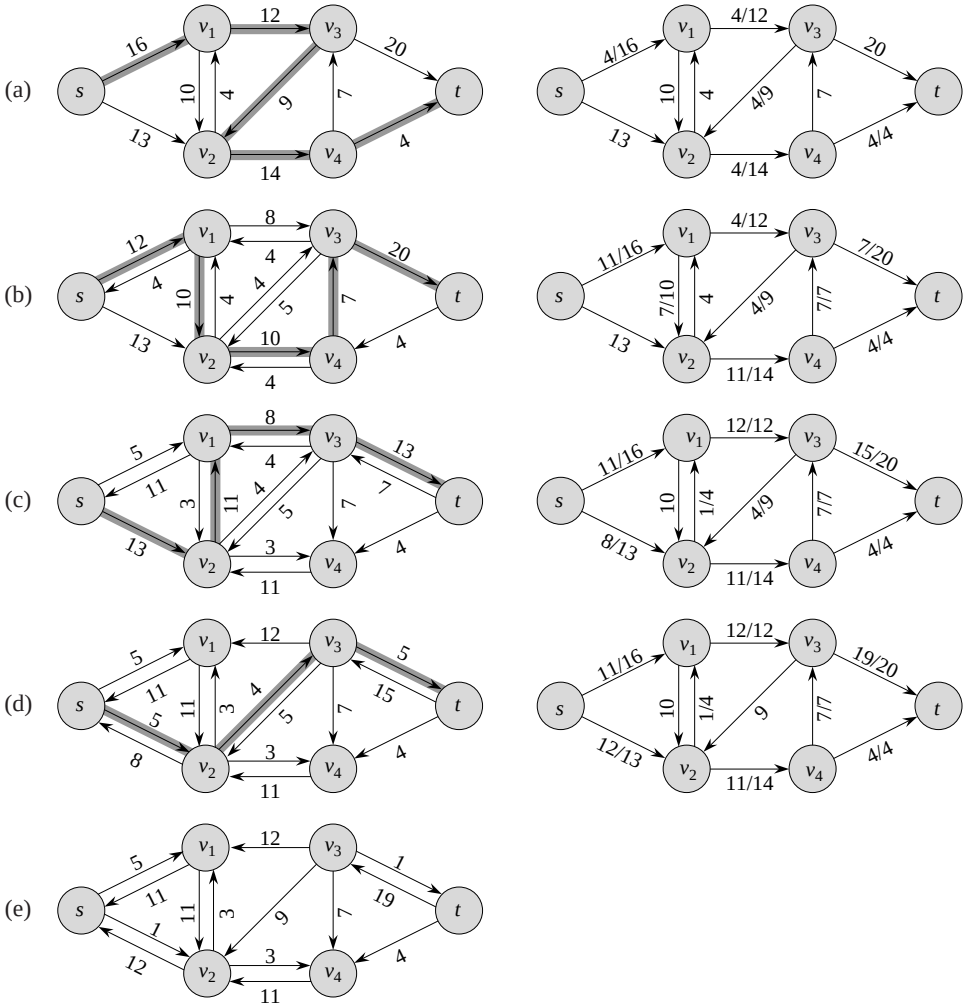
A Ford–Fulkerson-algoritmus futási ideje

A Ford–Fulkerson-algoritmus futási ideje nagyban függ attól, hogy a 4. sorban a p javítóutat hogyan keressük meg. Ha csak találomra választunk egy javítóutat, el őfordulhat, hogy az algoritmus sosem áll meg: a folyamérték folyamatosan növekszik, ám nem feltétlenül a maximális folyamnagysághoz konvergál.[2] Azonban ha az utat SZK-algoritmussal, azaz szélességi kereséssel (22.2. alfejezet) választjuk, akkor az algoritmus polinomiális id őben véget ér. Mielőtt ezt bebizonyítanánk, egy egyszerű meggondolással fels ő korlátot adunk a lépésszámra abban az esetben, amikor a javítóút választása ugyan tetsz őleges, de az élek kapacitásai mind egészek.

A gyakorlatban legtöbbször csakis egész kapacitásokkal találkozhatunk a maximális folyam problémával. Ha a kapacitások racionális számok, akkor a közös nevez őjükkel való felszorzás után egészeket kapunk. Ezen feltevések mellett a Ford–Fulkerson-algoritmus futási ideje $O(E\|f^*\|)$, ahol $\|f^*\|$ az algoritmus által megtalált maximális folyam nagysága. A következőképp láthatjuk ezt be: az 1–3. sor $\Theta(E)$ id őt igényel, a 4–8. sor **while** ciklusa legfeljebb $\|f^*\|$-szor kerül végrehajtásra, mivel minden javítóút reziduális kapacitása az egészértékűség miatt legalább 1.

A **while** ciklus lefutása hatékonnyá tehető, amennyiben a $G = (V, E)$ hálózatot megfelelő adatszerkezettel tároljuk. Tegyük fel, hogy a $G' = (V', E')$ irányított gráfhoz tartozó

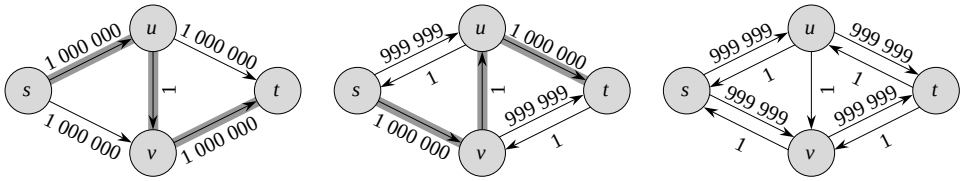
[1] A továbbiakban szögletes zárójellel jelezzük, ha egy program változójáról van szó, hagyományos zárójelet pedig továbbra is a függvények esetében használunk.
[2] Ez csak akkor fordulhat elő, ha irracionális élkapacitások is vannak.



26.5. ábra. A Ford–Fulkerson-algoritmus egy lefutása. **(a)–(d)** A **while** ciklus egymást követő lépései. A bal oldali gráfok a G_f reziduális gráfok, a program 4. sorában felhasznált p javítóutat szürkével emeltük ki. A jobb oldali gráfok a megtalált javítóút mentén való növelés után adódó folyamot mutatják. Az (a) ábra reziduális hálózata a $G = (V, E)$ hálózattal azonos. **(e)** A **while** ciklus utolsó reziduális hálózata. Ez már nem tartalmaz javítóutat, vagyis a (d) ábra f folyama maximális folyam.

adatstruktúránk adott, ahol $E' = \{(u, v) : (u, v) \in E \text{ vagy } (v, u) \in E\}$. A G hálózat élei mind a G élei közül valók, így az adott adatstruktúrával együtt nem okoz gondot a kapacitások és folyamértékek tárolása. A G-beli f folyamhoz tartozó G_f reziduális hálózat a G' azon (u, v) éleiből áll, melyekre $c(u, v) - f[u, v] \neq 0$. A reziduális hálózatban utat keresni tehát $O(V + E') = O(E)$, akár mélységi keresést, akár szélességi keresést alkalmazunk. A **while** ciklus minden egyes lefutása tehát $O(E)$ időt igényel, tehát a Ford–Fulkerson-algoritmus futási ideje $O(E\|f^*\|)$.

Amikor a kapacitások egészek, és az $\|f^*\|$ optimális folyamérték kicsi, akkor a Ford–Fulkerson-algoritmus futási ideje egészen jó. A 26.6(b) ábra egy olyan hálózatot ábrázol,



26.6. ábra. (a) Egy hálózat, amelyen a Ford–Fulkerson-algoritmus $\Theta(E\|f^*\|)$ lépést tesz meg, ahol f^* egy maximális folyam. $\|f^*\| = 2\,000\,000$. A szürkével jelzett egységnyi értékű javítóút mentén növelünk. **(b)** A kapott reziduális hálózat. Egy újabb egységnyi értékű javítóúttal. **(c)** A folyamnövelés utáni reziduális hálózat.

ami ugyan nagyon egyszerű, mégis a nagy $\|f^*\|$ érték miatt az algoritmus esetleg nagyon sok lépést kénytelen megtenni. Könnyen látható, hogy ebben a hálózatban a maximális folyamnagyság 2 000 000: a folyam 1 000 000 egysége a az $s \to u \to t$ úton, és 1 000 000 egysége az $s \to v \to t$ úton jut el t-be. Ha a Ford–Fulkerson-algoritmus által talált első javítóút az $s \to u \to v \to t$ (lásd az (a)-t), akkor a folyam nagysága a ciklus első lefutása után 1 lesz. Az ezután adódó reziduális hálózatot a 26.6(b) ábra mutatja. Ha a második lefutáskor az $s \to v \to u \to t$ javítóutat találja meg az algoritmus (lásd a (b)-t), akkor az új folyam nagysága 2 lesz. Az ezután adódó reziduális hálózatot a 26.6(c) ábra mutatja. Ha az algoritmus a lefutásakor ezen két javítóút mentén felváltva növeli a folyamot – minden egyes lépésben 1-gyel –, akkor a maximális folyam megtalálásáig 2 000 000 lépést tesz meg.

Az Edmonds–Karp-algoritmus

A Ford–Fulkerson-algoritmus lépésszáma a bemenet hosszának polinomjával lesz felülről becsülhető, ha mindig az egyik *legrövidebb* javítóút mentén javítunk, azaz olyan út mentén, amely a reziduális hálózatban élszámot tekintve legrövidebb az s-ből t-be vezető utak között. Legrövidebb javítóutat találunk, ha a program 4-edik sorában például SZK-algoritmussal keressük azt. Az így kapott algoritmust ***Edmonds–Karp-algoritmusnak*** nevezzük, mivel Edmonds és Karp javasolta először a javítóutak ilyen választását. Most bebizonyítjuk, hogy az Edmonds–Karp-algoritmus futási ideje $O(VE^2)$.

A bizonyításban fontos szerepet játszanak a pontok egymástól való távolsága a reziduális hálózatban. A továbbiakban $\delta_f(u, v)$ jelöli a G_f reziduális hálózatban az u-ból v-be vezető legrövidebb irányított út hosszát.

26.8. lemma. *Legyen $G = (V, E)$ egy hálózat az s termelővel és t fogyasztóval. Az Edmonds–Karp-algoritmus lefutása során minden $v \in V - \{s, t\}$ pontra a $\delta_f(s, v)$ érték minden egyes folyamnöveléssel monoton nő.*

Bizonyítás. Indirekten tegyük fel, hogy létezik olyan $v \in V - \{s, t\}$ pont, amelyre valamelyik folyamnövelés során $\delta_f(s, v)$ csökken. Legyen f az első ilyen folyamnövelés előtti folyam, f' pedig a folyamnöveléssel kapott folyam. Legyen v egy olyan pont, amelynek minimális a $\delta_{f'}(s, v)$ értéke azon pontok között, amelyeknek csökkent az s-től való távolságuk, azaz $\delta_{f'}(s, v) < \delta_f(s, v)$. Legyen $p = s \rightsquigarrow u \to v$ a $G_{f'}$-ben egy legrövidebb s-ből v-be vezető út, amelynek (u, v) az utolsó éle, ekkor

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1. \tag{26.7}$$

A v választása miatt tudjuk, hogy az u pont távolsága nem csökkent, azaz

$$\delta_{f'}(s,u) \geq \delta_f(s,u). \tag{26.8}$$

Azt állítjuk, hogy $(u,v) \notin E_f$. Miért igaz ez? Ha $(u,v) \in E_f$ teljesülne, akkor

$$\begin{aligned}
\delta_f(s,v) &\leq \delta_f(s,u)+1 && \text{(a 24.10. lemma és a háromszög-egyenlőtlenség szerint)}\\
&\leq \delta_{f'}(s,u)+1 && \text{(a (26.8) egyenlőtlenség szerint)}\\
&= \delta_{f'}(s,v), && \text{(a (26.7) egyenlőtlenség szerint),}
\end{aligned}$$

ami ellentmond a $\delta_{f'}(s,v) < \delta_f(s,v)$ feltevésnek.

Mit jelent az, hogy $(u,v) \notin E_f$ és $(u,v) \in E_{f'}$? A folyamnövelésnek növelnie kellett a folyamot a v-ből u-ba. Mivel az Edmonds–Karp-algoritmus mindig egy legrövidebb út mentén növeli a folyamot, ezért G_f-ben az s-ből u-ba vezető legrövidebb útnak a (v,u) az utolsó éle. Tehát

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u)-1 &&\\
&\leq \delta_{f'}(s,u)-1 && \text{(a (26.8) egyenlőtlenség szerint)}\\
&= \delta_{f'}(s,v)-2 && \text{(a (26.7) egyenlőtlenség szerint),}
\end{aligned}$$

ami ellentmond a $\delta_{f'}(s,v) < \delta_f(s,v)$ feltevésnek. Következésképpen az indirekt feltevés, miszerint ilyen v létezik, ellentmondásra vezet. ■

A következő tétel az Edmonds–Karp-algoritmus iterációszámára ad felső becslést.

26.9. tétel. *Legyen $G=(V,E)$ egy hálózat az s termelővel és t fogyasztóval. Az Edmonds–Karp-algoritmus lefutása során $O(VE)$ darab folyamnövelés történik.*

Bizonyítás. A G_f reziduális hálózat egy p javítóútjának az (u,v) éle ***kritikus,*** ha $c_f(p) = c_f(u,v)$. Miután növeltük a folyamot egy javítóút mentén, a javítóút minden kritikus éle kikerül a reziduális hálózatból. Világos, hogy minden javítóúton van legalább egy kritikus él. Megmutatjuk, hogy az $|E|$ darab él mindegyike legfeljebb $|V|/2-1$ alkalommal lehet kritikus él.

Legyen u és v a hálózat két pontja, amelyek között van él. Mivel a javítóutak legrövidebb utak, amikor az (u,v) él először kritikus,

$$\delta_f(s,v) = \delta_f(s,u)+1.$$

Amikor a folyamnövelés megtörténik, (u,v) kikerül a reziduális hálózatból, és nem is jelenhet meg egy másik javítóúton mindaddig, amíg az u-ból v-be menő folyam értéke nem csökken, ami viszont akkor következhet be, ha (v,u) rajta van egy javítóúton. Ha f' a G-nek a folyama, amikor ez megtörténik, akkor

$$\delta_{f'}(s,u) = \delta_{f'}(s,v)+1.$$

Mivel $\delta_f(s,v) \leq \delta_{f'}(s,v)$, ezért

$$\begin{aligned}
\delta_{f'}(s,u) &= \delta_{f'}(s,v)+1\\
&\geq \delta_f(s,v)+1\\
&= \delta_f(s,u)+2.
\end{aligned}$$

Következésképpen az (u, v) él kétszeri kritikussá válása között az u-ba vezető legrövidebb s-ből induló út hossza (azaz az s-től való távolsága) legalább 2-vel nőtt. Kezdetben az u s-től való távolsága legalább 0. Az s-ből u-ba vezető legrövidebb utak nem tartalmazhatják az s, u és t pontokat (mivel az, hogy (u, v) rajta van egy javítóúton, azt jelenti, hogy $u \neq t$). Tehát ha valamikor nem vezet út az s-ből az u-ba a reziduális hálózatban, akkor a távolsága a megelőző reziduális hálózatban legfeljebb $|V| - 2$, ami azt jelenti, hogy az (u, v) legfeljebb $(|V| - 2)/2 = |V|/2 - 1$ alkalommal válhat kritikussá. Mivel $O(E)$ él szerepelhet a reziduális gráfban, ezért a kritikus élek összszáma az Edmonds–Karp-algoritmus lefutása alatt $O(VE)$. Mivel minden javítóúton van legalább egy kritikus él, a tételünket be is láttuk. ■

Szélességi kereséssel $O(E)$ idő alatt meg tudunk keresni egy legrövidebb javítóutat, ami azt jelenti, hogy az Edmonds–Karp-algoritmus futási ideje $O(VE^2)$. Látni fogjuk, hogy az előfolyam-algoritmusok ennél jobb futási időt szolgáltatnak. A 26.4. alfejezetben megadott algoritmus $O(V^2E)$ futási idejű, amit aztán a 26.5. alfejezetben tovább javítunk $O(V^3)$-re.

Gyakorlatok

26.2-1. A 26.1(b) ábrán mekkora az $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$ vágáson áthaladó folyam? Mekkora ezen vágás értéke?

26.2-2. Futtassuk le az Edmonds–Karp-algoritmust a 26.1(a) ábra hálózatán.

26.2-3. A 26.5. ábra példáján melyik a megtalált maximális folyamhoz tartozó minimális vágás? A javítóutak közül melyik kettő csökkenti a korábbi folyamot valamelyik élen?

26.2-4. Bizonyítsuk be, hogy tetszőleges c kapacitásfüggvény és f folyam esetén bármely u-ra és v-re $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

26.2-5. A 26.1. alfejezetben megadott konstrukció szerint a többtermelős, többfogyasztós folyam problémát visszavezetjük egytermelős, egyfogyasztós folyam problémára. Bizonyítsuk be, hogy ekkor a kapott hálózat maximális folyamának nagysága véges szám, ha az eredeti többtermelős, többfogyasztós hálózat minden élének kapacitása véges.

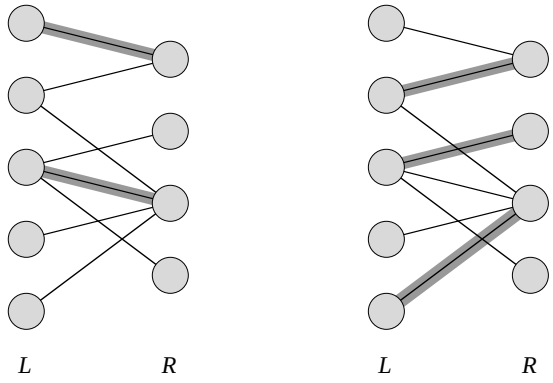
26.2-6. Tegyük fel, hogy egy többtermelős, többfogyasztós hálózatban az s_i termelőből p_i folyam indul ki, azaz $f(s_i, V) = p_i$, továbbá a t_j fogyasztó igénye q_j, azaz $f(V, t_j) = q_j$, és $\sum_i p_i = \sum_j q_j$. Keressünk olyan f folyamot, ami kielégíti az előző feltételeket. Mutassuk meg, hogyan vezethető vissza a fenti feltételeket kielégítő f folyam meghatározása egy egytermelős, egyfogyasztós hálózat maximális folyamának keresésére.

26.2-7. Bizonyítsuk be a 26.3. lemmát.

26.2-8. Mutassuk meg, hogy egy $G = (V, E)$ hálózat maximális folyama mindig megtalálható legfeljebb $|E|$ darab javítóút menti folyamnövelés után. *Útmutatás.* Az utakat *azután* határozzuk meg, hogy megtaláltuk a maximális folyamot.

26.2-9. Egy irányítatlan gráf ***élösszefüggőségi száma*** a legkisebb olyan k, amelyre igaz, hogy a gráfból legalább k élet el kell hagyni ahhoz, hogy ne legyen összefügg ő. Például egy fa élösszefüggőségi száma 1, egy kör élösszefüggőségi száma 2.) Adjunk eljárást, amely maximális folyamot megkereső algoritmus segítségével megadja a $G = (V, E)$ irányítatlan gráf élösszefüggőségi számát. Adjunk olyan eljárást is az élösszefüggőségre, amely legfeljebb $|V|$ darab hálózatban keres maximális folyamot, és a hálózatok pontszáma $O(V)$, élszáma $O(E)$.

26.2-10. Tegyük fel, hogy a $G = (V, E)$ hálózatban szimmetrikusak az élek, azaz ha $(u, v) \in E$, akkor $(v, u) \in E$ is teljesül. Mutassuk meg, hogy az Edmonds–Karp-algoritmus legfeljebb $|V||E|/4$ folyamnövelés után megtalálja a maximális folyamot. (*Útmutatás.* Vizs-



26.7. ábra. A $G = (V, E)$ páros gráf az L és R pontosztályokkal. **(a)** Egy kételemű párosítás. **(b)** Egy háromelemű, egyben maximális párosítás.

gáljuk meg egy kritikus (u, v) élre, hogy hogyan változik $\delta(s, u)$ és $\delta(v, t)$, mikorra (u, v) megint kritikus lesz.)

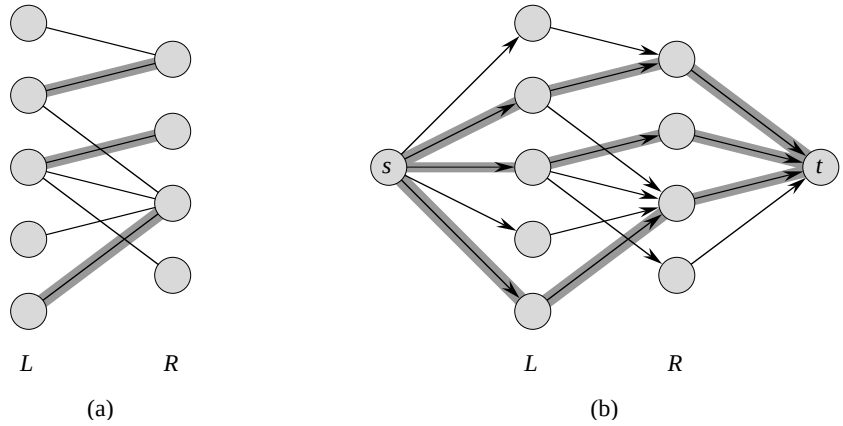
26.3. Maximális párosítás páros gráfban

Számos kombinatorikai problémát könnyen vissza tudunk vezetni maximális folyam problémákra. Ezt tettük a 26.1. alfejezet többtermelős, többfogyasztós maximális folyam problémájával. Más kombinatorikai problémáknak látszólag kevés közük van a hálózati folyamokhoz, mégis visszavezethetők maximális folyam problémákra. Ebben az alfejezetben egy ilyen visszavezetéssel ismerkedünk meg: páros gráf (lásd B.4. alfejezet) maximális párosítását keressük meg folyamok segítségével. Erősen támaszkodni fogunk a javítóutak módszerének egészértékűségi tulajdonságára. Látni fogjuk, hogy a Ford–Fulkerson-algoritmussal egy $G = (V, E)$ páros gráf maximális párosítását $O(VE)$ időben meg tudjuk keresni.

Párosítások

Adott a $G = (V, E)$ irányítatlan gráf. Az $M \subseteq E$ élhalmazt ***párosításnak*** nevezzük, ha minden $v \in V$ pontra legfeljebb egy M-beli él illeszkedik. Azt mondjuk, hogy a $v \in V$ pont ***fedett,*** ha létezik rá illeszkedő M-beli él, máskülönben ***fedetlennek*** nevezzük. Az M párosítást ***maximális párosításnak*** nevezzük, ha élszámra vonatkozóan maximális, azaz ha tetsz őleges M' párosításra $|M| \geq |M'|$. Ebben az alfejezetben páros gráfok maximális párosításait keressük. Páros gráfnak mondunk egy gráfot, ha ponthalmazának létezik egy $V = L \cup R$ partíciója, ahol $L \cap R = \emptyset$, és a gráf minden éle egy L-beli és egy R-beli pont között halad. Feltesszük továbbá, hogy minden $v \in V$ pontra legalább egy él illeszkedik. A 26.7. ábra egy páros gráf két párosítását ábrázolja.

Páros gráfbeli maximális párosítás keresésének nagyon sok gyakorlati alkalmazása van. Például képzeljük el munkagépek egy L halmazát és párhuzamosan végrehajtható feladatok egy R halmazát. Az $u \in L$ gép akkor legyen *összekötve* a $v \in R$ feladattal, ha az u gép



(a) (b)

26.8. ábra. Egy páros gráf és a hozzátartozó segédhálózat. **(a)** A 26.7. ábra páros gráfja. A szürkével jelzett élhalmaz a gráf egy maximális párosítása. **(b)** A G' segédhálózat és egy maximális folyama. Minden élnek a kapacitása 1. A szürke éleken a folyam értéke 1, az összes többi élen pedig 0 a folyamérték. Az L-ből R-be vezető szürke élek a G egy maximális párosítását adják.

el tudja azt látni. Az így kapott páros gráf egy maximális párosítása olyan feladatkiosztást jelent, amelyben a lehető legtöbb gép dolgozik egyszerre.

Maximális párosítás keresése páros gráfban

A Ford–Fulkerson-algoritmus segítségével a $G = (V, E)$ páros gráf maximális párosítását meg tudjuk keresni $O(VE)$ lépésben. A G gráfhoz egy $G' = (V', E')$ segédhálózatot fogunk definiálni, amelynek az egészértékű folyamai megfelelnek G párosításainak, ahogyan azt a 26.8. ábra szemlélteti. A G' hálózat legyen a következő. Vegyünk fel egy új s pontot, ez lesz a G' termelője, és egy új t pontot, ez lesz a fogyasztó. Legyen $V' = V \cup \{s, t\}$. Ha L és R a gráf két pontosztálya, akkor G' irányított élei E-nek az L-ből R-be vezető élei, továbbá $|V|$ új él:

$$\begin{aligned} E' &= \{(s,u) : u \in L\} \\ &\cup \{(u,v) : u \in L, v \in R \text{ és } (u,v) \in E\} \\ &\cup \{(v,t) : v \in R\}. \end{aligned}$$

Továbbá minden E'-beli él kapacitása legyen 1. Mivel a gráf minden csúcsára illeszkedik legalább egy él, $|E| \geq |V|/2$. Tehát $|E| \leq |E'| = |E| + |V| \leq 3|E|$, azaz $|E'| = \Theta(E)$.

Az alábbi lemma szerint a G párosításai egyértelműen megfelelnek a G' egészértékű folyamainak, ahol egy folyamot ***egészértékűnek*** nevezünk, ha az $f(u, v)$ folyamérték egész szám minden $(u, v) \in V \times V$-re.

26.10. lemma. *Legyen $G = (V, E)$ páros gráf a $V = L \cup R$ csúcspartícióval, és $G' = (V', E')$ a hozzátartozó segédhálózat. Ha M egy G-beli párosítás, akkor létezik egy f folyam G'-ben, amelynek nagysága $\|f\| = |M|$. Megfordítva: ha f egy egészértékű folyam G'-ben, akkor G-ben létezik M párosítás, amelynek az elemszáma $|M| = \|f\|$.*

Bizonyítás. Először megmutatjuk, hogy a G egy M párosítása megfelel a G' egy f egészértékű folyamának. Legyen f a következő. Ha $(u,v) \in M$, akkor $f(s,u) = f(u,v) = f(v,t) = 1$ és $f(u,s) = f(v,u) = f(t,v) = -1$. A összes többi $(u,v) \in E'$ élre legyen $f(u,v) = 0$. Egyszerű igazolni, hogy f-re teljesül a ferde szimmetria, a kapacitási megszorítások és a megmaradási szabály.

Látjuk, hogy minden $(u,v) \in M$ él megfelel 1 egységnyi folyamnak a G' hálózatban, amely az $s \to u \to v \to t$ úton halad. Továbbá az M élei által ily módon meghatározott utak pontdiszjunktak az s és t pontok kivételével. Az $(L+s, R+t)$ vágáson áthaladó folyam nagysága $|M|$, tehát a 26.5. lemma szerint az f folyam nagysága $\|f\| = |M|$.

A másik irány bizonyításához legyen f a G' egy egészértékű folyama, és

$$M = \{(u,v) : u \in L, v \in R, f(u,v) > 0\}.$$

Minden $u \in L$ csúcsba egy irányított él mutat, amelynek a kapacitása 1. Tehát minden $u \in L$ pontba legfeljebb 1 a bemenő folyam nagysága, és ha ez az 1 nagyságú folyam valóban beérkezik, akkor a folyam megmaradási szabálya szerint 1 nagyságú folyamnak el is kell hagynia az u-t. Mivel f egészértékű, minden $u \in L$ pontba az 1 nagyságú folyam legfeljebb egy élen érkezhet, és legfeljebb egy élen távozhat. Tehát akkor és csak akkor érkezik pozitív nagyságú folyam az u-ba, ha létezik egyértelműen egy $v \in R$ pont, amelyre $f(u,v) = 1$, és legfeljebb egy u-ból kiinduló él szállít nemnulla (azaz 1 nagyságú) folyamot. Analóg érvelés alkalmazható el minden $v \in R$ pontra. Tehát az M élhalmaz valóban párosítást alkot G-ben.

$|M| = \|f\|$ bizonyításához vegyük észre, hogy a párosítás által fedett minden u pontra $f(s,u) = 1$, és minden $(u,v) \in E - M$ élre $f(u,v) = 0$. Tehát

$$\begin{aligned} |M| &= f(L,R) \\ &= f(L,V') - f(L,L) - f(L,s) - f(L,t) \qquad \text{(a 26.1. lemma szerint).} \end{aligned}$$

A fenti kifejezést jelentősen egyszerűsíthetjük. A folyam megmaradási szabálya szerint $f(L,V') = 0$, a 26.1. lemma szerint $f(L,L) = 0$, a ferde szimmetria szerint pedig $-f(L,s) = f(s,L)$, és mivel L és t között nincsenek élek, ezért $f(L,t) = 0$ következik. Ezek szerint

$$\begin{aligned} |M| &= f(s,L) \\ &= f(s,V') \quad \text{mivel } s \text{ minden kimenő éle } L\text{-be megy} \\ &= \|f\|. \qquad (\|f\| \text{ definíciója miatt)} \end{aligned}$$

■

A 26.10. lemmára építve azt szeretnénk megmutatni, hogy a G páros gráf maximális párosítása megfelel a G' hálózat egy maximális értékű folyamának, és következésképpen a G maximális párosítását meghatározhatjuk a G' egy maximális folyamának a meghatározásával. Az egyetlen nehézség ezek után az, hogy a maximális folyamot meghatározó algoritmusunk esetleg a G' egy olyan folyamát adja, amelynek az $f(u,v)$ értéke valamely élen nem egész annak ellenére, hogy a folyam $\|f\|$ nagysága egész. A következő tétel mutatja, hogy a Ford–Fulkerson-algoritmust használva ez nem fordulhat elő.

26.11. tétel (egészértékűségi tétel). *Ha egy hálózat c kapacitásfüggvénye minden élen egész értéket vesz fel, akkor a Ford–Fulkerson-algoritmus által meghatározott f maximális folyam $\|f\|$ nagysága egész. Továbbá minden u és v pontpárra az $f(u,v)$ folyam érték is egész.*

Bizonyítás. A bizonyítás történhet például az iterációk számára vonatkozó indukcióval. Az Olvasóra bízzuk ennek kidolgozását a 26.3-2. gyakorlatban. ■

Ezek után már bebizonyíthatjuk a 26.10. lemma alábbi következményét.

26.12. következmény. *A G páros gráf egy M maximális párosításának az elemszáma megegyezik a G' segédhálózat egy maximális folyamának a nagyságával.*

Bizonyítás. A 26.10. lemma jelöléseit használjuk. Tegyük fel, hogy az M a G egy maximális párosítása, és az M-nek megfelelő f folyam nem maximális a G' hálózatban. Ekkor létezik egy f' folyam G'-ben, amelyre $\|f'\| > \|f\|$. Mivel minden G'-beli él kapacitása egész szám, ezért a 26.11. tétel szerint feltehetjük, hogy f' is egészértékű. Tehát f' megfelel a G egy M' párosításának, amelyre $|M'| = \|f'\| > \|f\| = |M|$, ami ellentmond annak a feltevésünknek, hogy M egy maximális párosítás. Hasonlóképpen beláthatjuk, hogy ha f a G' egy maximális folyama, akkor a neki megfelelő G-beli párosítás maximális. ■

Ezek szerint ha elkészítjük a G irányítatlan páros gráfhoz a G' segédhálózatot, és lefuttatjuk rajta a Ford–Fulkerson-algoritmust, akkor a kapott f egészértékű maximális folyamból azonnal meg tudjuk határozni a G egy M maximális párosítását. Mivel egy páros gráf maximális párosítása legfeljebb $\min\{L, R\} = O(V)$ méretű, ezért a G'-beli maximális folyam értéke is $O(V)$. A javítóutak reziduális kapacitása mindig 1, következésképpen algoritmusunk $O(VE)$ lépésben megadja egy páros gráf maximális párosítását.

Gyakorlatok

26.3-1. Futtassuk le a Ford–Fulkerson-algoritmust a 26.8(b) ábra hálózatán. Rajzoljuk meg a reziduális hálózatokat is. Számozzuk meg föntről lefelé az L pontjait 1-től 5-ig, az R pontjait pedig 6-tól 9-ig. Minden iterációban válasszuk a lexikografikusan legkisebb javítóutat.

26.3-2. Bizonyítsuk be a 26.11. tételt.

26.3-3. Legyen $G = (V, E)$ páros gráf az L és R pontosztályokkal és G' a neki megfelelő hálózat. Adjunk jó felső korlátot a Ford–Fulkerson-algoritmus lefutása során található javítóutak hosszára.

26.3-4.⋆ Egy párosítást ***teljes párosításnak*** nevezünk, ha a gráf minden pontját fedi. Legyen $G = (V, E)$ irányítatlan páros gráf az L és R pontosztályokkal, amelyre $|L| = |R|$. Az $X \subseteq V$ halmazra legyen $N(X)$ az X-beli pontok ***szomszédsága,*** azaz a szomszédainak a halmaza:

$$N(X) = \{y \in V : \text{létezik } x \in X, \text{amelyre } (x, y) \in E\}.$$

Bizonyítsuk be ***Hall tételét:*** a G páros gráfban akkor és csak akkor létezik teljes párosítás, ha minden $X \subseteq L$ halmazra $|X| \leq |N(X)|$.

26.3-5.⋆ A $G = (V, E)$ páros gráfot ***d-regulárisnak*** mondunk, ha minden pontjának d a foka, azaz a belőle kiinduló élek száma. Igazoljuk, hogy minden d-reguláris páros gráfra $|L| = |R|$ teljesül. Bizonyítsuk be, hogy minden d-reguláris páros gráfnak létezik $|L|$ elemű párosítása. Hivatkozzunk a segédhálózat minimális vágásainak az értékére.

★ 26.4. Előfolyam-algoritmusok

Ebben az alfejezetben megismerkedünk egy újabb, a maximális folyam problémát megoldó módszerrel, az előfolyamokra épülő, úgynevezett előfolyam-módszerrel. A ma ismert leggyorsabb maximális folyam algoritmusok erre a módszerre épülnek. Más egyéb folyamproblémákra vonatkozó hatékony algoritmusoknál is találkozhatunk hasonló elvekkel, mint például a minimális költségű folyam probléma esetében. Ez az alfejezet Goldberg „általános" maximális folyam algoritmusát tartalmazza, aminek egy egyszerű megvalósítása $O(V^2E)$ idejű algoritmust eredményez, ami gyorsabb az $O(VE^2)$ futási idejű Edmonds–Karp-algoritmusnál. A 26.5. alfejezet az általános előfolyam-algoritmus egy módosított változatát tartalmazza, ami még hatékonyabb: $O(V^3)$ lépésszámú.

Az előfolyam-algoritmusok a Ford–Fulkerson-féle javítóutas módszernél sokkal *lokálisabb* jellegűek. Ahelyett, hogy az egész reziduális hálózatot vizsgálnák, egyszerre csak egyetlen pontot és annak a reziduális gráfbeli szomszédait veszik szemügyre. A Ford–Fulkerson-féle módszerrel ellentétben az előfolyam-algoritmusok nem egy folyamot növelgetnek lefutásuk során, hanem a megmaradási szabályt jelentősen enyhítik, és csak egy úgynevezett előfolyamot változtatgatnak. Egy $f : V \times V \to \mathbf{R}$ függvényt ***előfolyamnak*** nevezünk, ha ferdén szimmetrikus, teljesíti a kapacitási megszorításokat, és a megmaradási szabály következő gyengítése igaz rá: $f(V, u) \geq 0$ minden $u \in V - \{s\}$ pontra. Az u pontba beáramló ***feleslegfolyamnak*** nevezzük az alábbi értéket:

$$e(u) = f(V, u). \tag{26.9}$$

Azt mondjuk, hogy az $u \in V - \{s, t\}$ pontnál az előfolyam ***túlcsordul,*** ha $e(u) > 0$.

A továbbiakban először leírjuk vázlatosan az előfolyam-algoritmusok működésének alapelveit, majd megvizsgáljuk azt a két műveletet, amit az ilyen algoritmusok használnak: az előfolyam „pumpálását" és egy pont „megemelését". Végül az általános előfolyam-algoritmust írjuk le, megmutatjuk, hogy csakugyan maximális folyamot ad, és a futási idejét is megbecsüljük.

Az elgondolás

Az előfolyam-algoritmusok működésének elve talán legjobban anyagi folyamokon szemléltetve érthető meg: a $G = (V, E)$ hálózat legyen egy vízvezetékrendszer, azaz meghatározott keresztmetszetű (kapacitású) csövek egy ágbogas rendszere. Ezen elgondolással a Ford–Fulkerson-féle javítóutak módszere úgy működik, hogy egy adott folyamhoz olyan javítóutat keres, amelynek mentén a termelőtől (forrástól) a fogyasztóig (nyelőig) egy elágazás nélküli *patak* küldhető még, és addig keres ilyen utakat, és küld el velük újabb mennyiségeket, ameddig csak tud.

Az előfolyam-algoritmusok más alapelv szerint működnek. Mint eddig is, az irányított élek *irányított* vezetékeknek felelnek meg, amelyekben a víz csak az egyik irányba tud folyni. A pontok, amik a vezetékek csatlakozási pontjai, két érdekes plusz tulajdonsággal rendelkeznek: egyrészt elvezetik a feleslegfolyamukat egy túlcsordulási lefolyón keresztül egy – az adott ponthoz tartozó, végtelen befogadó képességű – gyűjtőmedencébe, másrészt minden egyes pont, azaz vezetékcsatlakozási hely, egy meghatározott magasságban helyezkedik el, és ez a magasság az algoritmus lefutása alatt egyre nő.

A pontok magasságai közvetlen hatással vannak az előfolyam növelésére: egy élen csak lefelé tudunk folyamot *pumpálni*, azaz kikötjük, hogy csak egy magasabban lévő pontból egy alacsonyabban lévő pontba küldhetünk több vizet. Lehetséges, hogy alacsonyabban lévő pontból magasabban lévő pontba pozitív a folyam nagysága, de a *pumpálás* művelet ekkor nem végezhető el azon az élen (felfelé). A termelő magassága végig $|V|$, a fogyasztóé pedig végig 0. Az összes többi pont magassága a kezdeti 0 után fokozatosan növekszik. Az algoritmus az első lépésben annyi folyamot indít a termelőből a fogyasztó felé, amennyi csak lehetséges, azaz minden belőle kiinduló vezetéken elküld (leküld) annyi vizet, amennyi a vezeték kapacitása, tehát az s termelőből az $(s, V - s)$ vágás értéke (kapacitása) mennyiségű folyam indul el. Amikor egy ilyen élen a másik végponthoz ér a víz, azonnal annak a gyűjtőmedencéjébe vezetjük le.

Megtörténhet, hogy az u-ból kiinduló azon vezetékek, amelyek nem telítettek, az u-val azonos szinten lévő pontba vagy magasabban lévő pontba vezetnek. Ebben az esetben ahhoz, hogy az u gyűjtőmedencéjébe kerülő feleslegtől a későbbiekben megszabadulhassunk, az u magasságát meg kell növelnünk, ezt a műveletet nevezzük „megemelésnek". Nevezetesen az u-t eggyel magasabb szintre emeljük, mint azon legalacsonyabban lévő pontok, amelyekbe az u-ból telítetlen csővezeték halad. A *megemelés* után tehát az u-ból legalább egy telítetlen vezeték indul ki lefelé, amelyen aztán pumpálhatunk majd vizet.

Végül az a folyammennyiség, ami el tud jutni a fogyasztóhoz, el is jut. Nem érkezhet több, hiszen a csővezetéken átfolyó előfolyam eleget tesz a kapacitási előírásoknak, egy vágáson keresztülhaladó folyam nem lehet nagyobb, mint a vágás értéke. Ahhoz, hogy az előfolyamot „valódi" folyammá alakítsuk, az algoritmus a túlfolyó pontoknál a medencében felgyülemlő folyamot a magasságok $|V|$ fölé emelésével az s-be küldi vissza. Amint látni fogjuk, a gyűjtőmedencék megüresedésével az előfolyam nemcsak megengedett „valódi" folyammá válik, hanem már egyben maximális folyammá is.

Pumpálás és megemelés

Most megismerkedünk az előfolyam-algoritmusok két műveletével: egy adott pontból egy másikba való folyampumpálással, és egy pont magasságának megemelésével. Az, hogy mikor fogjuk ezen műveleteket alkalmazni, függeni fog a pontok magasságától, amit most formálisan is definiálunk.

Legyen $G = (V, E)$ egy hálózat az s termelővel és t fogyasztóval, f pedig legyen egy előfolyam. Egy $h : V \to \mathbf{N}$ függvényt ***magasságfüggvénynek***[3] nevezünk, ha $h(s) = |V|$, $h(t) = 0$, és

$$h(u) \leq h(v) + 1$$

teljesül minden $(u, v) \in E_f$ esetén. Azonnal adódik a következő lemma.

26.13. lemma. *Legyen $G = (V, E)$ egy hálózat, f egy G-beli előfolyam, h pedig egy magasságfüggvény V-n. Ekkor minden $u, v \in V$ pontpárra igaz, hogy ha $h(u) > h(v) + 1$, akkor az (u, v) él nem szerepel a reziduális hálózatban.*

[3] A szakirodalomban a magasságfüggvényt inkább „távolságfüggvénynek" nevezik, egy pont magasságát pedig „távolságnak". Mi a „magasság" elnevezést használjuk, mivel jobban illeszkedik az algoritmus elgondolását megragadó szemléletünkhöz. A „megemel" kifejezést fenntartjuk azon műveletek számára, amelyek megnövelik egy csúcs magasságát. Egy pont magassága a t fogyasztótól való távolságával van kapcsolatban, amit például a G^T transzponált gráfban való szélességi kereséssel meghatározhatunk.

A pumpálás művelet

A PUMPÁLÁS(u, v) műveletet definíció szerint akkor alkalmazhatjuk, ha u-nál a folyam túlcsordul, $c_f(u, v) > 0$, és $h[u] = h[v] + 1$. Az alábbi program végrehajtja a pumpálást a $G = (V, E)$ hálózat u és v pontja között, azaz megváltoztatja az előfolyam-értékeket a szabályoknak megfelelően. Feltesszük, hogy a c kapacitásfüggvény egy konstans időben számolható függvénnyel van adva, és a reziduális kapacitások konstans időben számolhatók c-ből és f-ből. Az u pontnál a feleslegfolyam nagyságát $e[u]$-ban tároljuk, az u magasságát pedig $h[u]$-ban. A $d_f(u, v)$ segédváltozót az u-ból v-be pumpálandó folyam mennyiségének tárolására használjuk.

PUMPÁLÁS(u, v)

1 ▷ **Akkor alkalmazható,** ha u-nál az előfolyam túlfolyik, $c_f(u, v) > 0$, és $h[u] = h[v] + 1$.
2 ▷ **Ezt csinálja** az algoritmus: u-ból v-be $d_f(u, v) = \min(e[u], c_f(u, v))$ egységnyi folyamot pumpál.
3 $d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$
4 $f[u, v] \leftarrow f[u, v] + d_f(u, v)$
5 $f[v, u] \leftarrow -f[u, v]$
6 $e[u] \leftarrow e[u] - d_f(u, v)$
7 $e[v] \leftarrow e[v] + d_f(u, v)$

A következő történik a PUMPÁLÁS(u, v) lefutása alatt. Mivel csak akkor alkalmazzuk, ha $e[u] > 0$ és $c_f(u, v) > 0$, ezért a $d_f(u, v) = \min(e[u], c_f(u, v))$ érték is nagyobb 0-nál, és ez a $d_f(u, v)$ adja meg azt a mennyiséget, amellyel az irányított (u, v) élen a folyamot megnövelhetjük anélkül, hogy $e[u]$ negatívvá válna, vagy a $c(u, v)$ kapacitást megsértenénk. A 3. sorban történik $d_f(u, v)$ meghatározása, a 4–5. sorokban az új f értékek, a 6–7. sorokban az új e értékek kiszámolása. Ha f korábban előfolyam volt, az is marad.

Figyeljük meg, hogy a PUMPÁLÁS(u, v) közben az u és v magasságától semmi sem függött, bár feltettük, hogy $h[u] = h[v] + 1$, és így a feleslegfolyamot csakis egy szinttel alacsonyabbra vezettük el. A 26.13. lemma szerint reziduális élek nem vezetnek olyan pontok között, amelyek magasságainak a különbsége 1-nél nagyobb lenne, és így nem nyernénk semmit sem, ha megengednénk, hogy akkor is tudjunk folyamot lefelé pumpálni két pont között, ha a magasságok különbsége 1-nél nagyobb.

Telítő pumpálásnak nevezünk egy folyampumpálást, ha az (u, v) él ***telítetté*** válik, azaz $c_f(u, v) = 0$ lesz a művelet után, máskülönben ***nemtelítő pumpálásról*** beszélünk. Emlékeztetünk, hogy ha egy él telített, akkor nem szerepel a reziduális hálózatban. Az alábbi egyszerű lemma egy nemtelítő pumpálás utáni helyzetről szól.

26.14. lemma. *Az u-ból a v-be történő nemtelítő pumpálás után az u pontnál a folyam nem csordul túl.*

Bizonyítás. A pumpálás nemtelítő, ezért a $d_f(u, v)$ értéknek meg kell egyeznie $e[u]$-val. Mivel éppen ennyi folyamot vezetünk el u-tól, ezért az $e[u]$ túlfolyás 0-ra csökken. ■

A megemelés művelet

A MEGEMELÉS(u) műveletet akkor alkalmazhatjuk, ha u-nál a folyam túlcsordul, és a $c_f(u, v)$ reziduális kapacitás csak olyan v-kre pozitív, amelyekre $h[v] \geq h[u]$. Vagyis egy pontot akkor emelhetünk meg, ha u-ból nem tudunk folyamot pumpálni a belőle induló még telí-

tetlen éleken, mert azok végpontjai ugyanazon a szinten vannak, mint u, vagy magasabban. (Definíció szerint s-nél és t-nél nem csordulhat túl a folyam, így a megemelést sem alkalmazhatjuk rájuk.)

MEGEMELÉS(u)

1 ▷ **Akkor alkalmazható,** ha u-nál a folyam túlcsordul, és ha $(u, v) \in E_f$, akkor $h[u] \leq h[v]$ teljesül.
2 ▷ **Ezt csinálja** az algoritmus: megemeli az u magasságát.
3 $h[u] \leftarrow 1 + \min\{h[v] : (u, v) \in E_f\}$

Amikor a MEGEMELÉS(u)-t hívjuk meg, azt mondjuk, hogy az u pontot ***megemeljük.*** Fontosnak tartjuk megjegyezni, hogy amikor u-t megemeljük, akkor legalább egy E_f-beli él kiindul u-ból, vagyis a programban a minimalizálás nemüres halmazon történik. Ez abból következik, hogy az u-nál a folyam túlcsordul: mivel $e[u] = f[V, u] > 0$, ezért létezik legalább egy v pont, hogy $f[v, u] > 0$, és ekkor

$$\begin{aligned} c_f(u, v) &= c(u, v) - f[u, v] \\ &= c(u, v) + f[v, u] \\ &> 0, \end{aligned}$$

ami szerint $(u, v) \in E_f$. A MEGEMELÉS(u) a lehető legmagasabbra emeli az u pontot, ami után még magasságfüggvényt kapunk.

Az általános algoritmus

Az általános előfolyam-algoritmus az alábbi szubrutinnal állítja elő a kiindulási előfolyamot.

INDULÓ-ELŐFOLYAM(G, s)

1 **for** minden $u \in V[G]$ pontra
2 **do** $h[u] \leftarrow 0$
3 $e[u] \leftarrow 0$
4 **for** minden $(u, v) \in E[G]$ élre
5 **do** $f[u, v] \leftarrow 0$
6 $f[v, u] \leftarrow 0$
7 $h[s] \leftarrow |V[G]|$
8 **for** minden $u \in \{u : (u, v) \in E[G]\}$ pontra
9 **do** $f[s, u] \leftarrow c(s, u)$
10 $f[u, s] \leftarrow -c(s, u)$
11 $e[u] \leftarrow c(s, u)$
12 $e[s] \leftarrow e[s] - c(s, u)$

Az INDULÓ-ELŐFOLYAM(G, s) a következő folyamot definiálja:

$$f[u, v] = \begin{cases} c(u, v), & \text{ha } u = s, \\ -c(v, u), & \text{ha } v = s, \\ 0 & \text{egyébként.} \end{cases}$$

Eszerint az s forrásból kiinduló minden élen az előfolyam értéke az él kapacitása, és minden más élen 0 az előfolyam értéke. Azaz az s minden v szomszédjára $e[v] = c(s, v)$, és $e[s]$ pedig ezen értékek összegének az ellentettje. Az általános algoritmus az alábbi h magasságfüggvénnyel indul:

$$h[u] = \begin{cases} |V|, & \text{ha } u = s, \\ 0 & \text{egyébként.} \end{cases}$$

Ez valóban magasságfüggvény, mert azok az (u, v) élek, amelyekre $h[u] > h[v] + 1$, csak azok, amelyekre $u = s$, ezek az élek viszont az induló előfolyamban telítettek, ami azt jelenti, hogy nem szerepelnek a reziduális hálózatban.

Ezen inicializálás után a pumpálások és megemelések tetszőleges sorrendben végrehajtott alkalmazását nevezzük általános előfolyam-algoritmusnak:

ÁLTALÁNOS-ELŐFOLYAM(G)

1 INDULÓ-ELŐFOLYAM(G, s)
2 **while** végrehajtható valamely pumpálás vagy megemelés művelet
3 **do** válasszunk ki egy végrehajtható pumpálás vagy megemelés műveletet,
és hajtsuk végre

A következő lemma szerint amíg található a hálózatban olyan pont, amelynél az előfolyam túlfolyik, addig a két alapművelet legalább egyikét alkalmazni tudjuk.

26.15. lemma (egy túlfolyó pontnál vagy tudunk pumpálni, vagy a pont megemelhető). *Legyen $G = (V, E)$ egy hálózat az s termelővel és t fogyasztóval. Legyen f egy előfolyam és h egy f-hez tartozó magasságfüggvény. Ha u-nál az előfolyam túlcsordul, akkor vagy a pumpálás művelet, vagy a megemelés művelet u-nál alkalmazható.*

Bizonyítás. A reziduális hálózat tetszőleges (u, v) élére teljesül, hogy $h(u) \le h(v) + 1$, mivel h egy f-hez tartozó magasságfüggvény. Ha a pumpálás művelet nem alkalmazható u-nál, akkor a reziduális hálózat minden (u, v) élére $h(u) < h(v) + 1$, azaz $h(u) \le h(v)$. Következésképpen a megemelés alkalmazható u-nál. ■

Az előfolyam-algoritmus helyessége

Most megmutatjuk, hogy az általános előfolyam-algoritmus csakugyan megoldja a maximális folyam problémáját, azaz megkeresi a maximális értékű folyamot. Először belátjuk, hogy ha az algoritmus leáll, akkor az f előfolyam egyúttal maximális folyam. Később pedig bebizonyítjuk, hogy az algoritmus egy idő után mindig meg is áll. Mindenekelőtt figyeljük meg a h magasságfüggvény néhány egyszerű tulajdonságát.

26.16. lemma (a pontok magassága sohasem csökken). *Az általános előfolyam-algoritmus lefutása alatt tetszőleges $u \in V$ pont $h[u]$ magassága sohasem csökken. Továbbá ha a megemelés műveletet hajtjuk végre u-n, akkor a $h[u]$ magasság legalább 1-gyel nő.*

Bizonyítás. Mivel egy pont magassága csak a megemelés során változhat, ezért elegendő csak a második állítást bizonyítani. Ha u-t megemeljük, akkor minden $(u, v) \in E_f$ él v végpontjára $h[u] \le h[v]$, ami szerint $h[u] < 1 + \min\{h[v] : (u, v) \in E_f\}$, tehát a megemelés csakugyan növeli $h[u]$-t. ■

26.17. lemma. *Az általános előfolyam-algoritmus lefutása alatt a h függvény végig magasságfüggvény marad.*

Bizonyítás. A bizonyítás a végrehajtott pumpálások és megemelések számára vonatkozó indukcióval történik. Láttuk, hogy a kezdeti h valóban magasságfüggvény.

Elegendő megmutatnunk, hogy a MEGEMELÉS(u) és a PUMPÁLÁS(u, v) műveletek után a megváltozott h szintén magasságfüggvény. Most megmutatjuk, hogy ha h magasságfüggvény, akkor a MEGEMELÉS(u) végrehajtása után az új h is magasságfüggvény. Tekintsük az u-ból kiinduló $(u, v) \in E_f$ reziduális élt. A MEGEMELÉS(u) biztosítja, hogy a $h[u]$ megváltozott értékére $h[u] \leq h[v] + 1$. Most tekintsünk egy u-ba mutató $(w, u) \in E_f$ reziduális élt. A 26.16. lemma szerint $h[w] \leq h[u] + 1$ miatt u megemelése után $h[w] < h[u] + 1$. Tehát a MEGEMELÉS(u) után a megváltozott h függvény szintén magasságfüggvény lesz.

Most tekintsük a pumpálás műveletet. Egy ilyen művelet után a (v, u) esetleg bekerül E_f-be, (u, v) pedig esetleg kikerül belőle. Az előbbi esetben $h[v] = h[u] - 1$, így $h[u] \leq h[v] + 1$ teljesül. Az utóbbi esetben pedig a magasságfüggvény definíciójában a $h[u] \leq h[v] + 1$ megszorítás nem fog szerepelni. ■

A következő lemma a magasságfüggvények egy fontos tulajdonságáról szól.

26.18. lemma. *Legyen $G = (V, E)$ egy hálózat az s termelővel és a t fogyasztóval, továbbá legyen f egy előfolyam, h pedig egy f-hez tartozó tetszőleges magasságfüggvény. Ekkor a az f előfolyamhoz tartozó G_f reziduális gráfban nem létezik s-ből t-be vezető út.*

Bizonyítás. Indirekten tegyük fel, hogy létezik egy $p = \langle v_0, v_1, \ldots, v_k \rangle$ élsorozat a $v_0 = s$-ből a $v_k = t$-be. Az általánosság megszorítása nélkül feltehetjük, hogy p egy út, azaz a v_i pontok közül bármelyik kettő különbözik, és ezért $k < |V|$. Minden $i = 0, 1, \ldots, k - 1$-re $(v_i, v_{i+1}) \in E_f$. Mivel h egy magasságfüggvény, ezért $h(v_i) \leq h(v_{i+1}) + 1$ minden lehetséges i-re. Ezekből $h(s) \leq h(t) + k$ adódik, ami ellentmond a magasságfüggvény $h(t) = 0$, $h(s) = |V|$ tulajdonságának. ■

Ezek után készen állunk annak bizonyítására, hogy az általános előfolyam-algoritmus megállása esetén a kapott előfolyam egyben maximális folyam is.

26.19. tétel (az általános előfolyam-algoritmus helyessége). *Ha az* ÁLTALÁNOS-ELŐFOLYAM *megáll a $G = (V, E)$ hálózaton futtatva, akkor az általa meghatározott f előfolyam maximális folyam.*

Bizonyítás. A következő ciklusinvariánst alkalmazzuk.

> ÁLTALÁNOS-ELŐFOLYAM második sorában lévő **while** teszt bármely végrehajtásakor az f előfolyamot alkot.

Teljesül: Az INDULÓ-ELŐFOLYAM által definiált f előfolyam.

Megmarad: A második és harmadik sor **while** ciklusában végrehajtott műveletek csak a megemelés és a pumpálás. A megemelések csak a magasságfüggvényt változtatják, az f értékeit nem; ezért az f előfolyam voltára nincsen hatásuk. Ahogyan a MEGEMELÉS(u, v) definiálása után láttuk, ha f a megemelés végrehajtása előtt előfolyam volt, az után is az marad.

Befejeződik: Leálláskor a 26.15 és a 26.17. lemma szerint, mivel az f végig előfolyam, a $V-\{s,t\}$ minden pontjában a többlet 0, azaz nincsenek olyan pontok, ahol az előfolyam túlfolyna. Tehát az f egyben folyam is. Mivel h magasságfüggvény, a 26.18. lemma szerint a G_f reziduális hálózatban nincs s-ből t-be vezető út. A maximális folyam minimális vágás tételt (26.7. tétel) alkalmazva azt nyerjük, hogy f maximális folyam. ■

Az általános előfolyam-algoritmus elemzése

Annak bizonyításához, hogy az általános előfolyam-algoritmus valóban megáll, megbecsüljük a végrehajtott műveleteinek a számát. Külön becsüljük meg a megemelések, a telítő pumpálások és a nemtelítő pumpálások számát. Ezen becslések ismeretében azonnal adódik az algoritmus egy $O(V^2E)$ futási idejű megvalósítása. Mindenekelőtt a következő alapvető lemmát bizonyítjuk be.

26.20. lemma. *Legyen $G=(V,E)$ egy hálózat az s termelővel és a t fogyasztóval, továbbá legyen f egy előfolyam. Ha az u pontnál az f előfolyam túlcsordul, akkor a G_f reziduális gráfban létezik u-ból s-be irányított út.*

Bizonyítás. Legyen $U=\{v: \text{létezik } u\text{-ból } v\text{-be irányított út } G_f\text{-ben}\}$, és indirekten tegyük fel, hogy $s\notin U$. Legyen $\overline{U}=V-U$.

Azt állítjuk, hogy minden $v\in U$, $w\in\overline{U}$ pontpárra: $f(w,v)\le 0$. Ugyanis ha $f(w,v)>0$, akkor $f(v,w)<0$, amiből $c_f(v,w)=c(v,w)-f(v,w)>0$ következik, és így $(v,w)\in E_f$ következne. De mivel u-ból létezik $v\in U$-ba irányított út, ezért $w\in U$ is igaz lenne, ellentmondásban $w\in\overline{U}$-val.

Eszerint $f(\overline{U},U)\le 0$, mivel az implicit összegzés minden tagja nempozitív, és így

$$\begin{aligned} e(U) &= f(V,U) && \text{((26.9) egyenlőség miatt)}\\ &= f(\overline{U},U)+f(U,U) && \text{(26.1. lemma 3. része miatt)}\\ &= f(\overline{U},U)\le 0. && \text{(26.1. lemma 3. része miatt)} \end{aligned}$$

Mivel minden $V-\{s\}$-beli pontban a feleslegfolyam nemnegatív, ezért a feltevésünk szerint minden $v\in U(\subseteq V-\{s\})$ pontra $e(v)=0$. Vagyis $e(u)=0$ is igaz, ami ellentmond annak, hogy u-ban túlcsordul a folyam. ■

A következő lemma a pontok lehetséges magasságaira ad felső korlátot, aminek következményeként a megemelések összes számát felülről meg tudjuk becsülni.

26.21. lemma. *Legyen $G=(V,E)$ egy hálózat az s termelővel és a t fogyasztóval. Az általános előfolyam-algoritmus G hálózaton való lefutása alatt minden $u\in V$ pontra $h[u]\le 2|V|-1$.*

Bizonyítás. Az s termelő és a t fogyasztó magassága nem változik, mivel definíció szerint a folyam nem csordulhat túl ezen pontokban, így hát $h[s]=|V|$ és $h[t]=0$ teljesül, továbbá mindkét h érték legfeljebb $2|V|-1$.

Tekintsünk egy tetszőleges $u\in V-\{s,t\}$ pontot. Kezdetben $h[u]=0\le 2|V|-1$. Megmutatjuk, hogy minden megemelés után továbbra is $h[u]\le 2|V|-1$. Az u megemelésekor az előfolyam u-nál túlcsordul, a 26.20. lemma szerint tehát létezik egy u-ból s-be vezető p út

G_f-ben. Legyen $p = \langle v_0, v_1, \ldots, v_k \rangle$, ahol $v_0 = u,\ v_k = s$, és $k \leq |V| - 1$, mivel p egyszerű út. Minden $i = 0, 1, \ldots, k-1$-re $(v_i, v_{i+1}) \in E_f$, ezért a 26.17. lemma szerint $h[v_i] \leq h[v_{i+1}] + 1$. Ebből következik, hogy $h[u] = h[v_0] \leq h[v_k] + k \leq h[s] + |V| - 1 = 2|V| - 1$. ■

26.22. következmény (a megemelések száma). *Legyen $G = (V, E)$ egy hálózat az s termelővel és a t fogyasztóval. Az általános előfolyam-algoritmus lefutása alatt egy pontot legfeljebb $2|V| - 1$-szer emelünk meg. Következésképpen a megemelés művelet legfeljebb $(2|V| - 1) \cdot (|V| - 2) < 2|V|^2$-szer kerül végrehajtásra.*

Bizonyítás. A $V - \{s, t\}$-beli pontok magassága változik csak az algoritmus lefutása alatt. Legyen $u \in V - \{s, t\}$ egy tetszőleges pont. A MEGEMELÉS(u) növeli $h[u]$-t. Kezdetben $h[u] = 0$, és a 26.21. lemma értelmében legfeljebb $(2|V| - 1)$-ig nő. Következésképpen minden $u \in V - \{s, t\}$ pontot legfeljebb $(2|V| - 1)$-szer emelhetünk meg, azaz összesen legfeljebb $(2|V| - 1)(|V| - 2) < 2|V|^2$ a megemelések száma az általános előfolyam-algoritmus lefutása alatt. ■

A 26.21. lemma segítségével a telítő pumpálások számát is meg tudjuk becsülni.

26.23. lemma (korlát a telítő pumpálások számára). *A $G = (V, E)$ hálózaton lefuttatott általános előfolyam-algoritmus során a telítő pumpálások száma legfeljebb $2|V||E|$.*

Bizonyítás. Tekintsük valamely $u, v \in V$ pontpárra az u-ból v-be történő telítő pumpálásokat és a v-ből u-ba történő telítő pumpálásokat (ezeket u és v közötti telítő pumpálásoknak fogjuk nevezni). Ha előfordulnak ilyen pumpálások, akkor az (u, v) és (v, u) él legalább egyike E-beli, azaz éle a G hálózatnak. Tegyük fel, hogy az algoritmus lefutása során történt egy u-ból v-be való telítő pumpálás. Ekkor $h[v] = h[u] - 1$. Ahhoz, hogy a továbbiakban u-ból v-be ismét történhessen pumpálás, az algoritmusnak előbb a v-ből u-ba kell pumpálnia valamennyi folyamot, amihez a v-t u fölé kell emelni, azaz a magasságát legalább 2-vel növelni kell. Hasonlóképpen: az u magasságának szintén 2-vel kell nőnie két v-ből u-ba vezető telítő pumpálás között. Mivel a magasságok kezdőértéke 0, és a 26.21. lemma értelmében soha sem haladja meg $2|V| - 1$-et, ezért tetszőleges pont magasságát 2-vel legfeljebb $|V|$-szer emelhetjük meg. Mivel $h[u]$ és $h[v]$ legalább egyike legalább 2-vel nő meg két u és v közötti telítő pumpálás között, ezért kevesebb, mint $2|V|$ telítő pumpálás történhet u és v között. Ezt az élek számával beszorozva adódik a $2|V||E|$ felső korlát a telítő pumpálások összes számára. ■

A következő lemma az általános előfolyam-algoritmus lefutása alatt elvégzett nemtelítő pumpálások számára ad felső korlátot.

26.24. lemma (a nemtelítő pumpálások száma). *A $G = (V, E)$ hálózaton lefuttatott* ÁLTALÁNOS-ELŐFOLYAM *során a nemtelítő pumpálások száma legfeljebb $4|V|^2(|V| + |E|)$.*

Bizonyítás. Tekintsük a $\Phi = \sum_{v:e(v)>0} h[v]$ potenciálfüggvényt. Kezdetben $\Phi = 0$, és Φ értéke minden megemelés, telítő pumpálás és nemtelítő pumpálás után változhat. Meg fogjuk becsülni, hogy a telítő pumpálások és a megemelések mennyivel tudják összesen növelni Φ-t. Ezután megmutatjuk, hogy minden egyes nemtelítő pumpálás legalább 1-gyel csökkenti Φ-t, ami után egy felső becslést kapunk a nemtelítő pumpálások számára.

Vizsgáljuk meg azt a két esetet, amikor a Φ növekedhet. Vegyük észre, hogy a megemelés művelet a Φ-t legfeljebb $2|V|$-vel növeli, mivel azon pontok halmaza, amelyeknél a folyam túlcsordul, nem változik, és egy pont nem emelhető meg jobban, mint a maximális elérhető magasság, ami a 26.21. lemma szerint $2|V|$. Egy u-ból v-be vezető telítő pumpálás során Φ legfeljebb $2|V|$-vel nő, mivel semelyik pont magassága sem változik, és esetleg a v pontnál – aminek a magassága legfeljebb $2|V| - 1$ – a folyam a pumpálás után túlcsordul.

Most megmutatjuk, hogy az u és v közötti nemtelítő pumpálás a Φ értéket legalább 1-gyel csökkenti. A nemtelítő pumpálás előtt az u-nál a folyam túlcsordult, v-nél pedig vagy túlcsordult, vagy nem. A nemtelítő pumpálás után viszont a 26.14. lemma értelmében az u-nál már nem folyik túl a folyam. Továbbá a pumpálás után a v-nél a folyam mindenképpen túlcsordul, kivéve a $v = s$ esetét. Következésképpen a Φ-ből pontosan $h[u]$-t vontunk ki, és $h[v]$-t vagy 0-t adtunk hozzá. Mivel $h[u] - h[v] = 1$, ezért a Φ potenciálfüggvény legalább 1-gyel csökkent.

Az algoritmus lefutása alatt tehát a 26.22. következmény és a 26.23. lemma szerint a Φ teljes növekedése legfeljebb $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|)$. Mivel $\Phi \geq 0$, ezért a Φ csökkenéseinek az összszáma – és így a nemtelítő pumpálások összszáma is – legfeljebb $4|V|^2(|V| + |E|)$. ■

Miután ily módon megbecsültük a megemelések, a telítő pumpálások és a nemtelítő pumpálások számát, elérkeztünk az ÁLTALÁNOS-ELŐFOLYAM helyességének bizonyításához, és egyben minden más ráépülő algoritmus helyességének a bizonyításához is.

26.25. tétel. *Az* ÁLTALÁNOS-ELŐFOLYAM *lefutása alatt egy tetszőleges $G = (V, E)$ hálózaton a pumpálások és a megemelések összes száma $O(V^2E)$.*

Bizonyítás. A 26.22. következmény, 26.23. és 26.24. lemma azonnali következménye. ■

Ezek szerint az algoritmus $O(V^2E)$ darab művelet végrehajtása után megáll. Ezek után már csak az egyes műveletek hatékony végrehajtásának és a soron következő művelet hatékony kiválasztásának megszervezésére van szükség.

26.26. következmény. *Az általános előfolyam-algoritmusnak létezik olyan konkrét megvalósítása, amelynek futási ideje egy $G = (V, E)$ hálózaton $O(V^2E)$.*

Bizonyítás. A 26.4-1. gyakorlat az algoritmusnak olyan megvalósítását kérdezi, ami egy megemelést $O(V)$ lépésben, egy pumpálást pedig $O(1)$ lépésben képes elvégezni. Továbbá egy olyan adatstruktúra meghatározását is kéri a gyakorlat, amely segítségével a soron következő végrehajtható művelet kiválasztás $O(1)$ lépésben történik. Ezek éppen a kívánt futási időt eredményezik. ■

Gyakorlatok

26.4-1. Hogyan kell az általános előfolyam-algoritmust megvalósítani, hogy a megemelés művelet $O(V)$ lépésben, a pumpálás $O(1)$ lépésben, a soron következő végrehajtandó művelet kiválasztása pedig $O(1)$ lépésben elvégezhető legyen?

26.4-2. Bizonyítsuk be, hogy az általános előfolyam-algoritmus a $O(V^2)$ darab megemelési műveletre összesen $O(VE)$ időt fordít.

26.4-3. Tegyük fel, hogy egy előfolyam-algoritmus a $G = (V, E)$ hálózatban megkeresett egy maximális folyamot. Adjunk minél gyorsabb algoritmust ezután a G egy minimális vágásának a megkeresésére.

26.4-4. Adjunk hatékony előfolyam-algoritmust egy páros gráf maximális párosításának megkeresésére. Határozzuk meg az algoritmus lépésszámát is.

26.4-5. A $G = (V, E)$ hálózat minden élének a kapacitása az $\{1, 2, \ldots, k\}$ halmazból való. Mekkora a futási ideje az általános előfolyam-algoritmusnak a $|V|$, $|E|$, k értékekben kifejezve? (*Útmutatás.* Mielőtt egy él telítetté válna, hányszor kerülhet sor rajta nemtelítő pumpálásra?)

26.4-6. Mutassuk meg, hogy az INDUL Ó-ELŐFOLYAM(G, s) 7-edik sora az algoritmus helyességének és az aszimptotikus viselkedésének a megváltozása nélkül helyettesíthető a következővel:

7 $h[s] \leftarrow |V[G]| - 2$.

26.4-7. Legyen $\delta_f(u, v)$ az u pontnak a G_f reziduális hálózatban a v-től való távolsága, azaz a legrövidebb u-ból v-be vezető irányított út élszáma. Mutassuk meg, hogy az ÁLTALÁNOS-ELŐFOLYAM lefutása alatt végig igazak a következők: ha $h[u] < |V|$, akkor $h[u] \leq \delta_f(u, t)$, és ha $h[u] \geq |V|$, akkor $h[u] - |V| \leq \delta_f(u, s)$.

26.4-8.⋆ Az előző gyakorlathoz hasonlóan legyen $\delta_f(u, v)$ az u pontnak a G_f reziduális hálózatban a v-től való távolsága. Mutassuk meg, hogy az általános előfolyam-algoritmus módosítható olyan módon, hogy fennálljanak a következők: ha $h[u] < |V|$, akkor $h[u] = \delta_f(u, t)$, és ha $h[u] \geq |V|$, akkor $h[u] - |V| = \delta_f(u, s)$. Az összes idő, amit az algoritmus ezen tulajdonság fenntartásával tölthet, $O(VE)$ lehet.

26.4-9. Mutassuk meg, hogy az ÁLTALÁNOS-ELŐFOLYAM a $G = (V, E)$ hálózaton legfeljebb $4|V|^2|E|$ darab nemtelítő pumpálást hajt végre, amennyiben $|V| \geq 4$.

⋆ 26.5. Az előreemelő algoritmus

Az általános előfolyam-algoritmus semmilyen megszorítást nem tartalmazott az elvégzendő műveletek sorrendjére vonatkozóan. Ezen sorrend, illetve az adatstruktúra okos megválasztásával a maximális folyam problémát gyorsabban is meg tudjuk oldani, mint a 26.26. következmény $O(V^2E)$ eredménye. Ebben az alfejezetben az úgynevezett előreemelő előfolyam-algoritmussal fogunk megismerkedni, ami egy speciális előfolyam-algoritmus. A futási ideje $O(V^3)$, ami aszimptotikusan legalább olyan jó, mint a $O(V^2E)$, sok élű gráfokon pedig határozottan jobb.

Az előreemelő algoritmus fontos eleme a hálózat pontjainak egy listája. Az algoritmus ezen lista elejéről indulva végigvizsgálja a pontokat, és ha talál egy olyat, amelynél a folyam túlcsordul, akkor azt „tehermentesíti”, azaz pumpálásokkal és megemelésekkel elvezeti a pont többletfolyamát. Amikor az algoritmus megemel egy pontot, akkor a lista elejére helyezi (innen származik az előreemelő elnevezés), és a túlfolyó pont utáni keresést elölről kezdi.

Az algoritmus helyességének a bizonyításában alapvető szerepet játszanak a „megengedett” élek: a reziduális hálózat azon élei, melyeken keresztül az előfolyam-algoritmus folyamot pumpálhat, azaz a reziduális kapacitásuk pozitív, és az aktuális magasságfüggvény szerint lefelé vezetnek. A megengedett élek néhány tulajdonságának bizonyítása

után megvizsgáljuk a *tehermentesítés* műveletet, majd rátérünk magára az előreemelő algoritmusra.

Megengedett élek és hálózatok

Legyen $G = (V, E)$ egy hálózat az s termelővel és t fogyasztóval. Legyen továbbá f egy G-beli előfolyam, h pedig egy hozzátartozó magasságfüggvény. Azt mondjuk, hogy az (u, v) él ***megengedett él,*** ha $c_f(u, v) > 0$, és $h(u) = h(v) + 1$. Máskülönben (u, v)-t ***nemmegengedettnek*** mondjuk. A $G_{f,h} = (V, E_{f,h})$-t ***megengedett hálózatnak*** nevezzük, ahol $E_{f,h}$ a megengedett élek halmaza.

A megengedett hálózat azon élekből áll, amelyeken az előfolyam-algoritmus folyamot pumpálhat. A következő lemma azt állítja, hogy egy ilyen hálózat irányított körmentes gráf, azaz nem tartalmaz irányított kört.

26.27. lemma (a megengedett hálózat körmentes). *Legyen $G = (V, E)$ egy hálózat, f egy G-beli előfolyam, h pedig egy hozzátartozó magasságfüggvény. Ekkor a $G_{f,h} = (V, E_{f,h})$ megengedett hálózat körmentes.*

Bizonyítás. Tegyük fel, hogy $G_{f,h}$ tartalmaz egy $p = \langle v_0, v_1, \ldots, v_k \rangle$ kört, ahol $v_0 = v_k$ és $k > 0$. Mivel p minden éle megengedett, ezért $h(v_{i-1}) = h(v_i) + 1$ igaz minden i-re $(i = 1, 2, \ldots, k)$. Ezeket összegezve kapjuk

$$\begin{aligned} \sum_{i=1}^{k} h(v_{i-1}) &= \sum_{i=1}^{k} (h(v_i) + 1) \\ &= \sum_{i=1}^{k} h(v_i) + k. \end{aligned}$$

Mivel a p körben minden pont pontosan egyszer fordul elő az összegzésben, azt kapjuk, hogy $0 = k$, ami ellentmondás. ■

A következő két lemma arról szól, hogyan változik meg a megengedett hálózat egy pumpálás, illetve egy megemelés után.

26.28. lemma. *Legyen $G = (V, E)$ egy hálózat f egy G-beli előfolyam, h pedig egy hozzátartozó magasságfüggvény. Ha az u pontnál a folyam túlcsordul, és az (u, v) él megengedett él, akkor a* Pumpálás(u, v) *művelet elvégezhető. A művelet után nem keletkezik megengedett él, de esetleg az (u, v) nemmegengedetté válik.*

Bizonyítás. A megengedett él definíciója szerint pumpálható folyam u-ból v-be, és az u-nál a folyam túlcsordul, tehát Pumpálás(u, v) csakugyan végrehajtható. Az egyetlen lehetséges reziduális él, ami egy u pontból v-be való pumpálás után keletkezhet, az a (v, u), de mivel $h(v) = h(u) - 1$, ezért ez nem lehet megengedett. Ha telítő pumpálás történt, akkor $c_f(u, v) = 0$ lesz, amire az (u, v) nemmegengedetté válik. ■

26.29. lemma. *Legyen $G = (V, E)$ egy hálózat f egy G-beli előfolyam, h pedig egy hozzátartozó magasságfüggvény. Ha az u pontnál a folyam túlcsordul, és u-ból nem indul ki megengedett él, akkor a* Megemelés(u) *alkalmazható. A megemelés után legalább egy megengedett él kiindul u-ból, de semelyik megengedett él sem végződik u-ban.*

Bizonyítás. Ha u-nál a folyam túlcsordul, akkor a 26.15. lemma szerint vagy a pumpálás alkalmazható u-nál, vagy az u megemelhető. Mivel nincs u-ból kiinduló megengedett él, ezért most csak a megemelés alkalmazható. A megemelés után $h[u] = 1 + \min\{h[v] : (u,v) \in E_f\}$. Tegyük fel, hogy a v pontra teljesül az előbbi minimum, ekkor az (u,v) él megengedetté válik. Vagyis a megemelés után legalább egy él csakugyan megengedett lesz.

Most tegyük fel indirekten, hogy a megemelés után létezik egy v pont, amelyre a (v,u) él megengedett. Ekkor $h[v] = h[u] + 1$, és ezért közvetlenül a megemelés előtt $h[v] > h[u] + 1$. De a 26.13. lemma szerint nem létezik reziduális él két olyan pont között, amelyek magasságának a különbsége 1-nél nagyobb, tehát a (v,u) sem a megemelés előtt, sem utána nincs a reziduális hálózatban, következésképpen a megengedett hálózatban sem. ■

Szomszédsági listák

Most megadjuk az előreemelő algoritmus során felhasználandó szomszédsági listák definícióját. A hálózat éleit szomszédsági listákba rendezzük. Adott a $G = (V,E)$ hálózat, egy $u \in V$ pont $N[u]$ szomszédsági listája az u G-beli szomszédainak egyszeresen láncolt listája. Vagyis a v pont akkor jelenik meg az $N[u]$ listában, ha $(u,v) \in E$ vagy $(v,u) \in E$. Az $N[u]$ szomszédsági lista pontosan azokat az éleket tartalmazza, amelyekre az (u,v) vagy a (v,u) reziduális él lehet. Az $N[u]$ lista első pontjára mutat a $fej[N[u]]$. A listában a v-t követő szomszédra a *köv-szomszéd*$[v]$ pointer mutat, ha v az utolsó szomszéd, akkor *köv-szomszéd*$[v]$ értéke NIL.

Az előreemelő algoritmus a hálózat pontjainak egy tetszőleges, de rögzített ciklikus permutációja szerint sorra veszi (körkörösen) a pontokat. Minden u pontra a *most-szomszéd*$[u]$ megadja az éppen vizsgálandó $N[u]$-beli szomszédot. Kezdetben *most-szomszéd*$[u]$ értéke $fej[N[u]]$.

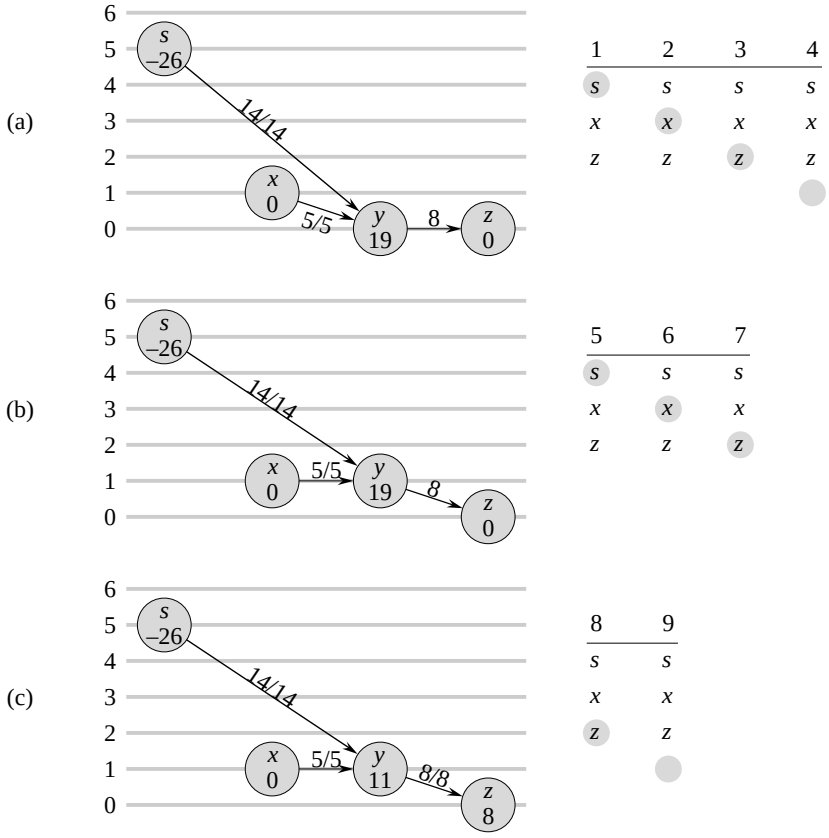
Egy túlfolyó pont tehermentesítése

Egy u pontot, ahol a folyam túlcsordul, úgy ***tehermentesítünk,*** hogy az összes feleslegfolyamát megengedett éleken keresztüli pumpálásokkal, és az u esetleges többszöri megemelésével elvezetjük. Az algoritmus a következő:

TEHERMENTESÍTÉS(u)

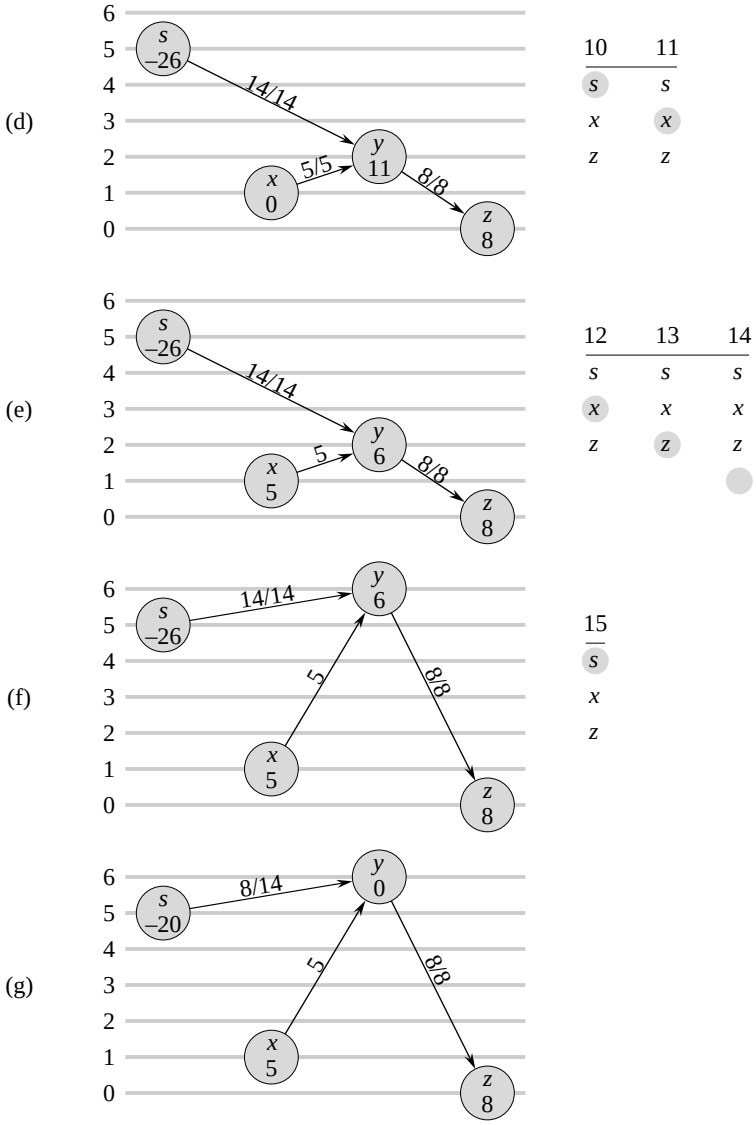
```
1  while e[u] > 0
2        do v ← most-szomszéd[u]
3           if v = NIL
4              then MEGEMELÉS(u)
5              most-szomszéd[u] ← fej[N[u]]
6           elseif c_f(u,v) > 0 és h[u] = h[v] + 1
7                 then PUMPÁLÁS(u,v)
8           else most-szomszéd[u] ← köv-szomszéd[v]
```

A 26.9. ábrán egy példán nyomon követjük a **while** ciklus lefutását, ami mindaddig fut, amíg az u-nak feleslegfolyama van. Minden egyes iterációban a következő három eset közül az egyik kerül végrehajtásra attól függően, hogy az $N[u]$ listában melyik az aktuális pont.



26.9. ábra. Egy pont tehermentesítése. A TEHERMENTESÍTÉS **while** ciklusának 15 iterációja alatt sikerül az y pont összes feleslegfolyamát elvezetni (*elpumpálni*). Az ábrákba csak az y szomszédait, és az y-ra illeszkedő éleket rajzoltuk be. Az ábrákon a pontok belsejébe az ábrához tartozó első iteráció kezdetén meglévő feleslegfolyam nagyságát írtuk, és minden részben a pontok a magasságuknak megfelelő szintre vannak berajzolva. A jobb oldalon az iterációk sorszámai, és alattuk az y $N[y]$ szomszédsági listája szerepel. A beszürkített pont a *most-szomszéd*$[y]$. **(a)** Kezdetben 19 egységnyi elvezetendő feleslegfolyama van y-nak, és *most-szomszéd*$[y] = s$. Az 1., 2., 3. iterációban csak *most-szomszéd*$[y]$ halad tovább a szomszédokon, mivel nincsen y-ból kiinduló megengedett él. A 4. iterációban *most-szomszéd*$[y] =$ NIL (ezt jelöli a szomszédsági lista alatti szürke pötty), és így az y-t megemeljük, *most-szomszéd*$[u]$ pedig a lista elejére ugrik. **(b)** A megemelés után y magassága 1 lett. Az 5. és 6. iterációban az (y, s) és (y, x) élek nemmegengedettnek bizonyulnak, de a 7-edikben az y-ból z-be 8 egységnyi feleslegfolyamot átpumpálunk. A pumpálás miatt *most-szomszéd*$[y]$ nem változik most. **(c)** Mivel a 7. iterációban az (y, z) él telítődött, ezért ez az él nemmegengedett a 8. iterációban. A 9. iterációban *most-szomszéd*$[y] =$ NIL, tehát az y-t megemeljük, és *most-szomszéd*$[y]$-t megint a lista elejére állítjuk. **(d)** A 10. iterációban (y, s) nemmegengedett, de a 11-edikben 5 egységet átpumpálunk x-be. **(e)** Mivel a 11. iterációban *most-szomszéd*$[y]$ nem változott, a 12. iteráció (y, x)-et vizsgálja, és nemmegengedettnek találja azt. A 13. iteráció (y, z)-t nemmegengedettnek látja, a 14. iteráció megemeli y-t, és *most-szomszéd*$[y]$-t előreállítja. **(f)** A 15. iteráció 6 egységnyi fölösleget átpumpál y-ból s-be. **(g)** Az y pontnak immáron nincs feleslegfolyama, ezért a TEHERMENTESÍTÉS(y) megáll. Ebben a példában a TEHERMENTESÍTÉS úgy indul, és úgy is végződik, hogy a *most-szomszéd* a szomszédsági lista első elemére mutat, de ennek általánosságban természetesen nem kell így lennie.

1. Ha $v =$ NIL, akkor túlfutottunk az $N[u]$ utolsó elemén. A 4. sor megemeli u-t, aztán az 5. sor az aktuálisan vizsgálandó szomszédot az $N[u]$ lista első elemére állítja. (A most következő 26.30. lemma azt állítja, hogy ebben az esetben a megemelés valóban alkalmazható.)



2. Ha $v \neq$ NIL, és (u, v) megengedett él (a 6. sor vizsgálja, hogy ez-e a helyzet), akkor a 7. sor az u valamennyi (esetleg az egész) feleslegfolyamát átpumpálja az (u, v) élen v-be.

3. Ha $v \neq$ NIL, de (u, v) nemmegengedett, akkor a 8. sor továbblépteti *most-szomszéd*$[u]$-t az $N[u]$ lista következő elemére.

Figyeljük meg, hogy amikor egy olyan u-ra hívjuk meg a TEHERMENTES ÍTÉS(u)-t, amelynél a folyam túlcsordul, akkor az általa utolsónak végrehajtott műveletnek a pumpálásnak kell lennie. Ugyanis a TEHERMENTESÍTÉS(u) csak akkor áll le, amikor az $e[u]$ 0-vá válik, és az $e[u]$ sem a megemelés, sem a *most-szomszéd* továbbléptetése során nem változik.

Most belátjuk, hogy amikor a TEHERMENTESÍTÉS a PUMPÁLÁS-t vagy a MEGEMELÉS-t hívja meg, akkor ezek valóban alkalmazhatók is.

26.30. lemma. *Amikor a* TEHERMENTESÍTÉS(u) *a* 7. *sorában meghívja a* PUMPÁLÁS(u, v)*-t, akkor az* (u, v)*-re csakugyan alkalmazható. Amikor a* TEHERMENTESÍTÉS(u) *a* 4. *sorában meghívja a* MEGEMELÉS(u)*-t, akkor az u csakugyan megemelhető.*

Bizonyítás. Az algoritmus az 1. és 6. sorban eldönti, hogy PUMPÁLÁS(u, v) csakugyan alkalmazható-e, és csak ezután hívja meg, amivel a lemma első állítását be is láttuk.

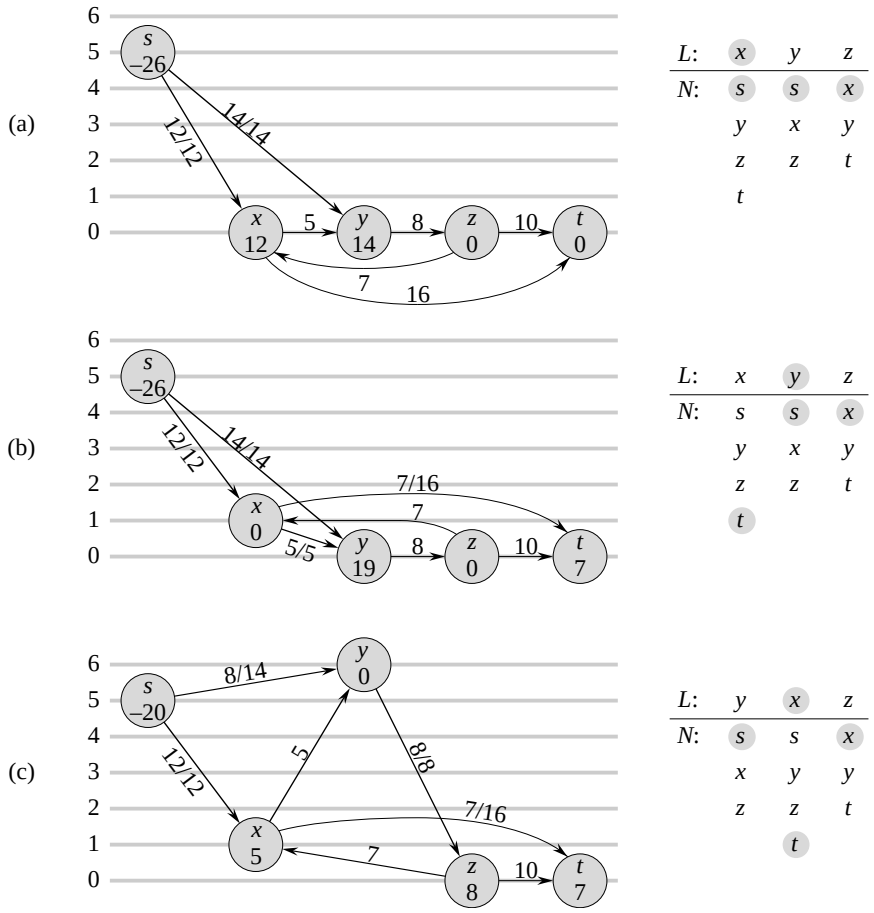
Mivel az 1. sor megvizsgálja, hogy van-e feleslegfolyama u-nak, ezért a lemma második állításának a bizonyításához a 26.29. lemma szerint elég megmutatnunk, hogy az u-ból kiinduló élek mind nemmegengedettek. Figyeljük meg, hogy amint a TEHERMENTESÍTÉS(u) **while** ciklusa sorban végrehajtódik, a *most-szomszéd*[u] vándorol lefelé az $N[u]$ listán. Minden „forduló” az $N[u]$ elején kezdődik, és akkor ér véget, amikor *most-szomszéd*[u] = NIL, amikor is u-t megemeljük, és egy új forduló veszi kezdetét. Ahhoz, hogy a *most-szomszéd*[u] a $v \in N[u]$ szomszédon továbblépjen, ahhoz az (u, v) élnek nemmegengedettnek kell lennie, amit a 6. sor vizsgál meg, és így amikorra a forduló véget ér, addigra minden u-ból kiinduló él nemmegengedettnek találtatott egyszer. Az igazság az, hogy a forduló végére minden él nemmegengedett marad, ugyanis a 26.28. lemma szerint egy pumpálás nem hoz létre új megengedett élt, ezért megengedett él csak megemelés során keletkezhet. De az u-t nem emeljük meg egy forduló közben, és a 26.29. lemma szerint minden más v pontba, amit megemelünk abban a fordulóban, nem megy megengedett él. Ezek szerint a forduló végére az u-ból kiinduló összes él nemmegengedett marad, amivel be is láttuk, amit akartunk. ■

Az előreemelő algoritmus

Az előreemelő algoritmus során fenntartjuk a $V - \{s, t\}$ pontok egy L láncolt listáját. Lényeges lesz, hogy L-ben a pontok a megengedett hálózat szerinti topologikus sorrendben fognak szerepelni, amint azt mindjárt látni fogjuk. (Emlékeztetünk arra, hogy a 26.27. lemma szerint a megengedett hálózat irányított körmentes gráf, ezért pontjainak létezik topologikus sorrendje.)

Most megadjuk magát az előreemelő algoritmust. Minden u pont $N[u]$ szomszédsági listáját már meglévőnek tekintjük. A *köv*[u] mutató az L listában az u-t közvetlenül követő pontra mutat, ha pedig u az L utolsó eleme, akkor szokás szerint *köv*[u] = NIL.

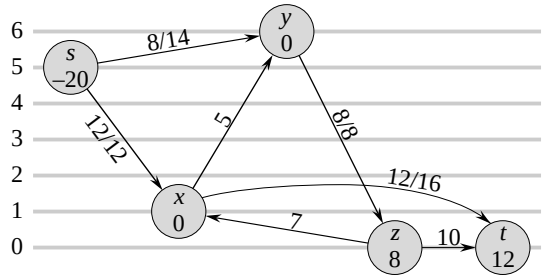
```
ELŐREEMEL(G, s, t)
 1  INDULÓ-ELŐFOLYAM(G, s)
 2  L ← V[G] – {s, t} tetszőleges sorrendben
 3  for minden u ∈ V[G] – {s, t} pontra
 4      do most-szomszéd[u] ← fej[N[u]]
 5  u ← fej[L]
 6  while u ≠ NIL
 7        do régi-magasság ← h[u]
 8           TEHERMENTESÍTÉS(u)
 9           if h[u] > régi-magasság
10              then rakjuk u-t az L elejére
11           u ← köv[u]
```



26.10. ábra. Az előreemelő algoritmus egy működése. **(a)** Egy hálózat az induló előfolyammal a **while** ciklus első lefutásának elkezdése előtt. Kezdetben 26 egységnyi folyam hagyja el az s termet, ő. Jobbra a pontok kezdő $L = \langle x, y, z \rangle$ listája, először $u = x$. Az L listában szereplő pontok alatt a szomszédsági listájukat adtuk meg, a *most-szomszéd*-okat szürke pöttyel emeltük ki. Az x pontra hívjuk meg először a tehermentesítést. Megemeljük az 1. szintre, a feleslegfolyamából 5 egységet átpumpálunk y-ba, a maradék 7-et pedig a t fogyasztóba. Mivel x-et megemeltük, ezért az L lista elejére helyezzük, ami nem okoz nagy változást, mert eredetileg is ott volt. **(b)** A következő pont, amit az x után tehermentesítünk, az y, mivel ő követi L-ben x-et. A 26.9. ábra mutatja a tehermentesítés részletes lefutását ebben az esetben. Mivel y-t megemeltük, ezért az L elejére helyezzük. **(c)** Az L-ben az y után most az x következik, ezért megintőt tehermentesítjük. Az 5 egységnyi feleslegfolyamát átpumpáljuk t-be. Mivel ezen tehermentesítés alatt x-et nem emeltük meg, ezért marad az L-ben a helyén. **(d)** Az L-ben x után most a z pont következik, őt tehermentesítjük. Magasságát 1-re emeljük, és az összes 8 egység feleslegfolyamát a t-be pumpáljuk. Mivel z-t megemeltük, ezért L elejére kerül. **(e)** z után soron következik az új L-ben tovább haladva y tehermentesítése. De mivel y-nak nincs feleslege, ezért a TEHERMENTESÍTÉS nyomban visszatér, y marad L-ben a helyén. Az x tehermentesítése következik, de mivel neki sincs feleslege, a TEHERMENTESÍTÉS megint befejeződik. Az ELŐREEMEL elérte a lista végét, ezért leáll. Nem maradt túlfolyó pont, a kiszámolt előfolyam: maximális folyam.

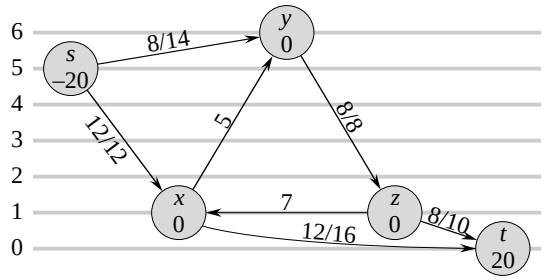
Az algoritmus 1. sora beállítja az induló előfolyamot és a magasságokat ugyanúgy, ahogyan az általános előfolyam-algoritmusnál. A 2. sor előállítja a túlfolyó pontok L listáját, amibe a pontok tetszőleges sorrendben kerülnek bele. A 3. és 4. sor minden u pontra beállítja a *most-szomszéd* mutatót a hozzátartozó szomszédsági lista első elemére.

(d)



L:	y	x	z
N:	s	s	x
	x	y	y
	z	z	t
		t	

(e)



L:	z	y	x
N:	x	s	s
	y	x	y
	t	z	z
			t

Ahogyan azt a 26.10. ábra mutatja, a 6–11. sorok **while** ciklusa sorra veszi az L lista elemeit, és tehermentesíti azokat. Az 5. sor az L lista első pontjáról indítja el a 6. sorban kezdődő ciklust. A ciklus minden egyes lefutásakor a 8. sorban az algoritmus tehermentesít egy u pontot. Amennyiben a TEHERMENTESÍTÉS alatt az u-t megemeltük, akkor a 10. sor az u-t az L elejére teszi. Az u régi magasságának a 7. sorban a *régi-magasság* változóban való tárolásának segítségével dönt a program a 9. sorában arról, hogy u-t megemeltük-e. A 11. sor az L listában továbblép eggyel, és újrakezdődik a ciklus. Ha az u az előbbiekben a lista elejére került, akkor a 11. sor az u-nak erről az új helyéről lép tovább L-ben.

Meg fogjuk mutatni, hogy az előreemelő algoritmus az általános előfolyam-algoritmus egy speciális megvalósítása, és eszerint az általa kiszámolt f folyam maximális folyam. Először figyeljük meg, hogy az algoritmus a TEHERMENTESÍTÉS alatt csak pumpálásokat és megemeléseket hajt végre, és a 26.30. lemma szerint csak akkor, amikor alkalmazhatóak is. Azt kell még belátnunk, hogy amikor ELŐREEMEL(G, s, t) megáll, akkor már nem marad alkalmazható megemelés vagy pumpálás. Ennek a bizonyításához megmutatjuk, hogy a következő tulajdonság végig teljesül.

> Az ELŐREEMEL(G, s, t) 6. sorának minden egyes alkalmazásakor az L lista a $G_{f,h} = (V, E_{f,h})$ megengedett hálózat pontjait topologikus sorrendben tartalmazza, továbbá a listában az u-t megelőző pontoknál nem folyik túl a folyam.

Teljesül: Közvetlenül az INDULÓ-ELŐFOLYAM lefutása után $h[s] = |V|$ és $h[v] = 0$ minden $v \in V\{s\}$ pontra. Mivel $|V| \geq 2$ (V ugyanis legalább s-et és t-t tartalmazza), semelyik él sem lehet megengedett. Így hát $E_{f,h} = \emptyset$, és a $V - \{s, t\}$ pontok tetszőleges sorrendje a $G_{f,h}$ topologikus sorrendje. Elsőként u az L első eleme, tehát az is fennáll, hogy az L-ben nem folyik túl a folyam semelyik őt megelőző pontnál.

Megmarad: Annak belátásához, hogy a **while** ciklus minden egyes iterációjában megmarad a topologikus sorrend, először vegyük észre, hogy a megengedett hálózat kizárólag a pumpálás és megemelés műveletek hatására változik. A 26.28. lemma szerint a pumpálás művelet nem hoz létre megengedett éleket, következésképpen megengedett élek csakis a megemelés hatására jöhetnek létre. Azonban a 26.29. lemma szerint az u pont megemelése után u-ba mutató megengedett élek nem jönnek létre, csak esetleg u-ból kilépő megengedett élek. Ami azt jelenti, hogy az u pontnak az L elejére helyezésével az algoritmus biztosítja azt, hogy megengedett élek csakis a listában hátrább lévő pontba mutassanak.

Annak megmutatásához, hogy az u-t a listában megelőző pontoknál nem folyik túl a folyam, jelöljük u'-vel azt a pontot, amely a következő iterációban u szerepét kapja. A soron következő iterációban az u'-t az L listában megelőző pontok az u (a 11. sor szerint) és vagy nincs más megelőző pont (abban az esetben, amikor u-t megemeltük), vagy még az u-t megelőző pontok is megelőzik u'-t (abban az esetben, amikor u-t nem emeltük meg). Mivel u-t tehermentesítettük, a folyam már nem folyik túl nála, tehát amennyiben a tehermentesítés alatt megemeltük u-t, akkor az u'-t megelőző pontoknál nem folyik túl a folyam. Ha pedig nem emeltük meg az u-t a tehermentesítése alatt, akkor az őt megelőző pontoknál nem keletkezhet túlfolyás ezen tehermentesítés alatt, mivel az L a pontok topologikus sorrendje marad ezen tehermentesítés alatt (ahogyan azt az előbb megemlítettük: megengedett élek kizárólag megemeléskor keletkezhetnek, pumpáláskor nem), vagyis a pumpálások túlfolyást csakis a listában hátrább szereplő pontoknál (vagy s-nél, vagy t-nél) okozhatnak. Azaz az u'-t megelőző pontoknál valóban nem folyik túl a folyam.

Befejeződik: Amikor a ciklus megáll, az u éppen az L utolsó eleme, tehát a fentiek szerint semelyik pontnál nem folyik túl a folyam, következésképp sem a megemelés, sem a pumpálás művelet nem alkalmazható.

Futási idő elemzése

Most megmutatjuk, hogy az Előreemel tetszőleges $G = (V, E)$ hálózaton $O(V^3)$ időben fut. Mivel az általános előfolyam-algoritmus egy speciális megvalósításáról van szó, ezért igaz itt is a 26.22. következmény állítása, miszerint egy pontot $O(V)$-szer emelünk meg, és az összes megemelések száma pedig $O(V^2)$. Továbbá a 26.4-2. gyakorlat szerint $O(VE)$ lépésben végrehajtható az összes megemelés. A 26.23. lemma pedig $O(VE)$ becslést ad a telítő pumpálások számára.

26.31. tétel. *Az* Előreemel *futási ideje a* $G = (V, E)$ *hálózaton* $O(V^3)$.

Bizonyítás. Nevezzük az előreemelő algoritmus egy fázisának két egymás utáni megemelés közötti működését. Mivel a megemelések száma $O(V^2)$, ezért ennyi fázisa van az algoritmusnak. Egy fázisban Tehermentesítés-t legfeljebb $O(V)$-szer hívjuk meg, ami a következőképp igazolható: ha Tehermentesítés nem hajt végre megemelést, akkor az L listában lépked lefelé, aminek hossza kisebb, mint $|V|$; ha Tehermentesítés végrehajt egy megemelést, akkor viszont a következő meghívása már a következő fázishoz tartozik. Eszerint Tehermentesítés-t az előreemelő algoritmus 8. sora $O(V^3)$-ször hívja meg, ami azt jelenti, hogy az algoritmus futási ideje a Tehermentesítés-ek „belső lépésszámát” nem számítva $O(V^3)$.

Most megbecsüljük a TEHERMENTESÍTÉS-ek alatt eltelt időt. A **while** ciklus egy lefutása három lehetőség közül egyet választ, és végrehajtja azt. Külön-külön megvizsgáljuk ezeket a lehetőségeket.

Ha a 4. és 5. sor megemelése hajtódik végre, akkor – a 26.4-2. gyakorlat szerint – ezek összideje az $O(V^2)$ megemelés alatt $O(VE)$.

Az algoritmus a 8. sorban *most-szomszéd*$[u]$-t, mielőtt egy u pontot megemelünk, $O(\text{fokszám}(u))$-szor lépteti tovább. Ez a művelet az u pont megemelésekor $O(\text{fokszám}(u))$-szor fordul elő, ami a csúcsra nézve összesen $O(V \cdot \text{fokszám}(u))$ lépést jelent. Így a szomszédsági listák mutatóinak frissítése az összes pontra együttvéve a kézfogási lemma szerint $O(VE)$ lépést eredményez (lásd B.4-1. gyakorlat).

A TEHERMENTESÍTÉS által választható harmadik lehetséges tevékenység a pumpálás művelet (7. sor). Tudjuk, hogy a telítő pumpálások száma $O(VE)$. Figyeljük meg, hogy amikor TEHERMENTESÍTÉS nemtelítő pumpálást hajt végre, már be is fejezi működését, hiszen az adott pontból az összes feleslegfolyamot sikerült elvezetnie. Tehát TEHERMENTESÍTÉS egy meghívásakor legfeljebb egy nemtelítő pumpálás történik, és mivel TEHERMENTESÍTÉS-t $O(V^3)$-ször hívjuk meg, ez azt jelenti, hogy a nemtelítő pumpálásokra fordított idő is $O(V^3)$.

Ezek szerint az ELŐREEMEL futási ideje $O(V^3 + VE) = O(V^3)$. ■

Gyakorlatok

26.5-1. Futtassuk le az ELŐREEMEL-t a 26.10. ábrán szereplő ábrázoláshoz hasonlóan a 26.1(a) ábra folyamhálózatán. Az L lista kezdetben legyen $\langle v_1, v_2, v_3, v_4 \rangle$, a szomszédsági listák pedig:

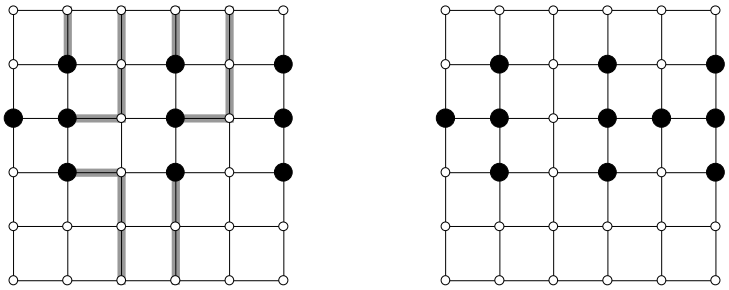
$$\begin{aligned} N[v_1] &= \langle s, v_2, v_3 \rangle, \\ N[v_2] &= \langle s, v_1, v_3, v_4 \rangle, \\ N[v_3] &= \langle v_1, v_2, v_4, t \rangle, \\ N[v_4] &= \langle v_2, v_3, t \rangle. \end{aligned}$$

26.5-2.⋆ Egy olyan előfolyam-algoritmust szeretnénk megvalósítani, amiben a túlfolyó pontok egy FIFO-ban vannak tárolva. Az algoritmus tehermentesíti a FIFO első pontját, majd azokat a pontokat, amelyeknek a tehermentesítés előtt nem volt feleslegfolyamuk, de utána lett, a lista végére teszi. Miután a lista első elemét tehermentesítette az algoritmus, kiveszi belőle. Amikor a lista üres lesz, az algoritmus leáll. Mutassuk meg, hogy ez az algoritmus csakugyan megoldja a maximális folyam problémát, és hogy létezik olyan megvalósítása, amelynek futási ideje $O(V^3)$.

26.5-3. Mutassuk meg, hogy az előreemelő algoritmus akkor is jól működik, ha a megemelés a $h[u]$-t a $h[u] \leftarrow h[u] + 1$ szabály szerint változtatja. Milyen hatással van ez az előreemelő algoritmus futási idejére?

26.5-4.⋆ Bizonyítsuk be, hogy ha mindig az egyik legmagasabb túlfolyó pontot tehermentesítjük, akkor olyan előfolyam-algoritmust kapunk, amelynek lépésszáma $O(V^3)$.

26.5-5. Tegyük fel, hogy egy előfolyam-algoritmus lefutásakor olyan helyzet adódik, hogy egy $0 < k < |V| - 1$ egész számra a hálózatban nincs olyan pont, melynek magassága $h[v] = k$. Mutassuk meg, hogy azon v pontok, melyekre $h[v] > k$, olyan minimális vágást alkotnak, amelyek az s-sel azonos komponensben vannak. Ha van ilyen k, akkor a ***rés-heurisztika*** minden $v \in V - s$ pontnak, amelyre $h[v] > k$, megváltoztatja a magasságát $h[v] \leftarrow \max(h[v], |V| + 1)$ szerint. Mutassuk meg, hogy az így kapott h is magasságfüggvény. (A rés-heurisztika a gyakorlatban működő előfolyam-algoritmusok fontos kelléke.)



(a) (b)

26.11. ábra. A kijutási probléma rácsokon. A kiindulási pontokat fekete pöttyel jelöltük. **(a)** Egy rács, amelyben a kijutási probléma megoldható, amint azt a berajzolt utak mutatják. **(b)** Egy rács, amelynek a kiindulási pontjaiból nem oldható meg a kijutási probléma.

Feladatok

26-1. Kijutási probléma

A 26.11. ábrán szereplő gráfot $n \times n$-es ***rácsnak*** nevezzük, amennyiben n sora és n oszlopa van. Az i-edik sor j-edik pontját (i, j)-vel jelöljük. A rács minden pontjának négy szomszédja van, kivéve a rács szélén elhelyezkedő pontokat, azaz azon (i, j) pontokat, amelyekre $i = 1$, $i = n$, $j = 1$ vagy $j = n$.

Adott a rácsnak $m \leq n^2$ darab $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ ún. kiindulási pontja. ***Kijutási feladatnak*** nevezzük annak az eldöntését, hogy vajon létezik-e m darab pontdiszjunkt út a kiindulási pontokból a rács szélének valamely m darab pontjába. A 26.11(a) ábra feladatán például létezik m darab ilyen út, a 26.11(b) ábrán pedig nem.

a. Tekintsünk egy olyan folyamhálózatot, amelyben nemcsak az éleknek, hanem a pontoknak is van kapacitása, ami azt jelenti, hogy az egyes pontokba befolyó pozitív folyamérték nem lehet nagyobb, mint a pont kapacitása. Mutassuk meg, hogy az ilyen feltételekkel meghatározandó maximális folyam megkeresése visszavezethető egy csak élkapacitásokkal rendelkező, hasonló méretű hálózatbeli maximális folyam megkeresésére.

b. Adjunk hatékony algoritmust a kijutási problémára. Határozzuk meg az algoritmus lépésszámát is.

26-2. Minimális útfedés

A $G = (V, E)$ irányított gráf ***útfedésének*** nevezzük pontdiszjunkt irányított utak egy P halmazát, ha a V halmaz minden egyes pontja a P pontosan egy útján szerepel. Megengedünk 0 hosszúságú utakat is, azaz olyan utakat, amelyeknek a kezdőpontja megegyezik a végpontjával, vagyis mindössze egy pontból állnak. A G ***minimális útfedése*** egy olyan útfedés, amely a lehető legkevesebb útból áll.

a. Adjunk hatékony algoritmust a $G = (V, E)$ körmentes irányított gráf minimális útfedésének a megkeresésére. (*Útmutatás.* Feltéve, hogy $V = \{1, 2, \ldots, n\}$, szerkesszük meg a következő $G' = (V', E')$ gráfot:

$$\begin{aligned} V' &= \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\}, \\ E' &= \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}), \end{aligned}$$

és futtassunk egy maximális folyam algoritmust.)

b. Működik-e algoritmusa olyan irányított gráfokra, amelyek köröket tartalmaznak algoritmusról? Indokolja válaszát.

26-3. Űrkísérletek

Spock professzor tanácsadóként a NASA-nál dolgozik. El kell döntenie, hogy egy tervezett űrkutatási sorozat során mely kísérleteket hajtsák végre, és hogy milyen műszereket vigyenek ehhez magukkal az egyes űrutazások alkalmával. Egy adott űrutazásra a NASA az $E_1, E_2, \ldots, E_m$ kísérlet végrehajtására kap megbízási ajánlatot. A megbízó az E_j kísérlet eredményéért p_j dollárt fizetne. A kísérletek elvégzéséhez műszerek $I = \{I_1, I_2, \ldots, I_n\}$ halmazára van szükség, mégpedig az E_j-hez pontosan az $R_j \subseteq I$ műszerekre. A NASA-nak c_k dollárba kerül az I_k műszert felvinni az űrbe és üzemeltetni. Spock professzor feladata kidolgozni egy olyan hatékony eljárást, amellyel kiválaszthatják azokat a kísérleteket, amelyek végrehajtásával a NASA jövedelme (azaz a kísérletekért kapott bevételből levonva a műszerek költségeit) a lehető legnagyobb. A tudomány most nem számít.

Tekintsük a következő $G = (V, E)$ hálózatot. $V = \{s, t, I_1, I_2, \ldots, I_n, E_1, E_2, \ldots, E_m\}$, és szokás szerint s a termelő, t a fogyasztó. Az (s, I_k) élek kapacitása legyen c_k minden $k = 1, 2, \ldots n$-re, az (E_j, t) élek kapacitása pedig legyen p_j minden $j = 1, 2, \ldots m$-re. Ha $I_k \in R_j$, akkor az (I_k, E_j) él kapacitása legyen végtelen. Minden más él kapacitása 0.

a. Mutassuk meg, hogy ha $E_j \in T$ teljesül valamely (S, T) véges értékű vágásra, akkor $I_k \in T$ minden $I_k \in R_j$-re.

b. Hogyan kell meghatározni a NASA lehető legmagasabb jövedelmét a G egy minimális vágásából és az adott p_j értékekből?

c. Adjunk hatékony algoritmust annak eldöntésére, hogy mely kísérleteket hajtsák végre, és mely műszereket vigyék fel az űrbe. Adjuk meg az algoritmus futási idejét a következő értékek függvényében kifejezve: $n, m, r = \sum_{j=1}^{m} |R_j|$.

26-4. Új maximális folyam

Legyen $G = (V, E)$ egy hálózat az s termelővel és t fogyasztóval, és a c egészértékű kapacitásfüggvénnyel. Továbbá legyen adva a G egy f maximális folyama.

a. Tegyük fel, hogy egy adott $(u, v) \in E$ él kapacitása 1-gyel megnő. Adjunk egy $O(V + E)$ idejű algoritmust az új maximális folyam megkeresésére.

b. Tegyük fel, hogy egy adott $(u, v) \in E$ él kapacitása 1-gyel csökken. Adjunk egy $O(V+E)$ idejű algoritmust az új maximális folyam megkeresésére.

26-5. Skálázási eljárás

Legyen $G = (V, E)$ egy hálózat az s termelővel és t fogyasztóval, és a c egészértékű kapacitásfüggvénnyel. Legyen $C = \max_{(u,v)\in E} c(u, v)$.

a. Vegyük észre, hogy a G egy minimális vágásának az értéke legfeljebb $C|E|$.

b. Mutassuk meg, hogy egy adott K értékre meg lehet keresni $O(E)$ időben egy legalább K kapacitású javítóutat feltéve, hogy létezik ilyen.

A Ford–Fulkerson-féle javítóutak módszerének az alábbi módosítása G-beli maximális folyamot eredményez.

MAX-FOLYAM-SKÁLÁZÁSSAL(G, s, t)

```
1 C ← max_{(u,v)∈E} c(u, v)
2 legyen az f kezdő folyam azonosan 0
3 K ← 2^⌊lg C⌋
4 while K ≥ 1
5       do while létezik p javítóút, amelynek kapacitása legalább K
6             do növeljük f-et p mentén
7          K ← K/2
8 return f
```

c. Mutassuk meg, hogy a MAX-FOLYAM-SKÁLÁZÁSSAL maximális folyamot ad.

d. Bizonyítsuk be, hogy a G_f reziduális gráf minimális vágásának értéke legfeljebb $2K|E|$, valahányszor a 4. sor végrehajtódik.

e. Lássuk be, hogy az 5–6. sorok **while** ciklusa minden egyes K értékre $O(E)$-szer kerül végrehajtásra.

f. Bizonyítsuk be, hogy az előzőek szerint a MAX-FOLYAM-SKÁLÁZÁSSAL(G, s, t) megvalósítható $O(E^2 \lg C)$ futási idővel.

26-6. Maximális folyam negatív kapacitásokkal

Tegyük fel, hogy a $G = (V, E)$ hálózatnak nemcsak pozitív kapacitású élei vannak, hanem negatív kapacitásúak is. Egy ilyen hálózatban nem feltétlenül létezik megengedett folyam.

a. Tekintsük a $G = (V, E)$ hálózat egy olyan (u, v) élét, melyre $c(u, v) < 0$. Milyen megszorítást jelent ez az u és v közötti folyamértékre vonatkozóan?

Legyen $G = (V, E)$ egy hálózat a c negatív értékeket is tartalmazó kapacitásfüggvénnyel, s és t pedig legyen a hálózat termelője, illetve fogyasztója. Készítsük el a következő közönséges folyamhálózatot a c' kapacitásfüggvénnyel, s' termelővel és t' fogyasztóval, ahol

$$V' = V \cup \{s', t'\},$$

és

$$\begin{aligned} E' &= E \cup \{(v, u) : (u, v) \in E\} \\ &\cup \{(s', v) : v \in V\} \\ &\cup \{(u, t') : u \in V\} \\ &\cup \{(s, t), (t, s)\}. \end{aligned}$$

Az $(u, v) \in E$ élek kapacitása legyen minden $(u, v) \in E$ élre

$$c'(u, v) = c'(v, u) = \frac{c(u, v) + c(v, u)}{2}.$$

Az $u \in V$ pontokra legyen

$$c'(s', u) = \max\left(0, \frac{c(V, u) - c(u, V)}{2}\right)$$

és

$$c'(u,t') = \max\left(0, \frac{c(u,V) - c(V,u)}{2}\right).$$

Legyen továbbá $c'(s,t) = c'(t,s) = \infty$.

b. Bizonyítsuk be, hogy ha G-ben létezik folyam a fenti feltételek mellett, akkor a G'-beli kapacitások mind nemnegatívak, és létezik olyan maximális folyam G'-ben, amelyre az összes t' fogyasztóba mutató élen a folyamérték a kapacitással egyenlő.

c. Bizonyítsuk be a (b) rész megfordítottját. Ha adott egy olyan G'-beli maximális folyam, amelynek az összes t'-be mutató élen a folyamértéke megegyezik a kapacitással, akkor hogyan kaphatunk ebből G-beli megengedett folyamot?

d. Adjunk algoritmust a G-beli maximális folyam megkeresésére. Jelöljük $MF(|V|,|E|)$-vel egy közönséges maximális folyam algoritmus legrosszabb futási idejét egy $|V|$ pontú, $|E|$ élű gráfon. Fejezzük ki a negatív kapacitások mellett is működő algoritmusunk futási idejét az MF értékkel.

26-7. A Hopcroft–Karp-féle párosítási algoritmus

Ebben a feladatban megadunk egy Hopcrofttól és Karptól származó algoritmust páros gráfbeli maximális párosítás megkeresésére. Az algoritmus futási ideje $O(\sqrt{V}E)$. Legyen $G = (V,E)$ egy páros gráf, és a ponthalmazának $V = L \cup R$ egy olyan partíciója, hogy minden élnek pontosan az egyik végpontja L-ben van. Legyen M a G egy párosítása. Azt mondjuk, hogy a G-beli P út az M-re vonatkozó ***javítóút,*** ha az L egy M által nem fedett pontjából indul, R egy M által nem fedett pontjában végződik, és felváltva $E - M$-beli és M-beli éleket tartalmaz. (A javítóutak ezen definíciója kapcsolódik a folyamokra bevezetett javítóút fogalmához, de különbözik tőle.) Ebben a szövegkörnyezetben egy utat az élei halmazának tekintünk, és nem a pontjai halmazának. Egy M-re vonatkozó legrövidebb javítóút egy olyan javítóút, amelynél nincs rövidebb javítóút.

Az A és B halmazok ***szimmetrikus differenciája*** $A \oplus B$ definíció szerint a következő halmaz: $(A - B) \cup (B - A)$, azaz azon elemek halmaza, amelyek az A és B halmazok közül pontosan az egyikben vannak benne.

a. Mutassuk meg, hogy ha M egy párosítás, P pedig egy M-re vonatkozó javítóút, akkor $M \oplus P$ párosítás, amelyre $|M \oplus P| = |M| + 1$. Mutassuk meg, ha $P_1, P_2, \ldots, P_k$ pontdiszjunkt javítóutak, akkor az $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ szimmetrikus differencia egy $|M| + k$ elemszámú párosítás.

Az algoritmus a következő.

HOPCROFT-KARP(G)

```
1  M ← ∅
2  repeat
3      legyen P ← {P_1 ∪ P_2 ∪ ··· ∪ P_k} M-re vonatkozó legrövidebb javítóutak
           maximális számú pontdiszjunkt útból álló rendszere
4      M ← M ⊕ (P_1 ∪ P_2 ∪ ··· ∪ P_k)
5      until P = ∅
6  return M
```

Ezen feladat fennmaradó része a megadott algoritmus iterációinak a számára kérdez, azaz a **repeat** ciklus lefutásainak a számára, továbbá a 3. sor megvalósításának a részleteire.

b. Mutassuk meg, hogy ha M és $M^\star$ G két párosítása, akkor a $G' = (V, M \oplus M^\star)$ gráfban minden csúcs fokszáma legalább 2. Ebből következtessünk arra, hogy G' egyszerű utak vagy körök diszjunkt uniója. Mutassuk meg, hogy minden ilyen egyszerű út vagy kör élei felváltva tartoznak M-hez, illetve $M^\star$-hoz. Bizonyítsuk be, hogy ha $|M| \leq |M^\star|$, akkor $M \oplus M^\star$ legalább $|M^\star| - |M|$ csúcsdiszjunkt javítóutat tartalmaz M-re nézve.

Legyen ℓ az M-re vonatkozó legrövidebb javítóút hossza, és legyen $P_1, P_2, \ldots, P_k$ az M-re vonatkozó, ℓ hosszúságú csúcsdiszjunkt javítóutak egy maximális halmaza. Legyen $M' = M \oplus (P_1 \cup \cdots \cup P_k)$, és tegyük fel, hogy P egy legrövidebb javítóút M'-re nézve.

c. Mutassuk meg, hogy ha P pontdiszjunkt a $P_1, P_2, \ldots, P_k$ utaktól, akkor több mint l darab élből áll.

d. Most tegyük fel, hogy P nem pontdiszjunkt a $P_1, P_2, \ldots P_k$ utaktól. Legyen A az $(M \oplus M') \oplus P$ élhalmaz. Mutassuk meg, hogy $A = (P_1 \cup P_2 \cup \ldots \cup P_k) \oplus P$, és $|A| \geq (k+1)l$. Lássuk be, hogy P több, mint l élből áll.

e. Bizonyítsuk be, hogy ha az M-re vonatkozó legrövidebb javítóút hossza l, akkor a maximális méretű párosítás legfeljebb $|M| + |V|/(l+1)$ elemű.

f. Mutassuk meg, hogy a fenti algoritmusban a **repeat** ciklus legfeljebb $2\sqrt{V}$-szer fut le. (*Útmutatás.* Milyen gyorsan nőhet M $\sqrt{V}$ iteráció után?)

g. Adjunk algoritmust, amely $O(E)$ futási időben megkeresi az adott M párosításra vonatkozó legrövidebb javítóutak egy maximális rendszerét. Ezt felhasználva mutassuk meg, hogy a Hopcroft-Karp(G) futási ideje $O(\sqrt{V}E)$

Megjegyzések a fejezethez

A hálózati folyamok és algoritmusainak kiváló összefoglalását adja Ahuja, Magnanti és Orlin [7], Even [87], Lawler [196], Papadimitriou és Steiglitz [237], valamint Tarjan [292]. Goldberg, Tardos és Tarjan [120] szintén nagyszerű áttekintést ad a hálózati folyamokra vonatkozó algoritmusokról, továbbá Schrijver [267] a hálózati folyamok elméletének fejlődését mutatja be.

A Ford–Fulkerson-algoritmus Fordtól és Fulkersontól származik [93], akik először kezdték el formálisan vizsgálni a hálózati folyamokkal kapcsolatos feladatokat, például a maximális folyam problémáját és a páros gráfokbeli maximális párosítás problémáját. A Ford–Fulkerson-algoritmus első implementációi szinte mind szélességi kereséssel keresték a javítóutakat. Edmonds és Karp [86] és tőlük függetlenül Dinic [76] bizonyította be, hogy ezen megvalósítás polinomiális futási idejű algoritmust eredményez. Egy ehhez kapcsolódó fogalom, a „blokkoló folyam” fogalmát szintén Dinic dolgozta ki [76]. Karzanovtól [176] származik az előfolyamok használatának ötlete. Az általános előfolyam-algoritmust Goldberg [117] és Goldberg és Tarjan [121] dolgozta ki. Goldberg és Tarjan megadott egy $O(V^3)$ futási idejű algoritmust is, ami a túlfolyó pontokat egy listában tárolja, továbbá egy dinamikus fák használatára épülő algoritmust, amely $O(VE \lg(V^2/E + 2))$ futási időt ér el. Számos más kutató dolgozott ki előfolyam-algoritmusokat. Ahuja és Orlin [9] és Ahuja, Orlin és

Tarjan [10] skálázást használó algoritmusokat adott. Cheriyan és Maheshwari [55] ajánlotta a maximális magasságú túlfolyó pontból történő pumpálásokat. Cheriyan és Hagerup [54] pedig a szomszédsági listák véletlenszerű permutálásával ért el eredményeket, azután számos más kutató [14, 178, 241] derandomizálta ezen ötletet, amely egy sor gyors algoritmust eredményezett. King, Rao és Tarjan [178] adta a leggyorsabb ilyen algoritmust, amelynek futási ideje $O(VE \log_{E/(V \lg V)} V)$.

Az aszimptotikusan leggyorsabb hálózati folyam algoritmust Goldberg és Rao [119] adta, melynek futási ideje $O(\min(V^{2/3}, E^{1/2})E \lg(V^2/E + 2) \lg C)$, algoritmusuk nem előfolyamokkal dolgozik, hanem blokkoló folyamokkal. A korábbi algoritmusok, beleértve az ebben a fejezetben bemutatott algoritmusokat, valamiféle távolságfüggvényt használtak (az előfolyam-algoritmusok esetében ezt magasságfüggvénynek neveztük), amelyben minden él hossza egységnyinek volt értendő. Ez az algoritmus egy másfajta megközelítési módot alkalmaz, amely szerint minden nagy kapacitású élnek 0 hosszúságot ad, és minden kis kapacitású élnek 1 hosszúságot. Ez nagyjából azt jelenti, hogy a termelőtől a fogyasztóba vezető legrövidebb utak nagy kapacitásúak lesznek, és összességében majd csak kevesebb iterációra lesz szükség.

A gyakorlatban az előfolyam-algoritmusok használhatóbbnak bizonyulnak a javítóutas vagy a lineáris programozási maximális folyam algoritmusoknál. Cherkassky és Goldberg kiemeli [56] két heurisztika fontosságát az előfolyam-algoritmusok esetében. Az első heurisztika szerint rendszeresen lefuttatnak a reziduális gráfon egy szélességi keresést, hogy minél használhatóbb magasságfüggvényt kapjanak. A második heurisztika pedig a 26.5-5. gyakorlatban szereplő rés heurisztika. Azt a következtetést vonják le, hogy a soron következő tehermentesítendő pont kiválasztására a legmegfelelőbb választás a legmagasabban lévő túlfolyó pont.

Páros gráf maximális párosításának a megkeresésére szolgáló algoritmusok közül a jelenleg leggyorsabb Hopcrofttól és Karptól [152] származik, amelynek futási ideje $O(\sqrt{V}E)$, ahogyan azt a 26-7. feladat megadja. Lovász és Plummer [207] könyve a párosítások elméletének kitűnő összefoglalása.

VII. VÁLOGATOTT FEJEZETEK

Bevezetés

Ebben a részben olyan algoritmusokat mutatunk be, amelyek kiszélesítik és kiegészítik a könyv eddig tárgyalt anyagát. Néhány fejezet olyan új kiszámítási modelleket tartalmaz, mint a kombinatorikus áramkörök és a párhuzamos számítógépek. Másokban olyan speciális területekkel foglalkozunk, mint a geometriai algoritmusok vagy a számelmélet. Az utolsó két fejezetben a hatékony algoritmusok tervezésével kapcsolatos ismert korlátokkal foglalkozunk, és módszereket ajánlunk ezeknek a kezelésére.

A 27. fejezet mutatja be első párhuzamos modellünket, az összehasonlító hálózatokat. Egyszerűen fogalmazva, az összehasonlító hálózat olyan algoritmus, amely lehet ővé teszi, hogy sok összehasonlítást végezzünk el egyidejűleg. Ebben a fejezetben megmutatjuk, hogy hogyan építhető olyan összehasonlító hálózat, amely n számot $O(\lg^2 n)$ idő alatt rendez.

A 28. fejezetben a mátrixokkal végzett hatékony műveleteket vizsgáljuk. Néhány alapművelettel kezdjük, majd rátérünk Strassen algoritmusára, amellyel két $n \times n$-es mátrix $O(n^{2,81})$ idő alatt összeszorozható. Azután két általános módszert mutatunk be – az LU és az LUP felbontást –, amelyekkel lineáris egyenletrendszereket oldhatunk meg Gauss-eliminációval $O(n^3)$ idő alatt. Azt is megmutatjuk, hogy a mátrixinverzió és mátrixszorzás ugyanolyan gyorsan végezhet ő el. A fejezet végén megmutatjuk, hogyan kaphatunk a legkisebb négyzetek módszerével közelít ő megoldást akkor, amikor a lineáris egyenletrendszernek nincs pontos megoldása.

A 29. fejezet a lineáris programozással foglalkozik, amelynek célja korlátozott er őforrások és adott megkötések esetében adott célfüggvény legnagyobb vagy legkisebb értékét kiszámítani. A lineáris programozás a gyakorlati élet számtalan területén megjelenik. Jelen fejezet a feladat megfogalmazását és megoldását mutatja be. A megoldási módszer a szimplex módszer, a lineáris programozás legrégebbi algoritmusa. Ellentétben a könyv legtöbb algoritmusával, a szimplex módszer legrosszabb esetben nem polinom idejű, de ennek ellenére igen hatékony és a gyakorlatban széles körben használt.

A 30. fejezetben a polinomokkal végezhet ő műveleteket tanulmányozzuk, és megmutatjuk, hogy a jól ismert jelfeldolgozó módszer – a gyors Fourier-transzformáció (FFT) – alkalmas két n-edfokú polinom $O(n \log n)$ id ő alatt történő összeszorzására. Megvizsgáljuk az FFT egy hatékony – párhuzamos áramköröket is magában foglaló – megvalósítását is.

A 31. fejezetben számelméleti algoritmusokat mutatunk be. A számelmélet elemeinek áttekintése után bemutatjuk Euklidész algoritmusát a legnagyobb közös osztó kiszámítására. Ezután lineáris kongruenciarendszereket megoldó, valamint számokat modulo n hatványozó algoritmusokat tárgyalunk. A számelméleti algoritmusok érdekes alkalmazása az

RSA nyilvános kulcsú titkosírás. Ez a titkosírás nemcsak üzeneteknek a kívülálló számára olvashatatlanná tételére alkalmas, hanem digitális aláírások előállítására is. Ezután Miller–Rabin véletlenített prímtesztjét mutatjuk be, amely alkalmazható nagy prímszámok hatékony előállítására – ami az RSA rendszer számára fontos követelmény. A fejezet végén bemutatjuk Pollard prímtényezőkre bontó heurisztikus algoritmusát, a „ρ”-t, és áttekintést adunk az egész számok prímtényezőkre bontásának problémaköréről.

A 32. fejezetben egy, a szövegszerkesztő programokban gyakran előforduló problémával foglalkozunk: hogyan lehet egy szövegben megtalálni adott minta előfordulásait. Először egy, Rabintól és Karptól származó, elegáns megoldást ismertetünk. Ezután – a véges automatákon alapuló, hatékony megoldást vizsgálva – bemutatjuk a Knuth–Morris–Pratt-algoritmust, amely a minta okos elő-feldolgozásával biztosítja a hatékonyságot.

A 33. fejezet témája a számítógépes geometria. Az alapfogalmak áttekintése után ebben a fejezetben megmutatjuk, hogyan lehet egy „söprő” algoritmussal hatékonyan eldönteni, vajon egy szakaszhalmaz tartalmaz-e metsző szakaszpárt. Ponthalmaz konvex burkának meghatározására alkalmas két okos algoritmus – Graham, illetve Jarvis algoritmusa – is illusztrálja a söprő módszerek erejét. A fejezet végén egy síkbeli ponthalmaz legközelebbi pontpárjának meghatározására alkalmas hatékony algoritmust mutatunk be.

A 34. fejezetben az NP-teljes problémákat vizsgáljuk. Sok érdekes kiszámítási probléma NP-teljes, és egyikük megoldására sem ismerünk polinomiális algoritmust. Ebben a fejezetben olyas módszereket ismertetünk, amelyekkel eldönthető, hogy egy probléma NP-teljes-e. Több klasszikus problémáról bebizonyítjuk, hogy azok NP-teljesek: van-e egy adott gráfban Hamilton-kör, kielégíthető-e egy adott logikai formula, van-e adott számhalmaznak adott összegű részhalmaza? Azt is bebizonyítjuk, hogy a híres utazóügynök-probléma NP-teljes.

A 35. fejezetben megmutatjuk, hogyan alkalmazhatók hatékonyan a közelítő algoritmusok NP-teljes problémák közelítő megoldásainak előállítására. Bizonyos NP-teljes problémákhoz viszonylag könnyű az optimálishoz közeli megoldásokat előállítani, míg másokhoz a legjobb közelítő algoritmusok is csak romló minőségű megoldásokat eredményeznek, ha a probléma mérete nő. Aztán vannak olyan problémák, amelyeknél a kiszámításra fordított idő növelésével egyre pontosabb megoldást kaphatunk. Ezeket a lehetőségeket a minimális lefedő csúcshalmaz (irányított és nem irányított gráfokban), az utazóügynök, minimális halmazlefedés és a részösszeg problémák vizsgálata során elemezzük.

27. Rendező hálózatok

A II. részben a rendezési algoritmusokat közvetlen elérésű számítógépeken (RAM) vizsgáltuk, amelyek egy adott pillanatban csak egyetlen művelet végzésére képesek. Ebben a fejezetben azokat a rendezési algoritmusokat mutatjuk be, amelyek olyan összehasonlító hálózati modellt használnak, amely egyszerre több összehasonlításra képes.

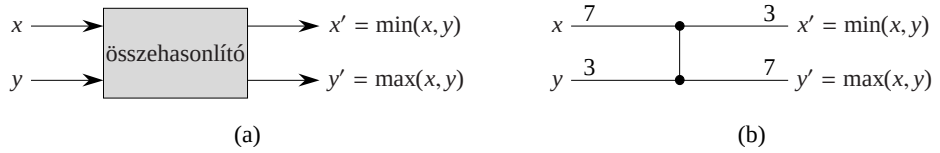
Az összehasonlító hálózatok két alapvető dologban különböznek a RAM-gépektől. Először, csak összehasonlításra képesek. Így egy olyan algoritmus, mint például a leszámláló rendezés (lásd 8.2. alfejezet) nem valósítható meg összehasonlító hálózatokkal. Másodsorban, ellentétben a RAM-gépekkel, ahol a műveletek sorosan következnek – azaz egymás után – az összehasonlító hálózatokban egy időben „párhuzamosan" több művelet is végezhető. Látni fogjuk, hogy ez a tulajdonság lehetővé teszi olyan összehasonlító hálózatok készítését, amelyek n értéket szublineáris időben rendeznek.

A 27.1. alfejezet az összehasonlító hálózatok és a rendező hálózatok meghatározásával foglalkozik. Ugyanakkor megadjuk az összehasonlító hálózat „futási idejének" egy természetes definícióját a hálózat mélységének függvényében. A 27.2. alfejezetben bebizonyítjuk a „nulla-egy elvet", amely nagyban megkönnyíti a rendező hálózatok helyességének a vizsgálatát.

A hatékony rendező hálózat, amelyet bemutatunk, tulajdonképpen a 2.3.1. pontbeli összefésülő rendezésnek a párhuzamos változata. A hálózat felépítése három lépésben történik. A 27.3. alfejezetben leírjuk a „biton" rendezőt, amely az alapvető építőkövünk. A 27.4. alfejezetben ezt úgy módosítjuk, hogy összefésülő hálózat legyen, amely két rendezett sorozatot egy rendezett sorozattá alakít. Végül, a 27.5. alfejezetben ezeket az összefésülő hálózatokat egy olyan rendező hálózattá szervezzük, amely n értéket $O(\lg^2 n)$ időben rendez.

27.1. Összehasonlító hálózatok

A rendező hálózatok olyan összehasonlító hálózatok, amelyek mindig rendezik a bemenetüket, ezért először az összehasonlító hálózatokat és azok tulajdonságait mutatjuk be. Egy összehasonlító hálózat csak összehasonlító egységekből és ezeket összekötő vezetékekből (vonalakból) áll. Az *összehasonlító egység* vagy röviden ***összehasonlító*** (27.1(a) ábra) két bemenetű (x és y) és két kimenetű (x' és y') berendezés, amely a következőket valósítja meg:



27.1. ábra. (a) Egy x és y bemenetű, x' és y' kimenetű összehasonlító. **(b)** Ugyanaz az összehasonlító egyetlen függőleges szakasszal ábrázolva. Bemenetek: $x = 7, y = 3$, kimenetek: $x' = 3, y' = 7$.

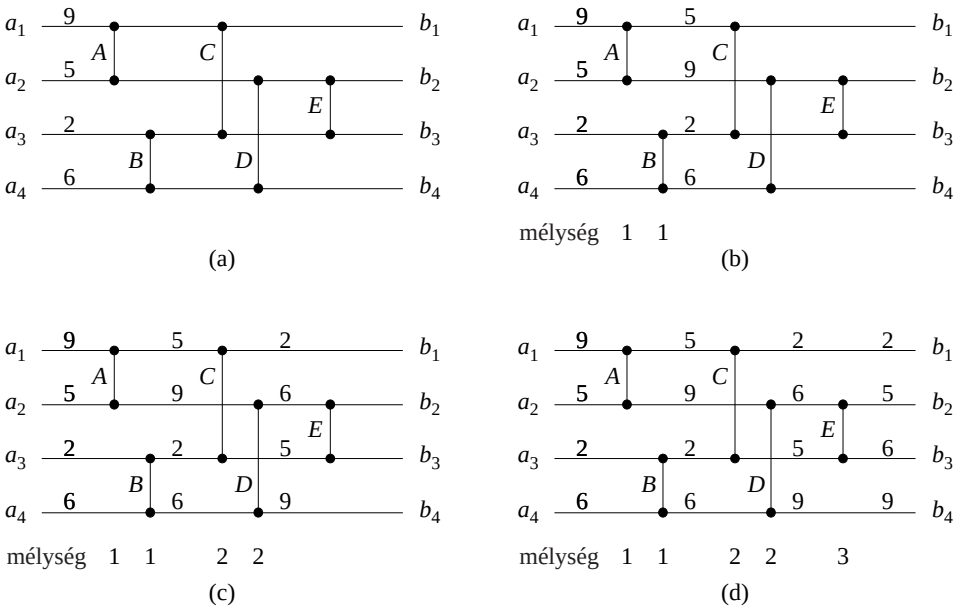
$$x' = \min(x, y),$$
$$y' = \max(x, y).$$

A 27.1(a) ábrán látható rajz helyett egy egyszerűbbet fogunk használni; az összehasonlítót egy függőleges szakasszal helyettesítjük, amint az a 27.1(b) ábrán látható. A bemenetek a bal oldalon, míg a kimenetek a jobb oldalon helyezkednek el, a kisebbik érték a felső, a nagyobbik pedig az alsó sorban. Az összehasonlító tehát úgy értelmezhető, hogy rendezi a két bemeneti elemét.

Feltételezzük, hogy minden összehasonlító egység $O(1)$ időben végzi el a két bemeneti érték rendezését. Más szóval, az x és y bemeneti értékek megjelenése és az x' és y' kimeneti értékek előállítása közötti időtartam állandó.

A ***vezeték*** értéket továbbít egyik helyről a másikra. A vezetékek összeköthetik egyik összehasonlító egység kimenetét egy másik bemenetével; ha nem, akkor vagy bemeneti, vagy kimeneti vezetékek. Ebben a fejezetben mindvégig feltételezzük, hogy egy összehasonlító hálózatnak van $a_1, a_2, \ldots, a_n$-nel jelölt, n darab ***bemeneti vezetéke,*** amelyeken a rendezendő értékek bejutnak a hálózatba, és $b_1, b_2, \ldots, b_n$-nel jelölt, n darab ***kimeneti vezetéke,*** amelyek az eredményt szolgáltatják. Hasonlóképpen, beszélhetünk az $\langle a_1, a_2, \ldots, a_n \rangle$ ***bemeneti sorozatról*** és a $\langle b_1, b_2, \ldots, b_n \rangle$ ***kimeneti sorozatról,*** amelyek a hálózat bemeneti, illetve kimeneti értékeit jelentik. Ugyanazzal a névvel jelöljük a vezetéket és a vezeték értékét. Ez nem zavaró, mert a szövegkörnyezetből mindig kiderül, hogy melyikről van szó.

A 27.2. ábrán egy ***összehasonlító hálózat*** látható, amely vezetékekkel összekapcsolt összehasonlító egységekből áll. Egy n bemenetű összehasonlító hálózatot n vízszintes ***vonallal*** ábrázolunk, ahol az összehasonlító egységeket függőleges szakaszok helyettesítik. Egy vízszintes vonal tulajdonképpen nem egy vezetéknek felel meg, hanem összehasonlító egységeket összekötő vezetékeknek. A 27.2. ábra felső vonala például három vezetéket jelent: az a_1 bemeneti vezetékét, amely az A összehasonlító egyik bemenete; egy másik vezetéket, amely összeköti az A összehasonlító felső kimenetét a C összehasonlító egyik bemenetével; és a b_1 kimeneti vezetéket, amelyik a C összehasonlító felső kimenete. Bármely összehasonlító bemenete vagy az n darab $\langle a_1, a_2, \ldots, a_n \rangle$ bemenet valamelyike vagy egy másik összehasonlító kimenete. Hasonlóképpen, bármely összehasonlító kimenete vagy az n darab $\langle b_1, b_2, \ldots, b_n \rangle$ kimenet valamelyike vagy egy másik összehasonlító bemenete. A legfontosabb követelmény az összehasonlító egységek összekapcsolásánál, hogy az összekapcsolási gráf körmentes legyen, azaz, ha egy adott összehasonlító kimenetétől elindulva eljutunk egy másik összehasonlító bemenetéhez, egyéb kimeneteken és bemeneteken keresztül, akkor onnan sohasem juthatunk vissza az eredeti összehasonlítóhoz, vagyis sohasem érinthetjük kétszer ugyanazt az összehasonlítót. Tehát, ahogy a 27.2. ábra is mutatja, az összehasonlító hálózatot lerajzolhatjuk úgy, hogy a hálózat bemenete a bal oldalon, a kimenete a jobb oldalon van, az adatok pedig balról jobbra mozognak.



27.2. ábra. (a) Egy 4 bemenetű, 4 kimenetű összehasonlító hálózat, amely valójában rendező hálózat. A 0. időpontban a bejelölt értékek a négy bemeneti vezetéken találhatók. **(b)** Az 1. időpontban a bejelölt értékek az *A* és *B* összehasonlítók kimeneti vezetékein vannak, az összehasonlítók 1 mélységűek. **(c)** A 2. időpontban a bejelölt értékek a 2 mélységű *C* és *D* összehasonlítók kimenetein vannak. A b_1 és b_4 kimeneti vezetékek már a végső értéket tartalmazzák, a b_2 és b_3 azonban még nem. **(d)** A 3. időpontban a bejelölt értékek a 3 mélységű *E* összehasonlító kimeneti vezetékein vannak. A b_2 és b_3 kimeneti vezetékek most már a végső értékeket tartalmazzák.

Minden összehasonlító egység csak akkor szolgáltat kimenetet, ha mindkét bemeneti értéke rendelkezésre áll. Például, a 27.2(a) ábra esetében tételezzük fel, hogy a hálózat bemenete a 0. időpontban $\langle 9, 5, 2, 6\rangle$. Ekkor, a 0. időpontban csak az *A* és *B* összehasonlítóknak van előállítva mindkét bemenete. Ha feltételezzük, hogy mindegyik összehasonlító egységnyi idő alatt állítja elő a kimenetét (futási ideje 1), az *A* és *B* összehasonlítók kimenete az 1. időpontban jelenik meg, amint azt a 27.2(b) ábra mutatja. Figyeljük meg, hogy az *A* és *B* összehasonlítók ugyanabban az időben, azaz „párhuzamosan” dolgoznak. Az 1. időpontban a *C* és *D* összehasonlítóknak előállt mindkét bemenete, az *E*-nek viszont még nem. Egy időegységgel később ez a két összehasonlító előállítja kimenetét, ahogy azt a 27.2(c) ábra mutatja. Természetesen *C* és *D* szintén párhuzamosan dolgoznak. Mivel a *C* összehasonlító felső kimenete egybeesik b_1-gyel és a *D* alsó kimenete b_4-gyel, ezek a kimenetek már a 2. időpontban a hálózat végső értékei. Az *E* összehasonlító, miután a 2. időpontban rendelkezésére áll mindkét bemenete, a 3. időpontban előállítja kimenetét (27.2(d) ábra). E két utóbbi érték a hálózat b_2 és b_3 kimeneti értéke, és ezzel előáll a $\langle 2, 5, 6, 9\rangle$ kimeneti sorozat.

Ha feltételezzük, hogy minden összehasonlító futási ideje egy egység, akkor könnyen definiálhatjuk az összehasonlító hálózat „futási idejét” mint azt az időt, amely ahhoz szükséges, hogy a hálózat minden kimeneti vezetékén megjelenjen az eredmény, miután megkapta összes bemenetét. Ez a futási idő egyenlő azon összehasonlítók legnagyobb számával, amelyeken egy bemeneti adat a hálózat egy bemeneti vezetékétől egy kimeneti vezetékéig áthalad. A pontosabb meghatározásért bevezetjük a vezetékek ***mélységének*** a fogalmát. A hálózat egy bemeneti vezetékének a mélysége 0. Ha egy összehasonlító bemeneti ve-

zetékeinek a mélysége d_x és d_y, akkor kimeneti vezetékeinek mélysége: $\max(d_x, d_y) + 1$. Mivel az összehasonlító hálózatok körmentesek, a vezeték mélysége jól meghatározott. Egy összehasonlító mélysége nem más, mint a kimeneti vezetékeinek a mélysége. A 27.2. ábrán példát láthatunk az összehasonlítók mélységére. Egy összehasonlító hálózat mélységének a kimeneti vezetékek legnagyobb mélységét nevezzük, vagy más szóval a hálózatot képező összehasonlítók legnagyobb mélységét. A 27.2. ábrán látható hálózat mélysége 3, mert az E összehasonlító egység mélysége 3. Ha mindegyik összehasonlító egységnyi idő alatt állítja elő a kimenetét, és a bemenetek a 0. időpontban jelennek meg, akkor egy d mélységű összehasonlító a d. időpontban állítja elő a kimeneteit; így a hálózat mélysége egyenlő azzal az idővel, amely ahhoz szükséges, hogy az összes kimenetét előállítsa.

Egy ***rendező hálózat*** olyan összehasonlító hálózat, amelynek kimeneti sorozata monoton növekvő (azaz $b_1 \leq b_2 \leq \ldots \leq b_n$) *bármely* bemenetre. Természetesen nem minden összehasonlító hálózat rendező hálózat, de a 27.2. ábrán látható az. Hogy ezt beláthassuk, figyeljük meg, hogy az 1. időpont után a négy bemeneti érték minimuma vagy az A vagy a B összehasonlító felső kimeneti vezetékén jelenik meg. A 2. időpont után a minimum a C összehasonlító felső kimenete. Hasonlóképpen, a szimmetria miatt könnyen belátható, hogy a négy bemeneti érték maximuma a D alsó kimenete a 2. időpont után. Az E-nek már csak azt kell biztosítania, hogy a két középső érték megfelelő sorrendbe kerüljön, ez a 3. időpontban meg is történik.

Az összehasonlító hálózat olyan eljáráshoz hasonlítható, amely megadja, hogyan kell összehasonlítani az elemeket, de nem szerencsés az olyan eljárás, amelynek hossza – a benne lévő összehasonlítók száma – függ a bemenetek és kimenetek számától. Ezért az összehasonlító hálózatok „családjait” fogjuk leírni. Jelen fejezetben, például hatékony rendező hálózatok RENDEZ nevű családját. A család egy hálózatára a család nevével, valamint az adott hálózat bemeneti elemeinek számával (amely megegyezik a kimeneti elemek számával) fogunk hivatkozni. Így a RENDEZ család egy n bemenetű, n kimenetű rendező hálózata: RENDEZ[n].

Gyakorlatok

27.1-1. Milyen értékek jelennek meg a 27.2. ábra hálózatának minden vezetékén, ha a bemenet $\langle 9, 6, 5, 2 \rangle$?

27.1-2. Legyen n 2-nek hatványa. Mutassuk meg, hogyan készíthető egy olyan n bemenetű, n kimenetű, $\lg n$ mélységű összehasonlító hálózat, amelyben a legkisebb elem mindig a legfelső kimeneti, a legnagyobb pedig mindig a legalsó kimeneti vezetékén van.

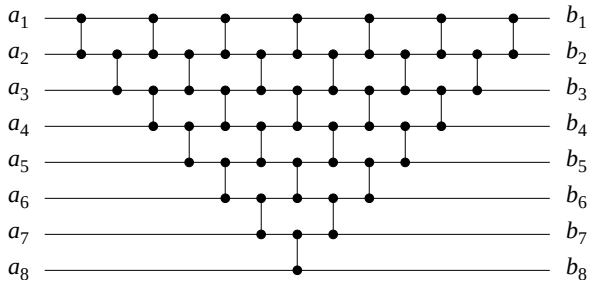
27.1-3. Egy rendező hálózatba mindig beilleszthetünk egy összehasonlítót úgy, hogy az eredményül kapott hálózat ne legyen rendező hálózat. Mutassuk meg, hogyan illeszthetünk be egy összehasonlítót a 27.2. ábra hálózatába úgy, hogy az ne rendezze a bemenet bármely permutációját.

27.1-4. Bizonyítsuk be, hogy bármely n bemenetű rendező hálózat mélysége legalább $\lg n$.

27.1-5. Bizonyítsuk be, hogy bármely n bemenetű rendező hálózatban az összehasonlítók száma $\Omega(n \lg n)$.

27.1-6. Tekintsük a 27.3. ábrán látható összehasonlító hálózatot. Bizonyítsuk be, hogy az valójában rendező hálózat, majd hasonlítsuk össze a beszúró rendezés szerkezetével (*lásd* 11. alfejezetet).

27.1-7. Egy c összehasonlítóból álló n bemenetű összehasonlító hálózatot ábrázolhatunk egy c számpárból álló listával (a számok 1 és n közé eső egészek). Ha két számpár tar-



27.3. ábra. Beszúró rendezésen alapuló rendező hálózat, melyet a 27.1-6. gyakorlat használ.

talmaz egy közös elemet, a megfelelő összehasonlítók hálózatbeli sorrendjét a számpárok listabeli sorrendje határozza meg. Ennek az ábrázolásnak a segítségével adjunk meg egy olyan $O(n + c)$ idejű (soros) algoritmust, amely meghatározza az összehasonlító hálózat mélységét.

27.1-8.$\star$ Tételezzük fel, hogy a szokásos összehasonlítók mellett „fordított” összehasonlítókat is használunk, amelyek a kisebbik értéket az alsó, míg a nagyobbikat a felső kimeneti vezetékükön állítják elő. Mutassuk meg, hogyan alakítható át egy összesen c szokásos és fordított összehasonlítót használó rendező hálózat egy c szokásos összehasonlítót tartalmazó hálózattá. Bizonyítsuk be, hogy az átalakító módszerünk helyes.

27.2. A nulla-egy elv

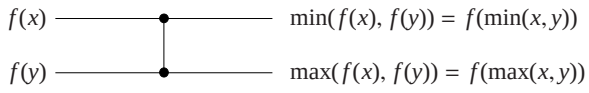
A ***nulla-egy elv*** kimondja, hogy ha egy rendező hálózat helyesen működik abban az esetben, amikor minden bemenete 0 vagy 1, akkor helyesen működik bármilyen bemeneti számokra is (a számok lehetnek egészek, valósak vagy bármely lineárisan rendezett halmaz elemei). Amikor rendező hálózatot vagy más összehasonlító hálózatot készítünk, a nulla-egy elv lehetővé teszi, hogy csak a 0 és 1 bemeneti értékekkel dolgozó műveletekre összpontosítsunk. Ha van egy rendező hálózatunk, és bebizonyítottuk, hogy helyesen rendez minden nulla-egy sorozatot, akkor a nulla-egy elv alapján állíthatjuk, hogy bármely sorozatot is helyesen rendez.

A nulla-egy elv bizonyítása a monoton növekvő függvények fogalmán alapszik (3.2. alfejezet).

27.1. lemma. *Ha egy összehasonlító hálózat az $a = \langle a_1, a_2, \ldots, a_n \rangle$ bemeneti sorozatot a $b = \langle b_1, b_2, \ldots, b_n \rangle$ kimeneti sorozattá alakítja, akkor tetszőleges monoton növekvő f függvényre a hálózat az $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ bemeneti sorozatot az $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$ kimeneti sorozattá alakítja.*

Bizonyítás. Először bebizonyítjuk, hogy ha f monoton növekvő függvény, akkor egy $f(x)$ és $f(y)$ bemenetű összehasonlító eredményül az $f(\min(x, y))$ és $f(\max(x, y))$ értékeket adja. Ezután indukcióval bizonyítjuk a lemmát.

Hogy bebizonyítsuk első állításunkat, tekintsünk egy x és y bemenetű összehasonlítót. Az összehasonlító felső kimenete $\min(x, y)$, alsó kimenete pedig $\max(x, y)$. Tegyük fel ezután, hogy a bemenet $f(x)$ és $f(y)$, ahogy azt a 27.4. ábra mutatja. Az összehasonlító ekkor



27.4. ábra. A 27.1. lemma bizonyításában használt összehasonlító művelet. Az f függvény monoton növekvő.

eredményül a felső kimenetén a $\min(f(x), f(y))$, alsó kimenetén pedig a $\max(f(x), f(y))$ értékeket szolgáltatja. Mivel f monoton növekvő függvény, ha $x \leq y$, akkor $f(x) \leq f(y)$. Tehát, a következőket kapjuk:

$$\begin{aligned} \min(f(x), f(y)) &= f(\min(x,y)), \\ \max(f(x), f(y)) &= f(\max(x,y)). \end{aligned}$$

Tehát, az összehasonlító az $f(\min(x,y))$ és $f(\max(x,y))$ értékeket adja kimenetként, ha a bemenet $f(x)$ és $f(y)$, és ezzel bebizonyítottuk állításunkat.

Indukciót alkalmazva egy általános összehasonlító hálózat bármely vezetékének a mélységére, a lemma kijelentésénél erősebb állítást is bizonyíthatunk: ha egy vezeték értéke a_i, amikor a bemeneti sorozat a, akkor értéke $f(a_i)$ lesz, ha a bemeneti sorozat $f(a)$. Mivel ez az állítás a kimeneti vezetékekre is igaz, ennek bizonyításával a lemmát is bizonyítjuk.

Először tekintsünk egy 0 mélységű, azaz bemeneti vezetéket, legyen ez a_i. Az állítás ekkor triviális: ha a hálózat bemenete $f(a)$, akkor a megfelelő bemenet értéke $f(a_i)$. Tekintsünk most egy d mélységű vezetéket ($d \geq 1$). Ez egy d mélységű összehasonlító kimeneti vonala, amelynek bemeneti vezetékei d-nél szigorúan kisebb mélységűek. Indukciós feltevésünk alapján, ha az összehasonlító bemeneti értékei a_i és a_j, amikor a hálózat bemenete a, akkor a bemeneti értékek $f(a_i)$ és $f(a_j)$ lesznek, ha a hálózat bemenete $f(a)$. Előbbi állításunk alapján ennek az összehasonlítónak a kimenete $f(\min(a_i, a_j))$ és $f(\max(a_i, a_j))$ lesz. Mivel ezen vezetékek értéke $\min(a_i, a_j)$ és $\max(a_i, a_j)$, ha a bemeneti sorozat a, a lemma bizonyítását befejeztük. ■

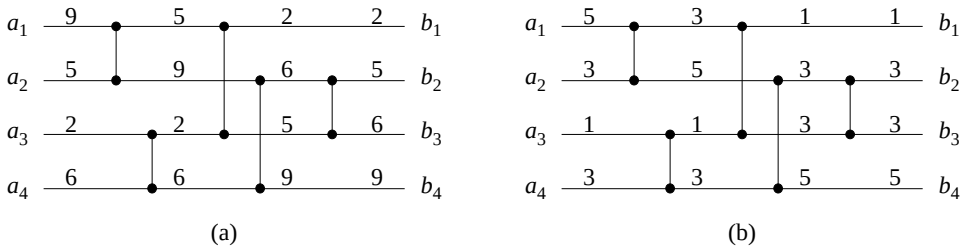
A 27.5. ábra példa a lemma alkalmazására: rendező hálózatként a 27.2. ábrán látható hálózatot vesszük, amelynek bemenetére alkalmazzuk az $f(x) = \lceil x/2 \rceil$ függvényt. Minden vezeték értéke egyenlő a 27.2. ábrán lévő megfelelő vezetékre alkalmazott f értékkel.

Ha az összehasonlító hálózat rendező hálózat, a 27.1. lemma segítségével bebizonyíthatjuk a következő jelentős eredményt.

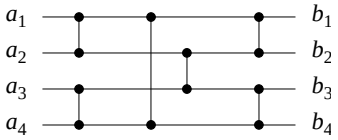
27.2. tétel (nulla-egy elv). *Ha egy n bemenetű összehasonlító hálózat helyesen rendezi mind a 2^n számú csak* 0*-ból és* 1*-ből álló sorozatot, akkor helyesen rendez bármilyen számsorozatot.*

Bizonyítás. Tételezzük fel, hogy a hálózat az összes csak 0-ból és 1-ből álló sorozatot helyesen rendezi, ellenben létezik egy tetszőleges számokból álló sorozat, amelyet nem rendez helyesen. Vagyis, létezik egy $\langle a_1, a_2, \ldots, a_n \rangle$ bemeneti sorozat, amelynek a_i és a_j elemére fennáll, hogy $a_i < a_j$, a hálózat az a_j-t mégis az a_i elé helyezi a kimeneti sorban. Definiáljuk az f monoton növekvő függvényt:

$$f(x) = \begin{cases} 0, & \text{ha } x \leq a_i, \\ 1, & \text{ha } x > a_i. \end{cases}$$



27.5. ábra. (a) A 27.2. ábra rendező hálózata a $\langle 9, 5, 2, 6\rangle$ bemeneti sorozatra. **(b)** Ugyanaz a rendező hálózat a bemenetre alkalmazott $f(x) = \lceil x/2 \rceil$ monoton növekvő függvénnyel. Minden vezeték értéke egyenlő az f függvénynek az (a) pontbeli hálózat megfelelő vezetékén vett értékével.



27.6. ábra. Négy szám rendezésére szolgáló rendező hálózat.

Ha a bemenet $\langle a_1, a_2, \ldots, a_n\rangle$, akkor a hálózat a kimenetben a_j-t a_i elé helyezi, és ezért, a 27.1. lemma alapján $f(a_j)$-t is $f(a_i)$ elé helyezi, ha a bemeneti sorozat $\langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$. De $f(a_j) = 1$ és $f(a_i) = 0$, ezért ellentmondáshoz jutunk: a hálózat nem rendezi helyesen a csupa 0-ból és 1-ből álló $\langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$ sorozatot. ■

Gyakorlatok

27.2-1. Bizonyítsuk be, hogy ha egy monoton növekvő függvényt alkalmazunk egy rendezett sorozatra, az eredményül kapott sorozat is rendezett lesz.

27.2-2. Bizonyítsuk be, hogy egy n bemenetű összehasonlító hálózat pontosan akkor rendezi helyesen az $\langle n, n-1, \ldots, 1\rangle$ bemeneti sorozatot, ha helyesen rendezi az $n-1$ darab $\langle 1, 0, 0, \ldots, 0, 0\rangle$, $\langle 1, 1, 0, \ldots, 0, 0\rangle$, $\ldots$, $\langle 1, 1, 1 \ldots, 1, 0\rangle$ csupa 0-ból és 1-ből álló sorozatokat.

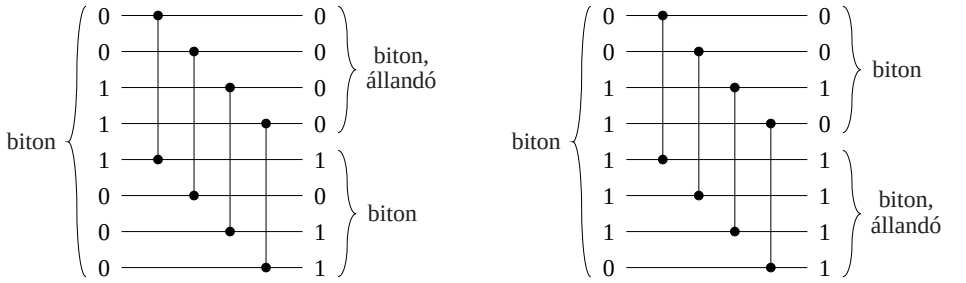
27.2-3. Felhasználva a nulla-egy elvet, bizonyítsuk be, hogy a 27.6. ábrán látható összehasonlító hálózat rendező hálózat.

27.2-4. Jelentsünk ki a nulla-egy elvhez hasonló elvet a döntési-fa-modellre, majd bizonyítsuk is be. (*Útmutatás.* Vigyázzunk az egyenlőség helyes kezelésére.)

27.2-5. Bizonyítsuk be, hogy egy n bemenetű rendező hálózatnak minden i-edik és $(i+1)$-edik vezetéke között legalább egy összehasonlítónak lennie kell ($i = 1, 2, \ldots, n-1$).

27.3. Biton sorozatokat rendező hálózat

A hatékony rendező hálózat megtervezésében első lépésünk egy olyan összehasonlító hálózat készítése, amely ***biton sorozatokat*** rendez. Ezek olyan sorozatok, amelyek monoton növekvők majd monoton csökkenők vagy ciklikus eltolással ilyenekké alakíthatók. Például az $\langle 1, 4, 6, 8, 4, 2\rangle$, $\langle 6, 9, 4, 2, 3, 5\rangle$ és a $\langle 9, 8, 3, 2, 4, 6\rangle$ sorozatok mindegyike biton sorozat. Határesetként azt mondhatjuk, hogy minden, egy vagy két számból álló sorozat, biton sorozat. A csupa 0-ból és 1-ből álló biton sorozatok szerkezete nagyon egyszerű. Ezek $0^i1^j0^k$



27.7. ábra. A FÉLIG-EGYENGET[8] összehasonlító hálózat. Két különböző nulla-egy példabemenet és a megfelelő kimenetek láthatók. A bemenetről feltételezzük, hogy biton. A félig-egyengető biztosítja, hogy a kimenet felső felének egyetlen eleme sem nagyobb, mint a kimenet alsó felének bármely eleme. Ezenkívül, a kimenet mindegyik fele biton, és legalább az egyik állandó.

vagy $1^i0^j1^k$ alakúak, valamely $i, j, k \geq 0$ értékekre. Figyeljük meg, hogy a monoton növekvő vagy monoton csökkenő sorozatok szintén biton sorozatok.

A biton rendező, amelyet a következőkben írunk le, olyan összehasonlító hálózat, amely 0-ból és 1-ből álló biton sorozatokat rendez. A 27.3-6. gyakorlat annak bizonyítását kéri, hogy a biton rendező tetszőleges biton sorozatot is helyesen rendez.

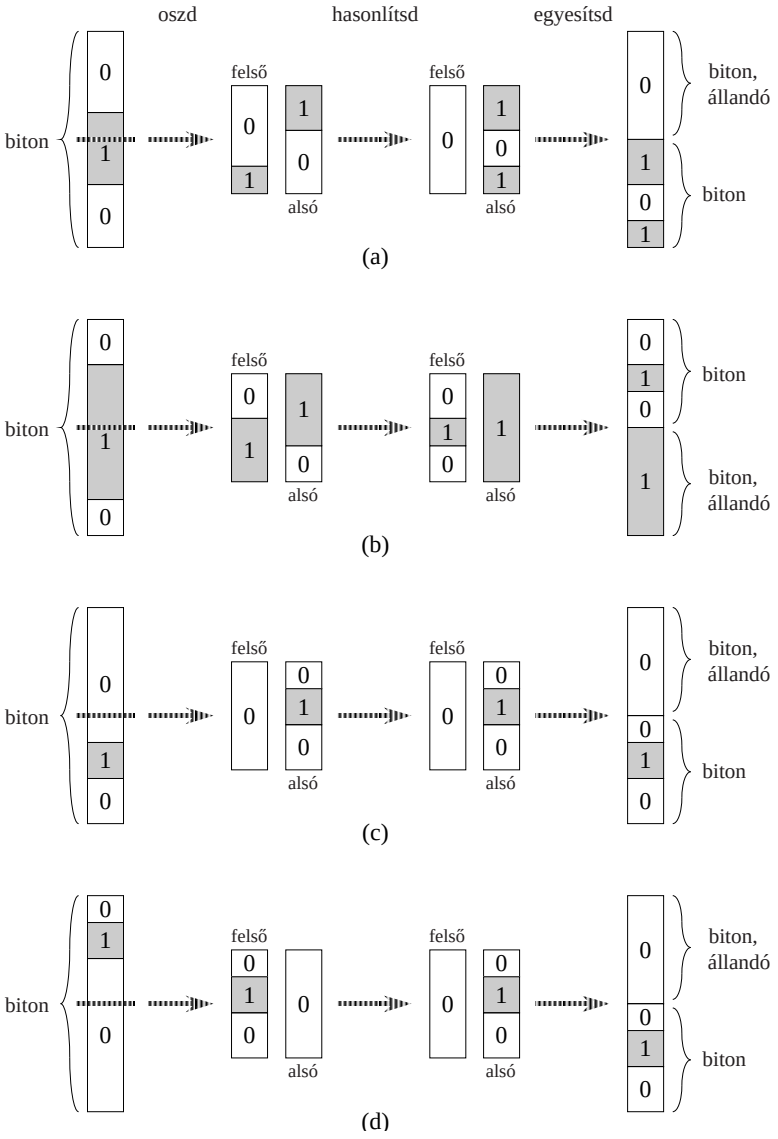
A félig-egyengető

A biton rendező több ***félig-egyengetőnek*** nevezett részből áll. Mindegyik félig-egyengető egy olyan 1 mélységű összehasonlító hálózat, amely az i-edik bemeneti értéket az $(i + n/2)$. bemeneti értékkel hasonlítja össze minden $i = 1, 2, \ldots, n/2$ értékre. (Feltételezzük, hogy n páros.) A 27.7. ábrán a FÉLIG-EGYENGET[8] 8 bemenetű és 8 kimenetű félig-egyengető látható.

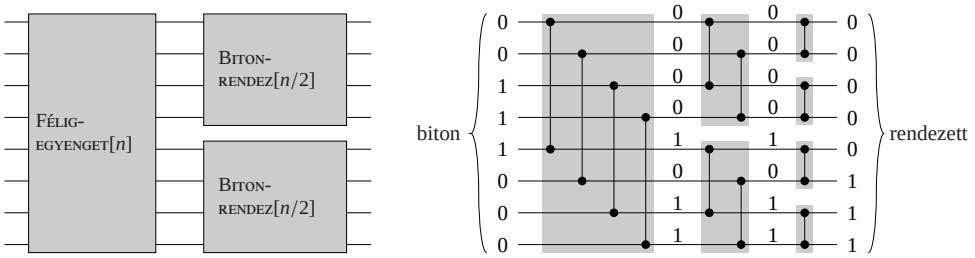
Amikor egy félig-egyengető bemenete egy csupa 0-ból és 1-ből álló biton sorozat, a kimeneti sorozat felső felébe kerülnek a kisebb és alsó felébe a nagyobb értékek, és mindkét fele biton sorozat. Tulajdonképpen a kimeneti sorozatnak legalább az egyik fele ***állandó*** – vagy csak csupa 0-ból áll, vagy csak csupa 1-ből –, és ennek a tulajdonságnak az alapján született a „félig-egyengető” elnevezés. (Vegyük észre, hogy minden állandó sorozat biton sorozat.) A következő lemma a félig-egyengetőnek ezt a tulajdonságát bizonyítja.

27.3. lemma. *Ha egy félig-egyengető bemenete csak 0-ból és 1-ből álló biton sorozat, akkor a kimeneti sorozatra igazak a következők: mind a felső, mind az alsó fele biton sorozat, a felső fele bármely eleme nem nagyobb az alsó fele egyetlen eleménél sem, továbbá legalább az egyik fele állandó.*

Bizonyítás. A FÉLIG-EGYENGET[n] összehasonlító hálózat az i-edik bemenetet összehasonlítja az $(i + n/2)$. bemenettel $(i = 1, 2, \ldots, n/2)$. Anélkül, hogy veszítenénk az általánosságból, feltételezhetjük, hogy a bemeneti sorozat $00\ldots011\ldots100\ldots0$ alakú (Az az eset, amikor a bemenet $11\ldots100\ldots011\ldots1$ alakú, ennek szimmetrikusa.) Attól függően, hogy az $n/2$ középpont a csak 0-ból és a csak 1-ből álló blokkok melyikébe kerül, három eset lehetséges, és egy esetben (amikor a középpont a csupa 1-ből álló blokkba esik) még van két aleset. A négy eset a 27.8. ábrán látható. A lemma mindegyik esetben igaz. ■



27.8. ábra. A FÉLIG-EGYENGET[n] lehetséges összehasonlításai. Feltételezzük, hogy a bemeneti sorozat 0-ból és 1-ből álló biton sorozat, és anélkül, hogy veszítenénk az általánosságból, feltételezzük, hogy $00\ldots011\ldots100\ldots0$ alakú. A csak 0-ból álló részsorozat fehér, míg a csak 1-ből álló részsorozat szürke. Úgy tekintjük, hogy a bemenet két részre van osztva úgy, hogy az i-edik és $(i + n/2)$-edik elemeket hasonlítjuk össze ($i = 1, 2, \ldots, n/2$). **(a)–(b)** Azok az esetek, amikor a felosztás a csupa 1-ből álló részsorozaton belül történik. **(c)–(d)** Azok az esetek, amikor a felosztás a csupa 0-ból álló részsorozaton belül történik. Mindegyik esetben a sorozat felső felének egyetlen eleme sem nagyobb, mint az alsó felének bármelyik eleme, mindegyik félrész biton sorozat, és legalább az egyik állandó.



27.9. ábra. A BITON-RENDEZ[n] összehasonlító hálózat, $n = 8$-ra. **(a)** A rekurzív felépítés: FÉLIG-EGYENGET[n], utána a BITON-RENDEZ[$n/2$] két példánya, amelyek párhuzamosan működnek. **(b)** A hálózat a rekurzív hívások kifejtése után. A félig-egyengetők szürkék. A vezetékekre példaként nulla-egy értékeket rajzoltunk.

A biton rendező

Félig-egyengetők rekurzív összekapcsolásával, amint az a 27.9. ábrán látható, tervezhetünk egy ***biton rendezőt,*** amely biton sorozatok rendezésére szolgáló hálózat. A BITON-RENDEZ[n] első része egy FÉLIG-EGYENGET[n], amelyik a 27.3. lemma szerint két olyan biton sorozatot eredményez, amelyeknek nagysága az eredeti sorozat fele, és a felső félrésznek minden eleme nem nagyobb az alsó félrész bármelyik eleménél. Ezután a két félrészre rekurzívan meghívjuk a BITON-RENDEZ[$n/2$]-t. A 27.9(a) ábrán világosan láthatjuk a rekurzív hívásokat, míg a 27.9(b) ábra bemutatja, kifejtve a rekurziót, az egyre kisebb félig-egyenget őket, amelyek a biton rendezést végzik. A BITON-RENDEZ[n] $D(n)$ mélységét a következő rekurzív képlet adja meg:

$$D(n) = \begin{cases} 0, & \text{ha } n = 1, \\ D(n/2) + 1, & \text{ha } n = 2^k \text{ és } k \geq 1, \end{cases}$$

amelynek a megoldása $D(n) = \lg n$.

Tehát egy nulla-egy biton sorozat rendezhető egy $\lg n$ mélységű BITON-RENDEZ eljárással. A 27.3-6. gyakorlat állításából, amely hasonló a nulla-egy elvhez, következik, hogy ezzel a hálózattal bármilyen számokból álló biton sorozat is rendezhető.

Gyakorlatok

27.3-1. Hány 0-ból és 1-ből álló, n hosszúságú biton sorozat létezik?

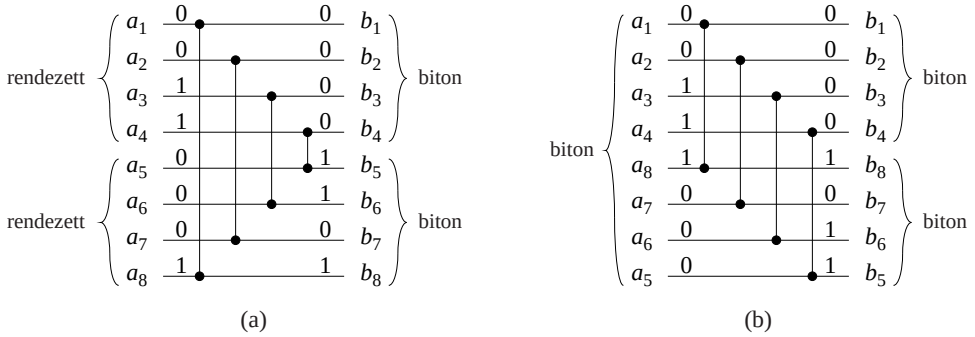
27.3-2. Bizonyítsuk be, hogy ha n kettőhatvány, akkor a BITON-RENDEZ[n] $\Theta(n \lg n)$ összehasonlítót tartalmaz.

27.3-3. Mutassuk meg, hogyan készíthető egy $O(\lg n)$ mélységű biton rendező, ha a bemenetek n száma 2-nek nem hatványa.

27.3-4. Bizonyítsuk be, hogy ha egy félig-egyenget ő bemenete tetszőleges számokból álló biton sorozat, akkor a kimenetének mind a felső, mind az alsó fele biton sorozat, és a felső felének egyetlen eleme sem nagyobb, mint az alsó felének bármely eleme.

27.3-5. Adott két, csak 0-ból és 1-ből álló sorozat. Bizonyítsuk be, hogy ha az egyiknek egyetlen eleme sem nagyobb, mint a másik bármely eleme, akkor az egyik sorozat állandó.

27.3-6. Bizonyítsuk be a következő, a nulla-egy elvhez hasonló állítást: ha egy összehasonlító hálózattal rendezhető bármely csak 0-ból és 1-ből álló biton sorozat, akkor tetszőleges számokból álló biton sorozat is rendezhető.



27.10. ábra. (a) Az ÖSSZEFÉSÜL[n] első részének összehasonlítása a FÉLIG-EGYENGET[n]-nel $n = 8$-ra. Az ÖSSZEFÉSÜL[n] első része átalakítja az $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ és az $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ bemeneti sorozatokat a $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ és $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n\rangle$ biton sorozatokká. **(b)** A FÉLIG-EGYENGET[n] egyenértékű műveletei. A bemeneti $\langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1}\rangle$ sorozatot a $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ és $\langle b_n, b_{n-1}, \ldots, b_{n/2+1}\rangle$ biton sorozatokká alakítja.

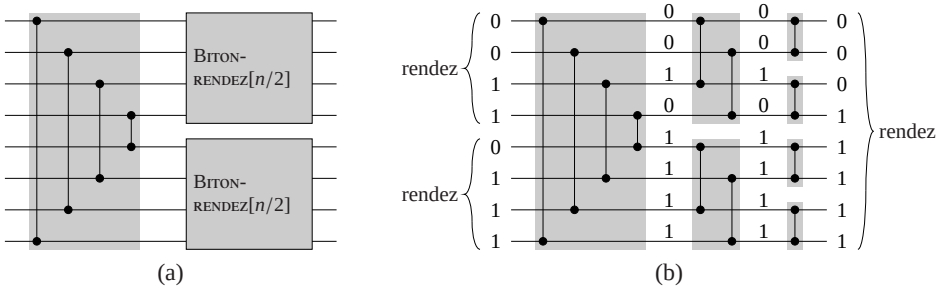
27.4. Összefésülő hálózat

A rendező hálózatunkat olyan ***összefésülő hálózatokból*** építjük fel, amelyek két bemeneti rendezett sorozatot egyetlen kimeneti rendezett sorozattá fésülnek össze. Az ÖSSZEFÉSÜL[n] nevű összefésülő hálózatot a BITON-RENDEZ[n] módosításából nyerjük. Akárcsak a biton rendező esetében, az összefésülő hálózat helyességét is csak nulla-egy sorozatokra bizonyítjuk be. A 27.4-1. gyakorlat azt kéri, hogy mutassuk meg, hogyan terjeszthető ki ez a bizonyítás tetszőleges bemeneti értékekre.

Az összefésülő hálózat a következő ötleten alapszik. Ha adott két rendezett sorozat, és a második sorozat elemeit fordított sorrendben az első után írjuk, akkor biton sorozatot nyerünk. Például adottak az $X = 00000111$ és $Y = 00001111$ nulla-egy rendezett sorozatok, ha Y-t megfordítjuk, az $Y^R = 11110000$ sorozatot kapjuk. X és Y^R összefűzése után a 0000011111110000 sorozatot kapjuk, amely nyilvánvalóan biton sorozat. Tehát, két X és Y bemeneti sorozat összefésüléséhez elegendő végrehajtani egy biton rendezést az X és Y^R konkatenáltján.

Az ÖSSZEFÉSÜL[n]-t a BITON-RENDEZ[n] első félig-egyengetőjének módosításával kapjuk meg. A lényeg az, hogy a bemenet második felét automatikusan megfordítsa. Ha adottak az $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ és az $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ összefésülendő rendezett sorozatok, az $\langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1}\rangle$ sorozat biton rendezésének eredménye érdekel bennünket. Mivel a BITON-RENDEZ[n] félig-egyengetője az i-edik és $(n/2 + i)$-edik elemeket hasonlítja össze, az összefésülő hálózat első részében összehasonlítjuk az i-edik és $(n - i + 1)$-edik elemeket. A 27.10. ábra mutatja a hasonlóságot. Az egyetlen eltérés, hogy az ÖSSZEFÉSÜL[n] első részének alsó kimenetét fordított sorrendben hasonlítjuk össze a szokásos félig-egyengető kimenetével. Mivel egy biton sorozat fordítottja is biton, az összefésülő hálózat első részének alsó és felső kimenete kielégíti a 27.3. lemma követelményeit, így a felső és alsó részt párhuzamos lehet bitonikusan rendezni, s ezzel megkapjuk az összefésülő hálózat kimenetét.

A kapott összefésülő hálózat a 27.11. ábrán látható. Az ÖSSZEFÉSÜL[n] és a BITON-RENDEZ[n] csak első részükben különböznek. Következésképpen, az ÖSSZEFÉSÜL[n] mélysége is $\lg n$, akárcsak a BITON-RENDEZ[n] esetében.



27.11. ábra. Hálózat, amely összefésül két bemeneti rendezett sorozatot egy kimeneti rendezett sorozattá. Az ÖSSZEFÉSÜL[n] úgy tekinthető, mint egy olyan BITON-RENDEZ[n], melynek az első félig-egyengetőjét úgy módosítottuk, hogy az i-edik és az $(n - i + 1)$-edik bemeneteket hasonlítsa össze ($i = 1, 2, \ldots, n/2$). Itt $n = 8$. **(a)** A hálózat első részét a BITON-RENDEZ[$n/2$] két párhuzamos példánya követi. **(b)** Ugyanaz a hálózat a rekurzió kifejtése után. A vezetékeken példaértékek vannak feltüntetve, a különböző részek be vannak satírozva.

Gyakorlatok

27.4-1. Bizonyítsunk be egy, a nulla-egy elvhez hasonlót az összefésülő hálózatokra. Pontosabban, bizonyítsuk be, hogy ha egy összehasonlító hálózat képes összefésülni bármely két monoton növekvő, csak 0-ból és 1-ből álló sorozatot, akkor képes összefésülni bármely két, tetszőleges számokból álló monoton növekvő sorozatot is.

27.4-2. Hány különböző nulla-egy bemeneti sorozatra kell kipróbálni egy összehasonlító hálózatot, hogy ellenőrizhessük, hogy összefésülő hálózat-e?

27.4-3. Bizonyítsuk be, hogy egy tetszőleges hálózatnak, amely képes egy n elemű rendezett sorozattá összefésülni egy 1 elemből és egy $n - 1$ elemből álló sorozatot, legalább $\lg n$ mélységűnek kell lennie.

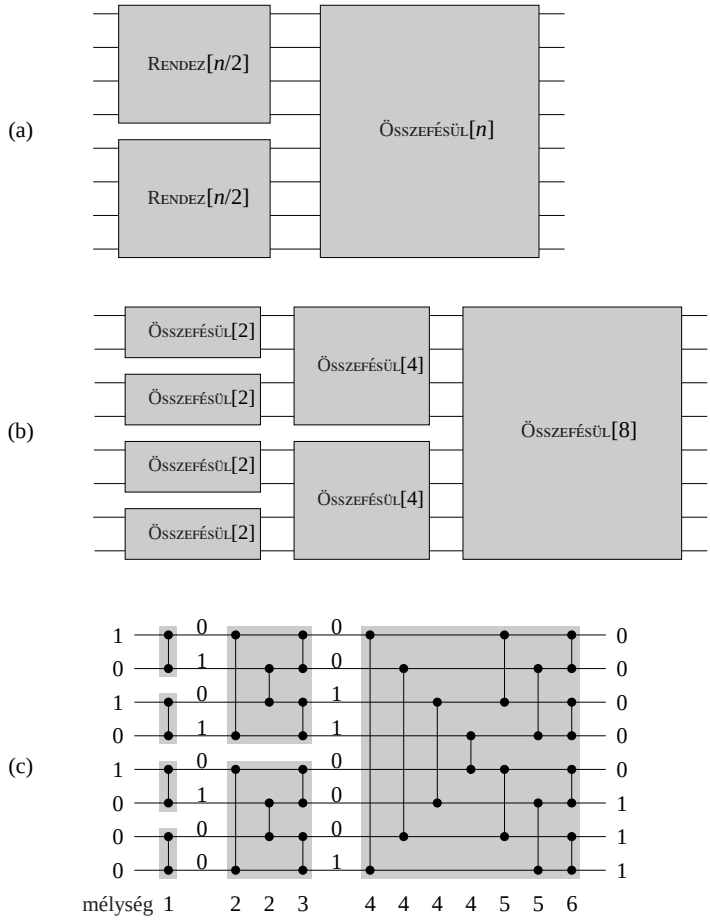
27.4-4.⋆ Tekintsük az $a_1, a_2, \ldots, a_n$ bemenetű összefésülő hálózatot, ahol n 2-nek hatványa, és a két összefésülendő sorozat $\langle a_1, a_3, \ldots, a_{n-1} \rangle$ és $\langle a_2, a_4, \ldots, a_n \rangle$. Bizonyítsuk be, hogy az ilyen típusú összefésülő hálózatokban az összehasonlítók száma $\Omega(n \lg n)$. Miért érdekes ez az alsó határ? (*Útmutatás.* Osszuk be az összehasonlítókat három halmazba.)

27.4-5.⋆ Bizonyítsuk be, hogy bármely összefésülő hálózat, függetlenül a bemenet sorrendjétől, $\Omega(n \lg n)$ összehasonlítót igényel.

27.5. Rendező hálózat

Most már minden eszköz rendelkezésünkre áll, hogy olyan hálózatot tervezzünk, amely bármilyen bemeneti sorozatot képes rendezni. A RENDEZ[n] rendező hálózat összefésülő hálózatot használ a 2.3.1. pontbeli összefésülő rendezés párhuzamos megvalósítására. A 27.12. ábra a rendező hálózat felépítését és működését mutatja be.

A 27.12(a) ábrán a RENDEZ[n] rekurzív felépítése látható. n bemeneti elemet úgy rendezünk, hogy rekurzívan kétszer meghívjuk a RENDEZ[$n/2$]-t, hogy két $n/2$ elemű részt párhuzamosan rendezzen. Az eredményül kapott két sorozatot az ÖSSZEFÉSÜL[n] fésüli egyetlen sorozattá. A rekurzív hívás megállási feltétele $n = 1$, amikor a rendezést egyetlen vezeték elvégezheti, mivel egy egyelemű sorozat már rendezett. A 27.12(b) ábra a rekurzió kifejtését mutatja, míg a 27.12(c) ábrán az a hálózat látható, amelyet úgy kapunk, hogy a 27.12(b) ábra ÖSSZEFÉSÜL dobozait helyettesítettük a megfelelő összefésülő hálózatokkal.



27.12. ábra. A RENDEZ[n] rendező hálózat rekurzív felépítése összefésülő hálózatokból. **(a)** A rekurzív felépítés. **(b)** A rekurzió kifejtése. **(c)** Az ÖSSZEFÉSÜL dobozok helyettesítése a megfelelő összefésülő hálózatokkal. Jelöltük mindegyik összehasonlító mélységét, a vezetékeken pedig példaértékek láthatók.

A RENDEZ[n] hálózatban az adatok $\lg n$ részen mennek keresztül. A hálózat bemenetének minden egyedi eleme egyelemű rendezett sorozat. A RENDEZ[n] első része az ÖSSZEFÉSÜL[2] $n/2$-szeri hívásából áll, amelyek párhuzamosan összefésülnek egyelemű sorozatokat, és így kételemű rendezett sorokat kapunk. A második rész $n/4$ ÖSSZEFÉSÜL[4] hívásból áll, amelyek ezeket a kételemű rendezett sorozatokat négyelemű rendezett sorozatokká fésülik össze. Általában, ha $k = 1, 2, \ldots, \lg n$, a k. részben az ÖSSZEFÉSÜL[2^k]-nak $n/2^k$ hívása szerepel, amelyek 2^{k-1} elemű rendezett sorozatokat fésülnek össze 2^k elemű rendezett sorozatokká. Az utolsó részben a bemeneti sorozat elemei egyetlen rendezett sorozattá alakulnak. Indukció segítségével bizonyítani lehet, hogy ez a rendező hálózat bármely nulla-egy sorozatot rendez, így a nulla-egy elv alapján (27.2. tétel) tetszőleges számokból álló sorozatot is rendez.

A rendező hálózat mélységét rekurzív képlettel vizsgálhatjuk. A RENDEZ[n] $D(n)$ mélysége egyenlő a RENDEZ[$n/2$] $D(n/2)$ mélységének (a RENDEZ[$n/2$]-t kétszer hívjuk meg, de

párhuzamosan működnek) és az ÖSSZEFÉSÜL[n] $\lg n$ mélységének összegével. Tehát a RENDEZ[n] mélységét a következő rekurzív képlet adja meg:

$$D(n) = \begin{cases} 0, & \text{ha } n = 1, \\ D(n/2) + \lg n, & \text{ha } n = 2^k \text{ és } k \geq 1, \end{cases}$$

amelynek megoldása $D(n) = \Theta(\lg^2 n)$. (Használjuk a mester tétel 4.4-2. gyakorlatbeli változatát.) Tehát n számot párhuzamosan $O(\lg^2 n)$ időben tudunk rendezni.

Gyakorlatok

27.5-1. Hány összehasonlító van a RENDEZ[n]-ben?

27.5-2. Bizonyítsuk be, hogy a RENDEZ[n] mélysége pontosan $(\lg n)(\lg n + 1)/2$.

27.5-3. Adott $2n$ elem: $\langle a_1, a_2, \ldots, a_{2n} \rangle$, amelyeket be akarunk osztani a legnagyobb n és a legkisebb n halmazokba. Bizonyítsuk be, hogy ezt megtehetjük állandó mélységgel, miután külön-külön már rendeztük az $\langle a_1, a_2, \ldots, a_n \rangle$ és $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$ sorozatokat.

27.5-4.⋆ Legyen $S(k)$ egy k bemenetű rendező hálózat mélysége, $M(k)$ pedig egy $2k$ bemenetű összefésülő hálózaté. Feltételezve, hogy olyan n elemű rendezendő sorozatunk van, amelyben minden szám legfeljebb k pozícióban tér el a rendezett sorozatban elfoglalt helyétől, bizonyítsuk be, hogy az n elemet $S(k) + 2M(k)$ mélységgel rendezhetjük.

27.5-5.⋆ Egy $m \times m$-es mátrix elemeit rendezhetjük a következő eljárás k-szoros ismétlésével:

1. Rendezzünk monoton növekvő sorrendbe minden páratlan indexű sort.
2. Rendezzünk monoton csökkenő sorrendbe minden páros indexű sort.
3. Rendezzünk monoton növekvő sorrendbe minden oszlopot.

Hányszor kell a fenti eljárást ismételnünk, és milyen sorrendben kell a mátrix elemeit olvasnunk, hogy rendezett kimenetet kapjunk?

Feladatok

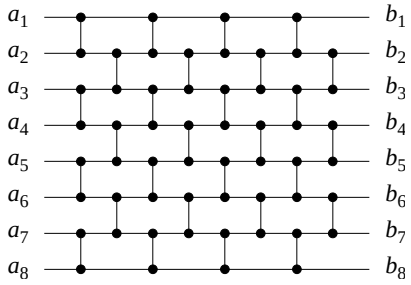
27-1. Transzponáló rendező hálózat

Egy összehasonlító hálózatot ***transzponáló hálózatnak*** nevezünk, ha mindegyik összehasonlítója szomszédos vezetékeket köt össze, mint a 27.3. ábrán látható hálózatban.

a. Mutassuk meg, hogy bármely transzponáló hálózat, amely n bemenő adatot rendez, $\Omega(n^2)$ összehasonlítót tartalmaz.

b. Bizonyítsuk be, hogy egy n bemenő adatú transzponáló hálózat akkor és csakis akkor rendező hálózat, ha rendezi az $\langle n, n-1, \ldots, 1 \rangle$ sorozatot. (*Útmutatás.* Használjunk a 27.1. lemma bizonyításában levő indukciós gondolathoz hasonlót.)

Egy $\langle a_1, a_2, \ldots, a_n \rangle$ bemenetű ***páros-páratlan rendező hálózat*** olyan transzponáló hálózat, amely n szinten téglaszerűen összekapcsolt összehasonlítókat tartalmaz, mint a 27.13. ábrán. Amint az ábrán is látható, $i = 2, 3, \ldots, n-1$ és $d = 1, 2, \ldots, n$ értékekre az i-edik vezetéket egy d mélységű összehasonlító a $j = i + (-1)^{i+d}$-edik vezetékkel kapcsolja össze, ha $1 \leq j \leq n$.



27.13. ábra. Egy 8 bemenetű páros-páratlan rendező hálózat.

c. Bizonyítsuk be, hogy a páros-páratlan rendező hálózatok tényleg rendeznek.

27-2. Batcher páros-páratlan összefésülő hálózata

A 27.4. alfejezetben láttuk, hogyan lehet biton rendezésen alapuló összefésülő hálózatot tervezni. Ebben a feladatban ***páros-páratlan összefésülő hálózatot*** készítünk. Legyen n kettőhatvány. Össze szeretnénk fésülni az $\langle a_1, a_2, \ldots, a_n \rangle$ vezetékek rendezett elemeit az $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$ vezetékek rendezett elemeivel. Ha $n = 1$, akkor az a_1 és a_2 vezetékek közé teszünk egy összehasonlítót. Különben, rekurzívan készítünk két páros-páratlan összefésülő hálózatot, amelyek párhuzamosan összefésülnek rendezett sorozatokat. Az első összefésüli az $\langle a_1, a_3, \ldots, a_{n-1} \rangle$ és $\langle a_{n+1}, a_{n+3}, \ldots, a_{2n-1} \rangle$ vezetékek értékeit (páratlan elemek). A második összefésüli az $\langle a_2, a_4, \ldots, a_n \rangle$ és $\langle a_{n+2}, a_{n+4}, \ldots, a_{2n} \rangle$ vezetékek értékeit (páros elemek). A két rendezett sorozat összefésülésére az a_{2i} és a_{2i+1} elemek közé egy-egy összehasonlítót teszünk ($i = 1, 2, \ldots, n-1$).

a. Rajzoljunk le egy $2n$ bemenetű összefésülő hálózatot $n = 4$-re.

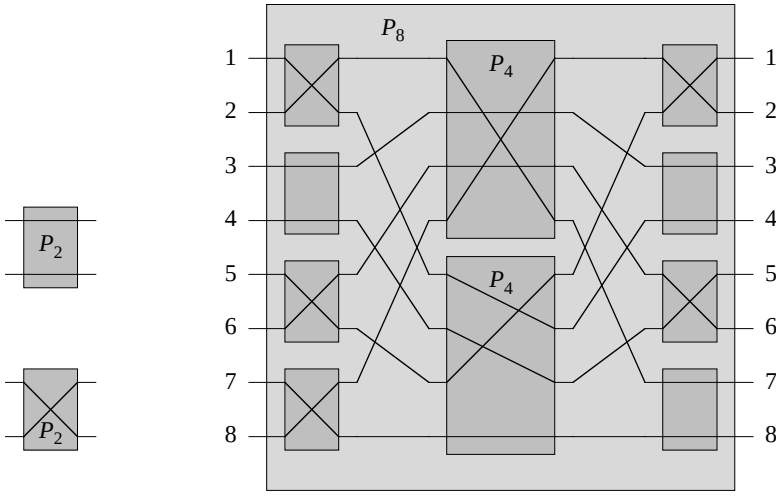
b. Corrigan professzor azt javasolja, hogy a rekurzív összefésülésből kapott két rendezett részsorozatot úgy egyesítsük, hogy ne az a_{2i} és a_{2i+1} ($i = 1, 2, \ldots, n-1$) elemeket kössük össze egy-egy összehasonlítóval, hanem az a_{2i-1} és a_{2i} elemeket ($i = 1, 2, \ldots, n$). Rajzoljunk le egy ilyen $2n$ bemenetű hálózatot $n = 4$-re, és adjunk egy ellenpéldát, amely megmutatja, hogy a professzor téved, amikor azt hiszi, hogy ez a hálózat összefésülő hálózat lesz. Mutassuk meg, hogy az (a) rész $2n$ bemenetű összefésülő hálózata helyesen működik az előbbi példán.

c. Felhasználva a nulla-egy elvet, bizonyítsuk be, hogy bármely $2n$ bemenetű páros-páratlan összefésülő hálózat tulajdonképpen összefésülő hálózat

d. Mennyi a mélysége egy $2n$ bemenetű páros-páratlan összefésülő hálózatnak? Mekkora a nagysága?

27-3. Permutáló hálózat

Az n bemenetű n kimenetű ***permutáló hálózat*** olyan kapcsolókkal rendelkezik, amelyek lehetővé teszik, hogy a bemeneteit az összes $n!$ permutáció szerint kapcsolja a kimenetekhez. A 27.14(a) ábra egy 2 bemenetű, 2 kimenetű P_2 permutáló hálózatot mutat, amelynek egyetlen kapcsolója van, és ez képes a bemeneti és kimeneti vezetékeket egyenesen vagy keresztbe kapcsolni.



27.14. ábra. Permutáló hálózat. **(a)** A P_2 permutáló hálózat, amely egyetlen kapcsolóból áll, és ez kétféleképpen állítható be, amint azt az ábra mutatja. **(b)** A P_8 rekurzív felépítése 8 kapcsolóból és két P_4-ből. A P_4 kapcsolói úgy vannak beállítva, hogy megvalósítsák a $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2\rangle$ permutációt.

a. Indokoljuk meg, hogy ha egy rendező hálózat minden összehasonlítóját a 27.14(a) ábrán látható kapcsolóval helyettesítjük, eredményül egy permutáló hálózatot kapunk. Azaz, bármely π permutációhoz található a kapcsolóknak egy olyan beállítása, hogy az i-edik bemenet a $\pi(i)$. kimenethez kapcsolódjék.

A 27.14(b) ábra egy 8 bemenetű, 8 kimenetű P_8 permutáló hálózat rekurzív felépítést mutatja, amely a P_4-nek két másolatát tartalmazza és 8 kapcsolót. A kapcsolók úgy vannak beállítva, hogy megvalósítsák a $\pi = \langle \pi(1), \pi(2), \ldots, \pi(8)\rangle\langle 4, 7, 3, 5, 1, 6, 8, 2\rangle$ permutációt, amely szerint a felső P_4-nek a $\langle 4, 2, 3, 1\rangle$, míg az alsó P_4-nek a $\langle 2, 3, 1, 4\rangle$ permutációt kell megvalósítania.

b. Mutassuk meg, hogyan valósítható meg az $\langle 5, 3, 4, 6, 1, 8, 2, 7\rangle$ permutáció P_8 segítségével, lerajzolva a kapcsolók beállítását és a két P_4 által megvalósított permutációkat.

Legyen n 2-nek hatványa. Határozzuk meg rekurzívan P_n-et két $P_{n/2}$ segítségével, hasonlóképpen ahogy a P_8 esetében tettük.

c. Írjunk le egy $O(n)$ idejű (közönséges RAM) algoritmust, amely beállítja a P_n bemeneteihez és kimeneteihez kapcsolt n kapcsolót, és megadja a két $P_{n/2}$ által megvalósítandó permutációt, hogy a megadott n elemű permutációt megvalósíthassa. Bizonyítsuk be, hogy az algoritmus helyes.

d. Mennyi P_n mélysége és nagysága? Mennyi ideig tart egy közönséges RAM-gépen, hogy kiszámítsuk az összes kapcsoló-beállítást, beleértve a $P_{n/2}$-kben lévőket is?

e. Indokoljuk meg, hogy $n > 2$-re bármely permutáló hálózatban – nemcsak a P_n-ben – permutációk megvalósításához a kapcsolók két különböző beállítása szükséges.

Megjegyzések a fejezethez

Knuth [185] részletesen tárgyalja a rendező hálózatokat, bemutatja történetüket. Úgy tűnik, hogy a rendező hálózatokat először P. N. Armstrong, R. J. Nelson és D. J. O'Connor vizsgálta 1954-ben. A 60-as évek elején K. E. Batcher leírta az első olyan összefésülő hálózatot, amely két n elemű sorozatot $O(\lg n)$ időben fésült össze. A páros-páratlan összefésülést használta (27-2. feladat), és ugyancsak megmutatta, hogyan lehet ezt a technikát n elemű sorozatok rendezésére használni $O(\lg^2 n)$ időben. Nem sokkal ezután ugyancsak ő egy $O(\lg n)$ mélységű biton rendezőt írt le, amely hasonló a 27.3. alfejezetben bemutatotthoz. A nulla-egy elvet Knuth W. G. Bouriciusnak (1954) tulajdonítja, aki ezt a döntési fák esetében bizonyította be.

Sokáig nyitott kérdés maradt, hogy létezik-e $O(\lg n)$ mélységű rendező hálózat. Erre 1983-ban egy nem teljesen kielégítő pozitív válasz született. Az AKS rendező hálózat (Ajtai, Komlós és Szemerédi [11] után), melynek mélysége $O(\lg n)$, képes n számot $O(n \lg n)$ összehasonlítót használva rendezni. Sajnos, az O-jelölésben megbúvó állandók nagyon nagyok (többezres nagyságrendűek), így ez a módszer gyakorlatilag használhatatlan.

28. Mátrixszámítás

Mátrixokon végzett műveletek képezik a tudományos számítások gerincét, ezért a hatékony mátrix-algoritmusok alapvető jelentőségűek. Ez a fejezet rövid betekintést nyújt a mátrixelméletbe és a mátrixszámításba, kiemelten tárgyalva a mátrixszorzásnak és a lineáris egyenletrendszerek megoldásának a feladatát.

Miután a 28.1. alfejezet bevezeti az alapvető mátrixfajtákat és jelöléseket, a 28.2. alfejezet bemutatja Strassen meglepő algoritmusát, amellyel két $n \times n$-es mátrix $\Theta(n^{\lg 7}) = O(n^{2,81})$ futási idő alatt szorozható össze. A 28.3. alfejezet lineáris egyenletrendszereknek az LUP felbontás módszerével történő megoldását ismerteti, míg a 28.4. alfejezet a mátrixszorzás és a mátrixinvertálás közötti szoros kapcsolatra mutat rá. Végül, a 28.5. alfejezet tárgyalja a szimmetrikus pozitív definit mátrixok fontos osztályát és szerepüket a túlhatározott lineáris egyenletrendszerek legkisebb négyzetes megoldásában.

A gyakorlat szempontjából igen fontos a ***numerikus stabilitás*** problémája. A számítógépek korlátozott pontosságú lebegőpontos számábrázolásának következménye, hogy aritmetikai műveletek elvégzésekor kerekítési hibák keletkeznek. Bizonyos számítási folyamatoknál ezek összessége hibás eredményhez vezethet. Az ilyen számítási folyamatok numerikusan instabilak. Ebben a fejezetben néhány alkalommal csupán utalni fogunk a numerikus stabilitás problémájára. Az érdeklődő Olvasónak Golub és Van Loan [125] kiváló könyvének a tanulmányozását javasoljuk.

28.1. Mátrixok alaptulajdonságai

Ebben az alfejezetben áttekintjük a mátrixelmélet alapfogalmait és mátrixok alapvető tulajdonságait, különös tekintettel azokra, amelyekre a későbbi alfejezetekben szükségünk lesz.

Mátrixok és vektorok

Mátrixnak számok téglalap alakú tömbjét nevezzük. Például,

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \end{aligned} \tag{28.1}$$

egy 2×3-as $A = (a_{ij})$ mátrix, ahol $i = 1, 2$ és $j = 1, 2, 3$ esetén a_{ij} a mátrix i-edik sorában és j-edik oszlopában álló elem. A mátrixokat nagybetűkkel, a mátrixelemeket kisbetűkkel, a valós elemű $m \times n$-es mátrixok halmazát pedig $\mathbf{R}^{m \times n}$-nel jelöljük. Általában, a valamely S halmaz elemeiből összeállított $m \times n$-es mátrixok összességére az $S^{m \times n}$ jelölést használjuk.

Az A mátrix ***transzponáltjának*** nevezzük, és az A^{T} szimbólummal jelöljük azt a mátrixot, amit A sorainak és oszlopainak a felcserélésével kapunk. A (28.1) egyenletben szereplő A mátrixra így

$$A^{\mathrm{T}} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Vektornak számok egydimenziós tömbjét nevezzük. Például

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{28.2}$$

egy 3 elemű vektor. Vektorokat kisbetűkkel, az n elemű x vektor i-edik elemét ($i = 1, 2, \ldots, n$) pedig x_i-vel jelöljük. A vektor – alapértelmezésben – mindig ***oszlopvektor,*** azaz egy $n \times 1$-es mátrix. A megfelelő ***sorvektort*** transzponálással kapjuk:

$$x^{\mathrm{T}} = \begin{pmatrix} 2 & 3 & 5 \end{pmatrix}.$$

Az e_i ***egységvektor*** az a vektor, amelynek az i-edik eleme 1, az összes többi eleme 0. Az egységvektor mérete (dimenziója) a szövegösszefüggésből rendszerint nyilvánvaló.

Nullmátrixnak nevezzük azt a mátrixot, amelynek minden eleme nulla. Az ilyen mátrixot gyakran jelöljük 0-val, mivel a nulla számmal való összetévesztés a szövegösszefüggés alapján többnyire kizárható. A továbbiakban nullmátrix méretét nem fogjuk megadni, mert ez az adott szövegkörnyezetből egyértelműen kiderül.

Gyakoriak az $n \times n$-es, ún. ***négyzetes*** mátrixok. Ezek néhány speciális esete különleges jelentőségű.

1. Egy mátrix ***diagonális mátrix,*** ha $a_{ij} = 0$ minden $i \neq j$ index esetén. Mivel a főátlón kívüli valamennyi elem nulla, ezért ilyen mátrix megadásánál elég a főátló elemeit felsorolni:

$$\operatorname{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix}.$$

2. Az $n \times n$-es I_n ***egységmátrix*** olyan diagonális mátrix, amelynek a főátlójában minden elem 1-es:

$$I_n = \operatorname{diag}(1, 1, \ldots, 1) = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix}.$$

   Az index nélkül I-vel jelölt egységmátrix mérete (dimenziója, rendje) a szövegösszefüggésből derül ki. Az egységmátrix i-edik oszlopa az e_i egységvektor.

3. A T mátrix ***tridiagonális mátrix,*** ha $|i-j|>1$ esetén $t_{ij}=0$. Nemnulla elemek tehát csak a főátlóban, közvetlenül felette ($t_{i,i+1},\ i=1,2,\ldots,n-1$), illetve közvetlenül alatta ($t_{i+1,i},\ i=1,2,\ldots,n-1$) lehetnek:

$$T=\begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

4. Az U mátrix ***felső háromszögmátrix,*** ha $i>j$ esetén $u_{ij}=0$. Ekkor minden főátló alatti elem 0:

$$U=\begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{21} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix}.$$

Egy felső háromszögmátrix ***felső háromszög-főmátrix,*** ha minden főátlóbeli eleme 1-es.

5. Az L mátrix ***alsó háromszögmátrix,*** ha $i<j$ esetén $l_{ij}=0$. Ilyen mátrixban a főátló feletti elemek mindegyike 0:

$$L=\begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix}.$$

Az alsó háromszögmátrix ***alsó háromszög-főmátrix,*** ha minden főátlóbeli eleme 1-es.

6. A P mátrix ***permutációs mátrix,*** ha minden sorában és minden oszlopában pontosan egy 1-es áll, a többi elem pedig 0. Például,

$$P=\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Az elnevezés onnan származik, hogy egy vektort ilyen mátrixszal szorozva a vektor elemeinek egy permutációját (átrendezését) kapjuk.

7. Az A ***szimmetrikus mátrix,*** ha $A=A^{\mathrm{T}}$. Például,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

egy szimmetrikus mátrix.

Mátrixműveletek

Egy mátrix vagy egy vektor elemei valamely számkörből – pl. valós vagy komplex számok vagy egy p prímmodulusra nézve az egészek mod p vett halmaza – vett számok. A számkörrel együtt a számok összeadásának és szorzásának a definíciója is adott. E definíciókat kiterjeszthetjük mátrixok összeadásának és szorzásának az értelmezésére.

Mátrixok összegét a következőképpen definiáljuk. Az $m \times n$-es $A = (a_{ij})$ és $B = (b_{ij})$ mátrixok összege az az $m \times n$-es $C = (c_{ij}) = A + B$ mátrix, amelyre

$$c_{ij} = a_{ij} + b_{ij}$$

minden $i = 1, 2, \ldots, m$ és $j = 1, 2, \ldots, n$ esetén. Tehát mátrixok összeadását elemenként végezzük. A mátrixösszeadás egységeleme a nullmátrix:

$$\begin{aligned} A + 0 &= A \\ &= 0 + A. \end{aligned}$$

Ha λ egy szám, $A = (a_{ij})$ pedig egy mátrix, akkor A-nak a ***skalárszorosa,*** $\lambda A = (\lambda a_{ij})$ az A elemeinek λ-val való szorzásával kapható. Speciálisan, $A = (a_{ij})$ ***negatívja*** $(-1) \cdot A = -A$, azaz $-A$-nak az (i, j) pozícióban levő eleme $-a_{ij}$. Ilyen módon

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A. \end{aligned}$$

E definíció birtokában ***mátrixok különbsége*** is értelmezhető a mátrix negatívjának a hozzáadásával: $A - B = A + (-B)$.

Mátrixok szorzatát a következőképpen definiáljuk. Azt mondjuk, hogy az A és B mátrix ***kompatibilis*** (azaz ebben a sorrendben összeszorozhatók), ha A oszlopainak a száma megegyezik B sorainak a számával. (Ha egy kifejezésben a továbbiakban AB szerepel, akkor a kompatibilitást mindig feltesszük.) Ha $A = (a_{ij})$ egy $m \times n$-es, $B = (b_{jk})$ pedig egy $n \times p$-s mátrix, akkor szorzatuk az az $m \times p$ méretű $C = (c_{ik})$ mátrix, amelyre

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{28.3}$$

minden $i = 1, 2, \ldots, m$ és $j = 1, 2, \ldots, p$ esetén. A 25.1. alfejezetben található Mátrixszorzás eljárás (28.3)-nak a közvetlen megvalósítása azon feltétel mellett, hogy a mátrixok négyzetesek: $m = n = p$. Az eljárás $n \times n$-es mátrixok összeszorzását n^3 szorzással és $n^2(n-1)$ összeadással, $\Theta(n^3)$ futási idő alatt végzi el.

A számok közötti algebrai műveletekre érvényes tulajdonságok közül többet (de nem mindet) a mátrixok is megőriznek. Az egységmátrixok a mátrixszorzás egységelemei:

$$I_m A = A I_n = A$$

minden $m \times n$-es A mátrixra. A nullmátrixszal való szorzás nullmátrixot ad:

$$A0 = 0.$$

A mátrixszorzás asszociatív, azaz kompatibilis A, B, C mátrixok esetén:

$$A(BC) = (AB)C. \tag{28.4}$$

Érvényesek a disztributivitási szabályok is:

$$\begin{aligned} A(B+C) &= AB+AC, \\ (B+C)D &= BD+CD. \end{aligned} \tag{28.5}$$

$n\times n$-es mátrixok szorzása azonban – az $n=1$ esettől eltekintve – nem kommutatív. Például, ha $A=\begin{pmatrix}0&1\\0&0\end{pmatrix}$ és $B=\begin{pmatrix}0&0\\1&0\end{pmatrix}$, akkor

$$AB=\begin{pmatrix}1&0\\0&0\end{pmatrix},$$

ugyanakkor

$$BA=\begin{pmatrix}0&0\\0&1\end{pmatrix}.$$

A mátrix-vektor, illetve vektor-vektor szorzatokat úgy definiáljuk, mintha a vektor $n\times 1$-es (sorvektor esetén $1\times n$-es) mátrix volna. Így, ha A egy $m\times n$-es mátrix és x egy n elemű (oszlop)vektor, akkor Ax dimenziója m. Ha x is és y is n elemű vektor, akkor

$$x^{\mathrm{T}}y=\sum_{i=1}^{n}x_iy_i$$

egy szám (egy 1×1-es mátrix), amelyet x és y ***belső szorzatának*** hívunk. Az xy^{T} egy $n\times n$-es $Z=(z_{ij})$ mátrix, ahol $z_{ij}=x_iy_j$. Ezt a mátrixot az x és y vektorok ***külső szorzatának*** nevezzük. Az n dimenziós x vektor $\|x\|$ szimbólummal jelölt ***(euklideszi) normáját*** az

$$\begin{aligned} \|x\| &= (x_1^2+x_2^2+\cdots+x_n^2)^{1/2} \\ &= (x^{\mathrm{T}}x)^{1/2} \end{aligned}$$

egyenlőséggel definiáljuk. Az x vektor normája tehát éppen a vektor n dimenziós euklideszi térbeli hossza.

Mátrix inverze, rangja és determinánsa

Az $n\times n$-es A mátrix ***inverze*** – ha létezik – az az A^{-1} szimbólummal jelölt $n\times n$-es mátrix, amelyre $AA^{-1}=I_n=A^{-1}A$. Például,

$$\begin{pmatrix}1&1\\1&0\end{pmatrix}^{-1}=\begin{pmatrix}0&1\\1&-1\end{pmatrix}.$$

Sok nemnulla $n\times n$-es mátrixnak nincs inverze. Az ilyeneket ***nem invertálható*** vagy ***szinguláris*** mátrixoknak nevezzük. Egy példa nemnulla szinguláris mátrixra a következő:

$$\begin{pmatrix}1&0\\1&0\end{pmatrix}.$$

Ha egy mátrixnak van inverze, akkor azt ***invertálhatónak*** vagy ***nemszingulárisnak*** mondjuk. Mátrix inverze, ha létezik, egyértelműen meghatározott (lásd a 28.1-3. gyakorlatot). Ha

A és B $n \times n$-es nemszinguláris mátrixok, akkor

$$(BA)^{-1} = A^{-1}B^{-1}. \tag{28.6}$$

Az invertálás és a transzponálás műveletei felcserélhetők:

$$(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1}.$$

Az $x_1, x_2, \ldots, x_n$ vektorok ***lineárisan összefüggők,*** ha vannak olyan, nem mind nullával egyenlő $c_1, c_2, \ldots, c_n$ együtthatók, amelyekkel $c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$. Például, az $x_1 = (1\,2\,3)^{\mathrm{T}}$, $x_2 = (2\,6\,4)^{\mathrm{T}}$ és az $x_3 = (4\,11\,9)^{\mathrm{T}}$ vektorok lineárisan összefüggők, mert $2x_1 + 3x_2 - 2x_3 = 0$. Ha a vektorok nem lineárisan összefüggők, akkor ***lineárisan függetleneknek*** nevezzük őket. Például, az egységmátrix oszlopai lineárisan függetlenek.

Egy nemnulla $m \times n$-es A mátrix ***oszloprangja*** A lineárisan független oszlopainak a maximális száma. Hasonlóan, A-nak a ***sorrangja*** A lineárisan független sorainak a maximális száma. Alapvető tény a következő: minden A mátrix sorrangja egyenlő az oszloprangjával. Ezt a közös értéket az A mátrix ***rangjának*** nevezzük. Egy $m \times n$-es mátrix rangja 0 és $\min(m, n)$ közötti egész szám, a két szélső értéket is beleértve. (A nullmátrix rangja 0, az $n \times n$-es egységmátrix rangja n.) Egy ettől eltérő, de vele ekvivalens, és gyakran hasznosabb definíció szerint az $m \times n$-es nemnulla A mátrix rangja az a legkisebb r szám, amelyre léteznek az

$$A = BC$$

egyenlőséget kielégítő $m \times r$-es B, illetve $r \times n$-es C mátrixok. Egy $n \times n$-es négyzetes mátrix ***teljes rangú,*** ha rangja n. Egy $m \times n$-es mátrix ***teljes oszloprangú,*** ha rangja n. A rang alapvető tulajdonságát mondja ki a

28.1. tétel. *Négyzetes mátrix akkor és csak akkor teljes rangú, ha nemszinguláris.*

Az A ***mátrix nullvektorának*** nevezünk egy nemnulla x vektort, ha $Ax = 0$. A következő tétel (amelynek a bizonyítását a 28.1-9. gyakorlatra hagyjuk) és következménye az oszloprangnak és a szingularitásnak a mátrix nullvektorával való kapcsolatát tárgyalja.

28.2. tétel. *Az A mátrix akkor és csak akkor teljes rangú, ha nincs nullvektora.*

28.3. következmény. *Az A négyzetes mátrix akkor és csak akkor szinguláris, ha van nullvektora.*

Egy $n \times n$-es A mátrix ij-edik ***minormátrixán*** ($n > 1$ esetén) az i-edik sor és a j-edik oszlop törlésével adódó, $A_{[ij]}$-nel jelölt $(n-1) \times (n-1)$-es mátrixot értjük. Az $n \times n$-es A mátrix ***determinánsát*** rekurzív módon értelmezzük:

$$\det(A) = \begin{cases} a_{11}, & \text{ha } n = 1, \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A_{[1j]}), & \text{ha } n > 1. \end{cases} \tag{28.7}$$

A $(-1)^{i+j} \det(A_{[ij]})$ számot az A mátrix a_{ij} eleméhez tartozó ***előjeles aldeterminánsnak*** (vagy másképpen ***algebrai komplementumnak***) is szokás nevezni.

A következő, bizonyítás nélkül közölt állítások a determináns alapvető tulajdonságait fejezik ki.

28.4. tétel (a determináns tulajdonságai). *Négyzetes A mátrix determinánsára fennállnak a következők*:

- *Ha A valamely sora vagy oszlopa* 0, *akkor* $\det(A) = 0$.
- *Ha A valamely sorának* (*vagy oszlopának*) *az elemeit λ-val szorozzuk, akkor A determinánsa is λ-val szorzódik.*
- *A-nak a determinánsa nem változik, ha egy sor* (*illetve oszlop*) *elemeit hozzáadjuk egy másik sor* (*illetve oszlop*) *elemeihez.*
- *A-nak a determinánsa egyenlő A^{T} determinánsával.*
- *Ha A két sorát* (*illetve oszlopát*) *felcseréljük, determinánsa* (-1)*-szeresére változik.*

Négyzetes A és B mátrixokra $\det(AB) = \det(A)\det(B)$.

28.5. tétel. *Az $n \times n$-es A mátrix akkor és csak akkor szinguláris, ha* $\det(A) = 0$.

Pozitív definit mátrixok

A pozitív definit mátrixok sok alkalmazásban fontos szerepet játszanak. Egy $n \times n$-es A mátrix ***pozitív definit,*** ha $x^{\mathrm{T}}Ax > 0$ minden n elemű nemnulla x vektor esetén. Például, az egységmátrix pozitív definit, mert minden nemnulla $x = (x_1\ x_2\ \ldots\ x_n)^{\mathrm{T}}$ vektorra

$$\begin{aligned} x^{\mathrm{T}}I_n x &= x^{\mathrm{T}}x \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0. \end{aligned}$$

Később látni fogjuk azt, hogy az alkalmazásokban el őforduló mátrixok gyakran pozitív definit mátrixok. Ez sok esetben az alábbi állításból következik.

28.6. tétel. *Minden teljes rangú A mátrixra $A^{\mathrm{T}}A$ pozitív definit.*

Bizonyítás. Azt kell belátnunk, hogy $x^{\mathrm{T}}(A^{\mathrm{T}}A)x > 0$ minden nemnulla x vektor esetén. Minden x vektorra:

$$\begin{aligned} x^{\mathrm{T}}(A^{\mathrm{T}}A)x &= (Ax)^{\mathrm{T}}(Ax) \qquad \text{(lásd a 28.1-2. gyakorlatot)} \\ &= \|Ax\|^2. \end{aligned}$$

Vegyük észre, hogy itt $\|Ax\|^2$ éppen az Ax vektor elemeinek a négyzetösszege, ezért $\|Ax\| \geq 0$. Ha $\|Ax\|^2 = 0$, akkor Ax minden eleme nulla, azaz $Ax = 0$. Mivel A teljes rangú, ezért a 28.2. tétel szerint $Ax = 0$-ból $x = 0$ következik, ami azt jelenti, hogy az $A^{\mathrm{T}}A$ mátrix pozitív definit. ■

A 28.5. alfejezetben pozitív definit mátrixok további tulajdonságait ismertetjük.

Gyakorlatok

28.1-1. Mutassuk meg, hogy ha A és B $n \times n$-es szimmetrikus mátrixok, akkor az $A + B$ és az $A - B$ is szimmetrikus mátrix.

28.1-2. Bizonyítsuk be, hogy tetszőleges A és B mátrix esetén $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$, és az $A^{\mathrm{T}}A$ mátrix szimmetrikus.

28.1-3. Igazoljuk, hogy mátrix inverze egyértelmű, azaz, ha B is és C is inverze az A mátrixnak, akkor $B = C$.

28.1-4. Bizonyítsuk be, hogy két alsó háromszögmátrix szorzata is alsó háromszögmátrix; alsó, illetve felső háromszögmátrix determinánsa a főátlóbeli elemek szorzata; alsó háromszögmátrix inverze, ha létezik, szintén alsó háromszögmátrix.

28.1-5. Mutassuk meg, hogy ha P egy $n \times n$-es permutációs mátrix és A egy tetszőleges $n \times n$-es mátrix, akkor PA-t a sorok, AP-t pedig az oszlopok permutációjával kapjuk az A mátrixból. Igazoljuk, hogy két permutációs mátrix szorzata is permutációs mátrix. Bizonyítsuk be, hogy ha P permutációs mátrix, akkor P invertálható és inverze P^{T}, valamint P^{T} is permutációs mátrix.

28.1-6. Legyenek A és B olyan $n \times n$-es mátrixok, amelyekre $AB = I$. Lássuk be, hogy ha A' az A mátrixból a j-edik sornak az i-edik sorhoz való hozzáadásával adódik, akkor A'-nek az inverzét a B mátrix i-edik oszlopának a j-edik oszlopból történő kivonásával kapjuk.

28.1-7. Legyen A egy $n \times n$-es nemszinguláris komplex mátrix. Mutassuk meg, hogy A^{-1} összes eleme akkor és csak akkor valós, ha A minden eleme valós.

28.1-8. Bizonyítsuk be, hogy ha A $n \times n$-es nemszinguláris szimmetrikus mátrix, akkor A^{-1} is szimmetrikus. Mutassuk meg, hogy ha B tetszőleges $m \times n$-es mátrix, akkor az $m \times m$-es BAB^{T} mátrix szimmetrikus.

28.1-9. Bizonyítsuk be a 28.2. tételt, azaz mutassuk meg, hogy az A mátrix akkor és csak akkor teljes oszloprangú, ha $Ax = 0$-ból $x = 0$ következik. (*Útmutatás.* Egy oszlopnak a többitől való lineáris függősége ekvivalens egy mátrix-vektor szorzatra fennálló egyenlőséggel.)

28.1-10. Bizonyítsuk be, hogy összeszorozható A és B mátrixokra

$$\operatorname{rang}(AB) \leq \min(\operatorname{rang}(A), \operatorname{rang}(B)),$$

és egyenlőség akkor és csak akkor áll fenn, ha A vagy B nemszinguláris négyzetes mátrix. (*Útmutatás.* Használjuk a rang másik, ekvivalens definícióját.)

28.1-11. Adottak az $x_0, x_1, \ldots, x_{n-1}$ számok. Bizonyítsuk be, hogy a

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix}$$

Vandermonde-mátrix determinánsa

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \leq j < k \leq n-1} (x_k - x_j).$$

(*Útmutatás.* Adjuk hozzá $i = n-1, n-2, \ldots, 1$ esetén az i-edik oszlop $(-x_0)$-szorosát az $(i+1)$-edik oszlophoz, és alkalmazzunk indukciót.)

28.2. Strassen mátrixszorzási algoritmusa

Ebben az alfejezetben Strassen figyelemre méltó rekurzív mátrixszorzási algoritmusát ismertetjük, amellyel két $n \times n$-es mátrixot $\Theta(n^{\lg 7}) = O(n^{2.81})$ futási idő alatt szorozhatunk össze. Ez az eljárás elég nagy n-re tehát gyorsabb, mint a 25.1. alfejezetbeli, $\Theta(n^3)$ műveletigényű MÁTRIXSZORZÁS algoritmus.

Az algoritmus vázlata

Strassen algoritmusa az oszd-meg-és-uralkodj elvként ismert tervezési technika egy alkalmazásának tekinthető. Tegyük fel, hogy az $n \times n$-es A és B mátrixok $C = AB$ szorzatát akarjuk kiszámítani. Ha az n szám 2-nek egy hatványa, akkor az A, B és C mátrixok mindegyikét négy, $n/2 \times n/2$ méretű mátrixra bontjuk, és $C = AB$-t így írjuk fel:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}. \tag{28.8}$$

(A 28.2-2. gyakorlat foglalkozik azzal az esettel, amikor n nem kettőhatvány.) A (28.8) egyenlet ekvivalens az alábbi négy egyenlőséggel:

$$r = ae + bg, \tag{28.9}$$
$$s = af + bh, \tag{28.10}$$
$$t = ce + dg, \tag{28.11}$$
$$u = cf + dh. \tag{28.12}$$

Ezek mindegyike két $n/2 \times n/2$-es mátrix összeszorzását és a szorzatok összeadását tartalmazza. Két $n \times n$-es mátrix összeszorzásának $T(n)$ futási idejére az oszd-meg-és-uralkodj stratégiát alkalmazva a

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2) \tag{28.13}$$

rekurzív egyenlőséget kapjuk. A (28.13) rekurzió megoldása sajnos $T(n) = \Theta(n^3)$, ezért ez a módszer nem gyorsabb a szokásosnál.

Ezzel szemben Strassen egy olyan rekurzív megközelítést fedezett fel, amelynek n-edik lépésében már csak 7 darab $n/2 \times n/2$-es mátrixszorzás és $\Theta(n^2)$ nagyságrendű skalár összeadás és kivonás szerepel, ezért a $T(n)$ futási idejére az alábbi rekurzió érvényes:

$$\begin{aligned} T(n) &= 7T\left(\frac{n}{2}\right) + \Theta(n^2) \\ &= \Theta(n^{\lg 7}) \\ &= O(n^{2,81}). \end{aligned} \tag{28.14}$$

Strassen módszere négy lépésből áll:

1. Az A és B bemenő mátrixokat (28.8)-nak megfelelően $n/2 \times n/2$-es részmátrixokra bontjuk.
2. A bemenő mátrixokból $\Theta(n^2)$ nagyságrendű skalár összeadás és kivonás segítségével 14 darab $n/2 \times n/2$ méretű, $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$-tel jelölt mátrixokat készítünk.

3. Kiszámítjuk a $P_i = A_i B_i$ ($i = 1, 2, \ldots, 7$) mátrixszorzatokat.
4. A C szorzatmátrix r, s, t és u részmátrixait a P_i mátrixok összegeinek és/vagy különbségeinek különböző kombinációjaként számítjuk ki. Ehhez csupán $\Theta(n^2)$ nagyságrendű skalár összeadást és kivonást kell elvégezni.

Egy ilyen eljárás kielégíti a (28.14) rekurzív egyenletet. Most megmutatjuk azt, hogy *létezik* ilyen eljárás.

A részmátrix-szorzatok meghatározása

Nem világos, hogy Strassen pontosan hogyan fedezte fel azokat a részmátrix-szorzatokat, amelyen algoritmusa alapul. Itt egy lehetséges „felfedező utat” járunk be.

Írjuk fel a P_i mátrixszorzatot az alábbi módon:

$$\begin{aligned} P_i &= A_i B_i \\ &= (\alpha_{i1} a + \alpha_{i2} b + \alpha_{i3} c + \alpha_{i4} d) \cdot (\beta_{i1} e + \beta_{i2} f + \beta_{i3} g + \beta_{i4} h), \end{aligned} \tag{28.15}$$

ahol az α_{ij}, β_{ij} együtthatók mindegyikét a $\{-1, 0, 1\}$ halmazból vesszük. Minden ilyen szorzat kiszámításához tehát az A és a B mátrix bizonyos részmátrixainak az összegeit vagy különbségeit, majd az így kapott mátrixok szorzatát kell képeznünk. Általánosabb stratégiák is léteznek, de ez az egyszerű kiindulás is célhoz vezet.

A fenti típusú szorzatokat rekurzív módon is meghatározhatjuk, éspedig a kommutativitás feltételezése nélkül, mivel minden szorzatban a bal oldali részmátrix A-ból, a jobb oldali pedig B-ből származik. Ez a tulajdonság a módszer rekurzív alkalmazhatósága szempontjából lényeges, hiszen a mátrixszorzás nem kommutatív.

A (28.15)-beli mátrixokat, azaz az A-ból és B-ből vett részmátrixok szorzatainak lineáris kombinációit alkalmas 4×4-es mátrixok felhasználásával is elő lehet állítani. A (28.9) egyenlőséget például így írhatjuk fel:

$$\begin{aligned} r &= ae + b \\ &= \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\ &= \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}. \end{aligned}$$

Az utolsó egyenletben a $+1$, 0, -1 számokat rendre a $+$, $.$, $-$ jelekkel helyettesítettük. (Mostantól kezdve a sor- és az oszlopcímkéket is el fogjuk hagyni.) E jelölések alkalmazásával a C eredménymátrix további részmátrixait így írhatjuk fel:

$$
\begin{aligned}
s &= af + bh \\
&= \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \\
t &= ce + dg \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix}, \\
u &= cf + dh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

A hagyományosnál gyorsabb mátrixszorzási algoritmus keresését azzal az észrevétellel kezdjük, hogy az s részmátrixot az alábbi mátrixok összegeként írhatjuk fel:

$$
\begin{aligned}
P_1 &= A_1 B_1 \\
&= a \cdot (f - h) \\
&= af - ah \\
&\quad \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}
\end{aligned}
$$

és

$$
\begin{aligned}
P_2 &= A_2 B_2 \\
&= (a + b) \cdot h \\
&= ah + bh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

A t mátrixot hasonlóan számíthatjuk ki $t = P_3 + P_4$ alapján, ahol

$$
\begin{aligned}
P_3 &= A_3 B_3 \\
&= (c + d) \cdot e \\
&= ce + de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}
\end{aligned}
$$

és

$$\begin{aligned} P_4 &= A_4 B_4 \\ &= d \cdot (g - e) \\ &= dg - de \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}. \end{aligned}$$

Nevezzük a (28.9)–(28.12) egyenlőségek jobb oldalán álló nyolc tag mindegyikét ***lényeges tagnak.*** Az imént 4 szorzatot használtunk az s és t részmátrixok kiszámítására; ezekben ag, bh, ce és df voltak a lényeges tagok, mégpedig P_1-ben af, P_2-ben bh, P_3-ban ce és P_4-ben dg. Három további szorzással kell tehát kiszámítanunk a fennmaradó r és u részmátrixokat. Ezek lényeges tagjai ae, bg, cf és dh. A P_5 mátrixban megpróbálunk egyszerre két lényeges tagot kiszámolni:

$$\begin{aligned} P_5 &= A_5 B_5 \\ &= (a + d) \cdot (e + h) \\ &= ae + ah + de + dh \\ &= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Az ae és dh lényeges tagokon kívül most az ah és de lényegtelen tagokat is kiszámítottuk. Ezeket P_4 és P_2 segítségével eltüntethetjük, de ekkor két másik lényegtelen tag is megjelenik:

$$\begin{aligned} P_5 + P_4 - P_2 &= ae + dh + dg - bh \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix}. \end{aligned}$$

Ha a fentihez még hozzáadjuk a

$$\begin{aligned} P_6 &= A_6 B_6 \\ &= (b - d) \cdot (g + h) \\ &= bg + bh - dg - dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix} \end{aligned}$$

szorzatot, akkor azt kapjuk, hogy

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= ae + bg \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Hasonlóan kaphatjuk meg az u mátrixot P_5-ből, P_1-et és P_3-at használva a lényegtelen tagok kiejtésére:

$$\begin{aligned} P_5 + P_1 - P_3 &= ae + af - ce + dh \\ &= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Ebből egy újabb

$$\begin{aligned} P_7 &= A_7 B_7 \\ &= (a - c) \cdot (e + f) \\ &= ae + af - ce - cf \\ &= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{aligned}$$

szorzatot kivonva megkapjuk az u mátrixot:

$$\begin{aligned} u &= P_5 + P_1 - P_3 - P_7 \\ &= cf + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

A $P_1, P_2, \ldots, P_7$ részmátrixszorzatokkal tehát valóban kiszámíthatjuk a $C = AB$ szorzatot, így Strassen módszerének a leírását befejeztük.

Megjegyzések

A Strassen-algoritmus gyakorlati alkalmazását az alábbi tények korlátozzák:

1. Az eljárás futási idejének képletében rejlő állandó nagyobb, mint a hagyományos módszer $\Theta(n^3)$ képletében foglalt állandó.
2. A ritka mátrixokra kidolgozott speciális algoritmusok gyorsabbak.

3. A Strassen-algoritmus numerikusan kevésbé stabil, mint a hagyományos módszer.
4. A rekurziós lépések során a részmátrixok tárolása a memóriában helyet igényel.

Az utóbbi két hátrányt 1990 körül sikerült csökkenteni. Higham [145] megmutatta, hogy a numerikus stabilitásban levő különbségek túlhangsúlyozottak. Igaz, hogy bizonyos esetekben a Strassen-algoritmus numerikusan instabil, ugyanakkor sok esetben a stabilitása elfogadható határok között van. Bailey, Lee és Simon [30] dolgoztak ki olyan technikákat, amelyekkel a Strassen-algoritmus memóriaigénye csökkenthető.

A gyakorlatban használt gyors mátrixszorzási algoritmusok sűrű mátrixok esetén a Strassen-algoritmust használják abban az esetben, ha a mátrix mérete egy „váltási érték" felett van. Az ellenkező esetben ezek a hagyományos módszert alkalmazzák. A váltási érték nagymértékben függ az adott rendszertől. Becslésére csak a szükséges műveletek számát figyelembe vevő elemzéseket többen is végeztek. Higham [145] $n = 8$-at kapott, Huss-Lederman és szerzőtársai [163] pedig 12-t. A gyakorlatban a váltási érték ennél nagyobb, „tipikus" esetekben 20 körül van. Egy adott rendszerhez tartozó váltási érték kísérletezéssel határozható meg.

Könyvünk keretein túlmutató, finomabb technikák alkalmazásával $n \times n$ méretű mátrixokat $\Theta(n^{\lg 7})$ időnél gyorsabban is össze lehet szorozni. A jelenlegi legjobb felső korlát $O(n^{2,376})$ nagyságrend körül van. A legjobb alsó korlát éppen a nyilvánvaló $\Omega(n^2)$ (ugyanis n^2 elemet kell kitöltenünk a szorzatmátrixban). A legjobb mátrixszorzási algoritmus futási idejének pontos nagyságrendjét ma még nem ismerjük.

Gyakorlatok

28.2-1. Strassen algoritmusával számítsuk ki az alábbi szorzatot:

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

28.2-2. Hogyan módosítsuk Strassen algoritmusát olyan $n \times n$-es mátrixok szorzására, amelyeknél n nem kettőhatvány? Mutassuk meg, hogy a kapott algoritmus futási ideje $\Theta(n^{\lg 7})$.

28.2-3. Melyik az a legnagyobb k, amelyik rendelkezik a következő tulajdonsággal: ha 3×3-as mátrixokat össze tudunk szorozni k szorzással (a szorzás kommutativitásának a feltételezése nélkül), akkor $n \times n$-es mátrixokat össze tudunk szorozni $o(n^{\lg 7})$ lépésben? Mi lenne ennek az algoritmusnak a futási ideje?

28.2-4. V. Pan olyan módszert talált, amellyel 68×68-as mátrixok szorzatát 132 464; 70×70-es mátrixok szorzatát 143 640; 72×72-es mátrixok szorzatát pedig 155424 szorzással tudja meghatározni. Melyik módszer adja az aszimptotikusan legjobb futási időt, ha egy oszd-meg-és-uralkodj típusú mátrixszorzási algoritmusban használjuk? Vessük össze Strassen algoritmusával.

28.2-5. Milyen gyorsan szorozható össze egy $kn \times n$-es és egy $n \times kn$-es mátrix a Strassen-algoritmussal? Adjunk választ a mátrixok fordított sorrendje esetén is.

28.2-6. Számítsuk ki az $a + bi$ és a $c + di$ komplex számok szorzatát csak három valós szorzással. Az algoritmus bemenő adatai legyenek a, b, c és d, ezekből határozzuk meg a szorzat $ac - bd$ valós részét és az $ad + bc$ képzetes részét.

28.3. Lineáris egyenletrendszerek megoldása

Lineáris egyenletrendszerek megoldása olyan alapvető feladat, amely különféle alkalmazásokban előfordul. Egy lineáris egyenletrendszert mátrix-vektor egyenlet alakban is felírhatunk, ahol a mátrix, illetve a vektor elemei egy testhez, leggyakrabban az $\mathbf{R}$ valós számtesthez tartoznak. Ebben az alfejezetben azt mutatjuk meg, hogyan lehet lineáris egyenletrendszereket az LUP felbontás módszerével megoldani.

Az n egyenletből álló, $x_1, x_2, \ldots, x_n$ ismeretleneket tartalmazó lineáris egyenletrendszer általános alakja

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned} \tag{28.16}$$

A fenti egyenletek mindegyikét kielégítő $x_1, x_2, \ldots, x_n$ számok összességét az egyenletrendszer ***megoldásának*** nevezzük. Ebben az alfejezetben csak azzal az esettel foglalkozunk, amikor az egyenletek és az ismeretlenek száma megegyezik.

A (28.16) egyenletrendszert átírhatjuk a kényelmes

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

mátrix-vektor egyenletbe, amit az $A = (a_{ij})$, $x = (x_i)$, $b = (b_i)$ jelölésekkel még tömörebben így is írhatunk:

$$Ax = b. \tag{28.17}$$

Ha A nemszinguláris mátrix, akkor A^{-1} létezik, és

$$x = A^{-1}b \tag{28.18}$$

a megoldásvektor. Könnyen beláthatjuk, hogy ez az x vektor az egyetlen megoldása a (28.17) egyenletnek. Valóban, ha x és x' is megoldás, akkor $Ax = Ax' = b$, és

$$\begin{aligned} x &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x'. \end{aligned}$$

Ebben az alfejezetben túlnyomórészt azzal az esettel foglalkozunk, amikor A nemszinguláris mátrix, ami a 28.1. tétel szerint azzal ekvivalens, hogy A-nak a rangja egyenlő az ismeretlenek n számával. Vannak azonban más lehetőségek is, amelyeket most csak röviden

vázolunk. Ha az egyenletek száma kevesebb, mint az ismeretleneké (n) – vagy általánosabban, ha A rangja kisebb, mint n –, akkor a rendszer ***alulhatározott.*** Ilyen rendszereknek tipikusan végtelen sok megoldása van, bár az is előfordulhat, hogy egyetlen megoldása sincs. Ez a helyzet akkor, amikor a rendszer egymásnak ellentmondó egyenleteket tartalmaz. Ha az egyenletek száma nagyobb, mint az ismeretlenek száma, akkor a rendszer ***túlhatározott,*** és lehet, hogy nincs megoldása. Az ilyen egyenletrendszerek ún. „legkisebb négyzetes megoldásának" a meghatározása fontos feladat, amelynek megoldását a 28.5. alfejezetben fogjuk bemutatni.

Most térjünk vissza az alapfeladatunkra, az n lineáris egyenletet és n ismeretlent tartalmazó $Ax = b$ rendszer megoldására. Egy lehetséges megközelítés a következő: kiszámítjuk A^{-1}-et és megszorozzuk az egyenlet mindkét oldalát A^{-1}-gyel, amiből $A^{-1}Ax = A^{-1}b$, azaz $x = A^{-1}b$ adódik. E módszer hátránya a ***numerikus instabilitás.*** Szerencsére van egy másik lehetőség is – az LUP felbontás módszere –, amelyik numerikusan stabil. Ennek a módszernek további előnye az, hogy az előzőnél jóval gyorsabb.

Az LUP felbontás módszerének áttekintése

Az LUP felbontás módszerének alapgondolata az, hogy keressünk olyan $n \times n$-es L, U és P mátrixokat, amelyekkel fennáll a

$$PA = LU \tag{28.19}$$

egyenlőség, ahol

- L alsó háromszög-főmátrix,
- U felső háromszögmátrix,
- P permutációs mátrix.

A (28.19) egyenletet kielégítő L, U és P mátrixokat az A mátrix egy ***LUP felbontásának*** nevezzük. Meg fogjuk mutatni, hogy minden nemszinguláris mátrixnak van ilyen felbontása.

Az A mátrix LUP felbontását azért célszerű kiszámítani, mert könnyű megoldani azokat a lineáris rendszereket, amelyekben az együtthatómátrix háromszögmátrix. (Az L és az U is ilyen mátrixok.) Háromszögrendszernek fogjuk nevezni az ilyen lineáris egyenletrendszereket. Az A mátrix LUP felbontásának az ismeretében a (28.17)-beli $Ax = b$ egyenlet megoldását két háromszögrendszer megoldására vezetjük vissza a következő módon. Az $Ax = b$ mindkét oldalát P-vel megszorozva, a vele ekvivalens $PAx = Pb$ egyenlet adódik, ami a 28.1-5. gyakorlat szerint a (28.16) egyenletek egy átrendezésével egyenértékű. A (28.19) felbontás felhasználásával innen az

$$LUx = Pb$$

egyenlőséget kapjuk. Ez pedig két háromszögrendszerre bomlik. Legyen $y = Ux$, ahol x a keresett megoldásvektor. Először megoldjuk az

$$Ly = Pb \tag{28.20}$$

alsó háromszögrendszert az y ismeretlen vektorra, az ún. „előre helyettesítés" módszerrel. Az y vektor ismeretében megoldjuk az

$$Ux = y \tag{28.21}$$

felső háromszögrendszert az x ismeretlen vektorra, az ún. „visszahelyettesítés” módszerrel. Az így kapott x vektor megoldása $Ax = b$-nek, mivel a P permutációs mátrix invertálható (lásd a 28.1-5. gyakorlatot):

$$\begin{aligned} Ax &= P^{-1}LUx \\ &= P^{-1}Ly \\ &= P^{-1}Pb \\ &= b. \end{aligned}$$

A továbbiakban először az előre helyettesítés és a visszahelyettesítés módszereket ismertetjük, majd utána mutatjuk meg azt, hogyan lehet meghatározni egy mátrix LUP felbontását.

Az előre helyettesítés és a visszahelyettesítés módszere

Adott L, P és b esetén az ***előre helyettesítés*** módszerével a (28.20) alsó háromszögrendszert $\Theta(n^2)$ lépésben oldhatjuk meg. A P permutációs mátrixot az egyszerűség kedvéért egy $\pi[1..n]$ tömbbel írjuk le: $i = 1, 2, \ldots, n$ esetén a $\pi[i]$ elem azt jelenti, hogy $P_{i,\pi[i]} = 1$ és $P_{ij} = 0$, ha $j \neq \pi[i]$. Eszerint a PA mátrix i-edik sorában és j-edik oszlopában álló elem $a_{\pi[i],j}$, és a Pb vektor i-edik eleme $b_{\pi[i]}$. Mivel L alsó háromszög-főmátrix, ezért a (28.20) egyenlet átírható az

$$\begin{array}{rcrcrcrcrcl} y_1 & & & & & & & & & = & b_{\pi[1]}, \\ l_{21}y_1 & + & y_2 & & & & & & & = & b_{\pi[2]}, \\ l_{31}y_1 & + & l_{32}y_2 & + & y_3 & & & & & = & b_{\pi[3]}, \\ & & & & & & & & & \vdots & \\ l_{n1}y_1 & + & l_{n2}y_2 & + & l_{n3}y_3 & + & \cdots & + & l_{nn}y_n & = & b_{\pi[n]} \end{array}$$

alakra. Az első egyenlet szerint $y_1 = b_{\pi[1]}$. Behelyettesítve ezt a második egyenletbe azt kapjuk, hogy

$$y_2 = b_{\pi[2]} - l_{21}y_1.$$

Most y_1-et és y_2-t írjuk be a harmadik egyenletbe:

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2).$$

Általánosan, ha az $y_1, y_2, \ldots, y_{i-1}$ értékeket behelyettesítjük az i-edik egyenletbe, akkor azt kapjuk, hogy

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j.$$

A ***visszahelyettesítés*** az előre helyettesítéshez hasonló. Adott U és y esetén megoldjuk az egyenleteket az n-ediktől visszafelé az elsőig. Az előre helyettesítéshez hasonlóan ennek a módszernek is $\Theta(n^2)$ a futási ideje. Mivel U felső háromszögmátrix, ezért a (28.21)

rendszert átírhatjuk az

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1,\\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2,\\
&\vdots\\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2},\\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1},\\
u_{nn}x_n &= y_n
\end{aligned}$$

alakba. Az $x_n, x_{n-1}, \ldots, x_1$ ismeretlenek tehát egymás után meghatározhatók

$$\begin{aligned}
x_n &= \frac{y_n}{u_{nn}},\\
x_{n-1} &= \frac{y_{n-1} - u_{n-1,n}x_n}{u_{n-1,n-1}},\\
x_{n-2} &= \frac{y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n)}{u_{n-2,n-2}},\\
&\vdots
\end{aligned}$$

illetve általában

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}}.$$

Az LUP-MEGOLD eljárás adott P, L, U és b esetén x-et az előre helyettesítés és a visszahelyettesítés kombinálásával határozza meg. A pszeudokód feltételezi, hogy az n dimenzió a $sorok[L]$ attribútumban, a P permutációs mátrix pedig a π tömbben található.

LUP-MEGOLD(L, U, π, b)

```
1  n ← sorok[L]
2  for i ← 1 to n
3       do y_i ← b_π[i] − Σ_{j=1}^{i−1} l_ij y_j
4  for i ← n downto 1
5       do x_i ← (y_i − Σ_{j=i+1}^{n} u_ij x_j)/u_ii
6  return x
```

Az LUP-MEGOLD eljárás y-t a 2–3. sorokban előre helyettesítéssel, majd x-et a 4–5. sorokban visszahelyettesítéssel számítja ki. Mivel mindegyik **for** cikluson belül még egy implicit összegezési ciklus is található, ezért a futási idő $\Theta(n^2)$.

Példaként tekintsük az

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix}$$

lineáris egyenletrendszert, ahol

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$
$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix}$$

és x az ismeretlen vektor. Az LUP felbontás a következő:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0,2 & 1 & 0 \\ 0,6 & 0,5 & 1 \end{pmatrix},$$
$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0,8 & -0,6 \\ 0 & 0 & 2,5 \end{pmatrix},$$
$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(A $PA = LU$ egyenlőség könnyen ellenőrizhető.) Most előre helyettesítéssel megoldjuk $Ly = Pb$-t y-ra:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0,2 & 1 & 0 \\ 0,6 & 0,5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

ahonnan először y_1, majd y_2, végül y_3 kiszámolása után azt kapjuk, hogy

$$y = \begin{pmatrix} 8 \\ 1,4 \\ 1,5 \end{pmatrix}.$$

Ezután az $Ux = y$ egyenletet oldjuk meg x-re a visszahelyettesítés módszerével:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0,8 & -0,6 \\ 0 & 0 & 2,5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1,4 \\ 1,5 \end{pmatrix},$$

ahonnan először x_3, majd x_2, végül x_1 kiszámolása után azt kapjuk, hogy

$$x = \begin{pmatrix} -1,4 \\ 2,2 \\ 0,6 \end{pmatrix}.$$

Egy LU felbontás meghatározása

Korábban már láttuk azt, hogy ha ismerjük egy nemszinguláris A mátrix LUP felbontását, akkor az előre helyettesítés és a visszahelyettesítés módszerét felhasználhatjuk az $Ax = b$

lineáris egyenletrendszer megoldására. Meg kell még mutatnunk azt, hogy az A mátrix egy LUP felbontását hogyan lehet hatékonyan meghatározni. Azzal az esettel kezdjük, amikor A nemszinguláris $n \times n$-es mátrix és P nincs jelen, azaz $P = I_n$. Ebben az esetben az $A = LU$ szorzat tényezőit kell meghatároznunk. Ilyen L és U mátrixokat az A mátrix egy ***LU felbontásának*** nevezzük.

Az A mátrix egy LU felbontását az ún. ***Gauss-féle kiküszöbölési eljárással*** fogjuk meghatározni. Tekintsük az $Ax = b$ egyenletet. Az első egyenlet alkalmas számszorosait vonjuk ki a többi egyenletből azzal a céllal, hogy az első változót a többi egyenletből kiküszöböljük. Ezután a második egyenlet számszorosait vonjuk ki a többiből, kiküszöbölendő az első két változót. Ezt az eljárást folytatva végül egy felső háromszögrendszert kapunk. Ennek az egyenletrendszernek az együtthatómátrixa lesz az U mátrix. Az L mátrix a szóban forgó szorzókból épül fel.

E stratégiát egy rekurzív algoritmussal valósítjuk meg. Feladatunk tehát az $n \times n$-es nemszinguláris A mátrix egy LU felbontásának a meghatározása. Ha $n = 1$, akkor készen vagyunk, mert $L = I_1$ és $U = A$ választható. Ha $n > 1$, akkor A-t négy részre bontjuk:

$$\begin{aligned} A &= \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{array}\right) \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}, \end{aligned}$$

ahol v egy $(n-1)$ dimenziós oszlopvektor, w^{T} egy $(n-1)$ dimenziós sorvektor, A' pedig egy $(n-1) \times (n-1)$-es mátrix. Az A mátrixot az alábbi módon két mátrix szorzataként írhatjuk fel

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}. \end{aligned} \tag{28.22}$$

(A szorzás elvégzésével igazolható az egyenlőség.) A nullák a fenti mátrixokban $(n-1)$-edrendű sor-, illetve oszlopvektorok. A vw^{T}/a_{11} tag, amely v és w külső szorzatának és a_{11}-nek a hányadosa, egy $(n-1) \times (n-1)$-es mátrix, így mérete megegyezik A' méretével, amelyből kivonjuk. A kapott $(n-1) \times (n-1)$-es

$$A' - \frac{vw^{\mathrm{T}}}{a_{11}} \tag{28.23}$$

mátrixot az A mátrix a_{11}-re vonatkozó ***Schur-komplemensének*** nevezzük.

Most azt igazoljuk, hogy a nemszinguláris A mátrix Schur-komplemense is nemszinguláris. Ezt indirekt módon bizonyítjuk. Az állítással ellentétben tegyük fel azt, hogy az A mátrix $(n-1)\times(n-1)$-es Schur-komplemense szinguláris. A 28.1. tétel alapján ennek rangja kisebb, mint $n-1$. Mivel az

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

mátrix első oszlopának utolsó $n-1$ eleme nulla, ezért utolsó $n-1$ sorának a sorrangja kisebb, mint $n-1$. Az egész mátrix rangja tehát kisebb, mint n. A 28.1-10. gyakorlatban megfogalmazott állítást (28.22)-re alkalmazva azt kapjuk, hogy A rangja kisebb, mint n. A 28.1. tétel alapján tehát arra az ellentmondásra jutottunk, hogy A szinguláris.

Mivel a Schur-komplemens nemszinguláris, ezért rekurzív úton meghatározhatjuk egy LU felbontását. Legyen

$$A' - \frac{vw^{\mathrm{T}}}{a_{11}} = L'U',$$

ahol L' alsó háromszög főmátrix és U' felső háromszögmátrix. A mátrixalgebra összefüggéseit felhasználva azt kapjuk, hogy

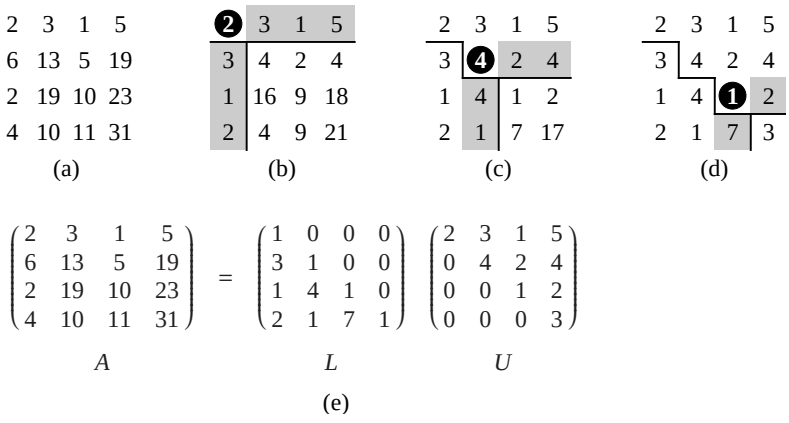
$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

ami a keresett LU felbontást adja. (Mivel L' alsó háromszög-főmátrix, ezért L is az, és U' felső háromszög jellegéből ugyancsak következik, hogy U is felső háromszögmátrix.)

A módszer $a_{11} = 0$ esetén nem használható, mert ekkor nullával kellene osztani. Ugyanez a helyzet, ha az $A' - vw^{\mathrm{T}}/a_{11}$ Schur-komplemens bal felső eleme nulla, mert a vele való osztásra a rekurzió következő lépésében kerül sor. Azokat az elemeket, amelyekkel az LU felbontás során osztunk, ***főelemeknek*** (***pivot elemeknek***) nevezzük – ezek az U mátrix főátlójában helyezkednek el. A P permutációs mátrixot éppen azért szerepeltetjük az LUP felbontásban, hogy az osztási hibát kiküszöböljük. Permutációknak a 0-val (illetve kicsi számmal) való osztás elkerülése céljából történő használatát ***főelemkiválasztásnak*** (***pivotálásnak***) nevezzük.

Mátrixoknak egy olyan fontos osztálya, amelyre az LU felbontás mindig helyesen működik, a szimmetrikus pozitív definit mátrixok. Ilyen mátrixok esetén nincs szükség főelemkiválasztásra, így a fent vázolt rekurzív stratégia a 0-val való osztás veszélye nélkül alkalmazható. Ezt – néhány más eredménnyel együtt – a 28.5. alfejezetben fogjuk bizonyítani.

Az LU felbontás megvalósítására szolgáló programrészletünk a rekurzív stratégiát követi. A rekurziót egy iterációs ciklus helyettesíti. (Ez egy szokásos optimalizálás az ún. „végén rekurzív” eljárások esetén, azaz olyankor, amikor az utolsó művelet egy rekurzív hívás.) Feltesszük, hogy A dimenzióját a *sorok*[A] attribútum tartalmazza. Mivel tudjuk, hogy a kimenő U mátrix főátló alatti elemei nullák, amelyeket LU-MEGOLD nem használ, a program ezekkel nem foglalkozik. Hasonlóan, mivel a kimenő L mátrix főátlójában 1-esek, felette pedig 0-k állnak, ezeket az elemeket sem kell feltöltenünk. Így L-nek és U-nak csak a „lényeges” elemeit számítjuk ki.



28.1. ábra. Az LU-FELBONTÁS eljárás. **(a)** Az A mátrix. **(b)** A fekete körben jelzett $a_{11} = 2$ elem a főelem, a szürke oszlop v/a_{11}, a szürke sor w^{T}. U-nak a már kiszámított elemei a vízszintes vonal fölött, L elemei pedig a függőleges vonaltól balra helyezkednek el. Az $A' - vw^{\mathrm{T}}/a_{11}$ Schur-komplemens a jobb alsó blokkot foglalja el. **(c)** Most a (b) részben előállított Schur-komplemenssel foglalkozunk. A feketével jelzett $a_{22} = 4$ elem a főelem, a szürke oszlop és sor pedig v/a_{22}, illetve w^{T}. A vonalak a mátrixot U-nak az eddig kiszámított elemeire (felül), L-nek az eddig meghatározott elemeire (balról) és az új (jobb alsó) Schur-komplemensre osztják. **(d)** A következő lépés a felbontás utolsó lépése. (Az új Schur-komplemensben a 3-as elem az U mátrix része lesz a rekurzió befejezése után.) **(e)** Az $A = LU$ felbontás.

LU-FELBONTÁS(A)

```
 1  n ← sorok[A]
 2  for k ← 1 to n
 3       do u_kk ← a_kk
 4            for i ← k + 1 to n
 5                 do   l_ik ← a_ik/u_kk          ▷ l_ik-ban v_i
 6                      u_ki ← a_ki               ▷ u_ki-ban w_i^T
 7            for i ← k + 1 to n
 8                 do for i ← k + 1 to n
 9                         do a_ij ← a_ij − l_ik u_kj
10  return L meg U
```

A 2. sorbeli külső **for** ciklus egyszer fut le minden rekurzív lépés esetén. E cikluson belül a 3. sorban számítjuk ki az $u_{kk} = a_{kk}$ főelemet. A 4–6. sorokban található **for** ciklus (amely $k = n$-re nem fut le) számítja ki a v és w^{T} vektorokat, az L és U mátrixokat módosítandó. A v vektor elemeit az 5. sorban számítjuk ki, majd v_i-t l_{ik}-ban tároljuk; hasonlóan a w^{T} vektor elemeit az 6. sorban számoljuk, miközben u_{ki} fogja tartalmazni a w_i értéket. Végül, a 7–9. sorokban számítjuk ki a Schur-komplemens elemeit, és ismét az A mátrixban tároljuk őket. (A 9. sorban nem kell a_{kk}-val osztani, mert ezt az 5. sorban az l_{ik} kiszámolásánál már megtettük.) Mivel a 9. sorbeli ciklus háromszorosan beágyazott, az LU-FELBONTÁS eljárás futási ideje $\Theta(n^3)$.

A 28.1. ábra az LU-FELBONTÁS eljárást illusztrálja. Mutatja az eljárás szokásos optimalizálását, amely szerint az L és U mátrixok az A mátrix „helyére” kerülnek. Ez azt jelenti, hogy megfeleltetést létesítünk az a_{ij} elemek, valamint az l_{ij} (ha $i > j$), illetve az u_{ij} (ha

$i \le j$) elemek között oly módon, hogy az eljárás végén A tartalmazza mind L-et, mind U-t. Nem nehéz belátni, hogy ennek az eljárásnak a pszeudokódja a fentiből úgy kapható, hogy az l-re, illetve u-ra történő hivatkozást mindenütt egyszerűen a-ra cseréljük.

Egy LUP felbontás kiszámítása

Az $Ax = b$ lineáris egyenletrendszer megoldása során a 0-val való osztás elkerülésére nem főátlóbeli főelemeket kell választanunk. Eközben nemcsak a 0-val, hanem a kis számmal történő osztás is elkerülendő, még nemszinguláris mátrixok esetén is, mert ez numerikus instabilitást okozhat. Ezért nagy főelemeket próbálunk választani.

Az LUP felbontás mögötti matematika az LU felbontás matematikájához hasonló. Emlékeztetünk arra, hogy adott $n \times n$-es nemszinguláris A mátrix esetén most egy P permutációs mátrixot, egy L alsó háromszög-főmátrixot és egy U felső háromszögmátrixot keresünk úgy, hogy $PA = LU$ teljesüljön. Mielőtt felosztanánk az A mátrixot az LU felbontásban megismert módon, először bemozgatunk egy nemnulla elemet, mondjuk a_{k1}-et az első oszlopból a mátrix $(1, 1)$ pozíciójába. (Ha az első oszlopban csak nulla elemek vannak, akkor A szinguláris, mert a 28.4. és 28.5. tételek alapján a determinánsa 0.) Az egyenletrendszer megtartása céljából az 1. sort kicseréljük a k-adikkal, ami egyenértékű azzal, hogy az A mátrixot balról egy Q permutációs mátrixszal szorozzuk (lásd a 28.1-5. gyakorlatot). Így QA-t

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}$$

alakba írhatjuk, ahol $v = (a_{21}, a_{31}, \ldots, a_{n1})^{\mathrm{T}}$, kivéve, hogy a_{11}-et a_{k1} helyettesíti, $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})^{\mathrm{T}}$ és A' egy $n \times n$-es mátrix. Mivel $a_{k1} \neq 0$, ezért az LU felbontás során használt lineáris algebrai módszerekkel (tudva, hogy nem osztunk 0-val), azt kapjuk, hogy

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix}. \end{aligned}$$

Az LU felbontásnál láttuk, hogy a nemszinguláris A mátrix $A' - vw^{\mathrm{T}}/a_{k1}$ Schur-komplemense is nemszinguláris. Tehát – indukcióval – megkereshetjük a Schur-komplemens LUP felbontását, azaz olyan L' alsó háromszög főmátrixot, U' felső háromszögmátrixot és P' permutációs mátrixot, amelyekre

$$P'(A' - vw^{\mathrm{T}}/a_{k1}) = L'U'.$$

Vezessük be a

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q$$

mátrixot, amely egy permutációs mátrix, mivel két permutációs mátrix szorzata (lásd a 28.1-5. gyakorlatot). Ekkor

$$
\begin{aligned}
PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\
&= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1}) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\
&= LU.
\end{aligned}
$$

Mivel L' alsó háromszög-főmátrix, azért L is az, és mivel U' felső háromszögmátrix, azért U szintén az. Tehát a fenti egyenlőségek az A mátrix LUP felbontását adják.

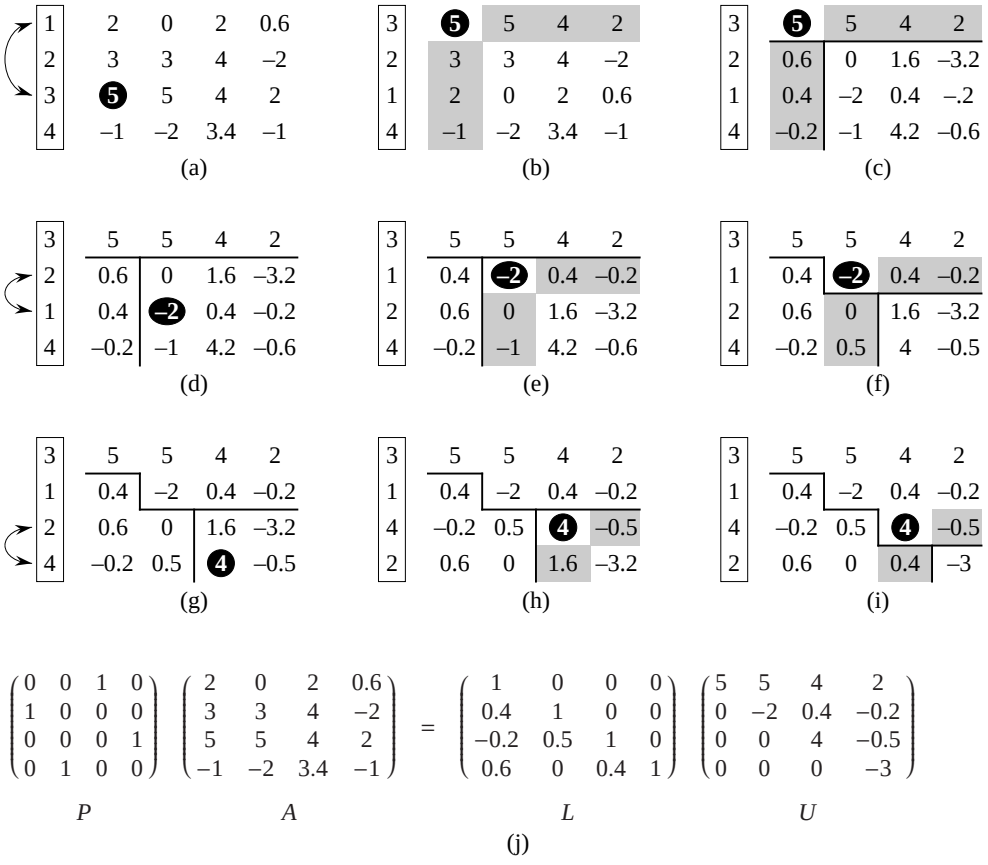
Vegyük észre, hogy ebben a levezetésben – ellentétben az LU felbontással – a v/a_{k1} oszlopvektort és az $A' - vw^{\mathrm{T}}/a_{k1}$ Schur-komplemenst is szorozni kell a P' permutációs mátrixszal.

A LUP-felbontás pszeudokódja a következő.

LUP-FELBONTÁS(A)

```
 1  n ← sorok[A]
 2  for i ← 1 to n
 3      do π[i] ← i
 4  for k ← 1 to n − 1
 5      do p ← 0
 6          for i ← k to n
 7              do if |a_ik| > p
 8                    then p ← |a_ik|
 9                         k′ ← i
10          if p = 0
11             then error „szinguláris mátrix”
12          π[k] ↔ π[k′] csere
13          for i ← 1 to n
14              do a_ki ↔ a_k′i csere
15          for i ← k + 1 to n
16              do a_ik ← a_ik/a_kk
17                 for j ← k + 1 to n
18                     do a_ij ← a_ij − a_ik a_kj
```

Az LU-FELBONTÁS eljáráshoz hasonlóan, az LUP felbontásra szolgáló algoritmusban is egy iterációs ciklussal helyettesítjük a rekurziót. Ennek közvetlen megvalósítása helyett a P permutációs mátrixot dinamikusan egy π tömbben tároljuk, ahol $\pi[i] = j$ azt jelenti, hogy P i-edik sora a j-edik oszlopban 1-et tartalmaz. Továbbá, L-et és U-t A-nak a „helyén”



28.2. ábra. Az LUP-FELBONTÁS eljárás. **(a)** Az A bemenő mátrix a sorok kezdeti (egységmátrixnak megfelelő) permutációjával a bal oldalon. Az algoritmus első lépése az első oszlop harmadik sorában a feketével jelzett 5-öst választja ki főelemként. (**b**) Az 1. és 3. sorokat felcseréljük, és a permutációt is módosítjuk. A szürke oszlop és sor v és w^{T}. (**c**) A v oszlopot $v/5$-re cseréljük, és a jobb alsó részt a Schur-komplemensre cseréljük. A vonalak a mátrixot három részre osztják: U-ra (felül), L-re (balról) és a Schur-komplemensre, a jobb alsó blokkra. (**d**)–(**f**) A második lépés. (**g**)–(**i**) Az algoritmus harmadik, befejező lépése. (**j**) Az LUP felbontás: $PA = LU$.

képezzük. Így, amikor az eljárás véget ér,

$$a_{ij} = \begin{cases} l_{ij}, & \text{ha } i > j \\ u_{ij}, & \text{ha } i \le j. \end{cases}$$

A 28.2. ábrán az eljárást illusztráljuk. A 2–3. sorokban beállítjuk a π tömböt az egység mátrixnak megfelelő permutációra. A 4. sorban kezdődő külső ***for*** ciklus felel meg a rekurziónak. Az 5–9. sorokban minden alkalommal meghatározzuk az aktuális $(n-k+1)\times(n-k+1)$-es mátrix LU felbontásához az $a_{k'k}$ főelemet mint a mátrix első (az eredeti mátrix k-adik) oszlopának a legnagyobb abszolút értékű elemét. Ha az aktuális első oszlop minden eleme nulla, a 10–11. sorok jelzik a szingularitást. A 12. sorban $\pi[k]$-t $\pi[k']$-re, a 13–14. sorokban pedig a k-adik sort a k'-edik sorra cseréljük. (Az egész sor részt vesz a cserében, hiszen – mint az a levezetésből látható – nemcsak $A' - vw^{\mathrm{T}}/a_{k1}$, hanem v/a_{k1} is szorzódik P'-vel.)

Végül, a 15–18. sorokban a Schur-komplemenst számítjuk ki, úgy, ahogy az LU-FELBONT ÁS 4–9. soraiban tettük, azzal a különbséggel, hogy most a műveleteket „helyben” végezzük.

Az LUP-FELBONTÁS futási ideje a háromszorosan egymásba ágyazott ciklusokat figyelembe véve $\Theta(n^3)$, azaz annyi, mint az LU-FELBONTÁS-é. A főelemkiválasztás tehát a futási időt legfeljebb egy állandó szorzóval növeli.

Gyakorlatok

28.3-1. Oldjuk meg az

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

egyenletet előre helyettesítéssel.

28.3-2. Adjuk meg az alábbi mátrix egy LU felbontását:

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

28.3-3. LUP felbontással oldjuk meg a következő egyenletrendszert:

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}.$$

28.3-4. Adjuk meg diagonális mátrix egy LUP felbontását.

28.3-5. Adjuk meg egy permutációs mátrix egy LUP felbontását, és lássuk be az egyértelműséget.

28.3-6. Mutassuk meg, hogy minden $n \geq 1$ esetén léteznek olyan szinguláris $n \times n$-es mátrixok, amelyeknek van LU felbontása.

28.3-7. Az LU-FELBONTÁS-ban végre kell-e hajtani a legkülső **for** ciklust, ha $k = n$? Mi a helyzet az LUP-FELBONTÁS algoritmus esetén?

28.4. Mátrixok invertálása

Bár a gyakorlatban általában nem használjuk mátrix inverzét lineáris egyenletrendszer megoldására, mert előnyben részesítjük a numerikusan stabilabb technikákat (mint például az LUP felbontást), azért néha szükség lehet egy mátrix inverzének a kiszámítására. Ebben az alfejezetben megmutatjuk, hogy ezt az LUP felbontás segítségével hogyan tehetjük meg. Azt is meg fogjuk mutatni, hogy a mátrixszorzás és a mátrixinvertálás lényegében azonos nehézségű problémák. Ez azt jelenti, hogy ha az egyik megoldására ismerünk egy algoritmust, akkor ennek felhasználásával – bizonyos technikai jellegű feltételek teljesülése esetén – a másik megoldására is kapunk egy aszimptotikusan ugyanannyi futási idejű algoritmust. Így Strassen mátrixszorzási algoritmusát is felhasználhatjuk mátrix inverzének a meghatározására. Strassen dolgozatának a célja éppen annak megmutatása volt, hogy az eljárásával lineáris egyenletrendszereket az addigi szokásos módszereknél gyorsabban is meg tud oldani.

Mátrix inverzének a meghatározása egy LUP felbontásából

Tegyük fel, hogy ismerjük az A mátrix egy LUP felbontását, azaz olyan L, U és P mátrixokat, amelyekre $PA = LU$ teljesül. Az LU-MEGOLD eljárás felhasználásával az $Ax = b$ egyenlet $\Theta(n^2)$ futási idő alatt oldható meg. Mivel az LUP felbontás csak A-tól függ és b-től nem, ezért egy másik $Ax = b'$ lineáris egyenletrendszert további $\Theta(n^2)$ futási idő alatt tudunk megoldani. Általában, ha rendelkezésünkre áll A egy LUP felbontása, akkor $\Theta(kn^2)$ futási idő alatt k darab $Ax = b$ alakú olyan egyenletet tudunk megoldani, amelyek csupán b-ben, azaz a jobb oldalban különböznek egymástól.

Az

$$AX = I_n \tag{28.24}$$

egyenlet n darab különböző $Ax = b$ alakú egyenlet együtteseként is felfogható. Az X mátrix tehát az A inverze. Jelöljük X_i-vel az X mátrix i-edik oszlopát, e_i-vel pedig az I_n egységmátrix i-edik oszlopát. A (28.24) mátrixegyenletet X-re az A mátrix LUP felbontásával úgy oldhatjuk meg, hogy külön-külön megoldjuk az

$$AX_i = e_i$$

egyenleteket X_i-re. Az n darab X_i mindegyikét $\Theta(n^2)$ idő alatt határozhatjuk meg, ezért X kiszámítása az A mátrix LUP felbontásának ismeretében $\Theta(n^3)$ időt vesz igénybe. Mivel A-nak az LUP felbontása is $\Theta(n^3)$ idő alatt határozható meg, ezért A^{-1} kiszámításának ugyanennyi, azaz $\Theta(n^3)$ a futási ideje.

Mátrixszorzás és mátrixinvertálás

Most megmutatjuk, hogy a mátrixszorzásra nyert elméleti gyorsítás átvihető a mátrixinvertálásra is. Valójában ennél többet fogunk bizonyítani. Nevezetesen azt, hogy a mátrixszorzás ekvivalens a mátrixinvertálással a következő értelemben. Ha $M(n)$ jelöli két $n \times n$-es mátrix összeszorzásának a futási idejét, akkor $n \times n$-es mátrixot $O(M(n))$ futási idő alatt is invertálhatunk (röviden: „az invertálás nem nehezebb, mint a szorzás”). Fordítva, ha $I(n)$ jelöli egy $n \times n$-es nemszinguláris mátrix invertálására fordított időt, akkor két $n \times n$-es mátrixot $O(I(n))$ futási idő alatt is összeszorozhatunk (röviden: „a szorzás nem nehezebb, mint az invertálás”). Az ekvivalenciára vonatkozó állítást két tételben fogalmazzuk meg.

28.7. tétel (a szorzás nem nehezebb, mint az invertálás). *Ha egy $n \times n$-es mátrixot $I(n)$ idő alatt tudunk invertálni, ahol $I(n) = \Omega(n^2)$, és $I(n)$ kielégíti az $I(3n) = O(I(n))$ regularitási feltételt, akkor két $n \times n$-es mátrixot $O(I(n))$ idő alatt is összeszorozhatunk.*

Bizonyítás. Jelölje A és B azokat az $n \times n$-es mátrixokat, amelyeknek a C szorzatát ki akarjuk számítani. Tekintsük az alábbi $3n \times 3n$ méretű D mátrixot:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Mivel D inverze

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

ezért a keresett AB szorzat D^{-1} jobb felső $n \times n$-es részmátrixa.

A D mátrixot $\Theta(n^2) = O(I(n))$, az inverzét pedig az I_n-re kirótt regularitási feltétel miatt $O(I(3n)) = O(I(n))$ futási idő alatt határozhatjuk meg. Így $M(n) = O(I(n))$. ■

Vegyük észre, hogy $I(n)$ kielégíti a regularitási feltételt, ha minden $c > 0$ és $d \geq 0$ számra $I(n) = \Theta(n^c \lg^d n)$ teljesül.

A fordított állítás igazolásánál felhasználjuk szimmetrikus pozitív definit mátrixok bizonyos tulajdonságait. Ezeket majd a 28.5. alfejezetben fogjuk bebizonyítani.

28.8. tétel (az invertálás nem nehezebb, mint a szorzás). *Tegyük fel, hogy két $n \times n$-es valós mátrixot $M(n)$ idő alatt tudunk összeszorozni, ahol $M(n) = \Omega(n^2)$, továbbá $M(n)$ kielégíti az $M(n+k) = O(M(n))$ $(0 \leq k \leq n)$ és $M(n/2) \leq cM(n)$ $(c < 1/2)$ regularitási feltételeket. Akkor bármelyik $n \times n$-es valós nemszinguláris mátrix inverzét $O(M(n))$ futási idő alatt is kiszámíthatjuk.*

Bizonyítás. Az általánosság megszorítása nélkül feltehetjük, hogy n egy kettőhatvány. Ennek igazolásához először azt jegyezzük meg, hogy minden $k > 0$ egész számra

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}.$$

Ha n nem kettőhatvány, akkor válasszuk meg k-t úgy, hogy $n+k$ az n-et követő kettőhatvány legyen, és fenti módon növeljük meg az A mátrixot. Ennek inverzéből A^{-1} már meghatározható. Az $M(n)$-re kirótt első regularitási feltétel biztosítja, hogy ez a méretnövelés a futási időnek legfeljebb egy állandó tényezővel való növekedését eredményezi.

Most tegyük fel ideiglenesen azt, hogy az $n \times n$-es A mátrix szimmetrikus és pozitív definit. Osszuk fel A-t négy $n/2 \times n/2$ méretű részmátrixra:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix}. \tag{28.25}$$

Legyen

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.26}$$

az A mátrix B-re vonatkozó Schur-komplemense (lásd a 28.5. alfejezetet). Az $AA^{-1} = I_n$ egyenlőség felhasználásával ellenőrizhetjük azt, hogy

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}. \tag{28.27}$$

Mivel A szimmetrikus és pozitív definit, ezért a 28.5. alfejezetbeli 28.9., 28.10. és 28.11. lemmák miatt a B^{-1} és az S^{-1} mátrixok léteznek, ugyanis ekkor B és S is szimmetrikus és pozitív definit. A 28.1-2. gyakorlat szerint $B^{-1}C^{\mathrm{T}} = (CB^{-1})^{\mathrm{T}}$ és $B^{-1}C^{\mathrm{T}}S^{-1} = (S^{-1}CB^{-1})^{\mathrm{T}}$. A (28.26) és a (28.27) egyenlőségeket felhasználhatjuk tehát egy olyan rekurzív algoritmus megadására, amely az alábbi négy $n/2 \times n/2$-es mátrixszorzást tartalmazza:

$$C \cdot B^{-1},$$
$$(CB^{-1}) \cdot C^{\mathrm{T}},$$
$$S^{-1} \cdot (CB^{-1}),$$
$$(C \cdot B^{-1})^{\mathrm{T}} \cdot (S^{-1} \cdot CB^{-1}).$$

Ezek szerint egy $n \times n$-es szimmetrikus pozitív definit mátrix inverzét meghatározhatjuk úgy, hogy kiszámítjuk az $n/2 \times n/2$ méretű B és S mátrixok inverzét, egy $n \times n$-es mátrixszorzó algoritmussal meghatározzuk az $n/2 \times n/2$-es mátrixokat tartalmazó fenti négy szorzatot, majd $O(n^2)$ művelettel előállítjuk A részmátrixait. Az így kapott mátrixokból csak állandó számú összeadást és kivonást alkalmazva (28.27) alapján megkapjuk az inverzet. Erre az eljárásra a következő rekurzív relációt kapjuk:

$$\begin{aligned} I(n) &\leq 2I\left(\frac{n}{2}\right) + 4M(n) + O(n^2) \\ &= 2I\left(\frac{n}{2}\right) + \Theta(M(n)) \\ &= O(M(n)). \end{aligned}$$

Az első egyenlőség az $M(n) = \Omega(n^2)$ feltételünk miatt igaz. A második regularitási feltételünkből következik, hogy a 4.1. tétel 3. esete alkalmazható, ezért a fenti második egyenl ős ég is teljesül.

Hátra van még annak az esetnek a vizsgálata, amikor A invertálható, de nem szimmetrikus és pozitív definit. Az alapötlet az, hogy tetszőleges nemszinguláris A mátrix invertálásának a problémáját visszavezetjük $A^{\mathrm{T}}A$ invertálásának a problémájára. A 28.1-2. gyakorlat szerint ugyanis minden nemszinguláris A mátrix esetén $A^{\mathrm{T}}A$ szimmetrikus és a 28.6. tétel miatt pozitív definit, aminek az inverzét a fentiek szerint határozhatjuk meg.

A visszavezetés alapja az az észrevétel, hogy ha A egy $n \times n$-es nemszinguláris mátrix, akkor

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}.$$

Ez következik az $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$ egyenlőségből és a mátrix inverzének az egyértelműségéből. Tehát A^{-1}-et kiszámíthatjuk úgy, hogy először az A^{T} mátrixot megszorozzuk A-val, aztán a fenti oszd-meg-és-uralkodj algoritmussal invertáljuk a pozitív definit $A^{\mathrm{T}}A$ mátrixot, és végül az eredményt megszorozzuk A^{T}-vel. E három lépés mindegyikének a futási ideje $O(M(n))$, azaz minden nemszinguláris, valós elemű mátrixot $O(M(n))$ futási idő alatt is invertálhatunk. ■

A 28.8. tétel bizonyítása az $Ax = b$ egyenlet főelemkiválasztás nélküli megoldására (nemszinguláris A esetén) a következő módszert sugallja. Szorozzuk meg az egyenlet mindkét oldalát A^{T}-val: $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b$. Mivel A^{T} invertálható, ezért ez az egyenlet ekvivalens az eredetivel. Határozzuk meg a szimmetrikus pozitív definit $A^{\mathrm{T}}A$ mátrixnak egy LU felbontását, majd az előre helyettesítés és a visszahelyettesítés segítségével számítsuk ki az x megoldást. Bár ez a módszer elméletileg helyes, a gyakorlatban az LUP-felbontás-t érdemes használni, mert ez kevesebb aritmetikai műveletet igényel, és jobb numerikus tulajdonságokkal is rendelkezik.

Gyakorlatok

28.4-1. Legyen $M(n)$ két $n \times n$-es mátrix szorzásának, $S(n)$ pedig egy $n \times n$-es mátrix négyzetre emelésének a futási ideje. Mutassuk meg, hogy két mátrix szorzása és egy mátrix négyzetre emelése lényegében azonos nehézségű, azaz bizonyítsuk be, hogy egy $M(n)$ futási idejű mátrixszorzási algoritmusból egy $O(S(n))$ futási idejű négyzetre emelési algoritmus is adódik, és egy $S(n)$ idejű négyzetre emelési algoritmusból egy $O(S(n))$ futási idejű mátrixszorzási algoritmus is következik.

28.4-2. Legyen $M(n)$ két $n \times n$-es mátrix szorzásának, $L(n)$ pedig egy $n \times n$-es mátrix LUP felbontása meghatározásának a futási ideje. Mutassuk meg, hogy két mátrix szorzása és az LUP felbontás meghatározása lényegében azonos nehézségű, azaz bizonyítsuk be, hogy egy $M(n)$ futási idejű mátrixszorzási algoritmusból egy $O(M(n))$ futási idejű LUP felbontási eljárás is adódik, és egy $L(n)$ idejű LUP felbontási algoritmusból egy $O(S(n))$ futási idejű mátrixszorzási eljárás következik.

28.4-3. Legyen $M(n)$ két $n \times n$-es mátrix szorzásának, $D(n)$ pedig egy $n \times n$-es mátrix determinánsa kiszámításának a futási ideje. Mutassuk meg, hogy két mátrix szorzása és egy mátrix determinánsának a meghatározása lényegében azonos nehézségű, azaz bizonyítsuk be, hogy egy $M(n)$ futási idejű mátrixszorzási algoritmusból egy $O(M(n))$ futási idejű determináns-algoritmus is adódik, és egy $D(n)$ idejű determináns-algoritmusból egy $O(D(n))$ futási idejű mátrixszorzási eljárás is következik.

28.4-4. Legyen $M(n)$ két $n \times n$-es Boole mátrix szorzásának, $T(n)$ pedig egy $n \times n$-es Boole mátrix tranzitív lezárása (lásd a 25.2. alfejezet) meghatározásának a futási ideje. Mutassuk meg, hogy egy $M(n)$ futási idejű szorzási algoritmusból következik egy $O(M(n) \lg n)$ idejű tranzitív lezárási algoritmus, és egy $T(n)$ idejű tranzitív lezárási eljárásból egy $O(T(n))$ futási idejű szorzási algoritmus is adódik.

28.4-5. Működik-e a 28.8. tételen alapuló mátrixinvertálási algoritmus, ha a mátrix elemei a modulo 2 vett egészek gyűrűjéből származnak? Adjunk magyarázatot.

28.4-6.⋆ Általánosítsuk a 28.8. tételbeli mátrixinvertálási algoritmust a komplex esetre, és mutassuk meg, hogy az általánosított algoritmus helyesen működik. (*Útmutatás.* Az A mátrix transzponáltja helyett használjuk az A^* ***konjugált transzponáltat,*** amely A transzponáltjából úgy adódik, hogy minden elemnek vesszük a komplex konjugáltját. Szimmetrikus mátrixok helyett pedig tekintsük az $A = A^*$ feltételt kielégítő ***önadjungált*** – vagy ***Hermite-féle*** – mátrixokat.)

28.5. Szimmetrikus pozitív definit mátrixok és a legkisebb négyzetes közelítés

Szimmetrikus pozitív definit mátrixoknak sok érdekes és hasznos tulajdonsága van. Például nemszingulárisak, az LU felbontásuk a 0-val való osztás veszélye nélkül elvégezhető. Ebben az alfejezetben megvizsgáljuk még néhány további fontos tulajdonságukat, és egy érdekes alkalmazásként bemutatjuk a legkisebb négyzetes közelítéssel történő görbeillesztést.

Az első tulajdonság talán a legfontosabb.

28.9. lemma. *Minden pozitív definit mátrix nemszinguláris.*

Bizonyítás. Tegyük fel, hogy a mátrix szinguláris. A 28.3. következmény szerint ekkor létezik olyan nemnulla x vektor, amelyre $Ax = 0$, ezért $x^{\mathrm{T}}Ax = 0$. Ez azt jelenti, hogy A nem lehet pozitív definit. ■

Annak a bizonyítása, hogy az LU felbontás szimmetrikus pozitív definit mátrixok esetén 0-val való osztás nélkül elvégezhető, már nehezebb. Előzőleg néhány, az A mátrix részmátrixaira vonatkozó tulajdonságot bizonyítunk. Definiáljuk az A mátrix k-adik ***vezető részmátrixát,*** mint azt az A_k mátrixot, amely A első k sorának és első k oszlopának a kereszteződésében áll.

28.10. lemma. *Ha A szimmetrikus pozitív definit mátrix, akkor minden vezető részmátrixa szimmetrikus és pozitív definit.*

Bizonyítás. Nyilvánvaló, hogy minden vezető részmátrix szimmetrikus. Indirekt módon fogjuk megmutatni azt, hogy A_k pozitív definit. Ha A_k nem pozitív definit, akkor létezik olyan k dimenziós nemnulla x_k vektor, amelyre teljesül $x_k^{\mathrm{T}}Ax_k \leq 0$. Legyen A egy $n \times n$-es mátrix, és tekintsük az n dimenziós $x = \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix}^{\mathrm{T}}$ (itt x_k után $n-k$ darab nulla áll) vektort. Ekkor

$$\begin{aligned} x^{\mathrm{T}}Ax &= \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix}\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}\begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix}\begin{pmatrix} A_k x_k \\ Bx_k \end{pmatrix} \\ &= x_k^{\mathrm{T}}A_k x_k \\ &\leq 0, \end{aligned}$$

és ez ellentmond annak, hogy A pozitív definit. ■

Most megmutatjuk a Schur-komplemens néhány lényeges tulajdonságát. Legyen A egy szimmetrikus pozitív definit mátrix és A_k az A mátrixnak a $k \times k$ típusú vezető részmátrixa. Bontsuk fel A-t a következő módon:

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}. \tag{28.28}$$

A (28.23) definíció általánosításaként A-nak az A_k-ra vonatkozó ***Schur-komplemensét*** az

$$S = C - BA_k^{-1}B^{\mathrm{T}} \tag{28.29}$$

képlettel értelmezzük. (A 28.10. lemma szerint A_k szimmetrikus és pozitív definit, így a 28.9. lemma alapján A_k^{-1} létezik, ezért S jól definiált.) Vegyük észre, hogy a (28.29)-ben definiált S mátrix $k = 1$ esetén éppen a (28.23)-ban értelmezett Schur-komplemenssel egyezik meg.

A következő lemma szerint szimmetrikus pozitív definit mátrixok Schur-komplemens mátrixai maguk is szimmetrikus pozitív definit mátrixok. Ezt az eredményt a 28.8. tételben, a következményét pedig szimmetrikus pozitív definit mátrixok LU felbontásánál már felhasználtuk.

28.11. lemma (a Schur-komplemensről). *Ha A szimmetrikus pozitív definit mátrix és A_k az A mátrixnak a $k \times k$-s vezető részmátrixa, akkor A-nak A_k-ra vonatkozó Schur-komplemense szimmetrikus és pozitív definit.*

Bizonyítás. Mivel A szimmetrikus, ezért C is az. A 28.1-8. gyakorlat alapján a $BA_k^{-1}B^{\mathrm{T}}$ mátrix szimmetrikus, és a 28.1-1. gyakorlat miatt S is az.

Ezután már csak azt kell bizonyítanunk, hogy S pozitív definit. Tekintsük A-nak a (28.28)-beli felbontását. Az A pozitív definit, ezért minden nemnulla x vektorra $x^{\mathrm{T}}Ax > 0$. Bontsuk fel x-et az A_k és C méretének megfelelő y és z vektorokra. Mivel A_k^{-1} létezik, ezért

$$\begin{aligned} x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}} z & \mathbf{by} + Cz \end{pmatrix} \\ &= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + z^{\mathrm{T}}Cz \\ &= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z. \end{aligned} \tag{28.30}$$

Az egyenlőségeket mátrixszorzással ellenőrizhetjük. Az utolsó a kvadratikus alak „teljes négyzetté kiegészítésének” felel meg (lásd a 28.5-2. gyakorlat).

Mivel $x^{\mathrm{T}}Ax > 0$ minden nemnulla x esetén, ezért egy tetszőleges nemnulla z vektorhoz megválaszthatjuk y-t úgy, hogy $y = -A_k^{-1}B^{\mathrm{T}}z$ teljesüljön. Ekkor (28.30) első tagja eltűnik, és így a kifejezés értéke

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz.$$

Minden $z \neq 0$ esetén tehát $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, ami azt jelenti, hogy S pozitív definit. ■

28.12. következmény. *Szimmetrikus pozitív definit mátrix LU felbontása során soha nem kell* 0*-val osztanunk.*

Bizonyítás. Legyen A szimmetrikus pozitív definit mátrix. Kicsit többet bizonyítunk az állításnál. Azt látjuk be, hogy minden főelem pozitív. Az első főelem a_{11}. Ha e_1 az első egységvektor, akkor $a_{11} = e_1^{\mathrm{T}}Ae_1 > 0$. Mivel az LU felbontás első lépése A-nak $A_1 = (a_{11})$-re vonatkozó Schur-komplemensét adja, ezért a 28.11. lemmából indukcióval következik, hogy minden főelem pozitív. ■

Legkisebb négyzetes közelítés

Adott adathalmaz pontjaira jól illeszkedő görbe meghatározása a szimmetrikus pozitív definit mátrixok egyik fontos alkalmazása. Tegyük fel, hogy az m elemű adathalmaz

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$$

adatai mérési eredmények, és *csak* y_i-k rendelkeznek mérési hibákkal. Olyan $F(x)$ (közelítő) függvényt szeretnénk meghatározni, amelyik jól illeszkedik a fenti adatokra abban az értelemben, hogy minden $i = 1, 2, \ldots, m$ esetén

$$y_i = F(x_i) + \eta_i, \tag{28.31}$$

továbbá az η_i közelítési hibák (valamilyen értelemben) kicsinyek. Az F függvényt a szóban forgó adatoktól függő alakban célszerű keresni. Itt most azt tételezzük fel, hogy F egy lineárisan súlyozott összeg, azaz

$$F(x) = \sum_{j=1}^{n} c_j f_j(x).$$

A tagok n számát, valamint az f_j ***alapfüggvényeket*** az adatoktól függő módon lehet megválasztani. Szokásos választás az $f_j(x) = x^{j-1}$, ami azt jelenti, hogy az F függvény x-nek egy $(n-1)$-edfokú polinomja:

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}.$$

Ha $n = m$, akkor a fenti problémának $\eta_i = 0$-val $(i = 1, 2, \ldots, m)$ van egyértelmű megoldása. Egy ilyen magas fokú F „illeszkedik" az adatokhoz, azonban gyenge eredményeket adhat az x_i-ktől különböző x-ek esetén. Általában jobb n-et jelentősen kisebbre választani m-nél, és abban bízni, hogy a c_j együtthatók alkalmas választásával kaphatunk olyan F függvényt, ami úgy találja meg az adatokban rejlő törvényszerűségeket, hogy nem fordít feleslegesen nagy figyelmet a „zajra". Vannak elméleti megfontolások n megválasztására, de ezek túlmutatnak könyvünk keretein. Mindenesetre, ha n-et így választjuk meg, akkor egy túlhatározott lineáris egyenletrendszert kapunk, és ennek általában nincs (pontos) megoldása. Most megmutatjuk, hogy ebben az esetben hogyan állíthatjuk elő az egyenletrendszernek egy közelítő megoldását.

Vezessük be az

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \ldots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \ldots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \ldots & f_n(x_m) \end{pmatrix}$$

mátrixot, amely tehát az alapfüggvények adott pontokban vett értékeit tartalmazza, és legyen $a_{ij} = f_j(x_i)$. Jelölje továbbá $c = (c_k)$ a meghatározandó együtthatók n dimenziós vektorát. Ekkor az m dimenziós

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \ldots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \ldots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \ldots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

vektort az y-ok F által „jósolt" értékeinek is tekinthetjük. Az m dimenziós

$$\eta = Ac - y$$

vektort ***hibavektornak*** fogjuk nevezni.

A hibavektort többféle értelemben is minimalizálhatjuk. Itt csak azzal az esettel foglalkozunk, amikor a hibavektor

$$\|\eta\| = \Big(\sum_{i=1}^{m} \eta_i^2\Big)^{1/2}$$

normáját minimalizáljuk. Olyan c vektort kell tehát meghatároznunk, amelyre a hibavektor normája a lehető legkisebb. Ezt a feladatot a ***legkisebb négyzetes közelítés*** problémájának nevezzük.

Mivel

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \Big(\sum_{j=1}^{n} a_{ij}c_j - y_i\Big)^2,$$

ezért a c vektortól függő $\|\eta\|$ függvény minimumát úgy határozhatjuk meg, hogy $\|\eta\|^2$-et differenciáljuk c_k szerint, és az eredményt 0-val tesszük egyenlővé:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\Big(\sum_{j=1}^{n} a_{ij}c_j - y_i\Big)a_{ik} = 0. \tag{28.32}$$

A (28.32)-beli, $k = 1, 2, \ldots, n$-re felírt egyenlőségek ekvivalensek az

$$(Ac - y)^{\mathrm{T}}A = 0$$

mátrixegyenlettel. A 28.1-2. gyakorlat alapján tehát

$$A^{\mathrm{T}}(Ac - y) = 0,$$

amiből azt kapjuk, hogy

$$A^{\mathrm{T}}Ac = A^{\mathrm{T}}y. \tag{28.33}$$

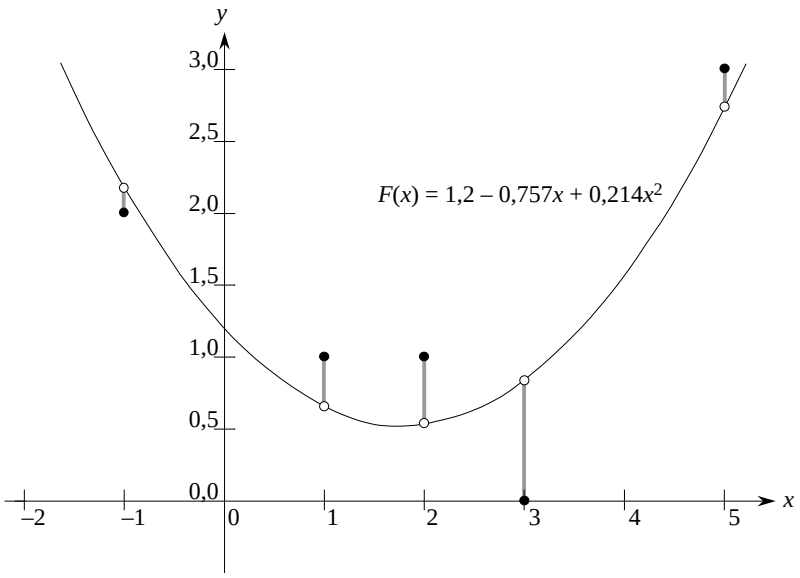
A statisztikában ezt ***normálegyenletnek*** nevezik. Az $A^{\mathrm{T}}A$ mátrix a 28.1-2. gyakorlat szerint szimmetrikus, és ha A teljes rangú, akkor $A^{\mathrm{T}}A$ egyszersmind pozitív definit. Ezért $(A^{\mathrm{T}}A)^{-1}$ létezik, és ekkor a (28.33) egyenlet egyetlen megoldása

$$\begin{aligned} c &= ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})y \\ &= A^{+}y. \end{aligned} \tag{28.34}$$

Az $A^{+} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}$ mátrixot az A mátrix ***általánosított inverzének*** nevezzük. Az általánosított inverz négyzetes mátrix inverzének a természetes általánosítása a nem négyzetes esetre. (A (28.34) képletben szereplő c vektort az $Ac = y$ túlhatározott lineáris egyenletrendszer ***legkisebb négyzetes megoldásának*** nevezzük. Vessük össze ezt a fogalmat a *pontos* megoldás fogalmával!)

A legkisebb négyzetes illesztés illusztrálására tegyük fel, hogy van 5 adatunk

$$\begin{aligned} (x_1, y_1) &= (-1, 2), \\ (x_2, y_2) &= (1, 1), \\ (x_3, y_3) &= (2, 1), \\ (x_4, y_4) &= (3, 0), \\ (x_5, y_5) &= (5, 3), \end{aligned}$$



28.3. ábra. Másodfokú polinom legkisebb négyzetes illesztése az öt pontot tartalmazó $\{(-1,2),(1,1),(2,1),(3,0),(5,3)\}$ adathalmazra. A fekete pontok az adatok, a fehérek pedig legkisebb négyzetes illesztéssel kapott $F(x) = 1,2 - 0,757x + 0,214x^2$ másodfokú polinom által „jósolt" közelítő értékek. Ez a polinom a hibák négyzetének az összegét minimalizálja. Az egyes pontokban a hibát a függőleges szakaszok jelzik.

amelyeket a 28.3. ábrán fekete pontokkal szemléltetünk. Olyan

$$F(x) = c_1 + c_2 x + c_3 x^2$$

másodfokú polinomot szeretnénk meghatározni, amelyik a legkisebb négyzetes értelemben jól illeszkedik az adatainkhoz.

Az alapfüggvények értékeiből képzett mátrix

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix}.$$

Ennek az általánosított inverze

$$A^+ = \begin{pmatrix} 0,500 & 0,300 & 0,200 & 0,100 & -0,100 \\ -0,388 & 0,093 & -0,190 & 0,193 & -0,088 \\ 0,060 & -0,036 & -0,048 & -0,036 & 0,060 \end{pmatrix}.$$

Az y vektort balról megszorozva az A^+ mátrixszal kapjuk a keresett

$$c = \begin{pmatrix} 1,200 \\ -0,757 \\ 0,214 \end{pmatrix}$$

együtthatóvektort, ebből pedig az

$$F(x) = 1,200 - 0,757x + 0,214x^2$$

másodfokú polinomot. Ez az adatainkra – a legkisebb négyzetes értelemben – legjobban illeszkedő másodfokú polinom.

A (28.33) normálegyenletet a gyakorlatban úgy oldjuk meg, hogy y-t beszorozzuk A^{T}-vel, majd megkeressük az $A^{\mathrm{T}}A$ mátrix egy LU felbontását. Ha A teljes rangú, akkor az $A^{\mathrm{T}}A$ mátrix biztosan nemszinguláris. A 28.1-2. gyakorlat és a 28.6. tétel alapján az $A^{\mathrm{T}}A$ mátrix szimmetrikus és pozitív definit, így a 28.9. lemma miatt valóban nemszinguláris. Ebben az esetben tehát a (28.33) egyenletnek pontosan egy megoldása van.

Gyakorlatok

28.5-1. Bizonyítsuk be, hogy szimmetrikus pozitív definit mátrix minden főátlóbeli eleme pozitív.

28.5-2. Legyen $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ egy 2×2-es szimmetrikus pozitív definit mátrix. A „teljes négyzetté alakítás” módszerével (lásd a 28.11. lemma bizonyítását) mutassuk meg, hogy A-nak a determinánsa – azaz $ac - b^2$ – pozitív.

28.5-3. Bizonyítsuk be, hogy szimmetrikus pozitív definit mátrix maximális eleme a főátlóban található.

28.5-4. Bizonyítsuk be, hogy szimmetrikus pozitív definit mátrix minden vezető részmátrixának a determinánsa pozitív.

28.5-5. Jelöljük A_k-val egy szimmetrikus pozitív definit A mátrix k-adik vezető részmátrixát. Bizonyítsuk be, hogy az LU felbontás kiszámítása során a k-adik főelem $\det(A_k)/\det(A_{k-1})$, ahol megállapodás szerint $\det(A_0) = 1$.

28.5-6. Keressük meg azt az

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

alakú függvényt, amelyik a legkisebb négyzetes értelemben legjobban illeszkedik az alábbi adatokhoz:

$$(1, 1), \quad (2, 1), \quad (3, 3), \quad (4, 8).$$

28.5-7. Mutassuk meg, hogy az A^+ általánosított inverz rendelkezik a következő tulajdonságokkal:

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^{\mathrm{T}} &= AA^+, \\ (A^+A)^{\mathrm{T}} &= A^+A. \end{aligned}$$

Feladatok

28-1. Tridiagonális lineáris egyenletrendszerek
Tekintsük az alábbi tridiagonális mátrixot:

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. Határozzuk meg A egy LU felbontását.

b. Oldjuk meg az $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$ egyenletet előre helyettesítéssel is és visszahelyettesítéssel is.

c. Határozzuk meg A inverzét.

d. Bizonyítsuk be, hogy minden $n \times n$-es szimmetrikus pozitív definit tridiagonális A mátrix és n dimenziós b vektor esetén az $Ax = b$ egyenlet LU felbontással $O(n)$ futási idő alatt oldható meg. Mutassuk meg azt is, hogy az A^{-1} kiszámításán alapuló minden megoldási módszer – a legrosszabb esetben – aszimptotikusan időigényesebb.

e. Mutassuk meg, hogy minden $n \times n$-es nemszinguláris tridiagonális A mátrix és n dimenziós b vektor esetén az $Ax = b$ egyenlet LUP felbontással $O(n)$ futási idő alatt oldható meg.

28-2. Spline-függvények
A gyakorlatban adott pontokat interpoláló görbe meghatározásához gyakran ***harmadfokú spline-okat*** használnak. Legyen adva az $n + 1$ számpárból álló $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ halmaz, ahol $x_0 < x_1 < \cdots < x_n$. E pontokra egy szakaszonként harmadfokú polinomokból álló $f(x)$ görbét (spline-t) fogunk illeszteni. Azaz, az $f(x)$ görbe n harmadfokú $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ polinom együttese, ahol $x_i \le x \le x_{i+1}$ esetén a függvény értéke $f(x) = f_i(x - x_i)$. Az x_i pontokat, amelyekben a harmadfokú polinomokat „összeragasztjuk", ***csomópontoknak*** nevezzük. Az egyszerűség kedvéért feltesszük, hogy $x_i = i$ minden $i = 0, 1, \ldots, n$ esetén.

Az $f(x)$ függvény folytonosságát biztosítandó megköveteljük, hogy minden $i = 0, 1, \ldots, n - 1$ indexre

$$\begin{aligned} f(x_i) &= f_i(0) = y_i\,, \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

teljesüljön. Minden egyes csomópontban az első derivált folytonosságát is feltételezzük annak érdekében, hogy az $f(x)$ függvény elég sima legyen. Ez azt jelenti, hogy az

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

egyenlőségeket is megköveteljük minden $i = 0, 1, \ldots, n - 1$ esetén.

a. Tegyük fel, hogy minden $i = 0, 1, \ldots, n$ indexre az $\{(x_i, y_i)\}$ számpárok mellett az első derivált csomópontokban felvett $D_i = f'(x_i)$ értékeit is előírjuk. Fejezzük ki az a_i, b_i, c_i és d_i együtthatókat az y_i, y_{i+1}, D_i és D_{i+1} mennyiségekkel. (Ne feledjük, hogy $x_i = i$.) Milyen gyorsan számítható ki a $4n$ együttható a számpárokból és az első deriváltakból?

Most már csak az a kérdés, hogy a csomópontokban hogyan válasszuk meg $f(x)$ első deriváltjait. Erre több lehetőség is van. Az egyik az, hogy a második deriváltak folytonosságát is megköveteljük ezekben a pontokban, azaz feltesszük még azt is, hogy

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

minden $i = 0, 1, \ldots, n-1$ esetén, az első csomópontban $f''(x_0) = f_0''(0) = 0$, az utolsóban pedig $f''(x_n) = f_n''(1) = 0$. Ezeket a feltételeket kielégítő $f(x)$-eket ***természetes spline-oknak*** nevezzük.

b. A második deriváltakra vonatkozó folytonossági feltételeket felhasználva mutassuk meg, hogy minden $i = 0, 1, \ldots, n-1$ esetén

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}). \tag{28.35}$$

c. Bizonyítsuk be, hogy

$$2D_0 + D_1 = 3(y_1 - y_0), \tag{28.36}$$

$$D_{n-1} - 2D_n = 3(y_n - y_{n-1}). \tag{28.37}$$

d. Írjuk át a (28.35)–(28.37) egyenleteket olyan mátrixegyenletté, amelyben az ismeretlen a $D = (D_0, D_1, \ldots, D_n)^{\mathrm{T}}$ vektor. Milyen tulajdonságú lesz az így adódó mátrix?

e. Mutassuk meg, hogy egy $n+1$ pontpárból álló rendszert természetes harmadfokú spline segítségével $O(n)$ idő alatt interpolálhatunk (lásd a 28-1. feladat).

f. Hogyan határozható meg az a természetes harmadfokú spline, amely az $x_0 < x_1 < \cdots < x_n$ feltételeket kielégítő $n+1$ darab (x_i, y_i) pontot interpolálja, de x_i nem egyenlő i-vel. Milyen mátrixegyenletet kell megoldani, és milyen gyorsan fut az algoritmus?

Megjegyzések a fejezethez

Sok olyan kitűnő könyvet írtak, amelyek az itteninél jóval részletesebben tárgyalják a numerikus és tudományos számításokat. Az alábbiak különösen ajánlhatók: George és Liu [113], Golub és Van Loan [125], Press, Flannery, Teukolsky és Vetterling [248, 249], valamint Strang [285, 286].

Golub és Van Loan [125] könyve a numerikus stabilitás problémájával foglalkozik. Megmutatják, hogy a $\det(A)$ miért nem alkalmas az A mátrix stabilitásának a jellemzésére, amelyre ők az $\|A\|_\infty \|A^{-1}\|_\infty$ szám használatát javasolják, ahol $\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. Felvetik azt a kérdést is, hogy ezt a számot hogyan lehet meghatározni A^{-1} kiszámítása nélkül.

Nagy feltűnést keltett Strassen [287] 1969-ben publikált algoritmusa. Előtte szinte elképzelhetetlennek tűnt, hogy a hagyományos mátrixszorzási algoritmus javítható. Azóta a mátrixszorzás futási idejének az aszimptotikus felső korlátját jelentősen sikerült megjavítani. A jelenleg ismert aszimptotikusan leghatékonyabb mátrixszorzási algoritmus, amely Coppersmith és Winograd [70] nevéhez fűződik, $n \times n$-es mátrixokra $O(n^{2,376})$ futási időt garantál. Strassen algoritmusának a grafikus megjelenítésével Paterson [238] foglalkozott.

A Gauss-féle kiküszöbölési eljárás – amelyen az LU és az LUP felbontás alapul – volt az első általános módszer lineáris egyenletrendszer megoldására. Ez volt egyszersmind a legkorábbi numerikus algoritmus. Bár már korábban is ismerték, felfedezését mégis C. F. Gaussnak (1777–1855) tulajdonítják. Strassen híres [287] cikkében azt is megmutatta, hogy egy $n \times n$-es mátrix $O(n^{\lg 7})$ futási idő alatt invertálható. Először Winograd [317] igazolta, hogy a mátrixszorzás nem nehezebb, mint a mátrixinvertálás. A fordított állítás bizonyítása Aho, Hopcroft és Ullman [5] érdeme.

Mátrixoknak egy másik fontos felbontása a ***szinguláris értékek szerinti felbontás.*** Itt egy $m \times n$ méretű A mátrix $A = Q_1 \sum Q_2^{\mathrm{T}}$ alakban való felírásáról van szó, ahol $\sum$ egy olyan $m \times n$-es mátrix, amelynek csak a főátlójában vannak nemnulla elemek. Q_1, illetve Q_2 pedig olyan $m \times m$-es, illetve $n \times n$-es mátrixok, amelyeknek az oszlopai páronként ortonormáltak. Két vektort akkor nevezünk ***ortonormáltnak,*** ha a belső szorzatuk 0 és mindegyik vektor normája 1. Strang [285, 286], valamint Golub és Van Loan [125] könyvei a szinguláris értékek szerinti felbontásnak egy jó kifejtését tartalmazzák.

Strang [286] műve szimmetrikus pozitív definit mátrixok és egyáltalán a lineáris algebra kiváló tárgyalását adja.

29. Lineáris programozás

Számos olyan probléma ismert, melynek megoldásához valamely célfüggvénynek a maximumát vagy minimumát kell megkeresnünk, különféle korlátozó feltételek mellett. A feltételek jelenthetnek például erőforrások szűkössége okozta korlátokat, vagy leírhatnak egymással versenyző elvárásokat. Ha a célfüggvény a döntési változók lineáris függvénye, és a korlátozó feltételek kifejezhetők a változók lineáris kifejezései közti egyenlőségekkel és egyenlőtlenségekkel, akkor ez ***lineáris programozási feladat.*** Sokféle gyakorlati alkalmazásban merül fel lineáris programozási feladat. Elsőként nézzünk egy alkalmazást a politikai választások területén.

Egy politikus problémája

Képzeljük magunkat egy politikus helyébe, aki győzni szeretne a választáson. A választókerület három régióból áll – városi, külvárosi és vidéki. Ezen régióknak rendre 100 000, 200 000 és 50 000 regisztrált szavazója van. A hatékony kormányzáshoz mindhárom régióban szeretnénk a szavazatok többségét megnyerni. Mint tiszteletre méltó politikus, soha nem támogatnánk olyan politikát, amellyel nem értünk egyet. Tény azonban, hogy bizonyos intézkedések hatékonyabbak az egyik vagy másik régió szavazóinak megnyerésében. Legfőbb törekvéseink: minél több autóút építése, fegyverkorlátozás, a földművelés támogatása és üzemanyagadó bevezetése, melyet a városi tömegközlekedés fejlesztésére szándékozunk fordítani. A kampányszervező csapat kutatásai szerint megbecsülhető, hogy a fenti célok reklámozására fordított minden újabb ezer dollár hány további szavazó megnyerését vagy elvesztését eredményezi az egyes régiókban. Az információkat a 29.1. ábra táblázatában foglaltuk össze. A tábla adatai mutatják, hogy az adott intézkedések reklámozására fordított ezer dollár befektetéssel hány ezer városi, külvárosi vagy vidéki szavazót nyerhetünk meg. A negatív értékek az elvesztett szavazatokat jelentik. A feladatunk az, hogy meghatározzuk azt a minimális költséget, amivel 50 000 városi, 100 000 külvárosi és 25 000 vidéki szavazat nyerhető.

Próbálkozással találhatunk olyan stratégiát, mellyel a szükséges szavazatok megnyerhetők, de elképzelhető, hogy nem a legolcsóbbat találjuk meg. Vegyük a következő példát: ha 20 000 dollárt költenénk az autóutak építésének reklámjára, 0 dollárt a fegyverkorlátozás, 4000 dollárt a földművelés támogatásának, és 9000 dollárt az üzemanyag adóztatásának reklámjára, akkor $20(-2) + 0(8) + 4(0) + 9(10) = 50$ ezer városi, $20(5) + 0(2) + 4(0) + 9(0) = 100$ ezer külvárosi és $20(3) + 0(-5) + 4(10) + 9(-2) = 82$ ezer vidéki szavazatot nyernénk.

távlati terv	városi	külvárosi	vidéki
autóút építése	–2	5	3
fegyverkorlátozás	8	2	–5
földművelés támogatása	0	0	10
üzemanyag megadóztatása	10	0	–2

29.1. ábra. A választási ígéretek hatása a szavazókra. Az értékek ezer főben adják meg azon városi, külvárosi vagy vidéki szavazók számát, akik megnyerhetők, ha ezer dollárt fektetünk be az adott témával kapcsolatos távlati terveink reklámozásába. A negatív értékek adott számú szavazat elvesztését jelentik.

A megnyert városi és külvárosi szavazatok száma egyenlő a tervezettel, a vidéki több mint elég. (Vegyük észre, hogy a modell szerint megszerzett vidéki szavazatok száma több, mint ahány vidéki szavazó van!) Ezen szavazatok megszerzéséhez $20+0+4+9=33$ ezer dollárt kellene költeni reklámozásra.

Természetesen kíváncsiak vagyunk, hogy ez a stratégia a lehető legjobb-e, vagy céljaink kisebb költséggel is elérhetők. További próbálkozások segíthetnek ennek eldöntésében, de jobb lenne egy megbízható módszer az ilyen kérdések megválaszolására. Ennek érdekében fogalmazzuk meg a kérdést a matematika nyelvén. Bevezetünk négy változót.

- x_1 az autóútépítés reklámjára fordított összeg ezer dollárban,
- x_2 a fegyverkorlátozás reklámjára fordított összeg ezer dollárban,
- x_3 a földművelés támogatásának reklámjára fordított összeg ezer dollárban, és
- x_4 az üzemanyag megadóztatásának reklámjára fordított összeg ezer dollárban.

A következőképpen írhatjuk le, hogy 50 000 városi szavazatot szeretnénk nyerni:

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50. \tag{29.1}$$

Hasonlóan felírhatjuk azon elvárásainkat, hogy 100 000 külvárosi és 25 000 vidéki szavazatot kívánunk nyerni:

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

és

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25. \tag{29.3}$$

Az x_1, x_2, x_3, x_4 változók értékeinek minden olyan beállítása, mely kielégíti a (29.1)–(29.3) egyenlőtlenségeket, megfelelő stratégiát ad a kívánt szavazatok megnyeréséhez. A legkisebb költségű stratégia megtalálásához a reklámra fordított összeget kell minimalizálni, azaz minimalizálni szeretnénk az

$$x_1 + x_2 + x_3 + x_4 \tag{29.4}$$

kifejezést. Bár a politikai kampányban ismert a negatív reklám fogalma, de olyan nem létezik, hogy a reklámra fordított költség negatív legyen, tehát feltesszük, hogy

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0 \text{ és } x_4 \geq 0. \tag{29.5}$$

Tehát a (29.4) célfüggvényt akarjuk minimalizálni a (29.1)–(29.3) és (29.5) feltételek mellett. Összefoglalva ez így írható:

$$\text{minimalizálandó} \quad x_1 + x_2 + x_3 + x_4 \tag{29.6}$$

feltéve, hogy

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \tag{29.7}$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.8}$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \tag{29.9}$$

$$x_1, x_2, x_3, x_4 \geq 0\,. \tag{29.10}$$

Az ilyen feladatot „lineáris programozási feladatnak” nevezzük. A fenti lineáris programozási feladat megoldása szolgáltatja a politikus legkedvezőbb stratégiáját.

Az általános lineáris programozási feladat

Az általános lineáris programozási feladatban lineáris függvényt kell optimalizálni lineáris egyenlőtlenségek mellett. Az $x_1, x_2, \ldots, x_n$ változóknak valamely f függvénye ***lineáris függvény,*** ha az alábbi formájú:

$$\begin{aligned} f(x_1, x_2, \ldots, x_n) &= a_1x_1 + a_2x_2 + \cdots + a_nx_n \\ &= \sum_{j=1}^{n} a_jx_j, \end{aligned}$$

ahol $a_1, a_2, \ldots, a_n$ valós számok. Legyen b valós szám és f lineáris függvény, akkor az

$$f(x_1, x_2, \ldots, x_n) = b$$

egyenlőség ***lineáris egyenlőség*** és az

$$f(x_1, x_2, \ldots, x_n) \leq b$$

és

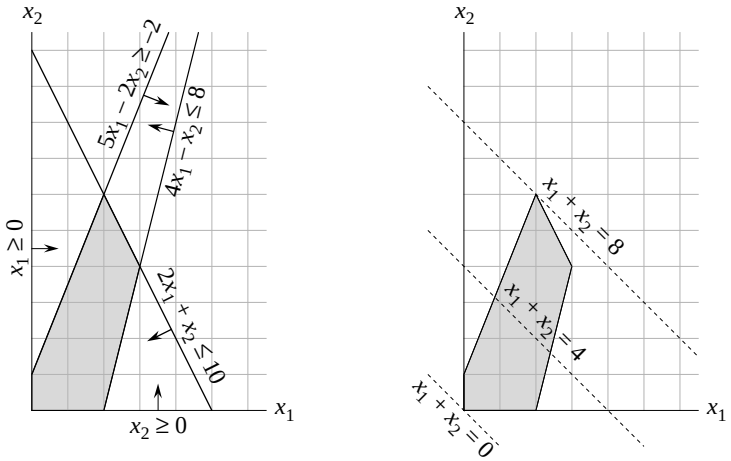
$$f(x_1, x_2, \ldots, x_n) \geq b$$

egyenlőtlenségek ***lineáris egyenlőtlenségek.*** A lineáris egyenlőség és a lineáris egyenlőtlenség fogalmakra együttesen a ***lineáris korlátozó feltétel*** kifejezést fogjuk használni. Szigorú egyenlőtlenséget nem engedünk meg a feltételek között. A ***lineáris programozási feladat*** formálisan egy lineáris függvény ***maximalizálása*** vagy ***minimalizálása*** véges számú lineáris korlátozó feltétel mellett.

A fejezet további részében a lineáris programozási feladatok leírásával és megoldásával foglalkozunk. A legrégibb lineáris programozási algoritmust, a szimplex módszert fogjuk tárgyalni. Bár az újabb módszerek nagyméretű általános feladatokra hatékonyabbak (és a szimplex módszerrel ellentétben polinom idejűek is), a szimplex módszer általában elég hatékony és a gyakorlatban széles körben használják.

A lineáris programozási feladat áttekintése

A feladat tulajdonságainak és az algoritmusoknak a tárgyalásához célszerű kanonikus alakokat bevezetni az ábrázoláshoz. Két alakot fogunk használni a fejezetben, a ***szabályos*** (standard) és a ***kiegyenlített*** (slack) alakot. Ezek pontos definícióját a 29.1. alfejezetben találjuk



29.2. ábra. (a) A (29.12)–(29.15) egyenlőtlenségek által meghatározott megengedett tartomány. Egy-egy korlátozó feltételnek megfelelő pontok egy-egy félsíkba esnek, ezen félsíkokat az ábra egy egyenessel és egy iránnyal jelzi. A félsíkok közös része, ami a megengedett tartomány, szürke színű. **(b)** A szaggatott vonalú egyenesek azokat a pontokat ábrázolják, melyre a célfüggvényérték 0, 4, illetve 8. A feladat optimális megoldása $x_1 = 4$, $x_2 = 6$, ahol a célfüggvényérték 8.

meg. Kissé pontatlanul azt mondhatjuk, hogy a szabályos alakú lineáris programozási feladatban a célfüggvényt lineáris *egyenlőtlenségek* mellett maximalizáljuk, míg a kiegyenlített alakban ugyanezt lineáris *egyenlőségek* mellett tesszük. A lineáris programozási feladatok megfogalmazásához leggyakrabban a szabályos alakot használják, azonban a szimplex algoritmus részleteinek elemzéséhez kényelmesebb lesz a kiegyenlített alakot használni. A továbbiakban n változójú lineáris célfüggvény maximalizálását fogjuk tárgyalni m lineáris egyenlőtlenség mellett.

Először tekintsük a következő kétváltozós lineáris programozási feladatot:

$$\text{maximalizálandó} \quad x_1 + x_2 \tag{29.11}$$

feltéve, hogy

$$4x_1 - x_2 \leq 8 \tag{29.12}$$

$$2x_1 + x_2 \leq 10 \tag{29.13}$$

$$5x_1 - 2x_2 \geq -2 \tag{29.14}$$

$$x_1, x_2 \geq 0. \tag{29.15}$$

A fenti lineáris programozási feladat ***megengedett megoldásának*** nevezzük az (x_1, x_2) változók minden olyan értékpárját, amely kielégíti a (29.12)–(29.15) feltételeket. Ha a korlátozó feltételeket egy (x_1, x_2) koordináta-rendszerben ábrázoljuk, akkor a megengedett megoldások konvex tartományt alkotnak[1] a síkon. A megengedett tartományt a 29.2(a) ábra szürkével jelzi. Ezt a konvex tartományt ***megengedett tartománynak,*** a maximalizálandó függvényt pedig ***célfüggvénynek*** nevezzük. Elvileg kiértékelhetnénk az $x_1 + x_2$ célfüggvényt

[1] A konvex tartomány egy intuitív definíciója, hogy bármely két pontjával együtt azőket összekötő szakaszt is tartalmazza.

a megengedett tartomány minden egyes pontjában. Ezután meghatározhatnánk egy olyan pontot, ahol a ***célfüggvényérték*** maximális, így kapnánk egy optimális megoldást. Példánkban (és a legtöbb lineáris programozási feladatban) a megengedett tartomány végtelen sok pontot tartalmaz, így hatékonyabb módszert kell találnunk annak a pontnak a megtalálására, ahol a célfüggvényérték maximális, mintsem hogy a megengedett tartomány minden pontjában kiértékeljük a célfüggvényt.

Két dimenzióban a megoldást grafikus módszerrel megkereshetjük. Adott z valós szám esetén azon pontok, melyekre $x_1 + x_2 = z$ fennáll, egy -1 meredekségű egyenesen helyezkednek el. Ha ábrázoljuk az $x_1 + x_2 = 0$ feltételt kielégítő pontokat, egy az origón átmenő -1 meredekségű egyenest kapunk, ahogy az a 29.2(b) ábrán látható. Ennek az egyenesnek és a megengedett tartománynak a közös része azon megengedett megoldásokból áll, melyekre a célfüggvényérték 0. Esetünkben az egyenes és a megengedett tartomány közös része a $(0, 0)$ pont. Általánosan szólva, tetszőleges z értékre, az $x_1 + x_2 = z$ egyenes és a megengedett tartomány közös része a megengedett megoldások azon halmaza, melyre a célfüggvényérték egyenlő z-vel. A 29.2(b) ábrán az $x_1 + x_2 = 0$, az $x_1 + x_2 = 4$ és az $x_1 + x_2 = 8$ egyeneseket láthatjuk. Mivel a 29.2. ábrán a megengedett tartomány korlátos, létezik azon z értékeknek maximuma, melyekre az $x_1 + x_2 = z$ egyenes és a megengedett tartomány közös része nem üres. Bármely pont, amelyhez a maximális z érték tartozik, a lineáris programozási feladatnak optimális megoldása. Ez ebben az esetben az $x_1 = 2$ és $x_2 = 6$ pont, ahol a célfüggvényérték 8. Nem véletlen, hogy az optimális megoldás a megengedett tartománynak csúcspontja. Mivel példánkban a célfüggvény szintvonalai egyenesek, az optimális célfüggvényértékhez tartozó szintvonalnak a megengedett tartomány határán kell haladnia, így a közös részük vagy egy pont, vagy egy szakasz lehet. Ha a közös rész egyetlen pont, akkor csak egy optimális megoldás létezik és az épp a szóban forgó pont. Ha a közös rész szakasz, akkor a szakasz minden pontjában azonos a célfüggvényérték; tehát a szakasz mindkét végpontja is optimális megoldás. Minthogy a szakasz mindkét végpontja csúcspont is egyben, ebben az esetben is van csúcspontbeli optimális megoldás.

Bár több mint két változó esetén nem lenne könnyű a lineáris programozási feladat ábrázolása, ugyanezen elvek érvényesek rá. Ha három változónk van, akkor egy-egy korlátozó feltételnek megfelelő pontok egy-egy féltérbe esnek. Ezen félterek közös része alkotja a megengedett tartományt. Nemkonstans célfüggvény esetén síkot alkotnak azon pontok, amelyekre a célfüggvény értéke adott z valós számmal egyenlő. Ha a célfüggvény összes együtthatója nemnegatív és ha az origó megengedett megoldása a lineáris programozási feladatnak, akkor ezt a síkot az origótól távolítva egyre nagyobb célfüggvényértékű megengedett pontokat találhatunk. (Ha az origó nem megengedett, vagy ha a célfüggvény együtthatói közt van negatív, akkor a szemléltető modell kissé bonyolultabbá válik.) A konvexitás miatt az optimális megoldások halmaza mindenképp tartalmazza a megengedett tartomány egy csúcspontját. Hasonlóan, ha n változónk van, mindegyik korlát egy félteret határoz meg az n-dimenziós térben. A félterek közös részét ***poliédernek*** nevezzük. Nemkonstans célfüggvény esetén hipersíkot alkotnak azon pontok, amelyekre a célfüggvény értéke adott z valós számmal egyenlő. A konvexitás miatt az optimális megoldások halmaza tartalmazza a megengedett poliéder egy csúcspontját.

A ***szimplex módszer*** bemenete egy lineáris programozási feladat, kimenete egy optimális megoldás. Az algoritmus a megengedett tartomány egyik csúcspontjáról indul és iterációk sorozatát hajtja végre. Minden egyes iterációban a megengedett tartomány egy éle mentén halad az aktuális csúcsból egy olyan szomszédos csúcsba, ahol a célfüggvényérték

nem kisebb (általában nagyobb), mint az aktuális csúcsban. A szimplex módszer akkor fejeződik be, amikor elér egy lokális maximumot, azaz egy olyan csúcsot, amelyben a célfüggvényérték nem kisebb, mint bármely szomszédjához tartozó célfüggvényérték. Minthogy a megengedett tartomány konvex és a célfüggvény lineáris, a lokális optimum egyben globális optimum is. A 29.4. alfejezetben használni fogjuk a „dualitás" elvét, hogy bebizonyíthassuk, a szimplex módszer által megtalált megoldás valóban optimális.

Bár a geometriai megközelítés jó képet ad a szimplex módszer működéséről, a 29.3. alfejezetben a szimplex módszer részletes elemzésénél nem ezt fogjuk használni, helyette az algebrai megközelítést követjük. Először is egyenlőségekké alakítjuk a lineáris korlátozó feltételeket, a változók nemnegativitását megkövetelve. Az így kapott lineáris programozási feladat változóit két halmazra osztjuk, a „bázisváltozók" és a „nembázis-változók" halmazára. (A bázisváltozók egyértelműen kifejezhetők lesznek az egyenlőségeinkből a nembázis-változók segítségével.) A megengedett tartomány egyik csúcsából a másik csúcsába való lépést egy bázisváltozónak nembázis-változóvá, és egy nembázis-változónak bázisváltozóvá való átminősítésével valósítjuk meg. Ezt a műveletet „pivotálásnak" nevezzük, algebrailag tekintve nem jelent mást, mint hogy a lineáris programozási feladat egyik alakjáról egy másik ekvivalens alakjára térünk át.

A fent bemutatott kétváltozós példa valójában egy igen egyszerű eset volt. Még számos részletet tisztázni kell a fejezet további részében. Ezek közé tartozik az olyan lineáris programozási feladatok vizsgálata, melyeknek nincs megengedett megoldása, vagy nincs optimális megoldása, vagy azoké, melyekre az origó nem megengedett megoldás.

A lineáris programozás alkalmazásai

Sok területen alkalmazható a lineáris programozás. Az operációkutatással foglalkozó kézikönyvek tele vannak példákkal, manapság már a legtöbb üzleti iskolában is megszokott eszközként tanítják a lineáris programozást. A választási forgatókönyv az egyik tipikus példa. Nézzünk két továbbit:

- Egy légitársaság el akarja készíteni a repülőgépek személyzetének beosztását. A Federal Aviation Administration (az Egyesült Államok repülést szabályozó hatósága) számos követelményt előír, például, hogy a személyzet tagjai hány órát dolgozhatnak egyfolytában, vagy hogy egy adott személy egy hónapon belül csak azonos típusú repülőgéppel repülhet. A légitársaság úgy kívánja beosztani a személyzetet az összes járatára, hogy a lehető legkevesebb főre legyen szükség.
- Egy olajtársaság szeretné eldönteni, hogy hol fúrjanak olajkutakat. Az egyes területeken ismert a fúrás költsége, és a geológiai vizsgálatok alapján ismert az egyes területeken fúrható kutak várható hozama. A társaságnak adott összeg áll rendelkezésére, hogy új kutakat fúrjon, és szeretné maximalizálni a várható olajhozamot a költségvetés lehetőségein belül.

A lineáris programozás felhasználható gráfelméleti és kombinatorikai problémák megoldásához is, ahogy ebben a könyvben is láthatjuk majd. A 24.4. alfejezetben találkozhattunk egy olyan speciális lineáris programozási feladattal, amely különbségi korlátrendszerek megoldására használható. A 29.2. alfejezetben látni fogjuk, hogyan fogalmazhatók meg különféle gráfelméleti és hálózati folyam problémák lineáris programozási feladatként.

A 35.4. alfejezetben eszközként használjuk a lineáris programozást, hogy egy gráfelméleti probléma közelítő megoldását megkapjuk.

Lineáris programozási feladatok megoldó algoritmusai

Ez a fejezet a szimplex módszert tárgyalja. Ez az algoritmus, ha körültekintően valósítjuk meg, a gyakorlatban legtöbbször gyorsan megoldja az általános lineáris programozási feladatokat. Néhány körmönfont feladatra azonban exponenciális ideig futhat. Az első polinom idejű lineáris programozási algoritmus az ***ellipszoid módszer*** volt, ami a gyakorlatban lassan fut. A polinom idejű algoritmusok másik osztályát képezik az úgynevezett ***belsőpontos módszerek.*** Ellentétben a szimplex módszerrel, amely a megengedett tartomány külső szélein halad, és az aktuális megoldás mindig a megengedett tartomány egy csúcspontja, ezek az algoritmusok a megengedett tartomány belsejében haladnak. A közbenső megoldások, bár megengedettek, nem feltétlenül csúcsai a megengedett tartománynak, de a végső megoldás csúcspont lesz. Az első ilyen algoritmust Karmarkar fejlesztette ki. Nagyszámú bemenő adatra a belsőpontos módszerek hasonlóan gyorsak, néha gyorsabbak, mint a szimplex módszer.

Ha azt a további kikötést tesszük, hogy a változók csak egész értékeket vehetnek fel, akkor ***egész értékű lineáris programozási feladatról*** beszélünk. Mint a 34.5-3. gyakorlatban szerepel, ilyen feladathoz már megengedett megoldás keresése is NP-nehéz probléma. Minthogy az NP-nehéz problémáknak nincs polinom idejű megoldása, így nincs ismert polinom idejű algoritmus az egész értékű lineáris programozási feladatra. Ezzel szemben az általános lineáris programozási feladat megoldható polinom időben.

A fejezetben, az $x = (x_1, x_2, \ldots, x_n)$ vektor a lineáris programozási feladat változóit jelöli, míg $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ a változók konkrét értékeit jelenti.

29.1. A szabályos és kiegyenlített alak

Ez az alfejezet a lineáris programozási feladat két olyan alakját mutatja be, melyek hasznosak lesznek a feladat megfogalmazásában és elemzésében. A szabályos alakban a korlátozó feltételek egyenlőtlenségek, míg a kiegyenlített alakban a korlátozó feltételek egyenlőségek.

A szabályos alak

A ***szabályos alak*** (standard alak) definíciója a következő: adott n és m darab valós szám, jelölje őket $c_1, c_2, \ldots, c_n$ és $b_1, b_2, \ldots, b_m$, továbbá adott mn darab valós szám, amiket a_{ij} $i = 1, 2, \ldots, m$ és $j = 1, 2, \ldots, n$ jelöl. Keressük azt az n darab $x_1, x_2, \ldots, x_n$ értékeket, melyre

maximalizálandó
$$\sum_{j=1}^{n} c_j x_j \tag{29.16}$$

feltéve, hogy

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, 2, \ldots, m), \tag{29.17}$$

$$x_j \geq 0 \quad (j = 1, 2, \ldots, n). \tag{29.18}$$

Általánosítva a kétdimenziós lineáris programozási feladatnál bevezetett fogalmakat a (29.16) kifejezést ***célfüggvénynek*** nevezzük, a (29.17) és (29.18) $m + n$ darab egyenlőtlenséget pedig ***korlátozó feltételeknek.*** A (29.18)-ban megadott feltételt ***nemnegativitási feltételnek*** hívjuk. Egy tetszőleges lineáris programozási feladat nem feltétlenül követeli meg, hogy az összes változó nemnegatív legyen, de a szabályos alakban ezt elvárjuk. Néhány esetben kényelmesebb még tömörebb formában ábrázolni a lineáris programozási feladatot. Legyen az $A = (a_{ij})$ egy $(m \times n)$-es mátrix, a $b = (b_i)$ egy m-dimenziós vektor és a $c = (c_j)$, valamint az $x = (x_j)$ n-dimenziós vektorok, ekkor (29.16)–(29.18) a lineáris programozási feladatot a következő alakban írhatjuk fel:

$$\text{maximalizálandó} \quad c^{\mathrm{T}}x \tag{29.19}$$

feltéve, hogy

$$Ax \leq b \tag{29.20}$$

$$x \geq 0. \tag{29.21}$$

A (29.19) képletben a $c^{\mathrm{T}}x$ kifejezés két vektor skaláris szorzata. A (29.20)-ban Ax kifejezés egy mátrix-vektor szorzat és a (29.21) $x \geq 0$ feltétel azt jelenti, hogy az x vektor minden elemére előírjuk a nemnegativitást. Vegyük észre, hogy a szabályos alakú lineáris programozási feladatot megadhatjuk az (A, b, c) hármassal, ahol az A, b és c megfelelő dimenziójúak.

Most tekintsük át a lineáris programozási feladat megoldásával kapcsolatos fogalmakat. Ezek közül néhányat már használtunk a korábban ismertetett kétdimenziós lineáris programozási feladatban. Az olyan $\bar{x}$ vektort, mely kielégíti valamennyi korlátozó feltételt, ***megengedett megoldásnak*** nevezzük. Ha azonban akár csak egy korlátozó feltétel is sérül, az $\bar{x}$ vektor ***nem megengedett.*** Az $\bar{x}$ vektorhoz a $c^{\mathrm{T}}\bar{x}$ ***célfüggvényérték*** tartozik. Az az $\bar{x}$ megengedett megoldás, melynek célfüggvényértéke maximális az összes megengedett megoldás célfüggvényértéke közül, a lineáris programozási feladat ***optimális megoldása.*** Ennek a $c^{\mathrm{T}}\bar{x}$ célfüggvényértékét ***optimális célfüggvényértéknek*** vagy ***optimumnak*** nevezzük. Előfordulhat, hogy a lineáris programozási feladatnak nincs megengedett megoldása. Ha a lineáris programozási feladatnak van megengedett megoldása, de a célfüggvényérték minden határon túl javítható, akkor ***nemkorlátos*** a célfüggvény. A 29.1-9. gyakorlatban adottak szerint bizonyítható, hogy a lineáris programozási feladatnak akkor is létezhet optimális megoldása, ha a megengedett tartomány nem korlátos.

A lineáris programozási feladatok szabályos alakjának felírása

Tetszőleges lineáris programozási feladat szabályos alakra hozható. Négy oka lehet annak, hogy a lineáris programozási feladat nem szabályos alakú:

1. A célfüggvényt nem maximalizálni, hanem minimalizálni kell.
2. Lehetnek olyan változók, melyekre nincs előírva a nemnegativitási feltétel.
3. Lehetnek ***lineáris egyenlőségként megadott korlátozó feltételek,*** melyekben a kisebb vagy egyenlő reláció helyett az egyenlő reláció szerepel.
4. Lehetnek olyan ***lineáris egyenlőtlenségként megadott korlátozó feltételek,*** melyekben a nagyobb vagy egyenlő relációjel szerepel a kisebb vagy egyenlő helyett.

Ha egy adott L lineáris programozási feladatot egy másik L' lineáris programozási feladattá alakítunk át, elvárjuk azt, hogy az L' optimális megoldása elvezessen az L optimális megoldásához. Ezt a következő módon fogalmazhatjuk meg: az L és L' maximumot kereső lineáris programozási feladatok ***ekvivalensek,*** ha az L minden olyan $\bar{x}$ megengedett megoldásához, melynek célfüggvényértéke z, tartozik L'-nek z célfüggvényértékű $\bar{x}'$ megengedett megoldása; és fordítva: az L' minden olyan $\bar{x}'$ megengedett megoldásához, melynek célfüggvényértéke z, tartozik L-nek z célfüggvényértékű $\bar{x}$ megengedett megoldása. (A definíció nem követeli meg az egy-egy értelmű megfeleltetést a megengedett megoldások között.) A minimumot kereső L és a maximumot kereső L' lineáris programozási feladatok ekvivalensek, ha az L minden olyan $\bar{x}$ megengedett megoldásához, melynek célfüggvényértéke z, tartozik L'-nek olyan $\bar{x}'$ megengedett megoldása, melynek $(-z)$ a célfüggvényértéke; és fordítva: az L' minden olyan $\bar{x}'$ megengedett megoldásához, melynek célfüggvényértéke z, tartozik L-nek olyan $\bar{x}$ megengedett megoldása, melynek $(-z)$ a célfüggvényértéke.

A következőkben pontról pontra bemutatjuk, hogyan oldhatók meg a felsorolt problémák, majd bebizonyítjuk, hogy az új lineáris programozási feladat ekvivalens a régivel.

A minimumot kereső L lineáris programozási feladat a célfüggvény együtthatóinak egyszerű negálásával könnyen átalakítható vele ekvivalens L' maximumot kereső lineáris programozási feladattá. Mivel az L és L' megengedett megoldásai azonosak, és bármely megengedett megoldásra az L-beli célfüggvényérték (-1)-szerese az L'-beli célfüggvényértéknek, a két lineáris programozási feladat ekvivalens egymással. Ha például a

$$\begin{array}{lrcrcl} \text{minimalizálandó} & -2x_1 & + & 3x_2 & & \\ \text{feltéve, hogy} & & & & & \\ & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & x_1 & & & \geq & 0 \end{array}$$

lineáris programozási feladatban negáljuk a célfüggvény együtthatóit, akkor a következőt kapjuk:

$$\begin{array}{lrcrcl} \text{maximalizálandó} & 2x_1 & - & 3x_2 & & \\ \text{feltéve, hogy} & & & & & \\ & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & x_1 & & & \geq & 0. \end{array}$$

Most megmutatjuk, hogyan alakítható át az a lineáris programozási feladat, melynek néhány változójára nem érvényes a nemnegativitási feltétel olyan feladattá, melynek minden változójára vonatkozik a nemnegativitási feltétel. Legyen x_j egy olyan változó, amelyre nincs előírva a nemnegativitási feltétel. Helyettesítsük x_j minden előfordulását az $x'_j - x''_j$ kifejezéssel, és vegyük hozzá a feladathoz az $x'_j \geq 0$ és $x''_j \geq 0$ feltételeket. Ha a célfüggvényben szerepel a $c_j x_j$, helyettesítsük a $c_j x'_j - c_j x''_j$ kifejezéssel, és ha az i-edik korlátozó feltételben szerepel az $a_{ij} x_j$ kifejezés, írjuk helyette az $a_{ij} x'_j - a_{ij} x''_j$ kifejezést. Az új lineáris programozási feladat tetszőleges $\bar{x}$ megengedett megoldása azonos az eredeti lineáris programozási feladat azon megengedett megoldásával, ahol $\bar{x}_j = \bar{x}'_j - \bar{x}''_j$, továbbá a célfüggvényérték is ugyanaz lesz, így a két lineáris programozási feladat ekvivalens. Ha ezt

az átalakítási sémát alkalmazzuk minden olyan változóra, melyre a nemnegativitási feltétel nem vonatkozik, az eredmény egy olyan ekvivalens lineáris programozási feladat lesz, melynek minden változójára vonatkozik a nemnegativitási feltétel.

Folytatva az előbbi példát, alakítsuk át úgy a feladatot, hogy minden változóra elő legyen írva a nemnegativitási feltétel. Az x_1 már ilyen, de az x_2 nem. Helyettesítsük az x_2 változót az x_2' és x_2'' változókkal, módosítsuk a feladatot, ekkor a következőket kapjuk:

$$\begin{array}{lrcrcrcl} \text{maximalizálandó} & 2x_1 & - & 3x_2' & + & 3x_2'' & & \\ \text{feltéve, hogy} & & & & & & & \\ & x_1 & + & x_2' & - & x_2'' & = & 7 \\ & x_1 & - & 2x_2' & + & 2x_2'' & \leq & 4 \\ & x_1, x_2', x_2'' & & & & & \geq & 0\,. \end{array} \tag{29.22}$$

Ezután alakítsuk át az egyenlőségként megadott korlátozó feltételeket egyenlőtlenségekké. Tegyük fel, hogy a lineáris programozási feladat egyik korlátozó feltétele az $f(x_1, x_2, \ldots, x_n) = b$. Mivel az $x = y$ akkor és csak akkor igaz, ha $x \geq y$ és $x \leq y$, az egyenlőségi feltételt helyettesíthetjük a következő egyenlőtlenségi feltételpárral: $f(x_1, x_2, \ldots, x_n) \leq b$ és $f(x_1, x_2, \ldots, x_n) \geq b$. Ezt végrehajtva valamennyi egyenlőségi feltételre, olyan lineáris programozási feladatot kapunk, melyben valamennyi korlátozó feltétel egyenlőtlenség.

Végül a nagyobb vagy egyenlő korlátozó feltételeket átalakíthatjuk kisebb vagy egyenlő feltétellé, ha az egyenlőtlenség mindkét oldalát megszorozzuk (-1)-gyel. Tetszőleges egyenlőtlenségre fennáll, hogy a

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

kifejezés ekvivalens a

$$\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i$$

kifejezéssel. Tehát az összes a_{ij} együtthatót helyettesítve a $-a_{ij}$ együtthatóval, továbbá valamennyi b_i értéket a $-b_i$ értékkel, egy ekvivalens kisebb vagy egyenlő korlátozó feltételt kapunk.

A példánkon is fejezzük be az átalakításokat, a (29.22) egyenlőséget egyenlőtlenségekké alakítva kapjuk a következőt:

$$\begin{array}{lrcrcrcl} \text{maximalizálandó} & 2x_1 & - & 3x_2' & + & 3x_2'' & & \\ \text{feltéve, hogy} & & & & & & & \\ & x_1 & + & x_2' & - & x_2'' & \leq & 7 \\ & x_1 & + & x_2' & - & x_2'' & \geq & 7 \\ & x_1 & - & 2x_2' & + & 2x_2'' & \leq & 4 \\ & x_1, x_2', x_2'' & & & & & \geq & 0\,. \end{array} \tag{29.23}$$

Végül a (29.23) korlátozó feltételt negáljuk. A változónevek következetes alkalmazása érdekében nevezzük át az x_2' változót x_2-vé és az x_2'' változót x_3-má. Így a következő szabályos alakot kapjuk:

maximalizálandó $2x_1 - 3x_2 + 3x_3$ (29.24)

feltéve, hogy

$$x_1 + x_2 - x_3 \leq 7 \quad (29.25)$$

$$-x_1 - x_2 + x_3 \leq -7 \quad (29.26)$$

$$x_1 - 2x_2 + 2x_3 \leq 4 \quad (29.27)$$

$$x_1, x_2, x_3 \geq 0. \quad (29.28)$$

A lineáris programozási feladat kiegyenlített alakjának felírása

A lineáris programozási feladat szimplex módszerrel történő hatékony megoldásához előnyben részesítjük a feladat olyan alakját, melyben a korlátozó feltételek egyenlőségként vannak megadva. Pontosabban fogalmazva, olyan alakra fogjuk átírni a feladatot, amelyben a nemnegativitási feltételeken kívül valamennyi korlátozó feltétel egyenlőségként van megadva. Legyen a

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \quad (29.29)$$

egy egyenlőtlenséggel megadott korlátozó feltétel. Bevezetjük az s segédváltozót, és az egyenlőtlenséget két feltétellé alakítjuk az alábbiak szerint:

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j, \quad (29.30)$$

$$s \geq 0. \quad (29.31)$$

Az s változót ***felesleg változónak*** (slack változónak) nevezzük, a (29.29) egyenlőtlenség jobb és bal oldala közti ***tágasságot,*** különbséget méri. Mivel a (29.29) egyenlőtlenség akkor és csak akkor igaz, ha a (29.30) egyenlőség és a (29.31) egyenlőtlenség egyszerre fennáll, így ezt az átalakítást a lineáris programozási feladat többi korlátozó feltételén is elvégezhetjük, míg megkapjuk azt az alakot, amelyben csak a nemnegativitási feltételek lesznek egyenlőtlenség alakban. A szabályos alak kiegyenlített alakba való átírásakor az i-edik egyenlőtlenség felesleg változójának jelölésére az x_{n+i} változót használjuk az s változó helyett. Az i-edik korlátozó feltételt tehát az

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j \quad (29.32)$$

alakra írjuk át, és kiegészítjük az $x_{n+i} \geq 0$ nemnegativitási feltétellel.

Ezt az átalakítást a szabályos alakban felírt lineáris programozási feladat valamennyi korlátozó feltételén alkalmazva új alakban kapjuk meg a feladatot. Nézzük ezt meg a példaként használt (29.24)–(29.28) egyenletekkel felírt lineáris programozási feladaton. Vezessük be az x_4, x_5 és x_6 felesleg változókat, majd végezzük el az átalakítást, az eredményül kapott alak a következő:

maximalizálandó

$$2x_1 - 3x_2 + 3x_3 \tag{29.33}$$

feltéve, hogy

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.34}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.35}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \tag{29.36}$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0\,. \tag{29.37}$$

Ebben a lineáris programozási feladatban a nemnegativitási feltételeken kívül valamennyi korlátozó feltétel egyenlőség, és valamennyi változóra elő van írva a nemnegativitási feltétel. Valamennyi egyenlőségként megadott korlátozó feltételre igaz, hogy az egyenlet bal oldalán mindig csak egyetlen változó áll, a többi változó az egyenlőség jobb oldalára került. Továbbá valamennyi egyenletben a jobb oldalon ugyanazok a változók szerepelnek, és csak ezek a változók szerepelnek a célfüggvényben. Az egyenletek bal oldalán álló változókat nevezzük ***bázisváltozónak,*** míg a jobb oldalon szereplőket ***nembázis-változónak.*** Néhány esetben, amikor egy lineáris programozási feladat eleget tesz ezeknek a feltételeknek, elhagyjuk a „maximalizálandó" és a „feltéve, hogy" kifejezéseket, és elhagyjuk a nemnegativitási feltétel kiírását is. A célfüggvényértéket a z változóval jelöljük. Az eredményül kapott alakot nevezzük kiegyenlített alaknak. A (29.33)–(29.37) lineáris programozási feladat kiegyenlített alakja a következő:

$$z = 2x_1 - 3x_2 + 3x_3 \tag{29.38}$$

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.39}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.40}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3\,. \tag{29.41}$$

Ahogy a szabályos alaknál is tettük, kényelmesebb lesz a kiegyenlített alakra is tömörebb ábrázolást bevezetni. A 29.3. alfejezetben látni fogjuk, hogy a szimplex algoritmus futása alatt a bázisváltozók és nembázis-változók halmazai változni fognak. A nembázisváltozók indexeinek halmazát N-nel, a bázisváltozók indexeinek halmazát B-vel fogjuk jelölni. A módszer alkalmazása alatt végig fennállnak az $|N| = n$, $|B| = m$, és $N \cup B = \{1, 2, \ldots, n+m\}$ egyenlőségek. Míg magukat az egyenleteket a B halmaz elemeivel indexeljük, addig az egyenlet jobb oldalán előforduló változók indexei az N halmazba tartoznak. A szabályos alakhoz hasonlóan b_i, c_j és a_{ij} jelölik a konstansokat és együtthatókat. Bevezetünk egy a célfüggvényben szereplő elhagyható v konstanst is. Tömören az (N, B, A, b, c, v) hatossal adjuk meg az alábbi kiegyenlített alakot:

$$z = v + \sum_{j \in N} c_j x_j \tag{29.42}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{minden } i \in B \text{ indexre,} \tag{29.43}$$

ahol az összes x változóra megköveteljük a nemnegativitási feltételt. Mivel a (29.43) kifejezésben a $\sum_{j \in N} a_{ij} x_j$ összeget kivonjuk, az a_{ij} együtthatók a (-1)-szeresei azon értékeknek, amelyek együtthatóként a kiegyenlített alakban „megjelennek”.

Például, ha a kiegyenlített alak

$$\begin{aligned} z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}, \end{aligned}$$

akkor $B = \{1, 2, 4\}$ és $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = \begin{pmatrix} c_3 & c_5 & c_6 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} -1/6 & -1/6 & -2/3 \end{pmatrix}^{\mathrm{T}}$ és $v = 28$. Vegyük észre, hogy az A, b és c indexei nem feltétlenül egymást követő egész számok, hanem a B és N indexhalmazoktól függnek. Azt pedig, hogy az A mátrix elemei (-1)-szeresei a kiegyenlített alakban megjelenő együtthatóknak, láthatjuk például az x_1-hez tartozó egyenletben, ott az $x_3/6$ tag szerepel, az a_{13} együttható viszont $-1/6$, nem pedig $+1/6$.

Gyakorlatok

29.1-1. Ha a (29.24)–(29.28) lineáris programozási feladatot a (29.19)–(29.21) tömör formában írjuk fel, mi lesz az n, m, A, b és c?

29.1-2. Adjuk meg a (29.24)–(29.28) lineáris programozási feladat három megengedett megoldását. Mi lesz a hozzájuk tartozó célfüggvényérték?

29.1-3. A (29.38)–(29.41) kiegyenlített alakban mi lesz N, B, A, b, c és v?

29.1-4. Adjuk meg a következő lineáris programozási feladat szabályos alakját:

minimalizálandó $2x_1 + 7x_2$

feltéve, hogy

$$\begin{aligned} x_1 &= 7 \\ 3x_1 + x_2 &\geq 24 \\ x_2 &\geq 0 \\ x_3 &\leq 0. \end{aligned}$$

29.1-5. Adjuk meg a következő lineáris programozási feladat kiegyenlített alakját:

maximalizálandó $2x_1 - 6x_3$

feltéve, hogy

$$\begin{aligned} x_1 + x_2 - x_3 &\leq 7 \\ 3x_1 - x_2 &\geq 8 \\ -x_1 + 2x_2 + 2x_3 &\geq 0 \\ x_1, x_2, x_3 &\geq 0 . \end{aligned}$$

Melyek a bázisváltozók és nembázis-változók?

29.1-6. Bizonyítsuk be, hogy az alábbi lineáris programozási feladatnak nincs megengedett megoldása:

maximalizálandó $3x_1 - 2x_2$

feltéve, hogy

$$\begin{aligned} x_1 + x_2 &\leq 2 \\ -2x_1 - 2x_2 &\leq -10 \\ x_1, x_2 &\geq 0 . \end{aligned}$$

29.1-7. Bizonyítsuk be, hogy az alábbi lineáris programozási feladat célfüggvénye nem korlátos:

maximalizálandó $x_1 - x_2$

feltéve, hogy

$$\begin{aligned} -2x_1 + x_2 &\leq -1 \\ -x_1 - 2x_2 &\leq -2 \\ x_1, x_2 &\geq 0 . \end{aligned}$$

29.1-8. Tegyük fel, hogy van egy általános lineáris programozási feladatunk, n változóval és m korlátozó feltétellel, és tegyük fel, hogy átalakítottuk szabályos alakra. Adjuk meg az így kapott lineáris programozási feladatban szereplő változók és korlátozó feltételek számának felső korlátját.

29.1-9. Mutassunk példát olyan lineáris programozási feladatra, amelyben a megengedett tartomány nem korlátos, de mégis van (véges) optimális célfüggvényérték.

29.2. Problémák mint lineáris programozási feladatok

Bár ebben a fejezetben elsősorban a szimplex módszerrel fogunk foglalkozni, az is fontos, hogy amikor egy konkrét problémával állunk szemben, felismerjük, mely esetben lehet azt lineáris programozási feladatként megfogalmazni. Ekkor ugyanis az ellipszoid módszerrel vagy a belsőpontos módszerekkel polinom időben megoldható a feladat. Számos lineáris programozási programcsomag létezik, melyek hatékonyan oldják meg a feladatokat, így ha a megoldandó problémát sikerül lineáris programozási feladatként megfogalmazni, akkor a gyakorlati megoldását már elvégzi egy megfelelő programcsomag.

Több példát is fogunk mutatni a lineáris programozási feladatként felírható problémákra. Két, korábban már tárgyalt témával kezdjük: egy gráf adott csúcsából induló legrövidebb utak problémája (lásd 24. fejezet) és a maximális folyam problémája (lásd 26. fejezet). Ezután a minimális költségű folyam problémát mutatjuk be. Létezik ugyan polinom idejű, nem lineáris programozási feladaton alapuló algoritmus a minimális költségű folyam problémára, de azt most nem vizsgáljuk. Végül a többtermékes folyam problémával foglalkozunk, melynek nem ismert más polinom idejű megoldó algoritmusa, csak az, amelyik a lineáris programozási feladaton alapul.

Legrövidebb utak

A 24. fejezetben bemutatott, adott csúcsból induló legrövidebb utak problémáját felírhatjuk lineáris programozási feladatként. Most csak az adott csúcspár közötti legrövidebb út problémát vizsgáljuk meg, a feladat azonban kiterjeszthető az általánosabb, adott csúcsból kiinduló legrövidebb utak problémára, ezt a 29.2-3. gyakorlatra hagyjuk.

Az adott csúcsból kiinduló legrövidebb utak problémájában adott egy élsúlyozott, irányított $G = (V, E)$ gráf, ahol a $w : E \to \mathbf{R}$ súlyfüggvény valós értékeket rendel az élekhez, a csúcsok között pedig ki van jelölve egy s kezdőcsúcs és egy t célcsúcs. A $d[t]$ értéket szeretnénk kiszámolni, mely az s csúcsból t csúcsba vezető egyik legrövidebb út súlya. A probléma lineáris programozási feladatként való megfogalmazásához meg kell határoznunk a változókat és a korlátozó feltételeket, amelyek kifejezik azt a tényt, hogy megtaláltuk az egyik legrövidebb utat az s és t csúcsok között. Szerencsére a Bellman–Ford-algoritmus pont ezt teszi. A Bellman–Ford-algoritmus minden v csúcsra kiszámol egy bizonyos $d[v]$ értéket úgy, hogy tetszőleges $(u, v) \in E$ él esetén a $d[v] \le d[u] + w(u, v)$ feltétel teljesüljön. A kezdőcsúcsra az algoritmus induláskor a $d[s] = 0$ értéket állítja be, amit aztán soha nem változtat meg. Ezek alapján a következő lineáris programozási feladatot kapjuk az s és t csúcsok közötti legrövidebb út problémára:

maximalizálandó $d[t]$ (29.44)

feltéve, hogy

$$d[v] \le d[u] + w(u, v) \quad \text{minden } (u, v) \in E \text{ élre,} \tag{29.45}$$

$$d[s] = 0. \tag{29.46}$$

Ennek a lineáris programozási feladatnak az egyes csúcsokhoz rendelt $d[v]$ értékek a változói, és mivel minden $v \in V$ csúcshoz egy ilyen érték tartozik, a változók száma $|V|$. A korlátozó feltételek száma $|E| + 1$, mivel minden élhez tartozik egy feltétel, egy további pedig kiköti, hogy az s kezdőcsúcshoz a $d[s] = 0$ érték tartozik.

Maximális folyamok

A maximális folyam probléma is kifejezhető lineáris programozási feladatként. Emlékeztetőül: adott egy irányított $G = (V, E)$ gráf, amelyben minden $(u, v) \in E$ élhez rendelünk egy $c(u, v) \ge 0$ nemnegatív kapacitás értéket, továbbá van két kitüntetett csúcs, az s forrás (a továbbiakban termelő) és a t nyelő (a továbbiakban fogyasztó) csúcsok. A 26.1. alfejezetben adott definíció szerint hálózati folyamnak nevezzük az $f : V \times V \to \mathbf{R}$ valós értékű függvényt, ha kielégíti a kapacitási megszorítás, a ferde szimmetria és a megmaradási szabály követelményeit. A maximális hálózati folyam olyan folyam, mely eleget tesz ezeknek

a korlátozó feltételeknek, és emellett a folyamnagyság maximális. Folyamnagyságon az s forrásból induló élekhez tartozó folyamértékek összegét értjük. Tudjuk tehát, hogy a hálózati folyam eleget tesz bizonyos lineáris korlátozó feltételeknek, továbbá a folyam nagysága lineáris függvénnyel írható le. Felidézve azt is, hogy $c(u,v) = 0$, ha $(u,v) \notin E$, a maximális folyam problémát az alábbi lineáris programozási feladatként írhatjuk fel:

maximalizálandó $\sum_{v \in V} f(s,v)$ (29.47)

feltéve, hogy

$$f(u,v) \leq c(u,v) \quad \text{minden } u,v \in V \text{ csúcsra,} \tag{29.48}$$

$$f(u,v) = -f(v,u) \quad \text{minden } u,v \in V \text{ csúcsra,} \tag{29.49}$$

$$\sum_{v \in V} f(u,v) = 0 \quad \text{minden } u \in V - \{s,t\} \text{ csúcsra.} \tag{29.50}$$

Ez egy $|V|^2$ változós lineáris programozási feladat. A változók a gráf két tetszőlegesen választott csúcsa közti folyamot jelentik. A korlátozó feltételek száma $2|V|^2 + |V| - 2$.

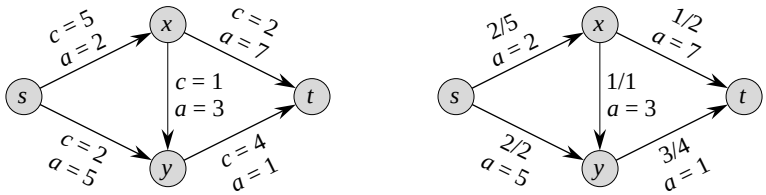
Kisebb méretű lineáris programozási feladat általában könnyebben megoldható. Vegyük észre, hogy minden olyan u, v csúcspárra, melyre $(u,v) \notin E$, a folyam és a kapacitás értéke 0, így a (29.47)–(29.50) lineáris programozási feladat átírható egy hatékonyabb, $O(V + E)$ korlátozó feltételt tartalmazó alakba, lásd a 29.2-5. gyakorlatot.

Minimális költségű folyam

Ebben a részben eddig olyan lineáris programozási feladatként megoldható problémákkal foglalkoztunk, amelyekre létezik más, hatékony megoldó algoritmus. Az olyan algoritmus, melyet speciálisan az adott feladat megoldására terveztek, mint például Dijkstra algoritmusa az egy adott csúcsból kiinduló minimális utak megkeresésére, vagy a előfolyam algoritmusa a maximális folyam meghatározásához, általában mind elméletben, mind gyakorlatban hatékonyabb a lineáris programozási megközelítésnél.

A lineáris programozás igazi erőssége akkor mutatkozik meg, amikor segítségével egy új, eddig még megoldatlan problémára adhatunk megoldó algoritmust. Emlékezzünk a fejezet elején bemutatott választási problémára, melyben a politikusnak azt a stratégiát kellett megtalálnia, mellyel a lehető legtöbb szavazatot nyerheti úgy, hogy nem költ el túl sok pénzt. Ez a feladat a könyvünkben eddig tárgyalt algoritmusokkal nem oldható meg, csak a lineáris programozási módszerek valamelyikével. Bőven találni olyan gyakorlati problémát, amelyek lineáris programozási feladatként megfogalmazhatók. Vannak olyan problémák, amelyekre még nem ismert hatékony megoldó algoritmus, de a probléma bizonyos változatai lineáris programozási megközelítéssel megoldhatók.

Tekintsük a maximális folyam problémának az alábbi általánosított változatát. Tartozzon minden (u,v) élhez a $c(u,v)$ kapacitáson kívül egy valós $a(u,v)$ érték is, amely egységnyi anyag szállításának a költségét adja meg az adott él mentén. Technikai okok miatt kikötjük, hogy ha $a(u,v) > 0$, akkor $a(v,u) = 0$. Ha az (u,v) élen $f(u,v)$ egységnyi anyagot szállítunk, akkor $a(u,v)f(u,v)$ költségünk keletkezik. A szállítandó anyagmennyiség d, tehát az s termelőtől a t fogyasztóhoz el kell juttatnunk d mennyiségű anyagot úgy, hogy a szállítási költség, $\sum_{(u,v) \in E} a(u,v)f(u,v)$ minimális legyen. Ezt a problémát ***minimális költségű folyam problémának*** nevezzük.



29.3. ábra. (a) Egy példa a minimális költségű folyam problémára. A kapacitást c-vel, a költséget a-val jelöljük. Az s csúcs a termelő, a t csúcs a fogyasztó. 4 egységnyi anyagot kell eljuttatnunk s-ből t-be. **(b)** Ez az ábra mutatja az előbbi minimális költségű folyam feladat megoldását. Az egyes élekre a folyamértéket és a kapacitást írtuk folyamérték/kapacitás alakban.

A 29.3(a) ábrán láthatunk egy példát a minimális költségű folyam problémára. 4 egységnyi anyagot kívánunk az s csúcsból a t csúcsba eljuttatni minimális költséggel. Valamely, a (29.48)–(29.50) feltételeknek eleget tevő f függvény – azaz megengedett folyam – összköltsége $\sum_{(u,v)\in E} a(u,v)f(u,v)$. Olyan 4 egység értékű megengedett folyamot kell találnunk, amelynek költsége minimális. A 29.3(b) ábra a feladat egy optimális megoldását mutatja be, az összköltség $\sum_{(u,v)\in E} a(u,v)f(u,v) = (2\cdot 2) + (5\cdot 2) + (3\cdot 1) + (7\cdot 1) + (1\cdot 3) = 27$.

Ismertek olyan polinom idejű algoritmusok, melyek speciálisan a minimális költségű folyam problémát oldják meg, de azok kívül esnek a könyv vizsgálódási területén. Azonban a minimális folyam problémát megfogalmazhatjuk lineáris programozási feladat formájában. A kapott lineáris programozási feladat hasonlít a maximális folyam problémára felírt feladathoz, azonban tartalmaz egy további feltételt, amely előírja, hogy a folyamérték pontosan d egység legyen, és a célfüggvényünk most a költség minimalizálása lesz:

minimalizálandó
$$\sum_{(u,v)\in E} a(u,v)f(u,v) \tag{29.51}$$

feltéve, hogy

$$f(u,v) \le c(u,v) \quad \text{minden } u,v \in V \text{ csúcsra,} \tag{29.52}$$

$$f(u,v) = -f(v,u) \quad \text{minden } u,v \in V \text{ csúcsra,} \tag{29.53}$$

$$\sum_{v\in V} f(u,v) = 0 \quad \text{minden } u \in V - \{s,t\} \text{ csúcsra,} \tag{29.54}$$

$$\sum_{v\in V} f(s,v) = d. \tag{29.55}$$

Többtermékes folyam

Utolsóként ismét egy folyam problémát vizsgálunk meg. Tegyük fel, hogy a 26.1. alfejezetben szereplő Hokipakk Vállalat elhatározza, hogy változatosabbá teszi a termékskáláját, és nemcsak jéghoki korongokat szállít, hanem ütőket és sisakokat is. Mindegyik terméknek megvan a saját gyára és a saját raktára, és minden nap az elkészült termékeket a gyárból a raktárba kell szállítani. Az ütőket Vancouverben gyártják és Saskatoonba kell szállítani, a sisakokat Edmontonban készítik és Reginába szállítják. A szállítási kapacitásokat leíró hálózat változatlan, a különféle árucikkeknek, azaz ***termékeknek*** osztozniuk kell a meglév ő szállítási hálózaton.

A példa bemutatja a ***többtermékes folyam probléma*** lényegét. Ismét adott egy $G = (V, E)$ irányított gráf, melynek mindegyik $(u, v) \in E$ éléhez tartozik egy nemnegatív $c(u, v) \geq 0$ kapacitás érték. A maximális folyam probléma definiálásához hasonlóan értelemszerű, hogy $c(u, v) = 0$ minden $(u, v) \notin E$ csúcspárra, továbbá kikötjük, hogy ha $c(u, v) > 0$, akkor $c(v, u) = 0$. Ezenkívül adott k fajta termék, jelölje őket $K_1, K_2, \ldots, K_k$, az i-edik terméket a $K_i = (s_i, t_i, d_i)$ hármas határozza meg, ahol s_i az i-edik termék termelője, t_i a fogyasztója és d_i az elvárt folyamérték, azaz az s_i termelőtől a t_i fogyasztóhoz vezető folyam előírt értéke. Definiáljuk az i-edik termékhez tartozó folyamot egy f_i valós értékű függvényként, mely kielégíti a kapacitási megszorítás, a ferde szimmetria és a megmaradási szabály követelményeit ($f_i(u, v)$ jelöli az i-edik termék folyamértékét u csúcsból v csúcsba). ***Összfolyamnak*** nevezzük a különféle termékek folyamainak összegét: $f(u, v) = \sum_{i=1}^{k} f_i(u, v)$. Egy adott (u, v) élen az összfolyam értéke nem lehet nagyobb, mint az (u, v) él kapacitása. A probléma megfogalmazásából következik, hogy nincs minimalizálandó függvény, csak azt kell eldöntenünk, hogy található-e a feltételeknek megfelelő folyam. Így egy olyan lineáris programozási feladatot kapunk, melynek célfüggvénye „konstans nulla”:

minimalizálandó 0

feltéve, hogy

$$\begin{aligned}
\sum_{i=1}^{k} f_i(u, v) &\leq c(u, v) && \text{minden } u, v \in V \text{ esetén,} \\
f_i(u, v) &= -f_i(v, u) && \text{minden } i = 1, 2, \ldots, k \text{ és} \\
& && \text{minden } u, v \in V \text{ esetén,,} \\
\sum_{v \in V} f_i(u, v) &= 0 && \text{minden } i = 1, 2, \ldots, k \text{ és} \\
& && \text{minden } u \in V - \{s_i, t_i\} \text{ esetén,} \\
\sum_{v \in V} f_i(s_i, v) &= d_i && \text{minden } i = 1, 2, \ldots, k \text{ esetén.}
\end{aligned}$$

Egyedüli polinom költségű megoldása ennek a problémának az, ha megfogalmazzuk lineáris programozási feladatként, majd egy polinom idejű lineáris programozási módszerrel megoldjuk.

Gyakorlatok

29.2-1. Alakítsuk szabályos alakra a két adott csúcs közti legrövidebb út problémára felírt (29.44)–(29.46) lineáris programozási feladatot.

29.2-2. Írjuk fel konkrét értékekkel lineáris programozási feladatként a 24.2(a) ábrán bemutatott feladatot, melyben a legrövidebb utat keressük az s és y csúcsok közt.

29.2-3. Az adott csúcsból induló legrövidebb utak problémában a gráf egy adott s csúcsából az összes v csúcshoz keresünk legrövidebb utat. Írjuk fel a problémát egy adott G gráfra lineáris programozási feladatként. A megoldásban minden $v \in V$ csúcsra a $d[v]$ érték az s-ből v-be vezető legrövidebb út súlya.

29.2-4. Írjuk fel konkrét értékekkel lineáris programozási feladatként a 26.1(a) ábrán bemutatott maximális folyam feladatot.

29.2-5. Alakítsuk át a (29.47)–(29.50) lineáris programozási feladatot olyan alakra, melyben a korlátozó feltételek száma $O(V + E)$.

29.2-6. Adjuk meg az egy adott $G = (V, E)$ páros gráfra vonatkozó maximális párosítási problémát leíró lineáris programozási feladatot.

29.2-7. A ***minimális költségű többtermékes folyam problémában*** adott egy $G = (V, E)$ irányított gráf, melynek mindegyik $(u, v) \in E$ éléhez tartozik egy nemnegatív $c(u, v) \geq 0$ kapacitás érték és egy $a(u, v)$ költség. A többtermékes folyam problémához hasonlóan adott k fajta $K_1, K_2, \ldots, K_k$ termék, ahol az i-edik termék a $K_i = (s_i, t_i, d_i)$ hármassal írható le. Az i-edik termékhez tartozó f_i folyamot, valamint az (u, v) élhez tartozó $f(u, v)$ összfolyamot hasonlóan definiáljuk, mint a többtermékes problémánál tettük. Egy megengedett folyam az, melyben az összfolyamérték minden (u, v) élen kisebb, mint az (u, v) kapacitása. Egy folyam költsége $\sum_{(u,v)\in V} a(u, v) f(u, v)$, a cél a minimális költségű megengedett folyam megtalálása. Fogalmazzuk meg a problémát lineáris programozási feladatként.

29.3. A szimplex módszer

A szimplex módszer a lineáris programozási feladat megoldásának klasszikus módszere. A könyvben ismertetett algoritmusok nagy részével ellentétben futási ideje a legrosszabb esetben nem polinom idejű. Azonban belátást enged a lineáris programozási feladatok természetébe, amellett számos konkrét feladat megoldásánál figyelemre méltóan gyors.

A fejezet korábbi részében láthattuk a módszer geometriai szemléltetését, emellett a szimplex módszer a 28.3. alfejezetben bemutatott Gauss-eliminációval is mutat némi rokonságot. A Gauss-elimináció lineáris egyenletrendszerek megoldásának módszere. Iterációs lépésekkel az egyenletrendszert egymással ekvivalens alakokba transzformáljuk, egyre közelítve azt az alakot, melyből a megoldás egyszerű visszahelyettesítésekkel megkapható. Hasonlóan működik a szimplex algoritmus is, ezért úgy is tekinthetjük, mintha egy lineáris egyenlőtlenségrendszeren futó Gauss-elimináció lenne.

Vizsgáljuk meg a szimplex módszer iterációs lépéseinek hátterében húzódó elveket. Minden egyes iterációban az eredményül kapott kiegyenlített alakból könnyen megkaphatjuk a hozzá tartozó „bázismegoldást”: a nembázis-változók értékét 0-nak választjuk, majd kiszámítjuk a bázisváltozók értékét a korlátozó feltételekből. A kapott bázismegoldás mindig a megengedett tartomány (poliéder) egy csúcsához tartozik. Algebrailag egy iteráció a feladat egyik kiegyenlített alakjáról egy vele ekvivalens másik kiegyenlített alakra tér át. Az aktuális megengedett bázismegoldáshoz tartozó célfüggvényérték legalább akkora, de általában nagyobb, mint az előző bázismegoldáshoz tartozó. Ezt a növekedést úgy biztosítjuk, hogy olyan nembázis-változót választunk, amelynek értékét növelve a célfüggvény értéke is nőni fog. Azt, hogy mekkora értékkel növelhető a változó, a feltételek korlátozzák. Addig növeljük, míg valamelyik bázisváltozó értéke 0 nem lesz. Ekkor újra felírjuk a kiegyenlített alakot úgy, hogy felcseréljük a bázisváltozó és a kiválasztott nembázis-változó szerepét. Bár az algoritmus bemutatásánál, majd a bizonyításnál a feladatot a változók adott értékei mellett vizsgáljuk, természetesen a módszer nemcsak ennek megoldására alkalmas. Egyszerűen szólva addig alakítjuk a feladatot leíró egyenletrendszert, míg a megoldás nyilvánvalóan adódik.

Példa a szimplex algoritmus működésére

Kezdjük egy kiterjesztett példával. Tekintsük az alábbi szabályos alakú lineáris programozási feladatot:

$$\text{maximalizálandó} \quad 3x_1 + x_2 + 2x_3 \tag{29.56}$$

feltéve, hogy

$$x_1 + x_2 + 3x_3 \leq 30 \tag{29.57}$$

$$2x_1 + 2x_2 + 5x_3 \leq 24 \tag{29.58}$$

$$4x_1 + x_2 + 2x_3 \leq 36 \tag{29.59}$$

$$x_1, x_2, x_3 \geq 0\,. \tag{29.60}$$

Elsőként kiegyenlített alakra kell hozni a feladatot, hogy a szimplex módszert alkalmazhassuk rá, ezt az eljárást már bemutattuk a 29.1. alfejezetben. Ez algebrai átalakítás. A felesleg változóknak algoritmikus szempontból fontos jelentésük van. Emlékezzünk vissza, hogy a 29.1. alfejezetben tárgyaltak szerint minden változóra nézve megköveteljük a nemnegativitást. Tekintsünk egy egyenlőségi feltételt. Azt mondjuk, hogy ez a feltétel a nembázis-változók adott értékbeállítása mellett ***szoros,*** ha a hozzá tartozó bázisváltozó értéke 0 lesz. Hasonlóan, ha az adott feltételhez tartozó bázisváltozó értéke negatív lesz, akkor a nembázis-változók értékbeállítása ***sérti*** a feltételt. Ebből adódik, hogy a felesleg változók pont azt jelenítik meg, hogy az egyes korlátozó feltételek milyen messze vannak a szorosságtól, és így segítenek eldönteni, hogy a nembázis-változók értékeit mennyivel növelhetjük meg, hogy még ne sértsék meg a korlátozó feltételeket.

Rendeljük hozzá a (29.57)–(29.59) egyenlőtlenségekhez az x_4, x_5 és x_6 felesleg változókat, majd írjuk fel a kiegyenlített alakot, az eredmény az alábbi lesz:

$$z = 3x_1 + x_2 + 2x_3 \tag{29.61}$$

$$x_4 = 30 - x_1 - x_2 - 3x_3 \tag{29.62}$$

$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \tag{29.63}$$

$$x_6 = 36 - 4x_1 - x_2 - 2x_3\,. \tag{29.64}$$

A (29.62)–(29.64) egyenletrendszer 6 változóból és 3 egyenletből áll. Az x_1, x_2 és x_3 tetszőleges beállítása egyértelműen meghatározza az x_4, x_5 és x_6 változók értékeit; így az egyenletrendszernek végtelen sok megoldása van. A megoldás megengedett, ha az $x_1, x_2, \ldots x_6$ értéke nemnegatív. A megengedett megoldások száma is végtelen lehet. A későbbi bizonyításoknál még hasznos lesz, hogy egy feladatnak végtelen sok megoldása lehet. Most koncentráljunk a ***bázismegoldásra:*** állítsuk az összes jobb oldalon szereplő (nembázis) változó értékét 0-ra és számítsuk ki a bal oldalon álló változók (ezek a bázisváltozók) így adódó értékeit. A példában a bázismegoldás $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ és a hozzá tartozó célfüggvényérték: $z = (3\cdot 0)+(1\cdot 0)+(2\cdot 0) = 0$ lesz. Vegyük észre, hogy a bázismegoldásban minden $i \in B$ esetén pont $x_i = b_i$ adódik. A szimplex módszer egy iterációja során az egyenletrendszert és a célfüggvényt átalakítjuk úgy, hogy a jobb oldalra egy új indexhalmazhoz tartozó változók kerüljenek. Így a módosított alakhoz tartozó bázismegoldás is különbözni fog az előzőtől. Hangsúlyozzuk azonban, hogy az átalakítás nem változtat a vizsgált lineáris programozási feladaton; annak megengedett megoldásai az iterációs lépés előtt és után ugyanazok lesznek. Viszont a módosított alakhoz tartozó bázismegoldás különbözni fog az iteráció előtti bázismegoldástól.

Ha a bázismegoldás egyben megengedett megoldás is, akkor ***megengedett bázismegoldásról*** beszélünk. A szimplex módszer végrehajtása alatt a bázismegoldások majdnem mindig megengedett megoldások is egyben. A 29.5. alfejezetben fogunk olyan példát látni, amikor az algoritmus első néhány iterációjában kapott bázismegoldás nem megengedett.

A cél az, hogy az átalakítással kapott bázismegoldáshoz tartozó célfüggvényérték nagyobb legyen az előző bázismegoldás célfüggvényértékénél. Választunk egy olyan x_e nembázis-változót, melynek együtthatója a célfüggvényben pozitív, majd az x_e értékét megnöveljük a lehetséges legnagyobb mértékben, úgy, hogy az még ne sértse a korlátozó feltételeket. Az x_e bázisváltozóvá válik, és valamelyik x_l bázisváltozóból nembázis-változó lesz. A többi bázisváltozó értéke és a célfüggvény értéke is megváltozhat.

Folytatva a példát, tegyük fel, hogy az x_1 nembázis-változó értékének megnövelését választjuk. Ha x_1 értékét növeljük, az x_4, x_5 és x_6 mindegyikének értéke csökken. Mivel mindegyik változóra érvényes a nemnegativitási feltétel, egyikük sem válhat negatívvá. Ha az x_1 értéke 30-nál nagyobb, az x_4 negatívvá válik, ha 12-nél nagyobb, az x_5 értéke, és ha 9-nél, az x_6 értéke válik negatívvá. A harmadik, a (29.64) korlátozó feltétel a legszorosabb, ez határozza meg x_1 lehetséges értékeinek felső korlátját. Így x_1 és x_6 változók fognak szerepet cserélni. Oldjuk meg x_1-re a (29.64) egyenletet:

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \,. \tag{29.65}$$

A többi egyenletet is átírjuk, úgy, hogy az x_1 helyett az x_6 szerepeljen a jobb oldalon, x_1 helyére beírjuk a (29.65) egyenlet jobb oldalát. Ezt a (29.62) egyenleten végrehajtva, majd rendezve, a következőt kapjuk:

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \,. \end{aligned} \tag{29.66}$$

Végezzük ezt el a (29.63) korlátozó feltételen és a (29.61) célfüggvényen is, ekkor a feladat új alakja az alábbi lesz:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \tag{29.67}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.68}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{29.69}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \,. \tag{29.70}$$

Ezt a műveletet ***pivotálásnak*** nevezzük. A fent bemutatottak szerint, a pivotálás azt jelenti, hogy választunk egy x_e nembázis-változót, ez lesz a ***bázisba belépő változó,*** és egy x_l bázisváltozót, ami a ***bázisból kilépő változó*** lesz, és felcseréljük a szerepüket.

A (29.67)–(29.70) lineáris programozási feladat ekvivalens a (29.61)–(29.64) lineáris programozási feladattal a következők miatt. A szimplex módszer alkalmazása során a következő műveleteket hajtjuk végre: egyenlet átalakítása, úgy, hogy változók a bal és jobb oldal között mozognak, valamint egy egyenlőség behelyettesítése egy másik egyenlőségbe. Nyilvánvaló, hogy az első művelet ekvivalens problémához vezet, és elemi lineáris algebrai szabályok miatt a második is.

Vizsgáljuk meg az ekvivalenciát a példánkon. Látható, hogy az eredeti $(0, 0, 0, 30, 24, 36)$ bázismegoldás kielégíti az új (29.68)–(29.70) egyenleteket, és a hozzá tartozó célfüggvényérték $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$ lesz. Most nézzük a másik irányt:

az új (29.68)–(29.70) rendszerhez tartozó bázismegoldás $(9, 0, 0, 21, 6, 0)$ lesz, melyhez a $z = 27$ célfüggvényérték tartozik. Utánaszámolással meggyőződhetünk, hogy ez a megoldás kielégíti a (29.62)–(29.64) egyenleteket is, és behelyettesítve a (29.61) célfüggvénybe a $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$ célfüggvényértéket kapjuk.

Folytatva a példát, egy újabb olyan változót kell találnunk, melynek értékét megnövelhetjük. Az x_6 nem jó, hisz ha ennek értékét növeljük, a célfüggvényérték csökken. Marad tehát az x_2 vagy az x_3 változó; válasszuk mondjuk az x_3-at. Milyen nagyságra növelhető x_3 értéke anélkül, hogy megsértenénk bármelyik korlátozó feltételt? A (29.68) szerint ennek felső határa a 18, a (29.69) szerint $42/5$ és a (29.70) szerint $3/2$. A harmadik korlát a legszorosabb, így a harmadik feltételt alakítjuk át úgy, hogy az x_3 kerüljön a bal oldalra és x_5 a jobb oldalra. Ezután a (29.67)–(29.69) egyenletekben x_3 helyére behelyettesítjük az így kapott egyenletet, és ezzel a következő, az eredetivel ekvivalens rendszert kapjuk:

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.71}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.72}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \tag{29.73}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}. \tag{29.74}$$

Az ehhez tartozó bázismegoldás a $(33/4, 0, 3/2, 69/4, 0, 0)$, melyhez a $111/4$ célfüggvényérték tartozik. A célfüggvény értékének további növelését egyedül az x_2 változó értékének növelésével érhetjük el. A korlátozó feltételek a növekedés felső korlátjára rendre a $132, 4$ és ∞ értéket adják. (A ∞ értéket a (29.74) korlátozó feltételből kapjuk, hiszen ha növeljük az x_2 értékét, az x_4 bázisváltozó értéke is növekedni fog. Így ez az egyenlet nem jelent felső korlátot az x_2 értékének megnövelésére.) Az x_2 értékét 4-re növeljük, és bázisból kilépő változónak választjuk. Megoldjuk x_2-re a (29.73) egyenletet, majd a behelyettesítés és rendezés után a következőt kapjuk:

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{29.75}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{29.76}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{29.77}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2}. \tag{29.78}$$

Most olyan ponthoz érkeztünk, amikor a célfüggvény valamennyi együtthatója negatív. A fejezet későbbi részében be fogjuk bizonyítani, hogy ilyenkor az aktuális kiegyenlített alakhoz tartozó bázismegoldás optimális. Ez a bázismegoldás most a $(8, 4, 0, 18, 0, 0)$, a hozzá tartozó optimális célfüggvényérték 28. Térjünk vissza az eredetileg kitűzött (29.56)–(29.60) lineáris programozási feladathoz. Abban csak az x_1, x_2 és x_3 változók szerepeltek, így a megoldás $x_1 = 8$, $x_2 = 4$ és $x_3 = 0$, a hozzá tartozó célfüggvényérték $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Megjegyezzük, hogy a felesleg változók értéke a végső megoldásban azt méri, hogy az egyes egyenlőtlenségek milyen bőven teljesülnek. Az x_4 felesleg változó értéke 18, és a (29.57) egyenlőtlenségben a bal oldal értéke $8 + 4 + 0 = 12$, ami pont 18-cal kevesebb a jobb oldalon álló 30-nál. Az x_5 és x_6 felesleg változók értéke 0, és valóban

a (29.58) és (29.59) egyenlőtlenségek jobb és bal oldala egyenlő. Vegyük észre, hogy bár az eredeti feladatra felírt kiegyenlített alakban az együtthatók egész számok, más lineáris programozási feladatokban ez nem szükségszerűen igaz, úgyszintén a közbens ő megoldások sem feltétlenül egész értékűek. Továbbá az sem biztos, hogy egy lineáris programozási feladat végső eredménye egész értékű: az pusztán a véletlen műve, hogy ebben a példában egész értékű végső megoldást kaptunk.

A pivotálás

Most leírjuk formálisan a pivotálást. A PIVOTÁLÁS eljárás bemenete egy az (N, B, A, b, c, v) hatos formájában megadott kiegyenlített alakú feladat, valamint a bázisból kilép ő x_l változó l indexe, és bázisba belépő x_e változó e indexe. Az eljárás eredményként az új kiegyenlített alakot megadó $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$ hatost küldi vissza. (Emlékeztetjük ismét az olvasót, hogy az A, illetve $\widehat{A}$ mátrix elemeinek értékei a megfelelő kiegyenlített alakban szereplő együtthatóknak a (-1)-szeresei.)

PIVOTÁLÁS(N, B, A, b, c, v, l, e)

```
 1  ▷ Az új x_e bázisváltozó egyenletéhez tartozó együtthatók meghatározása.
 2  b̂_e ← b_l / a_le
 3  for minden j ∈ N − {e} indexre
 4      do â_ej ← a_lj / a_le
 5  â_el ← 1 / a_le
 6  ▷ A többi korlátozó feltétel együtthatóinak kiszámítása.
 7  for minden i ∈ B − {l} indexre
 8      do b̂_i ← b_i − a_ie b̂_e
 9          for minden j ∈ N − {e} indexre
10              do â_ij ← a_ij − a_ie â_ej
11          â_il ← −a_ie â_el
12  ▷ A célfüggvény kiszámítása.
13  v̂ ← v + c_e b̂_e
14  for minden j ∈ N − {e} indexre
15      do ĉ_j ← c_j − c_e â_ej
16  ĉ_l ← −c_e â_ej
17  ▷ A bázis- és nembázis-változók indexeit tartalmazó halmazok módosítása.
18  N̂ = N − {e} ∪ {l}
19  B̂ = B − {l} ∪ {e}
20  return (N̂, B̂, Â, b̂, ĉ, v̂)
```

A PIVOTÁLÁS a következőképpen működik. A 2–5. sorokban kiszámolja az x_e változó egyenletének új együtthatóit. Ehhez átrendezi azt az egyenletet, amelynek bal oldalán az x_l változó szerepelt, úgy, hogy az x_e-t kifejezi a többi változóból, és a bal oldalra írja. A 7–11. sorokban módosítja a többi egyenletet, x_e minden előfordulásának helyére a fenti új egyenlet jobb oldalát írja. A 13–16. sorokban ugyanezt a behelyettesítést végzi el a célfüggvényen. Végül a 18. és 19. sorokban módosítja a bázisváltozók és nembázis-változók indexeit nyilvántartó halmazokat. A 20. sorban eredményként visszaadja az új kiegyenlített

alakot. Figyeljük meg, hogy a PIVOTÁLÁS eljárás 4. sorában 0-val osztanánk, ha az $a_{le} = 0$ lenne, azonban a 29.2. és 29.12. lemmák bizonyításában látni fogjuk, hogy a PIVOTÁLÁS csak akkor hívódik meg, ha $a_{le} \neq 0$.

Most összefoglaljuk, milyen hatással van a PIVOTÁLÁS eljárás a bázismegoldás változóinak értékeire.

29.1. lemma. *Tekintsük a* PIVOTÁLÁS(N, B, A, b, c, v, l, e) *eljáráshívást, ahol* $a_{le} \neq 0$. *Jelölje az eljárás visszatérési értékeit* $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$, *és* $\bar{x}$ *a hívás utáni bázismegoldást. Ekkor*

1. $\bar{x}_j = 0$ *minden* $j \in \widehat{N}$ *indexre.*
2. $\bar{x}_e = \dfrac{b_l}{a_{le}}$.
3. $\bar{x}_i = b_i - a_{ie}\widehat{b}_e$ *minden* $i \in \widehat{B} - \{e\}$ *indexre.*

Bizonyítás. Az első állítás azt a tényt fogalmazza meg, hogy a bázismegoldásban az összes nembázis-változó értéke 0. Ha az

$$\bar{x}_i = \widehat{b}_i - \sum_{j \in \widehat{N}} \widehat{a}_{ij}\bar{x}_j$$

korlátozó feltételben a nembázis-változók értékeit 0-ra állítjuk, akkor $\bar{x}_i = \widehat{b}_i$ adódik minden $i \in \widehat{B}$ indexre. Mivel $e \in B$, a PIVOTÁLÁS eljárás 2. sora miatt

$$\bar{x}_e = \widehat{b}_e = \frac{b_l}{a_{le}},$$

ezzel a második állítást beláttuk. Hasonlóan, minden $i \in \widehat{B} - \{e\}$ indexre a 8. sor miatt

$$\bar{x}_i = \widehat{b}_i = b_i - a_{ie}\widehat{b}_e,$$

tehát beláttuk a harmadik állítást is. ■

A szimplex módszer formális algoritmusa

Most már elkészíthetjük a példán keresztül bemutatott szimplex módszer algoritmusát. A példa igen barátságos volt, számos fontos kérdés vár még megvitatásra:

- Hogyan tudjuk eldönteni, hogy egy lineáris programozási feladatnak van-e megengedett megoldása?
- Mi a teendő, ha a lineáris programozási feladatnak van megengedett megoldása, de a kezdeti bázismegoldás nem megengedett?
- Hogyan lehet észrevenni, hogy egy lineáris programozási feladatnak nem korlátos a célfüggvénye?
- Hogyan válasszuk meg a bázisba belépő és a bázisból kilépő változókat?

A 29.5. alfejezetben megmutatjuk, hogyan dönthető el egy lineáris programozási feladatról, hogy van-e megengedett megoldása, és ha van, hogyan található olyan kiegyenlített alak, amelyhez megengedett bázismegoldás tartozik. Ezért most feltesszük, hogy van egy

SZIMPLEX-KEZDÉS(A, b, c) eljárásunk, melynek bemenete a lineáris programozási feladat szabályos alakja, azaz az A $(m \times n)$-es mátrix, az m dimenziós $b = (b_i)$ vektor és az n dimenziós $c = (c_j)$ vektor. Ha a feladatnak nincs megengedett megoldása, akkor ez az eljárás a nincs megengedett megoldás üzenetet küldi vissza és leáll, egyébként egy olyan kiegyenlített alakot küld vissza, amelyhez megengedett bázismegoldás tartozik.

A fentiek szerint a SZIMPLEX eljárás bemenete egy szabályos alakú lineáris programozási feladat, kimenete pedig az az n-dimenziós $\bar{x} = (\bar{x}_j)$ vektor, amely a (29.19)–(29.21) lineáris programozási feladat optimális megoldása.

SZIMPLEX(A, b, c)

```
 1  (N, B, A, b, c, v) ← SZIMPLEX-KEZDÉS(A, b, c)
 2  while van olyan j ∈ N, amelyre c_j > 0
 3        do válasszunk egy e ∈ N indexet, melyre c_e > 0
 4           for minden i ∈ B indexre
 5               do if a_ie > 0
 6                     then Δ_i ← b_i / a_ie
 7                     else Δ_i ← ∞
 8           legyen l ∈ B egy olyan index, amelyre a Δ_i minimális
 9           if Δ_l = ∞
10              then return „nem korlátos”
11              else (N, B, A, b, c, v) ← PIVOTÁLÁS(N, B, A, b, c, v, l, e)
12  for i ← 1 to n
13     do if i ∈ B
14           then x̄_i ← b_i
15           else x̄_i ← 0
16  return (x̄_1, x̄_2, ..., x̄_n)
```

A SZIMPLEX eljárás a következő módon működik. Az 1. sorban meghívja az előbb leírt SZIMPLEX-KEZDÉS(A, b, c) eljárást, mely vagy megállapítja, hogy a lineáris programozási feladatnak nincs megengedett megoldása, vagy visszaadja a feladat olyan kiegyenlített alakját, amelyhez megengedett bázismegoldás tartozik. Az algoritmus lényegi része a 2–11. sorokban található **while** ciklus. Ez a ciklus akkor fejeződik be, ha a célfüggvény valamennyi együtthatója negatív. Ha ez még nem igaz, a 3. sorban választunk egy x_e nembázis-változót, melynek célfüggvénybeli együtthatója pozitív, ez lesz a bázisba belépő változó. Ha a célfüggvénynek több pozitív együtthatója is van, akkor elvileg bármelyik választható. A megvalósításkor ezek közül valamely előre meghatározott szabály szerint választunk. Ezt követően azt a legnagyobb értéket keressük, amennyire az x_e értéke megnövelhető a változókra vonatkozó nemnegativitási feltétel megsértése nélkül. Ehhez a 4–8. sorokban mindegyik korlátozó feltételt megvizsgálva, kiválasztjuk a legszorosabb korlátot. A kiválasztott egyenlethez tartozó bázisváltozót jelöljük x_l-lel. Ismét elképzelhető, hogy több lehetséges változó közül kell kiválasztani a bázisból kilépő változót, most is feltesszük, hogy egy előre meghatározott szabály szerint választunk. Ha egyik korlátozó feltételből sem adódik felső korlát a bázisba belépő változó értékére, az algoritmus a „nem korlátos” üzenetet adja vissza a 10. sorban, egyébként a 11. sorban a bázisba belépő és a bázisból kilépő változók szerepet cserélnek a már ismertetett PIVOTÁLÁS(N, B, A, b, c, v, l, e) eljárás meghívásával. A 12–15. sorok kiszámítják az eredeti lineáris programozási feladat $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ megoldását, a nembázis-

változók 0-t kapnak értékül, míg valamennyi x_i bázisváltozó a neki megfelelő b_i értékre lesz állítva. A 29.10. tétel szerint, ahogy később látni fogjuk, ez egy optimális megoldása a lineáris programozási feladatnak. Végül az eljárás az utolsó, 16. sorban visszaadja az eredeti feladat változóinak értékeit.

A SZIMPLEX algoritmus helyességének bizonyításához először azt látjuk be, hogy ha az eljárás kezdetekor van megengedett megoldás, és az eljárás valamikor befejeződik, akkor az vagy egy megengedett megoldást ad vissza, vagy megállapítja, hogy a feladat célfüggvénye nem korlátos. Ezután belátjuk, hogy a SZIMPLEX algoritmus befejeződik. Végül a 29.4. alfejezetben megmutatjuk, hogy a visszaadott megoldás optimális.

29.2. lemma. *Adott egy* (A, b, c) *lineáris programozási feladat. Tegyük fel, hogy a* SZIMPLEX *eljárás első sorában meghívott* SZIMPLEX-KEZDÉS *eljárás a feladatot olyan kiegyenlített alakra hozza, amelyhez megengedett bázismegoldás tartozik. Ekkor ha a* SZIMPLEX *eljárás a 16. sorban visszaad egy megoldást, az megengedett megoldása a lineáris programozási feladatnak. Ha a* SZIMPLEX *eljárás a 10. sorban a „nem korlátos” üzenettel tér vissza, akkor a célfüggvény nem korlátos.*

Bizonyítás. A következő három részből álló ciklusinvariáns állítást fogjuk használni:

A 2–11. sorokban lévő **while** ciklus minden iterációjának kezdetekor fennáll, hogy

1. a kiegyenlített alak ekvivalens a SZIMPLEX-KEZDÉS eljárás által visszaadott kiegyenlített alakkal,
2. $b_i \geq 0$, minden $i \in B$ indexre,
3. a kiegyenlített alakhoz tartozó bázismegoldás megengedett.

Teljesül: A kiegyenlített alakok ekvivalenciája triviális az első iteráció kezdetekor. Feltesszük, hogy a SZIMPLEX-KEZDÉS eljárás olyan kiegyenlített alakot ad vissza, melynek a bázismegoldása megengedett, így az invariáns állítás harmadik része teljesül. Abból, hogy a bázismegoldásban minden x_i a b_i értéket veszi fel, valamint abból, hogy a bázismegoldás megengedett, következik, hogy $b_i \geq 0$, ezzel beláttuk, hogy az invariáns második állítása is teljesül.

Megmarad: Megmutatjuk, hogy a ciklusinvariáns minden iteráció után érvényben marad, feltéve, hogy a a 10. sorban lévő **return** utasítás nem hajtódik végre. Azt az esetet, amikor a 10. sor hajtódik végre, a befejeződés tárgyalásánál fogjuk megvizsgálni.

A **while** ciklus egy iterációja egy bázis- és egy nembázis-változó szerepét cseréli fel. A végrehajtott műveletek csak egyenletrendezésből és az egyik egyenletnek a többi egyenletbe történő behelyettesítéséből állnak, így a kapott kiegyenlített alak ekvivalens a ciklus iterációja előtti kiegyenlített alakkal, ami a ciklusinvariáns miatt ekvivalens a kezdeti kiegyenlített alakkal.

Most foglalkozzunk a ciklusinvariáns második állításával. Feltesszük hogy a **while** ciklus minden iterációjának kezdetekor $b_i \geq 0$, fennáll minden $i \in B$ esetén. Megmutatjuk, hogy ezek az egyenlőtlenségek a PIVOTÁLÁS eljárás 11. sorban történő meghívása után is érvényben maradnak. Mivel a b_i értékek, valamint a B halmaz (ami a bázisváltozók indexeit tartalmazza) csak a 11. sor utasítása során változhatnak, így azt kell belátnunk, hogy a 11. sor érvényben hagyja az invariáns ezen részét. A leírásban a b_i, a_{ij} és B a

PIVOTÁLÁS eljárás hívása előtti értékeket jelölik, a $\widehat{b_i}$ pedig az eljárás által visszaadott értéket jelöli.

Elsőként vegyük észre, hogy $\widehat{b_e} \geq 0$, mivel $b_l \geq 0$ a ciklusinvariáns miatt, $a_{le} > 0$ a SZIMPLEX 5. sora miatt, és a PIVOTÁLÁS eljárás 2. sora szerint $\widehat{b_e} = b_l/a_{le}$.

A fennmaradó $i \in B - \{l\}$ indexekre pedig

$$\begin{aligned} \widehat{b_i} &= b_i - a_{ie}\widehat{b_e} && \text{(PIVOTÁLÁS 8. sora)} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(PIVOTÁLÁS 2. sora).} \end{aligned} \tag{29.79}$$

Két esetet kell megvizsgálnunk, az egyik az $a_{ie} > 0$, a másik az $a_{ie} \leq 0$. Ha $a_{ie} > 0$, akkor, mivel l-et úgy választottuk, hogy

$$b_l/a_{le} \leq b_i/a_{ie} \quad \text{minden } i \in B \text{ indexre,} \tag{29.80}$$

tehát

$$\begin{aligned} \widehat{b_i} &= b_i - a_{ie}(b_l/a_{le}) && \text{(a (29.79) egyenlőség miatt)} \\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(a (29.80) egyenlőtlenség miatt)} \\ &= b_i - b_i \\ &= 0, \end{aligned}$$

így $\widehat{b_i} \geq 0$ igaz. Ha $a_{ie} \leq 0$, akkor az a_{le}, b_i és b_l nemnegatív voltából, valamint a (29.79) egyenlőségből következik, hogy $\widehat{b_i}$ nemnegatív.

Most megmutatjuk, hogy a bázismegoldás megengedett, azaz valamennyi változó értéke nemnegatív. A nembázis-változók értékét 0-ra állítjuk, így azok nemnegatívak. Az x_i bázisváltozók értékét az alábbi egyenletből kapjuk:

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j .$$

A bázismegoldásban $\bar{x}_i = b_i$, így a ciklusinvariáns második részéből következik, hogy valamennyi $\bar{x}_i$ bázisváltozó értéke nemnegatív.

Befejeződik: A **while** ciklus kétféleképp fejeződhet be. Ha a 2. sorban megadott feltétel miatt fejeződik be, akkor az aktuális bázismegoldás megengedett, és az eljárás ezt a megoldást adja vissza a 16. sorban. A másik esetben a 10. sorban fejeződik be, ekkor a „nem korlátos” üzenettel tér vissza. Ilyenkor a 4–7. sorok **for** ciklusán belül az 5. sor feltétele egyetlen bázisbeli indexre sem teljesül, tehát $a_{ie} \leq 0$ minden $i \in B$ indexre. Tartozzon az x bázismegoldás annak az iterációs lépésnek a kezdeti kiegyenlített alakjához, amely iterációs lépés a „nem korlátos” üzenettel tér vissza. Tekintsük a következő $\bar{x}$ megoldást:

$$\bar{x}_i = \begin{cases} \infty, & \text{ha } i = e, \\ 0, & \text{ha } i \in N - \{e\}, \\ b_i - \sum_{j \in N} a_{ij}\bar{x}_j, & \text{ha } i \in B. \end{cases}$$

Most megmutatjuk, hogy ez a megoldás megengedett, azaz minden változó értéke nemnegatív. A nembázis-változók értéke $\bar{x}_e$ kivételével 0, $\bar{x}_e$ pedig pozitív; így valamennyi nembázis-változó nemnegatív. Tudjuk, hogy az $\bar{x}_i$ bázisváltozók értéke

$$\begin{aligned}\overline{x_i} &= b_i - \sum_{j \in N} a_{ij}\overline{x_j} \\ &= b_i - a_{ie}\overline{x_e}.\end{aligned}$$

A ciklusinvariánsból következik, hogy $b_i \geq 0$, valamint az $a_{ie} \leq 0$ és $\overline{x_e} = \infty > 0$, amiből $\overline{x_i} \geq 0$.

Most megmutatjuk, hogy az $\overline{x}$ bázismegoldáshoz tartozó célfüggvényérték nem korlátos. A célfüggvényérték

$$\begin{aligned}z &= v + \sum_{j \in N} c_j\overline{x_j} \\ &= v + c_e\overline{x_e}.\end{aligned}$$

Minthogy $c_e > 0$ (a 3. sor miatt) és $\overline{x_e} = \infty$, a célfüggvény értéke ∞, tehát a lineáris programozási feladat célfüggvénye nem korlátos. ■

A Szimplex módszer minden iterációs menetben az A, b, c és v értékeit az N és B halmazoktól függően karbantartja. Bár a szimplex módszer hatékony megvalósításához lényeges, hogy az A, b, c és v adatok minden lépésben rendelkezésre álljanak, de elvben nincs szükség ezek tényleges tárolására. Más szavakkal a kiegyenlített alakot egyértelműen meghatározzák a bázis- és nembázis-változók. Mielőtt ezt belátjuk, egy hasznos algebrai lemmát bizonyítunk.

29.3. lemma. *Legyen az I indexek egy halmaza. Minden $i \in I$ esetén legyenek az α_i és β_i valós számok, x_i pedig valós értékű változó. Legyen γ egy tetszőleges valós szám. Ha x_i bármely értékére fennáll, hogy*

$$\sum_{i \in I} \alpha_i x_i = \gamma + \sum_{i \in I} \beta_i x_i, \tag{29.81}$$

akkor $\alpha_i = \beta_i$ minden $i \in I$ indexre, és $\gamma = 0$.

Bizonyítás. Mivel a (29.81) egyenlőség x_i bármely értékére fennáll, így tetszőleges önkényesen választott értékbeállításból következtethetünk az α, β és γ értékekre. Legyen $x_i = 0$ minden $i \in I$ indexre, ekkor azt kapjuk, hogy $\gamma = 0$. Most válasszunk egy tetszőleges $i \in I$ indexet, és legyen $x_i = 1$, valamint $x_k = 0$, minden $k \neq i$ esetén. Ebből azt kapjuk, hogy $\alpha_i = \beta_i$. Mivel i tetszőlegesen választott index volt, $\alpha_i = \beta_i$ minden $i \in I$ esetén fennáll. ■

Most megmutatjuk, hogy egy lineáris programozási feladat kiegyenlített alakját egyértelműen meghatározza a bázishoz tartozó változók halmaza.

29.4. lemma. *Legyen az (A, b, c) egy szabályos alakú lineáris programozási feladat. Ha adott a bázisváltozókat meghatározó B halmaz, akkor az ehhez tartozó kiegyenlített alak egyértelmű.*

Bizonyítás. Tegyük fel indirekt módon, hogy ugyanahhoz a bázisváltozókat meghatározó B halmazhoz két különböző kiegyenlített alak is tartozik. Szükségképp az ezen kiegyenlített alakokhoz tartozó nembázis-változókat meghatározó $N = \{1, 2, \ldots, n + m\} - B$ halmazok

egyenlők egymással. Legyen az első kiegyenlített alak:

$$z = v + \sum_{j \in N} c_j x_j \tag{29.82}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{minden } i \in B \text{ indexre,} \tag{29.83}$$

és a második:

$$z = v' + \sum_{j \in N} c'_j x_j \tag{29.84}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{minden } i \in B \text{ indexre.} \tag{29.85}$$

Vonjuk ki a (29.85) egyenletrendszerből a (29.83) egyenletrendszert, ekkor a következő egyenletrendszerhez jutunk:

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \quad \text{minden } i \in B \text{ indexre,}$$

ami ekvivalens a következővel:

$$\sum_{j \in N} a_{ij} x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij} x_j \quad \text{minden } i \in B \text{ indexre.}$$

Most minden $i \in B$ indexre alkalmazzuk a 29.3. lemmát, az $\alpha_i = a_{ij}$, $\beta_i = a'_{ij}$ és $\gamma = b_i - b'_i$ helyettesítésekkel. Tudjuk, hogy $\alpha_i = \beta_i$, tehát $a_{ij} = a'_{ij}$ minden $j \in N$ esetén, és mivel $\gamma = 0$, ezért $b_i = b'_i$. Így a két kiegyenlített alakban az A és A', valamint a b és b' azonosak. Hasonló meggondolást követve (a bizonyítást a 29.3-1. gyakorlatra hagyjuk) belátható az is, hogy $c = c'$ és $v = v'$ fennállnak, tehát a két kiegyenlített alak valóban megegyezik. ■

Hátravan még annak bizonyítása, hogy a SZIMPLEX eljárás befejeződik, és amikor befejeződik, a kimenetként adott megoldás optimális. A 29.4. alfejezetet szánjuk az optimalitás belátásának, most a befejeződést vitatjuk meg.

A befejezés

Az alfejezet elején bemutatott példában a szimplex módszer minden iterációs menetében a bázismegoldáshoz tartozó célfüggvényérték megnövekedett. Bebizonyítható (lásd a 29.3-2. gyakorlatot), hogy a SZIMPLEX eljárás egyetlen iterációs menetében sem csökkenhet a bázismegoldáshoz tartozó célfüggvényérték. Szerencsétlen esetben előfordulhat, hogy az iterációs menet nem változtat a célfüggvény értékén. Ezt ***degenerációnak*** nevezzük, és ezt az esetet most részletesebben meg fogjuk vizsgálni.

A célfüggvényérték a PIVOTÁLÁS eljárás 13. sorában, a $\widehat{v} \leftarrow v + c_e \widehat{b_e}$ utasításban változik meg. Mivel a SZIMPLEX eljárás csak olyan esetben hívja meg a PIVOTÁLÁS eljárást, amikor $c_e > 0$, az, hogy a célfüggvényérték változatlan marad (vagyis $\widehat{v} = v$) csak úgy fordulhat elő, ha $\widehat{b_e}$ értéke 0. A $\widehat{b_e}$ a PIVOTÁLÁS 2. sorában $\widehat{b_e} \leftarrow b_l / a_{le}$ szerint kap értéket. Tudjuk, hogy a PIVOTÁLÁS csak $a_{le} \neq 0$ esetén hívódik meg, tehát amikor a $\widehat{b_e} = 0$, és ezért a célfüggvényérték változatlan marad, akkor biztos, hogy $b_l = 0$.

Valóban előfordulhat ez az eset, nézzük például a következő lineáris programot:

$$\begin{array}{lcrcrcrcl} z & = & & & x_1 & + & x_2 & + & x_3 \\ x_4 & = & 8 & - & x_1 & - & x_2 & & \\ x_5 & = & & & & & x_2 & - & x_3. \end{array}$$

Tegyük fel, hogy az x_1 változót választjuk a bázisba belépő, és az x_4 változót a bázisból kilépő változónak. A pivotálás utáni alak a következő:

$$\begin{array}{lcrcrcrcl} z & = & 8 & & & + & x_3 & - & x_4 \\ x_1 & = & 8 & - & x_2 & & & - & x_4 \\ x_5 & = & & & x_2 & - & x_3. & & \end{array}$$

A pivotáláshoz most csak az x_3-at választhatjuk bázisba belépő, és az x_5-öt bázisból kilépő változónak. Mivel $b_5 = 0$, az aktuális célfüggvényérték, ami 8, változatlan marad a pivotálás után:

$$\begin{array}{lcrcrcrcl} z & = & 8 & + & x_2 & - & x_4 & - & x_5 \\ x_1 & = & 8 & - & x_2 & - & x_4 & & \\ x_3 & = & & & x_2 & & & - & x_5. \end{array}$$

A célfüggvényérték nem változott, viszont a feladat alakja igen. Szerencsére, ha újra pivotálunk, és x_2 lesz a bázisba belépő, és x_1 lesz a bázisból kilépő változó, a célfüggvényérték növekedni fog, és folytatódhat a szimplex módszer.

Most megmutatjuk, hogy egyedül a degeneráció az, ami a szimplex módszert visszatarthatja a befejeződéstől. Emlékezzünk vissza, hogy feltevésünk szerint a SZIMPLEX eljárás valamilyen előre meghatározott szabály szerint választja ki az e és l indexeket a 3. és 8. sorban. Azt mondjuk, hogy a szimplex módszer ***köröz,*** ha két különböző iterációs menetben a kapott kiegyenlített alakok megegyeznek. Mivel a SZIMPLEX eljárás determinisztikus, ilyenkor a kiegyenlített alakok ugyanazon sorozatán fog körözni a végtelenségig.

29.5. lemma. *Ha a* SZIMPLEX *eljárás legfeljebb* $\binom{n+m}{m}$ *iterációs menet alatt nem fejeződik be, akkor köröz.*

Bizonyítás. A 29.4. lemma szerint a bázisváltozókat meghatározó B halmaz egyértelműen meghatároz egy kiegyenlített alakot. Mivel a változók száma $n + m$, továbbá $|B| = m$, így legfeljebb $\binom{n+m}{m}$-féleképpen választhatjuk ki B halmaz elemeit. Emiatt csak legfeljebb $\binom{n+m}{m}$-féle különböző kiegyenlített alak lehetséges. Ebből adódik, ha a SZIMPLEX eljárás több mint $\binom{n+m}{m}$ iterációs menetet végez, akkor szükségképpen köröz. ■

Bár a szimplex módszer körözése elméletileg lehetséges, valójában rendkívül ritka. Elkerülhető, ha körültekintően választjuk ki a bázisba belépő és a bázisból kilépő változókat. Egyik lehetőség, ha kis mértékben megváltoztatjuk – perturbáljuk – a bemenetet, amivel kivédjük azt, hogy két különböző bázismegoldáshoz ugyanaz a célfüggvényérték tartozhasson. A másik lehetőség, hogy amikor több lehetséges bázisba belépő vagy bázisból kilépő változó van, a választás egy lexikografikus rendezés alapján történik. A harmadik, amikor a választás a ***Bland-szabályt*** alkalmazva történik, mely szerint mindig a legkisebb indexet kell választani. Eltekintünk annak bizonyításától, hogy ezekkel a módszerekkel elkerülhető a körözés.

29.6. lemma. *Ha a* SZIMPLEX *eljárás 3. és 8. sorában, amikor több lehetséges változó közül kell választani, mindig a legkisebb indexűt választjuk, a* SZIMPLEX *eljárás véges lépésben befejeződik.*

Ezt az alfejezetet a következő lemmában foglaljuk össze.

29.7. lemma. *Feltéve, hogy a* SZIMPLEX-KEZDÉS *egy olyan kiegyenlített alakot ad vissza, melynek bázismegoldása megengedett, a* SZIMPLEX *eljárás vagy azt az üzenetet küldi, hogy a célfüggvény nem korlátos, vagy legfeljebb* $\binom{n+m}{m}$ *iterációs menetben befejeződik, visszaadva egy megengedett megoldást.*

Bizonyítás. A 29.2. és a 29.6. lemmákból következik, hogy a SZIMPLEX-KEZD ÉS egy olyan kiegyenlített alakot ad vissza, melynek bázismegoldása megengedett, a SZIMPLEX eljárás vagy azt az üzenetet küldi, hogy a célfüggvény nem korlátos, vagy egy megengedett megoldással fejeződik be. A 29.5. lemma ellentett állítása épp az, hogy ha a SZIMPLEX eljárás befejeződik egy megengedett megoldással, akkor az legfeljebb $\binom{n+m}{m}$ iterációs meneten belül történik. ■

Gyakorlatok

29.3-1. Fejezzük be a 29.4. lemma bizonyítását, igazolva, hogy $c = c'$ és $v = v'$.

29.3-2. Mutassuk meg, hogy a SZIMPLEX eljárás 11. sorában történő PIVOTÁLÁS meghívása soha nem csökkentheti v értékét.

29.3-3. Tegyük fel, hogy az (A, b, c) szabályos alakú lineáris programozási feladatot átalakítjuk kiegyenlített alakra. Mutassuk meg, hogy a bázismegoldás akkor és csak akkor megengedett, ha $b_i \geq 0$ minden $i = 1, 2, \ldots, m$ esetén.

29.3-4. Oldjuk meg a következő lineáris programozási feladatot a SZIMPLEX eljárással:

$$
\begin{aligned}
&\text{maximalizálandó} && 18x_1 + 12.5x_2 && \\
&\text{feltéve, hogy} && && \\
& && x_1 + x_2 && \leq 20 \\
& && x_1 && \leq 12 \\
& && x_2 && \leq 16 \\
& && x_1, x_2 && \geq 0 .
\end{aligned}
$$

29.3-5. Oldjuk meg a következő lineáris programozási feladatot a SZIMPLEX eljárással:

$$
\begin{aligned}
&\text{maximalizálandó} && -5x_1 - 3x_2 && \\
&\text{feltéve, hogy} && && \\
& && x_1 - x_2 && \leq 1 \\
& && 2x_1 + x_2 && \leq 2 \\
& && x_1, x_2 && \geq 0 .
\end{aligned}
$$

29.3-6. Oldjuk meg a következő lineáris programozási feladatot a SZIMPLEX eljárással:

$$\begin{aligned}
&\text{minimalizálandó} && x_1 + x_2 + x_3 \\
&\text{feltéve, hogy} \\
&&& 2x_1 + 7.5x_2 + 3x_3 \geq 10000 \\
&&& 20x_1 + 5x_2 + 10x_3 \geq 30000 \\
&&& x_1, x_2, x_3 \geq 0 .
\end{aligned}$$

29.4. Dualitás

Az előzőekben beláttuk, hogy a SZIMPLEX eljárás – bizonyos feltételek mellett – befejező dik. Az még bizonyításra vár, hogy az eljárás az adott lineáris programozási feladat optimális megoldását találja meg. Ennek bizonyításához bevezetjük a ***lineáris programozási dualitás*** fogalmát.

A dualitás nagyon fontos fogalom. Egy optimalizálási feladat esetén a duál probléma megtalálása csaknem mindig együtt jár egy polinom idejű algoritmus megtalálásával. Most a dualitást annak megmutatásához használjuk, hogy a szimplex módszer megállásakor kapott megoldás optimális.

Nézzünk egy példát: legyen adott egy maximális folyam probléma, és tegyük fel, hogy találtunk egy olyan f folyamot, aminek a nagysága $|f|$. Hogyan látható be, hogy f maximális folyam? A 26.7. maximális folyam-minimális vágás tétel alapján, ha találunk egy olyan vágást, amelynek szintén $|f|$ a nagysága, akkor biztos, hogy f maximális folyam. Ez jól példázza a dualitást: adott egy maximalizálási feladat, definiálunk egy hozzá kapcsolódó minimalizálási feladatot úgy, hogy a két problémának ugyanaz legyen az optimális célfüggvényértéke.

Most nézzük meg, hogy egy adott lineáris programozási feladatnak, amiben a célfüggvény maximalizálása a cél, hogyan készíthetjük el a ***duál*** párját, amelyben a célfüggvény minimalizálása a feladat és amelynek optimuma az eredeti feladatéval azonos. A duál feladat bemutatása közben az eredeti feladatot ***primál*** lineáris programozási feladatnak fogjuk nevezni.

A (29.16)–(29.18) egyenletek szerinti szabályos alakú lineáris programozási feladat duál feladata a következő:

$$\text{minimalizálandó} \quad \sum_{i=1}^{m} b_i y_i \tag{29.86}$$

feltéve, hogy

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad (j = 1, 2, \ldots, n), \tag{29.87}$$

$$y_i \geq 0 \quad (i = 1, 2, \ldots, m). \tag{29.88}$$

A fenti duál feladatot a következő átalakításokkal kaptuk: a célfüggvény maximuma helyett a minimumát keressük, a jobb oldal és a célfüggvény szerepet cseréltek, és a kisebb vagy egyenlő jelek helyett nagyobb vagy egyenlő jelek állnak. A primál feladatban szereplő m számú korlátozó feltétel mindegyikéhez tartozik egy y_i változó a duál feladatban, és

fordítva a duál feladatban szereplő n számú korlátozó feltétel mindegyikéhez tartozik egy x_j változó a primál feladatban. Tekintsük például a (29.56)–(29.60) lineáris programozási feladatot. A duális feladat a következő:

$$\text{minimalizálandó} \quad 30y_1 + 24y_2 + 36y_3 \tag{29.89}$$

feltéve, hogy

$$y_1 + 2y_2 + 4y_3 \geq 3 \tag{29.90}$$

$$y_1 + 2y_2 + y_3 \geq 1 \tag{29.91}$$

$$3y_1 + 5y_2 + 2y_3 \geq 2 \tag{29.92}$$

$$y_1, y_2, y_3 \geq 0. \tag{29.93}$$

A 29.10. tételben bebizonyítjuk majd, hogy a duál lineáris programozási feladat optimális célfüggvényértéke egyenlő a primál feladat optimális célfüggvényértékével. Azt is belátjuk majd, hogy a szimplex módszer egyszerre megoldja a primál és duál feladatot is, amiből majd a megoldás optimalitása adódik.

A ***gyenge dualitási lemma*** bizonyításával kezdjük, mely szerint a primál feladat tetszőleges megengedett megoldásához tartozó célfüggvényérték nem nagyobb, mint a duál feladat tetszőleges megoldásához tartozó célfüggvényérték.

29.8. lemma (gyenge dualitási lemma). *Legyen az $\bar{x}$ megengedett megoldása a (29.16)–(29.18) primál lineáris programozási feladatnak, és legyen $\bar{y}$ tetszőleges megoldása a (29.86)–(29.88) duál lineáris programozási feladatnak. Ekkor fennáll a következő:*

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i.$$

Bizonyítás. Tudjuk, hogy

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \quad \text{((29.87) szerint)} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\
&\leq \sum_{i=1}^{m} b_i \bar{y}_i \qquad \text{((29.17) szerint).}
\end{aligned}$$

■

29.9. következmény. *Legyen az $\bar{x}$ megengedett megoldása az (A, b, c) primál lineáris programozási feladatnak, és legyen $\bar{y}$ megengedett megoldása a hozzá tartozó duál lineáris programozási feladatnak. Ha*

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i,$$

akkor $\bar{x}$ és $\bar{y}$ optimális megoldásai a primál, illetve a duál lineáris programozási feladatnak.

Bizonyítás. A 29.8. lemma szerint a primál feladat bármely megengedett megoldásához tartozó célfüggvényérték nem lehet nagyobb, mint a duál feladat megengedett megoldásához

tartozó célfüggvényérték. Mivel a primál feladat maximalizálási, a duál feladat pedig minimalizálási probléma, így ha az $\bar{x}$ és $\bar{y}$ megoldásokhoz tartozó célfüggvényérték megegyezik, egyiken sem lehet tovább javítani. ■

Mielőtt bebizonyítjuk, hogy mindig létezik a duál feladatnak olyan megoldása, melynek a célfüggvényértéke megegyezik a primál feladat optimumával, megmutatjuk, hogyan találhatunk egy ilyen megoldást. Amikor a (29.56)–(29.60) lineáris programozási feladaton futtattuk a szimplex módszert, a végső iteráció a (29.75)–(29.78) kiegyenlített alakhoz vezetett, $B = \{1, 2, 4\}$ és $N = \{3, 5, 6\}$ halmazokkal. Ahogy a következőkben be fogjuk bizonyítani, a végső kiegyenlített alakhoz tartozó bázismegoldás optimális megoldása a lineáris programozási feladatnak; tehát a (29.56)–(29.60) feladat optimális megoldása az $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (8, 4, 0)$ a $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$ célfüggvényértékkel. Ebből megkaphatjuk – később ezt bizonyítani is fogjuk – a duál feladat optimális megoldását: ha a primál feladat célfüggvényének együtthatóit (-1)-gyel szorozzuk, megkapjuk a duál feladat változóinak értékét. Pontosabban megfogalmazva, tegyük fel, hogy a primál feladat legutolsó kiegyenlített alakja a következő:

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ x_i &= b'_i - \sum_{j \in N} a'_{ij} x_j \qquad \text{minden } i \in B \text{ indexre.} \end{aligned}$$

Ekkor a duál feladat optimális megoldása:

$$\bar{y}_i = \begin{cases} -c'_{n+i}, & \text{ha } (n+i) \in \mathbf{N}, \\ 0 & \text{egyébként.} \end{cases} \tag{29.94}$$

Ebből az adódik, hogy a (29.89)–(29.93) által meghatározott duál lineáris programozási feladatnak egy optimális megoldása az $\bar{y}_1 = 0$ (mivel $n + 1 = 4 \in B$), $\bar{y}_2 = -c'_5 = 1/6$, és $\bar{y}_3 = -c'_6 = 2/3$. Kiszámítva a (29.89) célfüggvényértéket, az eredmény $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$, ami megerősíti, hogy a primál feladat célfüggvényértéke megegyezik a duál feladat célfüggvényértékével. Ez a számítás és a 29.8. lemma együttesen bizonyítják, hogy a primál lineáris programozási feladat optimális célfüggvényértéke 28. Most általános esetre is bebizonyítjuk, hogy ezzel a módszerrel megkaphatjuk a duál feladat optimális megoldását, valamint igazolhatjuk, hogy a primál feladat megoldása optimális.

29.10. tétel (a lineáris programozási feladat dualitása). *Tegyük fel, hogy a* Szimplex *eljárás az* $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots \bar{x}_n)$ *értékeket adja vissza az* (A, b, c) *primál lineáris programozási feladat megoldásaként. Jelölje* N *és* B *a végső kiegyenlített alak bázis- és nembázis-változóinak halmazát,* c' *a végső kiegyenlített alak célfüggvényének együtthatóit, és* $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ *legyenek a (29.94) szerinti értékek. Ekkor* $\bar{x}$ *optimális megoldása a primál lineáris programozási feladatnak,* $\bar{y}$ *optimális megoldása a duál lineáris programozási feladatnak, és*

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i. \tag{29.95}$$

Bizonyítás. A 29.9. következmény szerint, ha találunk olyan $\bar{x}$ és $\bar{y}$ megengedett megoldásokat, melyek kielégítik a (29.95) egyenletet, akkor $\bar{x}$ és $\bar{y}$ optimális megoldásai a primál,

illetve a duál feladatnak. Megmutatjuk, hogy ha $\bar{x}$ és $\bar{y}$ megoldások megfelelnek a tételben kimondott feltételeknek, akkor kielégítik a (29.95) egyenletet.

Tegyük fel, hogy lefuttatjuk a Szimplex eljárást egy a (29.16)–(29.18) szerint meg adott primál lineáris programozási feladatra. Az algoritmus kiegyenlített alakok sorozatán haladva egy végső kiegyenlített alakot elérve befejeződik, amelyhez az alábbi célfüggvény tartozik:

$$z = v' + \sum_{j \in N} c'_j x_j. \tag{29.96}$$

Mivel a Szimplex eljárás egy megoldást adva befejeződött, a 2. sor feltétele szerint

$$c'_j \leq 0 \quad \text{minden } j \in N \text{ indexre.} \tag{29.97}$$

Definiáljuk a c' vektornak a bázis-indexekhez tartozó komponenseit az alábbiak szerint:

$$c'_j = 0 \quad \text{minden } j \in B \text{ indexre.} \tag{29.98}$$

Akkor a (29.96) egyenletet az alábbiak szerint átalakíthatjuk:

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j \qquad (\text{mivel } c'_j = 0, \text{ ha } j \in B) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \qquad (\text{mivel } N \cup B = \{1, 2, \ldots, n+m\}). \end{aligned} \tag{29.99}$$

A végső kiegyenlített alakhoz tartozó $\bar{x}$ bázismegoldásban $\bar{x}_j = 0$ minden $j \in N$ indexre, és $z = v'$. Mivel a kiegyenlített alakok ekvivalensek, ha kiszámítjuk az eredeti célfüggvényt az $\bar{x}$ helyen, ugyanezt az értéket kell kapnunk, azaz

$$\sum_{j=1}^{n} c_j \bar{x}_j = v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j \tag{29.100}$$

$$\begin{aligned} &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot 0) + \sum_{j \in B} (0 \cdot \bar{x}_j) \\ &= v'. \end{aligned} \tag{29.101}$$

Most belátjuk, hogy a (29.94) egyenlőség szerint definiált $\bar{y}$ megengedett megoldása a duál lineáris programozási feladatnak, és a hozzá tartozó célfüggvényérték, ami $\sum_{i=1}^{m} b_i \bar{y}_i$, egyenlő $\sum_{j=1}^{n} c_j \bar{x}_j$-vel. A (29.100) egyenletből adódik, hogy ha a feladat első és utolsó kiegyenlített alakjának célfüggvényét kiszámítjuk az $\bar{x}$ helyen, a kapott értékek egyenlők. Általánosabban szólva a kiegyenlített alakok ekvivalenciája azt jelenti, hogy *tetszőleges* $x = (x_1, x_2, \ldots, x_n)$ értékekre igaz, hogy

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j.$$

Ebből azt kapjuk, hogy az x vektor tetszőleges értékeire fennáll az alábbi összefüggés:

$$\begin{aligned}
\sum_{j=1}^{n} c_j x_j &= v' + \sum_{j=1}^{n+m} c'_j x_j \\
&= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{j=n+1}^{n+m} c'_j x_j \\
&= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} c'_{n+i} x_{n+i} \\
&= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\overline{y}_i)\, x_{n+i} && \text{((29.94) szerint)} \\
&= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\overline{y}_i) \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right) && \text{((29.32) szerint)} \\
&= v' + \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \overline{y}_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \left(a_{ij} x_j \right) \overline{y}_i \\
&= v' + \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \overline{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} \left(a_{ij} \overline{y}_i \right) x_j \\
&= \left(v' - \sum_{i=1}^{m} b_i \overline{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \overline{y}_i \right) x_j,
\end{aligned}$$

tehát

$$\sum_{j=1}^{n} c_j x_j = \left(v' - \sum_{i=1}^{m} b_i \overline{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \overline{y}_i \right) x_j. \tag{29.102}$$

A 29.3. lemmát alkalmazva a (29.102) egyenletre a következőket kapjuk:

$$v' - \sum_{i=1}^{m} b_i \overline{y}_i = 0, \tag{29.103}$$

$$c'_j + \sum_{i=1}^{m} a_{ij} \overline{y}_i = c_j \quad (j = 1, 2, \ldots, n). \tag{29.104}$$

A (29.103) egyenletet átrendezve kapjuk, hogy $\sum_{i=1}^{m} b_i \overline{y}_i = v'$, és ezért a duál feladat célfüggvényértéke $\left(\sum_{i=1}^{m} b_i \overline{y}_i \right)$ egyenlő a primál feladat célfüggvényértékével (v'). Már csak azt kell bebizonyítanunk, hogy az $\overline{y}$ megengedett megoldása a duál feladatnak. (29.97) és (29.98) szerint $c'_j \le 0$ minden $j = 1, 2, \ldots, n+m$ indexre. Így bármely $i = 1, 2, \ldots, m$ esetén a (29.104) egyenletből következik, hogy

$$\begin{aligned}
c_j &= c'_j + \sum_{i=1}^{m} a_{ij} \overline{y}_i \\
&\le \sum_{i=1}^{m} a_{ij} \overline{y}_i,
\end{aligned}$$

ami épp megfelel a duál feladat (29.87) szerint adott korlátozó feltételeinek. Végül, mivel $c'_j \leq 0$ minden $j \in N \cup B$ indexre fennáll, ha a (29.94) szerint meghatározzuk $\bar{y}$ értékeit, az $\bar{y}_i \geq 0$ minden értékre teljesül, tehát az $\bar{y}$ kielégíti a nemnegativitási korlátozó feltételt is. ■

Megmutattuk, hogy ha a SZIMPLEX-KEZDÉS eljárás megengedett megoldást talál, és a SZIMPLEX eljárás nem a „nem korlátos” üzenettel áll meg, akkor a SZIMPLEX eljárás által talált megoldás valóban optimális. Egyúttal megmutattuk, hogyan számolható ki a duál feladatnak egy optimális megoldása.

Gyakorlatok

29.4-1. Adjuk meg a 29.3-4. gyakorlat lineáris programozási feladatának duál feladatát.

29.4-2. Tegyük fel, hogy van egy nem szabályos alakban megadott lineáris programozási feladatunk. A hozzá tartozó duál feladatot meghatározhatjuk úgy, hogy els ő lépésben szabályos alakúra hozzuk a feladatot. Kényelmesebb lenne, ha rögtön a duál feladatot tudnánk előállítani. Magyarázzuk el egy tetsz őleges, konkrét lineáris programozási feladaton, hogyan lehetne rögtön a duál feladatot felírni.

29.4-3. Írjuk fel a (29.47)–(29.50) által adott maximális folyam probléma duál feladatát. Magyarázzuk el, hogy az így kapott probléma hogyan értelmezhet ő, mint minimális vágási feladat.

29.4-4. Írjuk fel a (29.51)–(29.55) által adott minimális költségű folyam probléma duál feladatát. Értelmezzük, mit jelent a kapott feladat a gráfok és folyamok nyelvén.

29.4-5. Mutassuk meg, hogy a duál feladat duál feladata azonos a primál feladattal.

29.4-6. A 26. fejezet melyik eredményét tekinthetjük úgy, mint a maximális folyam probléma gyenge duálisa?

A kezdeti megengedett bázismegoldás

Ebben az alfejezetben megvizsgáljuk, hogyan dönthetjük el egy lineáris programozási feladatról, hogy van-e megengedett megoldása, és ha van, hogyan állítható el ő egy olyan kiegyenlített alak, amelyiknek a bázismegoldása megengedett. Végül kimondjuk és bebizonyítjuk a lineáris programozás alaptételét, mely szerint a SZIMPLEX eljárás minden esetben helyes megoldást ad.

29.4.1. A kezdeti megengedett bázismegoldás megkeresése

A 29.3. alfejezetben feltettük, hogy van egy SZIMPLEX-KEZDÉS nevű eljárásunk, mely eldönti, hogy a lineáris programozási feladatnak van-e megengedett megoldása, és ha van, el őállítja a feladat egy olyan kiegyenlített alakját, amelyhez tartozó bázismegoldás megengedett. Ezt az eljárást fogjuk most leírni.

Elképzelhető, hogy a lineáris programozási feladatnak van megengedett megoldása, de a nyilvánvalóan adódó kezdeti bázismegoldás nem megengedett. Tekintsük például a következő lineáris programozási feladatot:

maximalizálandó $2x_1 - x_2$ (29.105)

feltéve, hogy

$$2x_1 - x_2 \leq 2 \quad (29.106)$$

$$x_1 - 5x_2 \leq -4 \quad (29.107)$$

$$x_1, x_2 \geq 0. \quad (29.108)$$

Ha ezt a lineáris programozási feladatot átalakítanánk kiegyenlített alakra, a bázismegoldás $x_1 = 0$ és $x_2 = 0$ lenne. Ez a megoldás sérti a (29.107) korlátozó feltételt, így ez nem megengedett megoldás. Tehát a SZIMPLEX-KEZDÉS eljárás nem térhet vissza a nyilvánvalóan adódó kiegyenlített alakkal. Megvizsgálva a feladatot, az sem látható tisztán, egyáltalán létezik-e megengedett megoldása. Ezt eldöntendő, egy ***lineáris programozási segédfeladatot*** írhatunk fel. Ennek a segédfeladatnak könnyen megtalálhatjuk azt a kiegyenlített alakját, amelyhez megengedett bázismegoldás tartozik. Sőt a segédfeladat megoldása eldönti, hogy létezik-e az eredeti lineáris programozási feladatnak megengedett megoldása, és ha igen, egy olyan megengedett megoldást fog szolgáltatni, amely alkalmas lesz a SZIMPLEX eljárás kezdeti beállítására.

29.11. lemma. *Legyen L egy a (29.16)–(29.18) képletek szerint megadott szabályos alakú lineáris programozási feladat. Legyen L_s a következő $n+1$ változójú lineáris programozási feladat:*

maximalizálandó $-x_0$ (29.109)

feltéve, hogy

$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i \quad (i = 1, 2, \ldots, m), \quad (29.110)$$

$$x_j \geq 0 \quad (j = 0, 1, \ldots, n). \quad (29.111)$$

Ekkor az L lineáris programozási feladatnak akkor és csak akkor van megengedett megoldása, ha az L_s feladat optimuma 0.

Bizonyítás. Tegyük fel, hogy az L feladatnak van megengedett megoldása, legyen ez a $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$. Ekkor az $\bar{x}_0 = 0$ és az $\bar{x}$ egyesítésével kapott megoldás megengedett megoldása az L_s feladatnak, a 0 célfüggvényértékkel. Mivel az $x_0 \geq 0$ korlátozó feltétele az L_s feladatnak, és a $-x_0$ célfüggvényt maximalizáljuk, ez optimális megoldása az L_s-nek.

Visszafelé, tegyük fel, hogy az L_s optimális célfüggvényértéke 0. Ekkor $\bar{x}_0 = 0$, az $\bar{x}$ megoldás többi változója pedig kielégíti az L feladat korlátozó feltételeit. ■

Most leírjuk azt a módszert, amivel az L szabályos alakú lineáris programozási feladat egy kezdeti megengedett bázismegoldása megtalálható.

Az 1–3. sorokban az L lineáris programozási feladatnak az $N = \{1, 2, \ldots, n\}$ és a $B = \{n+1, n+2, \ldots, n+m\}$ indexhalmazok által meghatározott kezdeti kiegyenlített alakjához tartozó bázismegoldást vizsgáljuk, ami a következő: $\bar{x}_i = b_i$ minden $i \in B$ indexre és $\bar{x}_j = 0$ minden $j \in N$ esetén. (A kiegyenlített alak előállítása nem jelent külön munkát, hiszen az A, a b és a c értékek ugyanazok a kiegyenlített és a szabályos alakban.) Ha ez a bázismegoldás megengedett – ami azt jelenti, hogy $\bar{x}_i \geq 0$ minden $i \in N \cup B$ esetén –, akkor az eljárás visszaadja a kiegyenlített alakot. Ha ez nem teljesül, akkor a 4. sorban előállítjuk

a 29.11. lemma szerinti L_s lineáris programozási segédfeladatot. A k index a b vektor legnegatívabb elemének indexe. Az l index a b vektor legnegatívabb elemének a kiegyenlített forma szerinti sorcímkéje. A megfelelő index-átalakítást valósítja meg a 6. sor. Mivel az L feladat kezdeti bázismegoldása nem megengedett, az L_s feladat kiegyenlített alakjához tartozó megoldás sem lesz megengedett. Ezért a 8. sorban meghívjuk egyszer a PIVOTÁLÁS eljárást, az x_0 bázisba belépő, és az x_l bázisból kilépő változókkal. Kicsit később be fogjuk látni, hogy az a megoldás, amit a PIVOTÁLÁS ezen meghívásával kapunk, megengedett megoldás lesz. Ezután, ha már olyan kiegyenlített alakunk van, amelyhez megengedett bázismegoldás tartozik, a PIVOTÁLÁS eljárást addig hívjuk ciklusban a 10. sorban, amíg meg nem oldja a lineáris programozási segédfeladatot. Ahogy a 11. sorban elvégzett vizsgálat is mutatja, ha az L_s feladatnak találunk olyan optimális megoldását, amelyben a célfüggvény értéke 0, akkor elkészíthetjük az L feladat egy olyan kiegyenlített alakját, amihez tartozó bázismegoldás megengedett (12. sor). Ehhez valamennyi korlátozó feltételből töröljük az x_0 kifejezést, és visszaállítjuk az L eredeti célfüggvényét. Az eredeti célfüggvényben bázis- és nembázis-változók is szerepelhetnek. Helyettesítsük a célfüggvényben mindegyik bázisváltozót a hozzá tartozó korlátozó feltétel jobb oldalával. A másik esetben, ha a 11. sorban a vizsgálatnál kiderül, hogy az L feladatnak nincs megengedett megoldása, akkor az eljárás ezt az információt adja vissza a 13. sorban.

SZIMPLEX-KEZDÉS(A, b, c)

1 legyen k a legkisebb b_i érték indexe
2 **if** $b_k \geq 0$ ▷ A kezdeti bázismegoldás megengedett?
3 **then return**$(\{1, 2, \ldots, n\}, \{n+1, n+2, \ldots, n+m\}, A, b, c, 0)$
4 írjuk fel az L_s feladatot: valamennyi egyenlet bal oldalához adjunk hozzá $-x_0$-át, és $-x_0$ legyen a feladat célfüggvénye
5 legyen (N, B, A, b, c, v) az így kapott L_s feladat kiegyenlített alakja
6 $l \leftarrow n + k$
7 ▷ L_s feladatnak $n + 1$ nembázis és m bázisváltozója van.
8 $(N, B, A, b, c, v) \leftarrow$ PIVOTÁLÁS$(N, B, A, b, c, v, l, 0)$
9 ▷ A bázismegoldás most már megengedett megoldása az L_s feladatnak.
10 iteráljunk a SZIMPLEX eljárás 2–11. sorának **while** ciklusával, amíg az L_s feladat optimális megoldását megtaláljuk
11 **if** a bázismegoldásban $\bar{x}_0 = 0$
12 **then return** az utolsó lépéshez tartozó kiegyenlített alak, eltávolítva az x_0 változót és visszaállítva az eredeti célfüggvényt
13 **else return** „nincs megengedett megoldás”

Most bemutatjuk a SZIMPLEX-KEZDÉS eljárás működését a (29.105)–(29.108) lineáris programozási feladaton. Ennek a lineáris programozási feladatnak akkor van megengedett megoldása, ha találunk olyan nemnegatív x_1 és x_2 értékeket, amelyek kielégítik a (29.106) és (29.107) egyenlőtlenségeket. Írjuk fel a 29.11. lemma szerinti lineáris programozási segédfeladatot:

maximalizálandó $-x_0$ (29.112)

feltéve, hogy

$$2x_1 - x_2 - x_0 \leq 2 \tag{29.113}$$

$$x_1 - 5x_2 - x_0 \leq -4 \tag{29.114}$$

$$x_1, x_2, x_0 \geq 0.$$

A 29.11. lemma szerint, ha ennek a lineáris programozási segédfeladatnak az optimális célfüggvényértéke 0, akkor az eredeti lineáris programozási feladatnak van megengedett megoldása. Ha a segédfeladat optimális célfüggvényértéke negatív, akkor az eredeti lineáris programozási feladatnak nincs megengedett megoldása.

A feladatot kiegyenlített alakban felírva a következőt kapjuk:

$$\begin{aligned} z &= -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0. \end{aligned}$$

Ezzel még nem vagyunk készen, mivel a most adódó bázismegoldásban $x_4 = -4$, így ez nem megengedett megoldása a lineáris programozási segédfeladatnak. Viszont a PIVOTÁLÁS eljárás meghívásával ezt a kiegyenlített alakot átalakíthatjuk egy olyan kiegyenlített alakra, amelyhez megengedett bázismegoldás fog tartozni. A 8. sor szerint x_0 lesz a bázisba belépő változó. A bázisból kilépő változó az 1. sor miatt az x_4 lesz, hisz ez az a bázisváltozó, amelyiknek az értéke a legnegatívabb. A pivotálás után a következő kiegyenlített alakot kapjuk:

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4. \end{aligned}$$

Az ehhez tartozó bázismegoldás az $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$, amely megengedett. Most többször meghívjuk a PIVOTÁLÁS eljárást, míg meg nem kapjuk az L_s feladat optimális megoldását. Esetünkben a PIVOTÁLÁS egyszeri meghívása, amelyben az x_2 a belépő, és x_0 a kilépő változó a következő kiegyenlített alakhoz vezet:

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}. \end{aligned}$$

Ez a kiegyenlített alak a segédfeladat végső megoldása. Mivel a megoldásban $x_0 = 0$, tudjuk, hogy az eredeti problémának van megengedett megoldása. Továbbá, mivel $x_0 = 0$, így az x_0-ás tagok elhagyhatók a korlátozó feltételekből. Majd visszatérhetünk az eredeti célfüggvény használatára. Megfelelő helyettesítésekkel elérhetjük, hogy az csak nembázisváltozókat tartalmazzon. A példában a következő célfüggvényt kapjuk:

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right).$$

Legyen $x_0 = 0$, majd az egyszerűsítés után a célfüggvény:

$$\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5},$$

a kiegyenlített alak pedig:

$$\begin{aligned} z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5}. \end{aligned}$$

Ehhez a kiegyenlített alakhoz megengedett bázismegoldás tartozik, amit átadhatunk a SZIMPLEX eljárásnak.

Most bebizonyítjuk a SZIMPLEX-KEZDÉS eljárás helyességét.

29.12. lemma. *Ha az L lineáris programozási feladatnak nincs megengedett megoldása, akkor a* SZIMPLEX-KEZDÉS *eljárás a „nincs megengedett megoldás” üzenetet adja vissza, egyébként pedig egy olyan kiegyenlített alakot ad vissza, amelyhez megengedett bázismegoldás tartozik.*

Bizonyítás. Elsőként tegyük fel, hogy az L lineáris programozási feladatnak nincs megengedett megoldása. Ekkor a 29.11. lemmából az adódik, hogy a (29.109)–(29.111) szerint megadott L_s feladat optimuma nem nulla. Így az x_0-ra vonatkozó nemnegativitási feltétel miatt az optimális megoldásban a célfüggvény értéke csak negatív lehet. Ez az optimum véges, legyen ugyanis az $x_i = 0$, minden $i = 1, 2, \ldots, n$ indexre, valamint $x_0 = |\min_{i=1}^{m}\{b_i\}|$, ez a megoldás megengedett, és a $-|\min_{i=1}^{m}\{b_i\}|$ célfüggvényérték tartozik hozzá. Ezért a SZIMPLEX-KEZDÉS eljárás a 10. sorban talál egy megoldást, amelyhez negatív célfüggvényérték tartozik. Legyen $\bar{x}$ a végső kiegyenlített alakhoz tartozó bázismegoldás. Ekkor $\bar{x}_0 = 0$ nem lehetséges, hisz ez azt jelentené, hogy az L_s feladat célfüggvényértéke 0, pedig az a fentiek alapján negatív. Tehát a 11. sor vizsgálata után az eljárás a „nincs megengedett megoldás” üzenetet adja vissza a 13. sorban.

Most tegyük fel azt, hogy az L lineáris programozási feladatnak van megengedett megoldása. A 29.3-3. gyakorlat állítása szerint, ha $b_i \geq 0$ minden $i = 1, 2, \ldots, m$ esetén, akkor a kezdeti kiegyenlített alakhoz tartozó bázismegoldás megengedett. Ebben az esetben a 2–3. sorok visszaadják a bemenethez tartozó kiegyenlített alakot. (Nincs sok tennivaló a szabályos alak kiegyenlített alakba történő átírásához, mivel az A, b és c értékek mindkettőben ugyanazok.)

A bizonyítás hátralevő részében azzal az esettel foglalkozunk, amikor a lineáris programozási feladatnak van megengedett megoldása, de az eljárás nem tér vissza a 3. sorban. Megmutatjuk, hogy ekkor az eljárás a 4–10. sorokban talál egy olyan megengedett megoldást az L_s feladatra, amelyhez a 0 célfüggvényérték tartozik. Elsőként, az 1–2. sorok végrehajtása után fennáll, hogy:

$$b_k < 0,$$

és

$$b_k \leq b_i \quad \text{minden } i \in B \text{ indexre.} \tag{29.115}$$

A 8. sorban egy pivotálást hajtunk végre, az x_l kilépő változóval (emlékezzünk vissza, hogy $l = n + k$), amelyik annak az egyenletnek a bal oldala, amelyben a legkisebb b_i érték szerepel, és az x_0 belépő változóval, amelyik az eredeti feladathoz hozzáadott extra változó. Megmutatjuk, hogy ez után a pivotálás után az összes b érték nemnegatív lesz, így az L_s segédfeladat aktuális alakjához tartozó bázismegoldás megengedett. Legyen $\bar{x}$ a PIVOTÁLÁS utáni bázismegoldás és legyenek $\widehat{b}$ és $\widehat{B}$ a PIVOTÁLÁS által visszaadott értékek, ekkor a 29.1. lemmából adódik, hogy

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\widehat{b}_e, & \text{ha } i \in \widehat{B} - \{e\}, \\ b_l / a_{le}, & \text{ha } i = e. \end{cases} \tag{29.116}$$

A 8. sorban a PIVOTÁLÁS hívásakor $e = 0$, valamint a (29.110) alapján

$$a_{i0} = a_{ie} = -1 \quad \text{minden } i \in B \text{ indexre.} \tag{29.117}$$

(Megjegyezzük, hogy az a_{i0} most az az x_0-hoz tartozó együttható, ami a (29.110) egyenlőtlenségekben megjelenik, és nem a (-1)-szeres együttható, mivel az L_s szabályos és nem kiegyenlített alakú.) Minthogy $l \in B$, ezért $a_{le} = -1$. Ebből azt kapjuk, hogy $b_l / a_{le} > 0$, amiből $\bar{x}_e > 0$ adódik. A többi bázisváltozóra fennáll a következő:

$$\begin{aligned} \bar{x}_i &= b_i - a_{ie}\widehat{b}_e && \text{(a (29.116) egyenlőség szerint)} \\ &= b_i - a_{ie}\left(\frac{b_l}{a_{le}}\right) && \text{(PIVOTÁLÁS 2. sora szerint)} \\ &= b_i - b_l && \text{(a (29.117) egyenlőség és } a_{le} = -1 \text{ miatt)} \\ &\geq 0 && \text{(a (29.115) egyenlőtlenség szerint),} \end{aligned}$$

amiből az következik, hogy valamennyi bázisváltozó nemnegatív. Tehát a 8. sorban a PIVOTÁLÁS meghívása után kapott bázismegoldás megengedett. Ezután a 10. sor végrehajtása következik, mely megoldja az L_s feladatot. Mivel feltevésünk szerint az L feladatnak létezik megengedett megoldása, a 29.11. lemmából adódóan az L_s feladat optimális célfüggvényértéke 0. Mivel valamennyi kiegyenlített alak ekvivalens, az L_s feladathoz tartozó végső bázismegoldásban biztos, hogy az $x_0 = 0$. Eltávolítva x_0 változót a lineáris programozási feladatból egy olyan kiegyenlített alakot kapunk L-re, amelynek bázismegoldása megengedett. Ezt a kiegyenlített alakot adja vissza az eljárás a 12. sorban. ■

A lineáris programozás alaptétele

Befejezésül megmutatjuk, hogy a SZIMPLEX eljárás helyesen működik. Egy lineáris programozási feladatnál különféle esetek fordulhatnak elő: lehet, hogy nincs megengedett megoldása, lehet, hogy nem korlátos a célfüggvénye, végül lehet, hogy van optimális megoldása, amelyben a célfüggvény értéke véges. Állításunk szerint a SZIMPLEX eljárás mindezen esetekben helyesen működik.

29.13. tétel (a lineáris programozás alaptétele). *Minden szabályos alakú L lineáris programozási feladatnál az alábbi esetek fordulhatnak elő:*

1. *a feladatnak van optimális megoldása, amelyre a célfüggvényérték véges,*
2. *a feladatnak nincs megengedett megoldása,*

3. a feladat célfüggvénye nem korlátos.

Ha az L-nek nincs megengedett megoldása, akkor a SZIMPLEX *eljárás a „nincs megengedett megoldás” üzenetet küldi. Ha az L célfüggvénye nem korlátos, akkor a* SZIMPLEX *eljárás a „nem korlátos” üzenetet küldi. Egyébként pedig a* SZIMPLEX *eljárás visszaadja a feladat optimális megoldását, amelyre a célfüggvényérték véges.*

Bizonyítás. A 29.12. lemma szerint ha az L lineáris programozási feladatnak nincs megengedett megoldása, akkor a SZIMPLEX eljárás a „nincs megengedett megoldás” üzenetet küldi. Tegyük fel, hogy az L lineáris programozási feladatnak van megengedett megoldása. A 29.12. lemma szerint ilyenkor a SZIMPLEX-KEZD ÉS eljárás egy olyan kiegyenlített alakkal tér vissza, amelyhez tartozó bázismegoldás megengedett. Így a 29.7. lemmából adódóan a SZIMPLEX eljárás vagy a „nem korlátos” üzenetet küldi, vagy befejeződik egy megengedett megoldást visszaadva. Ha befejeződik egy véges megoldást visszaadva, akkor a 29.10. tétel szerint ez a megoldás optimális. A másik esetben, ha a SZIMPLEX eljárás a „nem korlátos” üzenetet küldi, a 29.2. lemma szerint az L lineáris programozási feladatnak valóban nem korlátos a célfüggvénye. Mivel a SZIMPLEX eljárás csak e három módon fejeződhet be, a tétel bizonyítását befejeztük. ■

Gyakorlatok

29.4-1. Részletezzük, milyen algoritmusokkal valósítható meg a SZIMPLEX-KEZD ÉS eljárás 5. és 12. sora.

29.4-2. Mutassuk meg, hogy amikor a SZIMPLEX-KEZD ÉS eljárás a SZIMPLEX eljárás fő ciklusát futtatja, soha nem fordulhat elő a „nem korlátos” eset.

29.4-3. Tegyük fel, hogy adott egy szabályos alakú L lineáris programozási feladat. Tegyük fel továbbá, hogy mind az L feladatnak, mind az L duális feladatának a kezdeti kiegyenlített alakjához tartozó bázismegoldások megengedettek. Mutassuk meg, hogy ekkor az L feladatnak létezik optimális megoldása, és ehhez a 0 célfüggvényérték tartozik.

29.4-4. Tegyük fel, hogy megengedünk szigorú egyenlőtlenségeket a lineáris programozási feladat korlátozó feltételeiben. Mutassuk meg, hogy ebben az esetben nem igaz a lineáris programozás alaptétele.

29.4-5. Oldjuk meg az alábbi lineáris programozási feladatot a SZIMPLEX eljárással:

$$
\begin{array}{lrcrcr}
\text{maximalizálandó} & x_1 & + & 3x_2 & & \\
\text{feltéve, hogy} & -x_1 & + & x_2 & \le & -1 \\
& -2x_1 & - & 2x_2 & \le & -6 \\
& -x_1 & + & 4x_2 & \le & 2 \\
& x_1, x_2 & & & \ge & 0\,.
\end{array}
$$

29.4-6. Oldjuk meg a (29.6)–(29.10) lineáris programozási feladatot.

29.4-7. A következő egyváltozós lineáris programozási feladatot nevezzük P-nek:

$$
\begin{array}{ll}
\text{maximalizálandó} & tx \\
\text{feltéve, hogy} & \\
& rx \le s \\
& x \ge 0\,,
\end{array}
$$

ahol r, s és t tetszőleges valós számok. Legyen D a P duál feladata.

Adjuk meg, hogy r, s és t milyen értékei mellett igazak az alábbi állítások:

1. mind P-nek, mind D-nek van optimális megoldása, véges célfüggvényértékkel,
2. P-nek van megengedett megoldása, de D-nek nincs,
3. D-nek van megengedett megoldása, de P-nek nincs,
4. sem P-nek, sem D-nek nincs megengedett megoldása.

Feladatok

29-1. Lineáris egyenlőtlenségrendszer megoldhatósága

Adott m lineáris egyenlőtlenség n számú, $x_1, x_2, \ldots, x_n$ változón. A ***lineáris egyenlőtlenségrendszer megoldhatósági*** problémája azt vizsgálja, hogy létezik-e a változóknak olyan értékbeállítása, amely egyszerre kielégíti valamennyi egyenlőtlenséget.

a. Mutassuk meg, hogy ha van megoldó algoritmusunk a lineáris programozási feladatra, akkor azt használhatjuk a lineáris egyenlőtlenségrendszer megoldhatósági problémájának megoldására. A lineáris programozási feladatban a változók és korlátozó feltételek száma n és m polinomiális függvénye legyen.

b. Mutassuk meg, hogy ha van megoldó algoritmusunk a lineáris egyenlőtlenségrendszer megoldhatósági problémájának megoldására, akkor azt használhatjuk lineáris programozási feladat megoldására is. A lineáris egyenlőtlenségrendszer megoldhatósági problémájában szereplő változók és egyenlőtlenségek száma legyen n és m polinomiális függvénye, ahol n és m a lineáris programozási feladatban a változók és korlátozó feltételek számát jelöli.

29-2. Kiegészítő eltérések

A ***kiegészítő eltérések*** tétele a primál feladat változói és a duál feladat korlátozó feltételei, valamint a duál feladat változói és a primál feladat korlátozó feltételei közti összefüggést írja le. Legyen $\bar{x}$ a (29.16)–(29.18) primál lineáris programozási feladat megengedett megoldása, valamint legyen $\bar{y}$ a (29.86)–(29.88) duál feladat megengedett megoldása. A kiegészítő eltérések tétele kimondja, hogy az $\bar{x}$ és $\bar{y}$ megoldások pontosan akkor optimálisak, ha az alábbi két összefüggés teljesül:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ vagy } \bar{x}_j = 0 \quad (j = 1, 2, \ldots, n)$$

és

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \text{ vagy } \bar{y}_i = 0 \quad (i = 1, 2, \ldots, m).$$

a. Igazoljuk, hogy a kiegészítő eltérések tétele fennáll a (29.56)–(29.60) lineáris programozási feladatra.

b. Mutassuk meg, hogy a kiegészítő eltérések tétele fennáll tetszőleges primál és hozzá tartozó duál lineáris programozási feladatra.

c. Igazoljuk, hogy a (29.16)–(29.18) lineáris programozási feladat $\bar{x}$ megengedett megoldása akkor és csak akkor optimális, ha léteznek hozzá olyan $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ értékek, amelyekre fennállnak a következők:

1. $\bar{y}$ megengedett megoldása a (29.86)–(29.88) duál lineáris programozási feladatnak,
2. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ teljesül minden olyan $j = 1, \ldots, n$ indexre, ahol $\bar{x}_j > 0$, és
3. $\bar{y}_i = 0$ teljesül minden olyan $i = 1, \ldots, m$ indexre, ahol $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

29-3. Egész értékű lineáris programozási feladat

Az ***egész értékű lineáris programozási feladat*** olyan lineáris programozási feladat, amelyben az x értékei csak egész számok lehetnek. A 34.5-3. gyakorlat kitűzi annak bizonyítását, hogy már annak eldöntése is NP-nehéz, hogy létezik-e egy egész értékű lineáris programozási feladatnak megengedett megoldása, amiből az következik, hogy erre a feladatra valószínűleg nem lehet polinom idejű megoldó algoritmust találni.

a. Mutassuk meg, hogy a gyenge dualitás (29.8. lemma) fennáll az egész értékű lineáris programozási feladatra is.

b. Mutassuk meg, hogy a dualitás (29.10. tétel) nem mindig igaz az egész értékű lineáris programozási feladatra.

c. Tekintsünk egy szabályos alakban adott lineáris programozási feladatot. Legyen P a primál feladat optimális célfüggvényértéke, D a duál feladat optimális célfüggvényértéke, legyen IP a primál feladat egész értékű változatához (az eredeti primál feladat kiegészítve azzal a megszorítással, hogy a változók csak egész értékeket vehetnek fel) tartozó optimális célfüggvényérték, és ID a duál feladat egész értékű változatához tartozó optimális célfüggvényérték. Mutassuk meg, hogy ha mind a primál, mind a duál egész értékű lineáris programozási feladatnak van megengedett megoldása, és mindkettő célfüggvénye korlátos, akkor

$$IP \leq P = D \leq ID.$$

29-4. Farkas-lemma

Legyen A $(m \times n)$-es mátrix, c egy n-dimenziós vektor. A Farkas-lemma szerint a következő két rendszer közül pontosan az egyiknek létezik megoldása:

$$\begin{aligned} Ax &\leq 0, \\ c^{\mathrm{T}}x &> 0 \end{aligned}$$

és

$$\begin{aligned} A^{\mathrm{T}}y &= c, \\ y &\geq 0, \end{aligned}$$

ahol az x egy n-dimenziós vektor, az y egy m-dimenziós vektor. Bizonyítsuk be a Farkas-lemmát.

Megjegyzések a fejezethez

Ez a fejezet csak a kezdet a lineáris programozás széles területének tanulmányozásához. Számos könyv foglalkozik kifejezetten a lineáris programozással, néhány közülük: V. Chvátal [62], S. Gass [111], H. Karloff [171], A. Schrijver [266] és R. J. Vanderbei [304]. Több más könyv jó áttekintést ad a lineáris programozásról, például C. H. Papadimitriou és K. Steiglitz [237], valamint R. K. Ahuja, T. L. Magnanti és J. B. Orlin [7]. Ez a fejezet V. Chvátal megközelítésmódját használja.

A szimplex módszert a lineáris programozási feladat megoldására G. Dantzig fejlesztette ki 1947-ben. Röviddel ezután felfedezték, hogy számos, a legváltozatosabb területeken felmerülő probléma megfogalmazható lineáris programozási feladatként és megoldható a szimplex módszerrel. Ez a felismerés a lineáris programozás felvirágzásához vezetett, és ezzel a szimplex algoritmuson kívül sok egyéb algoritmus is megszületett. A szimplex módszer különféle változatai ma is a legnépszerűbbek a lineáris programozási feladatok megoldására. A terület fejlődésének története több műben megtalálható, köztük a [62] és [171] jegyzetekben.

Az első polinom idejű algoritmus a lineáris programozási feladat megoldására az 1979-ben publikált ellipszoid módszer volt, mely L. G. Khachian nevéhez fűződik. A módszer N. Z. Shor, D. B. Judin és A. S. Nemirovskii korábbi munkáin alapszik. M. Grötschel, Lovász László és A. Schrijver [134] munkájukban az ellipszoid módszer alkalmazását mutatják be számos kombinatorikus optimalizálási probléma megoldására. Napjainkig az ellipszoid módszer nem vált a szimplex módszer valódi versenytársává a gyakorlati problémák megoldásában.

N. Karmarkar [172] cikkében mutatja be belső pontos algoritmusát. Ezt követően több kutató is tervezett belső pontos algoritmusokat. Jó áttekintést találunk D. Goldfarb és M. J. Todd [122] cikkében, valamint Y. Ye [319] könyvében.

A szimplex módszer elemzése ma is aktív kutatási terület. V. Klee és G. J. Minty készítettek egy olyan feladatot, amelyen a szimplex módszer $2^n - 1$ iterációt végez. A szimplex módszer a gyakorlatban legtöbbször igen hatékonyan működik, sok kutató próbálkozott ezen empirikus megfigyelés elvi bizonyításával. Ezen a területen K. H. Borgwardt kezdeményezte a kutatást, és sokan mások követték. A bemeneti adatokat véletlen számoknak tekintve, és eloszlásukra bizonyos feltételeket előírva, bizonyítani lehet, hogy a szimplex módszer várható futási ideje polinomiális. Ezen a területen a legfrissebb kutatásokat D. A. Spielman és S. Teng [284] végezték, bevezették az algoritmusok „simított elemzését”, és ezt alkalmazták a szimplex módszerre.

A szimplex algoritmus hatékonyabb bizonyos speciális problémák esetén. Említésre méltó például a hálózati szimplex algoritmus, mely a szimplex módszernek a hálózati folyam problémák megoldására kifejlesztett változata. Bizonyos hálózati problémák – ilyenek a legrövidebb utak, a maximális folyam és a minimális költségű folyam problémák – megoldásához használt hálózati szimplex algoritmusok különféle változatai polinom időben futnak. Ezekre példákat J. B. Orlin [234] cikkében, valamint az ott közölt hivatkozásokban találhatunk.

30. Polinomok és gyors Fourier-transzformáció

Két n-edfokú polinom összeadásához közvetlenül a definícióból kiindulva $\Theta(n)$ idő szükséges, míg a szorzás $\Theta(n^2)$ idő alatt végezhető el. Ebben a fejezetben megmutatjuk, hogy gyors Fourier-transzformáció (angolul: Fast Fourier Transform vagy röviden: FFT) alkalmazásával a polinomok szorzásához szükséges idő nagyságrendje $\Theta(n \lg n)$-re csökkenthető.

A Fourier-transzformációt és ezzel együtt az FFT algoritmust kiterjedten alkalmazzák a jelfeldolgozásban. Jelnek az ***időtartományban*** olyan függvényt neveznek, amely időhöz amplitúdót rendel. A Fourier-analízis segítségével a jeleket különböző frekvenciájú és fázisú szinuszrezgések súlyozott összegeként állíthatjuk elő. Az egyes frekvenciákhoz tartozó súlyok és fázisok a jelek leírására szolgálnak a ***frekvenciatartományban.*** A jelfeldolgozás igen gazdag irodalommal rendelkezik. A fejezet végén ezzel kapcsolatban megemlítünk néhány kitűnő könyvet.

Polinomok

Legyen F egy algebrai test és $a_0, a_1, \ldots, a_n$ F-beli elemek. Az

$$A(x) = \sum_{j=0}^{n} a_j x^j \quad (x \in F)$$

utasítással értelmezett A függvényt ***polinomnak*** nevezzük. Az $a_0, a_1, \ldots, a_n$ számokat a polinom ***együtthatóinak*** nevezzük. Az együtthatók az F test elemei; ez rendszerint a komplex számok C teste. Akkor mondjuk, hogy az A polinom ***fokszáma*** k, ha a legnagyobb indexű nullától különböző együttható a_k. Az A polinom fokszámát – a megfelelő angol elnevezés rövidítését használva – a fok(A) szimbólummal fogjuk jelölni. Bármely, a polinom fokszámánál nagyobb egész számot a polinom ***fokszámkorlátjának*** nevezünk.

A polinomok körében szokás műveleteket értelmezni. Legyen A és B két legfeljebb n-edfokú polinom. A $C(x) = A(x) + B(x)$ utasítással értelmezett, legfeljebb n-edfokú C polinomot az A és B ***összegének*** nevezzük. Azaz, ha

$$A(x) = \sum_{j=0}^{n} a_j x^j$$

és

$$B(x) = \sum_{j=0}^{n} b_j x^j,$$

akkor

$$C(x) = \sum_{j=0}^{n} c_j x^j,$$

ahol $c_j = a_j + b_j \quad (j = 0, 1, \ldots, n)$. Például, ha $A(x) = 6x^3 + 7x^2 - 10x + 9$ és $B(x) = -2x^3 + 4x - 5$, akkor $C(x) = 4x^3 + 7x^2 - 6x + 4$.

A legfeljebb n-edfokú A és B ***polinomok szorzatán*** a $C(x) = A(x)B(x)$ legfeljebb $2n$-edfokú polinomot értjük. A C polinomot úgy kapjuk, hogy az A minden tagját megszorozzuk a B minden tagjával, és az azonos hatványú tagokat összevonjuk. Az $A(x) = 6x^3 + 7x^2 - 10x + 9$ és a $B(x) = -2x^3 + 4x - 5$ polinomok szorzata a következőképpen állítható elő:

$$\begin{array}{rrrrrrrr}
 & & & 6x^3 & + 7x^2 & - 10x & + 9 & \\
 & & & - 2x^3 & & + 4x & - 5 & \\
\hline
 & & & - 30x^3 & - 35x^2 & + 50x & - 45 & \\
 & & 24x^4 & + 28x^3 & - 40x^2 & + 36x & & \\
- 12x^6 & - 14x^5 & + 20x^4 & - 18x^3 & & & & \\
\hline
- 12x^6 & - 14x^5 & + 44x^4 & - 20x^3 & - 75x^2 & + 86x & - 45 &
\end{array}$$

A C polinom másképpen is felírható:

$$C(x) = \sum_{j=0}^{2n} c_j x^j, \tag{30.1}$$

ahol

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}. \tag{30.2}$$

Megjegyezzük, hogy a $\text{fok}(C) = \text{fok}(A) + \text{fok}(B)$ egyenlőségből következik, hogy ha A egy n_A fokszámkorlátú és B egy n_B fokszámkorlátú polinom, akkor C egy $n_A + n_B - 1$ fokszámkorlátú polinom. Mivel a k fokszámkorlátú polinom egyúttal $k + 1$ fokszámkorlátú polinom is, rendszerint azt mondjuk, hogy a C szorzatpolinom $n_A + n_B$ fokszámkorlátú.

A fejezet tartalma

A 30.1. alfejezetben két lehetőséget mutatunk polinomok megadására az együtthatókból, illetve a polinom helyettesítési értékeiből kiindulva. A (30.1) és (30.2) egyenlőségeket alapul véve a szorzatpolinom együtthatóinak kiszámításához $\Theta(n^2)$ idő szükséges, viszont helyettesítési értékeket használva, a szükséges idő csak $\Theta(n)$. Megmutatjuk, hogy – a kétféle reprezentációt kombinálva – az együtthatóival adott polinomok szorzata $\Theta(n \lg n)$ id ő alatt is előállítható. Ennek az eljárásnak a megfogalmazásához felhasználjuk a komplex egységgyököket, amelyekkel a 30.2. alfejezetben foglalkozunk. Ezután ugyancsak a 30.2. alfejezetben leírjuk az FFT-t és inverzét, megmutatva a kétféle reprezentáció kapcsolatát. A 30.3. alfejezetben megadjuk az FFT egy-egy gyors soros és párhuzamos változatát.

Ebben a fejezetben mindvégig használjuk a komplex számokat, és a $\sqrt{-1}$ komplex számot az i szimbólummal fogjuk jelölni.

30.1. Polinomok megadása

A polinomok helyettesítési értékeivel, illetve együtthatóival történő megadása bizonyos értelemben ekvivalens egymással. Ebben a pontban ismertetjük a kétféle előállítást, és megmutatjuk, hogy ezek kombinációjával hogyan lehet n-edfokú polinomok szorzását $\Theta(n \lg n)$ idő alatt elvégezni.

Együtthatóikkal adott polinomok

Az $A(x) = \sum_{j=0}^{n} a_j x^j$ legfeljebb n-edfokú polinom ***együttható-reprezentációján*** az $a = (a_0, a_1, \ldots, a_n)$ F^{n+1}-beli vektort értjük. Ebben a pontban a mátrixokkal adott egyenletekben előforduló vektorokat általában oszlopvektorok formájában használjuk.

Az együttható-reprezentációt bizonyos polinomműveletek során célszerű használni. Ilyen például az A polinom adott x_0 pontban vett $A(x_0)$ helyettesítési értékének kiszámítása, más szóval a polinom ***kiértékelése*** az x_0 pontban. A kiértékeléshez az

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1} + x_0 a_n)\cdots))$$

úgynevezett ***Horner-féle elrendezést*** alkalmazva $\Theta(n)$ idő szükséges. Ehhez hasonlóan az $a = (a_0, a_1, \ldots, a_{n-1})$ és $b = (b_0, b_1, \ldots, b_{n-1})$ együtthatókkal adott polinomok összegét is $\Theta(n)$ idő alatt számíthatjuk ki: a $c = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$ vektort állítjuk elő, ahol $c_j = a_j + b_j$ $(j = 0, 1, \ldots, n-1)$.

Most vizsgáljuk az együtthatóikkal adott A és B polinomok szorzását. A (30.1) és (30.2) formulákat használva a polinomok szorzásához $\Theta(n^2)$ idő szükséges, mivel az a-ban előforduló együtthatók mindegyikét megszorozzuk a b-ben előforduló együtthatókkal. Ennek alapján az együtthatóival adott polinomok szorzása bonyolultabb feladatnak tekinthető, mint összeadásuk vagy kiértékelésük. A szorzatpolinom együtthatóiból készített, a (30.2) képlettel értelmezett c vektort az a és b vektorok ***konvolúciójának*** nevezzük és a $c = a \otimes b$ szimbólummal jelöljük. Mivel polinomok szorzatának és vektorok konvolúciójának kiszámítása számos, gyakorlati szempontból is fontos probléma kapcsán felvetődik, azért ebben a fejezetben egy erre vonatkozó hatékony algoritmust ismertetünk.

Helyettesítési értékeikkel adott polinomok

A legfeljebb n-edfokú A polinom ***pontreprezentációján*** grafikonjának bármely $n+1$ különböző pontjából álló

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$$

ponthalmazát értjük. Ez részletesen szólva azt jelenti, hogy az x_k számok páronként különbözőek és

$$y_k = A(x_k) \quad (k = 0, 1, \ldots, n). \tag{30.3}$$

Minden polinomnak nyilvánvalóan végtelen sok ilyen típusú reprezentációja van, hiszen grafikonjának minden, $n+1$ különböző pontból álló részrendszere egy-egy pontreprezentációnak felel meg.

Az együtthatók ismeretében a polinom pontreprezentációi közvetlenül el őállíthatók, kiszámítva az adott, n különböző $x_0, x_1, \ldots, x_{n-1}$ pontban az $A(x_k)$ $(k = 0, 1, \ldots, n-1)$ függvényértékeket. A Horner-féle elrendezést használva ehhez $\Theta(n^2)$ idő szükséges. Később megmutatjuk, hogy az x_k helyek ügyes megválasztásával a számítás gyorsabban, nevezetesen $\Theta(n \lg n)$ idő alatt is elvégezhető.

A most ismertetett eljárás inverzét, vagyis a polinom együtthatóinak meghatározását pontreprezentációjának ismeretében, ***interpolációnak*** nevezzük. A következ ő tételben megmutatjuk, hogy ha a kívánt interpolációs polinomnak olyan fokszámkorláttal kell rendelkeznie, mint a megadott pontok száma, akkor az interpolációs feladat egyértelműen megoldható.

30.1. tétel (az interpolációs polinom egyértelműsége). *Minden n pontból álló $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ pontrendszerhez, ahol $x_0, x_1, \ldots, x_n$ különböző számok, pontosan egy olyan n fokszámkorlátú polinom létezik, amely eleget tesz az $A(x_k) = y_k$ $(k = 0, 1, \ldots, n-1)$ feltételeknek.*

Bizonyítás. Az állítás egy speciális mátrix invertálhatóságából következik. A (30.3) feltétel ekvivalens a következő lineáris egyenletrendszerrel:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \tag{30.4}$$

A bal oldalon álló úgynevezett Vandermonde-féle mátrixot $V(x_0, x_1, \ldots, x_{n-1})$-gyel jelöljük. Ennek determinánsa a

$$\prod_{0 \le j < k \le n} (x_k - x_j)$$

szorzattal egyenlő (lásd a 28.1-11. gyakorlatot). Mivel az x_k $(k = 0, 1, \ldots, n)$ számok páronként különböz őek, ezért a determináns értéke nullától különböz ő, következésképpen a 28.5. tétel alapján a szóban forgó mátrix invertálható. Ezért

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y.$$

A polinom együtthatói tehát az adott függvényértékekb ől egyértelműen meghatározhatók. ■

A 30.1. tétel bizonyítása egyúttal egy algoritmust is ad az interpolációs feladat megoldására – visszavezetve azt a (30.4) lineáris egyenletrendszer megoldására. A 28. fejezetben ismertetett LU felbontást alkalmazva, ez a lineáris egyenletrendszer $O(n^3)$ idő alatt megoldható.

Gyorsabb algoritmust kapunk, ha az interpolációs polinomot a következ ő, úgynevezett ***Lagrange-féle interpolációs formulával*** adjuk meg:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k} (x - x_j)}{\prod_{j \ne k} (x_k - x_j)}. \tag{30.5}$$

Könnyen ellenőrizhető, hogy a jobb oldalon valóban olyan n fokszámkorlátú polinom áll, amely minden k-ra eleget tesz az $A(x_k) = y_k$ feltételnek. Lagrange-féle interpolációt használva A együtthatóit $\Theta(n^2)$ idő alatt számíthatjuk ki (lásd a 30.1-5. gyakorlatot).

Ezzel megmutattuk, hogy a kiértékelési és az interpolációs eljárást felhasználva, a pontreprezentációról áttérhetünk az együttható-reprezentációra.[1] Az erre vonatkozó, most ismertetett algoritmus műveleti ideje $\Theta(n^2)$.

Polinomok pontreprezentációja több, polinomokkal kapcsolatos művelet esetén is jól használható. Például, polinomok összeadását vizsgálva, legyen $C = A + B$. Ekkor minden x_k pontban $C(x_k) = A(x_k) + B(x_k)$. Más szóval, ha

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

az A-nak,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

pedig a B-nek egy (ugyanazokon a helyeken vett) pontreprezentációja, akkor az

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

pontrendszer az $A + B$ polinomnak egy pontreprezentációját adja. Nyilvánvaló, hogy két, helyettesítési értékeivel adott, n fokszámkorlátú polinom összeadásához $\Theta(n)$ idő szükséges.

Ehhez hasonlóan, jól alkalmazható a szóban forgó reprezentáció polinomok szorzása esetén is. Ha $C = AB$, akkor minden x_k helyen $C(x_k) = A(x_k)B(x_k)$, ezért a C szorzatot reprezentáló függvényértékek az A és B polinomot reprezentáló függvényértékek szorzataként kaphatók. Vegyük figyelembe, hogy fok(C) = fok(A) + fok(B); ha A és B n fokszámkorlátúak, akkor C $2n$ fokszámkorlátú. A szokásos eljárást használva mind A, mind B pontreprezentációja n pontot tartalmaz. Az ezekhez tartozó függvényértékeket összeszorozva n pontot kapunk. Egy $2n$ fokszámkorlátú C polinom egyértelmű interpolációjához $2n$ pontra van szükségünk. (Lásd a 30.1-4. gyakorlatot.) Ezért A és B „kiterjesztett” pontreprezentációjából kell kiindulnunk, amely mindkét polinom esetén $2n$ pontból áll. Ezek A esetében az

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$$

pontok, a B polinom esetében pedig az

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\}$$

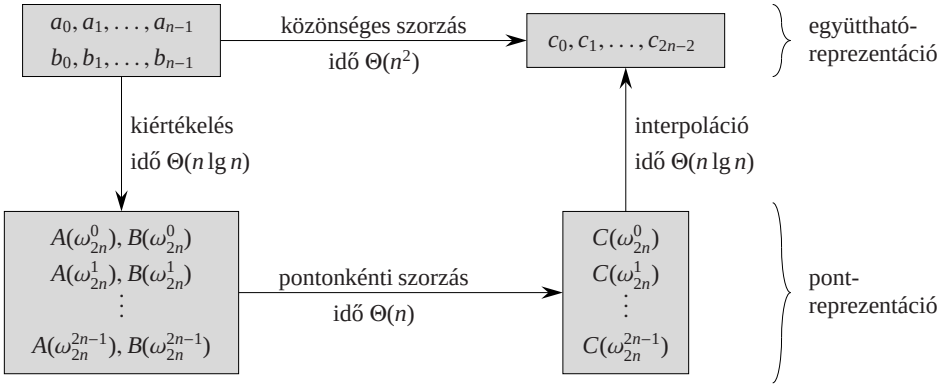
pontok. Ekkor a C polinom grafikonjának

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}$$

pontjaival reprezentálható. Ennek alapján látható, hogy a pontreprezentációból kiindulva, két polinom szorzatának pontreprezentációját $\Theta(n)$ idő alatt tudjuk előállítani, ugyanakkor együttható-reprezentáció esetén ugyanehhez a művelethez a jóval nagyobb $\Theta(n^2)$ idő szükséges.

Végül azt vizsgáljuk, hogy lehet egy pontreprezentációban adott polinom értékét egy újabb pontban kiszámítani. Ehhez célszerű áttérni az együttható-reprezentációra és ezt felhasználva kiszámítani a helyettesítési értéket az új pontban.

[1] Az interpoláció közismerten kényes feladat a numerikus stabilitás szempontjából. Annak ellenére, hogy a most ismertetett eljárás matematikai szempontból korrekt, előfordulhat, hogy a bemenő adatok kis hibái, illetve a kerekítési hibák a számítás során az eredményben nagy hibát okoznak.



30.1. ábra. Polinomok szorzására szolgáló hatékony algoritmus vázlata. Felül a polinomokat együtthatóikkal adjuk meg, míg alul helyettesítési értékeikkel. A balról jobbra mutató nyilak a szorzás műveletének felelnek meg. Az ω_{2n} hatványai a $2n$-edik komplex egységgyökök.

Együtthatóikkal adott polinomok gyors szorzása

Felhasználható-e a pontreprezentációval adott polinomok szorzására szolgáló – lineáris futási idejű – algoritmus abban az esetben, ha a polinom együtthatóit ismerjük. A válasz attól függ, hogy rendelkezésünkre állnak-e olyan algoritmusok, amelyekkel az együtthatók ismeretében a helyettesítési értékeket, és megfordítva, a polinom értékeib ől az együtthatókat gyorsan tudjuk kiszámítani. Más szóval, tudunk-e hatékonyan kiértékelni és interpolálni?

A kiértékeléshez tetszőleges pontokból kiindulhatunk, ahhoz azonban, hogy a kétféle reprezentáció közötti áttérést $\Theta(n \lg n)$ idő alatt végezhessük el, az alappontokat speciálisan kell megválasztani. A 30.2. alfejezetben megmutatjuk, hogy alappontoknak a komplex egységgyököket választva, az ezekhez tartozó helyettesítési értékek az együttható vektor diszkrét Fourier-transzformáltjaként (röviden: DFT-ként) származtathatók. Az inverz művelet, azaz az interpoláció ebben a speciális esetben a szóban forgó pontokban felvett értékek inverz diszkrét Fourier-transzformáltjaként adódik. A 30.2. alpontban megmutatjuk, hogy FFT-t alkalmazva a DFT-t és inverzét $\Theta(n \lg n)$ idő alatt lehet megkapni.

Ezt az eljárást a 30.1. ábrán grafikusan szemléltetjük. A továbbiakban az FFT-vel kapcsolatban n-edfokúak helyett célszerű legfeljebb $(n-1)$-edfokú polinomokból kiindulni. Ezeket n-dimenziós együttható vektorokkal jellemezhetjük. A fokszámmal kapcsolatban egy kis meggondolás szükséges. Két legfeljebb $(n-1)$-edfokú polinom szorzatának fokszáma legfeljebb $2n-2$. Mielőtt az algoritmus bemenetéül szolgáló A és B polinomokat kiértékelnénk – az n és $2n-1$ közé eső indexekre az együtthatókat 0-nak véve – az A-t és B-t legfeljebb $(2n-1)$-edfokú polinomnak tekintjük. Mivel az együttható vektoroknak $2n$ komponense van, azért az algoritmusban a $(2n)$-edik komplex egységgyököket használjuk, s ezeket a 30.1. ábrán az ω_{2n} hatványaival jelöljük.

Felhasználva az FFT-t, az együtthatóival adott, legfeljebb $(n-1)$-edfokú A és B polinomok szorzatát a következő, $\Theta(n \lg n)$ időt igénylő algoritmussal számíthatjuk ki. Célszerű feltenni, hogy n 2^k alakú, ahol k természetes szám. Ez a feltétel megfelelő számú nulla együttható hozzávételével mindig teljesíthető.

1. *A fokszám megduplázása.* Nulla együtthatók hozzávételével előállítjuk az A és B polinomok együttható-reprezentációját a legfeljebb $(2n-1)$-edfokú polinomok körében.
2. *Kiértékelés.* Előállítjuk az A és B értékeit a $2n$-edik egységgyököknek megfelelő pontokban, kétszer alkalmazva az FFT algoritmust. Ezzel megkapjuk a polinomok értékeit az ω_{2n}^k $(k = 0, 1, \ldots 2n-1)$ pontokban.
3. *Pontonkénti szorzás.* A megfelelő helyettesítési értékeket összeszorozva, előállítjuk a $C = AB$ polinom értékeit a $2n$-edik egységgyököknek megfelelő pontokban.
4. *Interpoláció.* Előállítjuk a C polinom együttható-reprezentációját, kiindulva a $2n$ függvényértékből, az inverz diszkrét Fourier-transzformált kiszámítására az FFT algoritmust használva.

Az 1. és 3. lépés végrehajtásához $\Theta(n)$ idő szükséges, míg a 2. és 4. lépés $\Theta(n \lg n)$ idő alatt hajtható végre. Ezzel – az FFT algoritmust ismertnek véve – megmutattuk, hogy fennáll a következő

30.2. tétel. *Két, együtthatóival adott n fokszámkorlátú polinom szorzatának együtthatóit $\Theta(n \lg n)$ idő alatt számíthatjuk ki.*

Gyakorlatok

30.1-1. A (30.1) és (30.2) egyenlőségeket felhasználva, állítsuk elő az $A(x) = 7x^3 - x^2 + x - 10$ és a $B(x) = 8x^3 - 6x + 3$ polinomok szorzatát.

30.1-2. A legfeljebb n-edfokú A polinom x_0 helyen vett helyettesítési értéke a következőképpen számítható ki: Osszuk el az A polinomot $(x - x_0)$-lal és jelöljük q-val a legfeljebb $(n-1)$-edfokú hányadost és r-rel a maradékot, azaz

$$A(x) = q(x)(x - x_0) + r.$$

Nyilvánvaló, hogy $A(x_0) = r$. Mutassuk meg, hogy az r maradék és a q együtthatói $\Theta(n)$ idő alatt számíthatók ki.

30.1-3. Állítsuk elő az $A^{\text{ford}}(x) = \sum_{j=0}^{n} a_{n-j} x^j$ helyettesítési értéket az $A(x) = \sum_{j=0}^{n} a_j x^j$ függvényérték segítségével, feltéve, hogy $x \neq 0$.

30.1-4. Igazoljuk, hogy n különböző helyen vett érték szükséges az n fokszámkorlátú polinomok egyértelmű meghatározásához. Más szóval, ha n-nél kevesebb pontot adunk meg, akkor több n) fokszámkorlátú polinom tesz eleget az interpolációs feltételnek. (*Útmutatás.* Alkalmazzuk a 30.1. tételt.)

30.1-5. Mutassuk meg, hogy a (30.5) egyenlőség alapján az interpoláció elvégzéséhez $\Theta(n^2)$ idő szükséges. (*Útmutatás.* Először számítsuk ki a $\prod_j (x - x_j)$ polinom együttható-reprezentációját, azután szükség szerint osszuk el minden tag számlálóját $(x - x_k)$-val; lásd 30.1-2. gyakorlat. Az n nevező mindegyike $O(n)$ idő alatt kiszámítható.

30.1-6. Mi okozhat problémát polinomok szokásos „osztásánál”, ha pontreprezentációból indulunk ki? Vizsgáljuk az eseteket aszerint, hogy az osztásnak van-e maradéka, vagy sem.

30.1-7. Legyen A és B két olyan n elemű halmaz, amelyeknek az elemei a $0, 1, 2, \ldots, 10n$ számok közül valók. Számítsuk ki az A és B halmaz alább értelmezett ún. ***Descartes-féle összegét:***

$$C = \{x + y : x \in A \text{ és } y \in B\}.$$

Vegyük figyelembe, hogy a C elemei 0 és $20n$ közé eső egész számok. Számítsuk ki a C elemeit és határozzuk meg az ehhez szükséges időt. Mutassuk meg, hogy a feladat $\Theta(n \lg n)$ idő alatt megoldható. (*Útmutatás.* Állítsuk elő az A és B halmazokat legfeljebb $10n$-edfokú polinom értékeiként.)

30.2. A DFT és az FFT algoritmus

A 30.1. fejezetben már említettük, hogy ha alappontoknak az n-edik komplex egységgyököket választjuk, akkor a függvények kiértékeléséhez és az interpolációs feladat megoldásához $\Theta(n \lg n)$ idő szükséges. Ebben a fejezetben emlékeztetünk a komplex egységgyökök értelmezésére, összefoglaljuk legfontosabb tulajdonságaikat, bevezetjük a DFT algoritmust, és megmutatjuk, hogy az FFT algoritmust alkalmazva a DFT-nek és inverzének a kiszámításához $\Theta(n \lg n)$ idő szükséges.

Komplex egységgyökök

Az ω komplex számot ***n*-edik egységgyöknek** nevezzük, ha

$$\omega^n = 1.$$

Pontosan n különböző n-edik egységgyök létezik, nevezetesen az $e^{2\pi ik/n}$ komplex számok, ahol $k = 0, 1, \ldots, n-1$. Ezek szemléltetéséhez felhasználjuk az

$$e^{iu} = \cos(u) + i \sin(u)$$

Euler-féle összefüggést. A 30.2. ábrán azt szemléltetjük, hogy az n-edik komplex egységgyökök a 0 középpontú, egységsugarú körön egy szabályos n oldalú sokszög csúcsaiban helyezkednek el. Az

$$\omega_n = e^{i\frac{2\pi}{n}} \tag{30.6}$$

komplex számot ***alap n-edik egységgyöknek*** nevezzük. Valamennyi n-edik egységgyök az ω_n hatványaként állítható elő.[2]

Az n számú

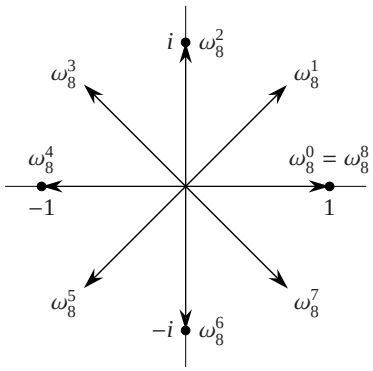
$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

komplex n-edik egységgyök a szorzásra nézve csoportot alkot (lásd a 31.3. alfejezetet). Mivel $\omega_n^n = \omega_n^0 = 1$ alapján $\omega_n^j \omega_n^k = \omega_n^k \omega_n^j = \omega_n^{(j+k) \bmod n}$, azért az n-edik egységgyökök csoportja hasonló szerkezetű, mint az egész számok $\mathbf{Z}$ halmazának modulo n vett csoportja. Ugyanígy igazolható, hogy $\omega_n^{-1} = \omega_n^{n-1}$. A következő lemmákban összefoglaljuk a n-edik komplex egységgyökök legfontosabb tulajdonságait.

30.3. lemma (egyszerűsítési szabály). *Tetszőleges $n \geq 0$, $k \geq 0$ és $d > 0$ egész számokra*

$$\omega_{dn}^{dk} = \omega_n^k. \tag{30.7}$$

[2] Vannak szerzők, akik az ω_n-et másként használják: $\omega_n = e^{-2\pi i/n}$. Ez az értelmezés a jelfeldolgozásban hasznos. A matematikai háttér azonban az ω_n mindkét értelmezése esetén ugyanaz.



30.2. ábra. Az $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ komplex számok a komplex számsíkon, ahol $\omega_8 = e^{2\pi i/8}$ az alap 8-adik komplex egységgyök.

Bizonyítás. A lemma közvetlenül következik a (30.6) definícióból:

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/(dn)}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^k \\ &= \omega_n^k. \end{aligned}$$

■

30.4. következmény. *Minden páros $n > 0$ egész számra*

$$\omega_n^{n/2} = \omega_2 = -1.$$

Bizonyítás. Lásd a 30.2.1. gyakorlatot. ■

30.5. lemma (felezési szabály). *Ha $n > 0$ páros egész szám, akkor a komplex n-edik egységgyökök négyzetei megegyeznek a komplex $(n/2)$-edik egységgyökökkel.*

Bizonyítás. Az egyszerűsítési szabály szerint minden k nemnegatív egész számra $(\omega_n^k)^2 = \omega_{n/2}^k$ teljesül. Vegyük figyelembe, hogy az n-edik komplex egységgyököket négyzetre emelve minden $(n/2)$-edik négyzetgyök kétszer fordul el ő, mivel

$$\begin{aligned} \left(\omega_n^{k+n/2}\right)^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= \left(\omega_n^k\right)^2. \end{aligned}$$

Következésképpen az ω_n^k és $\omega_n^{k+n/2}$ számok négyzete megegyezik. Ez az állítás a 30.4. következmény alapján is igazolható. Valóban, mivel $\omega_n^{n/2} = -1$ alapján $\omega_n^{k+n/2} = -\omega_n^k$, azért innen négyzetre emeléssel $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$ következik. ■

Látni fogjuk, hogy a felezési lemma alapvető szerepet játszik az oszd-meg-és-uralkodj elvű megközelítésben, amellyel az együttható- és a pontreprezentáció között kapcsolatot teremthetünk. Ezzel a gondolatmenettel ugyanis a feladat redukálható két ugyanolyan jellegű, de feleakkora méretű problémára.

30.6. lemma (összegezési formula). *Legyen $n \geq 1$ egész szám és k egy n-nel nem osztható, nemnulla egész szám. Ekkor*

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

Bizonyítás. Az (A.5) egyenlőség mind komplex, mind valós értékekre érvényes, ezért

$$\begin{aligned} \sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0. \end{aligned}$$

Az a feltétel, hogy k nem osztható n-nel, garantálja, hogy a nevező nem 0, ugyanis $\omega_n^k = 1$ csak akkor teljesül, ha k osztható n-nel. ■

A DFT

Emlékeztetünk arra, hogy az

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad (x \in \mathbf{C})$$

polinom értékeit akarjuk meghatározni az $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ helyeken, azaz az n-edik egységgyököknek megfelelő pontokban.[3] Az általánosság megszorítása nélkül feltehetjük, hogy n a 2 számnak egy hatványa, hiszen a polinom együtthatóit nullákkal kibővítve ezt mindig elérhetjük.[4] Legyenek az A együtthatói: $a = (a_0, a_1, \ldots, a_{n-1})$. A helyettesítési értékeket y_k-val jelölve, $k = 0, 1, \ldots, n-1$ esetén legyen

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \tag{30.8}$$

Az $y = (y_0, y_1, \ldots, y_{n-1})$ vektort az $a = (a_0, a_1, \ldots, a_{n-1})$ vektor ***diszkrét Fourier-transzformáltjának*** (röviden: ***DFT***-nek) nevezzük és az $y = \mathrm{DFT}_n(a)$ szimbólummal jelöljük.

---

[3] A 30.1. alpontban az $(n-1)$-edfokú polinomokat a megfelelő zéró együtthatók hozzávételével legfeljebb $(2n-1)$-edfokú polinomoknak tekintjük. Ezért ilyen polinomok szorzásával kapcsolatban a $(2n)$-edik komplex egységgyököket használjuk.

[4] Amikor az FFT algoritmust a jelfeldolgozásban alkalmazzuk, általában nem engedhető meg 0 együtthatójú tagok hozzávétele, hogy 2^s számú tagot kapjunk. Ez a lépés magas frekvenciájú mesterséges tagok fellépését eredményezi. Ezt a problémát a ***tükrözés*** alkalmazásával lehet kiküszöbölni. Legyen n' az n-nél nagyobb, legkisebb 2 hatvány és vezessük be az új együtthatókat az $a_{n+j} = a_{n-j-2}$ $(j = 0, 1, \ldots, n' - n - 1)$ utasítás szerint.

Az FFT

Az úgynevezett ***gyors Fourier-transzformáció*** (angolul: ***Fast Fourier Transform (FFT)***) alkalmazásával, a komplex egységgyökök speciális tulajdonságait felhasználva a $\mathrm{DFT}_n(a)$ vektor kiszámítása $\Theta(n \lg n)$ idő alatt végezhető el, ugyanakkor közvetlenül (30.8) alapján számolva $\Theta(n^2)$ időt használunk fel.

Az FFT algoritmusban egy oszd-meg-és-uralkodj elvű stratégiát alkalmazunk. Nevezetesen az A polinomot – páros és páratlan indexű együtthatóit felhasználva – előállítjuk a legfeljebb $(n/2-1)$-edfokú $A^{[0]}$ és $A^{[1]}$ polinomok segítségével, ahol

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}. \end{aligned}$$

Vegyük észre, hogy az $A^{[0]}$ együtthatóit a páros indexű együtthatók, az $A^{[1]}$ együtthatóit pedig a páratlan indexű együtthatók felhasználásával kaptuk. Az első esetben azokat az indexeket vettük figyelembe, amelyek bináris előállítása 0-ra végződik, a második esetben pedig azokat, amelyek 2-es számrendszerbeli előállítása 1-re végződik. Ezekkel az A polinom felírható

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{30.9}$$

alakban. Ezzel az n-edfokú $A(x)$ polinomnak az $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ halmazon való kiértékelésének problémáját visszavezettük

1. az $n/2$ fokszámú $A^{[0]}$ és $A^{[1]}$ polinomoknak az

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 \tag{30.10}$$

   halmazon való kiértékelésére, és

2. a kapott eredményeket felhasználva – (30.9) alapján – az $A(x)$ kiszámítására.

A felezési lemma szerint a (30.10) képletben felsorolt n komplex szám közül csak $n/2$ különböző, hiszen mindegyik pontosan kétszer fordul elő. Következésképpen a problémát visszavezettük az $(n/2-1)$ fokszámkorlátú $A^{[0]}$ és $A^{[1]}$ polinom kiértékelésére az $n/2$ számú $n/2$-ik egységgyökökön. Ezek a részproblémák ugyanolyan típusú, de fele akkora méretű feladatok, mint az eredeti. Ezzel a DFT_n kiszámítását sikerült visszavezetni a két $\mathrm{DFT}_{n/2}$ vektor kiszámítására. Ez a felbontás képezi a következő FFT algoritmus alapját, amellyel $n = 2^k$ esetén kiszámítjuk az $a = (a_0, a_1, \ldots, a_{n-1})$ vektor DFT-jét.

```
REKURZÍV-FFT(a)
 1  n ← hossz[a]    ▷ n kettőhatvány
 2  if n = 1
 3     then return a
 4  ω_n ← e^{2πi/n}
 5  ω ← 1
 6  a^[0] ← (a_0, a_2, ..., a_{n-2})
 7  a^[1] ← (a_1, a_3, ..., a_{n-1})
 8  y^[0] ← REKURZÍV-FFT(a^[0])
 9  y^[1] ← REKURZÍV-FFT(a^[1])
```

```
10  for k ← 0 to n/2 − 1
11      do y_k ← y_k^[0] + ω y_k^[1]
12         y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
13         ω ← ω ω_n
14  return y          ▷ y-t oszlopvektornak tekintjük.
```

A Rekurzív-FFT algoritmus a következőképpen működik. A 2. és 3. sor a rekurzió kezdő lépésének felel meg, ui. egydimenziós vektor DFT-je önmagával egyenlő:

$$\begin{aligned} y_0 &= a_0\omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0. \end{aligned}$$

A 6. és 7. sorban az $A^{[0]}$ és az $A^{[1]}$ polinomok együttható vektorait definiáljuk. A 4., 5. és 13. sorok garantálják az ω helyes beállítását, így a 11. és 12. sor végrehajtása után $\omega = \omega_n^k$. (Az ω értékének iterációról iterációra való megőrzésével a **for** cikluson belül időt takaríthatunk meg az ω_n^k kiszámításánál.) A 8. és 9. sorban végrehajtjuk a $\mathrm{DFT}_{n/2}$ rekurzív kiszámítását, meghatározva az

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) \end{aligned}$$

komplex számokat a $k = 0, 2, \ldots, n/2 - 1$ indexekre. Mivel az egyszerűsítési szabály szerint $\omega_{n/2}^k = \omega_n^{2k}$, azért

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}), \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}). \end{aligned}$$

A 11. és 12. sorban a rekurzív módon kiszámított $\mathrm{DFT}_{n/2}$ kombinálását végezzük. Az $y_0, y_1, \ldots, y_{n/2-1}$ értékekből kiindulva a 11. sor alapján

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) && \text{((30.9) szerint).} \end{aligned}$$

Kiindulva az $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$ számokból, $k = 0, 1, \ldots, n/2 - 1$ esetén a 12. sorban azt kapjuk, hogy

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} - \omega_n^{k+(n/2)} y_k^{[1]} && (\omega_n^{k+(n/2)} = -\omega_n^k \text{ miatt}) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && (\omega_n^{2k} = -\omega_n^{2k+n} \text{ alapján}) \\ &= A(\omega_n^{k+(n/2)}) && \text{((30.9) szerint).} \end{aligned}$$

Ezzel megmutattuk, hogy a Rekurzív-FFT algoritmussal kapott y vektor valóban a bemenő a vektornak a DFT-je.

A 10–13. sorban lévő **for** ciklusban az $y_k^{[1]}$ számokat $k = 0, 1, \ldots, n/2-1$ esetén megszoroztuk ω_n^k-val. A szorzatot egyrészt hozzáadtuk $y_k^{[0]}$-hoz, másrészt levontuk $y_k^{[0]}$-ból. Mivel az ω_n^k faktort és annak -1-szeresét is alkalmazzuk, azért ezt ***forgatási tényezőnek*** is nevezik.

A Rekurzív-FFT algoritmus futási idejének meghatározásához vegyük figyelembe, hogy a rekurzív hívásokat nem számítva, a többi utasítás $\Theta(n)$ időt igényel, ahol n az input vektor hosszát jelenti. A futási időre tehát a következő rekurziót kapjuk:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

Ezzel megmutattuk, hogy bármely n fokszámkorlátú polinom az n-edik egységgyököknek megfelelő pontokban FFT alkalmazásával $\Theta(n \lg n)$ idő alatt kiértékelhető.

Interpoláció a komplex egységgyökökön

A polinomok szorzására szolgáló eljárás sémájának ismertetését annak megmutatásával fejezzük be, hogyan lehet az interpolációs feladatot a komplex egységgyökökön megoldani, lehetővé téve a pontreprezentációról az együttható-reprezentációra való áttérést. Az interpolációs feltételeket egy DFT-nek megfelelő lineáris egyenletrendszerrel írjuk le, majd megvizsgáljuk az itt fellépő mátrix inverzét.

A (30.4) egyenlet alapján a DFT felírható $y = V_n a$ alakban, ahol V_n az ω_n hatványaiból készített Vandermonde-mátrix:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

A V_n mátrix (k, j) indexű eleme ω_n^{kj} $(j, k = 0, 1, \ldots, n-1)$. Maga a V_n egy szorzótáblának is felfogható.

Az inverz művelet eredményét, azaz a V_n mátrix V_n^{-1} inverzének és az y vektornak a szorzatát az $a = \mathrm{DFT}_n^{-1}(y)$ szimbólummal jelöljük.

30.7. tétel. *Tetszőleges $j, k = 0, 1, \ldots, n-1$ számra a V_n^{-1} mátrix (j, k) indexű eleme az ω_n^{-kj}/n komplex számmal egyenlő.*

Bizonyítás. Megmutatjuk, hogy $V_n^{-1} V_n = I_n$, ahol I_n az $n \times n$-es egységmátrixot jelöli. Írjuk fel a $V_n^{-1} V_n$ mátrix (j, j') indexű elemét:

$$\begin{aligned} [V_n^{-1} V_n]_{jj'} &= \sum_{k=0}^{n-1} \frac{\omega_n^{-kj}}{n} \omega_n^{kj'} \\ &= \sum_{k=0}^{n-1} \frac{\omega_n^{k(j'-j)}}{n}. \end{aligned}$$

Az összegezési formula (a 30.6 lemma) alapján a fenti összeg 1-gyel egyenlő, ha $j = j'$, és 0 különben. Mivel $-(n-1) < j' - j < n-1$, azért a második esetben $j' - j$ nem osztható n-nel, s így az összegezési formula alapján valóban nullát kapunk. ■

A V_n^{-1} inverz mátrix alakját felhasználva, $\mathrm{DFT}_n^{-1}(y)$ felírható

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \quad (j = 0, 1, \ldots, n-1) \tag{30.11}$$

alakban. Összehasonlítva a (30.8) és (30.11) egyenleteket látható, hogy az FFT algoritmusban az a és y szerepét felcserélve, továbbá ω_n helyett ω_n^{-1}-et írva és végül minden koordinátát n-nel elosztva a DFT inverzét kapjuk (lásd még a 30.2-4. gyakorlatot). Innen következik, hogy a $\mathrm{DFT}_n^{-1}(y)$ kiszámítása szintén $\Theta(n \lg n)$ idő alatt végezhető el.

Ezzel megmutattuk, hogy az FFT és inverze alkalmazásával az n fokszámkorlátú polinomok esetén a pontreprezentációról az együttható-reprezentációra és vissza való áttéréshez, illetve ennek fordítottjához $\Theta(n \lg n)$ idő szükséges. Polinomok szorzására vonatkozóan pedig a következőt igazoltuk.

30.8. tétel (konvolúciós tétel). *Legyen n kettőhatványa, továbbá a és b két n dimenziós vektor. Ekkor*

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)),$$

ahol az a és b vektorokat 0 koordináták hozzávételével kiegészítettük $2n$-dimenziós vektorokká, és $\cdot$ jelöli a $2n$ dimenziós vektorok koordinátánkénti szorzását.

Gyakorlatok

30.2-1. Igazoljuk a 30.4. következményt.

30.2-2. Határozzuk meg a $(0, 1, 2, 3)$ vektor DFT-jét.

30.2-3. Oldjuk meg a 30.1-1. gyakorlatot a $\Theta(n \lg n)$ séma alkalmazásával.

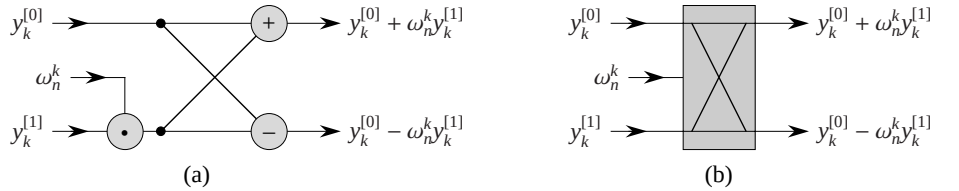
30.2-4. Írjunk pszeudokódot, amellyel a $\mathrm{DFT}_n^{-1}(y)$ vektor $\Theta(n \lg n)$ idő alatt számítható ki.

30.2-5. Általánosítsuk az FFT algoritmust arra az esetre, amikor n 3-nak egy hatványa. Adjunk egy rekurziós formulát a futási időre és oldjuk meg azt.

30.2-6. ⋆ Tegyük fel, hogy a komplex n dimenziós vektorokra vonatkozó FFT algoritmusban (ahol n páros szám) a komplex számtest helyett az egészek modulo m-vett $\mathbf{Z}_m$ gyűrűjéből indulunk ki, ahol $m = 2^{tn/2} + 1$ és t tetszőleges pozitív egész. Az ω_n alap n-edik gyök helyett vegyük a $w = 2^t$ számot. Igazoljuk, hogy a DFT és annak inverze értelmezhető ebben a rendszerben.

30.2-7. Adott $z_0, z_1, \ldots, z_{n-1}$ számokhoz (amelyek közül egyesek esetleg többször is előfordulhatnak) határozzuk meg annak a legfeljebb n-edfokú P polinomnak az együtthatóit, amelynek a $z_0, z_1, \ldots, z_{n-1}$ számok a gyökei (megfelelő multiplicitással). Írjunk olyan algoritmust, amelynek futási ideje $O(n \lg^2 n)$. (*Útmutatás.* A P polinomnak a z_j szám pontosan akkor gyöke, ha az $\ell(x) = x - z_j$ $(x \in \mathbf{C})$ lineáris polinom a P-nek maradék nélküli osztója.)

30.2-8. ⋆ Az $a = (a_0, a_1, \ldots, a_{n-1})$ vektor ***ciripelő transzformáltján*** az $y_k = \sum_{j=0}^{n-1} a_j z^{jk}$ $(k = 0, 1, \ldots, n-1)$ vektort szokás érteni, ahol z egy komplex szám. A DFT tehát a $z = \omega_n$ speciális esetnek megfelelő ciripelő transzformáció. Mutassuk meg, hogy a ciripelő transzformáció bármely z komplex szám esetén $O(n \lg n)$ idő alatt elvégezhető. (*Útmutatás.* Az



30.3. ábra. A pillangó művelet. **(a)** A két input adatot balról visszük be, majd az ω_n^k-t megszorozzuk $y_k^{[1]}$-gyel és az összeg, illetve különbség képezi jobbról a kimenetet. **(b)** A pillangó művelet egyszerűsített rajza. Ezt a reprezentációt a párhuzamos FFT algoritmusban fogjuk használni.

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2})(z^{-(k-j)^2/2})$$

azonosságból kiindulva, a ciripelő transzformációt fogjuk fel konvolúciónak.)

30.3. Az FFT egy hatékony megvalósítása

Mivel a DFT gyakorlati alkalmazásaiban, mint például a jelfeldolgozásban, a lehető legnagyobb sebességre van szükség, azért ebben a fejezetben az FFT két hatékony megvalósítását elemezzük. Először az FFT-nek egy iteratív változatát vizsgáljuk. Itt a futási idő nagyságrendje szintén $\Theta(n \lg n)$, azonban a Θ jelölésben rejtőző konstans kisebb, mint a 30.2. alfejezetben tárgyalt rekurzív implementáció esetében. Ezután az iteratív változatban alkalmazott ötletet felhasználjuk egy hatékony párhuzamos FFT algoritmus tervezéséhez.

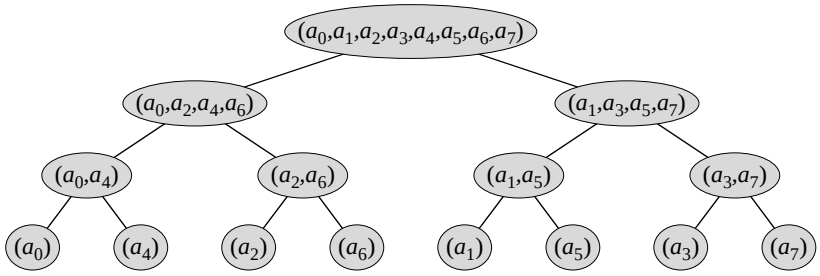
Az FFT egy iteratív megvalósítása

Először is megjegyezzük, hogy az ITERATÍV-FFT algoritmus **for** ciklusának 10–13. soraiban az $\omega_n^k y_k^{[1]}$ szorzatot kétszer számítjuk ki. Fordítóprogramokkal kapcsolatos terminológiát használva ezt az értéket ***közös alkifejezésnek*** szokták nevezni. A szóban forgó szorzatot egy új, ideiglenes változóban tárolva, a ciklus egyszerű módosításával elérhető, hogy ezt a szorzatot csak egyszer számítsuk ki:

```
1  for k ← 0 to n/2 − 1
2      do t ← ω y_k^[1]
3         y_k ← y_k^[0] + t
4         y_{k+(n/2)} ← y_k^[0] − t
5         ω ← ω ω_n
```

Ebben a ciklusban megvalósított számítást, nevezetesen az az $y_k^{[1]}$-nek $\omega\omega_n^k$-val való szorzását, ennek t-ben való tárolását, valamint a t és $y_k^{[0]}$ összegének és különbségének előállítását ***pillangó műveletnek*** szokás nevezni. Ennek a sémáját a 30.3. ábrán szemléltetjük.

Most megmutatjuk, hogyan lehet rekurzió helyett inkább iterációt alkalmazni az FFT-ben. A 30.4. ábrán a REKURZÍV-FFT hívása során alkalmazott bemenő vektorokat fa struktúrában helyeztük el, kiindulva az $n = 8$-nak megfelelő vektorból. Az algoritmus minden egyes hívásának a gráf egy-egy pontja felel meg, amelyet a megfelelő bemenő vektorral



30.4. ábra. A REKURZÍV-FFT algoritmus 6 hívása során alkalmazott input vektorok fa struktúrája. Az ábra az $n = 8$ esetnek felel meg.

címkéztünk. A REKURZÍV-FFT minden egyes hívása két rekurzív hívást jelent, mindaddig, amíg egydimenziós vektorhoz nem jutunk. A végrehajtás során el őször a bal oldali, másodszor a jobb oldali élt hívjuk.

Kiindulva a bemenő vektoroknak a fenti fa levelein látható rendezéséb ől, a REKURZÍV-FFT program végrehajtását a következ őképpen szemléltethetjük. Első lépésként vegyük a koordinátákból képzett párokat és a pillangó művelet alkalmazásával számítsuk ki a párok DFT-ját, majd a párokat helyettesítsük DFT-jukkal. Ezzel olyan vektort kaptunk, amely $n/2$-számú 2 komponensű DFT-t tartalmaz. Ezután az $n/2$ számú DFT-t ismét párokba soroljuk és két pillangó művelet végrehajtásával kiszámítjuk a 4 komponensű vektorok DFT-ját, két 2 komponensű DFT helyére az így kapott 4 komponensű DFT-t írva. A vektor most $n/4$ számú, egyenként 4-komponensű DFT-ból áll. Ezt az eljárást addig folytatjuk, amíg két, egyenként $(n/2)$ komponensb ől álló vektort kapunk. Ebb ől $(n/2)$ pillangó művelet alkalmazásával megkaphatjuk a végleges n koordinátát tartalmazó DFT-t.

Ahhoz, hogy a fenti észrevételt kódolhassuk, olyan $A[0 \ldots n-1]$ vektorból indulunk ki, amelyben a koordináták sorrendje megegyezik a 30.4. ábra leveleiben el őforduló koordináták sorrendjével. (Kés őbb megmutatjuk, hogyan adható meg ez a sorrend.) Ahhoz, hogy a fa különböző szintjein végzett műveleteket összekapcsolhassuk, bevezetünk egy s változót, amellyel a szinteket számozzuk 1-t ől (az aljától kiindulva, ahol a 2 komponensű DFT párokat állítottuk el ő) $\lg n$-ig (a tetejéig, ahol a két $(n/2)$ komponensű DFT-ból el őállítottuk a végeredményt). Így az algoritmus szerkezete a következ ő lesz:

```
1  for s ← 1 to lg n
2      do for k ← 0 to n − 1by 2^s
3          do az A[k..k2^(s−1) − 1] és A[k + 2^(s−1), ..k + 2^s − 1] vektorokban
               tárolt 2^(s−1) komponensű DFT-akból egy A[k..k + 2^s − 1]
               vektorban tárolt 2^s komponensű DFT-t készítünk
```

A ciklus magját (a 3. sort) ennél a pszeudokódnál részletesebben is megfogalmazhatjuk. Másoljuk át a REKURZÍV-FFT **for** ciklusát, az $y^{[0]}$ vektort az $A[k \ldots k2^{s-1}]$ vektorral, az $y^{[1]}$-et pedig az $A[k + 2^{s-1}, \ldots k2^s - 1]$ vektorral helyettesítve. Az egyes pillangó műveletekben el őforduló ω értéke az s értékét ől függ, ezt az ω_m jelölés használatával juttatjuk kifejezésre, ahol $m = 2^s$. (Az m változót egyedül az olvashatóság érdekében vezettük be.) Bevezetünk egy u-val jelölt, további ideiglenes változót is, amely lehet ővé teszi, hogy a pillangó műveletet egy helyben végezzük el. Ha a fenti általános algoritmus 3. sorát a ciklus magjával

helyettesítjük, olyan pszeudokódot kapunk, amely mind a végleges iteratív FFT algoritmusnak, mind pedig annak párhuzamos változatának az alapját képezi. Ez utóbbit kés őbb ismertetjük. Az elsőnek hívott BIT-FORDÍTÓ-MÁSOLÓ(a, A) szubrutin az a vektort átrendezi és átrakja A-ba.

ITERATÍV-FFT(a)

```
 1  BIT-FORDÍTÓ-MÁSOLÓ(a, A)
 2  n ← hossz[a] ▷ n kettőhatványa
 3  for s ← 1 to lg n
 4      do m ← 2^s
 5         ω_m ← e^(2πi/m)
 6         for k ← 0 to n − 1 by m
 7             do  ω ← 1
 8                 for j ← 0 to m/2 − 1
 9                     do t ← ωA[k + j + m/2]
10                        u ← A[k + j]
11                        A[k + j] ← u + t
12                        A[k + j + m/2] ← u − t
13                        ω ← ωω_m
```

Nézzük meg, hogyan kapjuk meg az a vektor koordinátáit a kívánt sorrendben a BIT-FORDÍTÓ-MÁSOLÓ(a, A) rutint felhasználva! Az a sorrend, amelyben a koordinátáka 30.4. ábra leveleiben előfordulnak, a „bináris bitfordításnak” felel meg. Legyen ford(k) az a lg n darab bitből képzett szám, amelyet a k szám bináris reprezentációjából úgy kapunk, hogy annak bitjeit fordított sorrendben írjuk fel. Az a vektor a_k koordinátáját az A tömb $A[(k)]$ helyébe írjuk. A 30.4. ábrán az alsó szinten a koordináták sorrendje: 0, 4, 2, 6, 1, 5, 3, 7. Ezeknek a számoknak a bináris alakja: 000, 100, 010, 110, 001, 101, 011, 111. A bitek sorrendjét megfordítva innen 000, 001, 010, 011, 100, 101, 110, 111, azaz az eredeti sorrend adódik. Annak igazolásához, hogy a bitfordító permutáció a kívánt sorrendet adja, elegend ő észrevenni a következőket. A fa legfelső szintjén a bal oldali levélen azok az indexek szerepelnek, amelyek alsó bitje 0-val egyenlő, a jobb oldali levélen pedig azok, amelyek alsó bitje 1-gyel egyenlő. Mindegyik levélen az alacsony és magas bitek szerint szétválasztva a komponenseket folytatjuk ezt az eljárást, és végül a bitek fordított sorrendjének megfelel ő átrendezést kapjuk.

Mivel a (k) függvény értékei könnyen kiszámíthatók, a BIT-FORD ÍTÓ-MÁSOLÓ algoritmust a következő rövidített formában adjuk meg.

BIT-FORDÍTÓ-MÁSOLÓ(a,A)

```
1  n ← hossz[a]
2  for k ← 0 to n − 1
3      do A[(k)] ← a_k
```

Az iteratív FFT algoritmus $\Theta(n \lg n)$ idő alatt fut le. A BIT-FORDÍTÓ-MÁSOLÓ(a, A) algoritmus összesen $O(n \lg n)$ időt használ fel, mivel a szóban forgó algoritmust a futás során n-szer hívjuk, és egy 0 és $n-1$ közé eső, lg n bittel rendelkező egész bitfordítottjának előállításához $O(\lg n)$ idő szükséges. (A gyakorlatban általában el őre ismerjük az n számú kezdeti adatot.

Célszerű a $k \to (k)$ leképezésre egy táblázatot készíteni, s ezzel a BIT-FORDÍTÓ-MÁSOLÓ rutin futási idejét $\Theta(n)$-re lehet csökkenteni, ahol a nagyságrendi konstans kicsi. Ehelyett alkalmazhatjuk a 17-1. feladatban leírt fordított bináris számláló algoritmust is.) Befejezve a bizonyítást, és az ITERATÍV-FFT algoritmus futási idejét $L(n)$-nel jelölve megmutatjuk, hogy $L(n) = \Theta(n \lg n)$ teljesül. A belső ciklus magjának (8–13. sor) végrehajtásához $\Theta(n \lg n)$ idő szükséges. A 6–13. sorban lévő **for** ciklusban az s minden értékére $n/m = n/2^s$, a 8–13. sorban lévő legbelső ciklusban $m/2 = 2^{s-1}$ iterációt hajtunk végre. Következésképpen

$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n). \end{aligned}$$

Egy párhuzamos FFT áramkör

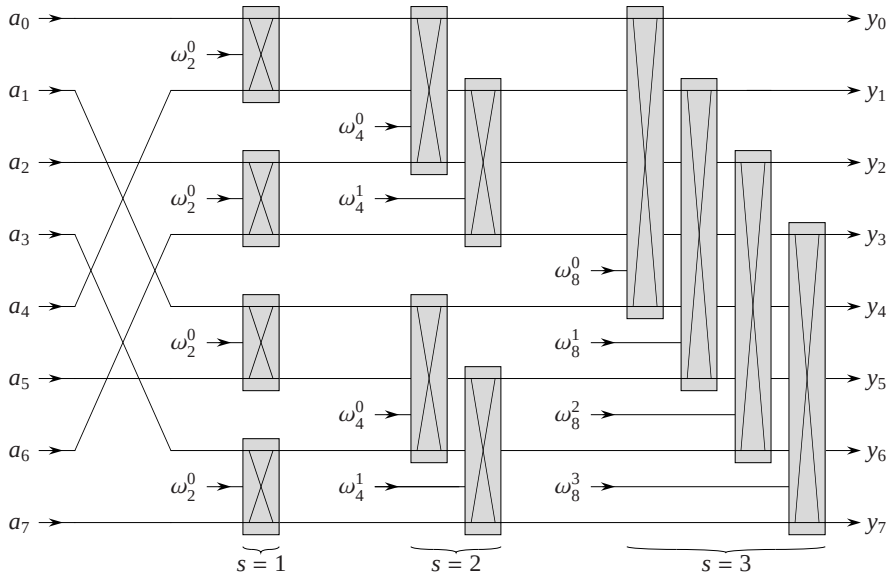
A hatékony iteratív FFT algoritmus lépései közül több is hasznosítható hatékony párhuzamos FFT algoritmusok szerkesztéséhez. (A párhuzamos FFT algoritmust olyan áramkörként adjuk meg, amely hasonló a 27. fejezetben leírt összehasonlító hálózathoz.) Komparátorok helyett az FFT ciklusaiban pillangó műveleteket használunk a 30.3(b) ábrának megfelelően. A mélység fogalmát, amelyet a 27. fejezetben vezettünk be, itt is felhasználjuk. Az n input adattal működő PÁRHUZAMOS-FFT logikai diagramját a 30.5. ábrán $n = 8$ esetén szemléltetjük. A diagram a bemenő adatoknak egy bitfordító permutációjával kezdődik, ezt $\lg n$ számú szint követi, amelyek mindegyike $n/2$ számú, párhuzamosan végrehajtott pillangó műveletből áll. Ennek megfelelően a diagram mélysége $\Theta(\lg n)$.

A PÁRHUZAMOS-FFT algoritmus diagramjának bal oldali része a bitfordító permutációt hajtja végre, a maradék rész pedig az iteratív ITERATÍV-FFT áramkör szerint működik. Itt kihasználjuk azt a lehetőséget, hogy a külső **for** ciklus minden egyes iterációja $n/2$ független pillangó műveletet tartalmaz, amelyek párhuzamosan végezhetők el. Az ITERATÍV-FFT programon belül végrehajtott iterációk során az s változó értéke a 30.5. ábrán bemutatott pillangó műveletek eredményétől függ. Minden $s = 1, 2, \ldots, \lg n$ esetén $n/2^s$ számú, pillangó műveletből álló csoport van (az ITERATÍV-FFT-ban szereplő k minden egyes értékének megfelelően), továbbá minden csoportban (az ITERATÍV-FFT-ben szereplő j értékeinek megfelelően) 2^{s-1} pillangó műveletet hajtunk végre. A 30.5. ábrán szereplő pillangók a belső ciklusban lévő (az ITERATÍV-FFT 9–12. sorában feltüntetett) pillangó műveleteknek felelnek meg. Megjegyezzük, hogy a pillangó műveletekben alkalmazott ω tényezők megfelelnek az ITERATÍV-FFT-ben használt tényezőknek: az s szinten ezek értéke $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, ahol $m = 2^s$.

Gyakorlatok

30.3-1. Szemléltessük a DFT kiszámítását az ITERATÍV-FFT alkalmazásával, kiindulva a $(0, 2, 3, -1, 4, 5, 7, 9)$ vektorból.

30.3-2. Hogyan módosul az FFT algoritmusa, ha a bitfordító permutációt nem az algoritmus kezdetén, hanem a végén alkalmazzuk. (*Útmutatás.* Alkalmazzuk az inverz DFT-t.)



30.5. ábra. Itt az FFT kiszámítására szolgáló PÁRHUZAMOS-FFT áramkört az $n = 8$-nak megfelelő bemenet esetén szemléltetjük. Mindegyik pillangó művelet bemenete két huzalon felvett érték, valamint egy forgatási tényező, kimenete pedig a két huzalon felvett érték. A pillangók szintjét úgy címkézzük, hogy megfeleljen az ITERATÍV-FFT eljárás legkülső ciklusában lévő iterációnak. Csak az alsó és a felső szinten lévő huzaloknak van a pillangókkal kölcsönhatása; azok a huzalok, amelyek a pillangó közepén haladnak át, nem hatnak a pillangóra és a pillangó sem változtatja meg értéküket. Például a 2-es szinten tetején lévő pillangó független az (y_1-gyel jelölt) 1-es huzaltól; inputjai és outputjai csak (az y_0-val, illetve y_2-vel jel ölt) 0-ás és 2-es huzalon vannak. Egy n bemenetű FFT $\Theta(\lg n)$ mélységű hálózattal $\Theta(n \lg n)$ pillangó művelettel számítható ki.

30.3-3. Mennyi idő szükséges az ITERATÍV-FFT algoritmus egy-egy állapotában a forgatási tényezők kiszámításához. Írjuk át az ITERATÍV-FFT algoritmust úgy, hogy az s-edik állapotban a forgatási tényezők kiszámításához csak 2^{s-1} időt használjunk fel.

30.3-4. ⋆ Tegyük fel, hogy az FFT áramkör pillangó műveletében az összeadást végző egység néha hibás és a bemenettől függetlenül mindig 0 eredményt szolgáltat. Tegyük fel, hogy pontosan egy ilyen hibás egység van, de nem tudjuk, hogy melyik az. Adjunk egy eljárást a hibás egység felkutatására, az FFT áramkörre adott alkalmas bemenetet és az ezekre adott kimenő jeleket felhasználva. Keressünk hatékony eljárást.

Feladatok

30-1. Oszd-meg-és-uralkodj elvű szorzás

a. Mutassuk meg, hogy az $ax + b$ és $cx + d$ elsőfokú polinomok szorzata három szorzással előállítható. (*Útmutatás.* Használjuk fel az $(a + b) \cdot (c + d)$ szorzatot.)

b. Adjunk két olyan oszd-meg-és-uralkodj elvű algoritmust n fokszámkorlátú polinomok szorzására, amelyek futási ideje $\Theta(n^{\lg 3})$. Mindkét algoritmusban a polinom együtthatóit két csoportra osztjuk, az első algoritmusban az alacsony és magas indexű együtthatókat, a másodikban pedig a páros és páratlan indexű együtthatókat különítve el egymástól.

c. Mutassuk meg, hogy két n bites egész szorzása elvégezhető $O(n \lg n)$ lépésben úgy, hogy minden egyes lépésben legfeljebb konstans számú 1 bites számot használunk fel.

30-2. Toeplitz mátrixok

Az $A = (a_{ij})$ mátrixot ***Toeplitz-féle mátrixnak*** nevezzük, ha $a_{ij} = a_{i-1,j-1}$ ($i = 2, 3, \ldots, n$, $j = 2, 3, \ldots, n$).

a. Igaz-e, hogy két Toeplitz mátrix összege is Toeplitz mátrix? Mit mondhatunk szorzatukról?

b. Hogyan reprezentáljuk az $n \times n$-es Toeplitz-mátrixokat, hogy összegüket $O(n)$ idő alatt lehessen kiszámítani?

c. Adjunk olyan algoritmust, amellyel egy $n \times n$-es Toeplitz-mátrix és egy n-dimenziós vektor szorzata $O(n \lg n)$ idő alatt kiszámítható. Alkalmazzuk a (b) részben használt reprezentációt.

d. Adjunk hatékony algoritmust két $n \times n$-es Toeplitz-mátrix szorzatának előállítására. Vizsgáljuk a futási időt.

30-3. Többdimenziós gyors Fourier-transzformáció

A (30.8) képlettel értelmezett egydimenziós diszkrét Fourier-transzformált általánosítható a d-dimenziós esetre. Ebben az esetben a bemenet egy $A = (a_{i_1,i_2,\ldots,i_d})$ d dimenziós tömb, ahol $0 \leqq i_j < n_j$, $(j = 0, 1, \ldots, d)$ és $n_1 n_2 \cdots n_d = n$. A d-dimenziós diszkrét Fourier-transzformációt az

$$y_{k_1,k_2,\ldots,k_d} = \sum_{i_1=0}^{n_1-1} \sum_{i_2=0}^{n_2-1} \cdots \sum_{i_d=0}^{n_d-1} a_{i_1,i_2,\ldots,i_d} \omega_{n_1}^{i_1 k_1} \omega_{n_2}^{i_2 k_2} \cdots \omega_{n_d}^{i_d k_d}$$

egyenletekkel értelmezzük, ahol $0 \leqq k_1 < n_1, 0 \leqq k_2 < n_2, \ldots, 0 \leqq k_d < n_d$.

a. Mutassuk meg, hogy a d-dimenziós DFT kiszámítása visszavezethető a koordinátánként vett egydimenziós DFT-ra. Azaz először az első változóra alkalmazva az egydimenziós DFT-t kiszámítjuk az összesen n/n_1 számú diszkrét Fourier-transzformáltat. Ezeket bemenetként felhasználva és a második változóban alkalmazva az egydimenziós DFT-t n/n_2 számú DFT-t határozunk meg. Ezt az eljárást folytatva végül a d-edik változóban elvégezve a DFT-t megkapjuk a végeredményt.

b. Mutassuk meg, hogy a fenti algoritmusban a változók sorrendje nem számít, azaz a számítást a változók bármely sorrendjében elvégezve ugyanazt az eredményt kapjuk.

c. Mutassuk meg, hogy az egydimenziós DFT-t FFT-algoritmussal számolva a d-dimenziós DFT ideje a d-től függetlenül $O(n \lg n)$.

30-4. Polinomok összes deriváltjának kiszámítása adott pontban

Adott n fokszámkorlátú $A(x)$ polinom t-edik deriváltját az

$$A^t(x) = \begin{cases} A(x), & \text{ha } t = 0, \\ \frac{d}{dx} A^{t-1}(x), & \text{ha } 1 \leq t \leq n-1, \\ 0, & \text{ha } t \geq n \end{cases}$$

módon definiáljuk.

$A(x)$ polinom $(a_0, a_1, \ldots, a_{n-1})$ együttható-reprezentációjából és egy adott x_0 pontból kiindulva meg akarjuk határozni $A^t(x_0)$-t a $t = 0, 1, \ldots, n-1$ indexekre.

a. Kiindulva azokból a $b_0, b_1, \ldots, b_{n-1}$ együtthatókból, melyekre

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j,$$

adjunk meg olyan eljárást, amellyel az $A^{(t)}(x_0)$ $(t = 0, 1, \ldots, n-1)$ deriváltak $O(n)$ idő alatt kiszámíthatók.

b. Adjunk $O(n \lg n)$ futási idejű eljárást a $b_0, b_1, \ldots, b_{n-1}$ együtthatók kiszámítására, kiindulva az $A(x_0 + \omega_n^k)$ $(k = 0, 1 \ldots, n-1)$ függvényértékekből.

c. Igazoljuk, hogy

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \Big(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r - j) \Big),$$

ahol $f(j) = a_j \cdot j!$ és

$$g(\ell) = \begin{cases} \frac{x_0^{-\ell}}{(-\ell)!}, & \text{ha} \quad -(n-1) \le \ell \le 0, \\ 0, & \text{ha} \quad 1 \le \ell \le n-1. \end{cases}$$

d. Mutassuk meg, hogy az $A(x_0 + \omega_n^k)$ $(k = 0, 1, \ldots, n-1)$ függvényértékek $O(n \lg n)$ idő alatt számíthatók ki. Ennek alapján mutassuk meg, hogy az A nemtriviális deriváltjai az x_0 pontban $O(n \lg n)$ idő alatt számíthatók ki.

30-5. Polinom kiértékelése több pontban

Korábban láttuk, hogy egy n fokszámkorlátú polinomnak egy pontban való kiértékelése a Horner-féle elrendezéssel $O(n)$ idő alatt megoldható. Azt is megmutattuk, hogy ugyanennek a polinomnak az n-edik komplex egységgyökökben való kiértékelése az FFT algoritmust alkalmazva $O(n \lg n)$ idő alatt elvégezhető. Most megmutatjuk, hogy a legfeljebb n-edfokú polinomoknak tetszőleges n helyen való kiértékelése $O(n \lg^2 n)$ idő alatt lehetséges.

Ennek igazolásához bizonyítás nélkül felhasználjuk azt a tényt, hogy két ilyen polinomot egymással való osztásakor kapott maradékot $O(n \lg n)$ idő alatt ki tudjuk számítani. Például a $3x^3 + x^2 - 3x + 1$ polinomnak az $x^2 + x + 2$ polinommal való osztásával kapott maradék:

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5.$$

Kiindulva az $A(x) = \sum_{k=0}^{n-1} a_k x^k$ polinom együttható-reprezentációjából és az $x_0, x_1, \ldots, x_{n-1}$ pontokból, meghatározzuk az $A(x_0), A(x_1), \ldots, A(x_{n-1})$ értékeket. Tetszőleges $0 \le i \le j \le n-1$ indexekre értelmezzük a $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ és $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$ polinomokat. Vegyük figyelembe, hogy Q_{ij} $(j - i)$-edfokú.

a. Mutassuk meg, hogy minden z esetén $A(x) \bmod (x - z) = A(z)$.

b. Igazoljuk, hogy $Q_{k,k}(x) = A(x_k)$ és $Q_{0,n-1}(x) = A(x)$.

c. Igazoljuk, hogy minden $i \le k \le j$ esetén $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ és $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Adjunk egy $O(n \lg^2 n)$ futási idejű algoritmust az $A(x_0), A(x_1), \ldots, A(x_{n-1})$ kiszámítására.

30-6. Moduláris aritmetikán alapuló FFT algoritmus

A diszkrét Fourier-transzformációban komplex számokat használunk, s az ezekkel való műveletek a valóságban kerekítési hibákat eredményeznek. Bizonyos feladatok kapcsán, amelyekben az eredmény egész szám, felvetődik annak igénye, hogy az FFT-nek egy moduláris aritmetikán alapuló változatát használjuk s ezzel garantálni tudjuk, hogy a végeredmény pontos lesz. Ilyen feladat pl. az egész együtthatós polinomok szorzása. A 30.2-6. gyakorlatban egy olyan eljárást ismertettünk, amelyben $\Omega(n)$ bit hosszúságú modulust használunk n pontos DFT kezelésére. Most a problémának egy másfajta megközelítését adjuk, amelyben az inkább indokolható, $O(\lg n)$ hosszúságú modulust használjuk; ehhez szükséges a 31. fejezet anyagának a megértése. Legyen n 2-nek egy hatványa.

a. Először is határozzuk meg azt a legkisebb k számot, amelyre $p = kn + 1$ prímszám. Adjunk egy heurisztikus indokolást arra, hogy k értéke közelítőleg $\lg n$ lesz. (A k értéke ennél jóval nagyobb vagy jóval kisebb is lehet, de átlagban $O(\lg n)$ számú értéket kell megvizsgálni k értékeként szóba jövő számok közül.) Hogyan viszonylik a p várható hossza az n hosszához?

Jelölje g a $\mathbf{Z}_p^*$ egy generátorát és legyen $w = g^k \bmod p$.

b. Mutassuk meg, hogy a DFT és az inverz DFT jól definiált műveletek és egymás inverzei maradnak, ha ezekben az alap n-edik egységgyököt a w-vel helyettesítjük.

c. Mutassuk meg, hogy az FFT és annak inverze elvégezhető $O(n \lg n)$ időben modulo p számolva, ha az $O(\lg n)$ bit hosszúságú szavakon a műveletek egységnyi időt vesznek igénybe. Tegyük fel, hogy az algoritmus a p és a w paraméterekkel adott.

d. Számítsuk ki a $(0, 5, 3, 7, 7, 2, 1, 6)$ vektor DFT-ját modulo $p = 17$. Vegyük figyelembe, hogy $g = 3$ a $\mathbf{Z}_{17}^*$ egy generátora.

Megjegyzések a fejezethez

Van Loan [303] könyvében a gyors Fourier-transzformáció egy kitűnő feldolgozását adja. Press, Flannery, Teukolsky és Vetterling [248, 249] a gyors Fourier-transzformáció és alkalmazásainak egy jó leírását adják. A jelfeldolgozás az FFT-nek egyik népszerű alkalmazási területe, s ehhez egy kiváló bevezetést nyújt Oppenheim és Schafer [232], valamint Oppenheim és Willsky [233] műve. Ez utóbbi könyvben megmutatják, hogyan kezelhetők azok az esetek, ahol az n nem kettőhatvány.

A Fourier-analízis nem csak egydimenziós adatokra alkalmazható. A képfeldolgozásban általában két- vagy többdimenziós adatokat használunk. Gonzalez és Woods [127], valamint Pratt [246] könyvükben többdimenziós Fourier-transzformációval és ennek alkalmazásaival foglalkoznak a képfeldolgozással kapcsolatosan. Tolimieri, An és Lu [300], valamint Van Loan [303] könyve a többváltozós gyors-Fourier transzformáció matematikai alapjait tárgyalja.

Cooley és Tukey [68] érdeme, hogy az 1960-as években rátaláltak az FFT-re. Valójában az FFT-t már sokkal korábban felfedezték, ezt azonban a modern digitális számítógépek megjelenése előtt nem tudták kihasználni. Press, Flannery, Teukolsky és Vetterling a módszert Rungenek és Kőnignek (1924) tulajdonítja. Heideman, Johnson és Burrus [141] az FFT történetét 1805-ig, Gauss munkásságára vezeti vissza.

31. Számelméleti algoritmusok

A számelmélet valaha az elméleti matematika szép, de jobbára haszontalan ágának számított. Napjainkban a számelméleti algoritmusokat széles körben használják, ami részben a nagy prímszámokon alapuló kriptográfiai eljárások elterjedésének köszönhető. Az eljárások alkalmazhatósága azon a képességünkön alapul, hogy könnyen tudunk nagy prímeket találni, miközben a titkosság garanciája az, hogy nagy prímek szorzatát képtelenek vagyunk tényezőkre bontani. Ez a fejezet az ilyen alkalmazások alapjául szolgáló számelméleti ismereteket és a kapcsolódó algoritmusokat tárgyalja.

A 31.1. alfejezet olyan számelméleti alapfogalmakat és tételeket vezet be, mint az oszthatóság, a kongruencia és a számelmélet alaptétele. A 31.2. alfejezet a világ egyik legrégebbi algoritmusát, a két egész szám legnagyobb közös osztóját kiszámító euklideszi algoritmust vizsgálja. A 31.3. alfejezet a maradékosztályok aritmetikáját tekinti át. A 31.4. alfejezet egy adott a szám többeseinek halmazát vizsgálja modulo n, és megmutatja, hogyan lehet megtalálni az $ax \equiv b \pmod{n}$ kongruencia összes megoldását az euklideszi algoritmus felhasználásával. A 31.5. alfejezet a kínai maradéktételt részletezi. A 31.6. alfejezet egy adott a szám hatványait tekinti modulo n, és egy iteratív négyzetre emelő algoritmust ad az $a^b \bmod n$ gyors kiszámítására rögzített a, b és n mellett. Ez a művelet a hatékony prímtesztelés és sok más modern titkosírás lelke. A 31.7. alfejezet az RSA nyilvános kulcsú titkosírást tárgyalja. A 31.8. alfejezet egy, a nagy prímek keresésénél hatékonyan működő valószínűségi prímtesztet mutat be, ami alapvető feladat az RSA titkosírás kulcsainak meghatározásához. Végül a 31.9. alfejezet egy egyszerű, de mégis hatékony heurisztikus módszert ismertet kis egész számok faktorizálására. Érdekes tény, hogy sokan (legalábbis azok, akik a titkosság megőrzésében érdekeltek) azt kívánják egy problémától, hogy megoldatlan maradjon. A faktorizáció problémája ilyen, mivel az RSA titkossága a nagy számok felbontásának nehézségére épül.

A bemenő adatok mérete és az aritmetikai számítások költsége

Mivel nagy számokkal fogunk dolgozni, meg kell állapodnunk abban, mit értsünk a bemenő adatok méretén és az elemi aritmetikai műveletek költségén.

Ebben a fejezetben a „nagy bemenő adat” „nagy egész számok” bevitelét jelenti, nem pedig „sok egész számét” (mint pl. a rendezésnél). Emiatt az input méretét most nem a bemenő egész adatok számával mérjük, hanem a bemenő adatok ábrázolásához szükséges bitek számával. Egy algoritmust ***polinomiális idejű algoritmusnak*** nevezünk, ha az

$a_1, a_2, \ldots, a_k$ egész bemenő adatokkal a futási idő $\lg a_1, \lg a_2, \ldots, \lg a_k$-ban polinomiális, azaz a futási idő a binárisan kódolt bemenő adatok hosszától polinomiálisan függ.

A könyv túlnyomó részében céljainknak kiválóan megfelel, ha az elemi aritmetikai műveleteket (szorzás, osztás vagy maradékosztály-műveletek) egységnyi időigényű egyszerű műveletnek tekintjük. Algoritmusaink végrehajtásához szükséges ilyen típusú aritmetikai műveletek száma szolgál majd alapul az algoritmusok tényleges számítógépes futási idejének ésszerű becsléséhez. Nagy bemenő adatok esetén azonban az elemi műveletek is időigényesek lehetnek, ezért szükség van a számelméleti algoritmusok ***bitműveleteinek*** mérésére is. Ebben a modellben két β bit hosszúságú egész szokásos módon való összeszorzása $\Theta(\beta^2)$ bitműveletet igényel. Hasonlóképp, egyszerű algoritmusok adhatók egy β bit hosszúságú egésznek egy nála kisebb számmal való osztására vagy az osztási maradékának meghatározására, amelyek időigénye szintén $\Theta(\beta^2)$ (31.1-11. gyakorlat). Ezeknél gyorsabb módszerek is ismeretesek. Például az egyszerű oszd-meg-és-uralkodj módszer felhasználásával két β bit hosszúságú egész szorzásának futási ideje $\Theta(\beta^{\lg 3})$, sőt a leggyorsabb ismert módszert alkalmazva a futási idő $\Theta(\beta \lg \beta \lg\lg \beta)$ nagyságrendű. Gyakorlati célokra azonban általában a $\Theta(\beta^2)$ futási idejű algoritmus a legalkalmasabb, elemzéseinkben mi is ezt vesszük alapul.

Ebben a fejezetben az algoritmusokat az igényelt aritmetikai és bitműveletek számával egyaránt jellemezni fogjuk.

31.1. Elemi számelméleti fogalmak

Ez az alfejezet rövid áttekintést ad az egész számok ($\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$) és a természetes számok ($\mathbf{N} = \{0, 1, 2, \ldots\}$) elemi számelméleti fogalmairól.

Oszthatóság és osztók

A számelméletben központi fogalom, hogy egy egész szám osztható-e egy másik egész számmal. A $d \mid a$ (olvasva: d ***osztja*** az a számot) jelölés azt jelenti, hogy valamilyen k egész számra $a = kd$ teljesül. Minden egész szám osztja a nullát. Ha $a > 0$ és $d \mid a$, akkor $|d| \leq |a|$. Ha $d \mid a$, akkor azt is szoktuk mondani, hogy a a d szám ***többszöröse*** vagy ***többese.*** Ha d nem osztja az a számot, akkor ezt $d \nmid a$ jelöli.

Ha $d \mid a$ és $d \geq 0$, akkor a d számot az a ***osztójának*** nevezzük. Megjegyezzük, hogy $d \mid a$ akkor és csak akkor teljesül, ha $-d \mid a$, vagyis az osztókat az általánosság megszorítása nélkül definiálhatnánk a nemnegatív számokra is, hozzátéve, hogy a bármelyik osztójának (-1)-szerese is osztja az a számot. Az a egész szám osztói nem kisebbek 1-nél, és nem nagyobbak $|a|$-nél. Például a 24 osztói az $1, 2, 3, 4, 6, 8, 12$ és 24 számok.

Minden a egész szám osztható ***triviális osztóival,*** 1-gyel és a-val. Az a nemtriviális osztóit az a ***tényezőinek*** vagy ***faktorainak*** nevezzük. Például a 20 tényezői a $2, 4, 5$ és 10 számok.

Prímszámok és összetett számok

Ha az $a > 1$ egész számnak csak a triviális 1 és az a az osztói, akkor az a számot ***prímszámnak*** (vagy egyszerűen ***prímnek***) nevezzük. A prímeknek sok speciális tulajdonságuk van és

különleges szerepet játszanak a számelméletben. Az első húsz prím növekvő sorrendben:

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ 53,\ 59,\ 61,\ 67,\ 71.$$

A 31.1-1. gyakorlat annak a bizonyítását kéri, hogy végtelen sok prím létezik. Ha az $a > 1$ egész szám nem prím, akkor ***összetett számnak*** (vagy egyszerűen ***összetettnek***) nevezzük. Például a 39 összetett, mert $3 \mid 39$. Az 1 számot ***egységnek*** nevezzük, és ez a szám se nem prím, se nem összetett. Hasonlóan, a 0 és a negatív számok sem prímek, sem összetettek.

A maradékos osztás tétele, maradékok és kongruenciák

Egy rögzített n pozitív egész számot véve az egészeket osztályozhatjuk aszerint, hogy azok az n többesei vagy sem. Sok számelméleti bevezetés ennek a felosztásnak a finomításán alapul: az n-nel nem osztható számokat az n-nel való osztási maradékuk szerint osztályozzuk. Az osztályozás létezését az alábbi tétel garantálja, amelyet itt nem bizonyítunk (a bizonyítás megtalálható például Niven–Zuckerman [231] könyvében).

31.1. tétel (maradékos osztás tétele). *Bármely a egész és n pozitív egész számhoz egyértelműen létezik olyan q és r egész, hogy $0 \le r < n$ és $a = qn + r$.*

A $q = \lfloor a/n \rfloor$ érték az osztás ***hányadosa,*** az $r = a \bmod n$ pedig az osztás ***maradéka.*** Tehát $n \mid a$ akkor és csak akkor teljesül, ha $a \bmod n = 0$.

Az egészeket ekvivalencia-osztályokba sorolhatjuk az n-nel való osztási maradékuk szerint. Az a egész számot tartalmazó ***ekvivalencia-osztályt*** vagy ***maradékosztályt modulo n*** az alábbi módon definiáljuk:

$$[a]_n = \{a + kn : k \in \mathbf{Z}\}.$$

Például $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$, vagy más jelöléssel $[-4]_7$, vagy $[10]_7$. Ezzel a jelöléssel $a \in [b]_n$ ugyanazt jelenti, mint a korábban definiált $a \equiv b \pmod{n}$. Az összes ekvivalencia-osztályt tartalmazó halmaz jelölése:

$$\mathbf{Z}_n = \{[a]_n : 0 \le a \le n - 1\}. \tag{31.1}$$

Gyakran találkozunk az alábbi definícióval is:

$$\mathbf{Z}_n = \{0, 1, \ldots, n - 1\}, \tag{31.2}$$

ami (31.1)-gyel ekvivalens azzal a megjegyzéssel, hogy a $[0]_n$ helyett a 0 áll, az 1 képviseli az $[1]_n$ maradékosztályt stb., azaz minden maradékosztályt a legkisebb nemnegatív eleme reprezentál. Mindamellett az alapul szolgáló ekvivalencia-osztályokat észben kell tartanunk. Például a -1 egész szám az $[n-1]_n$ maradékosztályt képviseli, hiszen $-1 \equiv n - 1 \pmod{n}$.

Közös osztó és legnagyobb közös osztó

Ha d osztja az a és a b egész számok mindegyikét, akkor a d számot az a és b számok ***közös osztójának*** nevezzük. Például a 30 osztói az 1, 2, 3, 5, 6, 10, 15 és 30 számok, így a 24 és 30 közös osztói az 1, 2, 3 és 6. Vegyük észre, hogy az 1 bármely két egésznek közös osztója.

A közös osztó egy fontos tulajdonsága, hogy

$$d \mid a \text{ és } d \mid b \Rightarrow d \mid (a+b) \text{ és } d \mid (a-b). \tag{31.3}$$

Sokkal általánosabb állítás is igaz, nevezetesen

$$d \mid a \text{ és } d \mid b \Rightarrow d \mid (ax+by) \tag{31.4}$$

bármely x, y egészre. Továbbá, ha $a \mid b$, akkor vagy $|a| \leq |b|$, vagy $b = 0$, így

$$a \mid b \text{ és } b \mid a \Rightarrow a = \pm b. \tag{31.5}$$

Ha a és b legalább egyike 0-tól különböző egész, akkor a és b közös osztói közül a legnagyobbat az a és b számok ***legnagyobb közös osztójának*** nevezzük, és $\text{lnko}(a,b)$-vel jelöljük. Például $\text{lnko}(24,30) = 6$, $\text{lnko}(5,7) = 1$ és $\text{lnko}(0,9) = 9$. Ha az a és b egészek legalább egyike nem 0, akkor $\text{lnko}(a,b)$ az 1 és $\min(|a|,|b|)$ egész számok közé esik. Megállapodás szerint $\text{lnko}(0,0) = 0$. Ez a definíció lehetővé teszi, hogy a lnko függvény tulajdonságait (például a (31.9) egyenlőséget) egységesen tárgyalhassuk.

A lnko függvény elemi tulajdonságai a következők:

$$\begin{aligned}
\text{lnko}(a,b) &= \text{lnko}(b,a), && (31.6)\\
\text{lnko}(a,b) &= \text{lnko}(-a,b), && (31.7)\\
\text{lnko}(a,b) &= \text{lnko}(|a|,|b|), && (31.8)\\
\text{lnko}(a,0) &= |a|, && (31.9)\\
\text{lnko}(a,ka) &= |a| \quad \text{minden } k \in \mathbf{Z}\text{-re}. && (31.10)
\end{aligned}$$

Az alábbi tétel a legnagyobb közös osztó egy alternatív és hasznos jellemzését adja.

31.2. tétel. *Ha az a és b egészek legalább egyike* 0*-tól különbözik, akkor* $\text{lnko}(a,b)$ *az a és b összes lineáris kombinációiból álló* $\{ax+by : x,y \in \mathbf{Z}\}$ *halmaz legkisebb pozitív eleme.*

Bizonyítás. Legyen s az a és b legkisebb pozitív lineáris kombinációja, és legyen $s = ax+by$ alkalmas $x, y \in \mathbf{Z}$ választással. Legyen továbbá $q = \lfloor a/s \rfloor$. A (31.8) egyenlőséget felhasználva

$$\begin{aligned}
a \bmod s &= a - qs \\
&= a - q(ax+by) \\
&= a(1-qx) + b(-qy),
\end{aligned}$$

vagyis $a \bmod s$ szintén az a és b lineáris kombinációja. De $0 \leq a \bmod s < s$, következésképp $a \bmod s = 0$, hiszen s a legkisebb ilyen pozitív lineáris kombináció. Emiatt $s \mid a$ és hasonló okoskodással $s \mid b$. Így s közös osztója a-nak és b-nek, ahonnan $\text{lnko}(a,b) \geq s$ következik. Ugyanakkor (31.4) miatt $\text{lnko}(a,b) \mid s$, mivel $\text{lnko}(a,b)$ osztja a és b mindegyikét és s az a és b lineáris kombinációja. De $\text{lnko}(a,b) \mid s$ és $s > 0$, ezért $\text{lnko}(a,b) \leq s$. Az $\text{lnko}(a,b) \geq s$ és $\text{lnko}(a,b) \leq s$ egyenletekből azonnal következik, hogy $\text{lnko}(a,b) = s$. Ezzel beláttuk, hogy s tényleg az a és b legnagyobb közös osztója. ■

31.3. következmény. *Ha valamely a és b egészekre d | a és d | b, akkor d |* lnko(a, b).

Bizonyítás. Az állítás (31.4)-ből rögtön következik, mivel lnko(a, b) a 31.2. tétel szerint az a és b lineáris kombinációja. ■

31.4. következmény. *Bármely a, b egész és n nemnegatív egész számokra*

$$\text{lnko}(an, bn) = n \,\text{lnko}(a, b).$$

Bizonyítás. Ha $n = 0$, akkor az állítás nyilvánvaló. Ha $n > 0$, akkor lnko(an, bn) az $\{anx + bny\}$ halmaz legkisebb pozitív eleme, ami éppen az $\{ax + by\}$ halmaz legkisebb pozitív elemének n-szerese. ■

31.5. következmény. *Ha az n, a és b pozitív egészekre n | ab és* lnko$(a, n) = 1$, *akkor n | b.*

Bizonyítás. A bizonyítást a 31.1-4. gyakorlatra hagyjuk. ■

Relatív prím számok

Azt mondjuk, hogy az a és b számok ***relatív prímek,*** ha közös osztójuk csak az 1, azaz ha lnko$(a, b) = 1$. Például a 8 és a 15 relatív prímek, mivel a 8 osztói az $1, 2, 4$ és 8, míg a 15 osztói az $1, 3, 5$ és a 15. A következő tétel azt állítja, hogy ha két egész szám mindegyike relatív prím egy p egész számhoz, akkor a szorzatuk is relatív prím a p-hez.

31.6. tétel. *Ha az a, b és p egész számokra* lnko$(a, p) = 1$ *és* lnko$(b, p) = 1$, *akkor* lnko$(ab, p) = 1$.

Bizonyítás. A 31.2. tétel miatt léteznek olyan x, y, x' és y' egészek, hogy

$$\begin{aligned} ax + py &= 1, \\ bx' + py' &= 1. \end{aligned}$$

A fenti egyenlőségeket összeszorozva és átrendezve azt kapjuk, hogy

$$ab(xx') + p(ybx' + y'ax + pyy') = 1.$$

Így 1 az ab és p számok pozitív lineáris kombinációja. A 31.2. tételre hivatkozva a bizonyítás kész. ■

Azt mondjuk, hogy az $n_1, n_2, \ldots, n_k$ egész számok ***páronként relatív prímek,*** ha lnko$(n_i, n_j) = 1$ minden $i \neq j$ esetén.

A számelmélet alaptétele

A prímekkel való oszthatósággal kapcsolatos elemi, mégis fontos állítás a következő.

31.7. tétel. *Ha p prím, a és b pedig olyan egészek, hogy p | ab, akkor p | a vagy p | b.*

Bizonyítás. Indirekt módon bizonyítunk. Tegyük fel, hogy $p \mid ab$, de $p \nmid a$ és $p \nmid b$. Így lnko$(a, p) = 1$ és lnko$(b, p) = 1$, mivel p osztói csak az 1 és a p, valamint a feltétel szerint

p nem osztja a és b egyikét sem. A 31.6. tételből következik, hogy ekkor $\text{lnko}(ab, p) = 1$, ami ellentmond a $p \mid ab$ feltételnek, hiszen $p \mid ab$-ből $\text{lnko}(ab, p) = p$ következne. ■

A 31.7. tétel következménye, hogy az egész számok egyértelműen bonthatók fel prímek szorzatára.

31.8. tétel (a számelmélet alaptétele). *Egy a összetett egész szám pontosan egyféleképpen írható fel*

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

alakban, ahol a p_i számok prímek, $p_1 < p_2 < \cdots < p_r$, az e_i kitevők pedig pozitív egészek.

Bizonyítás. A bizonyítást a 31.1-10. gyakorlatra hagyjuk. ■

Példaképpen a 6000 prímfelbontása $2^4 \cdot 3 \cdot 5^3$.

Gyakorlatok

31.1-1. Bizonyítsuk be, hogy végtelen sok prímszám van. (*Útmutatás.* Mutassuk meg, hogy a $p_1, p_2, \ldots, p_k$ prímek egyike sem osztja a $(p_1 p_2 \cdots p_k) + 1$ számot.)
31.1-2. Bizonyítsuk be, hogy ha $a \mid b$ és $b \mid c$, akkor $a \mid c$.
31.1-3. Bizonyítsuk be, hogy ha p prím és $0 < k < p$, akkor $\text{lnko}(k, p) = 1$.
31.1-4. Bizonyítsuk be a 31.5. következményt.
31.1-5. Bizonyítsuk be, hogy ha p prím és $0 < k < p$, akkor $p \mid \binom{p}{k}$. Ennek következményeként mutassuk meg, hogy bármely a, b egészre és p prímre

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

31.1-6. Bizonyítsuk be, hogy ha az a és b tetszőleges pozitív egész számokra $a \mid b$, akkor

$$(x \bmod b) \bmod a = x \bmod a$$

bármely x egészre teljesül. Mutassuk meg, hogy ugyanezen feltételek mellett

$$x \equiv y \pmod{b} \Rightarrow x \equiv y \pmod{a}$$

is teljesül minden x és y egész esetén.
31.1-7. Legyen k pozitív egész. Azt mondjuk, hogy az n egész szám ***k-adik hatvány,*** ha létezik olyan a egész, amelyre $a^k = n$. Azt mondjuk, hogy az 1-nél nagyobb n egy ***nemtriviális hatvány,*** ha k-adik hatvány valamilyen $k > 1$-re. Mutassuk meg, hogyan lehet β-ban polinomiális idő alatt eldönteni, hogy egy legfeljebb β bit hosszú n egész szám nemtriviális hatvány.
31.1-8. Bizonyítsuk be a (31.6)–(31.10) egyenlőségeket.
31.1-9. Mutassuk meg, hogy a lnko operátor asszociatív, azaz

$$\text{lnko}(a, \text{lnko}(b, c)) = \text{lnko}(\text{lnko}(a, b), c)$$

minden a, b, c egész esetén.
31.1-10.⋆ Bizonyítsuk be a 31.8. tételt.

31.1-11. Adjunk hatékony algoritmust egy β bit hosszúságú egésznek egy nála kisebb egésszel való osztására és a maradék meghatározására úgy, hogy az algoritmus futási ideje $O(\beta^2)$ legyen.
31.1-12. Adjunk hatékony algoritmust egy binárisan adott β bit hosszúságú egész 10-es számrendszerbe való átírására. Mutassuk meg, hogy ha a legfeljebb β bit hosszúságú egész számok szorzása vagy osztása $M(\beta)$ ideig tart, akkor a bináris-decimális átváltás $\Theta(M(\beta)\lg\beta)$ idő alatt végrehajtható. (*Útmutatás.* A szám első és második felére külön rekurziót alkalmazva használjuk az oszd-meg-és-uralkodj elvet.)

31.2. A legnagyobb közös osztó

Ebben az alfejezetben az euklideszi algoritmust használjuk két egész szám legnagyobb közös osztójának gyors meghatározására. A futási idő vizsgálata meglepő kapcsolatot mutat a Fibonacci-számokkal, mivel szomszédos Fibonacci-számokra lesz a leghosszabb.

Az alfejezetben nemnegatív egészek vizsgálatára szorítkozunk. A megszorítás a (31.8) azonosság miatt megtehető, hiszen $\text{lnko}(a, b) = \text{lnko}(|a|, |b|)$.

Elvileg az a és b egészek prímfelbontásából $\text{lnko}(a, b)$ meghatározható. Ugyanis ha

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \tag{31.11}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}, \tag{31.12}$$

ahol $p_1, p_2, \ldots, p_r$ az a-ban és b-ben előforduló összes prímet jelöli és a számokat nem osztó prímek megállapodás szerint 0 kitevővel szerepelnek, akkor

$$\text{lnko}(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)}. \tag{31.13}$$

(A bizonyítást a 31.2-1. gyakorlat kéri majd.) Később, a 31.9. alfejezetben megmutatjuk, hogy a prímtényezőkre bontás napjaink legjobb algoritmusaival sem tehető meg polinomiális időben. Emiatt valószínűtlen, hogy a legnagyobb közös osztó meghatározására az iménti módszer elég hatékony lenne.

A legnagyobb közös osztó euklideszi algoritmussal történő kiszámítása a következő tételen alapul.

31.9. tétel (rekurziós tétel a legnagyobb közös osztó kiszámítására). *Tetszőleges nemnegatív a és pozitív b egész számokra*

$$\text{lnko}(a, b) = \text{lnko}(b, a \bmod b).$$

Bizonyítás. Megmutatjuk, hogy $\text{lnko}(a, b)$ és $\text{lnko}(b, a \bmod b)$ kölcsönösen osztják egymást, tehát nemnegatívak lévén a (31.5) azonosság szerint egyenlők.

Először megmutatjuk, hogy $\text{lnko}(a, b) \mid \text{lnko}(b, a \bmod b)$. Legyen $d = \text{lnko}(a, b)$. Ekkor $d \mid a$ és $d \mid b$. A (3.8) egyenlet szerint $(a \bmod b) = a - qb$, ahol $q = \lfloor a/b \rfloor$. Mivel $(a \bmod b)$ az a és b számok lineáris kombinációja, ezért a (31.4) tulajdonság miatt $d \mid (a \bmod b)$. Figyelembe véve, hogy $d \mid b$ és $d \mid (a \bmod b)$, a 31.3. következményt alkalmazva $d \mid \text{lnko}(b, a \bmod b)$, azaz

$$\text{lnko}(a, b) \mid \text{lnko}(b, a \bmod b). \tag{31.14}$$

Majdnem ugyanígy látható be, hogy lnko$(b, a \bmod b) \mid$ lnko(a, b). Legyen most $d =$ lnko$(b, a \bmod b)$. Ekkor $d \mid b$ és $d \mid (a \bmod b)$. Mivel $a = qb + (a \bmod b)$, ahol $q = \lfloor a/b \rfloor$, így a előáll b és $(a \bmod b)$ lineáris kombinációjaként. A (31.4) tulajdonság miatt ezért $d \mid a$ is teljesül. A $d \mid a$ és $d \mid b$ tulajdonságokból a 31.3. következmény miatt $d \mid$ lnko(a, b), vagyis

$$\text{lnko}(b, a \bmod b) \mid \text{lnko}(a, b). \tag{31.15}$$

A (31.5) tulajdonság felhasználásával a (31.14) és (31.15) sorokból a kívánt eredmény azonnal következik. ■

Az euklideszi algoritmus

A lnko meghatározására szolgáló alábbi algoritmust Euklidész (kb. i. e. 300) írja le *Elemek* című munkájában, de lehetséges, hogy már korábban is ismerték. Az algoritmus a 31.9. tételre épül. Az a és b bemenő adatok tetszőleges nemnegatív egészek.

```
EUKLIDESZ(a, b)

1  if b = 0
2     then return a
3     else return EUKLIDESZ(b, a mod b)
```

Példaként nézzük meg a lnko(30,21) kiszámítását:

```
EUKLIDESZ(30,21 ) = EUKLIDESZ(21,9)
                  = EUKLIDESZ(9,3)
                  = EUKLIDESZ(3,0)
                  = 3.
```

A számítások során az EUKLIDESZ algoritmus rekurzív módon háromszor is meghívódik.

Az EUKLIDESZ algoritmus helyességét a 31.9. tétel garantálja, illetve az a tény, hogy ha az algoritmus az a értékkel tér vissza a 2. sorban, akkor b csak 0 lehet, következésképp (31.9) miatt lnko$(a, b) =$ lnko$(a, 0) = a$. Az algoritmus hívása nem ismétlődhet végtelen sokszor, mivel a második (nemnegatív) argumentum szigorúan monoton csökken minden újabb hívásnál. Vagyis az EUKLIDESZ algoritmus mindig a helyes választ adva ér véget.

Az euklideszi algoritmus futási ideje

Elemezni fogjuk az EUKLIDESZ algoritmus futási idejét az a és b függvényében a legrosszabb esetre. Az általánosság megszorítása nélkül feltehető, hogy $a > b \geq 0$. Ha ugyanis $b > a \geq 0$, akkor az EUKLIDESZ(a, b) eljárás rekurzívan hívja az EUKLIDESZ(b, a)-t, azaz ha az első argumentum kisebb a másodiknál, akkor az EUKLIDESZ algoritmus egy rekurzív hívást az argumentumok cseréjére fordít, és csak azután lép tovább. Ha pedig $b = a > 0$, akkor egy rekurzív hívás után a futás véget is ér, hiszen $a \bmod b = 0$.

Az EUKLIDESZ(a, b) algoritmus futási ideje arányos a rekurzív hívások számával. Elemzésünkben F_k a k-adik Fibonacci-számot jelöli, amit a (3.21)-ben definiáltunk.

31.10. lemma. *Ha $a > b \geq 1$ és az* EUKLIDESZ(a, b) *algoritmus rekurzív hívásainak száma $k \geq 1$, akkor $a \geq F_{k+2}$ és $b \geq F_{k+1}$.*

Bizonyítás. A bizonyítást k szerinti teljes indukcióval végezzük. A $k = 1$ esetben $b \geq 1 = F_2$, továbbá $a > b$ miatt $a \geq 2 = F_3$. Mivel $b > (a \bmod b)$, ezért az első argumentum minden rekurzív hívásnál szigorúan nagyobb, mint a második, vagyis az $a > b$ feltétel minden rekurzív hívásra teljesül.

Indukciós feltevésünk szerint legyen igaz a lemma $k - 1$ rekurzív hívás esetén. Be kell látnunk, hogy igaz lesz k rekurzív hívásra is. Mivel $k > 0$, így $b > 0$, és az Euklidesz(a, b) algoritmus rekurzívan hívja az Euklidesz$(b, a \bmod b)$-t, ami viszont $k - 1$ rekurzív hívást hajt végre. Az indukciós feltétel szerint tehát $b \geq F_{k+1}$ (ami részben bizonyítja is a lemmát), és $(a \bmod b) \geq F_k$. Mivel $a > b > 0$, ezért $\lfloor a/b \rfloor \geq 1$, így

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor a/b \rfloor b) \\ &\leq a, \end{aligned}$$

ahonnan

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2}. \end{aligned}$$

■

Az alábbi tétel a lemma közvetlen következménye.

31.11. tétel (Lamé tétele). *Tetszőleges $k \geq 1$ esetén ha $a > b \geq 1$ és $b < F_{k+1}$, akkor az* Euklidesz(a, b) *algoritmus k-nál kevesebb rekurzív hívást hajt végre.*

Megmutatjuk, hogy a 31.11. tétel felső korlátja a lehető legjobb. Az Euklidesz algoritmus legrosszabb bemenő adatai szomszédos Fibonacci-számok. Mivel az Euklidesz(F_3, F_2) algoritmus pontosan egy rekurzív hívást hajt végre, valamint $k \geq 2$-re $F_{k+1} \bmod F_k = F_{k-1}$, így

$$\begin{aligned} \text{lnko}(F_{k+1}, F_k) &= \text{lnko}(F_k, (F_{k+1} \bmod F_k)) \\ &= \text{lnko}(F_k, F_{k-1}). \end{aligned}$$

Következésképpen az Euklidesz(F_{k+1}, F_k) algoritmus pontosan $k - 1$ rekurzív hívást tartalmaz, ami egyben a 31.11. tétel felső korlátja.

Az F_k értéke megközelítőleg $\phi^k/\sqrt{5}$, ahol ϕ a (3.22)-ben definiált $(1 + \sqrt{5})/2$ aranymetszés, tehát az Euklidesz algoritmus rekurzív hívásainak száma $O(\lg b)$. (Jobb korlátot szolgáltat a 31.2-5. gyakorlat.) Ha az Euklidesz algoritmust két β bit hosszúságú számra alkalmazzuk, akkor $O(\beta)$ aritmetikai és $O(\beta^3)$ bitművelettel kell számolnunk (feltéve, hogy β bit hosszúságú számok szorzása és osztása $O(\beta^2)$ bitműveletet igényel). A 31.2. feladat a bitműveletek számának $O(\beta^2)$ korlátra való leszorítását kéri.

Az euklideszi algoritmus bővített változata

Az alábbiakban az euklideszi algoritmust úgy módosítjuk, hogy további hasznos információt szolgáltasson. Célunk azon x, y egész együtthatók kiszámítása, amelyekre

$$d = \text{lnko}(a, b) = ax + by. \tag{31.16}$$

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

31.1. ábra. A BŐVÍTETT-EUKLIDESZ algoritmus működésének bemutatása a 99 és 78 bemenő adatok esetén. Mindegyik sor a rekurzió egy újabb szintjét mutatja: az a és b bemenő adatokat, az $\lfloor a/b \rfloor$ számított értékét, valamint a d, x, y visszatérő értékeket. A rekurzió következő szintjén a (d', x', y') hármas játssza (d, x, y) szerepét. A BŐVÍTETT-EUKLIDESZ(99,78) algoritmus eredményül a (3, −11, 14) számhármast szolgáltatja, így lnko(99,78) = 3 = 99 · (−11) + 78 · 14.

Megjegyezzük, hogy x és y bármelyike lehet 0 és negatív is. Ezen együtthatók ismerete később, lineáris kongruencia-egyenletek megoldásánál lesz hasznos a moduláris multiplikatív inverz meghatározásához. A BŐVÍTETT-EUKLIDESZ algoritmus bemenő adata egy nemnegatív számpár, az eredmény pedig a (31.16) egyenletet kielégítő (d, x, y) számhármas.

BŐVÍTETT-EUKLIDESZ(a, b)

```
1  if b=0
2     then return (a, 1, 0)
3  (d', x', y') ← BŐVÍTETT-EUKLIDESZ(b, a mod b)
4  (d, x, y) ← (d', y', x' − ⌊a/ b⌋y')
5  return (d, x, y)
```

A BŐVÍTETT-EUKLIDESZ algoritmus működését a lnko(99,78) kiszámítására a 31.1. ábra szemlélteti.

A BŐVÍTETT-EUKLIDESZ algoritmus az EUKLIDESZ algoritmus egy változata. Az első sor azt vizsgálja, hogy „$b = 0$” teljesül-e az EUKLIDESZ algoritmus első sorában. Ha $b = 0$, akkor a BŐVÍTETT-EUKLIDESZ algoritmus nemcsak a $d = a$ eredményre vezet a 2. sorban, hanem az $x = 1$ és $y = 0$ együtthatókat is megadja, amelyekkel $a = ax + by$. Ha $b \neq 0$, akkor a BŐVÍTETT-EUKLIDESZ algoritmus először a (d', x', y') számhármast számítja ki, ahol $d' = \text{lnko}(b, a \bmod b)$ és

$$d' = bx' + (a \bmod b)y'. \tag{31.17}$$

Ahogy az EUKLIDESZ algoritmusnál, ebben az esetben is $d = \text{lnko}(a, b) = d' = \text{lnko}(b, a \bmod b)$. A $d = ax + by$ egyenletben szereplő x és y meghatározásához (31.17)-et átírjuk, felhasználva a $d = d'$ és (3.8) egyenleteket:

$$\begin{aligned} d &= bx' + (a - \lfloor a/\, b \rfloor b)y' \\ &= ay' + b(x' - \lfloor a/\, b \rfloor y'). \end{aligned}$$

Így az $x = y'$ és $y = x' - \lfloor a/\, b \rfloor y'$ választások kielégítik a $d = ax + by$ egyenletet, ami a BŐVÍTETT-EUKLIDESZ algoritmus helyes működését bizonyítja.

Mivel az EUKLIDESZ és a BŐVÍTETT-EUKLIDESZ algoritmusok működésekor a rekurzív hívások száma megegyezik, ezért futási idejük konstans szorzótól eltekintve azonos, azaz $a > b > 0$ esetén $O(\lg b)$.

Gyakorlatok

31.2-1. Bizonyítsuk be, hogy a (31.11) és (31.12) egyenlőségekből (31.13) levezethető.

31.2-2. Számítsuk ki a Bővített-Euklidesz(899, 493) algoritmus (d, x, y) kimeneti értékeit.

31.2-3. Bizonyítsuk be, hogy tetszőleges a, k és n egészekre

$$\text{lnko}(a, n) = \text{lnko}(a + kn, n).$$

31.2-4. Írjuk át az Euklidesz algoritmust olyan iteratív algoritmussá, ami konstans méretű (azaz csak konstans számú egész értéket tároló) memóriát használ.

31.2-5. Mutassuk meg, hogy $a > b \geq 0$ esetén az Euklidesz(a, b) algoritmus használatakor legfeljebb $1 + \log_\phi b$ rekurzív hívás történik. Javítsuk ezt a korlátot $1 + \log_\phi(b/\text{lnko}(a, b))$-re.

31.2-6. Mi a visszatérési értéke a Bővített-Euklidesz(F_{k+1}, F_k) algoritmusnak? A választ kellőképpen indokoljuk is.

31.2-7. A $\text{lnko}(a_0, a_1, \ldots, a_n) = \text{lnko}(a_0, \text{lnko}(a_1, \ldots, a_n))$ rekurzió segítségével definiáljuk a lnko függvényt kettőnél több változó esetére. Bizonyítsuk be, hogy a definíció független az argumentumok sorrendjétől. Mutassuk meg, hogyan találhatunk olyan $x_0, x_1, \ldots, x_n$ egészeket, amelyekkel $\text{lnko}(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \ldots + a_nx_n$. Mutassuk meg, hogy az algoritmusban az osztások száma $O(n + \lg(\max\{a_0, a_1, \ldots, a_n\}))$.

31.2-8. Definiáljuk az $a_0, a_1, \ldots, a_n$ egészek ***legkisebb közös többszörösét*** (jelölése: $\text{lkkt}(a_0, a_1, \ldots, a_n)$) úgy, mint azt a legkisebb nemnegatív egész számot, amelyik mindegyik a_i egész szám többese. Adjunk hatékony módszert a $\text{lkkt}(a_0, a_1, \ldots, a_n)$ kiszámítására. Használjuk szubrutinként a (kétváltozós) lnko eljárást.

31.2-9. Mutassuk meg, hogy n_1, n_2, n_3 és n_4 pontosan akkor páronként relatív prímek, ha

$$\text{lnko}(n_1n_2, n_3n_4) = \text{lnko}(n_1n_3, n_2n_4) = 1.$$

Bizonyítsuk be az általánosított változatot is: $n_1, n_2, \ldots, n_k$ akkor és csak akkor páronként relatív prímek, ha az n_i számokból kiválasztható szorzatpárok egy $\lceil \lg k \rceil$ elemű halmazának elemei mind relatív prímek.

31.3. Műveletek maradékosztályokkal

A maradékosztályokkal lényegében ugyanúgy számolhatunk, mint közönséges egészekkel, azzal a megkötéssel, hogy ha modulo n számolunk, minden x eredményt a $\{0, 1, \ldots, n-1\}$ halmaz egyetlen vele modulo n kongruens elemével kell helyettesítenünk (azaz x-et x mod n-re cserélnünk). Ez a leegyszerűsített modell megfelel, ha az összeadás, kivonás és szorzás műveletekre szorítkozunk. Az alábbiakban a moduláris aritmetika egy formálisabb modelljét mutatjuk be, ami legjobban a csoportelmélet keretei között tárgyalható.

Véges csoportok

Azt mondjuk, hogy az $(S, \oplus)$ struktúra ***csoport,*** ha S nem üres halmaz, és a következők teljesülnek:

1. ***Zártság:*** A $\oplus$ kétváltozós belső művelet, azaz, ha $a, b \in S$, akkor $a \oplus b \in S$.
2. ***Egységelem:*** Létezik olyan $e \in S$, hogy minden $a \in S$ elemre $e \oplus a = a \oplus e = a$.
3. ***Asszociativitás:*** $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ minden $a, b, c \in S$ esetén.
4. ***Inverz:*** Minden $a \in S$ elemhez egyértelműen létezik olyan $b \in S$, hogy $a \oplus b = b \oplus a = e$.

Példaként tekintsük az egészek jól ismert $(\mathbf{Z}, +)$ csoportját, ahol a művelet az összeadás. Itt 0 az egységelem, és az a inverze $-a$. Azt mondjuk, hogy az $(S, \oplus)$ csoport ***kommutatív,*** ha $a \oplus b = b \oplus a$ minden $a, b \in S$-re. Ha egy csoport kommutatív, akkor ***Abel-csoportnak*** nevezzük. Ha $|S| < \infty$, akkor azt mondjuk, hogy $(S, \oplus)$ ***véges csoport.***

A maradékosztályok additív és multiplikatív csoportja

Legyen n tetszőleges pozitív egész. Ekkor a modulo n vett összeadás és szorzás segítségével két Ábel-csoport is adható, melyek a 31.1. alfejezetben definiált modulo n maradékosztályokon alapulnak. Maradékosztályok halmazán két Abel-csoport is megadható, amennyiben a műveleteket a modulo n vett összeadásnak és szorzásnak választjuk.

Ahhoz, hogy a $\mathbf{Z}_n$ halmazon csoportot készítsünk, kétváltozós műveletekre lesz szükségünk. Ezen műveleteket a szokásos összeadás és szorzás módosításával nyerjük, amit azért könnyű definiálni, mert két ekvivalencia-osztály összege és szorzata egyértelműen meghatározott ekvivalencia-osztály. Ugyanis, ha $a \equiv a' \pmod{n}$ és $b \equiv b' \pmod{n}$, akkor

$$\begin{aligned} a + b &\equiv a' + b' \pmod{n}, \\ ab &\equiv a'b' \pmod{n}. \end{aligned}$$

Emiatt a modulo n tekintett összeadást és szorzást, amelyet $+_n$ illetve $\cdot_n$-nel jelölünk, a következőképpen definiáljuk:

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n, \\ [a]_n \cdot_n [b]_n &= [ab]_n. \end{aligned} \tag{31.18}$$

(A $\mathbf{Z}_n$-beli kivonás is hasonlóképpen definiálható: $[a]_n -_n [b]_n = [a - b]_n$, de az osztás – amint majd látni fogjuk – sokkal bonyolultabb.) Ezek a tények jogosítanak fel bennünket arra, hogy amikor $\mathbf{Z}_n$-ben dolgozunk, az ekvivalencia-osztályokat (megszokott és kényelmes módon) a legkisebb nemnegatív elemükkel helyettesítsük. Az összeadást, a kivonást és a szorzást a reprezentáns-elemekkel végezzük el, és az eredményt osztályának reprezentánsával helyettesítjük (azaz x mod n-nel).

A modulo n összeadás definícióját felhasználva a ***modulo n additív csoporton*** $(\mathbf{Z}_n, +_n)$ értendő. Emiatt a modulo n additív csoportnak $|\mathbf{Z}_n| = n$ eleme van. A 31.2(a) táblázat a $(\mathbf{Z}_6, +)$ csoport műveleti tábláját mutatja.

31.12. tétel. *A $(\mathbf{Z}_n, +_n)$ struktúra véges Abel-csoport.*

Bizonyítás. A $+_n$ művelet asszociativitása és kommutativitása az $+$ asszociativitásának és kommutativitásának következménye, mivel

$$\begin{aligned} ([a]_n +_n [b]_n) +_n [c]_n &= [a + b]_n +_n [c]_n \\ &= [(a + b) + c]_n \\ &= [a + (b + c)]_n \\ &= [a]_n +_n [b + c]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n) \end{aligned}$$

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

31.2. ábra. Példa két véges csoportra. Az ekvivalencia-osztályokat reprezentáns-elemeikkel jelöltük. **(a)** A $(\mathbf{Z}_6, +_6)$ csoport. **(b)** A $(\mathbf{Z}_{15}^*, \cdot_{15})$ csoport.

$$\begin{aligned}[a]_n +_n [b]_n &= [a+b]_n \\ &= [b+a]_n \\ &= [b]_n +_n [a]_n.\end{aligned}$$

A $(\mathbf{Z}_n, +_n)$ egységeleme a 0 (pontosabban a $[0]_n$). Az a (vagyis az $[a]_n$) elem (additív) inverze $-a$ (pontosabban $[-a]_n$ vagy $[n-a]_n$), mivel $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ■

A modulo n szorzás definícióját felhasználva definiáljuk a $(\mathbf{Z}_n^*, \cdot_n)$ ***modulo n multiplikatív csoportot.*** Legyen

$$\mathbf{Z}_n^* = \{[a]_n \in \mathbf{Z}_n : \text{lnko}(a, n) = 1\},$$

azaz $\mathbf{Z}_n^*$ a $\mathbf{Z}_n$ n-hez relatív prím maradékosztályainak halmaza. Megmutatjuk, hogy $\mathbf{Z}_n^*$ definíciója értelmes. Ha $0 \le a < n$, akkor $a \equiv (a + kn) \pmod{n}$ minden k egészre. A 31.2-3. gyakorlat miatt ha $\text{lnko}(a, n) = 1$, akkor $\text{lnko}(a + kn, n) = 1$ is teljesül bármely k egészre. Mivel $[a]_n = \{a + kn : k \in \mathbf{Z}\}$, így $\mathbf{Z}_n^*$ valóban jól definiált. Egy ilyen halmazra példa a

$$\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\},$$

ahol a csoportművelet a modulo 15 szorzás. (Itt az $[a]_{15}$ elem helyett a szerepel, például $[7]_{15}$ helyett 7.) A 31.2(b) ábra a $(\mathbf{Z}_{15}^*, \cdot_n)$ műveleti tábláját mutatja. Például $\mathbf{Z}_{15}^*$-ben $8 \cdot 11 \equiv 13 \pmod{15}$. A csoport egységeleme az 1.

31.13. tétel. *A $(\mathbf{Z}_n^*, \cdot_n)$ struktúra véges Abel-csoport.*

Bizonyítás. A 31.6. tétel szerint $\cdot_n$ valóban a $\mathbf{Z}_n^*$ halmazon értelmezett belső művelet. A $\cdot_n$ asszociativitása és kommutativitása a 31.12. tétel mintájára igazolható. Az egységelem az $[1]_n$. Már csak az inverz elem létezését kell bizonyítanunk. Legyen $a \in \mathbf{Z}_n^*$, és legyen (d, x, y) a Bővített-Euklidesz(a, n) algoritmus kimenete. Ekkor $a \in \mathbf{Z}_n^*$ miatt $d = 1$, továbbá

$$ax + ny = 1,$$

vagy (ami ezzel ekvivalens)

$$ax \equiv 1 \pmod{n},$$

azaz $[x]_n$ az $[a]_n$ multiplikatív inverze modulo n. Az inverz egyértelműsége majd a 31.26. következményből adódik. ■

A multiplikatív inverz kiszámítására nézzünk egy példát. Legyen $a = 5$ és $n = 11$. Ekkor a Bővített-Euklidesz(a, n) algoritmus visszatérő értéke $(d, x, y) = (1, -2, 1)$, vagyis $1 = 5 \cdot (-2) + 11 \cdot 1$. Ezért -2 (egészen pontosan 9 mod 11) az 5 multiplikatív inverze modulo 11.

A fejezet további részében a $(\mathbf{Z}_n, +_n)$ és a $(\mathbf{Z}_n^*, \cdot_n)$ csoportok elemeit – a modulo n maradékosztályokat, illetve az n-hez relatív prím maradékosztályokat – egy-egy reprezentánselemükkel helyettesítjük, a $+_n$ és $\cdot_n$ műveleti jelek helyett pedig az $+$, illetve a $\cdot$ (vagy egymás mellé írás) aritmetikai jeleket használjuk. Hasonlóképpen, a modulo n kongruenciák helyett $\mathbf{Z}_n$-ben gyakran egyenlőséget írunk. Például az alábbi állítások ekvivalensek:

$$\begin{aligned} ax &\equiv b \pmod{n}, \\ [a]_n \cdot_n [x]_n &= [b]_n. \end{aligned}$$

A továbbiakban – ugyancsak kényelmi szempontból – az $(S, \oplus)$ csoportjelölés helyett az S jelölést használjuk, amennyiben nyilvánvaló, hogy mi a művelet. Így a $(\mathbf{Z}_n, +_n)$ és $(\mathbf{Z}_n^*, \cdot_n)$ csoportokra egyszerűen a $\mathbf{Z}_n$, illetve a $\mathbf{Z}_n^*$ jelölésekkel hivatkozunk.

Az a elem (multiplikatív) inverzét ($a^{-1} \bmod n$) fogja jelölni. A $\mathbf{Z}_n$-beli osztást az $a/b \equiv ab^{-1} \pmod{n}$ kongruenciával definiáljuk. Például $\mathbf{Z}_{15}^*$-ben $7^{-1} \equiv 13 \pmod{15}$, mivel $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, tehát $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

$\mathbf{Z}_n^*$ elemszámát $\phi(n)$-nel jelöljük. Ez a függvény az ***Euler-féle ϕ függvény.*** Ismeretes, hogy

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right), \tag{31.19}$$

ahol p befutja n összes prímosztóját (az n számot is beleértve, ha az prím). Ezt a formulát most nem bizonyítjuk. Intuitív alapon világos, hogy az n maradékainak $\{0, 1, \ldots, n-1\}$ halmazából törölni kell az n számot osztó p prím összes többesét. Például a 45 prímosztói a 3 és az 5, így

$$\begin{aligned} \phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24. \end{aligned}$$

Ha p prím, akkor $\mathbf{Z}_p^* = \{1, 2, \ldots, p-1\}$, és

$$\phi(p) = p - 1. \tag{31.20}$$

Ha n összetett, akkor $\phi(n) < n - 1$.

Részcsoportok

Ha $(S, \oplus)$ csoport, $S' \subseteq S$ és $(S', \oplus)$ is csoport, akkor azt mondjuk, hogy $(S', \oplus)$ az $(S, \oplus)$ ***részcsoportja.*** Például a páros számok az egész számok részcsoportja az összeadás műveletére nézve. A részcsoportok felismeréséhez a következő tétel nyújt hasznos segítséget.

31.14. tétel (véges csoport műveletre zárt részhalmaza részcsoport).
Ha $(S, \oplus)$ *véges csoport és* S' *az* S *olyan nem üres részhalmaza, amelyben* S' *minden* a, b *elempárjára* $a \oplus b \in S'$ *is teljesül, akkor* $(S', \oplus)$ *az* $(S, \oplus)$ *részcsoportja.*

Bizonyítás. A bizonyítást a 31.3-2. gyakorlatra hagyjuk. ■

Például a $\{0, 2, 4, 6\}$ a $\mathbf{Z}_8$ részcsoportja, mivel nem üres és zárt a + műveletre (pontosabban zárt a $+_8$ műveletre).

A következő tétel nagyon hasznos megszorítást tartalmaz a részcsoport elemszámára vonatkozóan. A tételt nem bizonyítjuk.

31.15. tétel (Lagrange-tétel). *Ha* $(S, \oplus)$ *véges csoport és* $(S', \oplus)$ *az* $(S, \oplus)$ *részcsoportja, akkor* $|S'|$ *az* $|S|$ *osztója.*

Azt mondjuk, hogy S' az S ***valódi*** részcsoportja, ha $S' \neq S$. A Lagrange-tétel alábbi következményére a Miller–Rabin prímteszt (31.8. alfejezet) elemzésekor lesz szükség.

31.16. következmény. *Ha* S' *az* S *csoport valódi részcsoportja, akkor* $|S'| \leq |S|/2$.

Egy elem által generált (ciklikus) részcsoportok

A 31.14. tétel érdekes módszert ad a kezünkbe egy véges $(S, \oplus)$ csoport bizonyos részcsoportjainak előállítására. Vegyünk egy a elemet és vele együtt az összes olyan általa generált elemet, amelyet úgy kapunk, hogy a csoportműveletet ismételten alkalmazzuk a-ra. A $k \geq 1$ esetre $a^{(k)}$ definíciója a következő:

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \ldots \oplus a}_{k}.$$

Például ha a $\mathbf{Z}_6$ csoportból az $a = 2$ elemet vesszük, akkor az $a^{(1)}, a^{(2)}, \ldots$ sorozat a következő:

$$2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots .$$

A $\mathbf{Z}_n$ csoportban $a^{(k)} = ka \bmod n$, a $\mathbf{Z}_n^*$ csoportban $a^{(k)} = a^k \bmod n$. Az ***a által generált részcsoportot,*** amelyet $\langle a \rangle$-val vagy $(\langle a \rangle, \oplus)$-val jelölünk, a következőképpen definiáljuk:

$$\langle a \rangle = \{a^{(k)} : k \geq 1\}.$$

Azt mondjuk, hogy a ***generálja*** az $\langle a \rangle$ részcsoportot, vagy a az $\langle a \rangle$ ***generáló eleme.*** Mivel S véges, $\langle a \rangle$ az S véges részhalmaza, amely akár S összes elemét is tartalmazhatja. A művelet asszociativitása miatt

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)},$$

ezért $\langle a \rangle$ zárt a műveletvégzésre, vagyis a 31.14. tétel szerint $\langle a \rangle$ az S részcsoportja. Például a $\mathbf{Z}_6$ csoportban

$$\begin{aligned} \langle 0 \rangle &= \{0\}, \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\}, \\ \langle 2 \rangle &= \{0, 2, 4\}. \end{aligned}$$

Hasonlóan, a $\mathbf{Z}_7^*$ csoportban

$$\begin{aligned}\langle 1\rangle &= \{1\},\\ \langle 2\rangle &= \{1,2,4\},\\ \langle 3\rangle &= \{1,2,3,4,5,6\}.\end{aligned}$$

Az S véges csoportban az a elem ***rendje*** az a legkisebb t pozitív egész, amelyre $a^{(t)} = e$, szokásos jelölése: $\mathrm{ord}(a)$.

31.17. tétel. *Ha a az* $(S,\oplus)$ *véges csoport eleme, akkor az a elem rendje megegyezik az általa generált részcsoport elemszámával, azaz* $\mathrm{ord}(a) = |\langle a\rangle|$.

Bizonyítás. Legyen $t = \mathrm{ord}(a)$. Mivel $a^{(t)} = e$ és $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ bármely $k \geq 1$ esetén, így minden t-nél nagyobb i egészhez létezik olyan i-nél kisebb j pozitív egész, hogy $a^{(i)} = a^{(j)}$. Emiatt az $a^{(t)}$ utáni a-hatványok már csak ismétlődhetnek, azaz $\langle a\rangle = \{a^{(1)}, a^{(2)}, \ldots, a^{(t)}\}$. Vagyis $|\langle a\rangle| \leq t$. Most megmutatjuk, hogy $|\langle a\rangle| \geq t$. Indirekt módon, tegyük fel, hogy van olyan i és j, hogy $1 \leq i \leq j \leq t$ és $a^{(i)} = a^{(j)}$. Ekkor $a^{(i+k)} = a^{(j+k)}$ minden $k \geq 0$ esetén, ahonnan $a^{(i+t-j)} = a^{(j+(t-j))} = e$. Ez azonban nem lehetséges, hiszen $i + (t-j) < t$, és t a legkisebb pozitív érték, amelyre $a^{(t)} = e$. Ezért az $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ sorozat minden eleme különböző, vagyis $|\langle a\rangle| \leq t$. Azt kapjuk, hogy $|\langle a\rangle| = \mathrm{ord}(a)$. ■

31.18. következmény. *Az* $a^{(1)}, a^{(2)}, \ldots$ *sorozat periodikus* $t = \mathrm{ord}(a)$ *periódussal, azaz* $a^{(i)} = a^{(j)}$ *akkor és csak akkor teljesül, ha* $i \equiv j \pmod t$.

Az előző következménnyel összhangban $a^{(0)}$-t e-nek definiálhatjuk, $a^{(i)}$ pedig legyen $a^{(i \bmod t)}$ minden i egészre.

31.19. következmény. *Ha az* $(S,\oplus)$ *véges csoport egységeleme* e*, akkor bármely* $a \in S$ *elemre*

$$a^{|S|} = e.$$

Bizonyítás. A Lagrange-tételből következik, hogy $\mathrm{ord}(a) \mid |S|$, és így $|S| \equiv 0 \pmod t$, ahol $t = \mathrm{ord}(a)$. Azt kapjuk, hogy $a^{|S|} = a^{(0)} = e$. ■

Gyakorlatok

31.3-1. Készítsük el a $(\mathbf{Z}_4, +_4)$ és a $(\mathbf{Z}_5^*, \cdot_5)$ csoportok műveleti tábláit. Mutassuk meg, hogy ezek a csoportok izomorfak: amennyiben az elemeiket kölcsönösen egyértelműen megfeleltető α leképezést úgy definiáljuk, hogy $a + b \equiv c \pmod 4$, az izomorfia ekvivalens azzal, hogy $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod 5$.

31.3-2. Bizonyítsuk be a 31.14. tételt.

31.3-3. Mutassuk meg, hogy bármely p prímre és e pozitív egészre

$$\phi(p^e) = p^{e-1}(p-1).$$

31.3-4. Mutassuk meg, hogy ha $n > 1$ és $a \in \mathbf{Z}_n^*$, akkor az $f_a : \mathbf{Z}_n^* \to \mathbf{Z}_n^*$, $f_a(x) = ax \bmod n$ függvény a $\mathbf{Z}_n^*$ egy permutációja.

31.3-5. Adjuk meg a $\mathbf{Z}_9$ és a $\mathbf{Z}_{13}^*$ összes részcsoportjait.

31.4. Lineáris kongruenciák megoldása

Most arra a kérdésre keressük a választ, hogy hogyan kell megoldani az

$$ax \equiv b \pmod{n} \tag{31.21}$$

kongruencia-egyenletet, ahol b egész, a és n pozitív egész számok. A problémának számos alkalmazása létezik, például a 31.7. fejezetben az RSA nyilvános kulcsú titkosírásnál a kulcsok megkeresésére vonatkozó eljárásnak is részét képezi. Feltételezzük, hogy a, b és n adottak, ekkor keressük azokat az x értékeket modulo n, amelyek kielégítik a (31.21) kongruenciát. Látni fogjuk, hogy 0, 1, illetve 1-nél több megoldás is létezhet.

Jelölje $\langle a \rangle$ a $\mathbf{Z}_n$ a eleme által generált részcsoportot. Mivel $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, így a (31.21) kongruenciának akkor és csak akkor létezik megoldása, ha $b \in \langle a \rangle$. A Lagrange-tétel (31.15. tétel) szerint $|\langle a \rangle|$ csak n osztója lehet. A következő tétel $\langle a \rangle$ egy pontosabb jellemzését adja.

31.20. tétel. *Legyenek a és n pozitív egészek, és legyen* $d = lnko(a, n)$. *Ekkor*

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}, \tag{31.22}$$

$\mathbf{Z}_n$*-beliek és ennek következtében*

$$|\langle a \rangle| = \frac{n}{d} .$$

Bizonyítás. Először belátjuk, hogy $d \in \langle a \rangle$. Vegyük észre, hogy a Bővített-Euklidesz(a, n) algoritmus olyan x', y' számokat állít elő, amelyekre $ax' + ny' = d$. Emiatt $ax' \equiv d \pmod{n}$, vagyis $d \in \langle a \rangle$.

Mivel $d \in \langle a \rangle$, ezért d bármely többese benne van $\langle a \rangle$-ban, hiszen a egy többesének többese is a többese. Így $\langle a \rangle$ tartalmazza a $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$ halmaz összes elemét, azaz $\langle d \rangle \subseteq \langle a \rangle$.

Most belátjuk, hogy $\langle a \rangle \subseteq \langle d \rangle$. Ha $m \in \langle a \rangle$, akkor $m = ax \bmod n$ valamilyen x egészre, ezért $m = ax + ny$ alkalmas y egész esetén. De (31.4) miatt $d \mid a$ és $d \mid n$-ből $d \mid m$ következik, azaz $m \in \langle d \rangle$.

A kapott eredmények összevetéséből $\langle a \rangle = \langle d \rangle$. $|\langle a \rangle| = n/d$ abból az észrevételből adódik, hogy d-nek pontosan n/d többese esik a $[0, n - 1]$ zárt intervallumba. ■

31.21. következmény. *Az* $ax \equiv b \pmod{n}$ *kongruencia az x ismeretlenben akkor és csak akkor oldható meg, ha* $\text{lnko}(a, n) \mid b$.

31.22. következmény. *Az* $ax \equiv b \pmod{n}$ *kongruenciának vagy d különböző megoldása van modulo n, ahol* $d = \text{lnko}(a, n)$, *vagy nincs megoldása.*

Bizonyítás. Ha az $ax \equiv b \pmod{n}$ kongruencia megoldható, akkor $b \in \langle a \rangle$. A 31.17. tétel szerint $\text{ord}(a) = |\langle a \rangle|$, ezért a 31.18. következmény és a 31.20. tétel miatt az $ai \bmod n$ $(i = 0, 1, \ldots)$ sorozat periodikus, periódusa $|\langle a \rangle| = n/d$. Ha $b \in \langle a \rangle$, akkor b pontosan d-szer szerepel az $ai \bmod n$ $(i = 0, 1, \ldots, n - 1)$ sorozatban, mivel az $\langle a \rangle$ n/d hosszúságú blokkjai pontosan d-szer ismétlődnek, ahogy i nullától $n - 1$-ig nő. Kapjuk tehát, hogy azon d helyek x értékei szolgáltatják az $ax \equiv b \pmod{n}$ megoldásait, amelyekre $ax \pmod{n} = b$. ■

31.23. tétel. *Legyen* $d = \text{lnko}(a,n) = ax' + ny'$ *alkalmas* x', y' *egészekkel (például a* Bővített-Euklidesz *algoritmussal kiszámolható értékek megfelelnek). Ha* $d \mid b$*, akkor az* $ax \equiv b \pmod{n}$ *kongruenciának az egyik megoldása* x_0*, ahol*

$$x_0 = x' \left(\frac{b}{d}\right) \bmod n.$$

Bizonyítás. Mivel

$$\begin{aligned} ax_0 &\equiv ax'\frac{b}{d} \pmod{n} \\ &\equiv d\frac{b}{d} \pmod{n} \qquad (\text{mivel } ax' \equiv d \pmod{n}) \\ &\equiv b \pmod{n}, \end{aligned}$$

így x_0 tényleg megoldása az $ax \equiv b \pmod{n}$ kongruenciának. ■

31.24. tétel. *Tegyük fel, hogy az* $ax \equiv b \pmod{n}$ *kongruencia megoldható (azaz* $d \mid b$*, ahol* $d = \text{lnko}(a,n)$*) és legyen* x_0 *az egyik megoldás. Ekkor a kongruenciának pontosan* d *különböző megoldása van, mégpedig* $x_i = x_0 + i(n/d)$ $(i = 0, 1, 2, \ldots, d-1)$.

Bizonyítás. Mivel $n/d > 0$ és $0 \le i(n/d) < n$ $(i = 0, 1, \ldots, d-1)$, ezért az $x_0, x_1, \ldots, x_{d-1}$ értékek mind különbözőek modulo n. Mivel x_0 egy megoldása az $ax \equiv b \pmod{n}$ kongruenciának, ezért $ax_0 \bmod n = b$. Ekkor az $i = 0, 1, \ldots, d-1$ értékekre

$$\begin{aligned} ax_i \bmod n &= a\left(x_0 + \frac{in}{d}\right) \bmod n \\ &= \left(ax_0 + \frac{ain}{d}\right) \bmod n \\ &= ax_0 \bmod n \qquad (\text{mert } d \mid a) \\ &= b, \end{aligned}$$

vagyis x_i is megoldás. A 31.22. következmény miatt azonban pontosan d megoldás van, így $x_0, x_1, \ldots, x_{d-1}$ ténylegesen az összes megoldás. ■

Ezzel megteremtettük azt a matematikai hátteret, amelyre szükségünk van az $ax \equiv b \pmod{n}$ kongruencia megoldásához. A következő algoritmus megadja e kongruencia összes megoldását. Az a, n bemenő adatok tetszőleges pozitív egészek, b pedig tetszőleges egész szám.

Lineáris-kongruencia-megoldó(a, b, n)

```
1  (d, x', y') ← Bővített-Euklidesz(a, n)
2  if d | b
3     then x_0 ← x'(b/d) mod n
4          for i ← 0 to d − 1
5               do print x_0 + i(n/d) mod n
6     else print „nincs megoldás”
```

Példaként nézzük meg, hogyan működik a program a $14x \equiv 30 \pmod{100}$ kongruencia esetén (vagyis $a = 14, b = 30$ és $n = 100$). Az első sorban a BŐVÍTETT-EUKLIDESZ algoritmus meghívása után a $(d, x, y) = (2, -7, 1)$ számhármast kapjuk. Mivel $2 \mid 30$, a 3–5. sorok utasításai hajtódnak végre. A harmadik sorban kiszámítjuk az $x_0 = (-7)(15) \bmod 100 = 95$ értékét. A 4–5. sorok ciklusa esetünkben 2 megoldást nyomtat ki, a 95-öt és a 45-öt.

A LINEÁRIS-KONGRUENCIA-MEGOLDÓ algoritmus a következőképpen működik: az első sor kiszámítja a $d = \text{lnko}(a, n)$ értékét, valamint azon x', y' értékeket, amelyekkel $d = ax' + ny'$. Ezzel olyan x' értéket kapunk, amely megoldása az $ax' \equiv d \pmod{n}$ kongruenciának. Ha d nem osztja a b számot, akkor a 31.21. következmény miatt az $ax \equiv b \pmod{n}$ kongruenciának nincs megoldása. A második sor ellenőrzi, hogy $d \mid b$ teljesül-e. Ha nem, akkor a 6. sorban kiírja, hogy nincs megoldás. Különben a 31.23. tétel eredményének felhasználásával a 3. sorban kiszámítja az $ax \equiv b \pmod{n}$ kongruencia x_0 megoldását. Ha már egy megoldás ismert, a 31.24. tétel szerint a többi $d - 1$ megoldást úgy kapjuk, hogy a meglevő megoldáshoz n/d többeseit hozzáadjuk modulo n. A 4–5. sorok **for** ciklusa x_0-tól kezdve, majd az n/d értéket sorban hozzáadogatva modulo n kinyomtatja mind a d darab megoldást.

A LINEÁRIS-KONGRUENCIA-MEGOLDÓ algoritmus $O(\lg n + \text{lnko}(a, n))$ aritmetikai műveletet hajt végre, mivel a BŐVÍTETT-EUKLIDESZ algoritmus $O(\lg n)$, a 4–5. sorok **for** ciklusának minden egyes iterációja pedig az lnko-val arányos konstans számú aritmetikai műveletet igényel.

A 31.24. tétel alábbi következménye egy különleges speciális esetet ír le.

31.25. következmény. *Bármely $n > 1$ esetén ha $\text{lnko}(a, n) = 1$, akkor az $ax \equiv b \pmod{n}$ kongruenciának egyetlen megoldása van modulo n.*

A $b = 1$ eset azért fontos, mivel a keresett x megoldás éppen az *a* ***multiplikatív inverze*** modulo n.

31.26. következmény. *Bármely $n > 1$ esetén ha $\text{lnko}(a, n) = 1$, akkor az $ax \equiv 1 \pmod{n}$ kongruenciának egyetlen megoldása van modulo n, különben pedig nincs megoldása.*

Ha a és n relatív prímek, a 31.26. következmény miatt az a multiplikatív inverzére modulo n az $(a^{-1} \bmod n)$ jelöléssel is hivatkozhatunk. Ha $\text{lnko}(a, n) = 1$, akkor az $ax \equiv 1 \pmod{n}$ kongruencia megoldása a BŐVÍTETT-EUKLIDESZ algoritmussal kiszámított x érték, mivel az

$$\text{lnko}(a, n) = 1 = ax + by$$

egyenletből $ax \equiv 1 \pmod{n}$ következik. Vagyis a BŐVÍTETT-EUKLIDESZ algoritmussal az $(a^{-1} \bmod n)$ gyorsan kiszámolható.

Gyakorlatok

31.4-1. Keressük meg a $35x \equiv 10 \pmod{50}$ összes megoldását.

31.4-2. Bizonyítsuk be, hogy ha $\text{lnko}(a, n) = 1$, akkor az $ax \equiv ay \pmod{n}$ kongruenciából $x \equiv y \pmod{n}$ következik. Mutassuk meg, hogy $\text{lnko}(a, n) = 1$ szükséges feltétel, azaz adjunk ellenpéldát $\text{lnko}(a, n) > 1$ esetén.

31.4-3. Változtassuk meg a LINEÁRIS-KONGRUENCIA-MEGOLDÓ algoritmus 3. sorát a következőképpen:

3 **then** $x_0 \leftarrow x'(b/d) \bmod (n/d)$.

Működik-e így is? Magyarázzuk meg, hogy miért, illetve miért nem.

31.4-4.⋆ Legyen p prím és $f(x) \equiv f_0 + f_1 x + \ldots + f_t x^t \pmod{p}$ egy t-edfokú polinom, amelynek f_i együtthatói a $\mathbf{Z}_p$ elemei. Azt mondjuk, hogy $a \in \mathbf{Z}_p$ az f ***zérushelye*** vagy ***gyöke,*** ha $f(a) \equiv 0 \pmod{p}$. Bizonyítsuk be, hogy ha a az f zérushelye, akkor $f(x) \equiv (x-a)\,g(x) \pmod{p}$ valamely $(t-1)$-edfokú $g(x)$ polinomra. Bizonyítsuk be t szerinti indukcióval, hogy egy t-edfokú $f(x)$ polinomnak legfeljebb t különböző zérushelye lehet modulo p.

31.5. A kínai maradéktétel

Időszámításunk szerint 100 körül Sun-Ce kínai matematikus oldotta meg azt a problémát, hogy hogyan lehet olyan x egészeket találni, amelyek 3-mal, 5-tel és 7-tel osztva rendre 2, 3, illetve 2 maradékot adnak. Egy ilyen megoldás az $x = 23$, az összes megoldás pedig a $23 + 105k$ alakú számok halmaza, ahol k tetszőleges egész számot jelöl. A kínai maradéktétel egy megfeleltetést létesít páronként relatív prím modulusú kongruenciák rendszere (például 3, 5 és 7) és egy olyan kongruencia között, amelynek modulusa az iménti modulusok szorzata (az előbbi példa szerint 105).

A kínai maradéktételnek két nagyon fontos felhasználása van. Legyen $n = n_1 n_2 \cdots n_k$ egész szám páronként relatív prím n_i faktorokkal. Először is, a kínai maradéktétel egy struktúra-leíró tétel, mivel úgy jellemzi $\mathbf{Z}_n$ szerkezetét, mint a $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ direkt szorzattal izomorf struktúrát, amennyiben az összeadás és a szorzás minden i-re az i. komponensekben történik modulo n_i. Másodszor, a tételt gyakran alkalmazzuk gyors algoritmusok készítésekor, mivel az egyes $\mathbf{Z}_{n_i}$ halmazokban hatékonyabban lehet műveleteket végezni (bitműveletekben számolva), mint $\mathbf{Z}_n$-ben.

31.27. tétel (kínai maradéktétel). *Legyenek $n_1, n_2, \ldots, n_k$ páronként relatív prím egészek, $n = n_1 n_2 \cdots n_k$ és legyen $a \in \mathbf{Z}_n$. Legyenek továbbá $a_i \in \mathbf{Z}_{n_i}$ az alábbiak:*

$$a_i = a \bmod n_i \qquad i = 1, 2, \ldots, k.$$

Tekintsük az

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) \tag{31.23}$$

megfeleltetést. A (31.23) *leképezés egy kölcsönösen egyértelmű megfeleltetés (bijekció) $\mathbf{Z}_n$ és a $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ direkt szorzat elemei között. Ekkor a $\mathbf{Z}_n$ elemeivel végzett műveletek ekvivalensek a megfelelő szám k-asok minden egyes koordinátájával függetlenül végrehajtott ugyanolyan típusú műveletekkel. Vagyis ha*

$$a \leftrightarrow (a_1, a_2, \ldots, a_k),$$

$$b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

akkor

$$(a+b) \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k), \tag{31.24}$$

$$(a-b) \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k), \tag{31.25}$$

$$(ab) \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k). \tag{31.26}$$

Bizonyítás. A két reprezentáció egymásnak való megfeleltetése eléggé nyilvánvaló. Az a-nak megfeleltetett $(a_1, a_2, \ldots, a_k)$ k-as kiszámítása csak k osztást igényel. Az a kiszámítása

az $(a_1, a_2, \ldots, a_k)$ direkt szorzatból egy kicsit nehezebb, és a következő módon történik. Legyen $m_i = n/n_i$ $(i = 1, 2, \ldots, k)$, vagyis $m_i = n_1 n_2 \ldots n_{i-1} n_{i+1} \ldots n_k$. Definiáljuk az alábbi egyenleteket:

$$c_i = m_i(m_i^{-1} \bmod n_i) \qquad i = 1, 2, \ldots, k. \tag{31.27}$$

A (31.27) egyenletek jól definiáltak. Mivel m_i és n_i relatív prímek, a 31.6. tétel és a 31.26. következmény miatt ($m_i^{-1} \bmod n_i$) létezik. Végül a-t az $a_1, a_2, \ldots, a_k$ függvényeként az

$$a \equiv (a_1 c_1 + a_2 c_2 + \ldots + a_k c_k) \pmod{n} \tag{31.28}$$

kongruencia kiszámításából kapjuk. Most megmutatjuk, hogy a (31.28) egyenletből $a \equiv a_i \pmod{n_i}$ $(i = 1, 2, \ldots, k)$ következik. Vegyük észre, hogy ha $j \neq i$, akkor $m_j \equiv 0 \pmod{n_i}$, amiből azt kapjuk, hogy $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Azt is vegyük észre, hogy a (31.27) egyenlet miatt $c_i \equiv 1 \pmod{n_i}$. Vagyis a

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0)$$

megfeleltetést kapjuk, ahol a c_i vektornak csak az i-edik koordinátája különbözik 0-tól, amikor is 1. Így a c_i-k bizonyos értelemben a reprezentáció „bázisát" képezik. Emiatt minden i-re teljesül, hogy

$$\begin{aligned} a &\equiv a_i c_i \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) \pmod{n_i} \\ &\equiv a_i \pmod{n_i}, \end{aligned}$$

amit bizonyítani akartunk. Mivel a megfeleltetés mindkét irányban működik, így kölcsönösen egyértelmű. Végezetül, a (31.24)–(31.26) megfeleltetések a 31.1-6. gyakorlatból rögtön következnek, mivel $x \bmod n_i = (x \bmod n) \bmod n_i$ $(i = 1, 2, \ldots, k)$ minden x-re fennáll. ■

Az alábbi következményeket a fejezet későbbi részeiben használjuk fel.

31.28. következmény. *Ha $n_1, n_2, \ldots, n_k$ páronként relatív prímek, valamint $n = n_1 n_2 \cdots n_k$, akkor bármely $a_1, a_2, \ldots, a_k$ egészekkel az*

$$x \equiv a_i \pmod{n_i} \qquad (i = 1, 2, \ldots, k)$$

egyenleteknek kongruencia-rendszernek egyértelműen létezik közös megoldása modulo n.

31.29. következmény. *Ha $n_1, n_2, \ldots, n_k$ páronként relatív prímek, továbbá $n = n_1 n_2 \cdots n_k$, akkor az x és a egészekre az*

$$x \equiv a \pmod{n_i} \qquad (i = 1, 2, \ldots, k)$$

egyenletek akkor és csak akkor teljesülnek egyidejűleg, ha

$$x \equiv a \pmod{n}.$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

31.3. ábra. A kínai maradéktétel működésének bemutatása $n_1 = 5$ és $n_2 = 13$ esetén. Ebben a példában $c_1 = 26$ és $c_2 = 40$. Az i-edik sor j-edik oszlopa az a mod 65 értéket tartalmazza úgy, hogy (a mod 5) = i és (a mod 13) = j. Megjegyezzük, hogy a 0. sor 0. oszlopában 0 van. Hasonlóan kapjuk, hogy a 4. sor 12. oszlopában 64 szerepel (ami -1-gyel kongruens az $5, 13$, és így a 65 modulus szerint is). Mivel $c_1 = 26$, ezért egy sorral lejjebb lépve az a értéke 26-tal nő. Hasonlóan, $c_2 = 40$ azt jelenti, hogy egy oszloppal jobbra lépve a értéke 40-nel nő. Ha az a értékét 1-gyel akarjuk növelni, ennek az felel meg, hogy átlósan lefelé és jobbra mozdulunk (a legalsó sorból a legfelsőbe, a jobb szélről a bal szélre ugrunk).

Nézzük meg a kínai maradéktétel alkalmazását az

$$\begin{aligned} a &\equiv 2 \pmod 5, \\ a &\equiv 3 \pmod{13} \end{aligned}$$

példáján. Ekkor $a_1 = 2, n_1 = m_2 = 5, a_2 = 3, n_2 = m_1 = 13$, és ki szeretnénk számolni az a mod 65 értékét, hiszen most $n = 65$. Mivel $13^{-1} \equiv 2 \pmod 5$ és $5^{-1} \equiv 8 \pmod{13}$, ezért

$$\begin{aligned} c_1 &= 13(2 \bmod 5) = 26, \\ c_2 &= 5(8 \bmod 13) = 40, \end{aligned}$$

és

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65}. \end{aligned}$$

A kínai maradéktétel működését a 31.3. ábra mutatja be.

Eszerint vagy direkt módon dolgozunk modulo n, vagy helyette a megfelelő modulo n_i reprezentációkkal számolunk, ahogy éppen kényelmesebb. A számolások teljes mértékben ekvivalensek.

Gyakorlatok

31.5-1. Határozzuk meg az $x \equiv 4 \pmod 5$ és az $x \equiv 5 \pmod{11}$ kongruenciák összes megoldását.

31.5-2. Keressük meg az összes olyan x egész számot, amely 9-cel, 8-cal és 7-tel osztva rendre 1, 2 és 3 maradékot ad.

31.5-3. Bizonyítsuk be a 31.27. tételben szereplő definíció felhasználásával, hogy $\text{lnko}(a, n) = 1$ esetén

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)) .$$

31.5-4. Bizonyítsuk be a 31.27. tételben szereplő definíció felhasználásával, hogy az $f(x) \equiv 0 \pmod n$ kongruencia gyökeinek száma egyenlő az $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}, \ldots, f(x) \equiv 0 \pmod{n_k}$ egyenletek gyökei számának szorzatával.

31.6. Egy elem hatványai

Ahogy egy adott a elem többeseit vizsgáltuk modulo n, ugyanolyan természetes egy $a \in \mathbf{Z}_n^*$ elem

$$a^0, a^1, a^2, a^3, \ldots \tag{31.29}$$

hatványai sorozatának vizsgálata modulo n. A hatványozást a 0 kitevővel kezdve a sorozat első eleme $a^0 \bmod n = 1$, az $(i+1)$-edik eleme pedig $a^i \bmod n$ lesz. Például a 3 hatványai modulo 7 az alábbiak:

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	...

A 2 hatványai modulo 7 pedig

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	...

Ebben a fejezetben jelölje $\langle a \rangle$ a $\mathbf{Z}_n^*$ halmaz a eleme által generált ciklikus részcsoportot, $\mathrm{ord}_n(a)$ (olvasd: az a elem rendje modulo n) pedig az $a \in \mathbf{Z}_n^*$ elem rendjét. A 31.17. tétel szerint egy elem rendje az általa generált ciklikus részcsoport elemei számát jelenti. Például $\langle 2 \rangle = \{1, 2, 4\}$ a $\mathbf{Z}_7^*$ halmazban, vagyis $\mathrm{ord}_7(2) = 3$. Az Euler-féle ϕ-függvény definíciója szerint $\mathbf{Z}_n^*$ elemszáma $\phi(n)$ (31.3. fejezet). Most $\mathbf{Z}_n^*$ jelöléseit használva átfogalmazzuk a 31.19. következményt úgy, hogy eredményül Euler tételét és annak a p prím modulus esetén vett speciális esetét, a Fermat-tételt kapjuk.

31.30. tétel (Euler-tétel). *Bármely 1-nél nagyobb n egész szám esetén*

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ minden } a \in \mathbf{Z}_n^* \text{ elemre.}$$

31.31. tétel (Fermat-tétel). *Ha p prím, akkor*

$$a^{p-1} \equiv 1 \pmod{p} \text{ minden } a \in \mathbf{Z}_p^* \text{ elemre.}$$

Bizonyítás. A (31.20) egyenletből $\phi(p) = p - 1$, ha p prím. ■

Ez a következmény a 0 kivételével $\mathbf{Z}_p$ minden elemére igaz, hiszen $0 \notin \mathbf{Z}_p^*$. De ha p prím, akkor $\mathbf{Z}_p$ bármely a elemére teljesül, hogy $a^p \equiv a \pmod{p}$.

Ha $\mathrm{ord}_n(g) = |\mathbf{Z}_n^*|$, akkor $\mathbf{Z}_n^*$ minden eleme g valamilyen hatványa modulo n. Ekkor azt mondjuk, hogy a g ***primitív gyök*** vagy $\mathbf{Z}_n^*$ ***generáló eleme.*** Például a 3 egy primitív gyök modulo 7, de a 2 nem az. Ha $\mathbf{Z}_n^*$ tartalmaz primitív gyököt, akkor azt mondjuk, hogy a $\mathbf{Z}_n^*$ csoport ***ciklikus.*** A következő tétel bizonyítására itt nem térünk ki, az megtalálható például Niven és Zuckerman könyvében [231].

31.32. tétel. *A $\mathbf{Z}_n^*$ csoport $n > 1$-re pontosan az $n = 2, 4, p^e, 2p^e$ esetekben ciklikus, ahol p tetszőleges páratlan prímet, e pedig tetszőleges pozitív egészet jelöl.*

Ha g a $\mathbf{Z}_n^*$ primitív gyöke és a a $\mathbf{Z}_n^*$ tetszőleges eleme, akkor létezik olyan z, hogy $g^z \equiv a \pmod{n}$. Ezt a z számot az a modulo n vett g alapú ***diszkrét logaritmusának*** vagy ***indexének*** nevezzük és $\mathrm{ind}_{n,g}(a)$-val jelöljük.

31.33. tétel (diszkrét logaritmus tétele). *Ha g a $\mathbf{Z}_n^*$ primitív gyöke, akkor $g^x \equiv g^y \pmod{n}$ akkor és csak akkor teljesül, ha $x \equiv y \pmod{\phi(n)}$.*

Bizonyítás. Tegyük fel először, hogy $x \equiv y \pmod{\phi(n)}$. Ekkor $x = y + k\phi(n)$ valamely k egészre. Ezért

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} \pmod{n} \\ &\equiv g^y \cdot (g^{\phi(n)})^k \pmod{n} \\ &\equiv g^y \cdot 1^k \pmod{n} \qquad \text{(az Euler-tétel miatt)} \\ &\equiv g^y \pmod{n}. \end{aligned}$$

A fordított irány bizonyításához tegyük fel, hogy $g^x \equiv g^y \pmod{n}$. Mivel g hatványai $\langle g \rangle$ minden elemét előállítják, és $|\langle g \rangle| = \phi(n)$, így a 31.18. következmény miatt g hatványainak sorozata $\phi(n)$ periódusú. Vagyis ha $g^x \equiv g^y \pmod{n}$, akkor szükségképpen $x \equiv y \pmod{\phi(n)}$. ■

A diszkrét logaritmusra való áttérés leegyszerűsíti egyes kongruenciák megoldhatóságának vizsgálatát, mint ahogy ezt a következő tétel bizonyítása is mutatja.

31.34. tétel. *Ha p páratlan prím és e pozitív egész, akkor az*

$$x^2 \equiv 1 \pmod{p^e} \tag{31.30}$$

kongruenciának csak 2 megoldása van, mégpedig az $x = 1$ és az $x = -1$.

Bizonyítás. Legyen $n = p^e$. A 31.32. tételből következik, hogy erre a modulusra nézve létezik primitív gyök. Ezért a (31.30) kongruencia az

$$(g^{\mathrm{ind}_{n,g}(x)})^2 \equiv g^{\mathrm{ind}_{n,g}(1)} \pmod{n} \tag{31.31}$$

alakba írható. Mivel $\mathrm{ind}_{n,g}(1) = 0$, ezért a 31.33. tétel szerint a (31.35) kongruencia ekvivalens a

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)} \tag{31.32}$$

kongruenciával. Most $\mathrm{ind}_{n,g}(x)$ az ismeretlenünk. A kongruencia megoldásához a 31.4. alfejezet módszereit használjuk. A (31.19) egyenlet miatt $\phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}$. Legyen $d = \mathrm{lnko}(2, \phi(n)) = \mathrm{lnko}(2, (p-1)p^{e-1}) = 2$ és mivel $d \mid 0$, a 31.24. tétel miatt a (31.32) kongruenciának pontosan $d = 2$ megoldása van. Ennélfogva a (31.30) kongruenciának is pontosan 2 megoldása van. A megoldások az $x = 1$ és $x = -1$, amelyek szemmel láthatóan valóban megoldások. ■

Az x szám az ***1 nemtriviális négyzetgyöke modulo n,*** ha megoldása az $x^2 \equiv 1 \pmod{n}$ kongruenciának, de x nem kongruens a két triviális négyzetgyök, az 1 és a -1 egyikével sem modulo n. Például a 6 az 1 nemtriviális négyzetgyöke modulo 35. A 31.34. tétel alábbi következményére a Miller–Rabin prímtesztelő algoritmus (lásd a 31.8. alfejezetet) helyességének bizonyításához lesz szükségünk.

31.35. következmény. *Ha az 1-nek létezik nemtriviális négyzetgyöke* mod *n, akkor az n összetett szám.*

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

31.4. ábra. Az $a^b \bmod n$ kiszámítása a MODULÁRIS-HATVÁNYOZÓ algoritmus segítségével $a = 7, b = 560 = \langle 1000110000 \rangle$ és $n = 561$ egészek esetén. Az egyes oszlopokban a **for** ciklus ismételt végrehajtása utáni értékei vannak. A végeredmény 1.

Bizonyítás. A 31.34. tételből következik, hogy ha létezik 1-nek nemtriviális négyzetgyöke modulo n, akkor n nem lehet sem páratlan prím, sem páratlan prím hatványa. Ha $x^2 \equiv 1 \pmod 2$, akkor $x \equiv 1 \pmod 2$, vagyis az 1 minden négyzetgyöke triviális modulo 2. Vagyis n nem lehet prím. De mivel $n > 1$ és nemtriviális négyzetgyök létezik, ezért n összetett. ■

Moduláris hatványozás

A számelméleti számításokban gyakran előfordul, hogy egy számot hatványozni kell valamely modulusra nézve, amit ***moduláris hatványozásnak*** nevezünk. Azaz gyors eljárást keresünk az $a^b \bmod n$ kiszámítására, ahol a és b nemnegatív egészek, n pedig pozitív egészet jelöl. A moduláris hatványozás számos prímtesztelő algoritmusban és az RSA nyilvános kulcsú titkosírásnál is kulcsfontosságú művelet. Az ***ismételt négyzetreemelés módszere*** b bináris alakját használva ad hatékony megoldást a problémára.

Legyen $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ a b kettes számrendszerbeli alakja. (Vagyis az ábrázolás $k + 1$ bit hosszúságú, b_k jelöli a legnagyobb helyiértékű, b_0 pedig a legkisebb helyiértékű jegyet.) Az alábbi eljárás $a^c \bmod n$ értékét számítja ki. A c értéke duplázásokkal, illetve inkrementálásokkal nő 0-tól b-ig.

```
MODULÁRIS-HATVÁNYOZÓ(a, b, n)
 1  c ← 0
 2  d ← 1
 3  legyen ⟨b_k, b_{k−1}, ..., b_0⟩ a b bináris alakja
 4  for i ← k downto 0
 5       do c ← 2c
 6          d ← (d · d) mod n
 7          if b_i = 1
 8             then c ← c + 1
 9                  d ← (d · a) mod n
10  return d
```

A módszer lényege a négyzetreemelés alkalmazása minden iteráció 6. sorában, ami indokolja az „ismételt négyzetreemelés” nevet. Például az $a = 7, b = 560$ és $n = 561$ értékekkel az algoritmus működése a 31.4. ábrán látható. A kitevők sorozatát a táblázat c-vel jelölt sora mutatja.

A c változóra nincs tényleges szükség az algoritmus során, azt csak magyarázó célzattal hagytuk a programban. Vizsgáljuk meg az algoritmus alábbi, két részből álló ciklusinvariánsát:

A 4–9 sorok **for** ciklusában minden egyes iteráció megkezdése előtt a változók értékei az alábbiak:

1. A c változó értéke megegyezik b bináris ábrázolása $\langle b_k, b_{k-1}, \ldots, b_{i+1}\rangle$ kezdőszeletével, valamint
2. $d = a^c \bmod n$.

Ezt a ciklusinvariánst az alábbi módon használjuk.

Teljesül: Kezdetben $i = k$, vagyis a $\langle b_k, b_{k-1}, \ldots, b_{i+1}\rangle$ kezdőszelete üres. Ez a $c = 0$ értéknek felel meg. Továbbá az is igaz, hogy $d = 1 = a^0 \bmod n$.

Megmarad: Jelölje c' és d' a c és d változók értékét a **for** ciklus végrehajtása után és a következő iteráció végrehajtása előtt. Minden egyes iteráció után $c' \leftarrow 2c$ (ha $b_i = 0$) vagy $c' \leftarrow 2c + 1$ (ha $b_i = 1$), vagyis a c értéke a következő iteráció megkezdése előtt helyes. Ha $b_i = 0$, akkor $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n$. Ha $b_i = 1$, akkor $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. A következő iteráció megkezdése előtt mindkét esetben $d = a^c \bmod n$.

Befejeződik: Az iteráció az $i = -1$ értékkel ér véget. Ekkor $c = b$, mivel c a b bináris ábrázolásának teljes kezdőszelete, $\langle b_k, b_{k-1}, \ldots, b_0\rangle$. Ekkor $d = a^c \bmod n = a^b \bmod n$.

Ha az a, b és n bemenő adatok ábrázolásához β bit elegendő, akkor az algoritmus $O(\beta)$ aritmetikai műveletet és $O(\beta^3)$ bitműveletet igényel.

Gyakorlatok

31.6-1. Készítsünk olyan táblázatot, amely a $\mathbf{Z}_{11}^*$ elemeinek rendjét mutatja. Keressük meg a legkisebb g primitív gyököt modulo 11, és adjuk meg a táblázatban az $\text{ind}_{11,g}(x)$ értékeit $\mathbf{Z}_{11}^*$ minden x elemére.

31.6-2. Adjunk egy moduláris hatványozó algoritmust arra az esetre, amikor a b változó bitjeit a legkisebb helyi értékű jegytől kezdve olvassuk.

31.6-3. Magyarázzuk meg, hogy hogyan számítja ki a MODUL ÁRIS-HATVÁNYOZÓ algoritmus $a^{-1} \bmod n$ értékét valamely $a \in \mathbf{Z}_n^*$ esetén, ha $\phi(n)$-t ismertnek tételezzük fel.

31.7. Az RSA nyilvános kulcsú titkosírás

A nyilvános kulcsú titkosírás segítségével oly módon kódolható az egyik résztvevő által a másiknak szánt üzenet, hogy ha valaki meg is tudja szerezni a kódolt üzenetet, akkor sem tudja megfejteni. A nyilvános kulcsú titkosírás lehetővé teszi azt is, hogy a felek hamisíthatatlan „digitális aláírást” fűzzenek az elektronikus üzenet végéhez. Ez az aláírás a hivatalos dokumentumok kézírásos aláírásának elektronikus változata. Bárki könnyen összevetheti az eredetivel, de nem hamisítható, és érvényét veszti, ha az üzenetnek akár csak egy bitje is megváltozik. Azaz szavatolja mind az aláíró személyét, mind az aláírt üzenet tartalmát. Tökéletes eszköz elektronikusan aláírt üzleti szerződésekhez, elektronikus csekkekhez, elektronikus megrendelésekhez és egyéb olyan elektronikus közleményekhez, amelyek hitelesítést igényelnek.

Az RSA nyilvános kulcsú titkosírás létét annak a szinte drámai különbségnek köszönheti, hogy milyen könnyű nagy prímszámokat találni, és ezzel szemben milyen nehéz két

nagy prím szorzatát tényezőkre bontani. A 31.8. alfejezet nagy prímek keresésére ismertet egy hatékony eljárást, a 31.9. alfejezet pedig nagy egész számok tényezőkre bontásának problémáját vizsgálja.

Nyilvános kulcsú titkosírások

A nyilvános kulcsú titkosírásnál mindegyik félnek van egy ***nyilvános kulcsa*** és egy ***titkos kulcsa.*** Minden egyes kulcs információt hordoz. Például az RSA titkosírásnál a kulcsok egész számpárok. A titkosírásos példák hagyományos résztvevői „Aliz" és „Bob" névre hallgatnak. Aliz nyilvános és titkos kulcsát jelölje P_A illetve S_A, Bobét pedig P_B illetve S_B.

Minden résztvevő elkészíti a saját nyilvános és titkos kulcsát. Mindegyikük titokban tartja a titkos kulcsot, de felfedi a nyilvános kulcsot, sőt azt akár közzé is teheti. Általában kényelmes azt feltételezni, hogy a nyilvános kulcsok benne vannak egy nyilvános könyvtárban, hogy bármely résztvevő bármely másik résztvevő nyilvános kulcsához hozzáférhessen.

A nyilvános és titkos kulcsok függvényeket jelentenek, amelyeket bármely üzenetre alkalmazni lehet. Legyen $\mathcal{D}$ a megengedett üzenetek halmaza. Például $\mathcal{D}$ lehet a véges hosszúságú bitsorozatok halmaza. A nyilvános kulcsú titkosírás legegyszerűbb (és az eredetihez hű) leírása szerint megköveteljük, hogy a nyilvános és a titkos kulcsok egy-egyértelmű megfeleltetést létesítsenek $\mathcal{D}$ elemei között. Aliz P_A nyilvános kulcsának megfelelő függvényt jelöljük $P_A()$-val, az S_A titkos kulcsának megfelelő függvényt pedig $S_A()$-val. Észrevehetjük, hogy $P_A()$ és $S_A()$ értelmezési tartománya és értékkészlete is $\mathcal{D}$. Feltesszük, hogy P_A illetve S_A ismeretében $P_A()$ és $S_A()$ gyorsan kiszámolhatók.

A nyilvános és titkos kulcs minden résztvevő számára egy „összeillő párt" alkot, amennyiben olyan függvényeket határoznak meg, amelyek egymás inverzei. Azaz

$$M = S_A(P_A(M)), \tag{31.33}$$

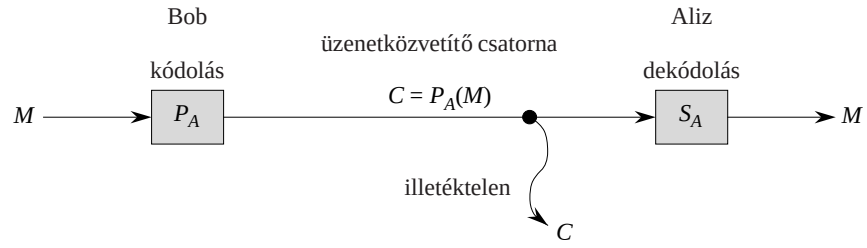
$$M = P_A(S_A(M)) \tag{31.34}$$

bármely $M \in \mathcal{D}$ üzenetre. Vagyis ha a P_A és S_A kulcsokat bármilyen sorrendben egymás után alkalmazzuk M-re, visszakapjuk az M üzenetet.

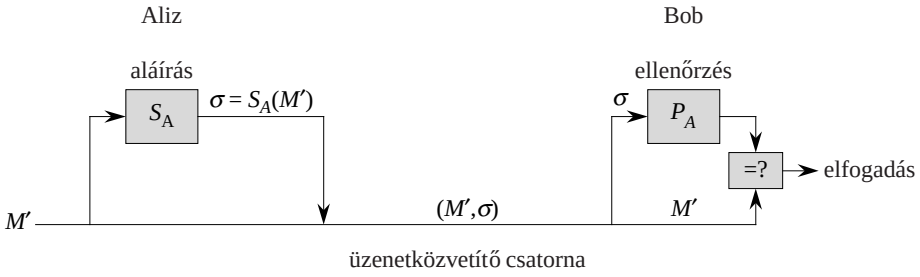
Egy nyilvános kulcsú titkosírásnál lényeges, hogy az $S_A()$ függvényt csakis Aliz legyen képes kiszámolni valamilyen elfogadhatóan rövid időn belül. Az Aliznak küldött kódolt levél titkossága és Aliz aláírásának hitelessége azon a feltételezésen alapul, hogy az $S_A()$ függvényt csak Aliz képes kiszámolni. Ezért tartja titokban Aliz az S_A kulcsot. Ha ezt nem tenné meg, elveszítené ezt a kivételezettségét, a titkosírás nem tudná biztosítani számára az egyértelműség lehetőséget. A feltételezésnek, hogy S_A kiszámolására csak Aliz képes, annak ellenére fenn kell állnia, hogy a P_A kulcsot mindenki ismerheti, és $P_A()$, az $S_A()$ inverz függvénye, gyorsan meghatározható. Egy működőképes nyilvános kulcsú titkosírás tervezésében a fő nehézséget annak a kitalálása jelenti, hogy hogyan lehet olyan rendszert találni, amelyikben a $P_A()$ függvényt felfedhetjük anélkül, hogy ezáltal a megfelelő $S_A()$ inverz függvényt elárulnánk.

Egy nyilvános kulcsú titkosírásnál a titkosítás a 31.5. ábrán látható módon történik. Tegyük fel, hogy Bob egy M titkos üzenetet akar küldeni Aliznak úgy, hogy az egy illetéktelen számára értelmetlen összevisszaságnak tűnjön. Az üzenetküldés forgatókönyve a következő:

- Bob megszerzi Aliz P_A nyilvános kulcsát (egy nyilvános könyvtárból vagy közvetlenül Aliztól).



31.5. ábra. Titkosítás a nyilvános kulcsú titkosírásnál. Bob Aliz P_A nyilvános kulcsát használva kódolja az M üzenetet, és elküldi a kódolt $C = P_A(M)$ szöveget Aliznak. Ha egy illetéktelen hozzá is jut a kódolt szöveghez, semmilyen információt nem tud nyerni M-ről. Aliz titkos kulcsa segítségével dekódolja C-t és visszakapja az eredeti $M = S_A(C)$ üzenetet.



31.6. ábra. Digitális aláírás a nyilvános kulcsú titkosírásnál. Aliz saját digitális aláírását, $\sigma = S_A(M')$-t hozzáfűzve az M' üzenethez aláírja azt, majd továbbítja az (M', σ) üzenet-aláírás párt Bobnak. Bob ellenőrzi, hogy az $M' = P_A(\sigma)$ egyenlőség teljesül-e. Ha igen, akkor elfogadja, hogy (M', σ) Aliz aláírt üzenete.

- Bob az M üzenet alapján kiszámítja a $C = P_A(M)$ ***kódolt üzenetet,*** és elküldi Aliznak.
- Amikor Aliz megkapja a C kódolt szöveget, alkalmazza rá saját S_A titkos kulcsát, és így visszakapja az eredeti $M = S_A(C)$ üzenetet.

Mivel $S_A()$ és $P_A()$ egymás inverzei, Aliz a C ismeretében ki tudja számolni az M üzenetet. De csak Aliz képes kiszámítani $S_A()$ függvényt, így csak Aliz tudja meghatározni az M üzenetet a C-ből. Az M üzenet $P_A()$ felhasználásával történő kódolása megvédi az M üzenetet attól, hogy Alizon kívül más is felfedhesse.

A digitális aláírás hasonlóan egyszerűen valósítható meg a nyilvános kulcsú titkosírás általunk használt leírása segítségével. (Meg kell jegyeznünk, hogy a digitális aláírás konstrukciójának létezik más megközelítése is, a részletekbe most nem megyünk bele.) Tegyük fel, hogy Aliz küldeni akar Bobnak egy digitálisan aláírt M' választ. A digitális aláírás forgatókönyve a 31.6. ábrának megfelelően történik.

- Aliz az S_A titkos kódja felhasználásával kiszámítja az M' üzenet ***digitális aláírását,*** $\sigma = S_A(M')$-t.
- Aliz elküldi az (M', σ) üzenet-aláírás párt Bobnak.
- Amikor Bob megkapja az (M', σ) párt, Aliz nyilvános kulcsát felhasználva ellenőrizni tudja, hogy Aliztól származik-e. Egyszerűen ellenőrzi, hogy az $M' = P_A(\sigma)$ teljesül-e. (Feltételezhető, hogy a levél tartalmazza Aliz nevét, és így Bob tudni fogja, kinek a nyilvános kulcsát kell használnia.) Ha az egyenlőség teljesül, akkor Bob azt a következtetést vonja le, hogy az M' üzenetet valóban Aliz írta alá. Ha az egyenlőség

nem teljesül, Bob arra következtet, hogy az M' üzenet vagy a σ digitális aláírás átviteli hibák miatt megsérült, vagy az (M', σ) pár hamisítási kísérlet.

Mivel a digitális aláírás szavatolja mind az aláíró személyazonosságát, mind az aláírt szöveg tartalmát, ezért egyenértékű egy kézzel írt aláírással egy írásos dokumentum végén.

A digitális aláírás fontos tulajdonsága, hogy bárki, aki csak hozzá tud férni az aláíró nyilvános kulcsához, ellenőrizheti azt. Az aláírt üzenet helyességéről bárki meggyőződhet, majd továbbküldheti másoknak, akik szintén ellenőrizhetik. Például az üzenet lehet egy elektronikus csekk, amelyet Aliz küld Bobnak. Miután Bob ellenőrizte Aliz aláírását a csekken, odaadja a csekket a bankjának, aki szintén verifikálni tudja az aláírást, és végrehajtja a kívánt pénzátutalást.

Megjegyezzük, hogy ez az aláírt üzenet nem titkosított, az üzenet „egyenesben" érkezik, és nincs védve a hamisítástól. De ha egyesítjük a titkosírás és az aláírás fentiekben vázolt menetrendjét, akkor aláírt titkos üzenetet is tudunk készíteni. Az aláíró először hozzáfűzi digitális aláírását az üzenethez, majd kódolja a kapott üzenet-aláírás párt a fogadó fél nyilvános kulcsával. A fogadó fél a saját titkos kulcsával dekódolja a kapott üzenetet, így megkapja az eredeti üzenetet és a digitális aláírást. Ekkor az aláíró nyilvános kulcsa segítségével ellenőrizni tudja az aláírás hitelességét. Ez annak a gyakorlatnak felel meg, hogy az aláírt dokumentumot borítékba teszik, amit csak a címzett bonthat fel.

Az RSA titkosírás

Mindenki, aki az ***RSA nyilvános kulcsú titkosírás*** segítségével szeretne titkosítani, a következő módon készítheti el nyilvános és titkos kulcsát:

1. Véletlenszerűen ki kell választani két nagy prímszámot, p-t és q-t úgy, hogy $p \neq q$. A p és q prímszámok legyenek például 512 bit hosszúak.
2. Ki kell számítani az $n = pq$ értéket.
3. Ki kell választani egy $\phi(n)$-hez relatív prím kis páratlan e egész számot. A (31.19) szerint $\phi(n) = (p-1)(q-1)$.
4. Ki kell számítani az e multiplikatív inverzét modulo $\phi(n)$, legyen ez az érték d. (A 31.26. következmény szerint d egyértelműen létezik. A 31.4. alfejezetben használt technika segítségével adott e és $\phi(n)$-ből d számolható.)
5. Nyilvánosságra kell hozni a $P = (e, n)$ párt, az ***RSA nyilvános kulcsát.***
6. Titokban kell tartani az $S = (d, n)$ párt, az ***RSA titkos kulcsát.***

Ebben a sémában $\mathcal{D} = \mathbf{Z}_n$. Egy M üzenet kódolása a $P(e, n)$ nyilvános kulcs szerint az alábbi kongruencia kiszámítását jelenti:

$$P(M) = M^e \pmod{n}. \tag{31.35}$$

A C kódolt üzenet dekódolása a $S(d, n)$ titkos kulcs szerint pedig az alábbi lesz:

$$S(C) = C^d \pmod{n}. \tag{31.36}$$

Ezek az egyenletek a kódolásra és az aláírásra egyaránt vonatkoznak. Az aláírásnál az üzenet küldője a kódolatlan üzenetre alkalmazza saját titkos kulcsát és nem a kódoltra. Az aláírás

ellenőrzése pedig úgy történik, hogy az üzenet olvasója az aláíró nyilvános kulcsát használja az iménti, kódolatlan üzenetre.

A kódolás és a dekódolás – a nyilvános és a titkos kulcsot használó műveletek – a 31.6. alfejezetben leírt MODULÁRIS-HATVÁNYOZÓ algoritmussal hajthatók végre. Vizsgáljuk meg az iménti műveletek futási idejét. Tegyük fel, hogy az (e, n) nyilvános kulcs és a (d, n) titkos kulcs kielégítik a $\lg e = O(1)$, $\lg d \leq \beta$ és $\lg n \leq \beta$ egyenleteket. Ekkor a nyilvános kulcs alkalmazása $O(1)$ moduláris szorzást, vagyis $O(\beta^2)$ bitműveletet, a titkos kulcs alkalmazása pedig $O(\beta)$ moduláris szorzást, vagyis $O(\beta^3)$ bitműveletet igényel.

31.36. tétel (az RSA séma korrektsége). *A* (31.35) *és* (31.36) *egyenletek* $\mathbf{Z}_n$ *olyan inverz transzformációit definiálják, amelyek kielégítik a* (31.33) *és* (31.34) *egyenleteket.*

Bizonyítás. A (31.35) és (31.36) egyenletek miatt bármely $M \in \mathbf{Z}_n$ esetén

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}.$$

Mivel e és d egymás multiplikatív inverzei modulo $\phi(n) = (p-1)(q-1)$, ezért

$$ed = 1 + k(p-1)(q-1)$$

alkalmas k egész számra. Ekkor az $M \not\equiv 0 \pmod{p}$ esetben

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} \pmod{p} \\ &\equiv M(1)^{k(q-1)} \pmod{p} \qquad \text{(a 31.31. tétel miatt)} \\ &\equiv M \pmod{p}. \end{aligned}$$

Világos, hogy az $M^{ed} \equiv M \pmod{p}$ egyenlet az $M \equiv 0 \pmod{p}$ esetben is teljesül. Ekkor

$$M^{ed} \equiv M \pmod{p}$$

minden M-re fennáll. Hasonlóképpen kapjuk, hogy

$$M^{ed} \equiv M \pmod{q}$$

minden M esetén. A kínai maradéktétel 31.29. következménye miatt az elő zőekből

$$M^{ed} \equiv M \pmod{n}$$

adódik minden M-re. ■

Az RSA titkosírás biztonságát főként a nagy számok tényezőkre bontásának nehézsége garantálja. Ha valaki tényezőkre tudná bontani a nyilvános kulcsban szereplő n modulust, akkor a nyilvános kulcs birtokában meg tudná határozni a titkos kulcsot is, hasonló módon használva a p és q tényezők ismeretét, ahogyan azt a nyilvános kulcs készítője tette. Ezért ha könnyű lenne nagy számokat prímtényezőkre bontani, akkor az RSA titkosírás feltörése is könnyű lenne. Az ellenkező irányú állítás, vagyis hogy ha nagy számok prímtényezőkre bontása nehéz, akkor az RSA titkosírás feltörése is az, nem bizonyított. De az RSA titkosírás feltörésére két évtizedes kutatómunka sem eredményezett az n modulus tényezőkre bontásánál jobb módszert. A 31.9. alfejezetben majd látni fogjuk, hogy nagy egészek tényezőkre

bontása meglepően nehéz. Véletlenszerűen választott két 512 bites prím és ezek szorzata segítségével olyan nyilvános kulcsot lehet készíteni, amelyet a jelen technológia szintjén nem lehet ésszerű időn belül feltörni. Ha nem következik be valamilyen alapvető változás a számelméleti algoritmusok tervezésében, és ha az implementálás elővigyázatosan, a javasolt szabványok betartásával történik, akkor az alkalmazásokban az RSA titkosírás nagyfokú biztonságot képes nyújtani.

Az RSA titkosírás biztonsága érdekében azonban több száz bites számokkal tanácsos dolgozni, mivel kisebb számok tényezőkre bontására léteznek hatékony algoritmusok és megfelelően gyors számítógépek. Ezen könyv készítése idején (2001) az RSA prímjei elfogadottan 768 bit és 2048 bit közöttiek voltak. Azaz a szükséges hosszúságú kulcsok előállításához képesnek kell lennünk legalább ekkora prímeket hatékonyan keresni. Ezt a problémát a 31.8. alfejezet vizsgálja.

A hatékonyság fokozása érdekében az RSA gyakran használatos „vegyes” vagy „kulcskezelő” módban, valamilyen gyors, nem nyilvános kulcsú titkosírással együtt. Ezeknél a rendszereknél a kódolás és dekódolás kulcsa ugyanaz. Ha Aliz küldeni akar Bobnak egy hosszú, titkos M üzenetet, akkor véletlenszerűen választ egyet a nem nyilvános kulcsú titkosírás K kulcsai közül, és a K szerint kódolva az M üzenetet előállítja a C kódolt szöveget. A C hasonló hosszú, mint az M, de a K elég rövid. Ezután Bob nyilvános RSA-kulcsát használva Aliz kódolja a K-t. Mivel K rövid, a $P_B(K)$ gyorsan számolható (sokkal gyorsabban, mint $P_B(M)$). Ezután Aliz elküldi Bobnak a $(C, P_B(K))$ párt, aki dekódolja a $P_B(K)$-t, hozzájut K-hoz, majd a K-val dekódolva a C-t, olvashatja az M üzenetet.

Hasonló hibrid megközelítés alkalmazható a digitális aláírásnál is. Ebben a variációban az RSA-t egy nyilvános ***ütközésmentes h hasító függvénnyel*** együtt használják. A h függvényt könnyű számolni, de a gyakorlatban mégis lehetetlen olyan M és M' üzeneteket találni, hogy $h(M) = h(M')$ legyen. A $h(M)$ értéke az M üzenet rövid (kb. 160 bites) „ujjlenyomata”. Ha Aliz alá akarja írni az M üzenetet, akkor először kiszámítja a $h(M)$ ujjlenyomatot, amelyet aztán titkos kulcsát használva kódol. Ezután az $(M, S_A(h(M)))$ párt, mint az aláírt M üzenetet elküldi Bobnak. Bob ellenőrizni tudja az aláírást: először kiszámítja $h(M)$ értékét, majd leellenőrzi, hogy ez megegyezik-e azzal, amit az $S_A(h(M))$ üzenetre Aliz nyilvános P_A kulcsát alkalmazva kap. Mivel az ujjlenyomat üzenetenként különböző, ezért az aláírás érvényességének megőrzése mellett nem lehet az üzenetet megváltoztatni.

Végül megjegyezzük, hogy ***bizonyítványok*** használatával a nyilvános kulcsok szétosztása nagyban megkönnyíthető. Például tegyük fel, hogy a T „hitelesítő hatóság” nyilvános kulcsát mindenki ismeri. Aliz – kérésére – egy aláírt üzenetet kap T-től (a bizonyítványát), ami azt állítja, hogy „Aliz nyilvános kulcsa a P_A”. Ez a bizonyítvány önmagában is hiteles, mivel a P_T kulcsot mindenki ismeri. Aliz az aláírt üzenettel együtt a bizonyítványát is elküldheti, hogy a címzettnek az aláírás ellenőrzéséhez Aliz nyilvános kulcsa rögtön rendelkezésre álljon. Mivel a kulcsot T írta alá, a címzett tudja, hogy Aliz kulcsa valóban Alizé.

Gyakorlatok

31.7-1. Tekintsük a $p = 11, q = 29, n = 319$ és $e = 3$ értékekkel definiált RSA-kulcsot. Mi lesz a titkos kulcsban a d értéke? Mi lesz az $M = 100$ üzenet titkosítva?

31.7-2. Bizonyítsuk be, hogy ha Aliz nyilvános kulcsában $e = 3$, és ha egy illetéktelen Aliz titkos kulcsa $0 < d < \phi(n)$ értékéhez hozzájut, akkor az n modulust az n bitszámától polinomiálisan függő idő alatt prímtényezőkre tudja bontani. (Bár a feladatnak nem része,

érdekes tény, hogy az eredmény e konkrét értékétől függetlenül is igaz. Ennek bizonyítása megtalálható Miller munkájában [221].)

31.7-3.⋆ Bizonyítsuk be, hogy az RSA multiplikatív a következő értelemben:

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}.$$

Ezt felhasználva lássuk be, hogy ha egy illetéktelen olyan eljárással rendelkezik, amelyik $\mathbf{Z}_n$-ből választott és P_A-val kódolt üzenetek 1 százalékának gyors dekódolására képes, akkor olyan valószínűségi algoritmust is képes konstruálni, ami minden P_A-val kódolt üzenetet nagy valószínűséggel dekódol.

⋆ 31.8. Prímtesztelés

Ebben az alfejezetben nagy prímek keresésének a problémáját vizsgáljuk. A prímek sűrűségének vizsgálatával kezdjük, majd egy nagyjából elfogadható (de korántsem tökéletes) prímtesztelő eljárással folytatjuk. Végül pedig egy gyors valószínűségi prímteszt következik, amelyet Miller és Rabin fejlesztett ki.

A prímszámok sűrűsége

Számos alkalmazás (ilyen például a titkosírás) igényel nagy, „véletlen” prímeket. Szerencsére a nagy prímek nem túlzottan ritkák, úgyhogy véletlenszerűen kiválasztott megfelelő nagyságú egészek között nem túlzottan időigényes prímeket találni. A ***prímek eloszlásfüggvénye*** – melyet $\pi(n)$-nel jelölünk – megadja az n-nél nem nagyobb prímek számát. Például $\pi(10) = 4$, mivel 10-ig 4 prím van, éspedig 2, 3, 5 és 7. Az alábbi prímszámtétel jól használható becslést ad $\pi(n)$ értékére.

31.37. tétel (prímszámtétel).

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1.$$

Az $n/\ln n$ közelítés elfogadhatóan pontos becslést ad $\pi(n)$-re kis n esetén is. Például $n = 10^9$ esetén $\pi(n) = 50\,847\,534$ és $n/\ln n \approx 48\,254\,942$, így az eltérés 6%-nál kisebb. (A számelmélettel foglalkozók számára a 10^9 egy kicsi szám.)

A prímszámtétel szerint egy véletlenszerűen választott n egész $1/\ln n$ valószínűséggel lesz prím. Vagyis ahhoz, hogy egy n-nel azonos nagyságrendű prímet találjunk, megközelítőleg $\ln n$ darab n-hez közeli véletlenszerűen választott egészet kell megvizsgálnunk. Például egy 512 bites prím kereséséhez körülbelül $\ln 2^{512} \approx 355$ véletlenszerűen választott 512 bites számot kellene tesztelni. (Ez a szám kizárólag páratlan számokat választva a felére csökkenthető.)

Az alfejezet hátralévő részében egy nagy páratlan n szám prím mivoltának eldöntésével foglalkozunk. Az egyszerűbb jelölés kedvéért feltesszük, hogy n prímtényezős alakja

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \tag{31.37}$$

ahol $r \geq 1$, $p_1, p_2, \ldots, p_r$ az n prímtényezői és $e_1, e_2, \ldots, e_r$ pozitív egészek. Ezzel a jelöléssel n akkor és csak akkor prím, ha $r = 1$ és $e_1 = 1$.

A prímtesztelés problémájára egyszerű, de hosszadalmas megoldást kínál az ***osztási próba.*** Az n számot osszuk el rendre a $2, 3, 5, \ldots, \lfloor \sqrt{n} \rfloor$ egészek mindegyikével. (A 2-nél nagyobb páros számokat most is kihagyhatjuk.) Könnyű belátni, hogy n akkor és csak akkor prím, ha egyik osztási próbálkozás sem eredményez egész számot. Feltéve, hogy minden osztási próba konstans időt igényel, a legrosszabb eset futási ideje $\Theta(\sqrt{n})$, ami az n bináris hosszának exponenciális függvénye. (Ha ugyanis n bináris alakja β bit hosszúságú, akkor $\beta = \lceil \lg(n+1) \rceil$, és így $\sqrt{n} = \Theta(2^{\beta/2})$.) Emiatt az osztási próba csak akkor elég gyors, ha n nagyon kicsi vagy van kis prímosztója. Ha az osztási próba használható, akkor megvan az az előnye is, hogy nemcsak azt lehet eldönteni vele, hogy egy n szám prím vagy összetett, de összetett esetben a szám egyik prímtényezőjét is megadja.

Ebben az alfejezetben kizárólag egy adott n szám prímségének eldöntésére koncentrálunk. Ha n összetett, nem foglalkozunk a prímtényezőkre bontásával. A 31.9. alfejezetben majd látni fogjuk, hogy egy szám prímtényezőkre való felbontásának kiszámítása nagyon időigényes. Talán meglepően hangzik, de sokkal egyszerűbb megmondani egy adott számról, hogy prím-e, mint prímtényezőire bontani, ha nem az.

Álprímek tesztelése

Most egy olyan prímtesztelő eljárást vizsgálunk, ami „majdnem mindig” működik, és gyakorlati alkalmazásokban is elég jól használható. A módszer finomított változatát, ami a „kis hiányosságot” is kiküszöböli, később mutatjuk be. Jelölje $\mathbf{Z}_n^+$ a $\mathbf{Z}_n$ nullától különböző elemeit, azaz

$$\mathbf{Z}_n^+ = \{1, 2, \ldots, n-1\}.$$

Ha n prím, akkor $\mathbf{Z}_n^+ = \mathbf{Z}_n^*$.

Azt mondjuk, hogy az n szám ***a alapú álprím,*** ha n összetett és

$$a^{n-1} \equiv 1 \pmod{n}. \tag{31.38}$$

Fermat tétele (31.31) szerint, ha n prím, akkor n minden $a \in \mathbf{Z}_n^+$ esetén kielégíti a (31.38) kongruenciát. Így, ha van olyan $a \in \mathbf{Z}_n^+$, amelyre n *nem* teljesíti a (31.38) feltételt, akkor n biztosan összetett. Meglepő módon az állítás megfordítása is *majdnem* igaz, ezért ez a feltétel majdnem tökéletes prímteszt. Megnézzük, hogy $a = 2$ esetén n kielégíti-e a (31.38) kongruenciát. Ha nem, akkor n-ről kijelentjük, hogy összetett. Egyébként azt a sejtést nyomtatjuk ki, hogy n prím (amikor is valójában csak annyit tudunk, hogy n vagy prím, vagy 2 alapú álprím).

A következő eljárás ezzel a módszerrel ellenőrzi, hogy az n szám prím-e. A 31.6. alfejezetben megismert MODULÁRIS-HATVÁNYOZÓ algoritmust is fel fogjuk használni. Az n bemenő adatról feltesszük, hogy 2-nél nagyobb páratlan egész.

ÁLPRÍM(n)

```
1  if MODULÁRIS-HATVÁNYOZÓ(2, n − 1, n) ≢ 1 (mod n)
2     then return ÖSSZETETT          ▷ Biztosan!
3     else return PRÍM               ▷ Valószínűleg!
```

Az eljárás csak egyféleképpen adhat hamis végeredményt. Ha az algoritmus azt eredményezi, hogy n összetett, akkor az állítás mindig helyes. Ha pedig azt állítja, hogy n prím, csak akkor hibázik, ha az n szám 2 alapú álprím.

Milyen gyakran téved az algoritmus? Meglepően ritkán. Például 10^3-ig csak 22 ilyen n érték van. Az első négy ilyen szám a 341, 561, 645 és az 1105. Megmutatható, hogy annak a valószínűsége, hogy a program egy véletlenszerűen választott β bit hosszúságú számnál hibás eredményt ad, 0-hoz tart, ha $\beta \to \infty$. Pomerance [244] adott nagyságú 2 alapú álprímekre vonatkozó becslését használva annak a valószínűsége, hogy az ÁLPRÍM algoritmus egy véletlenszerűen választott 512 bit hosszúságú 2 alapú álprímet szolgáltat, kisebb, mint 10^{-20}, egy 1024 bit hosszúságú egész esetén pedig az esély kisebb, mint 10^{-41}. Vagyis ha valamilyen alkalmazáshoz nagy prímet kell keresnünk, szinte sohasem tévedünk, amíg véletlenszerűen választott nagy egészekre az ÁLPRÍM algoritmus a PRÍM választ adja. De ha a prímtesztelés céljára választott számokat nem véletlenszerűen választjuk, jobb módszerre van szükségünk. Ahogy látni fogjuk, egy kis okoskodással és véletlen elemek használatával egy minden bemenő adat esetén megfelelően működő prímtesztelő eljáráshoz juthatunk.

Sajnos a tévedési lehetőségeket azzal nem tudjuk teljesen kiküszöbölni, hogy a (31.38) egyenletet egyszerűen egy másik alapú számra, például $a = 3$-ra is ellenőrizzük, mert vannak olyan összetett n egészek, amelyekre (31.38) *minden* $a \in \mathbf{Z}_n^*$ esetén teljesül. Ezeket az egészeket ***Carmichael-számoknak*** nevezzük. Az első három Carmichael-szám az 561, 1105 és az 1729. A Carmichael-számok nagyon ritkák, például 10^9-nél kisebb csak 255 van. A 31.8-2. gyakorlat rávilágít arra, hogy ez miért van így.

A továbbiakban megmutatjuk, hogy hogyan lehet prímtesztünket úgy megjavítani, hogy a Carmichael-számok ne csapjanak be bennünket.

A Miller–Rabin valószínűségi prímteszt

A Miller–Rabin prímteszt két módosítással szünteti meg az egyszerű ÁLPRÍM teszt hiányosságait.

- Egyetlen alap helyett több, véletlenszerűen választott a értéket próbál ki.
- A moduláris hatványozás kiszámításánál észleli, ha a nemtriviális négyzetgyöke 1-nek modulo n, amikor is leáll, és ÖSSZETETT eredményt ad. A 31.35. következmény szerint ekkor valóban összetett számokat kapunk.

A Miller–Rabin prímteszt pszeudokódja a következő: Legyen n egy 2-nél nagyobb páratlan szám és s az alapok száma, amelyeket véletlenszerűen választunk $\mathbf{Z}_n^+$-ből. Az eljáráshoz a 8.3. alfejezet véletlen számokat generáló VÉLETLEN programját használjuk: a VÉLETLEN$(1, n-1)$ eljárás véletlenszerűen kiválaszt egy egész számot az $[1, n-1]$ zárt intervallumból. Prímtesztünk felhasználja továbbá a TANÚ eljárást. A TANÚ(a, n) algoritmus akkor és csak akkor ad IGAZ eredményt, ha a az n összetettségének „tanúja”, azaz, ha az a segítségével (később ismertetett módon) be lehet látni, hogy az n összetett. A TANÚ algoritmus hasonlóan működik, mint az

$$a^{n-1} \not\equiv 1 \pmod{n}$$

teszt, ami (az $a = 2$ választással) az ÁLPRÍM algoritmus alapját képezte, de sokkal gyorsabb. Először bemutatjuk és indokoljuk a TANÚ algoritmus felépítését, azután pedig megmutatjuk, hogy hogyan használjuk fel a Miller–Rabin prímteszthez. Legyen $n - 1 = 2^t u$, ahol $t \geq 1$ és u páratlan egész. Vagyis az $n - 1$ binárisan ábrázolva nem más, mint az u páratlan egész bináris alakja és azután pontosan t darab nulla. Ekkor $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$, tehát az a^{n-1} mod n értékét úgy is kiszámíthatjuk, hogy először kiszámítjuk a^u mod n-et, majd egymás után t-szer négyzetre emelünk.

```
TANÚ(a, n)
1  legyen n − 1 = 2^t u, ahol t ≥ 1 és u páratlan
2  x_0 ← MODULÁRIS-HATVÁNYOZÓ(a, u, n)
3  for i ← 1 to t
4       do x_i ← x_{i−1}^2 mod n
5          if x_i = 1 és x_{i−1} ≠ 1 és x_{i−1} ≠ n − 1
6            then return IGAZ
7  if x_t ≠ 1
8    then return IGAZ
9  return HAMIS
```

A TANÚ algoritmus az $a^{n-1} \bmod n$ értéket számítja ki úgy, hogy először a második sorban meghatározza az $x_0 = a^u \bmod n$-et, majd a kapott eredményt t-szer modulárisan négyzetre emeli a 3–6. sorok **for** ciklusában. Továbbá i szerinti teljes indukcióval igazolható, hogy az $x_0, x_1, \ldots, x_t$ sorozat értékei minden $i = 0, 1, \ldots, t$-re kielégítik az $x_i \equiv a^{2^i u} \bmod n$ kongruencia-egyenletet, vagyis $x_t \equiv a^{n-1} \bmod n$ is teljesül. Azonban amikor a 4. sorban négyzetre emelés történik, a ciklus idő előtt véget érhet, ha az 5–6. sorok az 1 nemtriviális négyzetgyökét észlelik. Ha ez történik, akkor az algoritmus megáll és IGAZ értéket ad vissza. A 7–8. sorokból szintén IGAZ értékkel tér vissza az algoritmus, ha $x_t \equiv a^{n-1} \bmod n \neq 1$, ahogy az ÁLPRÍM algoritmus is ÖSSZETETT értéket adna vissza ebben az esetben. A 9. sor a HAMIS értéket szolgáltatja, amennyiben a 6. vagy a 8. sorban nem történt IGAZ visszatérés.

Most megindokoljuk, hogy ha a TANÚ(a, n) algoritmus kimenete IGAZ, akkor n összetett. Ezt az a segítségével bizonyítjuk.

Ha a TANÚ algoritmus a 8. sorban az IGAZ eredményt adja, akkor felfedezte, hogy $x_t = a^{n-1} \bmod n \neq 1$. Ha azonban n prím, akkor Fermat tétele (31.31. tétel) szerint $a^{n-1} \equiv 1 \pmod{n}$ minden $a \in \mathbf{Z}_n^+$ elemre. Emiatt n nem lehet prím, és az $a^{n-1} \bmod n \neq 1$ egyenlet a bizonyíték erre.

Ha a TANÚ algoritmus a 6. sorban ad IGAZ eredményt, akkor pedig felfedezte, hogy x_{i-1} az 1-nek nemtriviális négyzetgyöke modulo n, mivel ekkor $x_{i-1} \not\equiv \pm 1 \pmod{n}$, ugyanakkor $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$ teljesül. A 31.35. következmény azt állítja, hogy n csak akkor lehet az 1-nek nemtriviális négyzetgyöke modulo n, ha összetett. Így, ha x_{i-1} az 1-nek nemtriviális négyzetgyöke modulo n, akkor ez n összetettségét is bizonyítja.

Ezzel beláttuk, hogy a TANÚ algoritmus megfelelően működik. Azaz, ha a TANÚ(a, n) algoritmus IGAZ értéket eredményez, akkor n biztosan összetett, és n összetettségét a és n segítségével könnyen be is láthatjuk.

Ezen a ponton a TANÚ viselkedésének egy alternatív leírását is megmutatjuk, mint az $X = \langle x_0, x_1, \ldots, x_t \rangle$ sorozat egy függvényét. Ez a leírás később, a Miller–Rabin prímteszt hatékonyságának elemzésénél még nagyon hasznos lesz. Vegyük észre, hogy ha $x_i = 1$ valamilyen $0 \leq i < t$-re, akkor a TANÚ algoritmus nem biztos, hogy kiszámítja a sorozat hátralévő részét. Amennyiben ez a helyzet, az $x_{i+1}, x_{i+2,\ldots,x_t}$ sorozat minden eleme 1, vagyis az X sorozat ezen elemeit 1-eseknek vehetjük. Ekkor négy esetet lehet megkülönböztetni:

1. $X = \langle \ldots, d \rangle$, ahol $d \neq 1$: az X sorozat utolsó tagja nem 1. Ekkor a visszatérő érték IGAZ. A Fermat-tétel miatt ekkor a az n összetettségének egy tanúja.
2. $X = \langle 1, 1, \ldots, 1 \rangle$: az X sorozat minden tagja 1-es. A visszatérő érték HAMIS. Ekkor a nem tanúsítja n összetettségét.

3. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: az X sorozat utolsó tagja 1 és az utolsó 1-estől különböző tagja -1. A visszatérő érték HAMIS. Ekkor a nem tanúsítja n összetettségét.

4. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$, ahol $d \neq \pm 1$: az X sorozat utolsó tagja 1-es, de az utolsó 1-estől különböző tagja nem -1. Ekkor a visszatérő érték IGAZ és a az n összetettségének egy tanúja, mivel d az 1 nemtriviális négyzetgyöke.

A következőkben a TANÚ algoritmusra épülő Miller–Rabin prímtesztet vizsgáljuk.

MILLER–RABIN(n, s)

```
1  for j ← to s
2        do a ← VÉLETLEN(1, n − 1)
3           if TANÚ(a, n)
4              then return ÖSSZETETT        ▷ Biztosan.
5  return PRÍM                             ▷ Majdnem biztosan.
```

A Miller–Rabin prímteszt n összetettségét egy valószínűségi kereséssel bizonyítja. Az első sorban kezdődő fő ciklus a $\mathbf{Z}_n^+$ halmazból véletlenszerűen kiválaszt s darab a értéket (2. sor). Ha a kiválasztott a számok valamelyike tanúja az n összetettségének, akkor a MILLER–RABIN algoritmus a 4. sorban ÖSSZETETT eredményt ad. Ez az eredmény mindig helyes, és ezt a TANÚ algoritmus szavatolja. Ha az s darab szám egyike sem tanú, akkor a MILLER–RABIN algoritmus feltételezi, hogy nem is lehet tanút találni, ezért n prím. Látni fogjuk, hogy ha s elég nagy, akkor ez az eredmény nagy valószínűséggel helyes. De mindig van egy kis esély arra, hogy az eljárás szerencsétlenül választotta ki az a számokat, vagyis léteznek tanúk, csak nem találtuk meg őket.

A MILLER–RABIN teszt működését az $n = 561$ Carmichael-számon mutatjuk be. Ekkor $n - 1 = 560 = 2^4 \cdot 35$. Ha a választott alap $a = 7$, akkor a 31.4. ábra szerint a TANÚ algoritmus kiszámítja az $x_0 \equiv a^{35} \equiv 241 \pmod{561}$ értéket és az $X = \langle 241, 298, 166, 67, 1 \rangle$ sorozatot. Vagyis az utolsó négyzetre emelésnél felfedezi az 1 nemtriviális négyzetgyökét, mivel $a^{280} \equiv 67 \pmod{n}$ és $a^{560} \equiv 1 \pmod{n}$. Ezért az $a = 7$ az n összetettségének tanúja, a TANÚ$(7, n)$ algoritmus IGAZ, a MILLER–RABIN algoritmus pedig ÖSSZETETT eredményt ad.

Ha n egy β bit hosszúságú szám, akkor a MILLER–RABIN algoritmus $O(s\beta)$ aritmetikai és $O(s\beta^3)$ bitműveletet hajt végre, mivel aszimptotikusan legfeljebb s moduláris hatványozás történik.

A MILLER–RABIN prímteszt tévedési aránya

Ha a MILLER–RABIN algoritmus a PRÍM eredményt adja, akkor van még egy kis esély arra, hogy hibázott. Az ÁLPRÍM algoritmustól eltérően azonban a hibák száma nem függ n-től. Tehát ebben az eljárásban nincsenek rossz bemenő adatok. Ehelyett a hiba az s nagyságától és az a alap szerencsés választásától függ. Mivel minden ilyen tesztelés sokkal szigorúbb, mint a (31.38) egyszerű ellenőrzése, így általános elvi alapon elvárhatjuk, hogy egy véletlenszerűen választott n egészre a hibázás valószínűsége kicsi legyen. A következő tétel sokkal pontosabb állítást tartalmaz.

31.38. tétel. *Az n páratlan összetett számnak legalább $(n-1)/2$ darab összetettségét igazoló tanúja van.*

Bizonyítás. A bizonyítás során megmutatjuk, hogy a nemtanú számokból nincs $(n-1)/2$-nél több, amiből a tétel állítása már következik.

Először is vegyük észre, hogy minden nemtanú $\mathbf{Z}_n^*$ eleme, mivel minden a nemtanú esetén $a^{n-1} \equiv 1 \pmod{n}$, vagy másképp írva $a \cdot a^{n-2} \equiv 1 \pmod{n}$ teljesül. Vagyis az $ax \equiv 1 \pmod{n}$ lineáris kongruenciának létezik megoldása, nevezetesen az a^{n-2}. A 31.21. következmény miatt $\text{lnko}(a, n) \mid 1$, amiből nyilván $\text{lnko}(a, n) = 1$ adódik. Azt kapjuk tehát, hogy minden a nemtanú esetén $a \in \mathbf{Z}_n^*$.

A bizonyítást úgy fejezzük be, hogy megmutatjuk, a nemtanú számok $\mathbf{Z}_n^*$ valamely B valódi részcsoportjának is elemei. Emlékeztetünk, hogy B pontosan akkor valódi részcsoportja $\mathbf{Z}_n^*$-nek, ha részcsoportja $\mathbf{Z}_n^*$-nek, de nem egyenlő vele. A 31.16. következmény miatt ekkor $|B| \le |\mathbf{Z}_n^*|/2$. Mivel $|\mathbf{Z}_n^*| \le n-1$, ezért $|B| \le (n-1)/2$. Vagyis a nemtanú elemekből legfeljebb $(n-1)/2$ van, azaz a tanúk száma legalább $(n-1)/2$.

Most megmutatjuk, hogy hogyan találhatunk egy, a $\mathbf{Z}_n^*$ összes nemtanú elemét tartalmazó B valódi részcsoportot. Két esetet különböztetünk meg.

1. eset: Létezik olyan $x \in \mathbf{Z}_n^*$, hogy

$$x^{n-1} \not\equiv 1 \pmod{n}.$$

Más szavakkal n nem Carmichael-szám. Ahogy korábban megjegyeztük, a Carmichael-számok nagyon ritkán fordulnak elő, vagyis az 1. eset a fontos „gyakorlatban előforduló eset", amikor is az n értékét véletlenszerűen választjuk, majd prímségét teszteljük.

Legyen $B = \{b \in \mathbf{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. A B halmaz nem üres, ezért $1 \in B$. Mivel B zárt a szorzásra modulo n, ezért a 31.14. tétel miatt a $\mathbf{Z}_n^*$ egy részcsoportja. Vegyük észre, hogy minden nemtanú elem B-ben van, ugyanis minden nemtanú a számra $a^{n-1} \equiv 1 \pmod{n}$. Mivel $x \in \mathbf{Z}_n^* - B$, így B a $\mathbf{Z}_n^*$ valódi részcsoportja.

2. eset: Minden $x \in \mathbf{Z}_n^*$ elemre

$$x^{n-1} \equiv 1 \pmod{n}. \tag{31.39}$$

Más szavakkal n egy Carmichael-szám. Ez az eset a gyakorlatban nagyon ritkán fordul elő. Szerencsére az Álprím algoritmussal ellentétben a Miller–Rabin algoritmus hatékonyan képes a Carmichael-számok összetettségét eldönteni.

Először belátjuk, hogy n nem lehet prímhatvány. Tegyük fel ugyanis indirekt módon, hogy $n = p^e$ teljesül valamely p prím és $e > 1$ egész esetén. Nyilván, az n páratlanságából p páratlansága is következik. Ekkor a 31.32. tétel miatt a $\mathbf{Z}_n^*$ csoport ciklikus, vagyis létezik olyan g generátor eleme, amire $\text{ord}_n g = |\mathbf{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}$. A (31.39) egyenlőségből azt kapjuk, hogy $g^{n-1} \equiv 1 \pmod{n}$. Ekkor, a diszkrét logaritmus tételét használva (31.33 tétel, $y = 0$ választással) $n - 1 \equiv 0 \pmod{\phi(n)}$ adódik, azaz

$$(p-1)p^{e-1} \mid p^e - 1$$

teljesül. Ez azonban lehetetlen, hiszen $e > 1$, ezért $(p-1)p^{e-1}$ osztható a p prímmel, $p^e - 1$ pedig nem. Vagyis n nem lehet prímhatvány.

Mivel az n páratlan összetett szám nem prímhatvány, ezért alkalmas $n_1, n_2 > 1$ relatív prím egészek szorzatára bontható. (Ez többféleképpen is teljesülhet, és nem számít, hogy melyik felbontást választjuk. Ha például $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, akkor az $n_1 = p_1^{e_1}$ és $n_2 = p_2^{e_2} \cdots p_r^{e_r}$ választás megfelelő lesz.)

Emlékezzünk vissza, hogy a t és u számokat az $n-1 = 2^t u$ egyenlet definiálja, ahol $t \geq 1$ egész és u páratlan egész, valamint egy $a \in \mathbf{Z}_n^+$ elemhez a TANÚ algoritmus az

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

sorozatot eredményezi, ahol a hatványozás modulo n értend ő.

A (v, j) egész párt ***elfogadható*** párnak nevezzük, ha $v \in \mathbf{Z}_n^*$, $j \in \{0, 1, \ldots, t\}$ és

$$v^{2^j u} \equiv -1 \pmod{n}.$$

Elfogadható párok nyilván léteznek, hiszen u páratlan, ezért a $v = n-1$ és $j = 0$ választással $(n-1, 0)$ megfelelő lesz. Válasszuk a legnagyobb olyan j értéket, amihez található valamilyen (v, j) elfogadható pár és rögzítsük ezt a v-t. Legyen

$$B = \{x \in \mathbf{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Mivel B zárt a szorzásra modulo n, ezért a $\mathbf{Z}_n^*$ egy részcsoportja. A 31.16. következmény miatt $|B|$ osztja $|\mathbf{Z}_n^*|$-t. Minden nemtanú elem B-ben van, hiszen egy nemtanú elem által generált X sorozat vagy csupa 1-esből áll, vagy a j maximális választása miatt legkésőbb a j. pozíción van benne -1-es. (Amennyiben (a, j') egy elfogadható pár, ahol a egy nemtanú szám, akkor a j maximális választása miatt $j' \leq j$.)

Most v létezését kihasználva megmutatjuk, hogy $\mathbf{Z}_n^* - B$ nem üres. Mivel $v^{2^j u} \equiv -1 \pmod{n}$, így a kínai maradéktétel 31.29. következménye miatt $v^{2^j u} \equiv -1 \pmod{n_1}$ is teljesül. A 31.28. következmény szerint ekkor létezik olyan w, amelyre

$$\begin{aligned} w &\equiv v \pmod{n_1}, \\ w &\equiv -1 \pmod{n_2} \end{aligned}$$

egyszerre teljesül. Emiatt

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1}, \\ w^{2^j u} &\equiv 1 \pmod{n_2}. \end{aligned}$$

A 31.29. következmény miatt a $w^{2^j u} \not\equiv 1 \pmod{n_1}$ kongruenciából $w^{2^j u} \not\equiv 1 \pmod{n}$, valamint a $w^{2^j u} \not\equiv -1 \pmod{n_2}$ kongruenciából $w^{2^j u} \not\equiv -1 \pmod{n}$ adódik. Ezért $w^{2^j u} \not\equiv \pm 1 \pmod{n}$, azaz $w \notin B$.

Már csak azt kell megmutatni, hogy $w \in \mathbf{Z}_n^*$, amit külön-külön modulo n_1 és modulo n_2 esetén bizonyítunk. Mivel $v \in \mathbf{Z}_n^*$, így $\text{lnko}(v, n) = 1$ és $\text{lnko}(v, n_1) = 1$. Ha v-nek n-nel nincs közös osztója, akkor n_1-gyel sincs. Tudjuk, hogy $w \equiv v \pmod{n_1}$, ezért $\text{lnko}(w, n_1) = 1$ is teljesül. Modulo n_2 vizsgálódva vegyük észre, hogy a $w \equiv 1 \pmod{n_2}$ kongruenciából $\text{lnko}(w, n_2) = 1$ következik. Az iménti eredményeket a 31.6. tétel segítségével összerakva azt kapjuk, hogy $\text{lnko}(w, n_1 n_2) = \text{lnko}(w, n) = 1$. Ez azt jelenti, hogy $w \in \mathbf{Z}_n^*$.

Vagyis $w \in \mathbf{Z}_n^* - B$, így a 2. eset vizsgálatát azzal a konklúzióval zárjuk, hogy B a $\mathbf{Z}_n^*$ valódi részcsoportja.

Ezzel mindkét esetben beláttuk, hogy az n összetettségét tanúsító egészek száma legalább $(n-1)/2$. ■

31.39. tétel. *Legyen $n > 2$ páratlan egész, s pedig pozitív egész. A* MILLER–RABIN(n, s) *teszt tévedési valószínűsége legfeljebb* 2^{-s}.

Bizonyítás. A 31.38. tételből következik, hogy ha n összetett, akkor az 1–4. sorok **for** ciklusának minden újabb végrehajtása legalább 1/2 valószínűséggel felfedez egy x tanút, amely az n összetettségét igazolja. A MILLER–RABIN algoritmus csak akkor téved, ha olyan balszerencsés, hogy az s darab iterációs ciklus egyikében sem talál az n összetettségét bizonyító tanút. Egy ilyen balszerencse-sorozat valószínűsége legfeljebb 2^{-s}. ■

Vagyis az $s = 50$ választás már majdnem minden elképzelhető alkalmazásnál megfelel. Ha a MILLER–RABIN teszttel *véletlenszerűen választott* nagy egészek között akarunk prímeket találni, akkor megmutatható (de ennek bizonyításától itt eltekintünk), hogy kis s érték (például 3) választása esetén sem valószínű, hogy helytelen eredményt kapunk. Ugyanis egy véletlenszerűen választott n páratlan számhoz várhatóan $(n-1)/2$-nél jóval kevesebb nemtanú tartozik. Ha viszont az n számot nem véletlenszerűen választjuk, akkor az eddigi legjobb eredmény szerint – ami a 31.38. tétel tovább javított változata – n-hez legfeljebb $(n-1)/4$ nemtanú tartozik. Továbbá léteznek olyan n számok, amelyekhez pontosan $(n-1)/4$ nemtanú található.

Gyakorlatok

31.8-1. Bizonyítsuk be, hogy ha $n > 1$ páratlan egész nem prím vagy prímhatvány, akkor létezik az 1-nek nemtriviális négyzetgyöke modulo n.

31.8-2.⋆ Az Euler-tétel kissé általánosabb alakja a következő:

$$a^{\lambda(n)} \equiv 1 \pmod{n} \text{ minden } a \in \mathbf{Z}_n^*\text{-re},$$

ahol $n = p_1^{e_1} \cdots p_r^{e_r}$ és $\lambda(n)$ definíció szerint

$$\lambda(n) = \text{lkkt}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})). \tag{31.40}$$

Igazoljuk, hogy $\lambda(n) \mid \phi(n)$. Egy n összetett szám $\lambda(n) \mid n-1$ esetén biztosan Carmichael-szám. A legkisebb Carmichael-szám az $561 = 3 \cdot 11 \cdot 17$. Itt $\lambda(n) = \text{lkkt}(2, 10, 16) = 80$, ami 560 osztója. Mutassuk meg, hogy a Carmichael-számok olyan négyzetmentes számok (azaz nem oszthatók semelyik prím négyzetével sem), amelyeknek legalább három különböző prímtényezőjük van. Emiatt a Carmichael-számok nem túl gyakoriak.

31.8-3. Bizonyítsuk be, hogy ha x az 1 nemtriviális négyzetgyöke modulo n, akkor $\text{lnko}(x-1, n)$ és $\text{lnko}(x+1, n)$ mindegyike n nemtriviális osztója.

⋆ 31.9. Egészek prímfelbontása

Tegyük fel, hogy az n egész számot ***faktorizálni*** akarjuk, azaz prímek szorzatára akarjuk bontani. Az előző fejezet prímtesztje n összetettségét ugyan elárulja nekünk, de általában semmit sem mond n prímtényezőiről. Egy nagy n egész számot prímtényezőire bontani általában sokkal nehezebb feladat, mint egyszerűen csak azt eldönteni, hogy prím vagy összetett. A mai szuperszámítógépekkel és a legjobb ismert algoritmusokkal sem tudunk ugyanis akármilyen 1024 bites számot prímtényezőkre bontani.

Pollard ρ-heurisztikája

Egy konstans B egész számig elvégzett osztási próba garantáltan tényezőire bont minden számot B^2-ig. Ugyanennyi munkával a következő eljárás B^4-ig megteszi ezt (ha nem vagyunk túlzottan balszerencsések). Mivel az eljárás heurisztikus, sem a futási ideje, sem a sikere nem garantált, a gyakorlatban mégis nagyon eredményesen használható.

```
POLLARD-ρ(n)
 1  i ← 1
 2  x_1 ← VÉLETLEN(0, n − 1)
 3  y ← x_1
 4  k ← 2
 5  while IGAZ
 6        do i ← i + 1
 7           x_i ← (x_{i−1}^2 − 1) mod n
 8           d ← lnko(y − x_i, n)
 9           if d ≠ 1 és d ≠ n
10             then print d
11           if i = k
12             then y ← x_i
13                  k ← 2k
```

Az eljárás a következőképpen működik: Az 1–2. sorokban i értéke 1, x_1 pedig a $\mathbf{Z}_n$ egy véletlenszerűen választott eleme lesz. Az 5. sorban kezdődő **while** végtelen ciklus az n osztóit keresi. A **while** ciklus minden újabb iterációnál az

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \tag{31.41}$$

rekurziót alkalmazza a 7. sorban az

$$x_1, x_2, x_3, x_4, \ldots \tag{31.42}$$

végtelen sorozat soron következő elemének kiszámítására, ahol az i értéke a 6. sornak megfelelően egyesével nő. Az x_i változók csak az érthetőség kedvéért indexeltek, a program ugyanúgy működik, ha az indexelést elhagyjuk, mivel mindig csak a legutóbbi x_i értékre van szükség. Ezzel a megjegyzéssel az eljárás csak konstans számú memóriát foglal.

A program az x_i legutóbb kiszámított értékét megfelelő gyakorisággal elmenti az y változóba. Egészen pontosan csak a kettőhatvány indexű

$$x_1, x_2, x_4, x_8, x_{16}, \ldots$$

számokat tároljuk, ugyanis a 3. sor elmenti x_1, a 12. sor pedig x_k értékét, ha $i = k$. A k értéke 2 lesz a 4. sorban, majd megduplázódik a 13. sorban, miután y is új értéket kap. Ezért k az $1, 2, 4, 8, \ldots$ értékeket veszi fel, és mindig az ennek megfelelő indexű x_k számot tárolja y-ban.

A 8–10. sorokban az algoritmus az y-ban tárolt érték és az x_i aktuális értéke segítségével n egyik prímtényezőjét próbálja megtalálni. Pontosabban szólva a 8. sorban az algoritmus kiszámolja a $d = \text{lnko}(y - x_i, n)$ legnagyobb közös osztót. Ha d az n nemtriviális osztója (amit az algoritmus a 9. sorban ellenőriz), akkor a 10. sorban kinyomtatja d értékét.

Ez a faktorizáló eljárás első ránézésre némiképp misztikusnak tűnik. Azonban meg kell jegyeznünk, hogy a POLLARD-ρ algoritmus soha nem ad helytelen választ. Ha valamilyen számot nyomtat, akkor az ténylegesen n nemtriviális osztója. De az is lehetséges, hogy a POLLARD-ρ algoritmus semmit sem nyomtat ki. Nincs rá garancia, hogy ad valamilyen eredményt. Látni fogjuk azonban, hogy jó okunk van azt feltételezni, hogy a POLLARD-ρ algoritmus a **while** ciklus megközelítőleg $\sqrt{p}$ iterációja után megtalálja n egyik p tényezőjét. Vagyis összetett n esetén az eljárás közelítőleg $n^{1/4}$ lépés után várhatóan az n prímtényezőire bontásához elegendő osztót talál, hiszen az n minden p prímtényezője – esetleg a legnagyobbat kivéve – $\sqrt{n}$-nél kisebb.

Elemezzük az eljárást! Először vizsgáljuk meg azt, hogy mennyi ideig tart, amíg egy véletlenszerűen választott sorozatban modulo n ismétlődés lép fel. Mivel $\mathbf{Z}_n$ véges és a (31.42) sorozat minden egyes értéke csak az előző értéktől függ, a (31.42) sorozatban előbb-utóbb lesz ismétlődés. Amikor elérünk egy olyan x_i értéket, amelyre $x_i = x_j$ valamelyik $j < i$ indexre teljesül, akkor egy ciklusban vagyunk, mivel $x_{i+1} = x_{j+1}, x_{i+2} = x_{j+2}$ stb. A ρ-heurisztika elnevezés oka az (amint a 31.7. ábra mutatja), hogy az $x_1, x_2, \ldots, x_{j-1}$ rajzolható egy ρ betű „száraként" , míg az $x_j, x_{j+1}, \ldots, x_i$ ciklus a ρ „teste".

Vizsgáljuk meg, hogy mennyi ideig tart, amíg az x_i sorozatban ismétlődés kezdődik. Bár nem pontosan erre van szükségünk, a későbbiekben látni fogjuk, hogy gondolatmenetünk hogyan módosítható.

A becsléshez tegyük fel, hogy az

$$f_n(x) = (x^2 - 1) \bmod n$$

függvény „véletlen" függvényként viselkedik. Természetesen ez nem fedi a valóságot, de ezzel a feltételezéssel olyan eredményre jutunk, ami összhangban van a POLLARD-ρ algoritmus megfigyelt viselkedésével. Az egyes x_i értékekre úgy tekinthetünk, mint egyenletes eloszlást feltételezve a $\mathbf{Z}_n$ halmazból egymástól függetlenül választott elemekre. Az 5.4.1. alfejezetben látott születésnapi paradoxon vizsgálata szerint ekkor a sorozat várhatóan $\Theta(\sqrt{n})$ lépésszám után kezd valahonnan ismétlődni.

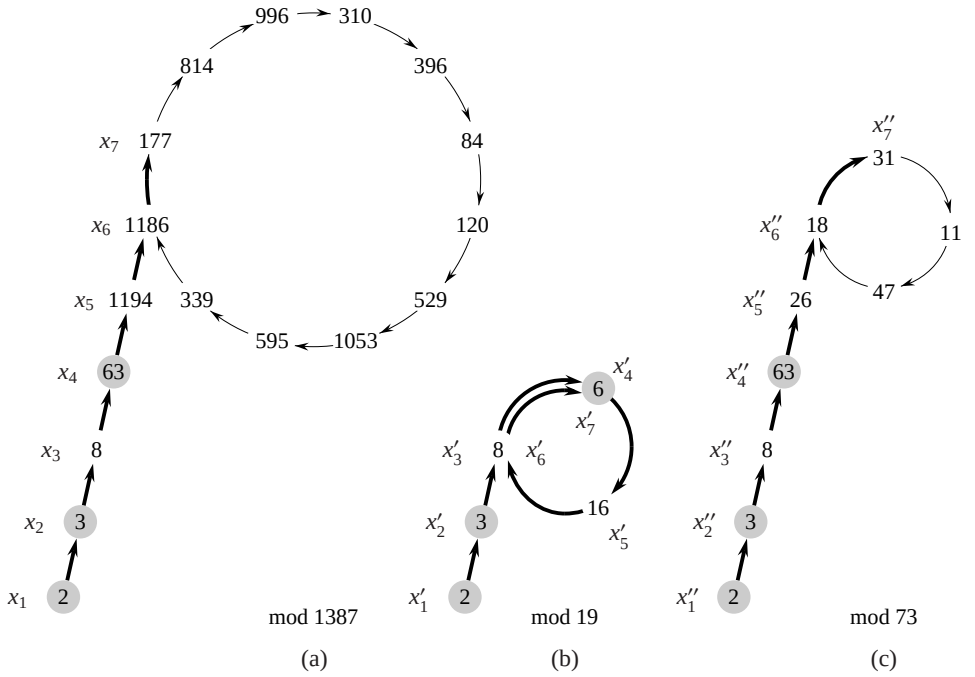
Ezután nézzük meg a szükséges módosításokat. Legyen p az n nemtriviális tényezője úgy, hogy $\text{lnko}(p, n/p) = 1$. Például ha n prímtényezős alakja $p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, akkor legyen $p{=}p_1^{e_1}$ egy ilyen tényező. (Ha $e_1 = 1$, akkor p az n legkisebb prímtényezője. Ezt érdemes lesz megjegyezni.)

Az $\langle x_i \rangle$ sorozatból elkészítjük az $\langle x_i' \rangle$ modulo p sorozatot, ahol

$$x_i' = x_i \bmod p$$

minden i-re.

Továbbá, mivel f_n definíciójában csak moduláris műveletek (négyzetre emelés és kivonás) találhatók, x_{i+1}' értéke x_i'-ből is könnyen számolható. A sorozat „modulo p" verziója kevesebb elemet tartalmaz, mint a modulo n verzió. Az alábbiakban megmutatjuk, hogy ámbár az $\langle x_i' \rangle$ sorozat elemeit nem kell explicit módon kiszámítani, a sorozat jól definiált és ugyanazzal a rekurzióval adható meg, mint $\langle x_i \rangle$:



31.7. ábra. Pollard ρ-heurisztikája. **(a)** Az $x_{i+1} \leftarrow (x_i^2 - 1) \bmod 1387$ rekurziós sorozat értékei az $x_1 = 2$ kezdőértékkel. Az 1387 prímtényezős alakja $19 \cdot 73$. A vastag nyilak a 19-es tényező felfedezéséig jelzik az iteráció lépéseit. A vékony nyilak az iteráció korábban el nem ért értékeit mutatják, megjelenítve a ρ formát. A sötét mezőbe írt számok a POLLARD-ρ algoritmus y-ban tárolt értékei. A 19-es tényezőt az $x_7 = 177$ elérése után fedezi fel az algoritmus, amikor a $\text{lnko}(63 - 177, 1387) = 19$ értéket kiszámolja. Az első ismétlődő x érték az 1186, de a 19-es tényezőt már előbb felfedezi az algoritmus. **(b)** Ugyanazon rekurzió lépései modulo 19. Minden (a)-beli x_i érték kongruens az itteni x'_i értékkel modulo 19. Például az $x_4 = 63$ és az $x_7 = 177$ számok mindegyike 6-tal kongruens modulo 19. **(c)** A rekurzió értékei modulo 73. Minden (a)-beli x érték kongruens az itteni x''_i értékkel modulo 73. A kínai maradéktétel szerint minden (a)-beli csomópont megfelel egy (b) és egy (c)-beli csomópont párnak.

$$\begin{aligned}
x'_{i+1} &= x_{i+1} \bmod p \\
&= f_n(x_i) \bmod p \\
&= ((x_i^2 - 1) \bmod n) \bmod p \\
&= (x_i^2 - 1) \bmod p \qquad \text{(a 31.1-6. gyakorlatból)} \\
&= ((x_i \bmod p)^2 - 1) \bmod p \\
&= ((x'_i)^2 - 1) \bmod p \\
&= f_p(x'_i) \, .
\end{aligned}$$

A korábbiakhoz hasonló érvelést használva azt kapjuk, hogy az $\langle x'_i \rangle$ sorozatban az ismétlődés megkezdése előtti lépésszám $\Theta(\sqrt{p})$. Ha p kicsi az n-hez képest, akkor az $\langle x'_i \rangle$ sorozat sokkal hamarabb kezdi el az ismétlődést, mint az $\langle x_i \rangle$. Ugyanis az $\langle x'_i \rangle$ sorozat már akkor elkezdi az ismétlődést, amikor két eleme modulo p kongruens (modulo n helyett). Ezt mutatja a 31.7. ábra (b) és (c) része.

Legyen t az $\langle x'_i \rangle$ sorozat első ismétlődő elemének indexe, $u > 0$ pedig jelölje a keletkező ciklus hosszát. Tehát t és $u > 0$ azok a legkisebb egészek, amelyekkel $x'_{t+i} = x'_{t+u+i}$ minden

$i \geq 0$ esetén. Az iménti érvelés szerint t és u mindegyikének várható értéke $\Theta(\sqrt{p})$. Jegyezzük meg, hogy ha $x'_{t+i} = x'_{t+u+i}$, akkor $p \mid x_{t+u+i} - x_{t+i}$, ahonnan $\text{lnko}(x_{t+u+i} - x_{t+i}, n) > 1$ következik.

Ezért amikor a POLLARD-ρ algoritmus $k \geq t$ esetén az x_k értéket y-ba menti, akkor y mod p mindig a modulo p körön van. (Új y érték esetén az is a modulo p körön lesz.) Vagyis k az u-nál előbb-utóbb nagyobb lesz, az algoritmus végigmegy a modulo p körön, miközben az y értéke nem változik. Az algoritmus akkor fedezi fel n egyik tényező jét, amikor x_i „beleszalad" az előzőleg tárolt y értékbe modulo p, azaz, ha $x_i \equiv y \pmod{p}$.

A megtalált tényező feltehetően a p, de az is előfordulhat, hogy a p többese. Mivel a t és az u várható értéke egyaránt $\Theta(\sqrt{p})$, így egy tényező felfedezéséhez várhatóan $\Theta(\sqrt{p})$ lépés szükséges.

Két oka is van, hogy az algoritmus nem egészen úgy működik, ahogy várnánk. El őször is, a heurisztikus elemzés futási ideje szigorúan véve nem pontos. Lehet, hogy a modulo p ciklusidő jóval hosszabb $\sqrt{p}$-nél. Ekkor az algoritmus ugyan helyesen működik, de a kívántnál sokkal lassúbb lesz. A gyakorlatban ez nem okoz gondot. A másik problémát az jelenti, ha az algoritmus n osztójaként mindig az 1 és n triviális osztókat találja. Tegyük fel például, hogy $n = pq$, ahol p és q prímek. El őfordulhat, hogy t és u értékei a p és q esetekben megegyeznek, vagyis a p és q tényez ők egyszerre, ugyanabban a lnko műveletben bukkannak el ő. Amennyiben a faktorok egyidejűleg bukkannak el ő, a triviális $n = pq$ tényezőt találtuk, ami hasznavehetetlen. A gyakorlatban ez sem tűnik valódi problémának. Ha szükséges, a heurisztikus eljárást újrakezdhetjük egy másik $x_{i+1} \leftarrow x_i^2 - c \bmod n$ rekurzióval. (A $c = 0$ és $c = 2$ értékek itt nem részletezett okokból kerülend ők, de más értékek megfelelnek.)

Természetesen ez az elemzés csak heurisztikus és nem precíz, mivel a rekurzió nem teljesen véletlenszerű. Az algoritmus mégis jól működik a gyakorlatban és olyan hatékony, mint ahogy azt a heurisztikus elemzés mutatja. Ez egy olyan „kiválasztásos módszer", amely nagy számok kis prímtényez őinek megkeresésére alkalmas. Egy β bit hosszúságú n összetett szám teljes prímfelbontásához csak az $\lfloor n^{1/2} \rfloor$-nél kisebb prímeket kell megtalálnunk, így a POLLARD-ρ algoritmus várhatóan legfeljebb $n^{1/4} = 2^{\beta/4}$ aritmetikai és legfeljebb $n^{1/4}\beta^2 = 2^{\beta/4}\beta^2$ bitműveletet igényel. A POLLARD-ρ algoritmus gyakran legvonzóbbnak tartott tulajdonsága éppen az, hogy várhatóan $\Theta(\sqrt{p})$ aritmetikai művelettel képes az n egy kis prímtényez őjét megtalálni.

Gyakorlatok

31.9-1. A 31.7(a) ábra adataival mikor nyomtatja ki a POLLARD-ρ algoritmus, hogy 1387 egyik tényezője 73?

31.9-2. Tegyük fel, hogy adott egy $f : \mathbf{Z}_n \to \mathbf{Z}_n$ függvény és egy $x_0 \in \mathbf{Z}_n$ kezdeti érték. Legyen $x_i = f(x_{i-1})(i = 1, 2, \ldots)$. Legyenek továbbá t és $u > 0$ a legkisebb olyan egészek, amelyekre $x_{t+i} = x_{t+u+i}(i = 0, 1, \ldots)$. A POLLARD-$\rho$ algoritmus jelölései szerint t a ρ szárának, u pedig a ρ testének (ciklusának) hossza. Adjunk gyors algoritmust a t és u értékek pontos meghatározására és elemezzük a futási id őt.

31.9-3. Várhatóan hány lépésre van szüksége a POLLARD-ρ algoritmusnak egy p^e alakú tényező felfedezéséhez, ha p prím és $e > 1$?

31.9-4.$\star$ A bemutatott POLLARD-ρ algoritmus egyik hátránya, hogy minden egyes rekurziós lépésben megköveteli a lnko kiszámítását. Egy javaslat szerint halmozni kellene az lnko-számolásokat: szorozzunk össze különböz ő x_i értékeket és x_i helyett használjuk a szorzatot

a lnko kiszámításakor. Írjuk le pontosan, hogyan lehetne ezt az ötletet megvalósítani, miért működik, és mi lenne a leghatékonyabb halmazméret egy β bit nagyságú n egész esetén.

Feladatok

31-1. Bináris lnko algoritmus

A legtöbb számítógépen bináris egészek kivonása, paritástesztje (páros vagy páratlan) és felezése sokkal gyorsabban elvégezhető, mint a maradékok kiszámítása. Ezért célszerű megvizsgálni a ***bináris lnko algoritmust,*** ami az euklideszi algoritmus használata során elkerüli a maradékokkal való számolást.

a. Bizonyítsuk be, hogy ha a és b párosak, akkor $\text{lnko}(a, b) = 2\,\text{lnko}(a/2, b/2)$.

b. Bizonyítsuk be, hogy ha a páratlan és b páros, akkor $\text{lnko}(a, b) = \text{lnko}(a, b/2)$.

c. Bizonyítsuk be, hogy ha a és b páratlanok, akkor $\text{lnko}(a, b) = \text{lnko}((a - b)/2, b)$.

d. Tegyük fel, hogy minden egyes kivonás, paritásteszt és felezés a számítógépen egységnyi időt vesz igénybe. Tervezzünk gyors – $O(\lg a)$ futási idejű – bináris lnko algoritmust adott $a \geq b$ egész bemenő adatok esetén.

31-2. Az euklideszi algoritmus bitműveletei elemzése

a. Tekintsük az egészek maradékos osztásakor alkalmazott közönséges „papír-ceruza” algoritmust: az a számot elosztjuk b-vel, kapjuk a q hányadost és az r maradékot. Mutassuk meg, hogy a módszer $O((1 + \lg q) \lg b)$ bitműveletet igényel.

b. Legyen $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Mutassuk meg, hogy a $\text{lnko}(a, b)$ kiszámításának problémáját a $\text{lnko}(b, a \bmod b)$ meghatározására visszavezetve az EUKLIDESZ algoritmus legfeljebb $c(\mu(a, b) - \mu(b, a \bmod b))$ bitműveletet igényel, ahol c egy megfelel ően nagy pozitív konstans.

c. Mutassuk meg, hogy az EUKLIDESZ(a, b) algoritmus általában $O(\mu(a, b))$, ha a bemen ő adatok β bit nagyságúak, akkor pedig $O(\beta^2)$ bitműveletet hajt végre.

31-3. Három algoritmus a Fibonacci-számok kiszámítására

Ez a feladat az F_n n-edik Fibonacci-szám kiszámítására szolgáló három módszer gyorsaságát hasonlítja össze adott n esetén. Tegyük fel el őször, hogy a számok nagyságától függetlenül két szám összeadása, kivonása és szorzása $O(1)$ költségű.

a. Mutassuk meg, hogy az F_n kiszámítására alkalmazott (3.21) egyszerű rekurziós módszer n-ben exponenciális futási időt igényel.

b. Mutassuk meg, hogy adattárolással az F_n érték $O(n)$ futási idő alatt is kiszámítható.

c. Mutassuk meg, hogy csak egészek összeadását és szorzását felhasználva hogyan lehet az F_n értékét meghatározni $O(\lg n)$ idő alatt. (*Útmutatás.* Dolgozzunk a

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

mátrixszal és hatványaival.)

d. Tegyük fel, hogy két β bit nagyságú szám összeadása $\Theta(\beta)$, szorzása pedig $\Theta(\beta^2)$ időt igényel. Mennyi lesz a fenti három algoritmus futási ideje, ha ezeket a sokkal reálisabb elemi aritmetikai műveleti költségeket vesszük alapul?

31-4. Kvadratikus maradékok

Legyen p páratlan prím. A $\mathbf{Z}_p^*$ halmaz a eleme ***kvadratikus maradék,*** ha az $x^2 \equiv a \pmod{p}$ az x ismeretlenben megoldható.

a. Mutassuk meg, hogy pontosan $(p-1)/2$ kvadratikus maradék létezik modulo p.

b. Legyen p prím és $a \in \mathbf{Z}_p^*$. Az $\left(\frac{a}{p}\right)$ ***Legendre-szimbólum*** definíciója a következő: Legyen 1, ha a kvadratikus maradék modulo p, és -1 egyébként. Bizonyítsuk be, hogy minden $a \in \mathbf{Z}_p^*$ számra

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Készítsünk gyors algoritmust annak eldöntésére, hogy egy adott a szám kvadratikus maradék-e modulo p. Elemezzük az algoritmus hatékonyságát.

c. Legyen p egy $4k + 3$ alakú prím, továbbá a legyen kvadratikus maradék $\mathbf{Z}_p^*$-ban. Bizonyítsuk be, hogy $a^{k+1} \bmod p$ az a négyzetgyöke modulo p. Mennyi időre van szükség egy a kvadratikus maradék négyzetgyökének megkereséséhez modulo p ?

d. Tervezzünk gyors valószínűségi algoritmust a kvadratikus nemmaradékok ($\mathbf{Z}_p^*$ azon elemei, amelyek nem kvadratikus maradékok) meghatározására prím modulus esetén. Átlagosan mennyi lesz az algoritmus végrehajtásának műveletigénye?

Megjegyzések a fejezethez

Niven és Zuckerman [231] könyve kiváló bevezetés az elemi számelméletbe. Knuth [183] könyve alaposan elmagyarázza a legnagyobb közös osztót kereső algoritmust éppúgy, mint más alapvető számelméleti algoritmusokat. Bach [28] és Riesel [258] munkái a számítógépes számelmélet legújabb eredményeit foglalják össze. Dixon [78] könyve a faktorizációt és a prímtesztelést tekinti át. A Pomerance [245] által szerkesztett konferencia-kötet számos kiváló összefoglaló cikket tartalmaz. Az egyik legújabb számítógépes számelméleti alapokat összefoglaló kivételes munka Bach és Shallit [29] nevéhez fűződik.

Knuth [183] az euklideszi algoritmus eredetét kutatja. Az euklideszi algoritmust Euklidész görög matematikus írta le i. e. 300 körül Elemek című munkájának 7. könyvében (1. és 2. állítás). Lehetséges, hogy Euklidész leírása Exodusz egy i. e. 375 körüli algoritmusán alapul. Euklidész algoritmusa a világ legrégebbi nemtriviális algoritmusa kitüntető címet is viselhetné. Csak az ókori egyiptomiak által is ismert szorzó algoritmus vetélkedhetne vele ezért a címért. Shallit [274] az euklideszi algoritmus analízisének történetét tekinti át.

Knuth a kínai maradéktétel (31.27. tétel) speciális esetét Sun-Ce kínai matematikusnak tulajdonítja, aki valamikor i. e. 200 és i. sz. 200 között élt (ennél pontosabbat nem tudunk). Ugyanez a speciális eset szerepel Nikomakhosz görög matematikusnál is i. sz. 100 körül. 1247-ben Csin Csiu-Sau általánosította a tételt. A kínai maradéktétel végleges formája és teljesen általános bizonyítása L. Eulertől származik, 1734-ből.

A könyvben bemutatott valószínűségi prímteszt algoritmus Miller [221] és Rabin [254] munkája. Konstans szorzó erejéig ez a leggyorsabb ismert valószínűségi prímteszt. A 31.39. tételre adott bizonyítás egy Bachtól [27] származó bizonyítás kissé módosított változata. A Miller–Rabin algoritmus erősebb eredményének bizonyítását Monier [224, 225] adta. A véletlenszerű választás szükségszerűnek tűnik ahhoz, hogy polinomiális idejű prímtesztelő algoritmust kapjunk. A leggyorsabb ismert determinisztikus prímteszt az Adleman, Pomerance és Rumely [3] prímteszt Cohen–Lenstra [65] változata. Ez a $\lceil \lg(n+1) \rceil$ nagyságú n számot $\lg n^{O(\lg \lg \lg n)}$ ideig teszteli, ami csak kevéssel haladja meg a polinomiális futási időt.

Nagy „véletlen” prímek keresésével foglalkozik Beauchemin, Brassard, Crépeau, Goutier és Pomerance [33] remek cikke is.

A nyilvános kulcsú titkosírás Diffie és Hellman [74] tollából született. Az RSA titkosírásra 1977-ben tett javaslatot Rivest, Shamir és Adleman [259]. Azóta a titkosírás tudománya – a kriptográfia – virágkorát éli. Az RSA titkosírás rejtelmeit egyre mélyebben ismerjük, az itt bemutatott alapvető technikák modern implementációi lényegi finomításokat tartalmaznak. Ezenkívül a titkosírások biztonságosabbá tételére számos új technikát fejlesztettek ki. Például Goldwasser és Micali [123] megmutatták, hogy nyilvános kulcsú kódolási rendszerek tervezésénél a véletlen elemek alkalmazása hatékony eszköz. Ami a digitális aláírást illeti, Goldwasser, Micali és Rivest [124] olyan digitális aláíró sémát találtak ki, amelynek mindenféle elképzelhető hamisítása bizonyíthatóan legalább olyan nehéz, mint a tényezőkre bontás. Menezes és társai [220] az alkalmazott kriptográfia szép áttekintését adták .

Az egész számok tényezőkre bontásához használt ρ-heurisztikát Pollard [242] fejlesztette ki. Az itt bemutatott verzió Brent [48] munkájának egy módosított változata.

Nagy számok prímfelbontására a legjobb ismert algoritmusok futási ideje is durván becsülve a faktorizálandó n szám hossza köbgyökével exponenciális nő. Buhler és társai [51] általános számtestekben megvalósított szita algoritmusával történő faktorizálás Pollard [243] és Lenstra [201] számtest-szitával tényezőkre bontó algoritmusának általánosítása, amit Coppersmith [69] és mások is finomítottak. Nagy bemenő adatokra talán ez a leghatékonyabb ilyen általános algoritmus. Bár az algoritmus szigorú elemzése meglehetősen nehéz, mégis ésszerű feltételek mellett a futási idő az $L(1/3, n)^{1.902+o(1)}$ függvénnyel becsülhető, ahol $L(\alpha, n) = e^{(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}$.

Lenstra [202] elliptikus görbéken alapuló módszere bizonyos bemenő adatokkal sokkal hatékonyabb, mint a számtest-szita, mivel a Pollard-ρ módszerhez hasonlóan elég gyorsan tud kis p prímtényezőt találni. A módszer egy p prímet a becslések szerint $L(1/2, p)^{\sqrt{2}+o(1)}$ idő alatt talál meg.

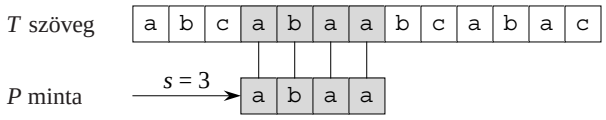
32. Mintaillesztés

Szövegszerkesztő programokban gyakori feladat megkeresni egy szövegben egy minta összes előfordulását. A szöveg rendszerint a szerkesztendő dokumentum, a keresendő minta pedig a felhasználó által megadott szó. A feladat megoldására szolgáló gyors algoritmusok jelentősen javíthatják a szövegszerkesztő programok hatékonyságát. Ezeket az ún. mintaillesztő algoritmusokat ennél jóval szélesebb körben használják, például, amikor egy bizonyos mintát keresnek DNS-láncokban.

A ***mintaillesztési probléma*** a következőképpen fogalmazható meg. Tegyük fel, hogy a szöveget egy n hosszúságú $T[1\,..\,n]$ tömb tartalmazza, a mintát pedig az m hosszúságú $P[1\,..\,m]$ tömbben tároljuk, és $m \leq n$. Mindkét tömb elemei a Σ véges ábécé jelei. Lehetséges ábécék, például: $\Sigma = \{0, 1\}$ vagy $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. A P, illetve T tömböt ***jelsorozatnak*** vagy röviden ***sorozatnak*** nevezzük.

Azt mondjuk, hogy a P minta ***előfordul s eltolással*** a T szövegben (vagy másképpen fogalmazva a P minta a T szöveg ***$(s + 1)$-edik pozíciójára illeszkedik***), ha $0 \leq s \leq n - m$ és $T[s{+}1\,..\,s{+}m] = P[1\,..\,m]$ (azaz: $\forall\, j \in [1\,..\,m] : T[s + j] = P[j]$). Ha P előfordul s eltolással T-ben, akkor s ***érvényes eltolás,*** ellenkező esetben s ***érvénytelen eltolás.*** A mintaillesztési probléma megoldásakor egy adott P minta összes érvényes eltolását kell megtalálnunk egy rögzített T szövegben. A 32.1. ábra szemlélteti a bevezetett fogalmakat.

Eltekintve az egyszerű algoritmustól, amelyet a 32.1. alfejezetben tekintünk át, minden ebben a fejezetben található mintaillesztő algoritmus első lépésben valamilyen módon feldolgozza a mintát, és ezután találja meg az érvényes eltolásokat. Az első lépést „előfeldolgozásnak”, a második részt „illesztésnek” nevezzük. A 32.2. ábra tartalmazza a tárgyalt algoritmusok előfeldolgozási és illesztési idejét. Az algoritmus teljes futási ideje ezen idők összege. A 32.2. alfejezetben bemutatunk egy érdekes, Rabin és Karp által kifejlesztett, mintaillesztő algoritmust. Noha az algoritmus futási ideje a legrosszabb esetben $\Theta((n - m + 1)m)$, ami megegyezik az egyszerű algoritmus futási idejével, az átlagos és a gyakorlatban előforduló esetekben sokkal hatékonyabb. Ezenkívül könnyen általánosítható egyéb mintaillesztési problémákra. A 32.3. alfejezetben egy véges automatát használó mintaillesztő algoritmust írunk le. Az automatát úgy tervezzük, hogy a P minta előfordulásait keresse a szövegben. Az algoritmus előfeldolgozási lépésének futási ideje $O(m|\Sigma|)$, azonban az illesztési idő csak $\Theta(n)$. Az ehhez hasonló, de sokkal ügyesebb Knuth–Morris–Pratt-algoritmust (vagy röviden KMP-algoritmust) mutatjuk be a 32.4. alfejezetben. A KMP-algoritmus illesztési ideje is $\Theta(n)$, ugyanakkor az előfeldolgozás ideje $\Theta(m)$-re csökken.



32.1. ábra. A mintaillesztési probléma. Célunk, hogy megtaláljuk a P = abaa minta összes előfordulását a T = abcabaabcabac szövegben. A minta csak egyszer, $s = 3$ eltolással fordul elő. Így $s = 3$ érvényes eltolás. A minta egyes jeleit függőleges vonallal kötöttük össze a szöveg illeszkedő jeleivel, és az összes illeszkedő jelet beszürkítettük.

Algoritmus	Előfeldolgozási idő	Illesztési idő
Egyszerű	0	$O((n-m+1)m)$
Rabin–Karp	$\Theta(m)$	$O((n-m+1)m)$
Véges automata	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth–Morris–Pratt	$\Theta(m)$	$\Theta(n)$

32.2. ábra. A fejezetben szereplő mintaillesztő algoritmusok előfeldolgozási és illesztési ideje.

Fogalmak és jelölések

A Σ ábécé jeleiből képzett összes véges hosszúságú sorozatok halmaza legyen Σ^*. Ebben a fejezetben csak véges hosszúságú sorozatokkal foglalkozunk. A nulla hosszú ***üres sorozat,*** jele ε, Σ^* eleme. Az x sorozat hossza: $|x|$. Az x és y sorozatok ***konkatenációja,*** jele xy, egy olyan sorozat, amelyben x jeleit y jelei követik, és a hossza $|x| + |y|$.

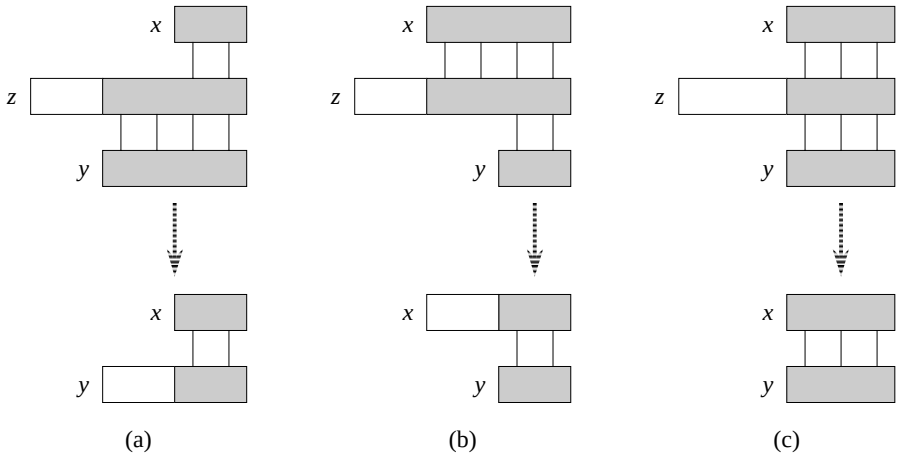
A w sorozat az x sorozat ***prefixe,*** jele $w \sqsubset x$, ha van olyan $y \in \Sigma^*$ sorozat, hogy $x = wy$. Vegyük észre, hogy ha $w \sqsubset x$, akkor $|w| \leq |x|$. Hasonlóan, a w sorozat az x sorozat ***szuffixe,*** jele $w \sqsupset x$, ha $x = yw$ fennáll valamely $y \in \Sigma^*$ sorozatra. Ha $w \sqsupset x$, akkor $|w| \leq |x|$. Az ε üres sorozat minden sorozatnak prefixe és szuffixe. Például ab $\sqsubset$ abcca és cca $\sqsupset$ abcca. Bármely x és y sorozat, valamint a jel esetén $x \sqsupset y$ akkor és csak akkor teljesül, ha $xa \sqsupset ya$ fennáll. Látható, hogy $\sqsubset$ és $\sqsupset$ tranzitív relációk. A következő lemma hasznos lesz a továbbiakban.

32.1. lemma (átfedő szuffixek). *Tegyük fel, hogy x, y és z olyan sorozatok, amelyekre $x \sqsupset z$ és $y \sqsupset z$ teljesül. Ha $|x| \leq |y|$, akkor $x \sqsupset y$. Ha $|x| \geq |y|$, akkor $y \sqsupset x$. Ha $|x| = |y|$, akkor $x = y$.*

Bizonyítás. A 32.3. ábráról leolvasható a bizonyítás. ■

A rövidség érdekében jelöljük P_k-val a $P[1\,..\,m]$ minta k hosszúságú $P[1\,..\,k]$ prefixét. Ennek megfelelően $P_0 = \varepsilon$ és $P_m = P = P[1\,..\,m]$. Hasonlóan T_k-val jelöljük a T szöveg k hosszúságú prefixét. A mintaillesztési problémát megfogalmazhatjuk ezekkel a jelölésekkel is: meg kell találnunk az összes olyan s eltolási értéket a $0 \leq s \leq n - m$ tartományban, amelyre $P \sqsupset T_{s+m}$ teljesül.

A továbbiakban az algoritmusok megadása során feltesszük, hogy azonos méretű sorozatok egyenlőségének vizsgálata megengedett, ún. primitív művelet. Ha a sorozatokat balról jobbra haladva hasonlítjuk össze, és megállunk, amikor két jel nem egyezik, akkor a vizsgálat idejét az egyező jelek számának lineáris függvényének tekintjük. Pontosabban, az $x = y$ feltétel vizsgálatának ideje $\Theta(t + 1)$, ahol t a leghosszabb olyan z sorozat hossza, amelyre: $z \sqsubset x$ és $z \sqsubset y$. ($\Theta(t + 1)$-et használunk $\Theta(t)$ helyett a $t = 0$ eset megfelelő kezelése miatt. Ekkor az első jelek nem egyeznek meg, de ennek ellenőrzése időt vesz igénybe.)



32.3. ábra. A 32.1. lemma grafikus bizonyítása. A feltétel szerint $x \sqsupset z$ és $y \sqsupset z$. Az ábra három része a lemma egyes eseteit szemlélteti. A sorozatok illeszkedő, szürke részeit függőleges vonalak kötik össze. **(a)** Ha $|x| \leq |y|$, akkor $x \sqsupset y$. **(b)** Ha $|x| \leq |y|$, akkor $x \sqsubset y$. **(c)** Ha $|x| = |y|$, akkor $x = y$.

32.1. Egy egyszerű mintaillesztő algoritmus

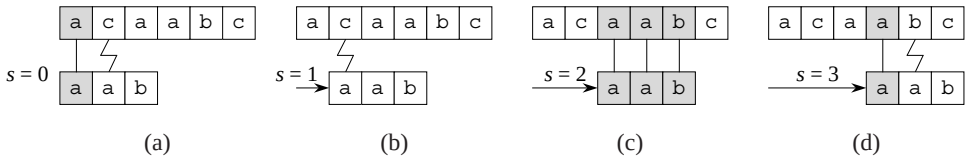
Az egyszerű algoritmus megtalálja az összes érvényes s eltolást úgy, hogy egy ciklusban vizsgálja a $P[1\,..\,m] = T[s+1\,..\,s+m]$ feltétel teljesülését az összes, $n-m+1$, lehetséges s értékre.

EGYSZERŰ-MINTAILLESZTŐ(T, P)

```
1  n ← hossz[T]
2  m ← hossz[P]
3  for s ← 0 to n − m
4      do if P[1 . . m] = T[s + 1 . . s + m]
5             then print „A minta illeszkedik az (”s + 1„)-edik pozícióra”
```

Az egyszerű mintaillesztő eljárást elképzelhetjük úgy is, mintha a mintát tartalmazó „sablont” tolnánk végig a szövegen, és közben minden egyes olyan eltolást feljegyzünk, amikor a sablonban található jelek megegyeznek a szöveg megfelelő jeleivel. Ezt szemlélteti a 32.4. ábra. Az algoritmus 3. sorában kezdődő **for** ciklus minden lehetséges eltolási értéket megvizsgál. A 4. sor feltétele eldönti, hogy az aktuális eltolás érvényes-e. Ez az ellenőrzés tartalmaz egy implicit ciklust, amely összeveti a megfelelő jeleket. A ciklus befejeződik, ha az összes jel egyezik, vagy ha különböző jeleket talál. Az 5. sorban minden egyes érvényes eltolási értéknek megfelelő pozíciót kiírunk.

Az EGYSZERŰ-MINTAILLESZTŐ eljárás futási ideje $O((n-m+1)m)$, és ez éles korlát a legrosszabb esetre. Például, legyen a szöveg $\mathtt{a}^n$ ($\mathtt{a}$ jelből álló n hosszúságú sorozat) és a minta $\mathtt{a}^m$. Ekkor a 4. sorban szereplő implicit ciklus s összes lehetséges értékére m összehasonlítást hajt végre, míg eldönti, hogy az eltolás érvényes. s $n-m+1$ értéket vehet fel, így a legrosszabb futási idő $\Theta((n-m+1)m)$, ami $\Theta(n^2)$, ha $m = \lfloor n/2 \rfloor$. Az EGYSZERŰ-MINTAILLESZTŐ futási ideje megegyezik az illesztési idővel, miután nem tartalmaz előfeldolgozást.



32.4. ábra. Az egyszerű mintaillesztő működése, ha a minta $P = \texttt{aab}$ és a szöveg $T = \texttt{acaabc}$. A P mintát elképzelhetjük egy „sablonnak”, amelyet mindig tovább csúsztatunk a szövegen. Az **(a)–(d)** részek mutatják a négy egymást követő elrendezést, amelyeket az egyszerű mintaillesztő megvizsgált. Minden részben függőleges egyenesek kötik össze az illeszkedő, szürke részeket, és törött vonalat húztunk az eltérő jelek közé, ha vannak ilyenek. A minta előfordul a szövegben $s = 2$ esetén. Ezt mutatja a (c) rész.

Látni fogjuk, hogy az EGYSZERŰ-MINTAILLESZTŐ nem optimális megoldás a kitűzött feladatra. A fejezetben később bemutatunk egy olyan algoritmust, amelynek a legrosszabb esetben az előfeldolgozási ideje $\Theta(m)$ és az illesztési ideje $\Theta(n)$. Az egyszerű mintaillesztés nem hatékony, mert új s értékek esetén figyelmen kívül hagyja azokat a szövegre vonatkozó ismereteket, amelyeket megelőző s értékek kipróbálása során már felderített. Ugyanakkor ez az információ nagyon értékes lehet. Például, ha $P = \texttt{aaab}$ és azt kapjuk, hogy $s = 0$ érvényes eltolás, akkor az 1, 2 vagy 3 mértékű eltolások egyike sem lehet érvényes, hiszen $T[4] = \texttt{b}$. A következőkben megvizsgáljuk, miként használhatjuk fel hatékonyan az ilyen jellegű információkat.

Gyakorlatok

32.1-1. Vizsgáljuk meg milyen összehasonlításokat végez az egyszerű mintaillesztő, ha a minta $P = \texttt{0001}$, a szöveg pedig $T = \texttt{000010001010001}$.

32.1-2. Tegyük fel, hogy a P minta összes jele különböző. Adjuk meg, miként lehet az EGYSZERŰ-MINTAILLESZTŐ algoritmust felgyorsítani úgy, hogy a futási ideje $O(n)$ legyen, ha a T szöveg hossza n.

32.1-3. Legyen $\Sigma_d = \{0, 1, \ldots, d-1\}$ egy d elemű ábécé ($d \geq 2$). Tegyük fel, hogy P és T m, illetve n hosszúságú, Σ_d elemeiből álló, véletlenszerűen választott sorozatok. Mutassuk meg, hogy az egyszerű algoritmus 4. sorában szereplő ciklusban végzett összehasonlítások száma a ciklus összes lefutását figyelembe véve várhatóan:

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1)\,.$$

(Az összehasonlítást a leírtaknak megfelelően hajtjuk végre, azaz befejezzük, ha különböző jelekhez vagy a minta végéhez érünk.) Ezt felhasználva azt mondhatjuk, hogy véletlen sorozatokra az egyszerű algoritmus elég hatékony.

32.1-4. Tegyük fel, hogy a P mintában előfordulhatnak ***lefedő jelek*** ($\diamond$), amelyek tetszőleges (akár nulla hosszú) jelsorozatra illeszkednek. Például az $\texttt{ab}\diamond\texttt{ba}\diamond\texttt{c}$ minta előfordul a $\texttt{cabccbacbacab}$ szövegben mint

$$\texttt{c}\underbrace{\texttt{ab}}_{\texttt{ab}}\underbrace{\texttt{cc}}_{\diamond}\underbrace{\texttt{ba}}_{\texttt{ba}}\underbrace{\texttt{cba}}_{\diamond}\underbrace{\texttt{c}}_{\texttt{c}}\texttt{ab}$$

és mint

$$\texttt{c}\underbrace{\texttt{ab}}_{\texttt{ab}}\underbrace{\texttt{ccbac}}_{\diamond}\underbrace{\texttt{ba}}_{\texttt{ba}}\underbrace{}_{\diamond}\underbrace{\texttt{c}}_{\texttt{c}}\texttt{ab}.$$

A lefedő jel tetszőlegesen sokszor előfordulhat a mintában, de nem szerepelhet a szöveg-

ben. Adjunk polinomiális idejű algoritmust annak eldöntésére, hogy ilyen típusú P minta előfordul-e egy adott T szövegben, és elemezzük az algoritmus futási idejét.

32.2. Rabin–Karp-algoritmus

Rabin és Karp közzétett egy mintaillesztő algoritmust, amely a gyakorlatban hatékonyan működik, továbbá hasonló feladatok megoldására is általánosítható. Például kétdimenziós mintaillesztésre is alkalmazható az átalakított algoritmus. A Rabin–Karp-algoritmus előfeldolgozási ideje $\Theta(m)$, és a legrosszabb esetben futási ideje $\Theta((n-m+1)m)$. Bizonyos feltételek esetén azonban a futási ideje átlagos esetben jobb.

Az algoritmus elemi számelméleti eszközöket használ, mint például a maradékosztályok, illetve a modulus képzés fogalma. A 31.1. alfejezetben megtalálhatóak a szükséges fogalmak, definíciók.

Az egyszerűség kedvéért tegyük fel, hogy $\Sigma = \{0, 1, 2, \ldots, 9\}$, azaz a jelek a decimális számjegyek. (Általában azt mondhatjuk, hogy a jelek egy d alapú számrendszer számjegyei, ahol $d = |\Sigma|$.) Ekkor egy k hosszúságú jelsorozatot tekinthetünk egy k jegyű decimális számnak. Például a `31415` jelsorozatnak megfelel a 31415 decimális szám. Miután megadtuk a jelek és a számjegyek közötti megfeleltetést, ebben a pontban számjegyként jelöljük a jeleket, azaz 1 helyett 1-et írunk.

Egy adott $P[1\,..\,m]$ minta decimális értékét jelöljük p-vel. Hasonlóan, egy $T[1\,..\,n]$ szöveg $T[s+1\,..\,s+m]$ m hosszúságú részsorozatának decimális értékét jelöljük t_s-sel, bármely $s = 0, 1, \ldots, n-m$ esetén. Ekkor $t_s = p$ akkor és csak akkor, ha $T[s+1\,..\,s+m] = P[1\,..\,m]$; azaz s érvényes eltolási érték akkor és csak akkor, ha $t_s = p$. Ha p értékét $\Theta(m)$ és az összes t_i értékét együttesen $\Theta(n-m+1)$ idő alatt ki tudjuk számolni,[1] akkor az összes s érvényes eltolás meghatározható $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ idő alatt, hiszen p értékét kell összehasonlítani minden t_s értékkel. (Egy pillanatra hagyjuk figyelmen kívül, hogy p és t_s értéke nagyon nagy is lehet.)

p értéke kiszámítható $\Theta(m)$ időben a Horner-séma segítségével (lásd 30.1. alfejezet):

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots)).$$

t_0 értéke hasonlóan meghatározható $T[1\,..\,m]$ segítségével $\Theta(m)$ idő alatt.

Könnyen látható, hogy a további $t_1, t_2, \ldots, t_{n-m}$ értékek $\Theta(n-m)$ idő alatt kiszámíthatóak, ha észrevesszük, hogy t_{s+1} konstans időben megkapható t_s-ből, mivel:

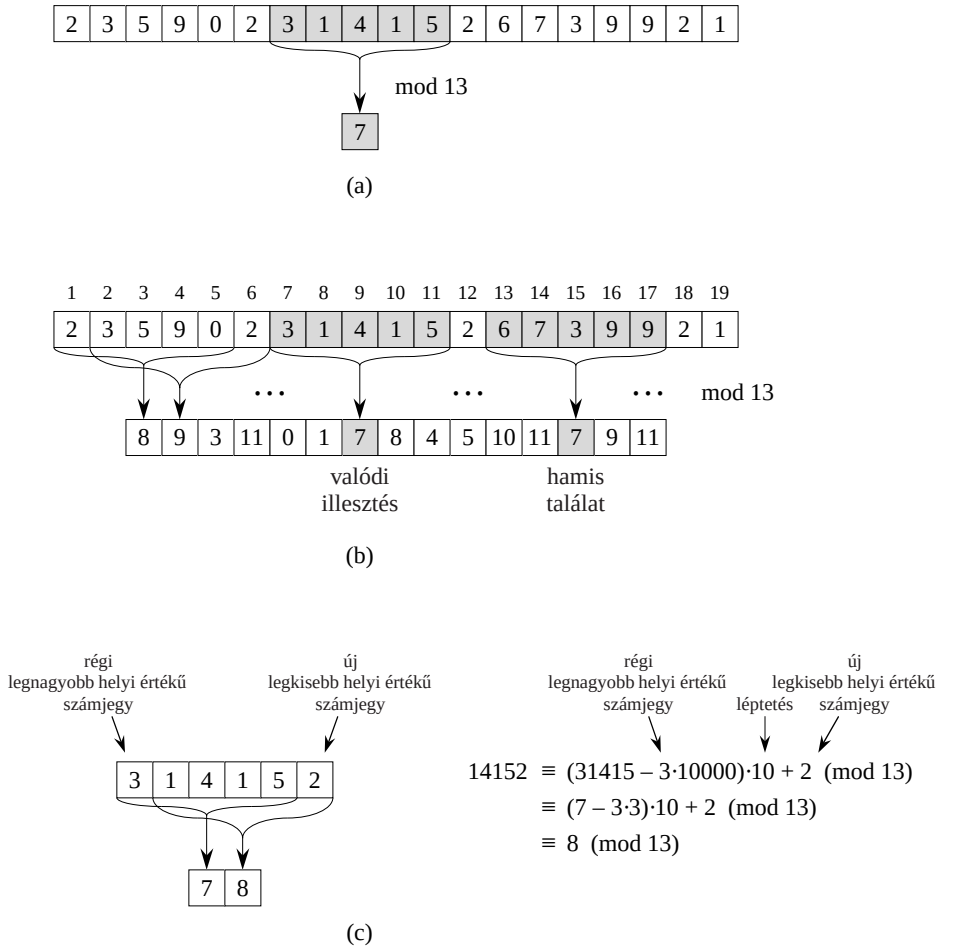
$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]. \tag{32.1}$$

Például, legyen $m = 5$ és $t_s = 31415$. Ha ki szeretnénk léptetni a legmagasabb helyi értékű $T[s+1] = 3$ számjegyet, és beléptetni egy új (pl. $T[s+5+1] = 2$) számjegyet a legalacsonyabb helyi értékre, akkor

$$t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152.$$

$10^{m-1}T[s+1]$ kivonása eltávolítja a legnagyobb helyi értékű számjegyet; az eredményt tízzel szorozva a számot eggyel balra léptetjük; és ehhez hozzáadva $T[s+m+1]$ értékét,

[1] $\Theta(n-m+1)$-et használunk $\Theta(n-m)$ helyett, mert s-nek $n-m+1$ különböző értéke lehet. A „+1” aszimptotikus értelemben fontos, hiszen ha $m = n$, akkor t_s értékének kiszámítása nem $\Theta(0)$, hanem $\Theta(1)$ ideig tart.



32.5. ábra. A Rabin–Karp-algoritmus. Minden jel egy decimális számjegy, és az értékeket modulo 13 számoljuk. **(a)** A szövegben egy 5 hosszúságú ablak tartalmát szürkére festettük. A befestett rész osztási maradéka 7 modulo 13. **(b)** Minden lehetséges 5 hosszúságú ablakra kiszámoltuk az ablakban szereplő szám értékét modulo 13. Ha a minta $P = 31415$, akkor olyan részeket keresünk, amelyeknek az értéke 7 modulo 13, hiszen $31415 \equiv 7 \pmod{13}$. Két ilyen ablak létezik, ezeket szürkére festettük. Az első, amelyik a 7. pozíción kezdődik, a minta egy tényleges előfordulása, ezzel szemben a 13. pozíción kezdődő második ablak egy hamis találat. **(c)** Egy ablakhoz tartozó érték konstans idejű kiszámítása a megelőző ablak értékének felhasználásával. Az első ablakban szereplő érték 31415. Elhagyva a rész legmagasabb helyi értékű számjegyét (3), balra léptetve (tízzel szorozva), és végül hozzáadva a következő rész legalacsonyabb helyi értékű jegyét (2), megkapjuk az új rész értékét, ami 14152. Minden számítást modulo 13 végzünk, ezért az első ablaknak megfelelő érték 7, a másodikhoz rendelt pedig 8.

a legalacsonyabb helyi értékre a megfelelő számjegy kerül. Ha a 10^{m-1} konstans értéket előre kiszámítjuk, akkor a (32.1) egyenlet alapján a következő t_i mindig konstans számú aritmetikai művelet végrehajtásával meghatározható. (10^{m-1} kiszámítható $O(\lg m)$ idő alatt a 31.6. alfejezetben bemutatott módszerrel, de ebben az esetben a kézenfekvő $O(m)$ idejű módszer is megfelelő.) Így p értéke $\Theta(m)$, és $t_0, t_1, \ldots, t_{n-m}$ értékek $\Theta(n-m+1)$ idő alatt kiszámíthatóak, és a $P[1\,..\,m]$ minta összes előfordulását a $T[1\,..\,n]$ szövegben megtalálhatjuk $\Theta(m)$ előfeldolgozási és $\Theta(n-m+1)$ illesztési idő alatt.

Az egyetlen probléma ezzel a módszerrel, hogy p és t_s értékek olyan nagyok lehetnek, amelyeket már nem lehet számítógépen a szokásos módon ábrázolni. (A számok m számjegyet tartalmaznak.) Ebben az esetben azonban nem tehetjük fel, hogy az aritmetikai műveletek „konstans időben” végrehajthatóak. Szerencsére, erre a problémára könnyű megoldást találni, amelyet a 32.5. ábrán szemléltetünk. Számítsuk ki p és t_s értékét modulo q, ahol q egy alkalmas modulus. Miután p, t_0 és a (32.1) egyenlettel megadott rekurzív képlet is számítható modulo q, ezért p modulo q meghatározható $\Theta(m)$ idő alatt, és az összes t_s érték q-val képzett osztási maradéka kiszámítható $\Theta(n - m + 1)$ idő alatt. A q értéket rendszerint olyan prímszámnak választják, hogy $10q$ még éppen ábrázolható legyen el őjel nélküli egész számként. Így az összes szükséges számítást hatékonyan el tudjuk végezni a szokásos aritmetikai műveletek segítségével. Általában, amikor az ábécé jeleinek száma d, és ezeket a $\{0, 1, \ldots, d-1\}$ értékekkel azonosítjuk, akkor q értékét úgy választjuk meg, hogy dq ábrázolható legyen a számítógépen el őjel nélküli egész számként. A (32.1) egyenletben szereplő összefüggést módosítjuk, és t_{s+1} értékét modulo q határozzuk meg, azaz:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q, \tag{32.2}$$

ahol $h \equiv d^{m-1} \pmod q$ egy m szélességű ablak legmagasabb helyi értékén szerepl ő „1”-nek megfelelő érték.

Az ábrázolhatósági korlátokat ugyan sikerült feloldanunk azzal, hogy modulo q számolunk, azonban ennek kellemetlen következményeként $t_s \equiv p \pmod q$ fennállásából nem következik $t_s = p$ teljesülése. Ugyanakkor, ha $t_s \not\equiv p \pmod q$, akkor $t_s \neq p$, azaz s érvénytelen eltolás. Így a $t_s \equiv p \pmod q$ feltételt gyors heurisztikaként alkalmazhatjuk érvénytelen eltolási értékek kiszűrésére. Minden olyan s eltolási érték, amelyre $t_s \equiv p \pmod q$ fennáll, további ellenőrzésre szorul, hogy eldönthessük, s valóban érvényes vagy csak egy ***hamis találatot*** határoz meg. Ez az ellenőrzés a $P[1\,..\,m] = T[s+1\,..\,s+m]$ feltétel explicit vizsgálatát jelenti. Ha q kellően nagy szám, akkor várhatóan ritkán lépnek fel hamis találatok, így ezek ellenőrzésének költsége alacsony.

A következő eljárásban pontosítjuk az eddig leírtakat. Az eljárás bemenete a T szöveg, a P minta, a számrendszer alapja d (ez rendszerint $|\Sigma|$) és a q prímszám.

RABIN–KARP-ILLESZTŐ(T, P, d, q)

```
 1  n ← hossz[T]
 2  m ← hossz[P]
 3  h ← d^(m−1) mod q
 4  p ← 0
 5  t_0 ← 0
 6  for i ← 1 to m                          ▷ Előfeldolgozás.
 7      do p ← (dp + P[i]) mod q
 8         t_0 ← (dt_0 + T[i]) mod q
 9  for s ← 0 to n − m                      ▷ Illesztés.
10      do if p = t_s
11            then if P[1..m] = T[s + 1..s + m]
12                    then print „A minta illeszkedik az (”s + 1„)-edik pozícióra”
13         if s < n − m
14            then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

A Rabin–Karp-illesztő működését írjuk le az alábbiakban. Minden jelhez egy d alapú számrendszerbeli számjegyet rendelünk. A t értékeket csak a követhetőség érdekében láttuk el indexekkel, a programban az indexek elhagyhatóak. A 3. sorban h értékét számítjuk ki a leírtaknak megfelelően. A 4–8. sorokban $P[1\,..\,m] \bmod q$ és $T[1\,..\,m] \bmod q$ értékeket állítjuk elő a p és t_0 változókban. A 9–14. sorokat alkotó **for** ciklus sorra veszi a lehetséges s értékeket, és teljesíti a következő invariáns tulajdonságot:

a 10. sor végrehajtásakor $t_s = T[s + 1\,..\,s + m] \bmod q$.

Ha $p = t_s$, akkor a 11. sorban ellenőrizzük, hogy $P[1\,..\,m] = T[s + 1\,..\,s + m]$ igaz-e, így kiszűrjük az esetleges hamis találatokat. Az érvényes eltolási értékeket kiírjuk a 12. sorban. Ha $s < n-m$ (amit a 13. sorban vizsgálunk), akkor a ciklust legalább még egyszer végrehajtjuk, így a 14. sorban biztosítani kell a ciklusinvariáns fennállását. A 14. sorban konstans idő alatt számoljuk ki $t_{s+1} \bmod q$ értékét $t_s \bmod q$ felhasználásával a (32.2) egyenlőség alapján.

A Rabin–Karp-illesztő előfeldolgozási ideje $\Theta(m)$ és illesztési ideje a legrosszabb esetben $\Theta((n - m + 1)m)$, hiszen (az egyszerű mintaillesztőhöz hasonlóan) a Rabin–Karp-algoritmus explicit ellenőriz minden érvényes eltolási értéket. Ha $P = \mathtt{a}^m$ és $T = \mathtt{a}^n$, akkor az ellenőrzésekhez $\Theta((n - m + 1)m)$ idő szükséges, miután a lehetséges $n - m + 1$ eltolás mindegyike érvényes.

Az alkalmazások többségében az érvényes eltolások száma alacsony (a számuk lehet egy c konstans). Ezekben az esetekben az algoritmus várható illesztési ideje csak $O((n - m + 1) + cm) = O(n + m)$, valamint a hamis találatok feldolgozásához szükséges idő. A következő heurisztikus elemzés alapja az a feltételezés, mely szerint az értékek redukálása modulo q, felfogható mint egy $\Sigma^* \to \mathbf{Z}_q$ leképezés. (Lásd az indexfüggvényekhez használt osztályozás elemzését a 11.3.1. pontban. Nehéz formálisan leírni és bizonyítani egy ilyen jellegű feltételezést, noha egy járható út annak feltételezése, hogy q egy véletlenszerűen választott megfelelő nagyságú egész szám. Itt eltekintünk ettől a formális tárgyalástól.) Ebben az esetben a hamis találatok száma várhatóan $O(n/q)$, hiszen $1/q$-val becsülhetjük annak az esélyét, hogy valamely s-re $t_s \equiv p \bmod q$. Miután $O(n)$ helyen ad rossz eredményt a 10. sorban szereplő vizsgálat, és $O(m)$ idő szükséges minden egyes találat esetén, a Rabin–Karp-algoritmus várható futási ideje

$$O(n) + O\left(m\left(v + \frac{n}{q}\right)\right),$$

ahol v az érvényes eltolások száma. Ez a futási idő $O(n)$, ha $v = O(1)$ és $q \geq m$. Ezek szerint, ha az érvényes eltolások száma várhatóan alacsony ($O(1)$), és q-nak a minta hosszánál nagyobb prímet választunk, akkor a Rabin–Karp-algoritmus várható illesztési ideje $O(n + m)$. Ugyanakkor $m \leq n$, ezért ez a várható illesztési idő $O(n)$.

Gyakorlatok

32.2-1. Legyen a szöveg $T = 3141592653589793$, a minta $P = 26$, a számításokhoz használt prímszám pedig $q = 11$. Mennyi a hamis találatok száma a Rabin–Karp-algoritmus végrehajtása során?

32.2-2. Miként lehetne a Rabin–Karp-algoritmust úgy módosítani, hogy a szövegben egy adott k elemű mintahalmaz tetszőleges elemének előfordulását keressük? Kiindulásként tegyük fel, hogy a k mindegyike ugyanolyan hosszú. Ezután általánosítsuk a megoldást úgy, hogy eltérő hosszúságú minták is megengedettek legyenek.

32.2-3. Fejlesszük tovább a Rabin–Karp-algoritmust annak a feladatnak a megoldására, amelyben egy $m \times m$-es mintát keresünk egy $n \times n$-es szövegben. (A minta függőleges és vízszintes irányban mozgatható, de nem forgatható.)

32.2-4. Antalnak rendelkezésére áll egy hosszú, n bitből álló fájl másolata, $A = \{a_{n-1}, a_{n-2}, \ldots, a_0\}$. Hasonlóan, Bélának is van egy n bites $B = \{b_{n-1}, b_{n-2}, \ldots, b_0\}$ fájlja. Szeretnék eldönteni, hogy a két fájl megegyezik-e. Elkerülendő teljes fájlok mozgatását, a következő gyors valószínűségi ellenőrzést használják. Rögzítenek egy $q > 1000n$ prímszámot és véletlenszerűen választanak egy x értéket a $\{0, 1, \ldots, q-1\}$ számok közül. Ezután Antal kiszámítja az

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i\right) \bmod q$$

értéket, és Béla ehhez hasonlóan $B(x)$-et. Bizonyítsuk be, hogy ha $A \neq B$, akkor legfeljebb egy az ezerhez annak az esélye, hogy $A(x) = B(x)$; ellenben ha a két fájl megegyezik, akkor $A(x) = B(x)$. (*Útmutatás.* Lásd 31.4-4. gyakorlatot.)

32.3. Mintaillesztés véges automatákkal

Sok mintaillesztő algoritmus olyan véges automatát állít elő, amely a P minta összes előfordulását keresi a T szövegben. Ebben a pontban módszert adunk arra, hogy miként lehet ilyen automatát létrehozni. Ezek a mintaillesztő automaták igen hatékonyak: a szöveg minden jelét pontosan egyszer vizsgálják meg, és egy jelet konstans idő alatt dolgoznak fel. Ezért az illesztési idő – miután az automatát létrehoztuk a minta előfeldolgozásával – $\Theta(n)$. Az automata felépítéséhez szükséges idő azonban nagy lehet, ha Σ elemszáma nagy. A 32.4. alfejezetben megmutatjuk, miként lehet ezt a problémát megkerülni.

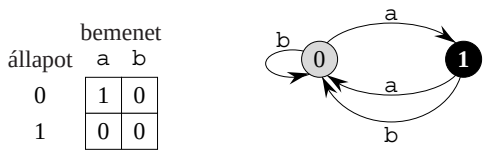
Először a véges automata fogalmát definiáljuk, majd megvizsgálunk egy speciális mintaillesztő automatát, és megmutatjuk hogyan használható az a mintaillesztési feladat megoldására. Részletesen bemutatjuk miként szimulálható egy ilyen mintaillesztő automata viselkedése adott szöveg esetén. Végül leírjuk, hogy adott minta esetén milyen módon hozható létre a megfelelő mintaillesztő automata.

Véges automaták

Egy M ***véges automata*** egy rendezett ötös $(Q, q_0, A, \Sigma, \delta)$, ahol

- Q véges halmaz, az ***állapotok*** halmaza,
- $q_0 \in Q$ a ***kezdőállapot,***
- $A \subseteq Q$ a ***végállapotok*** (vagy ***elfogadó állapotok***) halmaza,
- Σ véges halmaz, a ***bemeneti jelek halmaza*** (vagy ***bemeneti ábécé***),
- $\delta : Q \times \Sigma \to Q$ az automata ***átmeneti függvénye.***

Az automata működésének kezdetekor a q_0 állapotban van, és egyesével olvassa a bemeneti sorozatról a jeleket. Ha a q állapotban van és az a jelet olvassa be, akkor állapotot vált, „átmegy” a q állapotból a $\delta(q, a)$ állapotba. Ha az automata aktuális állapota q, eleme A-nak, akkor azt mondjuk, hogy az M automata ***elfogadja*** az eddig beolvasott sorozatot.



állapot	bemenet a	b
0	1	0
1	0	0

(a)

(b)

32.6. ábra. Egy egyszerű, kétállapotú véges automata. Az állapotok halmaza $Q = \{0, 1\}$, a kezdőállapot $q_0 = 0$, és a bemeneti jelek halmaza $\Sigma = \{a, b\}$. **(a)** A δ átmeneti függvény reprezentációja táblázattal. **(b)** Ekvivalens állapot-átmenet diagram. A befeketített 1 az elfogadó állapot. Az irányított élek reprezentálják az átmeneteket. Például, az 1-ből 0-ba vezető b címkéjű él $\delta(1, \mathtt{b}) = 0$-nak felel meg. Ez az automata azokat a sorozatokat fogadja el, amelyek végén páratlan számú a szerepel. Pontosabban, az x sorozatot az automata akkor és csak akkor fogadja el, ha $x = yz$, ahol $y = \varepsilon$ vagy y utolsó jele b, továbbá $z = a^k$ és k páratlan. abaaa bemenet esetén az automata sorban a $\langle 0, 1, 0, 1, 0, 1\rangle$ állapotokba kerül, és elfogadja ezt a bemenetet. abbaa esetén az állapotsorozat $\langle 0, 1, 0, 0, 1, 0\rangle$, ezért ezt elveti.

Ha egy sorozatot nem fogad el M, akkor azt mondjuk, hogy azt ***elveti.*** A 32.6. ábra egy egyszerű automatán szemlélteti a bevezetett fogalmakat.

Az M véges automata meghatároz egy ϕ ***végállapot függvényt.*** $\phi : \Sigma^* \to Q$, és egy w sorozat esetén $\phi(w)$ az az állapot, amelybe M kerül w elolvasása után. Ezért M elfogad egy w sorozatot akkor és csak akkor, ha $\phi(w) \in A$. A ϕ függvényt az alábbi rekurzív összefüggés definiálja:

$$\begin{aligned} \phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \qquad (w \in \Sigma^*, a \in \Sigma). \end{aligned}$$

Mintaillesztő automaták

Minden P mintához létezik mintaillesztő automata, amelyet létre kell hoznunk egy előkészítő lépésben, mielőtt a mintaillesztésben használhatnánk. A 32.7. ábra szemlélteti az előállítást a $P =$ ababaca minta esetén. A továbbiakban feltesszük, hogy P rögzített, és a rövidség érdekében a függőségek jelölésekor P-t nem tüntetjük fel.

Először egy σ segédfüggvényt definiálunk, amelyet majd a $P[1\,..\,m]$ mintához tartozó mintaillesztő automata meghatározásakor használunk fel. σ a P mintához rendelt ***szuffix függvény.*** $\sigma : \Sigma^* \to \{0, 1, \ldots, m\}$, és $\sigma(x)$ a leghosszabb olyan P-beli prefix hossza, amely szuffixe x-nek, azaz:

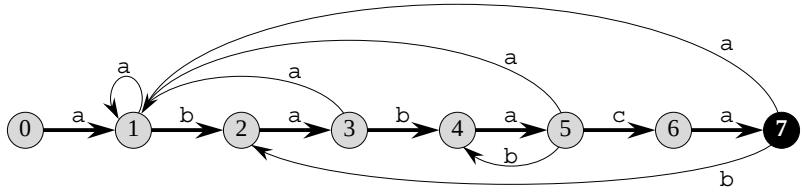
$$\sigma(x) = \max\{k : P_k \sqsupset x\}.$$

A σ szuffix függvény mindig értelmezett, hiszen az üres sorozat, $P_0 = \varepsilon$, minden sorozatnak szuffixe. Ha például $P =$ ab, akkor $\sigma(\varepsilon) = 0$, $\sigma($ccaca$) = 1$ és $\sigma($ccab$) = 2$. Egy m hosszúságú P minta esetén $\sigma(x) = m$ akkor és csak akkor, ha $P \sqsupset x$. A szuffix függvény definíciója miatt, ha $x \sqsupset y$, akkor $\sigma(x) \le \sigma(y)$.

A $P[1\,..\,m]$ mintának megfelelő mintaillesztő automatát a következőképpen definiáljuk.

- Az állapotok halmaza, Q, legyen $\{0, 1, \ldots, m\}$. A q_0 kezdőállapot legyen 0, és az egyetlen végállapot m.
- Tetszőleges q állapotra és a jelre, a δ átmeneti függvényt az alábbi egyenlőség adja meg:

$$\delta(q, a) = \sigma(P_q a). \tag{32.3}$$



(a)

állapot	bemenet a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
állapot $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

(c)

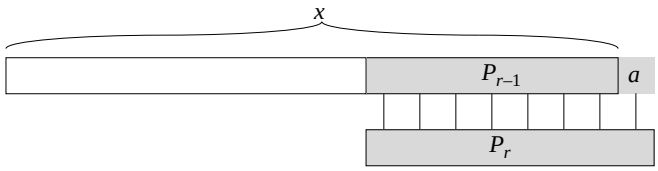
32.7. ábra. (a) Az ababaca sorozatra végződő sorozatokat elfogadó mintaillesztő automata állapot-átmenet diagramja. 0 a kezdőállapot, a befeketített 7 az egyetlen végállapot. Az i állapotból j-be vezető a-val címkézett irányított él $\delta(i, a) = j$-t reprezentálja. A jobbra haladó, megvastagított élek a minta és a szöveg jelei közötti sikeres, míg a balra mutató élek a sikertelen illesztéseknek felelnek meg. Az ábrán nem tüntettük fel az összes sikertelen illesztéshez rendelt élt. Megállapodás szerint, ha egy i állapotból nem vezet ki a címkéjű él valamely $a \in \Sigma$ esetén, akkor $\delta(i, a) = 0$. **(b)** A P = ababaca minta, és a megfelelő δ átmeneti függvény. A sikeres illesztéseknek megfelelő bejegyzéseket beszürkítettük. **(c)** Az automata működése T = abababacaba szöveg esetén. A szöveg összes $T[i]$ jele alatt feltüntettük az automata $\phi(T_i)$ állapotát, amelybe a T_i prefix feldolgozása után kerül. A minta egyszer fordul elő a szövegben, és ez az illeszkedés a 9. pozíción végződik.

A $\delta(q, a) = \sigma(P_q a)$ választás oka, hogy az automata működése során fennáll a

$$\phi(T_i) = \sigma(T_i) \tag{32.4}$$

invariáns, amit a későbbiekben a 32.4. tételben bizonyítunk be. Ez azt jelenti, hogy a T szöveg első i jelének elolvasása után az automata a $\phi(T_i) = q$ állapotba kerül, ahol $q = \sigma(T_i)$ a leghosszabb olyan T_i-beli szuffix hossza, amely a P mintának is prefixe. Ha a következő olvasandó jel $T[i+1] = a$, akkor az automatának a $\sigma(T_{i+1}) = \sigma(T_i a)$ állapotba kell kerülnie. A tétel bizonyítása során látni fogjuk, hogy $\sigma(T_i a) = \sigma(P_q a)$. Így a leghosszabb olyan $T_i a$-beli szuffix hosszát, amely egyben P prefixe is, kiszámíthatjuk úgy, hogy meghatározzuk a leghosszabb olyan $P_q a$-beli szuffixet, amely prefixe P-nek. Elegendő, ha az automata az egyes állapotokban csak azt ismeri, hogy mi a leghosszabb olyan P-beli prefix hossza, amely az eddig olvasott sorozatnak szuffixe. Ezért a $\delta(q, a) = \sigma(P_q a)$ választás fenntartja a kívánt (32.4) invariánst. Ezt az okfejtést a későbbiekben formálisan is bebizonyítjuk.

Például a 32.7. ábrán látható mintaillesztő automata esetén $\delta(5, \mathtt{b}) = 4$. Ennek indoka, hogy ha az automata egy b jelet olvas a $q = 5$ állapotban, akkor P_qb = ababab, és P leghosszabb olyan prefixe, amely egyben szuffixe az ababab sorozatnak is P_4 = abab.



32.8. ábra. A 32.2. lemma bizonyításának szemléltetése. Látható, hogy $r \leq \sigma(x) + 1$, ha $r = \sigma(xa)$.

A mintaillesztő automaták működésének megértését segíti a következő egyszerű és hatékony program, amely azt írja le, hogy ilyen (δ átmeneti függvénnyel reprezentált) automaták miként viselkednek a mintaillesztési feladat megoldása során. Amint azt már leírtuk, minden m hosszúságú mintához rendelt mintaillesztő automata állapotainak halmaza $Q = \{0, 1, \ldots, m\}$, a kezdőállapot 0, az egyedüli végállapot pedig m.

VÉGES-AUTOMATA-ILLESZTŐ(T, δ, m)

```
1  n ← hossz[T]
2  q ← 0
3  for i ← 1 to n
4      do q ← δ(q, T[i])
5         if q = m
6            then print „A minta illeszkedik az (”i − m + 1„)-edik pozícióra”
```

A VÉGES-AUTOMATA-ILLESZTŐ egy egyszerű ciklusból áll, amiből következik, hogy n hosszúságú szöveg esetén az illesztési ideje $\Theta(n)$. Ez az illesztési idő nem tartalmazza a δ átmeneti függvény meghatározásához szükséges előfeldolgozási időt. Ezzel később foglalkozunk, miután beláttuk, hogy a VÉGES-AUTOMATA-ILLESZTŐ helyesen működik.

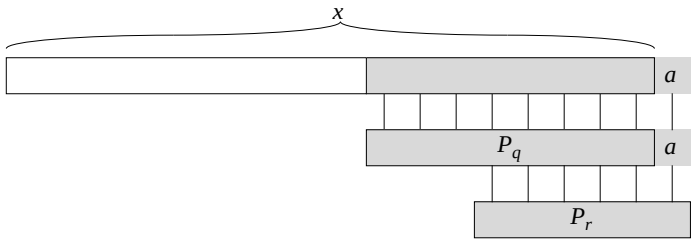
Az automata működését a $T[1\,..\,n]$ bemeneti szövegre vizsgáljuk. Bebizonyítjuk, hogy a $T[i]$ jel elolvasása után az automata a $\sigma(T_i)$ állapotba kerül. Mivel $\sigma(T_i) = m$ akkor és csak akkor, ha $P \sqsupset T_i$, ezért akkor és csak akkor jutottunk az m végállapotba, ha éppen P-t olvastuk be. A bizonyításhoz a következő két, a σ szuffix függvényre vonatkozó lemmát használjuk fel.

32.2. lemma (szuffix függvény egyenlőtlenség). *Bármely x sorozat és a jel esetén $\sigma(xa) \leq \sigma(x) + 1$.*

Bizonyítás. A 32.8. ábrának megfelelően, legyen $r = \sigma(xa)$. Ha $r = 0$, akkor az $\sigma(xa) = r \leq \sigma(x) + 1$ állítás nyilvánvalóan igaz, hiszen $\sigma(x)$ nem lehet negatív. Tegyük fel, hogy $r > 0$. Ekkor, σ definíciója miatt, $P_r \sqsupset xa$. Elhagyva P_r és xa sorozatok végéről az a jelet azt kapjuk, hogy $P_{r-1} \sqsupset x$. Így $r - 1 \leq \sigma(x)$, mivel $\sigma(x)$ a legnagyobb olyan k érték, amire $P_k \sqsupset x$ fennáll, így $\sigma(xa) = r \leq \sigma(x) + 1$. ■

32.3. lemma (szuffix függvény rekurzió). *Bármely x sorozat és a jel esetén, ha $q = \sigma(x)$, akkor $\sigma(xa) = \sigma(P_q a)$.*

Bizonyítás. σ definíciója miatt $P_q \sqsupset x$. A 32.9. ábra alapján $P_q a \sqsupset xa$. Ha $r = \sigma(xa)$, akkor a 32.2. lemma szerint $r \leq q + 1$. Tudjuk, hogy $P_q a \sqsupset xa$, $P_r \sqsupset xa$ és $|P_r| \leq |P_q a|$,



32.9. ábra. A 32.3. lemma bizonyításának szemléltetése. Látható, hogy $r = \sigma(P_q a)$, ha $q = \sigma(x)$ és $r = \sigma(xa)$.

ezért a 32.1. lemma miatt $P_r \sqsupset P_q a$. Így $r \leq \sigma(P_q a)$, azaz $\sigma(xa) \leq \sigma(P_q a)$. Azonban $\sigma(P_q a) \leq \sigma(xa)$ szintén teljesül, mivel $P_q a \sqsupset xa$. Ezért $\sigma(xa) = \sigma(P_q a)$. ■

Most már minden rendelkezésünkre áll ahhoz, hogy bebizonyítsuk azt a tételt, amely jellemzi a mintaillesztő automata működését egy adott szöveg esetén. Amint azt már megjegyeztük, a tételből kiderül, hogy az automata minden lépésben csak a minta leghosszabb olyan prefixét tartja nyilván, amely szuffixe a beolvasott sorozatnak. Másként fogalmazva: az automata fenntartja a (32.4) invariánst.

32.4. tétel. *Ha ϕ egy P mintához tartozó mintaillesztő automata végállapot függvénye és $T[1\,..\,n]$ az automata bemenete, akkor*

$$\phi(T_i) = \sigma(T_i) \qquad (i = 0, 1, \ldots, n).$$

Bizonyítás. i szerinti indukcióval látjuk be az állítást. Ha $i = 0$, a tétel nyilvánvalóan igaz, mivel $T_0 = \varepsilon$, és így $\phi(T_0) = 0 = \sigma(T_0)$.

Feltesszük, hogy $\phi(T_i) = \sigma(T_i)$, és belátjuk, hogy $\phi(T_{i+1}) = \sigma(T_{i+1})$. Legyen $q = \phi(T_i)$ és $a = T[i + 1]$. Ekkor

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && (T_{i+1} \text{ és } a \text{ definíciója miatt}) \\
&= \delta(\phi(T_i), a) && (\phi \text{ definíciója miatt}) \\
&= \delta(q, a) && (q \text{ definíciója miatt}) \\
&= \sigma(P_q a) && (\delta \text{ (32.3)-beli definíciója miatt}) \\
&= \sigma(T_i a) && (\text{az indukciós feltétel és a 32.3. lemma miatt}) \\
&= \sigma(T_{i+1}) && (T_{i+1} \text{definíciója miatt}).
\end{aligned}$$

■

A 32.4. tétel alapján, ha az automata az algoritmus 4. sorában a q állapotba kerül, akkor q a legnagyobb olyan érték, amelyre $P_q \sqsupset T_i$. Ezért az 5. sorban $q = m$ akkor és csak akkor állhat fenn, ha éppen a P minta egy előfordulását olvastuk be. Ennek alapján azt mondhatjuk, hogy a VÉGES-AUTOMATA-ILLESZTŐ helyesen működik.

Az átmeneti függvény meghatározása

A következő eljárás kiszámítja egy $P[1\,..\,m]$ mintához tartozó δ átmeneti függvény értékeit.

ÁTMENETI-FÜGGVÉNY-SZÁMÍTÁS(P,Σ)

```
1  m ← hossz[P]
2  for q ← 0 to m
3      do for minden a ∈ Σ jelre
4             do k ← min(m + 1, q + 2)
5                repeat k ← k − 1
6                 until  P_k ⊐ P_q a
7                δ(q, a) ← k
8  return δ
```

Ez az eljárás a definíció alapján határozza meg a $\delta(q, a)$ értékeket. A 2. és 3. sor ciklusai az összes lehetséges q és a értéket megvizsgálják, a 4–7. sorok pedig beállítják $\delta(q, a)$ értékét a legnagyobb olyan k-ra, amelyre $P_k \sqsupset P_q a$. k-nak kezdetben a szóba jöhető legnagyobb értéket választjuk, ez $\min(m, q + 1)$, és ezt csökkentjük, amíg $P_k \sqsupset P_q a$ be nem következik.

Az ÁTMENETI-FÜGGVÉNY-SZÁMÍTÁS futási ideje $O(m^3|\Sigma|)$, mert a külső ciklusok $m|\Sigma|$ értéket vesznek figyelembe, a belső **repeat** ciklus legfeljebb $m + 1$ iterációt hajthat végre, és a 6. sorban szereplő $P_k \sqsupset P_q a$ feltétel eldöntéséhez m jel összehasonlítására lehet szükség. Ismertek lényegesen gyorsabb eljárások; δ meghatározható $O(m|\Sigma|)$ idő alatt is, ha kihasználunk bizonyos, a P mintára kiszámolható információkat (lásd 32.4-6. gyakorlatot). Ha δ függvényt ezzel a hatékonyabb módszerrel számítjuk ki, akkor egy m hosszúságú minta összes előfordulásának megtalálása egy n hosszúságú, Σ jeleiből álló sorozatban $O(m|\Sigma|)$ előfeldolgozási és $\Theta(n)$ illesztési időt igényel.

Gyakorlatok

32.3-1. Készítsünk mintaillesztő automatát a $P = \mathtt{aabab}$ mintához, és szemléltessük ennek működését a $T = \mathtt{aaababaabaababaab}$ szövegen.

32.3-2. Adjuk meg az $\mathtt{ababbabbababbababbabb}$ mintához tartozó mintaillesztő automata állapot-átmenet diagramját. ($\Sigma = \{\mathtt{a}, \mathtt{b}\}$.)

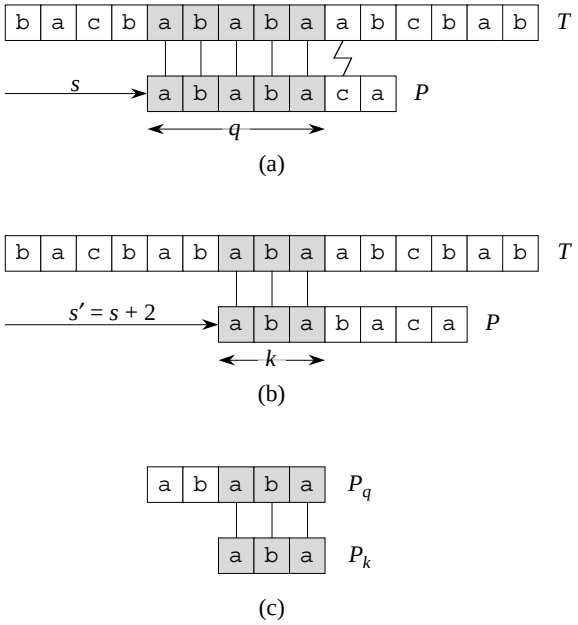
32.3-3. A P mintát ***átfedésmentesnek*** nevezzük, ha a $P_k \sqsupset P_q$ feltételből következik, hogy $k = 0$ vagy $k = q$. Írjuk le egy átfedésmentes mintához tartozó mintaillesztő automata állapot-átmenet diagramját.

32.3-4.⋆ Legyen P és P' két minta. Adjunk módszert olyan véges automata előállítására, amely a két minta összes előfordulását meghatározza. Törekedjünk az állapotok számának csökkentésére.

32.3-5. Legyen P olyan minta, amelyben lefedő jelek is szerepelhetnek (lásd 32.1-5. gyakorlatot). Készítsünk olyan véges automatát, amely megtalálja P egy előfordulását T szövegben $O(n)$ idő alatt, ahol $n = |T|$.

⋆ 32.4. Knuth–Morris–Pratt-algoritmus

Bemutatunk egy lineáris idejű mintaillesztő algoritmust, amelyet Knuth, Morris és Pratt tett közzé. Az algoritmus kiküszöböli a δ átmeneti függvény kiszámítását, és az illesztési ideje $\Theta(n)$. Egy $\pi[1\,..\,m]$ segédfüggvényt használunk, amelyet előzetesen számítunk ki a minta alapján $\Theta(m)$ idő alatt. A π tömb segítségével a δ átmeneti függvény értékei „menet közben”



32.10. ábra. A π prefix függvény. **(a)** A P = ababaca mintát úgy helyeztük a T szövegre, hogy az első $q = 5$ jel illeszkedik. A beszürkített, illeszkedő rész jeleit függőleges vonal köti össze. **(b)** Az 5 illeszkedő jel ismeretében arra következtethetünk, hogy az $s + 1$ eltolás érvénytelen, de az $s' = s + 2$ eltolás megfelel annak, amit eddig tudunk a szövegről, ezért ez érvényes lehet. **(c)** Az ilyen jellegű következtetésekhez használható információ előre kiszámítható, ha a mintát önmagával hasonlítjuk össze. Ebben az esetben látható, hogy P leghosszabb olyan prefixe, amely egyben P_5 szuffixe is P_3. Ezt előre meghatározzuk és a π tömbben tároljuk, így $\pi[5] = 3$. Ha s eltolás esetén q jel illeszkedik, akkor a következő lehetséges érvényes eltolás $s' = s + (q - \pi[q])$.

hatékonyan számolhatóak. Vázlatosan ez azt jelenti, hogy bármely $q = 0, 1, \ldots, m$ állapotra és $a \in \Sigma$ jelre, $\pi[q]$ érték adja meg $\delta(q, a)$ számítási módját, és ez független a-tól. (Később ezt pontosítjuk.) A π tömbnek csak m eleme van, míg δ-nak $O(m|\Sigma|)$, így az előfeldolgozás során megspóroljuk a $|\Sigma|$ tényezőt, ha π-t határozzuk meg δ helyett.

Minta prefix függvénye

Egy minta prefix függvénye tartalmazza azokat az ismereteket, amelyek megadják, hogyan illeszkedik a minta önmaga eltoltjaira. Ennek segítségével elkerülhetjük, hogy kizárható eltolási értékeket vizsgáljunk az egyszerű mintaillesztő algoritmusban, illetve a mintaillesztő automata δ függvényének előzetes kiszámítását.

Vizsgáljuk meg az egyszerű mintaillesztő működését. A 32.10. ábra (a) részén a P = ababaca minta látható s-sel eltolva a T szöveg kezdetéhez képest. Ebben a példában $q = 5$ jel illeszkedik, de a minta 6. jele eltér a szöveg megfelelő jelétől. Az első q jel illeszkedése alapján ismerjük a szöveg megfelelő jeleit. A szöveg ezen q jelének ismeretében azonnal meg tudjuk mondani, hogy bizonyos eltolások érvénytelenek lesznek. A konkrét példában, $s + 1$ szükségképpen érvénytelen eltolás, mivel ekkor a minta első jelének, a-nak, illeszkednie kellene egy olyan szövegbeli jelre, amely megegyezik a minta második jelével, ami b. Ugyanakkor az ábra (b) részén látható, $s' = s + 2$ eltolás a minta első három

jelét olyan szövegbeli jelekre illeszti, amelyek szükségszerűen megegyeznek. Általában a következő kérdésre kellene ismernünk a választ:

> Ha a minta $P[1\,..\,q]$ jelei illeszkednek a szöveg $T[s{+}1\,..\,s{+}q]$ jeleire, mi a legkisebb olyan $s' > s$ eltolás, amelyre fennáll
>
> $$P[1\,..\,k] = T[s'+1\,..\,s'+k], \tag{32.5}$$
>
> ahol $s'+k = s+q$?

A $T[s+1\,..\,s+q]$-ra vonatkozó ismereteink alapján s' az első s-nél nagyobb eltolás, amely nem feltétlen érvénytelen. A legjobb esetben $s' = s+q$, és az $s+1, s+2, \ldots, s+q-1$ eltolások azonnal kizárhatóak. Az s' új eltolásra felesleges ellenőriznünk a P minta első k jelének illeszkedését, mivel a (32.5) egyenlőség garantálja, hogy azok megegyeznek a szöveg megfelelő jeleivel.

A szükséges információt előre meghatározhatjuk, ha a mintát önmagával összehasonlítjuk, amint azt a 32.10. ábra (c) része szemlélteti. $T[s'+1\,..\,s'+k]$ a P_q sorozat szuffixe, mivel az ismert szöveg része. Ennek alapján a (32.5) egyenlőségben tulajdonképpen a legnagyobb olyan $k < q$ értéket keressük, amelyre $P_k \sqsupset P_q$. Így $s' = s + (q-k)$ a következő lehetséges érvényes eltolás. Célszerű az új s' eltolás illeszkedő jeleinek számát, k-t, nyilvántartani $s'-s$ helyett. Ennek segítségével mind az egyszerű, mind a véges automatát használó mintaillesztő algoritmus felgyorsítható.

A szükséges előfeldolgozást pontosítjuk a következőkben. π egy adott $P[1\,..\,m]$ mintához tartozó ***prefix függvény,*** ha $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ és

$$\pi[q] = \max\{k : k < q \text{ és } P_k \sqsupset P_q\}.$$

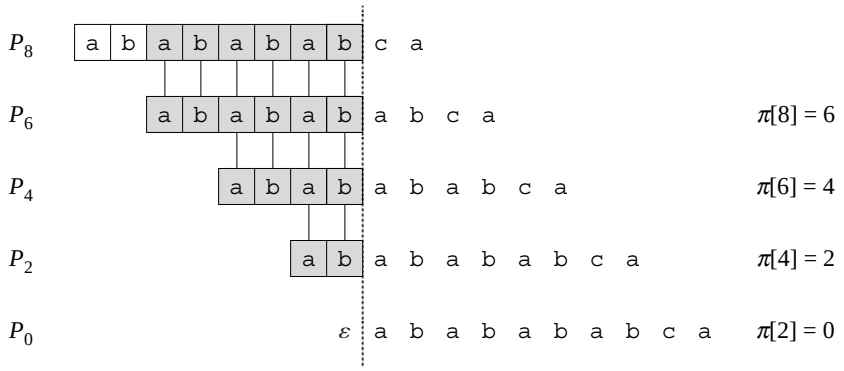
Szavakban, $\pi[q]$ a leghosszabb olyan P-beli prefix hossza, amely valódi szuffixe P_q-nak. A 32.11. ábra (a) része megadja az `abababacba` mintához tartozó π prefix függvény összes értékét.

Az alábbiakban megadjuk formálisan a Knuth–Morris–Pratt-mintaillesztő algoritmust (KMP-illesztő), amely legjobban a Véges-automata-illesztő eljárásra hasonlít. A KMP-illesztő a Prefix-függvény-számítás eljárás segítségével számítja ki π-t.

```
KMP-illesztő(T,P)
 1  n ← hossz[T]
 2  m ← hossz[P]
 3  π ← Prefix-függvény-számítás(P)
 4  q ← 0                                  ▷ Illeszkedő jelek száma.
 5  for i ← 1 to n                         ▷ Szöveg ellenőrzése balról jobbra.
 6      do while q > 0 és P[q + 1] ≠ T[i]
 7            do q ← π[q]                  ▷ A következő jel nem egyezik.
 8         if P[q + 1] = T[i]
 9            then q ← q + 1               ▷ A következő jel egyezik.
10         if q = m                        ▷ A teljes minta illeszkedik?
11            then print „A minta illeszkedik az (”i − m + 1„)-edik pozícióra”
12                 q ← π[q]                ▷ A következő illeszkedés keresése.
```

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

32.11. ábra. A 32.5. lemma szemléltetése, ha P = ababababca és $q = 8$. **(a)** A mintához tartozó π függvény. Miután $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$ és $\pi[2] = 0$, π ismételt alkalmazásával kapjuk, hogy $\pi^*[8] = \{6, 4, 2, 0\}$. **(b)** A P mintát tartalmazó sablont jobbra csúsztatjuk és feljegyezzük, ha P valamelyik P_k prefixe illeszkedik P_8 megfelelő szuffixére. Ez teljesül $k = 6, 4, 2$ és 0 esetén. Az ábra első sorában található P, szaggatott függőleges vonalat húztunk P_8 utolsó jele után. A következő sorok tartalmazzák az összes illeszkedő P_k prefixet. Az illeszkedő részeket beszürkítettük, az egyező jeleket függőleges vonalak kötik össze. Látható, hogy $\{k : k < q \wedge P_k \sqsupset P_q\} = \{6, 4, 2, 0\}$. A lemma szerint $\pi^*[q] = \{k : k < q \wedge P_k \sqsupset P_q\}$ minden q-ra.

PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P)

```
 1  m ← hossz[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5      do while k > 0 és P[k + 1] ≠ P[q]
 6             do k ← π[k]
 7         if P[k + 1] = P[q]
 8            then k ← k + 1
 9         π[q] ← k
10  return π
```

Először a fenti eljárások futási idejét elemezzük, ezután bizonyítjuk helyességüket, ami bonyolultabb feladat.

A futási idők elemzése

A PREFIX-FÜGGVÉNY-SZÁMÍTÁS futási ideje $\Theta(m)$, amit a 17.3. alfejezetben bemutatott amortizáló elemzés segítségével bizonyítunk. k potenciált rendelünk az algoritmus aktuális k értékéhez. Ennek kezdeti értéke 0 a 3. sorban. A 6. sor végrehajtása csökkenti k értékét, hiszen

$\pi[k] < k$. Ugyanakkor k nem lehet negatív, hiszen minden k-ra $\pi[k] \geq 0$. Ezenkívül csak a 8. sorban változik k, amelyben az legalább eggyel nő a **for** ciklus magjának minden egyes lefutásakor. Amikor belépünk a **for** ciklusba, akkor $k < q$, és a ciklus magja mindig eggyel növeli q-t, így $k < q$ mindig fennáll. (Ebből a 9. sor alapján az is következik, hogy $\pi[q] < q$.) A **while** ciklusmag egyes végrehajtásait (6. sor) kiegyenlíthetjük a potenciál függvény értékének csökkentésével, mert $\pi[k] < k$. A 8. sor legfeljebb eggyel növeli a függvény értékét, ezért az 5–9. sorokból álló ciklusmag amortizált költsége $O(1)$. A külső ciklus $\Theta(m)$-szer fut le, továbbá a potenciál függvény befejezéskor legalább akkora, mint az induláskor, így a PREFIX-FÜGGVÉNY-SZÁMÍTÁS legrosszabb futási ideje $\Theta(m)$.

Az előzőhöz hasonló amortizáló analízis segítségével bebizonyítható, hogy a KMP-ILLESZTŐ illesztési ideje $\Theta(n)$. Ebben az esetben potenciál függvényként q értékét használjuk.

A minta előfeldolgozásának idejét a VÉGES-AUTOMATA-ILLESZTŐ algoritmusban elért $O(m|\Sigma|)$-ről $\Theta(m)$-re csökkentettük azzal, hogy δ helyett π-t használtuk; ugyanakkor a tényleges illesztési idő felső határa $\Theta(n)$ maradt.

A prefix függvény számításának helyessége

Egy alapvető lemmát mutatunk be először, amelyből kiderül, hogy a π prefix függvény egymás utáni alkalmazásával sorban megkapjuk az összes olyan P_k prefixet, amelyek szuffixei egy adott P_q prefixnek. Legyen

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\},$$

ahol egyrészt $\pi^{(i)}[q]$ értékeket függvénykompozícióval definiáljuk, azaz $\pi^{(0)}[q] = q$, és $\pi^{(i+1)}[q] = \pi[\pi^{(i)}[q]]$, ha $i > 1$; másrészt a $\pi^*[q]$ sorozat véget ér, ha $\pi^{(t)}[q] = 0$-hoz érünk. A következő lemma, amelyet a 32.11. ábra szemléltet, jellemzi $\pi^*[q]$-t.

32.5. lemma (Prefix függvény iteráció). *Legyen P m hosszúságú minta, és π a hozzá tartozó prefix függvény. Ekkor $q = 1, 2, \ldots, m$ esetén $\pi^*[q] = \{k : k < q \wedge P_k \sqsupset P_q\}$ teljesül.*

Bizonyítás. Először belátjuk, hogy

$$i \in \pi^*[q]\text{-ből következik } P_i \sqsupset P_q. \tag{32.6}$$

Ha $i \in \pi^*[q]$, akkor $i = \pi^{(u)}[q]$ valamely u-ra. A (32.6) fennállását u szerinti indukcióval bizonyítjuk. Ha $u = 1$, akkor $i = \pi[q]$, és (32.6) teljesül, hiszen $i < q$ és $P_{\pi[q]} \sqsupset P_q$. Felhasználva, hogy $\pi[i] < i$ és $P_{\pi[i]} \sqsupset P_i$, továbbá, hogy $<$ és $\sqsupset$ tranzitív, azt kapjuk, hogy az állítás teljesül minden $\pi^*[q]$-beli i-re. Így $\pi^*[q] \subseteq \{k : k < q \wedge P_k \sqsupset P_q\}$.

A lemmában szereplő egyenlőség teljesüléséhez be kell még bizonyítani, hogy $\{k : k < q \wedge P_k \sqsupset P_q\} \subseteq \pi^*[q]$, amit indirekt módon látunk be. Tegyük fel, hogy állításunk nem igaz, és a $\{k : k < q \wedge P_k \sqsupset P_q\} - \pi^*[q]$ halmaznak van eleme, továbbá legyen j a legnagyobb ilyen elem. Ekkor $j < \pi[q]$, mert $\pi[q]$ a $\{k : k < q \wedge P_k \sqsupset P_q\}$ halmaz legnagyobb eleme és $\pi[q] \in \pi^*[q]$. Jelöljük j'-vel a legkisebb olyan egészt $\pi^*[q]$-ban, amely nagyobb j-nél. ($j' = \pi[q]$ biztosan jó, ha nem lenne más j-nél nagyobb eleme $\pi^*[q]$-nak.) Ekkor $P_j \sqsupset P_q$, mert $j \in \{k : k < q \wedge P_k \sqsupset P_q\}$, és $P_{j'} \sqsupset P_q$, mert $j' \in \pi^*[q]$; ezért a 32.1. lemma miatt $P_j \sqsupset P_{j'}$. Továbbá, j a legnagyobb j'-nél kisebb érték, amire teljesül ez a tulajdonság. Ebből következik, hogy $\pi[j'] = j$ fennáll, ugyanakkor $j' \in \pi^*[q]$, így $j \in \pi^*[q]$. Ez ellentmondás, így bebizonyítottuk a lemmát. ■

A PREFIX-FÜGGVÉNY-SZÁMÍTÁS $q = 1, 2, \ldots, m$ értékekre sorban határozza meg $\pi[q]$-t. Az algoritmus 2. sorában szereplő $\pi[1] = 0$ választás nyilván helyes, mert $\pi[q] < q$ minden q-ra. A következő lemma és következménye segítségével fogjuk bebizonyítani, hogy a PREFIX-FÜGGVÉNY-SZÁMÍTÁS helyes $\pi[q]$ értékeket számol ki akkor is, ha $q > 1$.

32.6. lemma. *Legyen P m hosszúságú minta, és π a hozzá tartozó prefix függvény. Bármely $q = 1, 2, \ldots, m$-re, ha $\pi[q] > 0$, akkor $\pi[q] - 1 \in \pi^*[q-1]$.*

Bizonyítás. Ha $r = \pi[q] > 0$, akkor $r < q$ és $P_r \sqsupset P_q$, ezért $r - 1 < q - 1$ és P_k és P_q utolsó jelét elhagyva kapjuk, hogy $P_{r-1} \sqsupset P_{q-1}$. Így a 32.4. lemma miatt $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. ■

$q = 2, 3, \ldots, m$ értékekre definiáljuk az $E_{q-1} \subseteq \pi^*[q]$ halmazt a következőképpen:

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \wedge P_k \sqsupset P_{q-1} \wedge P[k+1] = P[q]\} \quad \text{(32.5. lemma miatt)} \\ &= \{k : k < q-1 \wedge P_{k+1} \sqsupset P_q\}. \end{aligned}$$

Az E_{q-1} halmaz olyan $k < q - 1$ értékeket tartalmaz, amelyekre $P_k \sqsupset P_{q-1}$. Ugyanakkor a k elemekre $P_{k+1} \sqsupset P_q$ is teljesül, mert $P[k+1] = P[q]$. Másképp fogalmazva, E_{q-1} elemei olyan $k \in \pi^*[q-1]$ értékek, amelyek esetén P_k sorozatot kiegészíthetjük P_{k+1}-re úgy, hogy ez P_q megfelelő szuffixe legyen.

32.7. következmény. *Legyen P m hosszúságú minta, és π a hozzá tartozó prefix függvény. Bármely $q = 2, 3, \ldots, m$ értékre*

$$\pi[q] = \begin{cases} 0, & \text{ha } E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\}, & \text{ha } E_{q-1} \neq \emptyset. \end{cases}$$

Bizonyítás. Ha E_{q-1} üres, akkor nincs olyan $k \in \pi^*[q-1]$ (beleértve $k = 0$ esetet is), amelyre P_k-t kiegészíthetnénk P_{k+1}-re úgy, hogy az P_q szuffixe legyen. Ezért $\pi[q] = 0$.

Ha E_{q-1} nem üres, akkor minden $k \in E_{q-1}$ esetén $k + 1 < q$ és $P_{k+1} \sqsupset P_q$. Felhasználva $\pi[q]$ definícióját kapjuk, hogy:

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\}. \tag{32.7}$$

Vegyük észre, hogy $\pi[q] > 0$. Legyen $r = \pi[q] - 1$ úgy, hogy $r + 1 = \pi[q]$. Mivel $r + 1 > 0$, $P[r+1] = P[q]$ is fennáll. Továbbá a 32.6. lemma miatt $r \in \pi^*[q-1]$. Ezért $r \in E_{q-1}$, továbbá $r \leq \max\{k \in E_{q-1}\}$, vagy ami ezzel ekvivalens:

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\}. \tag{32.8}$$

A (32.7) és (32.8) egyenlőtlenségek együtt a bizonyítandó egyenlőséget adják. ■

Most folytatjuk annak bizonyítását, hogy PREFIX-FÜGGVÉNY-SZÁMÍTÁS helyes π értékeket határoz meg. Az eljárásban a **for** ciklus magjának végrehajtása előtt mindig teljesül, hogy $k = \pi[q-1]$. A ciklus megkezdése előtt a 2. és 3. sor, a továbbiakban pedig a 9. sor gondoskodik erről. Az 5–8. sorok módosítják k értékét úgy, hogy az $\pi[q]$ legyen. Az 5–6. sorokban lévő ciklus ellenőriz minden $k \in \pi^*[q-1]$ értéket, amíg olyat nem talál, amelyre

$P[k+1] = P[q]$. Ekkor k az E_{q-1} halmaz legnagyobb eleme lesz, így a 32.7. következmény miatt $\pi[q]$ értékének $k+1$ választható. Ha nincs ilyen k, akkor $k = 0$ a 7. sorban. Ha $P[1] = P[q]$, akkor k és $\pi[q]$ értékét is 1-re kell állítanunk, ellenkező esetben $\pi[q]$-t nullázni kell és k értékét nem kell változtatni. A 7–9. sorok minden esetben helyes értéket rendelnek k-hoz és $\pi[q]$-hoz. Ezzel bebizonyítottuk a PREFIX-FÜGGVÉNY-SZÁMÍTÁS helyességét.

A KMP algoritmus helyessége

A KMP-ILLESZTŐ eljárást tekinthetjük a VÉGES-AUTOMATA-ILLESZTŐ eljárás átírásának. Bebizonyítjuk, hogy a KMP-ILLESZTŐ 6–9. sora ekvivalens a VÉGES-AUTOMATA-ILLESZTŐ 4. sorával, amelyben q értékét $\delta(q, T[i])$-re állítottuk. Azonban nem egy előre kiszámolt $\delta(q, T[i])$ értéket használunk, hanem ezt szükség esetén π segítségével újra meghatározzuk. Ha belátjuk, hogy a KMP-ILLESZTŐ a VÉGES-AUTOMATA-ILLESZTŐ működését szimulálja, akkor a KMP-ILLESZTŐ helyessége következik a VÉGES-AUTOMATA-ILLESZTŐ helyességéből. (A KMP-ILLESZTŐ 12. sorára később még kitérünk.)

A KMP-ILLESZTŐ helyessége következik abból az állításból, hogy $\delta(q, T[i]) = 0$ vagy $\delta(q, T[i]) - 1 \in \pi^*[q]$ fennáll. Az állítás igazolásához legyen $k = \delta(q, T[i])$. Ekkor δ és σ definíciója miatt $P_k \sqsupset P_q T[i]$. Ezért $k = 0$, vagy pedig $k \geq 1$ és $P_{k-1} \sqsupset P_q$ (P_k és $P_q T[i]$ utolsó jelének elhagyásával). A második esetben $k - 1 \in \pi^*[q]$. Tehát, vagy $k = 0$, vagy $k - 1 \in \pi^*[q]$, így az állítás igaz.

Jelöljük q'-vel q 6. sor megkezdése előtti értékét. A 32.5. lemmából adódó $\pi^*[q] = \{k : k < q \wedge P_k \sqsupset P_q\}$ felhasználásával belátjuk, hogy a ciklusban $q \leftarrow \pi[q]$ utasítással a $\{k : P_k \sqsupset P_{q'}\}$ halmaz elemeit soroljuk fel. A 6–9. sorok $\pi^*[q']$ elemeit csökkenő sorrendben vizsgálják, és így határozzák meg $\delta(q', T[i])$-t. Az állítás alapján, az algoritmusban kezdetben $q = \phi(T_{i-1}) = \sigma(T_{i-1})$, és $q \leftarrow \pi[q]$-t addig hajtjuk végre, amíg $q = 0$ vagy $P[q+1] = T[i]$ be nem következik. Az első esetben $\delta(q', T[i]) = 0$, a második esetben pedig q a legnagyobb $E_{q'}$-beli elem, ezért a 32.7. következmény miatt $\delta(q', T[i]) = q + 1$.

A 12. sor azért kell a KMP-ILLESZTŐ eljárásban, hogy ne hivatkozhassunk a 6. sorban $P[m+1]$-re, miután megtaláltuk P egy előfordulását. (Továbbra is érvényes marad, hogy a 6. sor következő végrehajtásakor $q = \sigma(T_{i-1})$). Ez belátható a 32.4-6. gyakorlathoz adott útmutatás, $\forall a \in \Sigma : \delta(m, a) = \delta(\pi[m], a)$, illetve az ezzel ekvivalens $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ felhasználásával.) A helyességbizonyítás további részei következnek a VÉGES-AUTOMATA-ILLESZTŐ helyességéből, mert igazoltuk, hogy a KMP-ILLESZTŐ szimulálja a VÉGES-AUTOMATA-ILLESZTŐ működését.

Gyakorlatok

32.4-1. Határozzuk meg az ababbabbababbababbabb mintához tartozó π prefix függvényt, ha $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

32.4-2. Adjunk felső korlátot $\pi^*[q]$ méretére. (Tekintsük ezt q függvényének.) Mutassunk példát arra, hogy ez éles becslés.

32.4-3. Mutassuk meg, hogyan határozhatjuk meg a P minta előfordulásait a T szövegben a PT sorozathoz rendelt π függvény felhasználásával. (PT $m+n$ hosszúságú sorozat, a P és T sorozatok konkatenációja.)

32.4-4. Mutassuk meg, hogyan javítható a KMP-ILLESZTŐ úgy, hogy a 7. sorban π helyébe π'-t írunk (a 12. sor nem változik), ahol $q = 1, 2, \ldots, m$-re π' értékeit rekurzív módon, az

alábbi egyenlőséggel határozzuk meg:

$$\pi'[q] = \begin{cases} 0, & \text{ha } \pi[q] = 0, \\ \pi'[\pi[q]], & \text{ha } \pi[q] \neq 0 \text{ és } P[\pi[q]+1] = P[q+1], \\ \pi[q], & \text{ha } \pi[q] \neq 0 \text{ és } P[\pi[q]+1] \neq P[q+1]. \end{cases}$$

Indokoljuk a módosított algoritmus helyességét, és magyarázzuk meg, miért vezet ez a módosítás javuláshoz.

32.4-5. Adjunk lineáris idejű algoritmust annak eldöntésére, hogy T sorozat megegyezik-e egy másik T' sorozat ciklikus elforgatottjával. Például `arc` és `car` egymás ciklikus elforgatottjai.

32.4-6.⋆ Adjunk hatékony algoritmust egy P mintához tartozó mintaillesztő automata δ átmeneti függvényének meghatározására. Az algoritmus futási ideje $O(m|\Sigma|)$ legyen. (*Útmutatás.* Lássuk be, hogy $\delta(q, a) = \delta(\pi[q], a)$, ha $q = m$ vagy $P[q+1] \neq a$.)

Feladatok

32-1. Ismétlési tényezőn alapuló mintaillesztés

Jelölje y^i az y sorozat önmagával vett i-szeres konkatenációját. Például $(\texttt{ab})^3 = \texttt{ababab}$. Azt mondjuk, hogy egy $x \in \Sigma^*$ sorozatnak r ***ismétlési tényezője,*** ha $x = y^r$ valamely $y \in \Sigma^*$ sorozatra és $r > 0$ értékre. Jelölje $\rho(x)$ a legnagyobb olyan r értéket, amely ismétlési tényezője x-nek.

a. Adjunk hatékony algoritmust a $P[1\,..\,m]$ mintához tartozó $\rho(P_i)$ értékek meghatározására, ahol $i = 1, 2, \ldots, m$. Mi az algoritmus futási ideje?

b. Tetszőleges $P[1\,..\,m]$ minta esetén legyen $\rho^*(P) = \max_{1 \leq i \leq m} \rho(P_i)$. Bizonyítsuk be, hogy ha a P minta egy véletlenszerűen kiválasztott m hosszúságú bináris sorozat, akkor $\rho^*(P)$ várható értéke $O(1)$.

c. Lássuk be, hogy a következő mintaillesztő algoritmus $O(\rho^*(P)n+m)$ idő alatt megtalálja a P minta összes előfordulását a $T[1\,..\,n]$ szövegben.

```
ISMÉTLÉSES-ILLESZTŐ(P, T)
 1  m ← hossz[P]
 2  n ← hossz[T]
 3  k ← 1 + ρ*(P)
 4  q ← 0
 5  s ← 0
 6  while s ≤ n − m
 7        do if T[s + q + 1] = P[q + 1]
 8              then q ← q + 1
 9                   if q = m
10                      then print „A minta illeszkedik az (”s + 1„)-edik pozícióra”
11           if q = m vagy T[s + q + 1] ≠ P[q + 1]
12              then s ← s + max(1, ⌈q/k⌉)
13                   q ← 0
```

Az algoritmust Galil és Seiferas fejlesztette ki. A bemutatott gondolatmenet jelentős továbbfejlesztésével egy lineáris idejű mintaillesztő algoritmushoz jutottak, amely csak $O(1)$ tárterületet használ P-n és T-n kívül.

Megjegyzések a fejezethez

A mintaillesztés és a véges automaták elmélete közötti kapcsolatot mutatja be Aho, Hopcroft és Ullman [5]. A Knuth–Morris–Pratt-algoritmust [187] Knuth és Pratt, illetve Morris egymástól függetlenül fejlesztette ki; munkájukat közösen publikálták. A Rabin–Karp-algoritmust Rabin és Karp tette közzé [175]. Galil és Seiferas [107] egy érdekes, determinisztikus, lineáris idejű mintaillesztő algoritmust adott meg, amelynek tárigénye $O(1)$ eltekintve a minta és a szöveg által igényelt memóriától.

33. Geometriai algoritmusok

Ebben a fejezetben geometriai algoritmusokat tanulmányozunk. A geometriai algoritmusok alkalmazásait a korszerű mérnöki tudományok és a matematika területén megtaláljuk többek között a számítógépes tervezésben és grafikában, a robotikában, a VLSI-tervezésben és a statisztikában. Egy geometriai algoritmus bemenete általában egy geometriai objektumhalmaz leírása, például pontok, szakaszok vagy egy poligon óramutató járásával ellentétes körüljárás szerint felsorolt csúcsai. A kimenet gyakran csak válasz egy, az objektumokkal kapcsolatos kérdésre, mondjuk, hogy van-e az egyenesek között metsző, de lehet egy új alakzat is, például egy ponthalmaz konvex burka (a legszűkebb befoglaló konvex poligon).

Ebben a fejezetben néhány kétdimenziós, azaz síkbeli geometriai algoritmust tekintünk át. Ezek bemeneti objektumait egy-egy $\{p_1, p_2, p_3, \ldots\}$ ponthalmaz írja le, ahol $p_i = (x_i, y_i)$, $x_i, y_i \in \mathbf{R}$. Például egy n csúcsú P poligon megadható csúcspontjainak $\langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$ felsorolásával, abban a sorrendben, ahogy azok megjelennek P pereme mentén. Geometriai algoritmusokról három- vagy akár magasabb dimenziós terekben is beszélhetünk, de ezek a feladatok és megoldásaik nagyon nehezen szemléltethetők. Szerencsére már két dimenzióban is jó ízelítőt kaphatunk a geometriai algoritmusok módszereiből.

A 33.1. alfejezetben megmutatjuk, hogyan lehet pontosan és hatékonyan megválaszolni az egyenes szakaszokra vonatkozó alapvető kérdéseket: két közös végpontban csatlakozó szakasz egyike a másikhoz képest az óramutató járásával megegyező vagy ellentétes forgásirányba esik-e, merre fordulunk, amikor bejárunk két egymás után következő szakaszt, és vajon két adott szakasz metszi-e egymást. A 33.2. alfejezetben bemutatjuk a „söprés”-nek nevezett módszert, és ezt egy $O(n \lg n)$-es futási idejű algoritmus kidolgozása során alkalmazzuk annak eldöntésére, hogy van-e metsző pár egy n szakaszból álló halmazban. A 33.3. alfejezet két „forgatásos söprésen” alapuló algoritmust ad egy n elemű ponthalmaz konvex burkának meghatározására: az $O(n \lg n)$ futási idejű Graham-féle pásztázást és Jarvis menetelését, amely $O(nh)$ ideig tart, ahol h a konvex burok csúcsainak száma. Végül a 33.4. alfejezet egy $O(n \lg n)$ idejű, „oszd-meg-és-uralkodj” elvű algoritmust közöl egymáshoz legközelebbi két pont megkeresésére egy n elemű ponthalmazban a síkon.

33.1. A szakaszok tulajdonságai

A fejezetben tárgyalt geometriai algoritmusok közül többhöz is szükség lesz arra, hogy a szakaszok tulajdonságaira vonatkozó kérdéseket meg tudjuk válaszolni. Két különböző

$p_1 = (x_1, y_1)$ és $p_2 = (x_2, y_2)$ pont ***konvex kombinációja*** egy $p_3 = (x_3, y_3)$ pont, ha valamely $0 \leq \alpha \leq 1$ valós számra $x_3 = \alpha x_1 + (1 - \alpha)x_2$ és $y_3 = \alpha y_1 + (1 - \alpha)y_2$. Ezt úgy is írhatjuk, hogy $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Szemléletesen p_3 a p_1 és p_2 pontokon áthaladó egyenes egy pontja, mégpedig vagy p_1 vagy p_2, vagy a kettő közé esik. Két adott, különböző p_1 és p_2 pont esetén a $\overline{p_1p_2}$ ***szakasz*** p_1 és p_2 konvex kombinációinak halmaza. A p_1 és p_2 pontokat a $\overline{p_1p_2}$ szakasz ***végpontjainak*** nevezzük. Néha p_1 és p_2 sorrendje is számít, ilyenkor a $\overrightarrow{p_1p_2}$ ***irányított szakaszról*** beszélünk. Ha p_1 a koordinátarendszer $(0, 0)$ ***kezdőpontja,*** akkor a $\overrightarrow{p_1p_2}$ irányított szakasz egy p_2 ***vektorként*** is kezelhető.

Ebben a fejezetben a következő kérdéseket vizsgáljuk:

1. Adott két irányított szakasz, $\overrightarrow{p_0p_1}$ és $\overrightarrow{p_0p_2}$. A közös p_0 végpont körül, $\overrightarrow{p_0p_2}$-ből kiindulva $\overrightarrow{p_0p_1}$ az óramutató járásával egyező vagy ellentétes forgásirányba esik-e?
2. Adott a $\overline{p_0p_1}$ és a $\overline{p_1p_2}$ szakasz. Ha folyamatosan bejárjuk a $\overline{p_0p_1}$-et, majd a $\overline{p_1p_2}$-t, vajon balra fordulunk-e a p_1 pontban?
3. Metszi-e egymást a $\overline{p_1p_2}$ és a $\overline{p_3p_4}$ szakasz?

Az adott síkbeli pontok felvételére nincsenek megkötések.

Mindhárom kérdés konstans időben megválaszolható, ami nem meglepő, mert mindegyikük bemeneti mérete $O(1)$. Sőt, módszereink csak összeadást, kivonást, szorzást és összehasonlítást tartalmaznak. Így nem lesz szükségünk sem osztásra, sem trigonometrikus függvényekre, mely műveletek tovább tarthatnak, és hajlamosak növelni a kerekítési hibát. Például az „egyszerű” módszer annak eldöntésére, hogy metszi-e egymást két szakasz – írjuk fel mindkét szakasz egyenesét $y = mx + b$ alakban (m a meredekség, b az y-tengelymetszet), keressük meg az egyenesek metszéspontját, és ellenőrizzük, hogy ez rajta van-e mindkét szakaszon –, osztást használ a metszéspont kiszámításakor. Amikor a szakaszok közel párhuzamosak, ez a módszer nagyon érzékeny a számítógép osztási műveletének pontosságára. Az ebben az alfejezetben közölt osztás nélküli módszer sokkal megbízhatóbb.

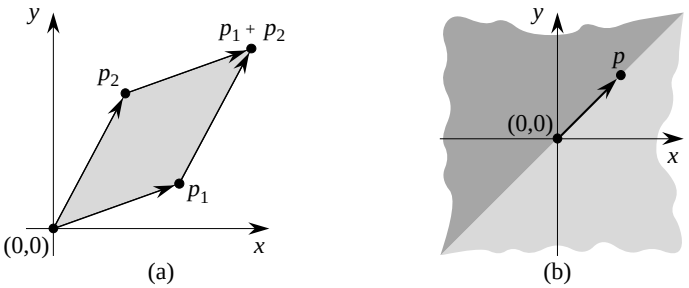
A keresztszorzat

Alfejezetünk módszereinek középpontjában keresztszorzatok kiszámítása áll. Tekintsük a 33.1(a) ábrán látható p_1 és p_2 vektorokat. A $p_1 \times p_2$ ***keresztszorzat*** a $(0, 0)$, p_1, p_2 és $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$ pontok által alkotott paralelogramma előjeles területeként értelmezhető. Egy ekvivalens, de hasznosabb definíció a keresztszorzatot egy mátrix determinánsaként adja meg:[1]

$$\begin{aligned} p_1 \times p_2 &= \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1y_2 - x_2y_1 \\ &= -p_2 \times p_1. \end{aligned}$$

Ha $p_1 \times p_2$ pozitív, akkor p_1 az óramutató járásával egyező forgásirányba esik p_2-höz képest, az origóból felmérve. Ha viszont ez a szorzat negatív, akkor p_1 az óramutató járásával ellentétes forgásirányba esik p_2-től. (Lásd a 33.1-1. gyakorlatot.) A 33.1(b) ábra p-hez képest az

[1] A keresztszorzat valójában háromdimenziós fogalom: egy mind p_1-re, mind p_2-re merőleges, azokkal jobbsodrású rendszert alkotó vektor, melynek hossza $|x_1y_2 - x_2y_1|$. Ebben a fejezetben azonban kényelmesebb, ha az $x_1y_2 - x_2y_1$ értéket tekintjük p_1 és p_2 keresztszorzatának.



33.1. ábra. (a) A p_1 és p_2 vektorok keresztszorzata a paralelogramma előjeles területe. **(b)** A világosabb árnyalatú tartomány azokat a vektorokat tartalmazza, amelyek p-től az óramutató járásával egyező forgásirányba esnek. A sötétebb árnyalatú tartomány pedig azokat, amelyek p-től az óramutató járásával ellentétes forgásirányba esnek.

óramutató járásával megegyező, és az azzal ellentétes forgásirányba eső tartományt szemlélteti. Határhelyzetben a keresztszorzat nulla. Ebben az esetben a vektorok ***kollineárisak,*** és vagy azonos, vagy ellentétes irányításúak.

Annak eldöntéséhez, hogy egy $\overrightarrow{p_0p_1}$ irányított szakasz óramutató járásával megegyező vagy azzal ellentétes forgásirányba esik-e egy másik, $\overrightarrow{p_0p_2}$ irányított szakaszhoz képest, közös p_0 végpontból felmérve őket, egyszerűen eltoljuk a vektorokat úgy, hogy p_0 az origóba essen. Azaz jelölje $p_1 - p_0$ azt a $p_1' = (x_1', y_1')$ vektort, amelyben $x_1' = x_1 - x_0$ és $y_1' = y_1 - y_0$, és definiáljuk hasonlóan $(p_2 - p_0)$-t is. Majd számoljuk ki a

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

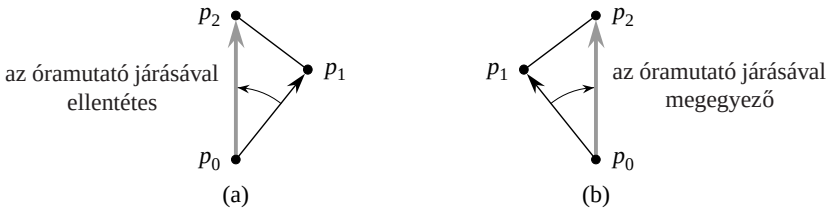
keresztszorzatot. Ha ez a keresztszorzat pozitív, akkor a $\overrightarrow{p_0p_1}$ az óramutató járásával megegyező forgásirányba esik $\overrightarrow{p_0p_2}$-től, ha negatív, akkor az óramutató járásával ellentétes forgásirányba.

Balra vagy jobbra fordul-e a következő szakasz?

A következő kérdés: vajon két egymást követő szakasz, $\overline{p_0p_1}$ és $\overline{p_1p_2}$ jobbra vagy balra fordul-e el egymáshoz képest a p_1 pontban. Ezzel egyenértékű, ha olyan módszert keresünk, amely meghatározza, hogy merre fordul el a $\angle p_0p_1p_2$ szög. Keresztszorzat segítségével a kérdést a szög kiszámolása nélkül is meg tudjuk válaszolni. Amint az a 33.2. ábrán látható, egyszerűen csak azt vizsgáljuk, hogy a $\overrightarrow{p_0p_2}$ irányított szakasz az óramutató járásával megegyező vagy azzal ellentétes forgásirányban helyezkedik-e el a $\overrightarrow{p_0p_1}$ irányított szakaszhoz képest. Ehhez kiszámoljuk a $(p_2 - p_0) \times (p_1 - p_0)$ keresztszorzatot. Ha e keresztszorzat előjele negatív, akkor $\overrightarrow{p_0p_2}$ az óramutató járásával ellentétes forgásirányba esik $\overrightarrow{p_0p_1}$-hez képest, így p_1-ben balra fordulunk. A pozitív keresztszorzat az óramutató járásával egyező irányítást és jobbra fordulást jelez. A keresztszorzat 0 értéke azt jelenti, hogy a p_0, p_1 és p_2 pontok kollineárisak.

Metszi-e egymást két szakasz?

Annak eldöntésére, hogy két szakasz metszi-e egymást, ellenőrizzük, hogy az egyik szakasz átfogja-e a másik egyenesét. A $\overline{p_1p_2}$ szakasz ***átfog*** egy egyenest, ha a p_1 pont az egyenes egyik oldalára, a p_2 pont pedig a másik oldalára esik. Határesetben p_1 vagy p_2 illeszkedik



33.2. ábra. Keresztszorzat alkalmazása annak eldöntésére, merre fordul a $\overline{p_0p_1}$ után következő $\overline{p_1p_2}$ szakasz a p_1 pontban. Azt vizsgáljuk, hogy a $\overrightarrow{p_0p_2}$ irányított szakasz az óramutató járásával megegyező vagy ellentétes forgásirányba esik-e a $\overrightarrow{p_0p_1}$ irányított szakaszhoz képest. **(a)** Ha az óramutató járásával ellentétes irányba esik, akkor balra fordul. **(b)** Ha az óramutató járásával megegyező irányba, akkor jobbra.

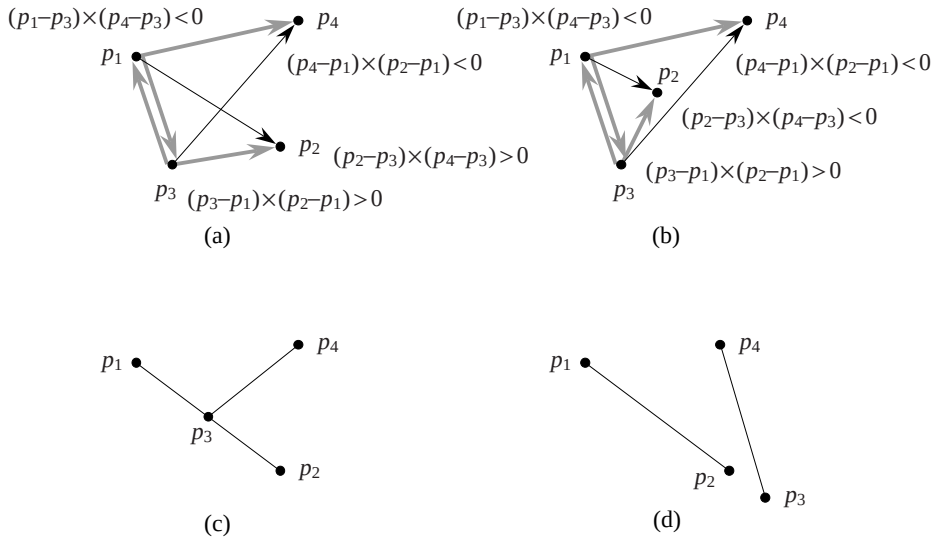
az egyenesre. Két szakasz akkor és csak akkor metszi egymást, ha a következő két feltétel valamelyike (vagy mindkettő) fennáll:

1. Mindkét szakasz átfogja a másik egyenesét.
2. Az egyik szakasz egyik végpontja illeszkedik a másik szakaszra. (Ez a feltétel felel meg a határesetnek.)

A következő eljárások valósítják meg ezt az ötletet. A METSZŐ-SZAKASZOK eljárás IGAZ-at ad vissza, ha a $\overline{p_1p_2}$ és $\overline{p_3p_4}$ szakaszok metszik egymást, és HAMIS-at, ha nem. Hívja a FORGÁSIRÁNY eljárást, amely a fenti keresztszorzat segítségével egy pontnak a másik kettő egyeneséhez viszonyított helyzetét számolja, és a SZAKASZON eljárást, amely meghatározza, vajon egy pont, melyről tudjuk, hogy kollineáris a másik kettővel, rajta van-e az általuk meghatározott szakaszon is.

```
METSZŐ-SZAKASZOK(p1, p2, p3, p4)
 1  d1 ← FORGÁSIRÁNY(p3, p4, p1)
 2  d2 ← FORGÁSIRÁNY(p3, p4, p2)
 3  d3 ← FORGÁSIRÁNY(p1, p2, p3)
 4  d4 ← FORGÁSIRÁNY(p1, p2, p4)
 5  if ((d1 > 0 és d2 < 0) vagy (d1 < 0 és d2 > 0)) és
       ((d3 > 0 és d4 < 0) vagy (d3 < 0 és d4 > 0))
 6    then return IGAZ
 7  elseif d1 = 0 és SZAKASZON(p3, p4, p1)
 8        then return IGAZ
 9  elseif d2 = 0 és SZAKASZON(p3, p4, p2)
10        then return IGAZ
11  elseif d3 = 0 és SZAKASZON(p1, p2, p3)
12        then return IGAZ
13  elseif d4 = 0 és SZAKASZON(p1, p2, p4)
14        then return IGAZ
15  else return HAMIS
```

```
FORGÁSIRÁNY(pi, pj, pk)
1  return (pk − pi) × (pj − pi)
```



33.3. ábra. A METSZŐ-SZAKASZOK eljárás esetei. **(a)** A $\overline{p_1p_2}$ és $\overline{p_3p_4}$ szakaszok átfogják egymás egyeneseit. Mivel a $\overline{p_3p_4}$ szakasz átfogja a $\overline{p_1p_2}$ szakasz egyenesét, a $(p_3-p_1)\times(p_2-p_1)$ és $(p_4-p_1)\times(p_2-p_1)$ keresztszorzatok előjelei különböznek. Mivel a $\overline{p_1p_2}$ szakasz átfogja a $\overline{p_3p_4}$ szakasz egyenesét, a $(p_1-p_3)\times(p_4-p_3)$ és $(p_2-p_3)\times(p_4-p_3)$ keresztszorzatok előjelei különböznek. **(b)** A $\overline{p_3p_4}$ szakasz átfogja a $\overline{p_1p_2}$ egyenesét, de a $\overline{p_1p_2}$ nem fogja át $\overline{p_3p_4}$ egyenesét. A $(p_1-p_3)\times(p_4-p_3)$ és a $(p_2-p_3)\times(p_4-p_3)$ keresztszorzat előjele megegyezik. **(c)** A p_3 pont a $\overline{p_1p_2}$ szakasz egyenesére illeszkedik, mégpedig p_1 és p_2 között van. **(d)** A p_3 pont a $\overline{p_1p_2}$ szakasz egyenesére illeszkedik, de nincs p_1 és p_2 között. A szakaszok nem metszik egymást.

SZAKASZON(p_i, p_j, p_k)

1 **if** $\min(x_i, x_j) \le x_k \le \max(x_i, x_j)$ és $\min(y_i, y_j) \le y_k \le \max(y_i, y_j)$
2 **then return** IGAZ
3 **else return** HAMIS

A METSZŐ-SZAKASZOK eljárás a következőképpen működik. Az 1–4. sorok az egyes p_i végpontok d_i forgásirány-jellemzőjét számolják a másik szakaszhoz viszonyítva. Ha egyik d_i relatív forgásirány sem nulla, akkor könnyen meghatározhatjuk, hogy a $\overline{p_1p_2}$ és $\overline{p_3p_4}$ szakaszok metszik-e egymást. A $\overline{p_1p_2}$ szakasz átfogja a $\overline{p_3p_4}$ szakasz egyenesét, ha a $\overrightarrow{p_3p_1}$ és $\overrightarrow{p_3p_2}$ irányított szakaszok különböző forgásirányba esnek $\overrightarrow{p_3p_4}$-hez képest. Ebben az esetben d_1 és d_2 előjele különböző. Hasonlóan $\overline{p_3p_4}$ átfogja $\overline{p_1p_2}$ egyenesét, ha d_3 és d_4 előjele különböző. Ha az 5. sor feltétele teljesül, akkor a szakaszok átfogják egymást, és a METSZŐ-SZAKASZOK eljárás IGAZ értékkel tér vissza. A 33.3(a) ábrán látható ez az eset. Egyébként a szakaszok nem fogják át egymás egyeneseit, de határeset még előfordulhat. Ha egyik relatív forgásirány sem nulla, nem fordulhat elő határeset. Ekkor a 7–13. sorok egyik feltétele sem teljesül, így a METSZŐ-SZAKASZOK eljárás HAMIS értékkel tér vissza a 15. sorból. A 33.3(b) ábrán látható ez az eset.

Határeset olyankor fordul elő, amikor a d_k relatív forgásirányok valamelyike 0. Ilyenkor tudjuk, hogy p_k illeszkedik a másik szegmens egyenesére. Ezen belül akkor és csak akkor illeszkedik a másik szegmensre is, ha a másik szegmens végpontjai között van. A SZAKASZON eljárás azt adja vissza, hogy p_k a $\overline{p_ip_j}$ szakasz, azaz a 7–13. sorok hívásai szerint a másik

szakasz végpontjai között van-e. Az eljárás feltételezi, hogy p_k a $\overline{p_i p_j}$ szakasz egyenesére illeszkedik. A 33.3(c) és (d) ábra az egyenesre illeszkedő pontok eseteit mutatja. A 33.3(c) ábrán p_3 a $\overline{p_1 p_2}$ szakaszon van, ezért a METSZŐ-SZAKASZOK eljárás IGAZ értéket ad vissza a 12. sorban. Egyik végpont sem esik a másik szakaszra a 33.3(d) ábrán, így a METSZŐ-SZAKASZOK eljárás HAMIS értéket ad vissza a 15. sorban.

A keresztszorzat egyéb alkalmazásai

A fejezet következő alfejezeteiben a keresztszorzat további alkalmazási lehetőségeivel is találkozhatunk. A 33.3. alfejezetben szükségünk lesz egy ponthalmaz adott középpont körüli, szög szerinti rendezésére. Ebben a rendezésben a szükséges összehasonlításokat végezhetjük keresztszorzat segítségével, lásd a 33.1-3. gyakorlatot. A 33.2. alfejezetben piros-fekete fákat fogunk használni egy szakaszhalmaz függőleges rendezettségének fenntartására. A kulcsok explicit értékeinek tárolását elkerülendő, annak eldöntése során, hogy az adott függőleges egyenest metsző két szakasz közül melyik van a másik fölött, a kulcsok összehasonlítását a piros-fekete fa kódjában keresztszorzat-számítással fogjuk helyettesíteni.

Gyakorlatok

33.1-1. Bizonyítsuk be, hogy ha $p_1 \times p_2$ pozitív, akkor az origóból ((0, 0) pont) nézve a p_1 vektor az óramutató járásával egyező irányba esik a p_2 vektortól, ha viszont ez a keresztszorzat negatív, akkor p_1 az óramutató járásával ellentétes irányba esik p_2-től.

33.1-2. Powell professzor javaslata szerint elég csak az x-dimenziót tesztelni a SZAKASZON eljárás első sorában. Mutassuk meg, hogy a professzornak nincs igaza.

33.1-3. A p_1 pontnak a p_0-beli kezdőpont körüli ***poláris szöge*** a $p_1 - p_0$ vektor (poláris) szöge közönséges polárkoordináta-rendszerben. Például a $(3, 5)$ pontnak a $(2, 4)$ kezdőpont körüli poláris szöge az $(1, 1)$ vektor (poláris) szöge, ami 45 fok, vagy ívmértékben $\pi/4$. A $(3, 3)$ pont $(2, 4)$ körüli poláris szöge az $(1, -1)$ vektor (poláris) szöge, ami 315 fok, vagy ívmértékben $7\pi/4$. Írjunk pszeudokódot n pont $\langle p_1, p_2, \ldots, p_n \rangle$ sorozatának poláris szög szerinti rendezésére adott p_0 kezdőpont körül. Az eljárás $O(n \lg n)$ ideig tartson, és használjon keresztszorzatot a szögek összehasonlítására.

33.1-4. Hogyan lehet meghatározni $O(n^2 \lg n)$ idejű algoritmussal, hogy egy n elemű ponthalmazban van-e kollineáris ponthármas?

33.1-5. A ***poligon*** egy szakaszokból álló zárt görbe a síkon. Más szóval egy önmagába záródó görbe, amelyet egyenes szakaszoknak, a ***poligon oldalainak*** sorozata alkot. Két egymás után következő oldal a poligon egy ***csúcsában*** kapcsolódik egymáshoz. Általában fel fogjuk tételezni, hogy a poligon ***egyszerű,*** azaz nem metszi önmagát. Egy egyszerű poligon által körbezárt pontok halmaza a síkon a poligon ***belseje,*** a poligon pontjai alkotják a poligon ***határát,*** a poligonon kívüli pontok pedig a poligon ***külsejét.*** Egy egyszerű poligon ***konvex,*** ha bármely két pontra, melyek elemei a poligon belsejének vagy határának, az őket összekötő szakasz összes pontja szintén a poligon belsejében vagy határán van.

Amundsen professzor a következő módszert javasolja annak meghatározására, hogy egy n elemű $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ pontsorozat egy konvex poligon egymás után következő csúcsait tartalmazza-e. A válasz „igaz”, ha a $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, \text{n-1}\}$ halmaz, melyben az indexre az összeadás modulo n értendő, vagy csak balra vagy csak jobbra fordulásokat tartalmaz. Egyébként a válasz „hamis”. Mutassuk meg, hogy ez a módszer, bár lineáris futási

idejű, nem mindig ad helyes eredményt. Módosítsuk a professzor módszerét úgy, hogy lineáris időben mindig helyes megoldást adjon.

33.1-6. Adott a $p_0 = (x_0, y_0)$ pont. A p_0-ból kiinduló ***jobb vízszintes sugár*** a $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ és } y_i = y_0\}$ ponthalmaz, vagyis azon pontok halmaza, amelyek jobbra esnek p_0-tól, p_0-lal együtt. Hogyan dönthető el $O(1)$ időben, hogy egy adott p_0-ból kiinduló jobb vízszintes sugár metszi-e a $\overline{p_1 p_2}$ szakaszt? Vezessük vissza a feladatot annak eldöntésére, hogy két szakasz metszi-e egymást.

33.1-7. Az egyik lehetséges módszer annak eldöntésére, hogy a p_0 pont egy egyszerű, de nem feltétlenül konvex P poligon belsejében van-e, az, hogy veszünk egy tetsz őleges, p_0-ból kiinduló sugarat, és megnézzük, tejesül-e, hogy a sugár páratlan sokszor metszi P határát, ugyanakkor maga a p_0 pont nincs rajta P határán. Hogyan lehet $\Theta(n)$ id őben meghatározni, hogy a p_0 pont egy n csúcsú P poligon belsejében van-e? (*Útmutatás.* Használjuk fel a 33.1-6. gyakorlat eredményét. Gy őződjünk meg róla, hogy az algoritmus akkor is helyes, ha a sugár a poligon határát egy csúcsban metszi, és akkor is, ha tartalmazza a poligon egy oldalát.)

33.1-8. Hogyan számítható ki egy n csúcsú, egyszerű, de nem feltétlenül konvex poligon területe $\Theta(n)$ időben? (A poligonokra vonatkozó definíciókat lásd a 33.1-5. gyakorlatban.)

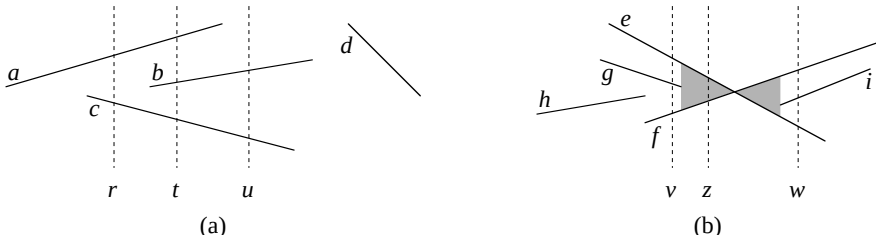
33.2. Metsző szakaszpár létezésének vizsgálata

Ez az alfejezet algoritmust ad annak eldöntésére, hogy egy szakaszhalmazban létezik-e két olyan szakasz, melyek metszik egymást. Az algoritmus használ egy „söprésnek” nevezett technikát, mely sok geometriai algoritmusban el őfordul. Továbbá, mint azt az alfejezet végén található gyakorlatok mutatják, ez az algoritmus, vagy ennek kissé átalakított változatai felhasználhatók más geometriai feladatok megoldására is.

Az algoritmus $O(n \lg n)$ időben fut, ahol n az adott szakaszok száma. Csak azt dönti el, hogy létezik-e metszés, de nem keresi meg az összeset. (A 33.2-1. gyakorlat szerint $\Omega(n^2)$ idő szükséges legrosszabb esetben ahhoz, hogy egy n szakaszból álló halmazban az *összes* metszést megtaláljuk.)

A ***söprés*** során egy képzeletbeli függ őleges ***söprő egyenes*** halad át a geometriai elemek adott halmazán, általában balról jobbra. Azt a térbeli dimenziót, amelyben az egyenes halad, jelen esetben az x-dimenziót, id ődimenziónak tekintjük. A söprés egy módszer a geometriai elemek egyfajta rendezésére, általában azok dinamikus adatstruktúrába szervezésével, és a közöttük fennálló viszony kihasználására. Az ebben az alfejezetben található szakaszmetsző algoritmus balról jobbra veszi az összes szakaszvégpontot, és minden egyes végpont figyelembevételekor megvizsgálja, van-e metszés.

Az n szakasz közötti metsző pár létezésének eldöntésére szolgáló algoritmusunk leírásához és helyességének bizonyításához két egyszerűsít ő feltételt vezetünk be. Egyrészt feltételezzük, hogy az adott szakaszok közül egyik sem függ őleges. Másrészt feltételezzük, hogy az adott szakaszok között nincs három egy pontban metsz ő. A 33.2-8. és a 33.2-9. gyakorlatok kérik annak bizonyítását, hogy az algoritmus eléggé stabil, azaz csak kis módosítás szükséges ahhoz, hogy akkor is működjön, ha ezek a feltételek nem teljesülnek. Gyakran pont az ilyen egyszerűsítő feltételek megszüntetése és a határesetekkel való foglalkozás jelenti a legnehezebb részét a geometriai algoritmusok programozásának és helyességük bizonyításának.



33.4. ábra. A szakaszok rendezése különböző függőleges söprő egyeneseknél. **(a)** Itt $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ és $b >_u c$. A d szakasz nem hasonlítható össze a többi látható szakasz egyikével sem. **(b)** Az e és f szakaszok metszésekor sorrendjük megfordul: $e >_v f$, de $f >_w e$. Minden olyan söprő egyenes (például z) esetén, amely áthalad a szürke tartományon, e és f egymás után következik a teljes rendezésben.

A szakaszok rendezése

Mivel feltételezzük, hogy függőleges szakaszok nincsenek, bármely adott szakasz, amely egy adott függőleges söprő egyenest metsz, egyetlen pontban metszi azt. Így a függőleges söprő egyenest metsző szakaszokat rendezni tudjuk metszéspontjaik y-koordinátái szerint.

Szabatosabban: tekintsünk két szakaszt, s_1-et és s_2-t. Azt mondjuk, hogy ez a két szakasz ***összehasonlítható*** x-nél, ha az x értékű x-koordinátájú függőleges söprő egyenes mindkettőjüket metszi. Azt mondjuk, hogy s_1 ***fölötte*** van s_2-nek x-nél, amit úgy is írunk, hogy $s_1 >_x s_2$, ha s_1 és s_2 összehasonlítható x-nél, és s_1 metszéspontja az x-nél lévő söprő egyenessel magasabban van, mint s_2 metszéspontja ugyanazzal a söprő egyenessel. A 33.4(a) ábrán például a következő relációk állnak fenn: $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ és $b >_u c$. A d szakasz nem hasonlítható össze a többi szakasz közül egyikkel sem.

Bármely adott x értékre a „$>_x$” reláció az x-nél lévő söprő egyenest metsző szakaszok teljes rendezése (lásd B.2. alfejezet). A sorrend különbözhet x különböző értékeire, mivel szakaszok lépnek be a rendezésbe, és lépnek ki onnan. Egy szakasz akkor lép be a rendezésbe, amikor a bal végpontján halad át a söprés, és akkor lép ki a rendezésből, amikor a jobb végpontján halad át.

Mi történik, amikor egy söprő egyenes áthalad két szakasz metszéspontján? Mint a 33.4(b) ábra mutatja, helyzetük a teljes rendezésben felcserélődik. A v és a w egyenesek rendre balra, illetve jobbra vannak az e és f szakaszok metszéspontjától, így $e >_v f$ és $f >_w e$. Mivel feltételezzük, hogy három szakasz nem metszheti egymást ugyanabban a pontban, kell lennie olyan x függőleges söprő egyenesnek, melyre a metsző e és f szakaszok *egymás után következnek* a $>_x$ teljes rendezésben. Bármely söprő egyenes (például z) esetén, amely áthalad a 33.4(b) ábra szürke tartományán, e és f egymás után következik a teljes rendezésben.

A söprő egyenes mozgatása

A söprő algoritmusok jellemzően kétféle adathalmazt kezelnek:

1. A ***söprő egyenes állapotleírása*** megadja az egyenes által metszett elemek helyzetét.
2. Az ***esetpontok jegyzéke*** x-koordináták egy sorozata balról jobbra rendezve, amely a söprő egyenes megállási helyzeteit határozza meg. Minden ilyen megállási helyzetet ***esetpontnak*** nevezünk. A söprő egyenes állapotleírásában változás csak esetpontokban történik.

Néhány algoritmusban (például ilyen a 33.2-7. gyakorlatban szereplő algoritmus), az esetpontok jegyzékét dinamikusan határozzuk meg az algoritmus folyamán. Jelen algoritmus viszont az esetpontokat statikusan határozza meg, kizárólag a bemeneti adatok egyszerű jellemzői alapján. Nevezetesen: minden szakaszvégpont egy esetpont. A szakaszvégpontokat növekvő x-koordináta szerint rendezzük, és balról jobbra haladunk. (Ha két vagy több pont ***kovertikális,*** azaz ugyanaz az x-koordinátájuk, a holtversenyt úgy döntjük el, hogy az összes kovertikális bal végpontot a kovertikális jobb végpontok elé soroljuk be. A kovertikális bal végpontok egy halmazán belül pedig a kisebb y-koordinátájúakat soroljuk be el őbb, és ugyanezt tesszük kovertikális jobb végpontok egy halmazán belül is.) Egy szakaszt akkor illesztünk be a söprő egyenes állapotleírásába, amikor a bal végpontján haladunk át, és akkor töröljük a söprő egyenes állapotleírásából, amikor a jobb végpontján haladunk át. Valahányszor két szakasz először válik közvetlen szomszéddá a teljes rendezésben, megvizsgáljuk, hogy metszik-e egymást.

A söprő egyenes állapotleírása egy T teljes rendezés, amelyhez a következő műveletekre van szükségünk.

- BESZÚR(T, s): beszúrja az s szakaszt T-be.
- TÖRÖL(T, s): törli az s szakaszt T-ből.
- FÖLÖTTI(T, s): a T-ben közvetlenül s fölött lévő szakaszt adja vissza.
- ALATTI(T, s): a T-ben közvetlenül s alatt lévő szakaszt adja vissza.

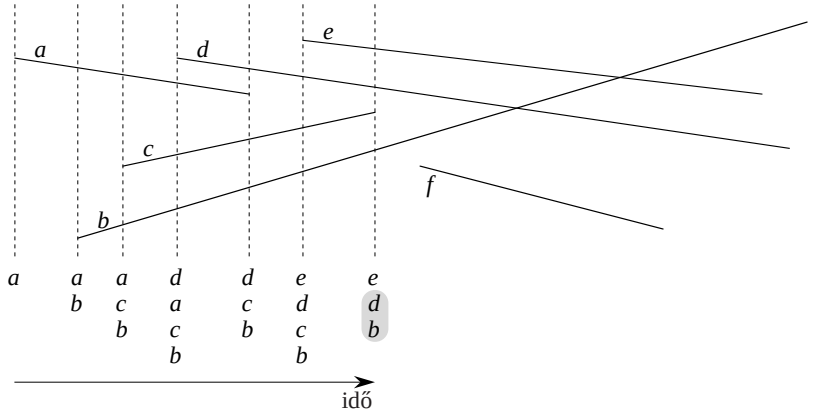
Ha n szakaszunk van, piros-fekete fák alkalmazásával a fenti műveletek mindegyikét $O(\lg n)$ időben tudjuk végrehajtani. Emlékezzünk vissza, hogy a 13. fejezetben a piros-fekete fák műveletei kulcsok összehasonlítását tartalmazzák. Itt a kulcsok összehasonlítását olyan összehasonlításokkal tudjuk helyettesíteni, amelyek keresztszorzatot használnak két szakasz egymáshoz viszonyított helyzetének meghatározására (lásd a 33.2-2. gyakorlatot).

A szakaszmetszés pszeudokódja

A következő algoritmus bemenete egy n szakaszból álló S halmaz, kimenete pedig egy logikai érték, amely IGAZ, ha van metsző szakaszpár az S halmazban, egyébként HAMIS. A T teljes rendezést piros-fekete fa segítségével valósítjuk meg.

A 33.5. ábra szemlélteti az algoritmus végrehajtását. Az 1. sor üresnek inicializálja a teljes rendezést. A 2. sor meghatározza az esetpontok jegyzékét a $2n$ szakaszvégpont balról jobbra rendezésével, és a holtversenyek fent leírtak szerinti eldöntésével. Megjegyezzük, hogy a 2. sor végrehajtható a végpontok (x, e, y) szerinti lexikografikus rendezésével, ahol x és y a szokásos koordináták, és bal végpont esetén $e = 0$, jobb végpont esetén pedig $e = 1$.

A 3–11. sorok **for** ciklusa minden végrehajtása során egy p esetpontot dolgoz fel. Ha p a bal végpontja az s szakasznak, akkor az 5. sor hozzáadja s-et a teljes rendezéshez, továbbá a 6–7. sorok IGAZ-at adnak vissza, ha s metszi valamelyiket azok közül a szakaszok közül, amelyekkel közvetlenül szomszédos a p ponton áthaladó söprő egyenes által definiált teljes rendezésben. (Határeset fordul elő, ha p egy másik s' szakaszon fekszik. Ebben az esetben csak azt követeljük meg, hogy s és s' egymást követően kerüljenek be T-be.) Ha p a jobb végpontja az s szakasznak, akkor s törlendő a teljes rendezésből. A 9–10. sorok IGAZ-at adnak vissza, ha a p ponton áthaladó söprő egyenes által definiált teljes rendezésben s-et közrefogó szakaszok metszik egymást; ezek a szakaszok közvetlenül szomszédossá válnak a teljes rendezésben amikor s törlődik. Ha ezek a szakaszok nem metszik egymást, a 11. sor



33.5. ábra. A VAN-E-METSZŐ-SZAKASZPÁR eljárás végrehajtásának menete. Minden szaggatott vonal a söprő egyenest jelöli egy esetpontban, a rendezett szakasznevek felsorolása a söprő egyenesek alatt pedig maga a T teljes rendezés annak a **for** ciklusnak a végénél, amelyben a megfelelő esetpont kerül feldolgozásra. A d és b szakaszok metszését akkor találja meg, amikor a c szakaszt törli.

törli az s szakaszt a teljes rendezésből. Végül, ha a $2n$ esetpont feldolgozása során egyetlen metszést sem találunk, a 12. sor HAMIS-at ad vissza.

VAN-E-METSZŐ-SZAKASZPÁR(S)

```
 1  T ← ∅
 2  rendezzük az S-beli szakaszok végpontjait balról jobbra, döntsük el a holtversenyt
              a bal végpontok jobb végpontok elé sorolásával, a további holtversenyt
              pedig a kisebb y-koordinátájú pontok nagyobbak elé sorolásával
 3  for minden p pontra a végpontok rendezett listájában
 4      do if p egy s szakasz bal végpontja
 5           then BESZÚR(T, s)
 6                if (FÖLÖTTI(T, s) létezik és metszi s-et) vagy
                     (ALATTI(T, s) létezik és metszi s-et)
 7                then return IGAZ
 8         if p egy s szakasz jobb végpontja
 9           then if FÖLÖTTI(T, s) és ALATTI(T, s) is létezik, és
                     FÖLÖTTI(T, s) metszi ALATTI(T, s)-t
10                then return IGAZ
11                TÖRÖL(T, s)
12  return HAMIS
```

Helyesség

Ahhoz, hogy VAN-E-METSZŐ-SZAKASZPÁR helyességét megmutassuk, be fogjuk bizonyítani, hogy VAN-E-METSZŐ-SZAKASZPÁR(S) akkor és csak akkor ad vissza IGAZ-at, ha van metszés az S-ben lévő szakaszok között.

Könnyen látható, hogy VAN-E-METSZŐ-SZAKASZPÁR csak akkor ad vissza IGAZ-at (a 7. és a 10. sorban), ha metszést talál két megadott szakasz között. Ezért aztán, ha IGAZ-at ad vissza, akkor van metszés.

Meg kell mutatnunk a fordítottját is, azaz, ha van metszés, akkor a VAN-E-METSZŐ-SZAKASZPÁR eljárás IGAZ-at ad vissza. Tételezzük fel, hogy van legalább egy metszés. Legyen p a bal szélső metszéspont, döntse el a holtversenyt az alacsonyabb y-koordináta kiválasztása, és legyenek a és b azok a szakaszok, amelyek p-nél metszik egymást. Mivel p-től balra nem fordul elő metszés, a T által megadott sorrend teljesül minden p-től balra lévő pontra. Mivel semelyik három szakasz sem metszi egymást ugyanabban a pontban, létezik olyan z söprő egyenes, amelynél a és b szomszédossá válik a teljes rendezésben.[2] Ráadásul z vagy p-től balra van, vagy áthalad p-n. Létezik olyan q szakaszvégpont a z söprő egyenesen, amely az az esetpont, ahol a és b szomszédossá válik a teljes rendezésben. Ha p a z söprő egyenesen van, akkor $q = p$. Ha p nincs a z söprő egyenesen, akkor q balra van p-től. Mindkét esetben a T által adott sorrend helyes q figyelembevételét megelőzően. (Ez az a hely, ahol azt a lexikografikus rendezést használjuk, amelyben az algoritmus esetpontokat dolgoz fel. Mivel p még akkor is a legalacsonyabban lévő a bal szélső metszéspontok közül, ha p egy olyan z söprő egyenesen van, amelyen van egy másik p' metszéspont is, ezért a $q = p$ esetpontot azelőtt dolgozzuk fel, mielőtt egy másik, p' metszés megbolygathatná a T teljes rendezést. Sőt akkor is, ha p az egyik szakasz, mondjuk a bal végpontja, és a másik szakasz, mondjuk b jobb végpontja, mivel a bal végponthoz tartozó események a jobb végponthoz tartozók előtt következnek be, a b szakasz T-ben van, amikor az a szakaszt először vesszük figyelembe.) Akár feldolgozta már a q esetpontot a VAN-E-METSZŐ-SZAKASZPÁR eljárás, akár nem.

Ha a VAN-E-METSZŐ-SZAKASZPÁR eljárás feldolgozza q-t, csak két eset lehetséges:

1. Vagy a-t vagy b-t most szúrjuk be T-be, és a másik szakasz vagy fölötte vagy alatta van a teljes rendezésben. A 4–7. sorok észlelik ezt az esetet.
2. Az a és a b szakasz már T-ben van, és most törlünk ki egy szakaszt közülük a teljes rendezésben, ami a-t és b-t közvetlen szomszédokká teszi. A 8–11. sorok észlelik ezt az esetet.

Bármelyik eset áll fenn, a VAN-E-METSZŐ-SZAKASZPÁR eljárás a p metszést megtalálja, és IGAZ-at ad vissza.

Ha VAN-E-METSZŐ-SZAKASZPÁR nem dolgozná fel a q esetpontot, akkor minden bizonnyal az összes esetpont feldolgozása nélkül térne vissza. Ez csak akkor fordulhatna elő, ha VAN-E-METSZŐ-SZAKASZPÁR már korábban talált volna metszést, és IGAZ-at adott volna vissza.

Így ha van metszés, a VAN-E-METSZŐ-SZAKASZPÁR eljárás IGAZ-at ad vissza. Mint már korábban láttuk, ha IGAZ-at ad vissza VAN-E-METSZŐ-SZAKASZPÁR, akkor van metszés. Következésképpen VAN-E-METSZŐ-SZAKASZPÁR mindig helyes választ ad vissza.

A futási idő

Ha n szakasz van az S halmazban, akkor a VAN-E-METSZŐ-SZAKASZPÁR eljárás $O(n \lg n)$ időben fut. Az 1. sor $O(1)$ ideig tart. A 2. sor $O(n \lg n)$ ideig tart, ha összefésülős vagy kupacren-

[2] Ha megengedjük, hogy három szakasz egy pontban metssze egymást, akkor lehet olyan közbülső c szakasz, amely mind a-t, mind b-t metszi a p pontban. Azaz lehetséges $a \prec_w c$ és $c \prec_w b$ minden olyan p-től balra eső w söprő egyenesre, amelyre $a \prec_w b$. A 33.2-8. gyakorlat kéri annak bizonyítását, hogy VAN-E-METSZŐ-SZAKASZPÁR akkor is helyes, ha három szakasz egy pontban metszi egymást.

dezést használ. Mivel $2n$ esetpont van, a **for** ciklus 3–11. sorokban lévő magja legfeljebb $2n$-szer fut. Minden végrehajtása $O(\lg n)$ ideig tart, mert a piros-fekete fa mindegyik művelete $O(\lg n)$ idejű, és mivel a 33.1. alfejezet módszerét felhasználva egy metszés $O(1)$ ideig tart. A teljes futási idő így $O(n \lg n)$.

Gyakorlatok

33.2-1. Mutassuk meg, hogy lehet $\Theta(n^2)$ metszés egy n elemű szakaszhalmazban.

33.2-2. Adott az a és b szakasz, amelyek összehasonlíthatók x-nél. Tételezzük fel, hogy egyik szakasz sem függőleges. Hogyan határozható meg $O(1)$ időben, hogy $a >_x b$ és $b >_x a$ közül melyik teljesül? (*Útmutatás.* Ha a és b nem metszi egymást, elég, ha keresztszorzatokat használunk. Ha a és b metszi egymást – amit persze meghatározhatunk csupán keresztszorzatok alkalmazásával is –, még mindig elég, ha csak összeadást, kivonást és szorzást használunk, elkerülhetjük az osztást. Természetesen a $>_x$ reláció alkalmazása során itt, ha a és b metszi egymást, megállhatunk, és jelenthetjük, hogy metszést találtunk.)

33.2-3. Maginot professzor azt javasolja, módosítsuk a VAN-E-METSZŐ-SZAKASZPÁR eljárást úgy, hogy amikor metszést talál, a visszatérés helyett nyomtassa ki a metsző szakaszokat, és folytassa a **for** ciklust magjának következő végrehajtásával. A professzor az így kapott eljárást METSZŐ-SZAKASZOKAT-NYOMTAT eljárásnak hívja, és azt állítja, hogy kinyomtatja az összes metszést, balról jobbra, ahogyan a szakaszhalmazban előfordulnak. Mutassuk meg, hogy a professzornak nincs igaza. Adjunk meg egy olyan szakaszhalmazt, amelyre a METSZŐ-SZAKASZOKAT-NYOMTAT eljárás nem a bal szélső metszést találja meg először. Majd adjunk meg egy másik szakaszhalmazt, amelyre METSZŐ-SZAKASZOKAT-NYOMTAT nem tudja megtalálni az összes metszést.

33.2-4. Adjunk meg egy $O(n \lg n)$ futási idejű algoritmust annak eldöntésére, hogy egy n csúcsú poligon egyszerű-e vagy sem.

33.2-5. Adjunk meg egy $O(n \lg n)$ futási idejű algoritmust annak eldöntésére, hogy két egyszerű poligon, melyeknek összesen n éle van, metszi-e egymást.

33.2-6. A ***körlemez*** a körből meg a kör belsejéből áll, és középpontjával, valamint sugarával írjuk le. Két körlemez metszi egymást, ha van közös pontjuk. Adjunk egy $O(n \lg n)$ futási idejű algoritmust annak eldöntésére, hogy van-e a körlemezek egy n elemű halmazában kettő, melyek metszik egymást.

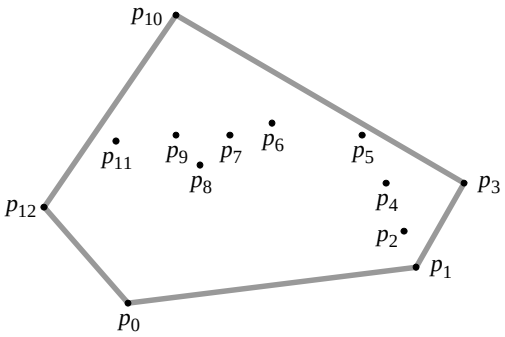
33.2-7. Adott egy n szakaszból álló halmaz, amely összesen k metszést tartalmaz. Hogyan lehet megkeresni mind a k metszést $O((n + k) \lg n)$ időben?

33.2-8. Bizonyítsuk be, hogy a VAN-E-METSZŐ-SZAKASZPÁR eljárás helyesen működik akkor is, ha három vagy több szakasz ugyanabban a pontban metszi egymást.

33.2-9. Mutassuk meg, hogy VAN-E-METSZŐ-SZAKASZPÁR helyesen működik függőleges szakaszok megengedése esetén is, ha minden függőleges szakasz alsó végpontját úgy dolgozzuk fel, mintha bal végpont, a felső végpontját pedig, mintha jobb végpont volna. Hogyan változik a 33.2-2. gyakorlatra adott válasza, ha függőleges szakaszokat is megengedünk?

33.3. Ponthalmaz konvex burka

Egy Q ponthalmaz ***konvex burka*** az a legkisebb P konvex poligon, amelyre Q minden pontja vagy P határán van, vagy a belsejében. (A konvex poligon pontos definíciójához lásd a 33.1-5. gyakorlatot.) Q konvex burkát $\mathrm{CH}(Q)$-val jelöljük. Szemléletesen Q pontjait táblából



33.6. ábra. Egy $Q = \{p_0, p_1, \ldots, p_{12}\}$ ponthalmaz, és szürkével az ő CH(Q) konvex burka.

kiálló szegeknek képzelhetjük. Ekkor a konvex burok annak a szoros gumiszalagnak az alakja, amely az összes szeget körülfogja. A 33.6 ábra egy ponthalmazt és az ő konvex burkát mutatja.

Ebben az alfejezetben két algoritmust fogunk bemutatni, amelyek n elemű ponthalmaz konvex burkát határozzák meg. Mindkét algoritmus a konvex burok csúcsait adja vissza az óramutató járásával ellentétes sorrendben. Az első, amely Graham-féle pásztázásként ismert, $O(n \lg n)$ időben fut. A második, melyet Jarvis menetelésének hívunk, $O(nh)$ időben fut, ahol h a konvex burok csúcsainak száma. Mint azt a 33.6. ábrán láthatjuk, CH(Q) minden csúcsa a Q halmaz egy pontja. Mindkét algoritmus kihasználja ezt a tulajdonságot, és csak azt dönti el, hogy mely Q-beli csúcsokat tartsa meg a konvex burok csúcsaként, és mely Q-beli csúcsokat dobja el.

Több olyan algoritmus is ismeretes, melyek $O(n \lg n)$ időben tudnak konvex burkot számolni. Mind a Graham-féle pásztázás, mind Jarvis menetelése használja a „forgatásos söprés” technikát, amely a csúcsokat egy vonatkoztatási csúcs körüli poláris szög szerinti sorrendben dolgozza fel. További módszerek például a következők.

- A ***növekményes*** módszerben a pontokat balról jobbra rendezzük, így egy $\langle p_1, p_2, \ldots, p_n \rangle$ sorozatot kapunk. Az i-edik lépésben CH($\{p_1, p_2, \ldots, p_{i-1}\}$)-et, az $i-1$ bal szélső pont konvex burkát a balról i-edik pontnak megfelelően aktualizáljuk, így CH($\{p_1, p_2, \ldots, p_i\}$)-t képezzük. Ezt a módszert $O(n \lg n)$ időben meg lehet valósítani. A bizonyítást a 33.3-6. gyakorlat kéri.
- Az ***oszd-meg-és-uralkodj*** elvű módszerben egy n pontból álló halmazt $\Theta(n)$ időben két részhalmazra osztunk úgy, hogy az egyik a bal szélső $\lceil n/2 \rceil$ pontot, a másik a jobb szélső $\lfloor n/2 \rfloor$ pontot tartalmazza, ezzel a részhalmazok konvex burkait rekurzívan kiszámítjuk, és egy ügyes módszerrel egyesítjük a konvex burkokat $O(n)$ időben. A futási időt az ismerős $T(n) = 2T(n/2) + O(n)$ képlet jellemzi, így az „oszd-meg-és-uralkodj” módszer $O(n \lg n)$ időben fut.
- Az ***eltávolító és kereső*** módszer hasonló a 9.3. alfejezet legrosszabb esetben lineáris futási idejű, mediánt kereső algoritmusához. Megkeresi a felső részét (vagy „felső láncát”) a konvex buroknak oly módon, hogy ismételten kidobja egy állandó hányadát a megmaradó pontoknak mindaddig, míg csak a konvex burok felső lánca marad meg. Aztán ugyanezt csinálja az alsó láncra. Aszimptotikusan ez a módszer a leggyorsabb: ha a konvex burok h csúcsot tartalmaz, futási ideje csak $O(n \lg h)$.

Ponthalmaz konvex burkának kiszámítása önmagában is érdekes probléma. Sok más geometriai algoritmus konvex burok kiszámításával indul. Tekintsük például a kétdimenziós ***legtávolabbi pár feladatot:*** adott egy n pontból álló halmaz a síkon, keressük azt a két pontot, melyek távolsága egymástól a legnagyobb. Ez a két pont biztosan csúcsa a konvex buroknak: a bizonyítást a 33.3-3. gyakorlat kéri. Bár ezt az állítást nem bizonyítjuk, egy n csúcsú konvex poligon legtávolabbi csúcsait $O(n)$ időben meg lehet találni. Ezért, ha n adott pontnak először meghatározzuk a konvex burkát $O(n \lg n)$ időben, majd megkeressük az így kapott konvex poligon csúcsai között a legtávolabbi párt, tetszőleges n pontból álló halmazban meg tudjuk keresni a legtávolabbi párt $O(n \lg n)$ időben.

A Graham-féle pásztázás

A ***Graham-féle pásztázás*** a konvex burok meghatározását oldja meg a jelölt pontokat tartalmazó S verem segítségével. Az adott Q halmaz minden pontját beírjuk egyszer a verembe, majd azokat a pontokat, amelyek nem csúcsai CH(Q)-nak, előbb vagy utóbb kivesszük a veremből. Amikor az algoritmus véget ér, S pontosan CH(Q) csúcsait tartalmazza a határpoligonban az óramutató járásával ellenkező irányban való megjelenésük sorrendjében.

A GRAHAM-PÁSZTÁZÁS eljárás bemenete egy Q ponthalmaz, melyre $|Q| \geq 3$. Hívja a LEGFELSŐ függvényt, amely visszaadja az S verem legfelső pontját S megváltoztatása nélkül, és a LEGFELSŐ-ALATTI függvényt, amely visszaadja az S verem legfelső eleme alatt eggyel lévő pontot S megváltoztatása nélkül. Amint azt hamarosan bebizonyítjuk, a GRAHAM-PÁSZTÁZÁS által visszaadott S verem pontosan CH(Q) csúcsait tartalmazza felülről lefelé az óramutató járásával ellenkező irányban.

GRAHAM-PÁSZTÁZÁS(Q)

```
 1  legyen p_0 a minimális y-koordinátájú Q-beli pont,
            vagy egyezés esetén a bal szélső ilyen pont
 2  legyen ⟨p_1, p_2, ..., p_m⟩ a többi Q-beli pont, p_0 körül poláris szög szerint az
            óramutató járásával ellenkező sorrendben (ha több mint egy pontnak
            ugyanaz a szöge, távolítsuk el mindet, a p_0-tól legtávolabbi kivételével)
 3  VEREMBE(p_0, S)
 4  VEREMBE(p_1, S)
 5  VEREMBE(p_2, S)
 6  for i ← 3 to m
 7      do while a LEGFELSŐ-ALATTI(S), LEGFELSŐ(S) és p_i pontok
                szöge nem fordul balra
 8                do VEREMBŐL(S)
 9         VEREMBE(p_i, S)
10  return S
```

A 33.7. ábra a GRAHAM-PÁSZTÁZÁS-t folyamatában szemlélteti. Az 1. sor kiválasztja a legkisebb y-koordinátájú pontot, p_0-t, egyezés esetén az ilyenek közül a bal szélsőt. Mivel nincs olyan pont Q-ban, amely p_0 alatt lenne, és ha van ugyanilyen y koordinátájú pont, akkor az tőle jobbra van, p_0 csúcsa CH(Q)-nak. A 2. sor Q többi pontját p_0 körüli poláris szög szerint rendezi, ugyanazzal a módszerrel – keresztszorzatok összehasonlításával –, mint a 33.1-3. gyakorlat. Ha két vagy több pontnak ugyanaz a poláris szöge p_0 körül, akkor a legtávolabbi pont kivételével mindegyik konvex kombinációja p_0-nak és a legtávolabbi pontnak,

így csak a legtávolabbit vesszük közülük figyelembe a továbbiakban. Jelöljük m-mel a p_0-tól különböző, megmaradó pontok számát. Q minden pontjának radiánban mért p_0 körüli poláris szöge a $[0, \pi)$ félig nyílt intervallumba esik. Mivel a pontok a poláris szögeknek megfelelően vannak rendezve, sorrendjük p_0 körül az óramutató járásával ellentétes. Jelöljük ezt a rendezett pontsorozatot $\langle p_1, p_2, \ldots, p_m \rangle$-mel. Figyeljük meg, hogy a p_1 és a p_m pont csúcsa CH(Q)-nak (lásd a 33.3-1. gyakorlatot). A 33.7(a) ábrán a 33.6. ábra pontjait láthatjuk, p_0 körüli növekvő poláris szög szerint sorszámozva.

Az eljárás hátralévő része használja az S vermet. A 3–5. sorok beállítják a verem kezdeti tartalmát az első három pontra, alulról felfelé p_0-ra, p_1-re és p_2-re. A 33.7(a) ábra mutatja az S verem kezdeti állapotát. A 6–9. sorokban lévő **for** ciklus a $\langle p_3, p_4, \ldots, p_m \rangle$ részsorozat minden elemére lefut egyszer. Az a szándékunk, hogy a p_i pont feldolgozása után az S verem CH($\{p_0, p_1, \ldots, p_i\}$) csúcsait tartalmazza alulról felfelé, az óramutató járásával ellenkező irányban. A 7–8. sorok **while** ciklusa távolítja el a pontokat a veremből, ha az derül ki róluk, hogy nem csúcsai a konvex buroknak. Amikor a konvex burkot az óramutató járásával ellentétes irányban járjuk be, minden csúcsában balra kell fordulnunk. Ezért minden alkalommal, amikor a **while** ciklus olyan csúcsot talál, melynél nem fordulunk balra, kivesszük az illető csúcsot a veremből. (Azzal, hogy a „balra nem fordulást” vizsgáljuk a jobbra fordulás helyett, kizárjuk az egyenesszög lehetőségét az eredményül kapott konvex burok csúcsainál. Nem akarunk egyenesszöget, mivel a konvex poligon csúcsa nem állhat elő a poligon többi csúcsának konvex kombinációjaként.) Miután minden olyan csúcsot kivettünk, amelyeknél p_i felé haladva nem fordulunk balra, betesszük p_i-t a verembe. A 33.7(b)–(k) ábrák az S verem állapotát mutatják a **for** ciklus magjának minden egyes végrehajtása után. Végül a GRAHAM-PÁSZTÁZÁS az S vermet adja vissza a 10. sorban. A 33.7(l) ábra mutatja a megfelelő konvex burkot.

A következő tétel formálisan bizonyítja a GRAHAM-PÁSZTÁZÁS helyességét.

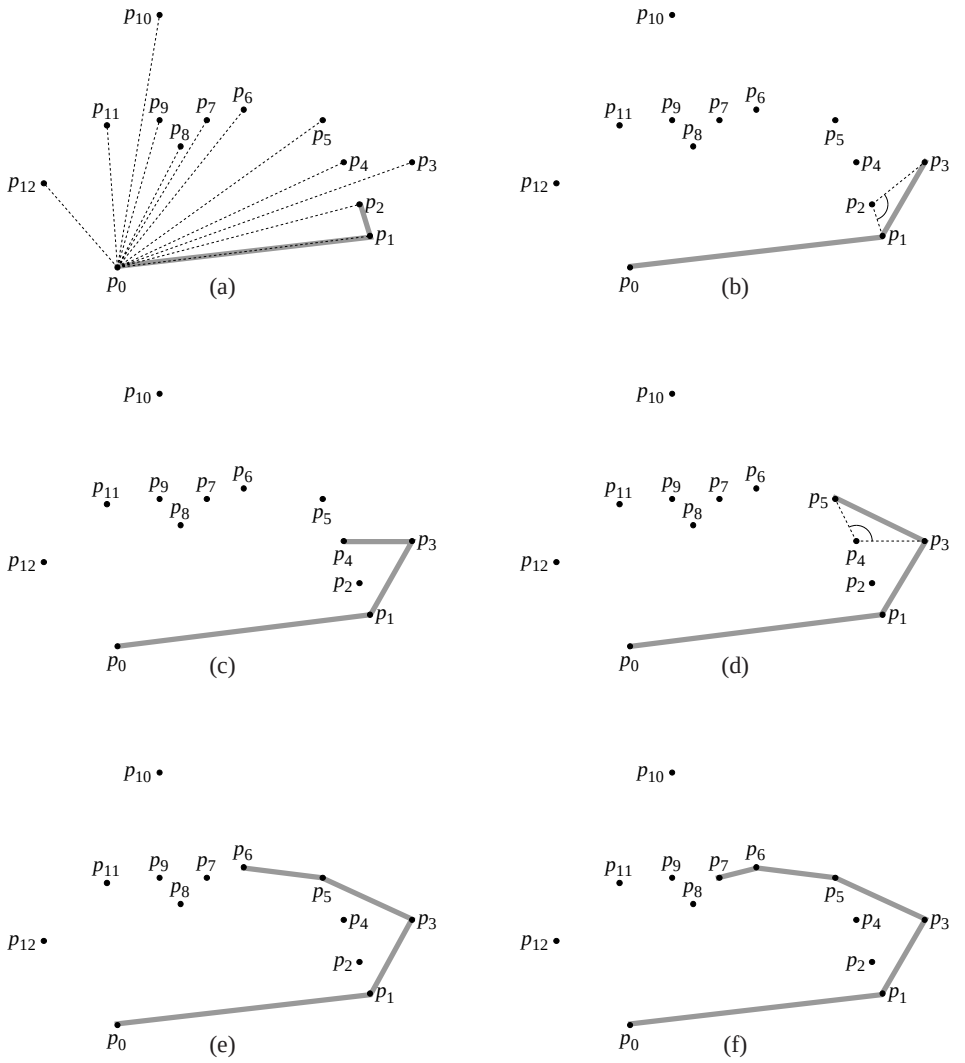
33.1. tétel (a Graham-pásztázás helyessége). *Ha a* GRAHAM-PÁSZTÁZÁS *eljárást egy olyan Q ponthalmazra futtatjuk, melyre $|Q| \geq 3$, akkor befejezésekor az S verem pontosan CH(Q) csúcsaiból áll, alulról felfelé az óramutató járásával ellentétes sorrendben.*

Bizonyítás. A 2. sor után van egy $\langle p_1, p_2, \ldots, p_m \rangle$ pontsorozatunk. Definiáljuk $i = 2, 3, \ldots, m$ esetén a pontok $Q_i = \{p_0, p_1, \ldots, p_i\}$ részhalmazát. A $Q - Q_m$ halmaz pontjai azok, amelyek el lettek távolítva, mert p_0 körül ugyanaz volt a poláris szögük, mint valamely más ponté Q_m-ben. Ezek a pontok nincsenek benne CH(Q)-ban, így CH(Q_m) = CH(Q). Ezért elég azt megmutatni, hogy ha a GRAHAM-PÁSZTÁZÁS befejeződik, akkor az S verem CH(Q_m) csúcsaiból áll, az óramutató járásával ellentétes sorrendben, alulról felfelé. Vegyük észre, hogy ugyanúgy, ahogy p_0, p_1 és p_m csúcsai CH(Q)-nak, a p_0, p_1 és p_i pontok is mind csúcsai CH(Q_i)-nek.

A bizonyításhoz felhasználjuk a következő ciklusinvariánst:

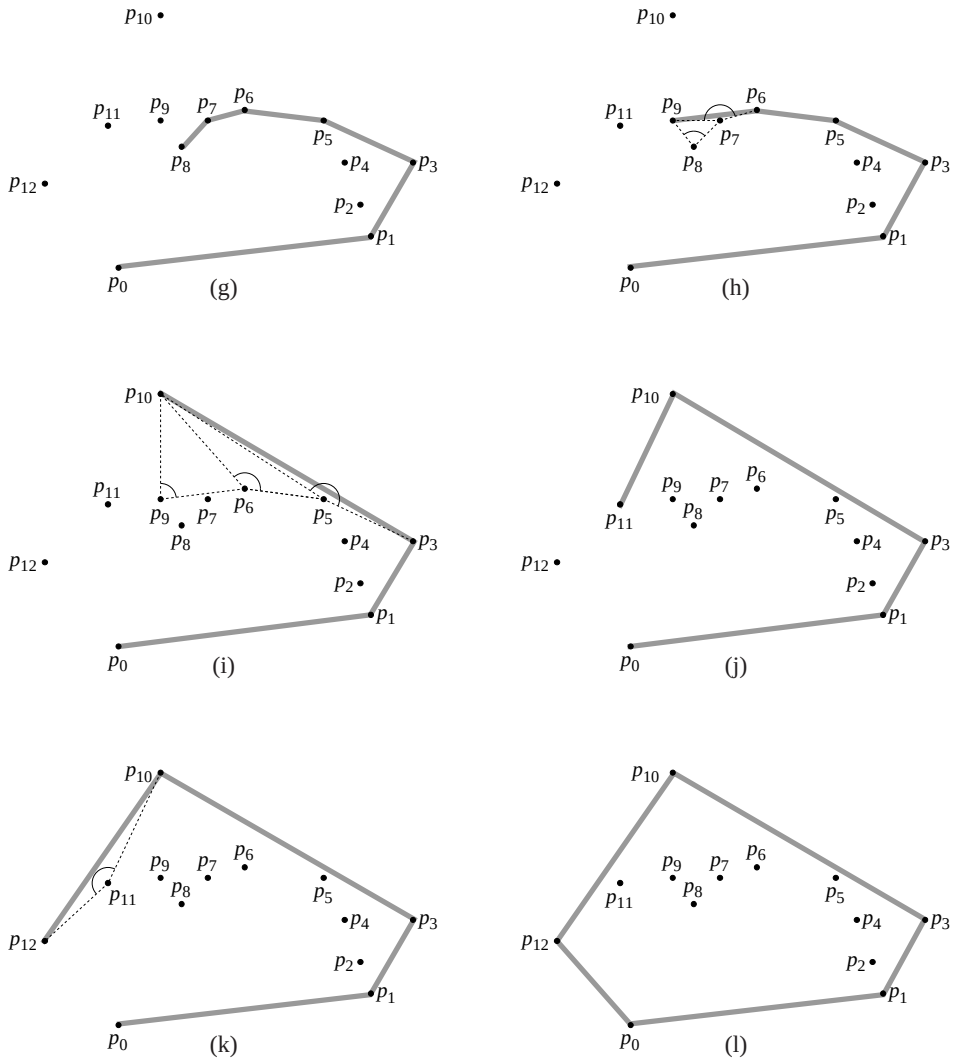
> A 6–9. sorban lévő **for** ciklus magjának minden végrehajtása elején az S verem pontosan CH(Q_{i-1}) csúcsaiból áll, alulról fölfelé az óramutató járásával ellentétes irányban.

Teljesül: Az invariáns fennáll, amikor először hajtjuk végre a 6. sort, mert akkor az S verem pontosan a $Q_2 = Q_{i-1}$ csúcsokból áll, és az ebből a három csúcsból álló halmaz alkotja saját konvex burkát. Ráadásul ezek az óramutató járásával ellentétes sorrendben vannak alulról fölfelé.



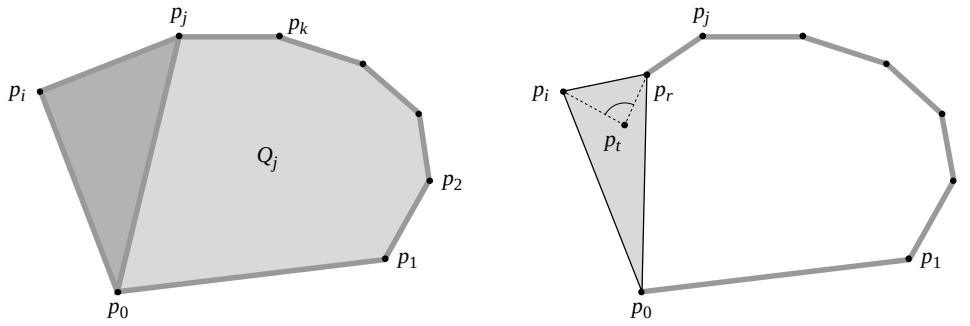
33.7. ábra. A GRAHAM-PÁSZTÁZÁS végrehajtása a 33.6. ábra Q halmazára. Az egyes lépésekben pillanatnyilag az S veremben lévő konvex burkot szürkével jelöljük. **(a)** A $\langle p_1, p_2, \ldots, p_{12} \rangle$, növekvő p_0 körüli poláris szög szerint rendezett pontsorozat és a kezdeti S verem, amely p_0-t, p_1-et és p_2-t tartalmazza. **(b)–(k)** Az S verem a 6–9. sorban lévő **for** ciklus magjának minden egyes végrehajtása után. Szaggatott vonalak jelölik azokat a nem balra fordulásokat, amelyek miatt pontokat veszünk ki a veremből. Az ábra (h) részén például a $\angle p_7 p_8 p_9$ szögnél lévő jobbra fordulás miatt vesszük ki p_8-at, majd a $\angle p_6 p_7 p_9$ szögnél lévő jobbra fordulás miatt vesszük ki p_7-et. **(l)** Az eljárás által visszaadott konvex burok, amely megegyezik a 33.6. ábrán láthatóval.

Megmarad: A **for** ciklus magjába való belépéskor az S veremben a legfelső pont p_{i-1}, amelyet az előző végrehajtás végén tettünk be (vagy az első végrehajtás előtt, $i = 3$ esetén). Legyen p_j az a pont, amelyik a legfelső S-ben a 7–8. sorokban lévő **while** ciklus végrehajtása után, de még mielőtt a 9. sor beteszi p_i-t, és legyen p_k az S-ben közvetlenül p_j alatt lévő pont. Abban a pillanatban, amikor p_j a legfelső pont S-ben, és még nem tettük bele p_i-t, az S verem pontosan ugyanazokat a pontokat tartalmazza,



amelyek a **for** ciklus j-re való végrehajtása után benne voltak. A ciklusinvariánsnak megfelelően ezért S pontosan CH(Q_j) csúcsait tartalmazza ekkor, és ezek éppen az óramutató járásával ellentétes sorrendben vannak alulról fölfelé.

Összpontosítsunk ismét erre a p_i verembe tétele előtti pillanatra. A 33.8(a) ábrára vonatkozóan, mivel p_i poláris szöge p_0 körül nagyobb, mint p_j poláris szöge, és mivel a $\angle p_k p_j p_i$ szög balra fordul (egyébként p_j-t kivettük volna), látjuk, hogy mivel S pontosan CH(Q_j) csúcsait tartalmazza, ha beszúrjuk p_i-t, akkor az S verem pontosan CH($Q_j \cup \{p_i\}$)-t fogja tartalmazni, továbbra is az óramutató járásával ellentétes irányban alulról fölfelé.



33.8. ábra. (a) Mivel p_i poláris szöge p_0 körül nagyobb, mint p_j poláris szöge, és mivel a $\angle p_k p_j p_i$ szög balra fordul, p_i-nek CH(Q_j)-hez való hozzáadásával pontosan CH($Q_j \cup \{p_i\}$) csúcsait kapjuk meg. **(b)** Ha a $\angle p_r p_t p_i$ szög nem fordul balra, akkor p_t vagy a p_0, p_r és p_i által alkotott háromszög belsejében van, vagy a háromszög egyik oldalán, így nem lehet CH(Q_t) csúcsa.

Most megmutatjuk, hogy CH($Q_j \cup \{p_i\}$) ugyanaz a ponthalmaz, mint CH(Q_i). Tekintsünk egy p_t pontot, amelyet a **for** ciklus magjának i-re való végrehajtása során vettünk ki, és legyen p_r a közvetlenül p_t alatti pont az S veremben (p_t kivételekor (p_r akár p_j is lehet). A $\angle p_r p_t p_i$ szög nem fordul balra, és p_t poláris szöge p_0 körül nagyobb, mint p_r poláris szöge. Mint azt a 33.8(b) ábra mutatja, p_t-nek vagy a p_0, p_r és p_i által alkotott háromszög belsejében kell lennie, vagy e háromszög valamelyik oldalán (de nem lehet csúcsa a háromszögnek). Mivel p_t a belsejében van egy Q_i három másik pontja által alkotott háromszögnek, nem lehet csúcsa CH(Q_i)-nek. Mivel p_t nem csúcsa CH(Q_i)-nek,

$$\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i). \tag{33.1}$$

Legyen P_i azon pontok halmaza, melyeket a **for** ciklus magjának i-re való végrehajtása során vettünk ki. Mivel a (33.1) egyenlőség minden P_i-beli pontra fennáll, alkalmazhatjuk ismételten annak bizonyítására, hogy CH($Q_i - P_i$) = CH(Q_i). De $Q_i - P_i = Q_j \cup \{p_i\}$, így CH($Q_j \cup \{p_i\}$) = CH($Q_i - P_i$) = CH($Q_i$) következik.

Megmutattuk, hogy amint p_i-t beletesszük, az S verem pontosan CH(Q_i) csúcsait tartalmazza alulról fölfelé az óramutató járásával ellentétes irányban. A ciklusinvariáns így, i növelésével, a ciklusmag következő végrehajtására is fenn fog állni.

Befejeződik: Amikor a ciklus befejeződik, $i = m + 1$, így a ciklusinvariánsból következik, hogy az S verem pontosan CH(Q_m) csúcsaiból áll, amely most CH(Q), alulról fölfelé az óramutató járásával ellentétes irányban. Ezzel bizonyításunk teljes. ■

Most megmutatjuk, hogy a GRAHAM-PÁSZTÁZÁS futási ideje $O(n \lg n)$, ahol $n = |Q|$. Az 1. sor $\Theta(n)$ ideig tart. A 2. sor $O(n \lg n)$ futási idejű, ha a poláris szögek rendezésére az összefésülő rendezést vagy a kupacrendezést használjuk, a szögek összehasonlítására pedig a 33.1. alfejezet keresztszorzatos módszerét. (Az azonos poláris szögű pontok közül a legtávolabbi megtartása és a többi eltávolítása $O(n)$ időben elvégezhető.) A 3–5. sorok végrehajtása $O(1)$ ideig tart. Mivel $m \leq n-1$, a 6–9. sorok **for** ciklusának magját legfeljebb $n-3$ alkalommal hajtjuk végre. A VEREMBE eljárás $O(1)$ ideig tart, ezért a ciklusmag mindegyik végrehajtása $O(1)$ ideig tart, kivéve azt az időt, amit a 7–8. sorokban lévő **while** ciklusban töltünk, így az egész **for** ciklus $O(n)$ ideig tart, kivéve a beágyazott **while** ciklust.

Az összesítési módszert használjuk annak bizonyítására, hogy a **while** ciklus összesen $O(n)$ ideig tart. Minden p_i pontot, $i = 0, 1, \ldots, m$, pontosan egyszer teszünk be a verembe. Megfigyeljük, hogy most is, mint a 17.1. alfejezet TÖBBSZÖRÖS-VEREMBŐL eljárásának elemzése során, minden egyes VEREMBE műveletre legfeljebb egy VEREMBŐL művelet jut. Legalább három pontot – p_0-t, p_1-et és p_m-et – sohasem veszünk ki a veremből, így valójában legfeljebb $m-2$ alkalommal végzünk VEREMBŐL műveletet. A **while** ciklus magjának minden végrehajtása egy VEREMBŐL műveletet jelent, így ez a ciklusmag legfeljebb $m-2$ alkalommal hajtódik végre. Mivel a 7. sorban lévő vizsgálat $O(1)$ ideig tart, mivel a VEREMBŐL művelet minden hívása $O(1)$ ideig tart, és mivel $m \leq n-1$, a **while** ciklus összes futási ideje $O(n)$. Így a GRAHAM-PÁSZTÁZÁS futási ideje $O(n \lg n)$.

Jarvis menetelése

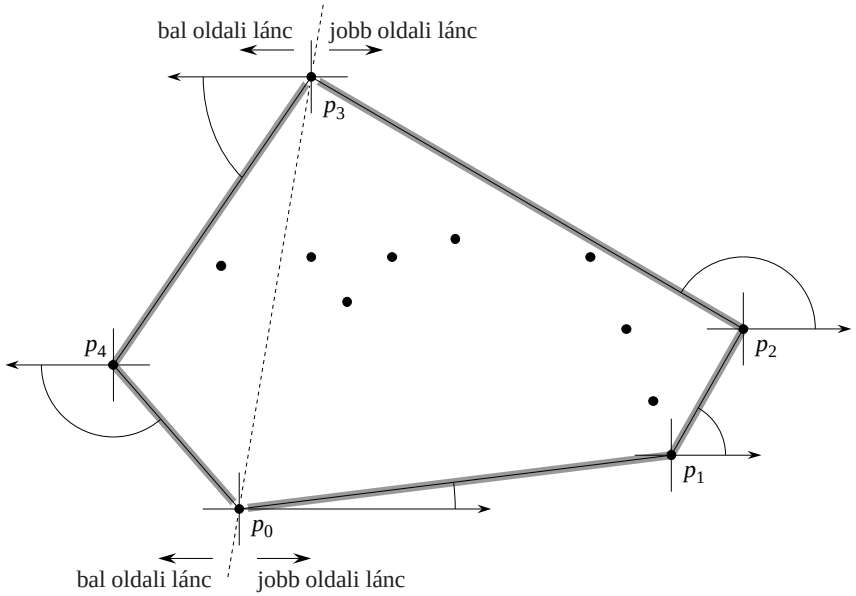
Jarvis menetelése egy Q ponthalmaz konvex burkát határozza meg a ***csomagolás elve*** (vagy ***ajándékcsomagolás elve***) néven ismert módszer segítségével. Az algoritmus $O(nh)$ időben fut, ahol $\mathrm{CH}(Q)$ csúcsainak száma h. Ha h értéke $o(\lg n)$, Jarvis menetelése aszimptotikusan gyorsabban fut, mint a Graham-féle pásztázás.

Szemléletesen Jarvis menetelése olyan, mintha a Q halmazt egy feszesen tartott papírral akarnánk becsomagolni. Először is odaragasztjuk a papír végét a halmaz legalsó pontjához, vagyis ugyanahhoz a p_0 ponthoz, amellyel a Graham-féle pásztázást indítjuk. Ez a pont a konvex burok egy csúcsa. Jobbra húzva feszesen tartjuk a papírt, majd feljebb húzzuk, míg el nem érünk vele egy pontot. Ennek a pontnak szintén a konvex burok csúcsának kell lennie. A papírt feszesen tartva addig haladunk tovább ily módon a csúcsok halmaza körül, míg vissza nem érünk a kiindulási p_0 pontba.

Formálisan azt mondhatjuk, hogy Jarvis menetelése $\mathrm{CH}(Q)$ csúcsaiból egy $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ sorozatot alkot. A bejárást p_0-lal kezdjük. Mint azt a 33.9. ábra mutatja, a konvex burok következő p_1 csúcsa az a pont, amelynek a legkisebb a p_0 körüli poláris szöge. (Holtverseny esetén a p_0-tól legtávolabbi pontot választjuk.) Hasonlóan p_2 a p_1 körül legkisebb poláris szögű pont és így tovább. Mire elérjük a legfölső csúcsot, mondjuk p_k-t (a holtversenyeket a legtávolabbi csúcs választásával döntjük el), meg is szerkesztjük a $\mathrm{CH}(Q)$ konvex burok ***jobb oldali láncát,*** mint azt a 33.9. ábra mutatja. A ***bal oldali lánc*** megszerkesztéséhez induljunk p_k-ból, és válasszuk p_{k+1}-nek azt a pontot, amelynek a *negatív x-tengelytől mérve* legkisebb a poláris szöge p_k körül. Ezt folytatjuk, a poláris szögeket a mindig negatív x-tengelytől véve, egészen addig, míg vissza nem érünk a kiindulási p_0 csúcsba, és így kialakítjuk a bal oldali láncot.

Jarvis menetelését a konvex burok körül egyetlen fordulóban is megírhatnánk, azaz a jobb és a bal lánc külön létrehozása nélkül. Az ilyen megvalósítások jellegzetessége, hogy a konvex burok utoljára kiválasztott oldalának szögét tartják nyilván, és azt követelik meg, hogy az oldalak szöge szigorúan monoton növekedjen (a $[0, 2\pi)$ tartományban, radiánban). Két külön láncot szerkeszteni azért előnyös, mert így nem kell egyetlen szöget sem kiszámítanunk, a 33.1. alfejezet módszerei elegendőek a szögek összehasonlításához.

Ha jól valósítjuk meg, Jarvis menetelése $O(nh)$ futási idejű. A $\mathrm{CH}(Q)$ minden h csúcsára megkeressük a legkisebb poláris szögű csúcsot. Ha a 33.1. alfejezet módszereit használjuk, a poláris szögek közötti minden összehasonlítás $O(1)$ ideig tart. A 9.1. alfejezet szerint n érték minimumát $O(n)$ időben tudjuk megkeresni, ha minden összehasonlítás $O(1)$ idejű. Így Jarvis menetelése $O(nh)$ ideig tart.



33.9. ábra. Jarvis menetelése működés közben. Az első csúcs, p_0, a legalsó pont. A következő csúcs, p_1, az a pont, amelynek a legkisebb a poláris szöge p_0 körül. Majd p_2-nek a legkisebb a poláris szöge p_1 körül. A jobb oldali lánc addig tart, míg el nem éri a legfölső pontot, p_3-at. Ezután a bal oldali láncot szerkesztjük meg, a negatív x-tengelytől mért legkisebb poláris szögek kiválasztásával.

Gyakorlatok

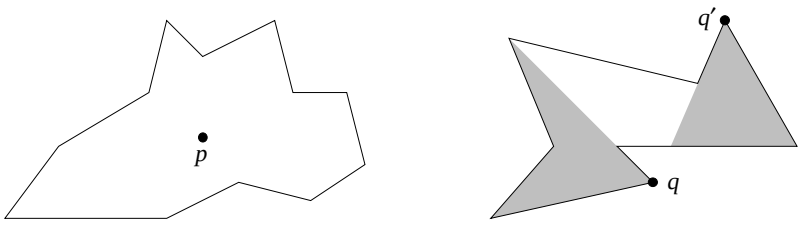
33.3-1. Bizonyítsuk be, hogy a GRAHAM-PÁSZTÁZÁS eljárásban a p_1 és p_m pontok biztosan csúcsai CH(Q)-nak.

33.3-2. Tekintsünk egy olyan számítási modellt, amely támogatja az összeadás, összehasonlítás és szorzás műveleteket, és amellyel az n szám rendezéséhez szükséges időre $\Omega(n \lg n)$ alsó korlát. Bizonyítsuk be, hogy egy ilyen modellben $\Omega(n \lg n)$ nagyságrendjét tekintve alsó korlátja az n pontból álló halmaz konvex burkának meghatározásához szükséges időnek is.

33.3-3. Adott egy Q ponthalmaz. Bizonyítsuk be, hogy Q egymástól legtávolabbi két pontjának CH(Q) csúcsának kell lennie.

33.3-4. Adott egy P poligon és a határán egy q pont. Azon r pontok halmazát, melyekre a $\overline{qr}$ szakasz teljes egészében P belsejében vagy határán van, q ***árnyékának*** nevezzük. Egy poligon ***csillag alakú,*** ha létezik olyan p pont P belsejében, amely a P határán lévő minden pont árnyékában benne van. Az összes ilyen p pont halmazát P ***magjának*** hívjuk. (Lásd a 33.10. ábrát.) Adott egy n csúcsú, csillag alakú P poligon az óramutató járásával ellentétes irányban felsorolt csúcsaival. Hogyan lehet $O(n)$ időben CH(P)-t meghatározni?

33.3-5. Az ***on-line konvex burok feladatban*** egyenként kapjuk meg a Q ponthalmaz n pontját. Amint megkapunk egy újabb pontot, meg akarjuk határozni a már ismert pontok konvex burkát. Magától értetődő, hogy lefuttathatnánk a Graham-féle pásztázást minden újabb pont után, de ekkor a teljes futási idő $O(n^2 \lg n)$ volna. Hogyan lehet megoldani az on-line konvex burok feladatot $O(n^2)$ futási időben?



33.10. ábra. A csillag alakú poligon definíciója a 33.3-4. gyakorlatban. **(a)** Egy csillag alakú poligon. Minden olyan szakasz, amely a p pontot egy q határponttal köti össze, a poligon határát csak q-ban metszi. **(b)** Egy olyan poligon, amely nem csillag alakú. A bal oldali árnyalt tartomány q árnyéka, a jobb oldali pedig q' árnyéka. Mivel ezek a tartományok diszjunktak, a poligon magja üres.

33.3-6. ⋆ Hogyan lehet megvalósítani az n pont konvex burkának meghatározására szolgáló növekményes módszert úgy, hogy $O(n \lg n)$ időben fusson?

33.4. Az egymáshoz legközelebbi két pont megkeresése

Most azzal a feladattal foglalkozunk, hogyan lehet megkeresni az egymáshoz legközelebbi két pontot egy $n \geq 2$ pontból álló Q halmazban. Az, hogy „legközelebbi” a szokásos ***euklideszi távolságra*** vonatkozik, mely szerint a $p_1 = (x_1, y_1)$ és $p_2 = (x_2, y_2)$ pontok közötti távolság $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. A Q halmaz két pontja egybe is eshet, ebben az esetben a közöttük lévő távolság nulla. E feladat alkalmazásai például forgalomirányító rendszerekben fordulnak elő. Egy légi vagy tengeri közlekedést irányító rendszernek a lehetséges ütközések felismerése céljából tudnia kell, hogy melyik az a két jármű, amelyek legközelebb vannak egymáshoz.

Egy egyszerű, „nyers erőn” alapuló, legközelebbi pontokat kereső algoritmus megnézheti például mind az $\binom{n}{2} = \Theta(n^2)$ pontpárt. Ebben az alfejezetben leírunk egy „oszd-meg-és-uralkodj” elvű algoritmust erre a feladatra, melynek futási idejét a jól ismert $T(n) = 2T(n/2) + O(n)$ rekurziós formula adja meg. Így ennek az algoritmusnak a futásához csak $O(n \lg n)$ idő szükséges.

Az oszd-meg-és-uralkodj elvű algoritmus

Az algoritmus minden rekurzív hívásának egy $P \subseteq Q$ részhalmaz, valamint az X és Y tömb a bemenete, melyek külön-külön tartalmazzák a P részhalmaz összes elemét. Az X tömbben lévő pontok x-koordinátájuk szerint monoton növekvő sorrendben vannak rendezve. Hasonlóképpen az Y tömb monoton növekvő y-koordináta szerint van rendezve. Megjegyezzük, hogy az $O(n \lg n)$ időkorlát elérése érdekében nem engedhetjük meg magunknak, hogy minden rekurzív hívásban rendezzük a pontokat. Ha ezt tennénk, a futási idő rekurziós formulája $T(n) = 2T(n/2) + O(n \lg n)$ volna, melynek megoldása $T(n) = O(n \lg^2 n)$. (Alkalmazzuk a mester módszernek a 4.4-2. gyakorlatban megadott változatát.) Egy kicsit kés őbb látni fogjuk, hogyan kell használni az „előrendezést” e rendezettség fenntartása érdekében, anélkül, hogy tényleg minden rekurzív hívásban rendeznénk.

A P, X és Y bemenetű rekurzív hívás először megnézi, hogy $|P| \leq 3$ teljesül-e. Ha igen, a hívás egyszerűen végrehajtja a fent leírt „nyers erő” alapú módszert: megvizsgálja az

összes $\binom{|P|}{2}$ pontpárt, és visszaadja a legközelebbi párt. Ha viszont $|P| > 3$, a rekurzív hívás a következő „oszd-meg-és-uralkodj” elvű megfontolások szerint jár el.

Felosztás: Keresünk egy l függőleges egyenest, amely a P ponthalmazt egy P_L és egy P_R halmazra vágja ketté, úgy, hogy $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, P_L minden pontja vagy rajta van az l egyenesen, vagy balra van tőle, és P_R minden pontja vagy rajta van az l egyenesen, vagy jobbra van tőle. Az X tömböt szintén két, X_L és X_R tömbre osztjuk, melyek rendre P_L és P_R pontjait tartalmazzák x-koordinátájuk szerint monoton növekvő sorrendben. Hasonlóképpen az Y tömböt is két, Y_L és Y_R tömbre osztjuk, melyek rendre P_L és P_R pontjait tartalmazzák y-koordinátájuk szerint monoton növekvő sorrendben.

Uralkodás: Miután P-t kettéosztottuk P_L-re és P_R-re, két rekurzív hívást hajtunk végre. Az egyikkel a P_L-beli pontok között keressük meg a legközelebbi pontpárt, a másikkal pedig a P_R-beliek között. Az első hívás bemenete a P_L részhalmaz és az X_L és Y_L tömb. A második hívás a P_R, X_R és Y_R bemeneti adatokat kapja. Legyen a P_L-re és P_R-re visszaadott legközelebbi párokon belüli távolság rendre δ_L és δ_R, és legyen $\delta = \min(\delta_L, \delta_R)$.

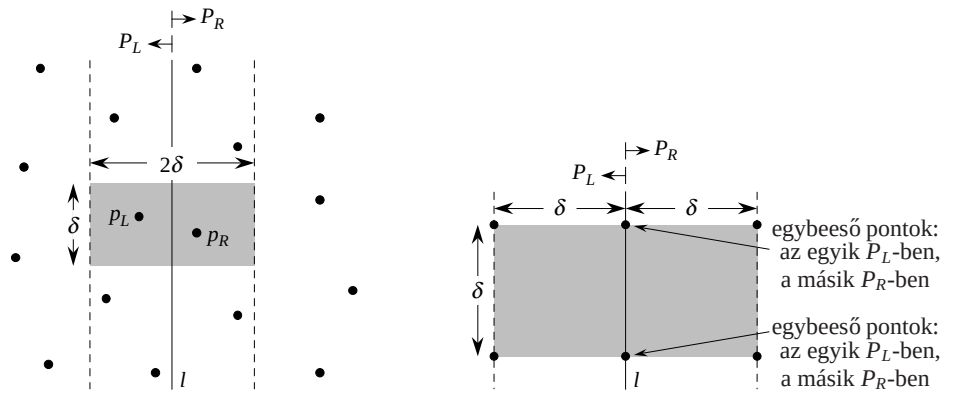
Egyesítés: A legközelebbi pár vagy a két rekurzív hívás egyikében megtalált δ távolságú pár, vagy egy olyan pár, melynek egyik pontja P_L-ben a másik pedig P_R-ben van. Az algoritmus meghatározza, hogy van-e ilyen utóbbi típusú pár, melyen belül a távolság kisebb, mint δ. Vegyük észre, hogy ha van ilyen, δ-nál kisebb távolságú pontpár, akkor a pár mindkét pontjának az l egyeneshez képest δ egységen belül kell lennie. Így, amint azt a 33.11(a) ábra mutatja, mindkettőjüknek az l egyenesre szimmetrikus, 2δ szélességű függőleges sávban kell lenniük. Egy ilyen pár megkeresése céljából, feltéve, hogy létezik ilyen, az algoritmus során a következőképpen járunk el.

1. Létrehozzuk az Y' tömböt az Y tömb azon pontjaiból, melyek a 2δ szélességű függőleges sávba esnek. Az Y' tömböt, ugyanúgy, mint Y-t, az y-koordináta szerint rendezzük.
2. Az Y' tömb minden p pontjára megpróbáljuk megkeresni azokat a pontokat Y'-ben, amelyek p-hez képest δ egységen belül vannak. Mint azt rövidesen látni fogjuk, csak az Y'-ben közvetlenül p-t követő 7 pontot kell figyelembe vennünk. E 7 pont mindegyikére kiszámoljuk a p-től vett távolságot, és ezt folytatva meghatározzuk az Y' összes pontpárjára vonatkozó δ' legkisebb távolságot.
3. Ha $\delta' < \delta$, akkor tényleg van a függőleges szalagban egy közelebbi pár annál, mint amit a rekurzív hívásokkal már megtaláltunk. Ekkor ezt a párt és az ő δ' távolságukat adjuk vissza. Egyébként a rekurzív hívások által megtalált legközelebbi párt és azok δ távolságát adjuk vissza.

A fenti leírásból kimaradt néhány, a megvalósítás során az $O(n \lg n)$ futási időre elérésére szükséges részlet. Az algoritmus helyességének bizonyítása után meg fogjuk mutatni azt is, hogyan kell az algoritmust megvalósítani úgy, hogy elérjük a kívánt időkorlátot.

Helyesség

E legközelebbi pontpárt kereső algoritmus helyessége két részlet kivételével magától értetődik. Az egyik: azzal, hogy a rekurziót $|P| \leq 3$ esetén már nem hajtjuk végre, biztosítjuk, hogy soha ne próbáljuk meg megoldani az egyetlen pontból álló részfeladatot. A másik



33.11. ábra. A legközelebbi pontpárt kereső algoritmusban az Y' tömb minden egyes pontjához elég a soron következő 7 pontot vizsgálni. A bizonyítás lépései. **(a)** Ha $p_L \in P_L$ és $p_R \in P_R$ távolsága kisebb, mint δ, akkor létezik olyan l egyenesre szimmetrikus $\delta \times 2\delta$ méretű téglalap, amelyben mindketten benne vannak. **(b)** Így lehet elhelyezni 4, páronként legalább δ távolságra lévő pontot egy $\delta \times \delta$ méretű négyzetben. A bal oldalon van 4 pont P_L-ben, a jobb oldalon pedig van 4 pont P_R-ben is. Azaz összesen 8 pont lehet egy $\delta \times 2\delta$ méretű téglalapban, ha az l egyenesen látható pontok egybeeső pontpárok, úgy, hogy egyikük P_L-ben, a másikuk P_R-ben van.

bizonyítandó részlet az, hogy az Y' tömb minden p pontjára csak az utána következő 7 pontot kell vizsgálnunk. Ezt most bebizonyítjuk.

Tételezzük fel, hogy a rekurzió egy bizonyos szintjén a legközelebbi pontpár $p_L \in P_L$ és $p_R \in P_R$. Így a p_L és p_R közti δ' távolság szigorúan kisebb, mint δ. A p_L pontnak vagy rajta kell lennie az l egyenesen, vagy δ-nál kisebb távolsággal tőle balra. Hasonlóképpen p_R vagy rajta van az l egyenesen, vagy δ-nál kisebb távolsággal tőle jobbra. Továbbá p_L és p_R egymáshoz képest függőlegesen δ távolságon belül vannak. Így, mint azt a 33.11(a) ábra mutatja, p_L és p_R egy l-re szimmetrikus $\delta \times 2\delta$ méretű téglalapban vannak. (Más pontok is lehetnek ebben a téglalapban.)

Most megmutatjuk, hogy P-nek legfeljebb 8 pontja lehet ebben a $\delta \times 2\delta$ méretű téglalapban. Tekintsünk egy $\delta \times \delta$ méretű négyzetet, ennek a téglalapnak a bal oldalát. Mivel P_L pontjai legalább δ távolságra vannak egymástól, legfeljebb 4 lehet ezen a négyzeten belül: a 33.11(b) ábra mutatja, hogyan. Hasonlóképpen, a P_R-beli pontok közül legfeljebb 4 lehet a téglalap jobb felét alkotó $\delta \times \delta$ méretű négyzetben is. Így P-nek legfeljebb 8 pontja lehet a $\delta \times 2\delta$ méretű téglalapban. (Megjegyezzük, hogy mivel az l egyenesen fekvő pontok akár P_L-ben, akár P_R-ben lehetnek, legfeljebb 4 pont eshet l-re. Ez a határeset akkor fordul elő, ha két egybeeső pontpárunk van úgy, hogy mindegyik pár egy P_L-beli és egy P_R-beli pontból áll, az egyik pár az l egyenes és a téglalap tetejének metszéspontjában van, a másik pár pedig ott, ahol l a téglalap alját metszi.)

Mivel már megmutattuk, hogy P-nek legfeljebb 8 pontja lehet a téglalapban, könnyű lesz belátni, hogy csak azt a 7 pontot kell megvizsgálnunk, melyek az Y' tömb egyes pontjai után következnek. Továbbra is feltételezve, hogy a legközelebbi pontpár p_L és p_R, tételezzük fel az általánosság megszorítása nélkül, hogy p_L megelőzi p_R-t az Y' tömbben. Ekkor még abban az esetben is, ha Y'-n belül p_L a lehető legkorábban, p_R pedig a lehető legkésőbb fordul elő, p_R a p_L-t követő 7 helyen lévő pont valamelyike. Így bebizonyítottuk a legközelebbi párt kereső algoritmus helyességét.

A megvalósítás és a futási idő

Mint azt már jeleztük, a futási időre $T(n) = 2T(n/2) + O(n)$ alakú rekurziós formulát szeretnénk kapni, ahol $T(n)$ a futási idő egy n elemű ponthalmaz esetén. A legnagyobb nehézség ebben annak biztosítása, hogy az X_L, X_R, Y_L és Y_R tömbök, melyeket a rekurzív hívások során átadunk, a megfelelő koordináták szerint rendezve legyenek, és az Y' tömb is rendezve legyen az y-koordináta szerint. (Megjegyezzük, hogy ha az X tömb, melyet egy rekurzív hívás megkap, már rendezett, akkor a P halmaz P_L-re és P_R-re osztása lineáris időben végrehajtható.)

Kulcsfontosságú megfigyelés, hogy minden hívásban egy rendezett tömb rendezett részhalmazát akarjuk előállítani. Például egy bizonyos híváskor megkapjuk a P részhalmazt és az Y tömböt, az utóbbit az y-koordináta szerint rendezve. Miután felosztottuk P-t P_L-re és P_R-re, szükségünk van az y-koordináta szerint rendezett Y_L és Y_R tömbökre is. Ráadásul ezeket a tömböket lineáris időben kell létrehozni. A módszer a 2.3. alfejezetbeli összefésülő rendezés ÖSSZEFÉSÜL eljárása ellenkezőjének is tekinthető: kettéosztunk egy rendezett tömböt két rendezett tömbbé. A következő pszeudokód mutatja be az ötletet.

```
1  hossz[Y_L] ← hossz[Y_R] ← 0
2  for i ← 1 to hossz[Y]
3      do if Y[i] ∈ P_L
4            then hossz[Y_L] ← hossz[Y_L] + 1
5                 Y_L[hossz[Y_L]] ← Y[i]
6            else hossz[Y_R] ← hossz[Y_R] + 1
7                 Y_R[hossz[Y_R]] ← Y[i]
```

Egyszerűen csak sorban megvizsgáljuk az Y tömbben lévő pontokat. Ha egy $Y[i]$ pont P_L-ben van, hozzáfűzzük az Y_L tömb végéhez, egyébként pedig az Y_R tömb végéhez fűzzük hozzá. Hasonló pszeudokóddal megoldható az X_L, X_R és Y' tömbök előállítása is.

Az egyetlen fennmaradó kérdés az, honnan lesznek rendezett pontjaink az első alkalommal. Ezt ***előrendezéssel*** oldjuk meg, azaz egyszer s mindenkorra rendezzük a pontokat az első rekurzív hívás *előtt*. Ezeket a rendezett tömböket adjuk át az első rekurzív híváskor, és ettől kezdve szükség szerint belőlük faragunk le részeket a rekurzív hívások során. Az előrendezés egy újabb $O(n \lg n)$ tagot ad a futási időhöz, de ezután a rekurzió minden lépése lineáris idejű, a rekurzív hívásokat leszámítva. Így, ha minden egyes rekurzív lépés futási ideje $T(n)$, az egész algoritmus futási ideje pedig $T'(n)$, akkor $T'(n) = T(n) + O(n \lg n)$ és

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + O(n), & \text{ha } n > 3, \\ O(1), & \text{ha } n \leq 3. \end{cases}$$

Így $T(n) = O(n \lg n)$ és $T'(n) = O(n \lg n)$.

Gyakorlatok

33.4-1. Smothers professzor olyan elképzeléssel állt elő, amely lehetővé teszi a legközelebbi pontot kereső algoritmus számára, hogy az Y' tömb minden egyes pontjára csak a soron következő 5 pontot vizsgálja. Az ötlet az, hogy az l egyenesre eső pontokat tegyük mindig a P_L halmazba. Ekkor nem lehetnek egybeeső párok az l egyenesen úgy, hogy közülük az egyik P_L-ben a másik P_R-ben van. Így legfeljebb 6 pont lehet a $\delta \times 2\delta$ méretű téglalapban. Hol a hiba a professzor elképzelésében?

33.4-2. Hogyan lehet az algoritmus aszimptotikus futási idejének növelése nélkül biztosítani, hogy a legelső rekurzív hívás során átadott ponthalmaz ne tartalmazzon egybeeső pontokat? Bizonyítsuk be, hogy ekkor már tényleg elegendő az Y' tömb minden egyes pontjára a tömbben a soron következő 5 helyen lévő pontot vizsgálni.

33.4-3. Két pont távolsága az euklideszitől eltérő módon is definiálható. A síkon a p_1 és p_2 pontok közötti ***L_m-távolságot*** az $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$ kifejezés adja meg. Az euklideszi távolság ezek szerint az L_2-távolság. Módosítsuk a legközelebbi pontpárt kereső algoritmust úgy, hogy az a – ***Manhattan-távolságnak*** is nevezett – L_1-távolságot használja.

33.4-4. Adott p_1 és p_2 pont között a síkon az L_∞-távolságot a $\max(|x_1 - x_2|, |y_1 - y_2|)$ kifejezés adja meg. Módosítsuk a legközelebbi pontpárt kereső algoritmust úgy, hogy az az L_∞-távolságot használja.

33.4-5. Hogyan lehetne úgy megváltoztatni a legközelebbi pontpárt kereső algoritmust, hogy elkerüljük az Y tömb előrendezését, ugyanakkor a futási idő változatlanul $O(n \lg n)$ maradjon. (*Útmutatás.* A rendezett Y_L és Y_R tömbök összefésülésével hozzuk létre a rendezett Y tömböt.)

Feladatok

33-1. Konvex rétegek

Adott Q ponthalmaz ***konvex rétegeit*** a síkon a következő rekurzív módon definiáljuk. A Q halmaz első konvex rétege azon Q-beli pontokból áll, amelyek $\mathrm{CH}(Q)$ csúcsai. Álljon a Q_i halmaz, $i > 1$ esetén, Q azon pontjaiból, amelyek az $1, 2, \ldots, i-1$ sorszámú konvex rétegekben nem szerepelnek. Majd a Q halmaz i-edik konvex rétege legyen $\mathrm{CH}(Q_i)$, ha $Q_i \neq \emptyset$, egyébként ne értelmezzük.

a. Adjunk $O(n^2)$ futási idejű algoritmust egy n elemű ponthalmaz konvex rétegeinek megkeresésére.

b. Bizonyítsuk be, hogy $\Omega(n \lg n)$ idő szükséges egy n elemű ponthalmaz konvex rétegeinek meghatározására minden olyan számítási modell esetén, amely n valós szám rendezésére $\Omega(n \lg n)$ időt igényel.

33-2. Maximális rétegek

Legyen Q egy n elemű ponthalmaz a síkon. Azt mondjuk, hogy az (x, y) pont ***uralkodik*** az (x', y') pont fölött, ha $x \geq x'$ és $y \geq y'$. Az olyan Q-beli pontot, amely fölött nem uralkodik más Q-beli pont, ***maximális pontnak*** nevezzük. Megjegyezzük, hogy Q-nak több maximális pontja is lehet, és ezeket ***maximális rétegekbe*** szervezhetjük az alábbiak szerint. Az első maximális réteg, L_1, a Q halmaz maximális pontjainak halmaza. Az i-edik maximális réteg, $i > 1$ esetén, a $Q - \bigcup_{j=1}^{i-1} L_j$ halmaz maximális pontjainak halmaza.

Tételezzük fel, hogy Q-nak van k nem üres maximális rétege, és legyen y_i az L_i-beli bal szélső pont y-koordinátája, $i = 1, 2, \ldots, k$. Tegyük fel a továbbiakban, hogy nincs két olyan pont Q-ban, melyek x- vagy y-koordinátája megegyezik.

a. Mutassuk meg, hogy $y_1 > y_2 > \cdots > y_k$.

Tekintsük azt az (x, y) pontot, amely minden Q-beli ponttól balra van, és amelyre y különbözik a többi Q-beli pont y-koordinátájától. Legyen $Q' = Q \cup \{(x, y)\}$.

b. Legyen j az a legkisebb index, melyre $y_j < y$, kivéve, ha $y < y_k$, amikor legyen $j = k+1$. Mutassuk meg, hogy Q' maximális rétegeire teljesülnek a következők.

- Ha $j \leq k$, akkor Q' maximális rétegei ugyanazok, mint Q maximális rétegei, azzal az eltéréssel, hogy L_j-ben (x, y) is szerepel, mint annak bal szélső pontja.
- Ha $j = k + 1$, akkor Q' első k maximális rétege ugyanaz, mint Q-é, de ezeken kívül Q'-nek van egy $(k + 1)$-edik nem üres maximális rétege is, $L_{k+1} = \{(x, y)\}$.

c. Adjunk egy $O(n \lg n)$ idejű algoritmust egy n elemű Q ponthalmaz maximális rétegeinek meghatározására. (*Útmutatás.* Mozgassunk egy söprő egyenest jobbról balra.)

d. Okoz-e nehézséget az, ha megint megengedjük, hogy legyenek az adott pontok között egyező x- vagy y-koordinátájúak? Javasoljunk megoldást ezekre a problémákra.

33-3. Szellemirtók és szellemek

Egy n szellemirtóból álló csoport n szellem ellen harcol. Mindegyik szellemirtó protonpuskával van felfegyverezve, melyből a szellemre pusztító hatású sugárnyalábot tud kilőni. A sugárnyaláb egyenes vonalban halad, és véget ér, ha eltalál egy szellemet. A szellemirtók a következő stratégia mellett döntenek. Párba állnak a szellemekkel, azaz n darab szellemirtó-szellem párt alakítanak ki, majd egyidejűleg mindegyik szellemirtó lelövi az ő kiválasztott szellemét. Mint közismert, *nagyon* veszélyes, ha a sugárnyalábok keresztezik egymást, így a szellemirtóknak olyan párosítást kell találniuk, amelyben a sugárnyalábok nem fogják keresztezni egymást.

Tételezzük fel, hogy mindegyik szellemirtó és szellem helyzete egy rögzített pont a síkon, és ezek közül semelyik három sem kollineáris.

a. Mutassuk meg, hogy létezik olyan, szellemirtón és szellemen áthaladó egyenes, melyre az egyenes egyik oldalán lévő szellemirtók száma megegyezik az ugyanazon az oldalon lévő szellemek számával. Hogyan lehet egy ilyen egyenest $O(n \lg n)$ időben megkeresni?

b. Adjunk egy $O(n^2 \lg n)$ idejű algoritmust a szellemirtók szellemekkel való párosítására, úgy, hogy a sugárnyalábok ne keresztezzék egymást.

33-4. Pálcikák felszedegetése

Charon professzornak van egy n pálcikából álló halmaza. A pálcikák egy bizonyos elrendezés szerint, egymás hegyén-hátán fekszenek. Az egyes pálcikák helyzetét végpontjaik határozzák meg. Minden végpont egy rendezett számhármassal, (x, y, z) koordinátáival adott. Egyik pálcika sem függőleges. Charon professzor egyesével fel akarja szedni az összes pálcikát, azzal a feltétellel, hogy egy pálcikát csak akkor szabad felemelni, ha nincs fölötte másik pálcika.

a. Adjunk egy eljárást, amely veszi az a és b pálcikát, és eldönti, hogy vajon a fölötte van-e b-nek, alatta van-e b-nek, vagy nincs közöttük ilyen kapcsolat.

b. Írjunk egy hatékony algoritmust, amely meghatározza, hogy fel lehet-e szedni az összes pálcikát, és ha igen, akkor megadja ennek egy megengedett sorrendjét.

33-5. Ritka burkú eloszlások

Adott egy ismert eloszlás szerint véletlenszerűen választott ponthalmaz a síkon. Tekintsük a konvex burok meghatározására irányuló feladatot. Adott eloszlás szerint véletlenszerűen

választott n pontra, bizonyos esetekben, a konvex burok méretének, vagyis a benne szerepl ő pontok számának várható értéke $O(n^{1-\epsilon})$, valamely $\epsilon > 0$ állandó esetén. Az ilyen eloszlást ***ritka burkúnak*** nevezzük. Ritka burkú eloszlások például a következ ők:

- Az egység sugarú körlemezr ől egyenletes eloszlás szerint választott pontok. A konvex burok várható mérete $\Theta(n^{1/3})$.
- Egy k oldalú konvex poligon belsejéb ől egyenletes eloszlás szerint választott pontok, bármely rögzített k esetén. A konvex burok várható mérete $\Theta(\lg n)$.
- A kétdimenziós normális eloszlás szerint választott pontok. A konvex burok $\Theta(\sqrt{\lg n})$ várható méretű.

a. Adott két, rendre n_1 és n_2 csúcsú, konvex poligon. Hogyan határozható meg mind az $n_1 + n_2$ pont konvex burka $O(n_1 + n_2)$ időben? (A poligonok átfedhetik egymást.)

b. Mutassuk meg, hogy ritka burkú eloszlás szerint egymástól függetlenül véletlenszer űen választott n pont halmazának konvex burka $O(n)$ várható id őben meghatározható. (*Útmutatás.* Rekurzívan keressük meg az els ő $n/2$ pont és a második $n/2$ pont konvex burkát, és egyesítsük az eredményeket.)

Megjegyzések a fejezethez

Ez a fejezet épphogy érinti a geometriai algoritmusok és módszerek felszínét. Geometriai algoritmusok témakörét tárgyaló könyv, például, Preparata és Shamos [247], Edelsbrunner [83] valamint O'Rourke [235] műve.

Bár a geometriát már az ókor óta műveli az ember, geometriai feladatok megoldása céljából algoritmusok fejlesztésével foglalkozni viszonylag új keletű. Preparata és Shamos szerint egy feladat bonyolultságáról el őször E. Lemoine alkotott fogalmat 1902-ben. Lemoine az euklideszi szerkesztéseket tanulmányozta – azokat, amelyek körz ővel és vonalzóval végrehajthatók –, és öt elemi műveletre vezette vissza őket: a körző egyik szárának egy adott pontra helyezése, a körz ő egyik szárának egy adott egyenesre helyezése, kör rajzolása, a vonalzó élének egy adott pontra illesztése és egyenes rajzolása. Lemoine-t az érdekelte, hogy egy adott szerkesztés végrehajtásához hány elemi lépésre van szükség. Ezt a számot a szerkesztés „egyszerűségének" hívta.

A 33.2. alfejezet algoritmusa annak eldöntésére, hogy van-e metsz ő szakaszpár a szakaszhalmazban, Shamos és Hoey [275] eredménye.

A Graham-féle pásztázás eredeti változatát Graham [130] közölte. Az ajándékcsomagolás elve Jarvis [165] eredménye. Döntésifa modell felhasználásával Yao [318] bebizonyította, hogy bármely konvex burok meghatározására szolgáló algoritmus futási idejének alsó határa $\Omega(n \lg n)$. Kirkpatrick és Seidel [180] eltávolító és keres ő algoritmusa, amely a konvex burok csúcsainak számát, h-t is figyelembe véve $O(n \lg h)$ futási idejű, aszimptotikusan optimális.

Az $O(n \lg n)$ idejű „oszd-meg-és-uralkodj" elvű algoritmus az egymáshoz legközelebbi pontpár megkeresésére Shamos nevéhez fűz ődik, és Preparata és Shamos [247] könyvében található meg. Preparata és Shamos azt is megmutatja, hogy az algoritmus aszimptotikusan optimális egy döntésifa modellben.

34. NP-teljesség

Az eddig vizsgált algoritmusok csaknem valamennyien ***polinomiális idejűek,*** azaz n méretű bemeneten futási idejük a legrosszabb esetben is $O(n^k)$, valamely k konstanssal. Természetes a kérdés: vajon *minden* probléma megoldható-e polinomiális időben. A válasz természetesen: nem. Vannak problémák – mint például Turing híres megállási problémája –, melyek egyáltalán nem oldhatók meg számítógéppel, bármennyi idő áll is rendelkezésre. Vannak olyanok is, melyek megoldhatók ugyan, de nem polinomiális időben. A polinomiális idejű algoritmussal megoldható problémákat általában könnyűnek tekintjük, a szuperpolinomiális időt igénylőket pedig nehéznek.

E fejezet témája egy rendkívül érdekes problémaosztály – az úgynevezett NP-teljes problémák osztálya. NP-teljes problémára mind ez idáig senki sem adott polinomiális algoritmust, mint ahogy szuperpolinomiális alsó becslést sem. A P $\neq$ NP sejtés felvetése, azaz 1971 óta az elméleti számítógéptudomány egyik legmélyebb, legnehezebb kutatási területe.

Az NP-teljes problémák egyik különösen kellemetlen tulajdonsága, hogy számosan közülük ránézésre igen hasonlóak olyan problémákhoz, melyekre létezik polinomiális algoritmus. A következő problémapárok mindegyikében az egyik probléma polinomiális időben megoldható, míg a másik NP-teljes, annak ellenére, hogy a köztük lévő különbség csekélynek tűnik.

Legrövidebb és leghosszabb utak: A 24. fejezetben láttuk, hogy még negatív élsúlyok mellett is meg tudjuk találni az egy adott pontból a többi pontba menő *legrövidebb* utakat egy irányított $G = (V, E)$ gráfban $O(VE)$ idő alatt. A két adott pont közötti egyik *leghosszabb* út megtalálása viszont nehéz. Már annak eldöntése is NP-teljes, hogy létezik-e egy gráfban egy adott k számú élnél többet tartalmazó egyszerű út.

Euler- és Hamilton-körök: egy összefüggő, irányított $G = (V, E)$ gráf ***Euler-körének*** nevezünk egy körsétát, ha G minden élén pontosan egyszer megy végig, egy csúcson azonban többször is átmehet. A 22-3. feladatnál láttuk, hogy annak eldöntése, hogy egy adott irányított gráf tartalmaz-e Euler-kört, elvégezhető $O(E)$ időben, sőt $O(E)$ idő alatt meg is találhatunk egy Euler-kört (ha van). Egy irányított $G = (V, E)$ gráf ***Hamilton-körének*** nevezünk egy egyszerű kört, ha a V-ben szereplő valamennyi csúcsot tartalmazza. Annak eldöntése, hogy egy adott gráf tartalmaz-e Hamilton-kört, NP-teljes. (Később bebizonyítjuk, hogy annak eldöntése, hogy egy *irányítatlan* gráf tartalmaz-e Hamilton-kört, NP-teljes.)

2-CNF kielégíthetőség és 3-CNF kielégíthetőség: egy Boole-formula 0 vagy 1 értékű változókból, egy- vagy kétváltozós Boole-függvényekből, mint például $\wedge$ (ÉS), $\vee$ (VAGY), $\neg$ (NEM) és zárójelekből áll. Egy Boole-formula ***kielégíthető,*** ha létezik a változóknak olyan behelyettesítése, amelyre a formula értéke 1. A most következő fogalmakat később pontosan is definiáljuk, egyelőre érjük be annyival, hogy egy Boole-formula ***k-konjunktív normálformájú*** vagy k-CNF (conjunctive normal form), ha olyan zárójeles tagok ÉS-ekkel való összekapcsolásából áll, melyek pontosan k változót vagy negált változót tartalmaznak. Például az $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ Boole-formula 2-konjunktív normálformájú (és kielégíthető: az $x_1 = 1, x_2 = 0, x_3 = 1$ behelyettesítésre az értéke 1). Annak eldöntésére, hogy egy 2-CNF kielégíthető-e, létezik polinomiális idejű algoritmus, a 3-CNF formulák kielégíthetőségének kérdése viszont – amint azt látni is fogjuk – NP-teljes.

A P és NP osztályok, NP-teljesség

E fejezetben három problémaosztállyal fogunk foglalkozni: a P, NP és NPC osztályokkal. Informálisan leírjuk őket most is, a pontos definíciót azonban csak később adjuk meg.

A P osztály a polinom időben megoldható problémákból áll, azaz olyan problémákból, melyekhez létezik olyan k konstans, hogy a probléma n hosszú bemenet esetén $O(n^k)$ idő alatt megoldható. Az eddigi fejezetekben vizsgált problémák túlnyomó többsége P-be tartozik.

Az NP osztály olyan problémákból áll, amelyek polinom időben ellenőrizhetők. Ez valami olyasmit jelent, hogy ha valaki egy „bizonyítékot” adna nekünk a megoldásról, akkor annak helyességét polinomiális időben le tudnánk ellenőrizni. Például a Hamilton-kör probléma esetén egy $G = (V, E)$ gráfhoz tartozó bizonyítéknak megfelelne egy $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ különböző V-beli csúcsokból álló sorozat. Könnyen ellenőrizhető polinomiális időben, hogy a csúcsok valóban különbözők-e, továbbá az is, hogy $(v_i, v_{i+1}) \in E$ minden $i = 1, 2, 3, \ldots, |V|-1$-re és $(v_{|V|}, v_1) \in E$ teljesül-e. Másik példaként említhetjük a 3-CNF kielégíthetőséget, itt a változók egy behelyettesítése a bizonyíték, melyről valóban ellenőrizhető polinom időben, hogy csakugyan kielégíti-e az adott Boole-formulát.

Bármely P-beli probléma az NP osztályba is beletartozik, hiszen a P-beli problémákat meg tudjuk oldani polinomiális időben, még bizonyíték nélkül is. Ezt az elképzelést később pontosan leírjuk, addig kénytelenek vagyunk elhinni, hogy P $\subseteq$ NP. Nyitott kérdés, hogy P valódi részhalmaza-e NP-nek.

Az ***NP-teljes*** problémák osztályát NPC-vel jelöljük. Ide azok a problémák tartoznak, melyek „legalább olyan nehezek”, mint bármely NP-beli probléma. Hogy a „legalább olyan nehéz” mit is jelent, azt később pontosan definiálni fogjuk. Addig is kimondjuk bizonyítás nélkül, hogy ha az NP-teljes problémák közül *akár csak egy is* megoldható polinomiális időben, akkor *minden* NP-beli probléma megoldható polinomiális időben. A matematikusok nagy része úgy gondolja, hogy az NP-teljes problémák nem polinomiálisak. Napjainkig sokféle NP-teljes problémát vizsgáltak, a polinomiális idejű algoritmus irányába történő legcsekélyebb előrehaladás nélkül, így igencsak meglepő lenne, ha kiderülne, hogy ezen problémák mindegyikére létezik polinomiális megoldás. Tekintetbe véve mindazonáltal azt is, hogy a P $\neq$ NP állítás bizonyításának szentelt jelentős erőfeszítések sem hoztak eredményt, nem zárható ki annak lehetősége, hogy az NP-teljes problémák mégiscsak megoldhatók polinom időben.

Aki jó algoritmusokat szeretne tervezni, annak feltétlenül értenie kell az NP-teljesség elméletének alapjait. Ha például valaki mérnöki munkája során NP-teljes problémával találkozik, többnyire az a legjobb, ha alkalmas közelítő algoritmust (lásd 35. fejezet) próbál találni, és nem vesződik a gyors, pontos megoldás kitalálásával. Ezen túlmenően igen sok érdekes és természetesen felmerülő probléma, amely első ránézésre nem tűnik nehezebbnek, mint a keresés, a rendezés vagy a hálózati folyamok, NP-teljesnek bizonyul. Ezért fontos, hogy közelebbről megismerkedjünk ezzel a figyelemre méltó problémaosztállyal.

Hogyan mutatjuk meg, hogy egy probléma NP-teljes?

Azok a módszerek, melyeket egy adott probléma NP-teljességének bizonyítására használunk, lényegesen különböznek az e könyvben algoritmusok tervezésére és elemzésére használatos technikák legnagyobb részétől. Ennek alapvetően az az oka, hogy ilyenkor azt mutatjuk meg, hogy egy probléma mennyire nehéz (vagy legalábbis mennyire nehéznek gondoljuk), nem pedig azt, hogy mennyire könnyű. Nem azt próbáljuk meg bebizonyítani, hogy létezik hatékony algoritmus, hanem éppen ellenkezőleg: azt, hogy valószínűleg egyáltalán nem létezik hatékony algoritmus. Ilyeténképpen az NP-teljességi bizonyítások leginkább arra a bizonyításra hasonlítanak, melyet a 8.1. alfejezetben láttunk az összehasonlító rendezések lépésszámának $\Omega(n \lg n)$-es alsó becslésére, bár az ott használt, döntési fákon alapuló módszer itt nem fordul elő.

Az NP-teljességi bizonyításoknak három fő építőköve van.

Döntési és optimalizálási problémák

Az érdeklődésünk középpontjában álló problémák jelentős része ***optimalizálási probléma,*** ahol minden megengedett megoldáshoz egy érték tartozik, feladatunk pedig olyan megengedett megoldás megtalálása, amelyhez a legjobb érték van rendelve. Például a LEGRÖVIDEBB-ÚT-nak nevezett probléma esetében adott egy G irányítatlan gráf és az u, v csúcsok, feladatunk pedig olyan u és v közötti út megtalálása, melynek a lehető legkevesebb éle van. (Más szavakkal LEGRÖVIDEBB-ÚT a két adott pont közti legrövidebb út probléma egy súlyozatlan, irányítatlan gráfban.) Az NP-teljesség fogalmát azonban nem optimalizálási problémákra alkalmazzuk közvetlenül, hanem úgynevezett ***döntési problémákra,*** ahol a válasz egyszerűen „igen” vagy „nem” (formálisabban „1” vagy „0”).

Bár problémák NP-teljességének bizonyításakor tevékenységünket a döntési problémák birodalmára kell korlátoznunk, valójában nyilatkozhatunk optimalizálási problémákról is. Létezik ugyanis egy rendkívül hasznos összefüggés az optimalizálási és a döntési problémák között. Egy adott optimalizálási problémának megfeleltethetünk egy döntési problémát oly módon, hogy az optimalizálandó értékre egy korlátot állapítunk meg. A LEGRÖVIDEBB-ÚT esetében például a megfelelő döntési probléma, melyet ÚT-nak fogunk nevezni, a következő: adott egy G irányítatlan gráf és az u, v csúcsok, továbbá egy k egész szám, döntsük el, hogy létezik-e u és v között olyan út, amely legfeljebb k élből áll.

Az optimalizálási problémák és a hozzájuk rendelt döntési problémák között fennálló kapcsolat nagy segítségünkre van abban, hogy megmutassuk, az adott optimalizálási probléma valóban „nehéz”. Ennek az az oka, hogy a döntési probléma bizonyos értelemben „könnyebb”, vagy legalábbis „nem nehezebb”, mint az optimalizálási probléma, amihez tartozik. Például az ÚT problémát könnyűszerrel meg tudjuk oldani, ha a LEGRÖVIDEBB-ÚT probléma megoldását már ismerjük: egyszerűen össze kell hasonlítanunk a legrövidebb

út hosszát az ÚT döntési problémában szereplő k paraméterrel. Más szóval, ha egy optimalizálási probléma könnyű, akkor a hozzá tartozó döntési probléma is könnyű. Az NP-teljesség elméletéhez jobban illően megfogalmazva: ha bizonyítékunk van rá, hogy egy döntési probléma nehéz, akkor arra is van bizonyítékunk, hogy az az optimalizálási probléma, amelyhez tartozik, szintén nehéz. Ily módon, bár az NP-teljesség elmélete csak döntési problémákkal foglalkozik, gyakran alkalmazható optimalizálási problémákra is.

Visszavezetések

Az imént látott módszer annak megmutatására, hogy egy probléma nem nehezebb egy másiknál, alkalmazható akkor is, ha mindkét probléma döntési. Ezt az ötletet csaknem minden NP-teljességi bizonyításban használni fogjuk, a következőképpen. Tekintsünk egy A döntési problémát, amelyet szeretnénk polinom időben megoldani. Egy adott probléma bemenetét a probléma egy ***esetének*** nevezzük; az ÚT probléma egy esete például egy adott G gráf, az adott u és v csúcsai G-nek és az adott k egész szám. Tegyük fel, hogy van egy másik döntési problémánk, nevezzük B-nek, melyről már tudjuk, hogyan lehet polinomiális időben megoldani. Végül tegyük fel, hogy rendelkezésünkre áll egy eljárás, melynek segítségével az A probléma egy α esetét átalakíthatjuk a B probléma egy β esetévé oly módon, hogy az alábbiak teljesüljenek:

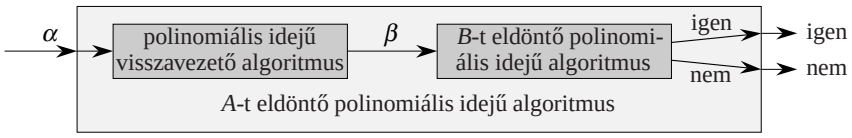
1. Az átalakítás polinomiális ideig tart.
2. A válaszok azonosak, azaz a válasz az α esetre akkor és csak akkor „igen”, ha a válasz a β esetre „igen”.

Az ilyen eljárások, melyeket polinomiális idejű ***visszavezető algoritmusnak*** hívunk, a 34.1. ábrán látható módon szolgáltatnak polinomiális idejű megoldást az A problémára.

1. Az A probléma adott α esetét a polinomiális idejű visszavezető algoritmus segítségével a B probléma egy β esetévé alakítjuk át.
2. Lefuttatjuk a B döntési problémát polinomiális időben megoldó algoritmust a β esetre.
3. A β esetre kapott választ adjuk meg az A probléma α esetéhez tartozó válaszként.

Amennyiben e három lépés mindegyike polinomiális időben megvalósítható, akkor a három lépés együttesen is polinomiális ideig tart, így az A problémát valóban el tudjuk dönteni polinomiális idő alatt. Más szóval, az A probléma megoldását „visszavezetve” a B probléma megoldására, B „könnyűségét” használjuk arra, hogy A „könnyűségét” bizonyítsuk.

Az NP-teljesség elmélete azonban nem arról szól, hogy hogyan láthatjuk be, hogy egy probléma könnyű, hanem éppen az ellenkezőjéről, ezért a polinomiális visszavezetés módszerét a fordított irányban fogjuk alkalmazni, problémák NP-teljességének bizonyítására. Gondoljuk tovább az ötletet és nézzük meg, hogyan láthatnánk be, hogy egy adott B problémára nem létezik polinomiális idejű algoritmus. Tegyük fel, hogy rendelkezésünkre áll egy A döntési probléma, melyről tudjuk, hogy nem oldható meg polinomiális időben. (Egyelőre ne foglalkozzunk azzal, hogyan is találhatnánk ilyen A problémát.) Tegyük fel továbbá, hogy ugyancsak rendelkezésünkre áll egy polinomiális idejű visszavezető algoritmus, amely A eseteit B eseteivé alakítja át. Ekkor egyszerű indirekt bizonyítást adhatunk arra, hogy B nem oldható meg polinom időben. Tegyük fel ugyanis, hogy B megoldására létezik polinomiális algoritmus. A 34.1. ábrán látható módszert alkalmazva A egy polinomiális idejű megoldását kapjuk, ami ellentmond annak a feltevésünknek, hogy A nem oldható meg polinom időben.



34.1. ábra. Polinomiális idejű visszavezető algoritmus használata az A döntési probléma megoldására, a B problémára ismert polinomiális algoritmus ismeretében. A egy α esetét átalakítjuk B egy β esetévé, megoldjuk B-t a β esetre, végül a β-ra kapott választ adjuk meg az A α esetéhez tartozó válaszként.

Az NP-teljesség esetében persze nem tehetjük fel, hogy egyáltalán nem létezik polinomiális megoldás az A problémára. A bizonyítási módszer azonban annyiban hasonló lesz, hogy B NP-teljességét úgy fogjuk belátni, hogy feltesszük, hogy A NP-teljes.

Kezdeti NP-teljes probléma

Minthogy a visszavezetési technika használata során nélkülözhetetlen egy NP-teljes probléma ahhoz, hogy egy másik probléma NP-teljességét bizonyítsuk, szükségünk van (legalább) egy „kezdeti” NP-teljes problémára. Erre a célra a Boole-hálózatok kielégíthetőségének problémáját fogjuk használni, melyben adott egy ÉS, VAGY és NEM kapukból álló Boole-hálózat, melyről el kell döntenünk, hogy léteznek-e olyan bemenetei, melyekre a hálózat kimenete 1. E probléma NP-teljességét a 34.3. alfejezetben bizonyítjuk.

A fejezet tartalma

E helyütt az NP-teljesség azon megközelítéseit tanulmányozzuk, melyek a legszorosabb kapcsolatban állnak az algoritmusok elemzésével. A 34.1. alfejezetben formalizáljuk a „probléma” fogalmát, és definiáljuk a polinomiális időben megoldható problémák P osztályát. Megvizsgáljuk továbbá, hogyan illeszkednek ezen fogalmak a formális nyelvek elméletének szerkezetébe. A 34.2. alfejezetben definiáljuk a polinomiális időben ellenőrizhető megoldású eldöntési problémák NP osztályát. Itt foglalkozunk először a $P \neq NP$ kérdéssel is.

A 34.3. alfejezetben megmutatjuk, hogyan vizsgálhatók a problémák közti összefüggések a ***polinomiális visszavezetés*** segítségével. Definiáljuk az NP-teljességet, és vázoljuk egy bizonyítását annak, hogy egy konkrét probléma – a Boole-hálózatok kielégíthetősége – NP-teljes. Miután találtunk egy NP-teljes problémát, a 34.4. alfejezetben azt mutatjuk meg, miképp bizonyíthatjuk újabb problémák NP-teljességét már jóval egyszerűbben, a polinomiális visszavezetés módszerével. A módszer illusztrációjaként megmutatjuk két formulakielégíthetőségi probléma NP-teljességét. NP-teljességi bizonyítások széles választéka található a 34.5. alfejezetben.

34.1. Polinomiális idő

Az NP-teljesség tanulmányozását a polinomiális időben megoldható probléma fogalmának definiálásával kezdjük. E problémákat – mint már említettük – többnyire jól kezelhetőknek tartjuk. Ennek alátámasztására három érvünk is van, ezek persze sokkal inkább filozófiai, mint matematikai jellegűek.

Először is, noha indokolt egy $\Theta(n^{100})$ lépést igénylő problémát nehezen kezelhetőnek tekinteni, valójában igen kevés gyakorlati probléma létezik, amelyhez ilyen magas fokú polinomiális lépésszám szükséges. A gyakorlatban előforduló polinomiális problémák ennél többnyire lényegesen gyorsabban megoldhatók. Emellett a tapasztalat azt mutatja, hogy egy probléma első polinomiális idejű megoldását gyakran követik újabb, hatékonyabb algoritmusok. Még ha igaz is, hogy egy problémára a leggyorsabb ismert algoritmus $\Theta(n^{100})$ lépést igényel, valószínű, hogy hamarosan felfedeznek majd egy lényegesen gyorsabb algoritmust.

Másodszor, számos ésszerű számítási modellre igaz, hogy egy probléma, amely polinom időben megoldható az egyikben, az a másikban is polinomiális. Például a – könyvünkben általában használatos – véletlen elérésű számítógéppel (RAM) polinom időben megoldható problémák osztálya azonos a Turing-géppel[1] polinom időben megoldható problémák osztályával. Sőt, ugyanezt az osztályt kapjuk akkor is, ha olyan párhuzamos számítógépet használunk, ahol a processzorok száma a bemenet méretével polinomiálisan nő.

Harmadszor, a polinom időben megoldható problémák osztályának szép zártsági tulajdonságai vannak, mivel a polinomok zártak az összeadásra, a szorzásra és a kompozícióra. Például, ha egy polinomiális algoritmus eredményét betápláljuk egy másik polinomiális algoritmusba, a kapott összetett eljárás is polinomiális lesz, csakúgy, mint az az algoritmus, amely polinomiális számú lépésén kívül konstansszor hív meg egy polinomiális szubrutint.

Absztrakt problémák

Ahhoz, hogy a polinom időben megoldható problémák osztályát (vagy egyáltalán bármiféle problémaosztályt) átlássunk, először is meg kell mondanunk, mit is értünk „problémán”. A Q ***absztrakt problémát*** kétváltozós relációként definiáljuk a probléma ***eseteinek*** (bemeneteinek) I és a probléma ***megoldásainak*** S halmazán. A LEGRÖVIDEBB-ÚT probléma egy esete például egy gráf és annak két csúcsa által alkotott hármas, egy megoldás pedig csúcsok valamely sorozata, beleértve az üres sorozatot is, arra az esetre, ha két pont közt nincs út. Maga a LEGRÖVIDEBB-ÚT probléma az a reláció, mely a gráf és a két adott csúcs által alkotott hármashoz egy olyan csúcssorozatot rendel, melynek hossza a legrövidebb a két csúcsot összekötő sorozatok között. Minthogy a legrövidebb út nem egyértelmű, egy esethez több megoldás is tartozhat.

Az absztrakt probléma fenti definíciója jóval általánosabb, mint amire szükségünk van. Amint láttuk, az NP-teljesség elmélete csak ***döntési problémákkal*** foglalkozik, azaz olyanokkal, melyekre a megoldás „igen”, vagy „nem”. Egy absztrakt döntési problémát tekinthetünk olyan függvénynek, mely az I esethalmazt a $\{0, 1\}$ megoldáshalmazra képezi. Ilyen például a már látott ÚT probléma, melyet LEGRÖVIDEBB-ÚT módosításával kaptunk. Legyen $i = \langle G, u, v, k\rangle$ az ÚT döntési probléma egy esete. Ekkor ÚT(i) = 1 („igen”), ha az u és v közötti (egyik) legrövidebb út hossza legfeljebb k, és ÚT(i) = 0 („nem”), különben. Sok absztrakt probléma nem döntési, hanem ***optimalizálási probléma,*** amelyben valamely értéket minimalizálni vagy maximalizálni szeretnénk. Amint azt már láttuk, az optimalizálási problémák rendszerint könnyen átalakíthatók olyan döntési problémákká, melyek nem nehezebbek az eredetinél.

[1] A Turing-gép-modell alapos áttekintése megtalálható pl. Hopcroft és Ullmann [156] vagy Lewis és Papadimitriou [204] műveiben.

Kódolások

Ha számítógépes programmal szeretnénk absztrakt problémákat megoldani, az eseteket úgy kell megadnunk, hogy azt a program megértse. Egy absztrakt objektumokból álló S halmaz ***kódolása*** egy e leképezés S-ről a bináris sorozatokba.[2] Valamennyien jól ismerjük a természetes számok ($\mathbf{N} = \{0, 1, 2, 3, 4, \ldots\}$) bináris kódolását: $\{0, 1, 10, 11, 100, \ldots\}$. E kódolást használva például $e(17) = 10001$. Bárki, aki foglalkozott a számítógép karaktereinek ábrázolásával, otthonos az ASCII vagy az EBCDIC kódok valamelyikében. ASCII kódban például e (A)=1000001. Összetett objektumok is kódolhatók binárisan, az összetevők kódjainak kombinációjaként. Sokszögek, gráfok, függvények, rendezett párok, programok – valamennyien kódolhatók bináris sorozatként.

Egy számítógépes algoritmus, amely megold valamilyen absztrakt döntési problémát, ezek szerint a probléma eseteinek egy kódolását kapja bemenetként. Az olyan problémát, melynek esetei a bináris sorozatok, ***konkrét problémának*** hívjuk. Azt mondjuk, hogy egy algoritmus egy konkrét problémát $O(T(n))$ idő alatt ***megold,*** ha bármely n hosszúságú i esetre a megoldás $O(T(n))$ lépést igényel.[3] Egy konkrét probléma ***polinomiális időben megoldható*** (röviden: polinom időben megoldható), ha létezik algoritmus, ami $O(n^k)$ idő alatt megoldja, valamely k szám mellett.

Most már definiálhatjuk a **P** ***bonyolultsági osztályt***: P a polinom időben megoldható konkrét döntési problémák halmaza.

Kódolás segítségével absztrakt problémákat konkrét problémákká tudunk átalakítani. A Q absztrakt döntési problémát, amely az I esethalmazt $\{0, 1\}$-re képezi, az $e : I \to \{0, 1\}^*$ kódolással konkrét döntési problémává alakíthatjuk, amit $e(Q)$-val jelölünk.[4] Ha az absztrakt probléma egy i esetére $Q(i) \in \{0, 1\}$ a megoldás, akkor a kapott konkrét probléma $e(i)$ esetére is $Q(i)$ a megoldás. Természetesen előfordulhat, hogy bizonyos bináris sorozatok nem állnak elő egyetlen absztrakt eset képeként sem. A kényelem kedvéért megállapodunk abban, hogy az ilyen sorozatok képe a 0. Ily módon a konkrét probléma ugyanazon megoldásokat adja, mint az absztrakt, azokon a bináris sorozatokon, melyek az absztrakt probléma eseteinek felelnek meg.

Szeretnénk a polinomiális idejű megoldhatóság definícióját kiterjeszteni az absztrakt problémákra is, a kódolások felhasználásával, de azt is szeretnénk, hogy a definíció független legyen minden konkrét kódolástól. Ez azt jelentené, hogy a probléma megoldásának hatékonysága nem függene attól, hogy miképp kódoljuk. Sajnálatos módon ez egyáltalán nincs így. Például képzeljük el, hogy egy algoritmusnak a k természetes számot szeretnénk beadni, és tegyük fel, hogy az algoritmus futási ideje $\Theta(k)$. Ha k-t ***unárisan*** kódoljuk, azaz k darab egyesből álló sorozatként, akkor n hosszúságú bemenetre a futási idő $\Theta(n)$, ami persze polinomiális. Ha a lényegesen természetesebb bináris kódolást használjuk, akkor a bemenet hossza csak $n = \lceil \log(k) \rceil + 1$, így a futási idő: $\Theta(k) = \Theta(2^n)$, tehát a bemenet méretében exponenciális. Ez jól mutatja, hogy ugyanaz az algoritmus a kódolástól függően lehet polinomiális és szuperpolinomiális is.

Az absztrakt problémák kódolása tehát igen lényeges a polinomiális idejűség tanulmányozásában. Absztrakt problémák megoldásáról még csak nem is beszélhetünk, amíg nem rögzítünk egy kódolást. Mindamellett a gyakorlatban, ha kizárjuk az olyan „költséges”

[2] Bináris sorozatok helyett bármilyen véges, legalább 2 elemű halmaz elemeiből képzett sorozatok is megfelelnek.
[3] Feltesszük, hogy az algoritmus kimenete elkülönül a bemenetől. Minthogy a kimenet minden bitjének kiírása egy időegységet igényel, a kimenet mérete is $O(T(n))$.
[4] Amint azt hamarosan látni fogjuk, $\{0, 1\}^*$ a 0 és 1 szimbólumokból álló összes lehetséges sztring halmazát jelöli.

kódolásokat, mint például az unáris, a probléma konkrét kódolása nemigen befolyásolja a polinomialitás kérdését. Például az egész számok kettes helyett hármas számrendszerben való megadása nincs hatással a polinomiális megoldhatóságra, hiszen ezek a reprezentációk polinomiális időben átalakíthatók egymásba.

Azt mondjuk, hogy az $f : \{0,1\}^* \to \{0,1\}^*$ függvény ***polinom időben kiszámítható,*** ha létezik olyan A polinomiális algoritmus, amely tetszőleges $x \in \{0,1\}^*$ bemenetre $f(x)$-et adja eredményként. Legyen I egy probléma eseteinek halmaza. Azt mondjuk, hogy az e_1 és e_2 kódolások ***polinomiálisan kapcsoltak,*** ha léteznek olyan f_{12} és f_{21} polinom időben kiszámítható függvények, hogy bármely $i \in I$ esetén $f_{12}(e_1(i)) = e_2(i)$ és $f_{21}(e_2(i)) = e_1(i)$,[5] vagyis az $e_2(i)$ kódolás polinomiális idejű algoritmussal kiszámítható $e_1(i)$-ből és fordítva. A következő lemma szerint ha egy absztrakt probléma e_1 és e_2 kódolásai polinomiálisan kapcsoltak, akkor mindegy, hogy melyiküket használjuk annak eldöntésére, hogy a probléma polinomiális időben megoldható-e.

34.1. lemma. *Legyen Q absztrakt döntési probléma az I esethalmazzal, legyenek továbbá e_1 és e_2 I-nek polinomiálisan kapcsolt kódolásai. Ekkor*

$$e_1(Q) \in P \iff e_2(Q) \in P.$$

Bizonyítás. Szimmetriaokokból nyilván elég azt bizonyítanunk, hogy ha $e_1(Q) \in P$, akkor $e_2(Q) \in P$. Tegyük fel tehát, hogy $e_1(Q)$ $O(n^k)$ időben megoldható, és hogy minden $i \in I$-re az $e_1(i)$ kódolás $O(n^c)$ időben kiszámítható az $e_2(i)$ kódolásból, ahol $n = |e_2(i)|$, k és c konstansok. $e_2(Q)$ megoldásához az $e_2(i)$ bemeneten először ki kell számítanunk $e_1(i)$-t, majd le kell futtatnunk az $e_1(Q)$-t megoldó algoritmust az $e_1(i)$ bemeneten. Mennyi ideig tart ez? Az átkódolás $O(n^c)$ lépés, így $|e_1(i)| = O(n^c)$, hiszen egy (soros) számítógép kimenete nem lehet hosszabb, mint a futási idő. Eszerint a teljes megoldás $O(|e_1(i)|^k) = O(n^{ck})$ lépést igényel, ami polinomiális, hiszen c és k konstans. ■

Láttuk, hogy míg a problémák kettő, illetve három elemű ábécé feletti kódolása nincs hatással „bonyolultságukra”, azaz polinomiális idejű megoldhatóságukra, addig pl. az unáris kódolás megváltoztathatja azt. Annak érdekében, hogy a tárgyalásmódot kódolás-függetlenné tehessük, általában fel fogjuk tenni, hogy a problémák esetei valamely ésszerű, tömör formában vannak kódolva, hacsak mást nem mondunk. Pontosabban megfogalmazva: feltesszük, hogy az egész számok kódolása polinomiálisan kapcsolt bináris reprezentációjukkal, a véges halmazok kódolása pedig polinomiálisan kapcsolt azon kódolásukkal, melyet elemeik zárójelbe tett, vesszőkkel elválasztott felsorolásával kapunk. (Ilyen kódolás például az ASCII.) Ezen „szabványos” kódolás segítségével más matematikai objektumok, mint például vektorok, gráfok, formulák ésszerű kódolását is nyerhetjük. Objektumok szabványos kódolásának jelölésére a hegyes zárójelet használjuk, például $\langle G \rangle$-vel jelöljük a G gráf szabványos kódolását.

Amíg csak a szabványossal polinomiálisan kapcsolt kódokat használunk, beszélhetünk közvetlenül az absztrakt problémák „bonyolultságáról”, anélkül, hogy egy konkrét kódolást kiemelnénk. Éppen ezért általában feltesszük, hogy a problémák esetei bináris sorozatok formájában vannak kódolva, a szabványos kódolás szerint, mindaddig, amíg mást nem

[5] Technikai okokból azt is megköveteljük, hogy az f_{12} és f_{21} függvények „nem eseteket” „nem esetekbe” képezzenek. A ***nem eset*** az $e \in \{0,1\}^*$ sztring olyan dekódolása, amelyre nem létezik olyan i sztring, amelyre $f(i) = e$. Megköveteljük, hogy $f_{12}(x) = y$ teljesüljön az e_1 dekódolás minden x nem esetére, ahol y az e_2 valamelyik nem esete, továbbá hogy $f_{21}(x') = y'$ fennálljon e_2 minden x' nem esetére, ahol y' az e_1 valamely nem esete.

mondunk. Az absztrakt és a konkrét problémákat többnyire nem különböztetjük meg. Az olvasónak hasznos lesz időnként átgondolni az olyan problémák reprezentációját, melyek szabványos kódolása nem nyilvánvaló, és a kódolás befolyásolhatja a bonyolultságot.

Formális nyelvek

E ponton érdemes áttekinteni a formális nyelvek elméletének néhány alapvető definícióját. ***Ábécén*** szimbólumok egy véges halmazát értjük. ***Szavak*** az ábécé elemeiből képzett véges sorozatok, a ***nyelv*** pedig nem más, mint szavak egy tetszőleges halmaza. Például a $\Sigma = \{0, 1\}$ ábécé feletti $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ nyelv a prímszámok bináris reprezentációjának nyelve. Az ***üres szót*** ε-nal jelöljük, az ***üres nyelvet*** $\emptyset$-val, a Σ feletti összes szót tartalmazó nyelvet pedig Σ^*-gal. Ha például $\Sigma = \{0, 1\}$, akkor $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Bármely Σ feletti nyelv Σ^* részhalmaza.

A nyelveken igen sok művelet értelmezhető. Halmazelméleti műveletek, mint például az ***unió*** és a ***metszet*** azonnal következnek abból, hogy a nyelveket halmazokként definiáltuk. Az L nyelv ***komplementere*** $\overline{L} = \Sigma^* - L$. Az L_1 és L_2 nyelvek ***konkatenációja*** (egymás mellé írása, összefűzése) az

$$L = \{x_1 x_2 : x_1 \in L_1,\ x_2 \in L_2\}$$

nyelv. Az L nyelv ***lezártja,*** vagy Kleene-féle csillaga az

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cdots \quad \text{nyelv, ahol } L^k = \underbrace{LL \ldots L}_{k}.$$

A formális nyelvek szempontjából tetszőleges Q döntési probléma esethalmaza egyszerűen Σ^*, ahol $\Sigma = \{0, 1\}$. Minthogy Q-t egyértelműen jellemzi az esetek azon részhalmaza, amire az 1 („igen”) választ kapjuk, Q-t tekinthetjük a következő, $\Sigma = \{0, 1\}$ feletti nyelvnek:

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

Például az ÚT döntési problémához tartozó nyelv:

$$\begin{aligned} \text{ÚT} = \{\langle G, u, v, k\rangle :\ & G = (V, E) \text{ irányítatlan gráf,} \\ & u, v \in V, \\ & k \text{ nemnegatív egész,} \\ & \text{létezik út } G\text{-ben } u \text{ és } v \text{ között, ami legfeljebb } k \text{ hosszú}\}. \end{aligned}$$

(Ahol nem okozhat félreértést, ott a döntési problémákra és a hozzájuk tartozó nyelvekre ugyanazon a néven hivatkozunk.)

A formális nyelvek elmélete lehetővé teszi, hogy a döntési problémák és az őket megoldó algoritmusok közti összefüggéseket tömören kifejezzük. Azt mondjuk, hogy az A algoritmus ***elfogadja*** az $x \in \{0, 1\}^*$ szót, ha az x bemeneten A 1-et ad, röviden, ha $A(x) = 1$; ha $A(x) = 0$, akkor A ***elutasítja*** x-et. Az A algoritmus elfogadja az L nyelvet, ha L minden szavát elfogadja.

Elképzelhető, hogy az L nyelvet elfogadó A algoritmus nem utasít vissza egy x szót, annak ellenére, hogy $x \notin L$; például, ha az algoritmus végtelen ciklusba kerül. Azt mondjuk, hogy az A algoritmus ***eldönti*** az L nyelvet, ha az L-beli szavakat elfogadja, a többi szót pedig elutasítja. (Azaz, ha $L = \{x \in \{0, 1\}^* : A(x) = 1\}$ és $\overline{L} = \{x \in \{0, 1\}^* : A(x) = 0\}$.) Az A algoritmus ***polinom időben elfogadja*** az L nyelvet, ha bármely n hosszúságú $x \in L$ szót

$O(n^k)$ időben elfogad, valamely k szám mellett. Az A algoritmus ***polinom időben eldönti*** az L nyelvet, ha bármely n hosszúságú $x \in \{0,1\}^*$ szót $O(n^k)$ időben elfogad, vagy elutasít, aszerint, hogy a szó L-beli, vagy sem (k most is konstans). Az L nyelv (polinom időben) elfogadható/eldönthető, ha létezik algoritmus, amely (polinom időben) elfogadja/eldönti.

Nézzük meg példaként az ÚT nyelvet, ez polinom időben elfogadható: először egy – szélességi keresésen alapuló – polinomiális algoritmussal kiszámítjuk az u és v közti (egyik) legrövidebb út hosszát, majd ezt összehasonlítjuk k-val. Ha a kapott szám legfeljebb k, az algoritmus kiadja az 1-et eredményként, ha nem, akkor fut tovább a végtelenségig. Ez az algoritmus nem dönti el az ÚT nyelvet, hiszen nem ad 0-t azon esetekben, amikor egy legrövidebb út hossza nagyobb, mint k. Az algoritmus utolsó lépésének csekély módosításával ez persze elérhető lenne. Léteznek azonban nyelvek – mint például a Turing-féle megállási problémához (halting problem) tartozó nyelv – melyek elfogadhatók ugyan, de nem eldönthetők.

A ***bonyolultsági osztályok*** informális definíciója: bonyolultsági osztálynak nevezzük nyelvek egy halmazát, ahol a halmazhoz tartozást az adott nyelvet eldöntő algoritmusok valamilyen ***bonyolultsági mértéke*** – mint például futási idő – határozza meg. A bonyolultsági osztályok tényleges definíciója némiképp technikaibb jellegű – az érdeklődő Olvasó megtalálhatja pl. Hartmanis és Stearns cikkében [140].

A megismert nyelvelméleti fogalmak segítségével megadhatjuk a P bonyolultsági osztály alternatív definícióját: P a $\{0,1\}$ feletti, polinom időben eldönthető nyelvek halmaza. Sőt, P megegyezik a polinom időben elfogadható nyelvek halmazával is.

34.2. tétel. $\mathrm{P} = \{L : L \text{ polinom időben elfogadható}\}$.

Bizonyítás. Mint az a definícióból azonnal látható, a polinomiálisan eldönthető problémák a polinomiálisan elfogadható problémák halmazának részhalmazát képezik, így csak azt kell megmutatnunk, hogy ha létezik L-et elfogadó polinomiális algoritmus, akkor létezik olyan is, ami polinom időben eldönti L-et. Legyen L tetszőleges nyelv, amit az A polinomiális algoritmus elfogad. Egy klasszikus szimulációs trükk segítségével konstruálunk egy olyan A' algoritmust, ami polinom időben el is dönti L-et. Mivel A az n hosszúságú $x \in L$ bemenetet $O(n^k)$ (k konstans) időben elfogadja, létezik c konstans, hogy A x-et legfeljebb $T = cn^k$ lépésben fogadja el. Tetszőleges x bemenetre A' az első T lépésben szimulálja A működését. Ezután A' megvizsgálja, hogy A miképpen viselkedett. Ha A elfogadta x-et, akkor persze A' is elfogadja (azaz kiadja az 1-est végeredményként), ha viszont A nem fogadta el, akkor 0-t ad. Nyilvánvaló, hogy A' polinomiális, és az is, hogy eldönti L-et. ■

Meg kell jegyeznünk, hogy a fenti bizonyítás nem konstruktív. Egy adott $L \in \mathrm{P}$ nyelvhez nem feltétlen ismerjük egy őt elfogadó algoritmus futási idejének korlátját. Mindazonáltal tudjuk, hogy ez a korlát létezik, és így persze az az A' algoritmus is, ami ezt a korlátot ellenőrzi. (Jóllehet ennek megadása sokszor cseppet sem egyszerű.)

Gyakorlatok

34.1-1. Definiáljuk a LEGHOSSZABB-ÚT optimalizálási problémát olyan relációként, ami egy gráfnak és két kijelölt pontjának a két pont közti (egyik) leghosszabb egyszerű út hosszát felelteti meg. Definiáljuk az ÚT′ döntési problémát a következőképp: $\text{ÚT}' = \{\langle G, u, v, k\rangle : G = (V, E)$ irányítatlan gráf, $u, v \in V$, $k \in \mathbf{N}$, létezik egyszerű út G-ben

u és v között, ami legalább k hosszú}. Mutassuk meg, hogy LEGHOSSZABB-ÚT $\in$ P pontosan akkor, ha ÚT′ $\in$ P.

34.1-2. Adjunk formális definíciót a következő problémára: találjuk meg egy irányítatlan gráf leghosszabb körét. Fogalmazzuk meg a megfelelő optimalizálási és az ahhoz tartozó döntési problémát, végül adjuk meg a döntési problémához tartozó nyelvet.

34.1-3. Adjuk meg az irányított gráfok bináris szavakként való kódolásainak egyikét, a szomszédsági mátrixok felhasználásával. Adjunk meg ilyen kódolást a szomszédsági listák segítségével is. Igazoljuk, hogy a két kódolás polinomiálisan kapcsolt.

34.1-4. Igaz-e, hogy a 0-1 hátizsák-problémára adható dinamikus programozási algoritmus (lásd 16.2-2. gyakorlat) polinomiális?

34.1-5. Mutassuk meg, hogy egy olyan (egyéb lépéseit tekintve polinomiális) algoritmus, mely konstansszor hív meg polinomiális szubrutinokat, polinomiális, míg ha polinomiális számú hívást engedünk meg, akkor az algoritmus exponenciális időt is igényelhet.

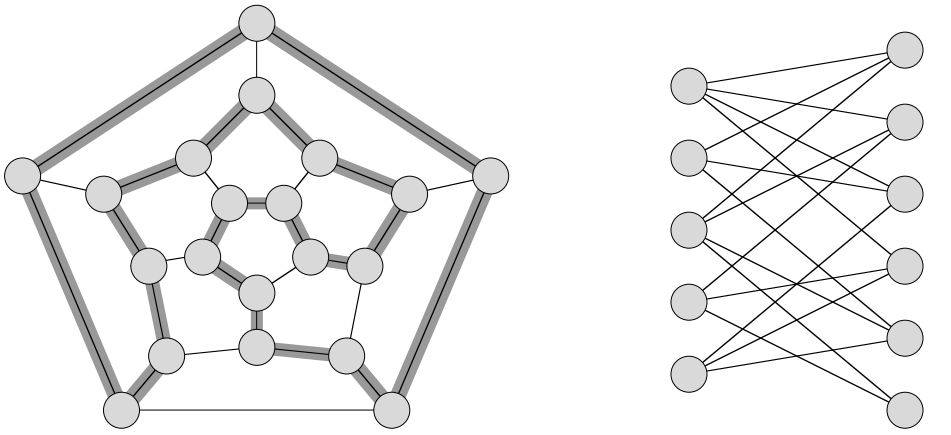
34.1-6. Igazoljuk, hogy a P osztály mint nyelvek egy halmaza, zárt az unióképzés, metszetképzés, konkatenáció, komplementálás és lezárás műveletekre.

34.2. Polinomiális idejű ellenőrzés

Ebben az alfejezetben olyan algoritmusokkal fogunk foglalkozni, amelyek azt „ellenőrzik”, hogy egy adott szó benne van-e egy adott nyelvben. Például tegyük fel, hogy az ÚT döntési probléma egy bizonyos (G, u, v, k) esetéhez ismerünk egy p utat is u-ból v-be. Könnyűszerrel ellenőrizhetjük, hogy p hossza legfeljebb k-e, és ha igen, akkor p-t egyfajta „tanúsítványnak” tekinthetjük, ami bizonyítja, hogy ÚT $((G, u, v, k)) = 1$. Az ÚT döntési probléma esetében ez a tanúsítvány nem tűnik igazán hasznosnak, hiszen ÚT P-ben van, s őt, lineáris időben megoldható, így a tanúsítvány ellenőrzése körülbelül ugyanannyi időt igényel, mint magának a problémának a megoldása. Az alábbiakban egy olyan problémával foglalkozunk, amelyre nem ismeretes polinomiális algoritmus, de egy adott tanúsítvány ellenőrzése könnyű.

Hamilton-körök

A Hamilton-kör keresésének problémáját irányítatlan gráfban már több, mint száz éve tanulmányozzák. Egy irányítatlan $G = (V, E)$ gráf ***Hamilton-körén*** olyan kört értünk, amely G minden pontján pontosan egyszer megy át. A Hamilton-kört tartalmazó gráfokat szokás ***hamiltoni,*** vagy Hamilton-gráfoknak nevezni, a Hamilton-kört nem tartalmazókat pedig – értelemszerűen – ***nem hamiltoni*** gráfoknak. Bondy és Murty [45] idézi W. R. Hamilton egy levelét, mely egy, a dodekaéderen (34.2(a) ábra) játszható matematikai játék leírását tartalmazza. Az egyik játékos kijelöl egy tetszőleges 5 hosszúságú utat, s a másiknak ezt kell az összes csúcsot tartalmazó körré kiegészítenie. Ebből is sejthető, hogy a dodekaédernek van Hamilton-köre, egy ilyen látható is a 34.2(a) ábrán. Természetesen nem minden gráf hamiltoni. Például, ha egy gráfban nincsen kör, akkor persze Hamilton-kör sincs. A 34.2(b) ábrán egy – köröket is tartalmazó – páros gráf látható, melynek páratlan sok csúcsa van. (A 34.2-2. gyakorlatban az Olvasónak kell belátnia, hogy az ilyen gráfok nem tartalmaznak Hamilton-kört.)



(a) (b)

34.2. ábra. (a) Dodekaéder gráfreprezentációja; a vastagított élek Hamilton-kört alkotnak. **(b)** Páros gráf, páratlan számú csúccsal. Az ilyen gráfoknak soha nincs Hamilton-körük.

A ***Hamilton-kör probléma*** (HAM): „Van-e a G gráfnak Hamilton-köre?”. Formális nyelvként definiálva:

$$\text{HAM} = \{\langle G\rangle : G \text{ Hamilton-gráf}\}.$$

Hogyan lehetne algoritmussal eldönteni a HAM nyelvet? Egy adott $\langle G\rangle$ esetre egy lehetséges algoritmus az, ha felsoroljuk G összes csúcsának minden lehetséges permutációját, és mindegyikről megnézzük, hogy Hamilton-kör-e. Mennyi időt igényel ez az algoritmus? Ha a gráf „ésszerű”, szomszédsági mátrixos kódolását használjuk, a csúcsok számára $m = \Omega(\sqrt{n})$ adódik, ahol $n = |\langle G\rangle|$ a kódolás hossza. A csúcsoknak $m!$ lehetséges permutációja van, így a futási idő $\Omega(m!) = \Omega((\sqrt{n})!) = \Omega(2^{\sqrt{n}})$, ami nem $O(n^k)$ egyetlen k konstanssal sem. Ez a „naiv” algoritmus tehát nem polinomiális, sőt a 34.5. alfejezetben látni fogjuk, hogy a Hamilton-kör probléma NP-teljes.

Ellenőrző algoritmusok

Foglalkozzunk most az előbbinél kicsivel egyszerűbb problémával. Tegyük fel, hogy egy jó barátunk megsúgja nekünk, hogy egy adott G gráfban van Hamilton-kör, és felajánlja, hogy bizonyítékul megad egyet. Ezt a bizonyítékot meglehetősen könnyű ellenőrizni: egyszerűen megvizsgáljuk, hogy a megadott pontsorozat permutációja-e G csúcsainak, és hogy az egymás utáni pontok (meg persze az első és az utolsó) szomszédosak-e. Ez az eljárás kényelmesen végrehajtható $O(n^2)$ lépésben, ahol n a G gráf kódolásának hossza. Ezek szerint a Hamilton-kör létezésének effajta bizonyítéka polinom időben ellenőrizhető.

Ellenőrző algoritmusnak olyan két bemenetű algoritmust nevezünk, ahol az egyik bemenet megegyezik a döntési algoritmusoknál megszokottal, azaz a probléma egy esetének (bináris) kódolása, a másik egy bináris szó, amit ***tanúnak*** nevezünk. Azt mondjuk, hogy az A ellenőrző algoritmus ***bizonyítja*** az x szót, ha létezik olyan y tanú, hogy $A(x, y) = 1$. A bizonyítja az L nyelvet, ha minden szavát bizonyítja, és minden szó, amit bizonyít, L-ben van. Röviden:

$$L = \{x \in \{0,1\}^* : \text{létezik } y \in \{0,1\}^*, \text{ amelyre } A(x,y) = 1\}.$$

Például HAM esetén a tanúk pontsorozatok (kódolásai) voltak, az ismertetett ellenőrző eljárásról pedig jól látható, hogy megfelel a kritériumoknak. Ha egy gráf nem tartalmaz Hamilton-kört, akkor nincs olyan csúcssorozat, amelynek alapján az ellenőrző algoritmus hibásan következtetne, mivel az algoritmus gondosan ellenőrzi a a javasolt „kört."

Az NP bonyolultsági osztály

Az **NP *bonyolultsági osztály*** azon nyelvek halmaza, melyek polinomiális algoritmussal bizonyíthatók.[6] Pontosabban: az L nyelv pontosan akkor van NP-ben, ha létezik két bemenetű polinomiális A algoritmus, és c konstans, hogy

$$L = \{x \in \{0,1\}^* : \text{létezik } y \in \{0,1\}^* \text{ tanú, melyre } |y| = O(|x|^c),\ A(x,y) = 1\}.$$

Ekkor azt mondjuk, hogy A ***polinom időben bizonyítja*** L-et.

A Hamilton-kör probléma eddigi vizsgálatából látható, hogy HAM $\in$ NP. (Mindig kellemes érzés megtudni, hogy egy fontos halmaz nem üres.) Ezen túlmenően, ha $L \in$ P, akkor $L \in$ NP is, hiszen ha L az A polinomiális algoritmussal eldönthető, akkor ebből az algoritmusból könnyen kaphatunk megfelelő ellenőrző algoritmust, ami a második bemenetet egyszerűen figyelmen kívül hagyja, és akkor ad az (x,y) párra 1-et, ha $A(x) = 1$. Ezzel beláttuk, hogy P $\subseteq$ NP.

Nem tudjuk azonban, hogy itt valódi tartalmazás áll-e fenn, azaz P $\subset$ NP, vagy P = NP; a kutatók többsége azonban úgy gondolja, hogy P és NP különböző osztályok. Intuitíve: P-ben a gyorsan megoldható problémák, NP-ben a gyorsan leellenőrizhető megoldású problémák vannak. Tapasztalhattuk, hogy többnyire lényegesen nehezebb egy problémát megoldani, mint egy adott megoldás helyességét ellenőrizni. Ez arra „utal", hogy lehetnek NP-ben olyan nyelvek, melyek P-ben nincsenek benne. Van jobb okunk is, hogy elhiggyük a P $\neq$ NP sejtést: az NP-teljes nyelvek létezése. Ezzel az osztállyal a 34.3. alfejezetben fogunk foglalkozni.

A témakörben a P $\neq$ NP sejtésen kívül is számos alapvető kérdés máig megoldatlan. A kutatók erőfeszítései ellenére senki sem tudja például, hogy az NP osztály zárt-e a komplementerképzésre. Definiáljuk a **co-NP *bonyolultsági osztályt*** a következőképp: $L \in$ co-NP pontosan akkor, ha $\overline{L} \in$ NP. NP komplementerképzésre való zártsága tehát azt jelentené, hogy NP = co-NP. Mivel P zárt a komplementálásra (34.1-6. gyakorlat), P $\subseteq$ NP $\cap$ co-NP. Itt sem tudjuk, hogy a tartalmazás valódi-e. A négy lehetséges változat a 34.3. ábrán látható.

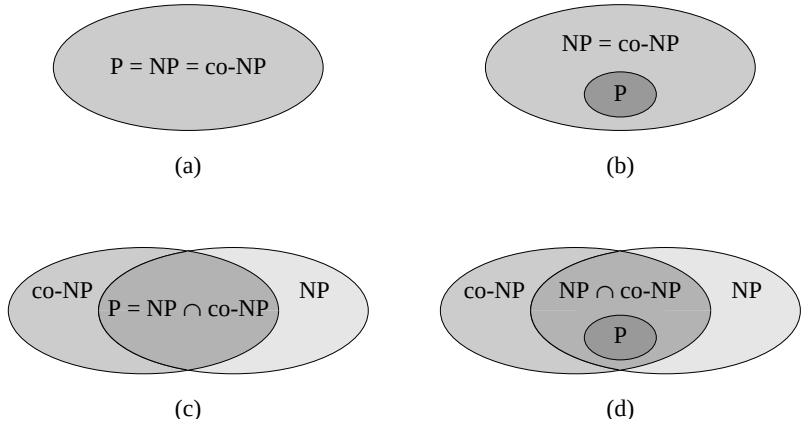
A P és NP közti kapcsolatról való tudásunk tehát elkeserítően hiányos. Mindazonáltal az NP-teljesség elméletének beható vizsgálata során látni fogjuk, hogy lemaradásunk a problémák kezelhetetlenségének bizonyításában gyakorlati szempontból nem is olyan nagy, mint gondolnánk.

Gyakorlatok

34.2-1. Mutassuk meg, hogy a GRÁF-IZOMORFIZMUS = $\{\langle G_1, G_2\rangle : G_1$ és G_2 izomorf gráfok$\}$ nyelv NP-ben van.

---

[6] NP a „nondeterministic polinomial (time)" (nemdeterminisztikus polinomiális (idejű)) rövidítése. Az NP osztályt eredetileg a nemdeterminizmussal összefüggésben vizsgálták (és definiálták); jelen könyv a némiképp egyszerűbb, ekvivalens „ellenőrzéses" megközelítést részesíti előnyben. Az NP-teljesség nemdeterminisztikus számítási modellel való tárgyalása megtalálható például Hopcroft és Ullmann [156] művében.



34.3. ábra. Az eddig megismert bonyolultsági osztályok közti négy lehetséges kapcsolat. **(a)** P = NP = co-NP. A kutatók többsége szerint ez a legkevésbé valószínű lehetőség. **(b)** NP zárt a komplementálásra, de P ≠ NP. **(c)** P = NP ∩ co-NP, de NP nem zárt a komplementálásra. **(d)** NP ≠ co-NP, és P ≠ NP ∩ co-NP. A tudósok többsége ezt tartja a legvalószínűbb esetnek.

34.2-2. Lássuk be, hogy ha egy páros gráfnak páratlan számú csúcsa van, akkor nem lehet Hamilton-köre.

34.2-3. Mutassuk meg, hogy ha HAM ∈ P, akkor polinom időben meg is lehet adni tetszőleges G gráf egy Hamilton-körét (ha van).

34.2-4. Bizonyítsuk be, hogy NP zárt az unióképzés, metszetképzés, konkatenáció és lezárás műveletekre. Gondolkozzunk el NP komplementálásra való zártságának kérdésén.

34.2-5. Mutassuk meg, hogy az NP-beli nyelvek eldönthetők $2^{O(n^k)}$ idejű algoritmussal, valamely k konstans mellett.

34.2-6. Egy G gráf ***Hamilton-útjának*** nevezzük az olyan egyszerű utakat, amelyek minden csúcsot pontosan egyszer érintenek. Mutassuk meg, hogy a HAM-ÚT = $\{\langle G, u, v\rangle$: van Hamilton-út u és v közt G-ben$\}$ nyelv NP-ben van.

34.2-7. Mutassuk meg, hogy a Hamilton-út probléma polinom időben megoldható irányított körmentes gráfokra. Adjunk minél gyorsabb algoritmust.

34.2-8. Legyen ϕ Boole-formula az $x_1, x_2, \ldots, x_n$ változókon. ϕ ***tautológia,*** ha a változók bármely behelyettesítésére 1 az értéke. Mutassuk meg, hogy TAU = $\{\langle\phi\rangle$: ϕ tautológia$\} \in$ co-NP.

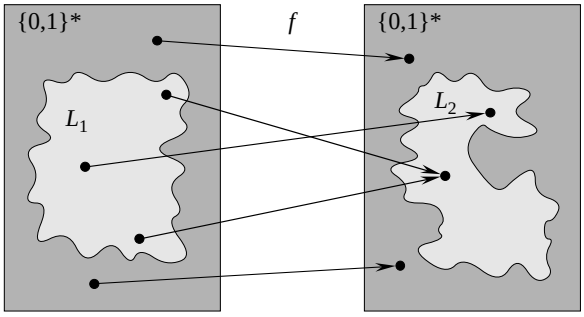
34.2-9. Bizonyítsuk be, hogy P ⊆ co-NP.

34.2-10. Bizonyítsuk be, hogy ha NP ≠ co-NP, akkor P ≠ NP.

34.2-11. Legyen G legalább 3 csúcsú összefüggő, irányítatlan gráf. Jelöljük G^3-nel azt a gráfot, melyet úgy kapunk, hogy összekötjük G azon csúcsait, melyek legfeljebb 3 hosszú úton elérhetők egymásból. Mutassuk meg, hogy G^3-ben van Hamilton-kör. (*Útmutatás.* Induljunk ki G egy feszítőfájából, és használjunk indukciót.)

34.3. NP-teljesség és visszavezethetőség

A P ≠ NP sejtés melletti legnyomósabb érv talán az NP-teljes problémák létezése. Ezen problémák rendelkeznek a következő meglepő tulajdonsággal: ha közülük akár csak egy



34.4. ábra. Az L_1 nyelv polinomiális visszavezetése L_2-re az f visszavezető függvény segítségével. Bármely $x \in \{0,1\}^*$ bemenetre $x \in L_1$ pontosan akkor, ha $f(x) \in L_2$.

is megoldható polinom időben, akkor *valamennyi* NP-beli probléma megoldható polinom időben, azaz P = NP. Mint arról már szó esett, az évtizedek óta folyó kutatások ellenére sem sikerült egyelőre polinomiális algoritmust találni egyetlen NP-teljes problémára sem.

A HAM NP-teljes probléma. Ha el tudnánk dönteni HAM-et polinomiális idő alatt, akkor minden NP-beli problémát meg tudnánk oldani polinomiális idő alatt. Ha kiderülne, hogy NP − P nem üres, akkor teljes bizonyossággal mondhatnánk, hogy HAM ∈ NP − P.

Az NP-teljes nyelvek bizonyos értelemben a „legnehezebb" nyelvek NP-ben. Ebben az alfejezetben megmutatjuk, hogyan hasonlíthatjuk össze a nyelvek „nehézségét", a „polinomiális idejű visszavezethetőség" segítségével. Először is definiáljuk az NP-teljes nyelvek halmazát, majd vázoljuk annak bizonyítását, hogy a C-SAT nevű nyelv ide tartozik. A 34.4. és 34.5. alfejezetekben pedig számos más probléma NP-teljességét bizonyítjuk a visszavezetés segítségével.

Visszavezethetőség

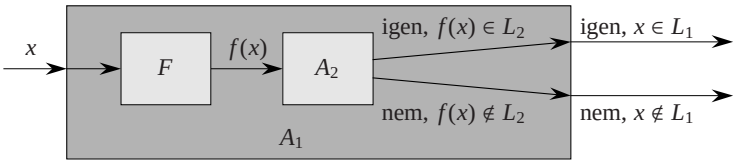
Intuitíve a P probléma visszavezethető Q-ra, ha P bármely x esete „könnyen átfogalmazható" Q egy olyan y esetévé, melynek megoldásából következik x megoldása. Például az elsőfokú egyismeretlenes egyenlet megoldásának problémája visszavezethető a másodfokú egyenlet megoldására. Az $ax + b = 0$ egyenletet ugyanis a $0y^2 + ay + b = 0$ egyenletté alakíthatjuk, s ennek megoldása kielégíti $ax + b = 0$-t is. Ez azt is jelenti, hogy ha P visszavezethető Q-ra, akkor P-t bizonyos értelemben „nem nehezebb megoldani", mint Q-t.

A döntési problémák nyelvelméleti tárgyalásához visszatérve, azt mondjuk, hogy az L_1 nyelv ***polinomiálisan visszavezethető*** az L_2 nyelvre (jelölése: $L_1 \leq_P L_2$), ha létezik $f : \{0,1\}^* \to \{0,1\}^*$ polinom időben kiszámítható függvény, melyre minden $x \in \{0,1\}^*$ esetén

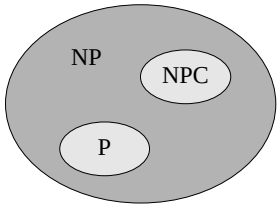
$$x \in L_1 \text{ pontosan akkor, ha } f(x) \in L_2. \tag{34.1}$$

Az f függvényt ***visszavezető függvénynek*** hívjuk, az f-et kiszámító polinomiális algoritmust pedig ***visszavezető algoritmusnak.***

A 34.4. ábrán látható egy L_1 nyelv L_2-re való visszavezetésének illusztrációja. A nyelvek természetesen $\{0,1\}^*$ részhalmazai, és jól látható, hogy az L_1-beli x-ek $f(x)$ képe L_2-ben van, az L_1-en kívülieké pedig L_2-n kívül. A visszavezetés fenti definíciója tehát megfelel intuitív elképzeléseinknek, az L_1 által reprezentált probléma eseteit megfeleltettük az L_2-vel reprezentált probléma eseteinek úgy, hogy az utóbbiakra kapott megoldás meghatározza az



34.5. ábra. A 34.3. lemma bizonyítása. F az L_1-et L_2-re visszavezető függvényt kiszámító polinomiális algoritmus, A_2 az L_2 nyelvet eldöntő polinomiális algoritmus. A képen látható A_1 algoritmus először előállítja $f(x)$-et x-ből F segítségével, majd A_2 segítségével eldönti, hogy $f(x) \in L_2$ vagy sem, ily módon eldöntve az L_1 nyelvet is.



34.6. ábra. A legtöbb számítógéptudós így képzeli el a P, NP és NPC osztályok viszonyát.

előbbiek megoldásait.

A visszavezetési módszer jól használható nyelvek polinomialitásának bizonyítására.

34.3. lemma. *Ha* $L_1, L_2 \subseteq \{0,1\}^*$, $L_1 \leq_P L_2$, *és* $L_2 \in$ P, *akkor* $L_1 \in$ P.

Bizonyítás. Legyen A_2 polinomiális algoritmus, ami eldönti L_2-t, és legyen F az L_1-et L_2-re visszavezető f függvényt kiszámító polinomiális algoritmus. Megadunk egy L_1-et polinom időben eldöntő A_1 algoritmust.

A_1 konstrukcióját a 34.5. ábra szemlélteti. Az $x \in \{0,1\}^*$ bemenetből A_1 először F segítségével előállítja $f(x)$-et, majd A_2 felhasználásával eldönti,hogy $f(x)$ L_2-ben van-e, és az erre kapott választ adja ki eredményként.

Az, hogy A_1 helyes választ ad, a (34.1.) feltételből következik. A_1 polinomiális, hiszen F és A_2 is az. (Lásd a 34.1-5. gyakorlatot.) ■

NP-teljesség

A polinomiális visszavezetés arra is kiválóan felhasználható, hogy belássuk: egy adott probléma legalább olyan nehéz, mint egy másik, legalábbis polinom tényező erejéig. Ha ugyanis $L_1 \leq_P L_2$, akkor L_1 legfeljebb polinomszor „nehezebb” L_2-nél, ami megmagyarázza a visszavezethetőségre használt $\leq_P$ jelölést. Most már készen állunk az NP-teljes nyelvek definiálására, amelyek a legnehezebb NP-beli nyelvek.

Az $L \subseteq \{0,1\}^*$ nyelv ***NP-teljes,*** ha

1. $L \in$ NP és
2. Minden $L' \in$ NP-re $L' \leq_P L$.

Ha egy nyelv rendelkezik a második tulajdonsággal, de az elsővel nem feltétlenül, akkor ***NP-nehéznek*** nevezzük. Az NP-teljes nyelvek halmazát NPC-vel jelöljük.

Mint a következő tétel mutatja, az NP-teljességnek döntő szerepe van P és NP viszonyának eldöntésében.

34.4. tétel. *Ha létezik polinomiális időben megoldható NP-teljes probléma, akkor* P = NP. *Más szóval: ha létezik* NP*-ben polinom időben nem megoldható probléma, akkor egyetlen* NP*-teljes probléma sem polinomiális.*

Bizonyítás. Tegyük fel, hogy $L \in \mathrm{P} \cap \mathrm{NPC}$. Tudjuk, hogy bármely $L' \in \mathrm{NP}$ esetén $L' \leq_\mathrm{P} L$, az NP-teljesség definíciójának 2. feltétele miatt. Mivel $L \in \mathrm{P}$, a 34.3. lemma szerint $L' \in \mathrm{P}$, ezzel az első állítást beláttuk. A második állítás nyilván ekvivalens az elsővel. ■

Érthető tehát, hogy a P ≠ NP sejtéssel kapcsolatos kutatások középpontjában az NP-teljes problémák állnak. A számítógéptudósok többsége úgy gondolja, hogy P ≠ NP, ami az iménti tétel szerint a P, NP és NPC osztályok 34.6. ábra szerinti viszonyát jelenti. Jelenlegi tudásunk szerint persze az sem elképzelhetetlen, hogy valaki előáll egy NP-teljes probléma polinomiális megoldásával, bizonyítva ezzel, hogy P = NP. Minthogy ilyesmi eddig nem történt, és nem is igen várható, egy probléma NP-teljessége jó ok annak feltételezésére, hogy a probléma nehezen kezelhető.

Boole-hálózatok kielégíthetősége

Most már tudjuk, hogy mit is értünk NP-teljes problémán, de nem tudjuk, hogy ilyen egyáltalán létezik-e. Amint egy problémáról belátjuk, hogy NP-teljes, a polinomiális visszavezetést használva már számos más probléma NP-teljességét is igazolni tudjuk. Elsőként tehát mutatnunk kell egy NP-teljes problémát. Ez a Boole-hálózatok kielégíthetőségének problémája (circuit-satisfiability, C-SAT) lesz.

A formális bizonyítás sajnos olyan eszközöket igényel, amelyek meghaladják e fejezet lehetőségeit. Ehelyett egy olyan informális bizonyítást adunk, mely csak a Boole-hálózatokkal kapcsolatos alapvető ismeretekre épít.

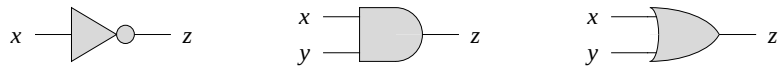
A Boole-hálózatok huzallal összekötött elemekből állnak. A ***logikai áramkör*** a hálózat olyan eleme, amelynek konstans számú be-, illetve kimenete van, és valamilyen jól meghatározott függvényt számít ki. A Boole-változók lehetséges értékei a {0, 1} halmaz elemei, ahol a 0 jelenti a HAMIS, az 1 pedig az IGAZ értéket.

Azok a logikai áramkörök, amiket a C-SAT probléma esetében használni fogunk, az úgynevezett ***logikai kapuk,*** egyszerű Boole-függvényeket számítanak ki. A 34.7. ábrán látható a három alapvető logikai kapu: a ***NEM kapu,*** vagy ***megfordító***, az ***ÉS kapu*** és a ***VAGY kapu.*** A NEM kapunak egy *x* bináris bemenete és egy *y* bináris kimenete van, a 0 bemenetre 1-et, az 1 bemenetre 0-t ad kimenetként. A másik két kapunak két bináris bemenete (*x* és *y*) és egy *z* bináris kimenete van.

Ezen logikai kapuk működését (csakúgy, mint bármely kapu működését) az ***igazságtáblázat*** segítségével írhatjuk le, melyet a 34.7. ábrán az egyes kapuk alatt találunk. Az igazságtáblázat megadja a kapu kimenetét a bemenetek minden lehetséges értékére. Például a VAGY kapu igazságtáblázata elárulja nekünk, hogy ha a bemenetek $x = 0$ és $y = 1$, akkor a kimenet $z = 1$. A NEM függvényt a $\neg$, az ÉS függvényt a $\wedge$, a VAGY függvényt a $\vee$ szimbólummal jelöljük. Így például $0 \vee 1 = 1$.

Az ÉS és VAGY kapukat általánosíthatjuk, hogy kettőnél több bemenetet is elfogadjanak. Az általánosított ÉS kapu kimenete pontosan akkor 1, ha minden bemenete 1, egyébként 0. Az általánosított VAGY kapu kimenete pontosan akkor 1, ha legalább egy bemenete 1, egyébként 0.

A ***Boole-hálózatok*** kapukból állnak, melyeket ***huzalok*** kötnek össze. Egy huzal össze-



x	$\neg x$
0	1
1	0

(a)

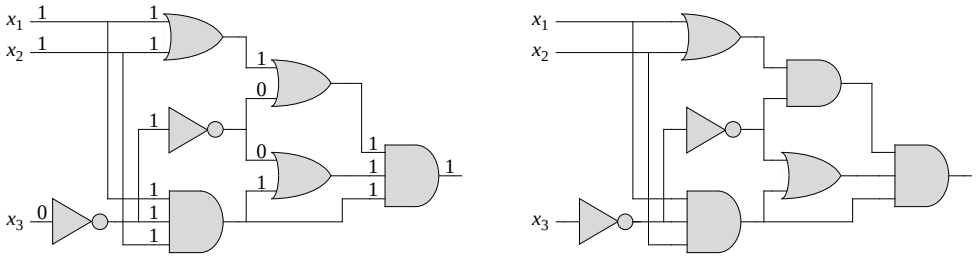
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

34.7. ábra. A három alapvető logikai kapu. A kapuk alatt találhatók azőket leíró igazságtáblázatok. **(a)** A NEM kapu. **(b)** Az ÉS kapu. **(c)** A VAGY kapu.



(a) (b)

34.8. ábra. A C-SAT probléma két esete. **(a)** Az $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ bemenetekre az eredmény 1, a hálózat tehát kielégíthető. **(b)** A bemenetek semmilyen értékére sem lesz 1 az eredmény, vagyis ez a hálózat nem kielégíthető.

kötheti egy kapu kimenetét egy másik kapu bemenetével, így az első kapunál megjelenő kimeneti érték lesz a második kapu egyik bemeneti értéke. A 34.8. ábrán két olyan Boole-hálózat látható, melyek csak egyetlen kapuban különböznek. Az ábra (a) részén az egyes huzalokon megjelenő értékek is láthatók az $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ bemenet mellett. Egy huzal természetesen legfeljebb egy kimenethez csatlakozhat, a hozzá csatlakozó bemenetek száma azonban tetszőleges nemnegatív egész lehet. Ez utóbbi számot a huzal ***ki-fokának*** hívjuk. Ha egy huzal nem kapcsolódik kimenethez, akkor ***hálózati bemenetnek*** hívjuk, míg ha a ki-foka 0, akkor a neve ***hálózati kimenet.*** A hálózati kimenetek továbbítják a hálózat számításainak eredményeit a külvilágba. (Az is elképzelhető ugyanakkor, hogy egy belső huzalon lévő érték is hozzáférhető a külvilág számára.) A C-SAT probléma definiálásához korlátozzuk most a huzalok ki-fokát, legyen minden kifok legfeljebb 1.

A Boole-hálózatok nem tartalmaznak köröket. Más szóval, készítsünk minden hálózathoz egy irányított gráfot a következőképp: legyenek a kapuk a gráf csúcsai, a k ki-fokú huzaloknak pedig feleljen meg k irányított él, oly módon, hogy az u kapunak megfelelő csúcsból a v kapunak megfelelő csúcsba mutasson él, ha a huzal kimenete u-nak és bemenete v-nek; az így kapott gráfnak kell körmentesnek lennie.

Egy Boole-hálózat ***behelyettesítése*** Boole-változók egy, a bemenetek számával megegyező elemszámú sorozata. Azt mondjuk, hogy egy egy kimenetű Boole-hálózat ***kielégíthető,*** ha van ***kielégítő behelyettesítése,*** azaz olyan behelyettesítése, melyre a kimenet 1.

Például, a 34.8(a) ábrán látható hálózat az $x_1 = 1,\ x_2 = 1,\ x_3 = 0$ bemenetekre 1-et ad kimenetként, tehát kielégíthető. A 34.3-1. gyakorlatban viszont azt kell megmutatnunk, hogy a 34.8(b) ábrán látható hálózat az x_1, x_2, x_3 változók egyetlen behelyettesítésére sem ad 1-et, tehát nem kielégíthető.

A ***Boole-hálózatok kielégíthetőségének problémája:*** „Adott – ÉS, VAGY és NEM kapukból álló – Boole-hálózat kielégíthető-e?”. A kérdés formalizálásához meg kell állapodnunk a hálózatok valamilyen szabványos kódolásában. A gráfokéhoz hasonló kódolás alkalmas lesz (a hálózat tulajdonképpen nem más, mint egy speciális, irányított gráf), ekkor $\langle C \rangle$ hossza nem sokkal nagyobb, mint a C hálózat mérete. Most már definiálhatjuk a

$$\text{C-SAT} = \{\langle C \rangle : \ C \text{ kielégíthető Boole-hálózat}\}$$

formális nyelvet.

A C-SAT probléma igen jelentős szerepet játszik a hardver-optimalizációban. Ha ugyanis egy hálózat minden bemenetre 0-t ad, akkor helyettesíthető egy konstans 0-t adó kapuval, időt és erőforrásokat takarítva meg. Egy polinomiális algoritmusnak tehát számottevő gyakorlati haszna lenne.

Ha adott a C hálózat, megpróbálhatjuk a kielégíthetőség kérdését úgy eldönteni, hogy minden lehetséges bemenetre kiszámítjuk az eredményt. k bemenet esetén a lehetséges behelyettesítések száma 2^k. Ha például C mérete k polinomja. akkor minden eset ellenőrzése $\Omega(2^k)$ időt igényel, azaz szuperpolinomiális az áramkör méretében.[7] Persze, mint arról már szót ejtettünk, C-SAT-ra valószínűleg nincs is polinomiális algoritmus, hiszen C-SAT NP-teljes. A bizonyítás két lépésben történik, először a definíció első, majd a második feltételét igazoljuk.

34.5. lemma. C-SAT $\in$ NP.

Bizonyítás. Megadunk egy C-SAT-ot ellenőrző kétbemenetű algoritmust. Az A algoritmus egyik bemenete a C hálózat (szabványos kódolása), a másik egy t 0-1 sorozat, ami a huzalokon szereplő értékeknek felel meg. (Aki rövidebb tanúsítványt szeretne látni, oldja meg a 34.3-4. gyakorlatot.)

Az algoritmus minden kapura ellenőrzi, hogy az adott bemenetekhez az adott kimenet helyesen van-e kiszámítva, majd, ha a hálózat kimenete 1, akkor 1-et ad kimenetként, hiszen ekkor a hálózat bemenetei kielégítő behelyettesítést adnak. Minden más esetben az algoritmus 0-t ad kimenetként.

Ha C kielégíthető, akkor létezik C hosszúságában polinomiális hosszúságú t sorozat, amire $A(C, t) = 1$. Ha viszont C nem kielégíthető, akkor ez egyetlen t sorozatra sem teljesülhet. Az A algoritmus nyilván polinomiális (akár lineáris időben is megvalósítható). Mindezek együttesen azt jelentik, hogy az A algoritmus polinomiális tanúval, polinom időben ellenőrzi a C-SAT nyelvet, azaz C-SAT $\in$ NP. ■

[7] Másfelől, ha C mérete $\Theta(2^k)$, akkor egy $O(2^k)$ futásidejű algoritmus polinomiális a hálózat méretében. Ez még akkor sem mond ellent C-SAT NP-teljességének, ha P $\neq$ NP: egy speciális esetre adott polinomiális algoritmus nem jelenti azt, hogy létezik polinomiális algoritmus minden esetre.

A következő lépés annak bizonyítása, hogy C-SAT NP-nehéz, azaz minden NP-beli nyelv polinomiálisan visszavezethető rá. A tényleges bizonyítás – mely a számítógépek működésének alapelveire épül – számos technikai bonyodalmat tartalmaz, ezért vázlatos ismertetésére szorítkozunk.

A számítógépes programok a gép memóriájában vannak elhelyezve, utasítások sorozataiként. Egy tipikus utasítás a következő három dolog kódolása: egy végrehajtandó művelet, a művelet argumentumainak címei a memóriában és az a memóriacím, ahová az eredményt kell írni. Egy speciális memóriahely, az úgynevezett ***programszámláló*** nyomon követi, hogy melyik utasítás következik. A programszámláló automatikusan növekszik, valahányszor egy utasítás végrehajtódik, ami azt eredményezi, hogy a számítógép a műveleteket sorban egymás után hajtja végre. Egy művelet a programszámláló átírását is okozhatja, így a végrehajtás módosulhat, lehetővé téve ezzel ciklusok futását és a feltételes elágazásokat.

A program futása során a számítás pillanatnyi állapotát a gép memóriájában tároljuk. A memória tartalmazza magát a programot, a programszámlálót, a munkaterületet és a számtalan állapotjelző bitet, amiket a számítógép fenntart a „könyveléshez”. A memória egyes állapotait ***konfigurációknak*** hívjuk. Egy utasítás végrehajtását tekinthetjük olyan leképezésnek, amely egy konfigurációt egy másikba visz. Fontos, hogy a hardver, ami ezt a leképezést végrehajtja, megvalósítható Boole-hálózatként, amit M-mel fogunk jelölni a következő lemmában.

34.6. lemma. C-SAT NP-*nehéz.*

Bizonyítás. Legyen L tetszőleges NP-beli nyelv. Megadunk egy F polinomiális algoritmust, mely az alábbi f visszavezető függvényt számítja ki: minden $x \in \{0,1\}^*$ szó képe egy olyan $f(x) = \langle C \rangle$ kódolt hálózat, amelyre $x \in L$ pontosan akkor teljesül, ha $\langle C \rangle \in$ C-SAT.

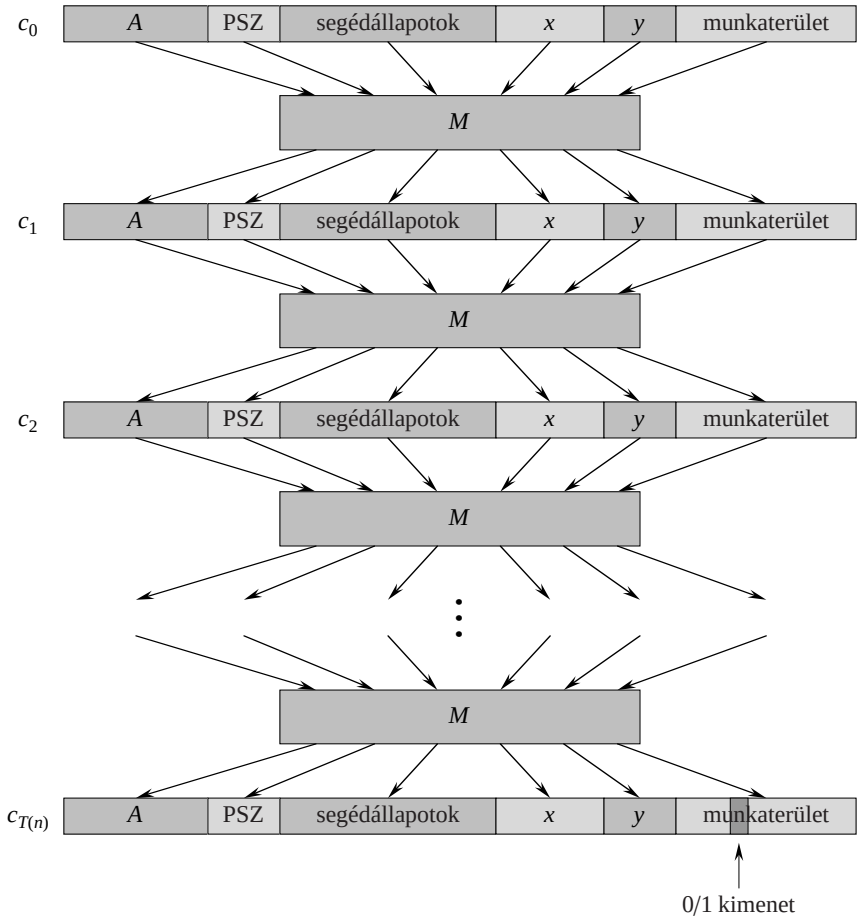
$L \in$ NP, így létezik olyan algoritmus, mely L-et polinom időben ellenőrzi. Megkonstruálunk egy olyan F algoritmust, amely a kétbemenetű A algoritmust fogja felhasználni az f visszavezetőfüggvény kiszámítására.

Jelöljük $T(n)$-nel A legrosszabb futási idejét az n hosszúságú bemeneti sztringeken, és legyen $k \geq 1$ olyan szám, melyre igaz, hogy $T(n) = O(n^k)$ és a tanú hossza is $O(n^k)$. (A futási ideje a két bemenet együttes hosszában polinomiális, de mivel a tanú maga is polinom n-ben, ami egy adott eset kódolásának hossza, a futási idő n-ben is polinomiális.)

A bizonyítás alapgondolata, hogy A számításait konfigurációk sorozataként jelenítjük meg. Amint az a 34.9. ábrán látható, minden egyes konfiguráció az alábbi részekre bontható: az A-nak megfelelő program, a programszámláló, a segédállapotok, az x bemenet (egy eset kódolása), az y tanú és a munkaterület. A c_0 kezdeti konfigurációból indulva, a c_i konfigurációkat c_{i+1}-be képezzük, az M Boole-hálózat segítségével, amely a hardver tevékenységét reprezentálja. Az algoritmus kimenete – 0 vagy 1 – a munkaterület egy kijelölt helyén található, amikor A futása befejeződik (legfeljebb $T(n)$ lépés), így az eredmény $c_{T(n)}$ egy kijelölt bitjével lesz azonos.

Az F algoritmus létrehoz egy olyan Boole-hálózatot, amely egy adott kezdő konfigurációhoz tartozó összes konfigurációt kiszámítja az alábbi módon. Az M hálózat $T(n)$ darab példányát összefűzzük, ami azt jelenti, hogy az i-edik hálózat kimenetét (azaz a c_i konfigurációt) betápláljuk az $(i+1)$-edikbe.

Mi is az F algoritmus feladata? Egy adott x bemenetre olyan $f(x) = C$ hálózatot kell adnia, amely pontosan akkor kielégíthető, ha létezik y tanú, melyre $A(x,y) = 1$. F először kiszámítja a bemenet $n = |x|$ hosszát, majd létrehozza a C' hálózatot, mely az M hálózat $T(n)$



34.9. ábra. Az A algoritmus (x, y) bemeneten való futása során kapott konfigurációk sorozata. A c_0 konfiguráció az y tanút leszámítva állandó. A konfigurációkat az M Boole-hálózat képezi a következő konfigurációba. Az eredmény a munkaterület egy kitüntetett bitje.

darab összekapcsolt példányából áll. C' bemenete az $A(x, y)$ számításához tartozó kezdő konfiguráció, kimenete pedig a $C_{T(n)}$ konfiguráció.

Az F által konstruálandó $C = f(x)$ hálózatot C' kis módosításával kapjuk. Először is C'-nek az A-t megvalósító programhoz tartozó bemeneteit – a programszámlálót, a kezdő-állapotokat és az x bemenetet – beállítjuk a megfelelő ismert értékekre. Hálózatunk „igazi" bemenetei tehát valamennyien az y tanúhoz tartoznak. Másodszor, a hálózat kimeneteit a $C_{T(n)}$ bit kivételével figyelmen kívül hagyjuk. Az ily módon kapott C hálózat minden $O(n^k)$ hosszúságú y tanúhoz kiszámolja $C(y) = A(x, y)$-t.

Meg kell mutatnunk, hogy F visszavezető függvényt számít ki (azaz, hogy C pontosan akkor kielégíthető, ha létezik y tanú, melyre $A(x, y) = 1$), és hogy ez a számítás polinom időben véget ér.

Tegyük fel először, hogy létezik $O(n^k)$ hosszúságú y tanú, melyre $A(x, y) = 1$. Adjuk most y bitjeit C-nek bemenetként, ekkor $C(y) = A(x, y) = 1$. Ha tehát a megfelelő tanú

létezik, akkor C kielégíthető. Tegyük fel mármost, hogy C kielégíthető. Ez azt jelenti, hogy létezik y, melyre $C(y) = 1$, így $A(x, y) = C(y) = 1$. F tehát valóban visszavezető függvényt számít ki.

Hátravan még annak bizonyítása, hogy F polinomiális $|x| = n$-ben. Vegyük észre, hogy a konfigurációk leírásához n-ben polinomiális számú bit elég, hiszen az A-t megvalósító program konstans méretű, a bemenet mérete n, az y tanú $O(n^k)$ hosszú és a munkaterület is polinomiális számú bitet használ, mivel az algoritmus $O(n^k)$ lépésben véget ér. (Feltesszük, hogy a memória összefüggő; a 34.3-5. gyakorlatban ki kell terjesztenünk a bizonyítást arra az esetre, amikor a program által elérni kívánt címek a memória egy nagyobb részén vannak szétszórva, és ez a szétszórás bemenetenként különböző lehet.)

A hardver működését megvalósító M hálózat polinomiális nagyságú a konfigurációk méretében, így n-ben is. A C hálózat M-nek legfeljebb $t = O(n^k)$ darab példányából áll, így szintén polinomiális méretű n-ben és konstrukciója polinomiális számú lépést igényel. (Ennek a hálózatnak a nagyobb része a memóriarendszert valósítja meg.) Az x bemenetből $f(x) = C$-t előállító F visszavezető algoritmus tehát polinomiális, hiszen a konstrukció minden lépése megvalósítható polinom időben. ■

A C-SAT nyelv ezek szerint legalább olyan nehéz, mint bármely NP-beli nyelv, és mivel NP-hez tartozik, NP-teljes.

34.7. tétel. *A* C-SAT *nyelv* NP-*teljes.*

Bizonyítás. A 34.5. és 34.6. lemmák és az NP-teljesség definíciójának közvetlen következménye. ■

Gyakorlatok

34.3-1. Igazoljuk, hogy a 34.8(b) ábrán látható hálózat nem kielégíthető.

34.3-2. Mutassuk meg, hogy a nyelveken értelmezett $\leq_P$ reláció tranzitív, azaz ha $L_1 \leq_P L_2$ és $L_2 \leq_P L_3$, akkor $L_1 \leq_P L_3$.

34.3-3. Bizonyítsuk be, hogy $L \leq_P \overline{L}$ akkor és csak akkor, ha $\overline{L} \leq_P L$.

34.3-4. Mutassuk meg, hogy egy kielégítő behelyettesítés felhasználható tanúként a 34.5. lemma egy másik bizonyításához. Az új vagy a már látott tanú esetén kapunk egyszerűbb bizonyítást?

34.3-5. A 34.6. lemma bizonyításában feltettük, hogy a munkaterület a memória egy összefüggő, polinomiális méretű része. Hol használtuk fel a bizonyítás során ezt? Mutassuk meg, hogy e feltevés nem megy az általánosság rovására.

34.3-6. Az L nyelv ***teljes*** a C nyelvosztályra nézve (a polinomiális visszavezetés tekintetében), ha $L \in C$ és minden $L' \in C$ esetén $L' \leq_P L$. Mutassuk meg, hogy az üres halmaz, és $\{0, 1\}^*$ az egyedüli P-beli nyelvek, amelyek nem teljesek P-re.

34.3-7. Lássuk be, hogy L pontosan akkor teljes NP-re nézve, ha $\overline{L}$ teljes co-NP-re nézve.

34.3-8. A 34.6. lemma bizonyításában szereplő F algoritmus létrehoz egy $C = f(x)$ hálózatot az x, az A és a k ismeretében. Sartre professzor megfigyelte, hogy az x szó valóban bemenete F-nek, de F A-nak és k-nak csupán a létezéséről tud, konkrétan nem ismeri őket. Ebből azt a következtetést vonja le, hogy F nem lehet képes C megkonstruálására, tehát a C-SAT nyelv nem feltétlenül NP-nehéz. Magyarázzuk el neki, hol a hiba az okoskodásában.

34.4. NP-teljességi bizonyítások

C-SAT NP-teljességét közvetlenül a definícióból láttuk be, azaz minden NP-beli L nyelvre igazoltuk, hogy $L \leq_P$ C-SAT. Ebben az alfejezetben megmutatjuk, hogyan látható be nyelvek NP-teljessége anélkül, hogy minden egyes NP-beli nyelvről igazolnánk, hogy visszavezethető az adott nyelvre. A technika illusztrációjaként két, a Boole-formulák kielégíthetőségével kapcsolatos problémáról látjuk be, hogy NP-teljes. A 34.5. alfejezetben számos más példát is látunk majd.

Az alábbi lemma minden további NP-teljességi bizonyítás alapja.

34.8. lemma. *Ha L olyan nyelv, melyhez létezik $L' \in$ NPC, hogy $L' \leq_P L$, akkor L* NP*-nehéz. Ha $L \in$ NP is teljesül, akkor $L \in$* NPC.

Bizonyítás. Mivel $L' \in$ NPC, ezért minden $L'' \in$ NP-re igaz, hogy $L'' \leq_P L'$. Ha tehát $L' \leq_P L$, akkor a tranzitivitás miatt (34.3-2. feladat) $L'' \leq_P L$ minden $L'' \in$ NP nyelvre, vagyis L NP-nehéz. Ha $L \in$ NP is, akkor persze $L \in$ NPC. ■

Más szavakkal, egy ismert L' NP-teljes probléma visszavezetése L-re közvetve minden NP-beli problémát visszavezet L-re. A 34.8. lemma szerint tehát a következő módszerrel bizonyíthatjuk egy L nyelv NP-teljességét.

1. Belátjuk, hogy $L \in$ NP.
2. Választunk egy ismert L' NP-teljes nyelvet.
3. Megadunk egy F algoritmust, mely kiszámít egy olyan f függvényt, ami L' eseteinek L eseteit felelteti meg.
4. Bebizonyítjuk, hogy tetszőleges $x \in \{0,1\}^*$ esetén $x \in L'$ akkor és csak akkor teljesül, ha $f(x) \in L$.
5. Igazoljuk, hogy F polinomiális.

(A 2–5. lépések bizonyítják azt, hogy L NP-nehéz.) Ez a módszer természetesen jóval egyszerűbb, mint az NP-teljesség közvetlen igazolása. Azzal, hogy C-SAT $\in$ NPC-t bebizonyítottuk, megtettük a legfontosabb lépést; az NP-teljességi bizonyítások ezentúl lényegesen könnyebbek lesznek. Sőt, ahogy egyre több problémáról látjuk be, hogy NP-teljes, úgy lesz egyre könnyebb dolgunk, hiszen módszerünk 2. lépésénél az NP-teljes problémák egyre népesebb táborából válogathatunk.

Boole-formulák kielégíthetősége

A visszavezetési technika első alkalmazásaként belátjuk, hogy a Boole-formulák kielégíthetőségének problémája (satisfiability problem, SAT) NP-teljes. E problémának történelmi jelentősége (is) van, róla bizonyították elsőként az NP-teljességet.

A ***(formula) kielégíthetőségi problémát*** a SAT nyelv segítségével fogalmazzuk meg. SAT esetei a ϕ Boole-formulák, melyek az alábbi elemekből épülnek fel:

1. $x_1, x_2, x_3, \ldots$ Boole-változók;
2. egy- vagy kétváltozós Boole-függvények, mint például $\wedge$ (ÉS), $\vee$ (VAGY), $\neg$ (NEM), $\rightarrow$ (implikáció), $\leftrightarrow$ (ekvivalencia);

3. zárójelek. (Anélkül, hogy ez az általánosság rovására menne, feltehetjük, hogy nincsenek felesleges zárójelek, azaz minden Boole-függvényhez legfeljebb egy zárójelpár tartozik.)

A ϕ Boole-formulát könnyűszerrel kódolhatjuk olyan hosszúságban, mely $n + m$ polinomja. Az olyan Boole-formulákat, melyeknek csak az elején és a végén van zárójel, ***klózoknak*** nevezzük. Csakúgy, mint a Boole-hálózatoknál, ϕ egy behelyettesítésének egy megfelelő hosszúságú 0-1 sorozatot nevezünk. Egy behelyettesítés kielégítő, ha az elemeit rendre az $x_1, x_2, \ldots$ változók helyébe írva ϕ értéke 1. Egy formula ***kielégíthető,*** ha létezik kielégítő behelyettesítése. A SAT probléma: „Egy adott Boole-formula kielégíthető-e vagy sem?”. Formális nyelvként:

$$\text{SAT} = \{\langle\phi\rangle : \ \phi \text{ kielégíthető Boole-formula}\}.$$

Vegyük például a

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

formulát; ezt az $(x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1)$ behelyettesítés kielégíti, hiszen

$$\begin{aligned} \phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1, \end{aligned} \tag{34.2}$$

így $\langle\phi\rangle \in$ SAT.

Ugyanúgy, mint C-SAT esetében, az összes eset naiv végigpróbálása 2^n lépést igényel n változó esetén, így ha $\langle\phi\rangle$ mérete polinomiális n-ben, akkor minden behelyettesítés $\Omega(2^n)$ lépést igényel, azaz szuperpolinomiális $\langle\phi\rangle$ méretének függvényében. A következő tétel szerint polinomiális algoritmus nem is igen várható SAT-ra.

34.9. tétel. SAT NP-*teljes.*

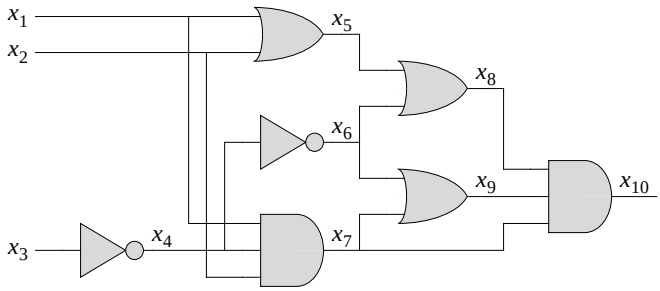
Bizonyítás. Először belátjuk, hogy SAT $\in$ NP, majd igazoljuk, hogy C-SAT $\leq_P$ SAT, amiből a 34.8. lemma szerint a tétel következik.

SAT $\in$ NP bizonyításához megmutatjuk, hogy a ϕ formula egy kielégítő behelyettesítése mint tanú, polinom időben ellenőrizhető. Az ellenőrző algoritmus egyszerűen minden változó helyére beírja a megfelelő értéket, és a kapott kifejezést kiszámítja, nagyjából úgy, ahogy azt (34.2) esetében tettük. Ez az eljárás könnyűszerrel elvégezhető polinom időben. Ha a kifejezés értéke (valamely tanúra) 1, akkor a formula kielégíthető, különben nem.

Ezzel SAT $\in$ NP-t beláttuk, térjünk rá C-SAT-ra való visszavezetésére. A hálózatok tulajdonképpen könnyen formulákká alakíthatók: tekintjük a hálózat kimenetét szolgáltató kaput, és felírjuk a neki megfelelő műveletet a bemeneteire, amiket rekurzívan, ugyanezen módszerrel alakítunk formulákká.

Sajnos, ez a kézenfekvő módszer nem mindig lesz polinomiális: a 34.4-1. gyakorlatban azt kell megmutatnunk, hogy olyan kapuk, melyek kimenete legalább kettő, a kapott formula méretének exponenciális növekedését okozhatják. Így tehát valamilyen ügyesebb visszavezetést kell találnunk.

A 34.10. ábrán egy ilyen jobb visszavezetés alapgondolata látható, a 34.8(a) ábra hálózatára alkalmazva. A C hálózat minden huzaljának (azaz a bemeneteknek és a kapuk



34.10. ábra. C-SAT visszavezetése SAT-ra. A visszavezető algoritmus a hálózat bemenetein kívül a hálózat minden kapujának kimenetéhez is a formula különböző változóit rendeli.

kimeneteinek) megfeleltetünk egy x_i változót. A kapuk működése ekkor leírható a hozzájuk csatlakozó huzaloknak megfelelő változók alkalmas formulájaként. Például az ábrán látható hálózat kimeneti ÉS kapuját leíró formula: $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

A teljes hálózatnak megfelelő formula a kimeneti kapuhoz tartozó változó és az egyes kapukat leíró formulák konjunkciója. Például a 34.10. ábra hálózatára:

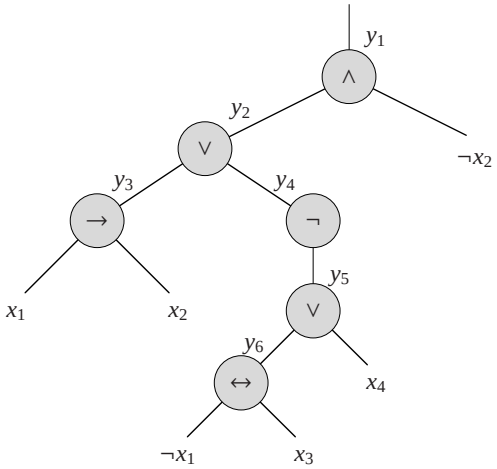
$$\begin{aligned}\phi = x_{10} \quad & \wedge \quad (x_4 \leftrightarrow \neg x_3) \\ & \wedge \quad (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge \quad (x_6 \leftrightarrow \neg x_4) \\ & \wedge \quad (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge \quad (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge \quad (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge \quad (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).\end{aligned}$$

Egy adott C hálózatra a megfelelő formula nyilván előállítható polinomiális időben.

Miért lesz a C hálózat, és a neki megfelelő ϕ formula ugyanakkor kielégíthető? Tekintsünk egy C-t kielégítő behelyettesítést. Ekkor a huzalokhoz tartozó változók jól definiált értékeket kapnak, melyek – a konstrukció értelmében – kielégítik a kapuknak megfelelő formulákat. Mivel a kimeneti kapuhoz tartozó változó értéke is 1 (hiszen a behelyettesítés kielégíti C-t), a ϕ változóira kapott értékek kielégítik ϕ-t. Megfordítva, ha ϕ-t kielégíti egy adott behelyettesítés, akkor a kimeneti változó 1, és a kapuknak megfelelő formulák értékének is egynek kell lennie, így ϕ azon változói, melyek C bemeneteihez tartoznak, kielégítik C-t. Beláttuk tehát, hogy C-SAT $\leq_P$ SAT, ezzel a bizonyítást befejeztük. ■

A 3-SAT probléma

Igen sok probléma NP-teljességét bizonyíthatjuk azzal, hogy visszavezetjük rá SAT-ot. A visszavezető algoritmusnak ilyenkor persze minden formulát tudnia kell bemenetként kezelni, ami nagyon sok eset áttekintését teszi szükségessé. Az esetek számának csökkentése érdekében sokszor kívánatos lenne a Boole-formulák egy szűkebb nyelvére szorítkozni. Annyira természetesen nem szabad leszűkíteni SAT-ot, hogy a kapott nyelv polinomiálisan eldönthető legyen (sőt, persze azt szeretnénk, ha még a szűkebb nyelv is NP-teljes lenne).



34.11. ábra. A $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$ formulához tartozó fa.

Céljainknak a 3-SAT elnevezésű nyelv tökéletesen meg fog felelni; leírásához az alábbi definíciók szükségesek. ***Literálnak*** nevezzük egy változónak vagy negáltjának megjelenését egy Boole-formulában. Egy Boole-formula ***konjunktív normálformájú,*** ha olyan, ÉS-ekkel összekapcsolt klózokból áll, melyeket egy vagy több, VAGY-okkal összekapcsolt literál alkot. Ha mindegyik klózban pontosan 3 különböző literál van, akkor ***3-konjunktív normálformáról*** beszélünk. A 3-konjunktív normálformák halmazát 3-CNF (3-conjunctive normal form) jelöli.

Például

$$(x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

egy 3-konjunktív normálforma, amely három klózból áll, ezek közül az első az $x_1,\ \neg x_1,\ x_2$ literálokat tartalmazza.

A 3-SAT probléma az, hogy adott, 3-CNF-beli ϕ Boole-formula kielégíthető-e. A következő tétel azt mutatja, hogy Boole-formulák kielégíthetőségének eldöntésére valószínűleg még akkor sem létezik polinomiális algoritmus, ha ebben az egyszerű normálformában vannak felírva.

34.10. tétel. 3-SAT NP-*teljes.*

Bizonyítás. A 34.9. tételben SAT $\in$ NP-re adott bizonyítás egy az egyben alkalmazható 3-SAT esetében is, így 3-SAT $\in$ NP. Meg kell még mutatnunk, hogy 3-SAT NP-nehéz. Ehhez belátjuk, hogy SAT $\leq_{\mathrm{P}}$ 3-SAT, amiből a 34.8. lemma szerint az állítás következik.

A visszavezető algoritmus három fő részből áll, mindegyik lépés közelebb viszi a bemeneti ϕ formulát a kívánt 3-konjunktív normálformájú alakhoz.

Az első lépés hasonló ahhoz, amit C-SAT $\leq_{\mathrm{P}}$ SAT bizonyításánál használtunk. Először felépítünk egy bináris „elemző” fát a ϕ formulához, melynek levelei a literálok, belső csúcsai pedig a műveletek. A 34.11. ábrán a

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \tag{34.3}$$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

34.12. ábra. Az $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ formula igazságtáblázata.

formula elemző fája látható. Az elemző fák valóban binárisak, hiszen a műveletek egy- vagy kétváltozósak (az ÉS és VAGY műveletek többváltozósak is lehetnek, ezeket az asszociativitás felhasználásával több kétváltozós műveletté írhatjuk át), így minden csúcsnak legfeljebb két gyereke lehet. A bináris elemző fát szemügyre véve láthatjuk, hogy nem más, mint egy hálózat, ami a ϕ formulát számítja ki.

Hasonlóan a 34.9. tétel bizonyításához, a belső csúcsok (azaz a műveletek) kimeneteihez (az ősök felé eső élekhez) rendeljük az y_i változókat. Ezt követően a ϕ formulát átírjuk, ismét csak úgy, mint 34.9. bizonyításában, tehát tekintjük a gyökérhez tartozó kimeneti változó, és a belső csúcsok működését leíró formulák konjunkcióját. A (34.3) formula esetén a kapott kifejezés a

$$\begin{aligned} \phi' = y_1 \quad & \wedge \quad (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge \quad (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge \quad (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge \quad (y_4 \leftrightarrow \neg y_5) \\ & \wedge \quad (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge \quad (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \end{aligned}$$

alakot ölti. Vegyük észre, hogy a kapott ϕ' formula (nem csak most, hanem mindig) olyan ϕ_i' részformulák konjunkciója, melyek legfeljebb három literált tartalmaznak.

A visszavezetés második lépése a ϕ_i' klózok konjunktív normálformába írása. Ehhez elkészítjük minden ϕ_i' igazságtáblázatát. Az igazságtáblázat soraiban a klózban lévő változók lehetséges értékei, és a klóz ezen behelyettesítés mellett felvett értéke szerepelnek. ϕ_i' igazságtáblázatának 0-t adó sorait felhasználva megadunk egy ***diszjunktív normálformát (DNF)*** (ez a konjunktív normálforma „fordítottja”: kívül vannak ÉS-ek, belül pedig VAGY-ok), mely ekvivalens $\neg\phi_i'$-vel. Ezt azután a DeMorgan-azonosságok ((B.2.) azonosságok) segítségével konjunktív normálformává írjuk át, mely ekvivalens ϕ_i'-vel.

Lássuk, hogy is megy ez, például a

$$\phi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$$

formulára. Vegyük szemügyre igazságtáblázatát (34.12. ábra); eszerint ϕ_1' a 0 értéket veszi föl, ha $(y_1 = 1, y_2 = 1, x_2 = 1)$, vagy $(y_1 = 1, y_2 = 0, x_2 = 1)$, vagy $(y_1 = 1, y_2 = 0, x_2 = 0)$,

vagy ha $(y_1 = 0, y_2 = 1, x_2 = 0)$. A $\neg\phi_1'$-vel ekvivalens diszjunktív normálforma tehát:

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2).$$

E formula tagadása ekvivalens ϕ_1'-vel, és a DeMorgan-azonosságok alkalmazásával a következő alakba írható:

$$\phi_1'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2).$$

konjunktív normálformulát kapjuk, amely ekvivalens ϕ_1'-vel.

Ily módon a ϕ' formula mindegyik ϕ_i' részformuláját ϕ_i'' konjunktív normálformává tudjuk alakítani, és így ϕ' ekvivalens a ϕ_i'' klózok konjunkciójából álló ϕ' konjunktív normál formával. Továbbá a kapott ϕ'' formula minden klózát legfeljebb három literál alkotja.

A bizonyítás harmadik és egyben utolsó lépésében tovább alakítjuk a formulát úgy, hogy minden klózban *pontosan* 3 literál legyen. Két segédváltozót fogunk használni, legyenek ezek p és q. ϕ'' minden C_i klózához a következőképp veszünk be klózokat ϕ'''-be:

- Ha C_i-ben 3 különböző literál szerepel, akkor egyszerűen magát C_i-t vesszük be.
- Ha C_i-ben 2 különböző literál szerepel, azaz $C_i = (l_1 \vee l_2)$, akkor vegyük be ϕ'''-be az $(l_1 \vee l_2 \vee p)$ és az $(l_1 \vee l_2 \vee \neg p)$ klózokat. A p és $\neg p$ literálok csak azt a formai követelményt hivatottak garantálni, hogy klózonként pontosan 3 literál szerepeljen: $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ nyilván ekvivalens $(l_1 \vee l_2)$-vel.
- Ha C_i-ben egyedül az l literál szerepel, akkor vegyük be ϕ'''-be az $(l \vee p \vee q)$, az $(l \vee p \vee \neg q)$, az $(l \vee \neg p \vee q)$ és az $(l \vee \neg p \vee \neg q)$ klózokat. Nyilvánvaló, hogy e négy klóz konjunkciója ekvivalens l-lel.

A fenti három lépés végigkövetéséből kiderült, hogy a ϕ''' 3-CNF-beli formula pontosan akkor elégíthető ki, ha ϕ kielégíthető. Az, hogy a visszavezetés polinom időben végrehajtható, mindegyik lépés esetében magától értetődő. ■

Gyakorlatok

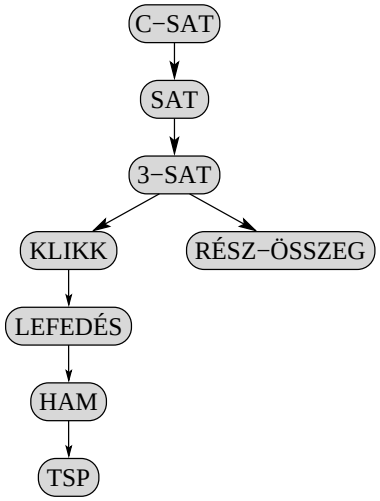
34.4-1. Adjunk meg olyan n méretű hálózatot, melynek a 34.9. tétel bizonyításában leírt kézenfekvő módszerrel való formulává alakítása n-ben exponenciális méretű formulához vezet.

34.4-2. Alakítsuk a (34.3) formulát a 34.10. tétel bizonyításában szereplő módszerrel 3-konjunktív normálformává.

34.4-3. Jagger professzor kizárólag az igazságtáblázat-technika felhasználásával szeretné megmutatni, hogy SAT $\leq_P$ 3-SAT, azaz a ϕ Boole-formula igazságtáblázata alapján szándékozik egy olyan 3-diszjunktív normálformát megadni, amely ekvivalens $\neg\phi$-vel, majd ezt negálva – a DeMorgan-szabályok alkalmazása után – megkapni a megfelelő, ϕ-vel ekvivalens 3-konjunktív normálformát. Mutassuk meg, hogy ez a módszer nem szolgáltat polinomiális visszavezetést.

34.4-4. Bizonyítsuk be, hogy annak eldöntése, hogy egy Boole-formula tautológia-e, teljes co-NP-re nézve. (*Útmutatás.* Lásd a 34.3-7. feladatot.)

34.4-5. Igazoljuk, hogy a diszjunktív normálformák kielégíthetőségének problémája polinom időben megoldható.



34.13. ábra. A 34.4. és 34.5. alfejezetek NP-teljességi bizonyításainak szerkezete. Valamennyi nyelv NP-teljességének bizonyítása C-SAT NP-teljességén alapul.

34.4-6. Tegyük fel, hogy rendelkezésünkre áll egy SAT-ot polinom id őben eldöntő A algoritmus. Hogyan tudnánk ekkor kielégítő behelyettesítéseket találni polinom időben tetszőleges ϕ formulára?

34.4-7. Legyen 2-CNF-SAT azoknak a kielégíthető konjunktív normál formuláknak a halmaza, amelyekben minden klóz pontosan két literált tartalmaz. Mutassuk meg, hogy 2-SAT $\in$ P. Adjunk minél gyorsabb algoritmust! (*Útmutatás.* Vegyük észre, hogy $(x \vee y)$ ekvivalens $(\neg x \rightarrow y)$-nal. Vezessük vissza 2-SAT-ot egy olyan, irányított gráfokra vonatkozó problémára, mely gyorsan megoldható.)

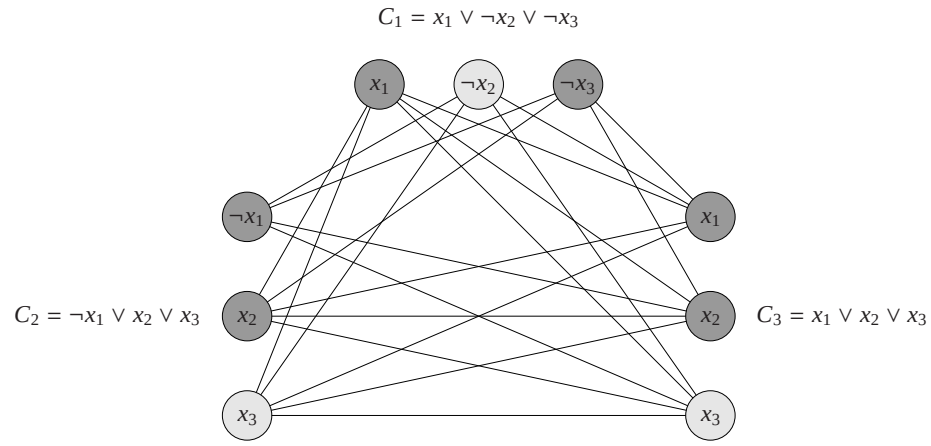
34.5. NP-teljes problémák

NP-teljes problémák a legváltozatosabb helyeken bukkannak fel: a logikában, a számtanban, a gráfelméletben, a számelméletben, az algebrában, a nyelv- és automataelméletben, az optimalizálásban, hálózatok tervezésénél, halmazok és partíciók kapcsán, tervezési és raktározási feladatoknál, játékok vizsgálatakor és még sorolhatnánk. Ebben az alfejezetben néhány gráfelméleti és halmazparticionálási feladat NP-teljességét bizonyítjuk, a visszavezetési technika segítségével.

A 34.13. ábra mutatja a visszavezetések szerkezetét; mindegyik nyelvet arra vezetjük vissza, amelyikből a nyíl rámutat.

34.5.1. A klikk probléma

Csúcsok egy V' halmaza a $G = (V, E)$ irányítatlan gráfban ***klikket*** alkot, ha V' bármely két csúcsa szomszédos G-ben. Egy klikk tehát nem más, mint G egy teljes részgráfja. A klikk ***mérete*** a benne lévő csúcsok száma. A ***klikk probléma*** „Mekkora a legnagyobb klikk egy G gráfban?”. Ez egy optimalizálási probléma, alakítsuk át döntésivé: „Létezik-e



34.14. ábra. A $\phi = C_1 \wedge C_2 \wedge C_3$ 3-CNF formulából kapott gráf, ahol $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, $C_3 = (x_1 \vee x_2 \vee x_3)$. A formula egy kielégítő behelyettesítése $(x_1 = 0, x_2 = 0, x_3 = 1)$. Ekkor C_1-ben $\neg x_2$, C_2-ben és C_3-ban x_3 az a változó, amely miatt a részformulák értéke 1; a nekik megfelelő klikk csúcsai világosabb színűek.

a G gráfban k méretű klikk?" (k-as klikk ugyanis pontosan akkor létezik, amikor legalább k-as). A formális definíció:

$$\text{KLIKK} = \{\langle G, k\rangle : G \text{ olyan gráf, melyben van } k \text{ méretű klikk}\}.$$

Azt, hogy egy gráfban van-e k-as klikk, eldönthetjük egyszerűen úgy, hogy a csúcsok minden k elemű részhalmazáról megvizsgáljuk, hogy klikk-e vagy sem. Ezen egyszerű algoritmus futásideje $\Omega(k^2\binom{|V|}{k})$ (V a csúcsok halmaza), ami polinomiális, ha k konstans. k azonban lehet $|V|/2$-höz közeli érték, mely esetben a lépésszám már szuperpolinomiális. Valószínű, hogy a klikk probléma megoldására nem létezik hatékony algoritmus.

34.11. tétel. *A klikk probléma* NP*-teljes.*

Bizonyítás. A probléma NP-beli, hiszen a klikket alkotó csúcsok tanúként való ellenőrzése nyilván megoldható polinomiális időben: csak azt kell megnézni, hogy bármely kettő szomszédos-e.

Azt, hogy a klikk probléma NP-nehéz, 3-SAT-ra való visszavezetésével bizonyítjuk. Ez kicsit meglepően hangzik, hiszen első ránézésre a logikai formuláknak kevés közük van a gráfokhoz.

A visszavezető algoritmus 3-SAT egy esetéből indul ki. Legyen $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ egy k klózból álló 3-CNF-beli formula. A C_r klóz $r = 1, 2, \ldots, k$ esetén pontosan 3 literálból áll, legyenek ezek l_1^r, l_2^r és l_3^r.

Megadunk egy G gráfot, melyben pontosan akkor lesz k méretű klikk, ha ϕ kielégíthető. $G = (V, E)$ konstrukciója a következő. Minden $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ klóz esetén bevesszük V-be a v_1^r, v_2^r, v_3^r csúcsokat. A v_i^r és v_j^s csúcsok közé pontosan akkor húzunk élt, ha az alábbi két feltétel teljesül:

- $r \neq s$
- az l_i^r és l_j^s literálok nem egymás negáltjai.

ϕ ismeretében G nyilván megadható polinomiális időben. Nézzük meg a gráf konstrukcióját egy konkrét példán. Legyen

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3).$$

A kapott G gráf a 34.14. ábrán látható.

Meg kell mutatnunk, hogy ez a $\phi \mapsto G$ transzformáció visszavezetés. Tegyük fel először, hogy ϕ kielégíthető. Ekkor minden C_r-ben van legalább egy l_i^r literál, ami a kielégítő behelyettesítésnél 1-et vesz fel. Minden klózból egy ilyen „igaz” literált véve egy k elemű csúcshalmazt kapunk, amely nyilván klikket alkot G-ben.

Tegyük fel most, hogy G-ben van egy V' k-as klikk. Azonos hármasban lévő csúcsok nincsenek összekötve, így V' pontosan egy elemet tartalmaz klózonként. A $v_i^r \in V'$ csúcsokhoz tartozó l_i^r literálok értékét 1-nek adhatjuk meg, hiszen nem szerepel köztük változó a negáltjával együtt. Ily módon minden klóz értéke 1, így ϕ értéke is 1, tehát ϕ kielégíthető. (A klikkben nem szereplő csúcsokhoz tartozó változók értéke tetszőleges lehet.) ■

A 34.14. ábra példájában ϕ egy kielégítő behelyettesítésében $x_2 = 0$ és $x_3 = 1$. A megfelelő 3 elemű klikk az első klóz $\neg x_2$, a második klóz x_3 és a harmadik klóz x_3 literáljaihoz tartozó csúcsokból áll. Minthogy a klikk nem tartalmaz sem x_1-hez, sem $\neg x_1$-hez tartozó csúcsot, x_1 értéke lehet 0 és 1 is bármely kielégítő behelyettesítésben.

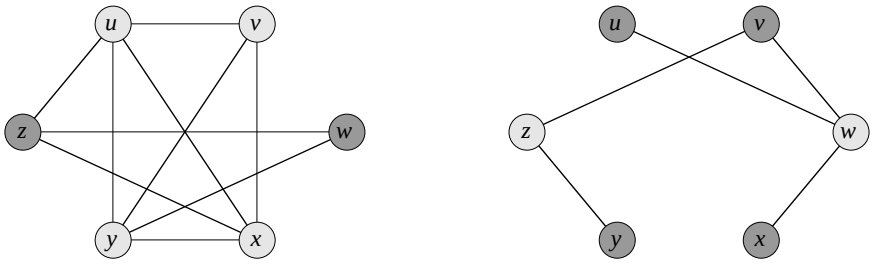
Vegyük észre, hogy a 34.11. tétel bizonyításakor 3-CNF egy tetszőleges esetét vezettük vissza KLIKK egy speciális szerkezettel bíró esetére. Ez alapján úgy tűnhet, hogy KLIKK NP-teljességét csak olyan gráfokra bizonyítottuk, melyekben a csúcsok olyan hármasokra oszthatók, melyeken belül nem mennek élek. Valóban igaz, hogy KLIKK NP-teljességét ilyen gráfokra bizonyítottuk, ebből azonban már következik, hogy a KLIKK probléma maga NP-teljes. Miért? Ha KLIKK-et meg tudjuk oldani általában, akkor nyilván meg tudjuk oldani minden speciális esetben is.

Nem volna ugyanakkor elegendő, ha 3-SAT-nak csak speciális eseteit vezetnénk vissza KLIKK eseteire. Miért? Előfordulhatna ugyanis az, hogy 3-SAT-nak csak „könnyű” eseteit vezetnénk vissza KLIKK eseteire, mely esetben a probléma, amelyet végeredményben visszavezetnénk KLIKK-re, nem lenne NP-teljes.

Érdemes azt is megfigyelni, hogy a visszavezetés csak 3-SAT eseteit használja, a megoldást magát nem. Polinomiális idejű visszavezetést ugyanis öreg hiba lenne arra a tudásra alapozni, hogy meg tudjuk mondani egy Boole-formuláról, hogy kielégíthető-e. Nem tudjuk ugyanis, hogy ezt a tudást hogyan szerezhetnénk meg polinom időben, sőt azt sem, hogy ez egyáltalán lehetséges-e.

34.5.2. A minimális lefedő csúcshalmaz probléma

Egy $G = (V, E)$ irányítatlan gráf ***lefedő csúcshalmazának*** nevezzük a $V' \subseteq V$ csúcshalmazt, ha minden $(u, v) \in E$ esetén $u \in V'$ vagy $v \in V'$ (esetleg mindkettő). Más szóval minden csúcs lefedi a hozzá tartozó éleket, G egy csúcshalmaza pedig akkor lefedő, ha minden élt legalább egy eleme lefed. Egy lefedő csúcshalmaz ***mérete*** a benne lévő csúcsok száma. Például a 34.15(b) ábra gráfjában $\{w, z\}$ egy kételemű lefedő csúcshalmaz.



(a) (b)

34.15. ábra. KLIKK visszavezetése LEFEDÉS-re. **(a)** Egy hat csúcsú $G = (V, E)$ ($V = \{u, v, w, x, y, z\}$) irányítatlan gráf a $V' = \{u, v, x, y\}$ klikkel. **(b)** A $\overline{G}$ gráfban a $V - V' = \{w, z\}$ csúcshalmaz lefedi az éleket.

A ***minimális lefedő csúcshalmaz probléma:*** keressük meg egy adott gráf minimális lefedő csúcshalmazát. Döntési problémaként megfogalmazva: van-e egy adott gráfnak k elemű lefedő csúcshalmaza. A megfelelő nyelv:

$$\text{LEFEDÉS} = \{\langle G, k\rangle \ : \text{a } G \text{ gráfnak van } k \text{ méretű lefedő csúcshalmaza.}\}$$

34.12. tétel. LEFEDÉS NP-*teljes.*

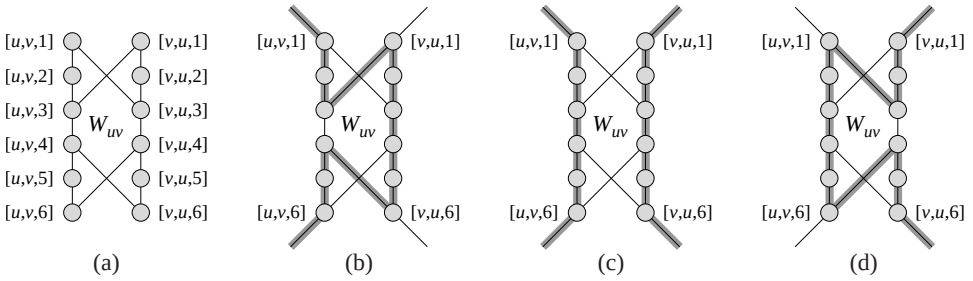
Bizonyítás. Először megmutatjuk, hogy LEFEDÉS $\in$ NP. Legyen adott a G gráf és a k egész szám. Tanúnak magát a $V' \subseteq V$ csúcshalmazt választjuk. Az ellenőrző algoritmus megnézi, hogy $|V'| = k$ teljesül-e, majd minden $(u, v) \in E$ élre megvizsgálja, hogy $(u \in V') \vee (u \in V')$ igaz-e. Mindez nyilván elvégezhető polinom időben.

A probléma NP-nehézségét a KLIKK $\leq_{\text{P}}$ LEFEDÉS visszavezetés segítségével mutatjuk meg, ehhez fel fogjuk használni a komplementer gráf fogalmát. A $G = (V, E)$ egyszerű gráf ***komplementere*** a $\overline{G} = (V, \overline{E})$ gráf, ahol $\overline{E} = \{(u, v) : (u, v) \notin E\}$. Más szóval $\overline{G}$ pontosan azokat az éleket tartalmazza, amik G-ből hiányoznak. A 34.15. ábra egy gráfot és a komplementerét mutatja, egyben illusztrálja a KLIKK-ről LEFEDÉS-re való visszavezetést.

A visszavezető algoritmus a KLIKK probléma egy $\langle G, k\rangle$ esetéből indul ki és meghatározza a $\overline{G}$ komplementert, ez könnyen végrehajtható polinom időben. Az algoritmus kimenete LEFEDÉS egy $\langle \overline{G}, |V| - k\rangle$ esete. A bizonyítást annak megmutatásával fejezzük be, hogy algoritmusunk valóban visszavezető függvényt számít ki, azaz a G gráfban akkor és csak akkor van k méretű klikk, ha a $\overline{G}$ gráfnak van $|V| - k$ méretű lefedő csúcshalmaza.

Tegyük fel, hogy G-ben $V' \subseteq V$ egy k méretű klikk. Azt állítjuk, hogy $V - V'$ lefedő csúcshalmaz $\overline{G}$-ban. Legyen (u, v) a $\overline{G}$ gráf tetszőleges éle. Ekkor $(u, v) \notin E$, ezért u és v közül legfeljebb az egyik tartozik V'-höz, hiszen bármely két V'-beli csúcs össze van kötve G-ben. Más szóval u és v közül legalább az egyik $(V - V')$-ben van, azaz $V - V'$ lefedi az (u, v) élt. Mivel (u, v) az $\overline{E}$ élhalmaz tetszőleges éle volt, $\overline{E}$ minden élét lefedi legalább egy $(V - V')$-beli csúcs, következésképp a $|V| - k$ méretű $V - V'$ halmaz lefedő $\overline{G}$-ban.

Megfordítva, tegyük fel, hogy $V - V'$ lefedő csúcshalmaz a $\overline{G}$ gráfban, ahol $|V'| = |V| - k$. Ekkor minden $u, v \in V$ esetén, ha $(u, v) \in \overline{E}$, akkor $u \in V'$ és $v \in V'$ közül legalább az egyik teljesül. Eszerint minden $(u, v) \in E$-re, ha $u, v \notin V'$, akkor $(u, v) \in E$. Azaz $V - V'$ klikk, és mérete $|V| - |V'| = k$. ■



34.16. ábra. A LEFEDÉS HAM-ra történő visszavezetése során használt segédgráf. A G gráf egy (u, v) éléhez tartozik a W_{uv} segédgráf a visszavezetés során előállított G' gráfban. **(a)** A segédgráf. **(b)–(d)** A vastagított utak az egyedüliek, melyek áthaladnak a segédgráf minden csúcsán, feltéve, hogy a segédgráfot G többi részével csak az $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ és $[v, u, 6]$ csúcsokon keresztül kötjük össze.

Mivel LEFEDÉS NP-teljes, természetesen nem várható, hogy polinomiális algoritmust találunk rá. A 35.1. alfejezetben azonban bemutatunk egy polinomiális közelítő algoritmust, amely a minimálishoz képest legfeljebb kétszeres méretű lefedő csúcshalmazt ad meg.

Nem kell tehát azonnal feladni a reményt, ha egy probléma NP-teljesnek bizonyul; létezhet olyan polinomiális algoritmus, amely közel optimális megoldást talál. A 35. fejezet számos közelítő algoritmust mutat NP-teljes problémákra.

34.5.3. A Hamilton-kör probléma

Visszatérünk a 34.2. alfejezetben definiált Hamilton-kör problémához.

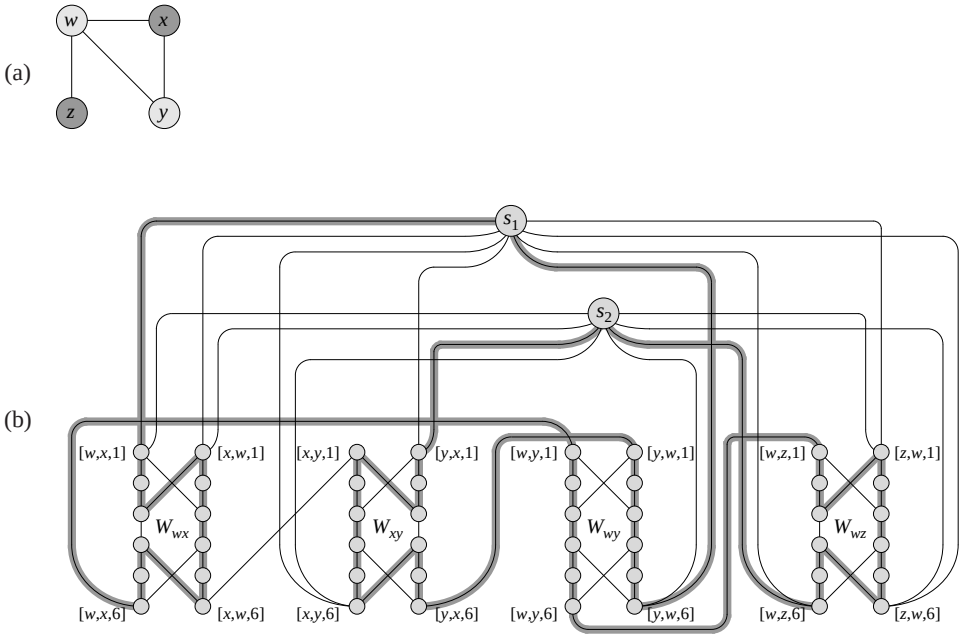
34.13. tétel. HAM NP-*teljes.*

Bizonyítás. Először megmutatjuk, hogy HAM $\in$ NP. A $G = (V, E)$ gráf esetén a tanú egy $|V|$ csúcsból álló sorozat. Az ellenőrző algoritmus megvizsgálja, hogy a sorozat V permutációja-e, és hogy a szomszédos csúcsok (az első és az utolsó is szomszédosnak számít) össze vannak-e kötve. Ez az ellenőrzés nyilván végrehajtható polinom időben.

Most bebizonyítjuk, hogy a Hamilton-kör probléma NP-nehéz, visszavezetjük a LEFEDÉS problémát HAM-ra. Egy adott $G = (V, E)$ irányítatlan gráfhoz és adott k természetes számhoz konstruálunk egy olyan $G' = (V', E')$ irányítatlan gráfot, melynek pontosan akkor van Hamilton-köre, ha G-nek létezik k elemű lefedő csúcshalmaza.

A konstrukció egy speciális szerkezetű ***segédgráfon*** alapul, amelyet a 34.16(a) ábrán láthatunk. A G gráf minden (u, v) éléhez G' tartalmazni fogja a segédgráf egy példányát, melyet W_{uv}-vel fogunk jelölni. A W_{uv}-ben szereplő 12 csúcsot $[u, v, i]$-vel, illetve $[v, u, i]$-vel fogjuk jelölni, ahol $i = 1, 2, \ldots, 6$. A W_{uv}-ben szereplő 14 élt a 34.16(a) ábrán látható módon húzzuk be.

Az egyes segédgráfok csúcsai közül csak az $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ és $[v, u, 6]$ csúcsok lehetnek összekötve a G' gráf többi csúcsával, a többi 8 csúcsnak csak a segédgráfon belül vannak szomszédai. Ennek következtében G' bármely Hamilton-köre csak a 34.16(b)–(d) ábrákon látható három út valamelyikén haladhat keresztül W_{uv} csúcsain. Ha a Hamilton-kör az $[u, v, 1]$ csúcsban lép be a segédgráfba, akkor az $[u, v, 6]$ csúcsban kell kilépnie, és keresztülmegy vagy a segédgráf mind a 12 csúcsán (34.16(b) ábra), vagy az $[u, v, 1]$, $[u, v, 2]$, …, $[u, v, 6]$ csúcsokon (34.16(c) ábra). Az utóbbi esetben a körnek természetesen vissza kell



34.17. ábra. LEFEDÉS egy esetének visszavezetése HAM egy esetére. **(a)** A G irányítatlan gráf éleit a w és y csúcsok lefedik. **(b)** A visszavezetés során kapott G' irányítatlan gráf. A vastagított élek alkotják a w és y csúcsokkal történő fedéshez tartozó Hamilton-kört.

térnie valamikor a segédgráfba, hogy átmehessen a $[v, u, 1]$, $[v, u, 2]$, ..., $[v, u, 6]$ csúcsokon. Hasonlóképpen, ha a Hamilton-kör a $[v, u, 1]$ csúcsban lép be a segédgráfba, akkor a $[v, u, 6]$ csúcsban kell kilépnie, és keresztül kell mennie vagy a segédgráf mind a 12 csúcsán (34.16(d) ábra), vagy a $[v, u, 1]$, $[v, u, 2]$, ..., $[v, u, 6]$ csúcsokon (34.16(c) ábra). A felsoroltakon kívül nem létezik más út, amely átmenne a segédgráf mind a 12 csúcsán.

A G' gráf többi (segédgráfban nem szereplő) csúcsai az úgynevezett ***választó*** csúcsok: $s_1, s_2, \ldots, s_k$. A választó csúcsokhoz csatlakozó éleket használjuk arra, hogy kiválasszuk a G gráf egy k elemű lefedő csúcshalmazát.

A segédgráfok élein kívül kétféle típusú él lehet G'-ben, melyek a 34.17. ábrán láthatók. Az első típusú élek arra szolgálnak, hogy a G gráf minden u csúcsához az u-hoz csatlakozó éleknek megfelelő segédgráfokat egy útra tudjuk felfűzni. Ezt a következőképpen valósítjuk meg. Minden $u \in V$-re tekintsük az u-val szomszédos éleket G-ben és rendezzük sorba őket tetszőlegesen. Legyenek az így kapott csúcsok $u^{(1)}, u^{(2)}, \ldots, u^{(d(u))}$, ahol $d(u)$ az u csúccsal szomszédos csúcsok száma. Létrehozunk G'-ben egy utat az u-val szomszédos éleknek megfelelő segédgráfokon keresztül oly módon, hogy hozzávesszük az E' élhalmazhoz az $([u, u^{(i)}, 6], [u, u^{(i+1)}, 1])$ éleket $i = 1, 2, \ldots, (k-1)$-re. A 34.17. ábrán például a w-vel szomszédos csúcsokat az x, y, z sorrendben nézzük, így a G' gráfba (amit a (b) ábrán láthatunk) a $([w, x, 6], [w, y, 1])$ és a $([w, y, 6], [w, z, 1])$ éleket vesszük be.

Az ily módon bevett élek arra a célra szolgálnak, hogy ha az u csúcs szerepel G egy lefedő csúcshalmazában, akkor meg tudjunk adni egy utat a G' gráf $[u, u^{(1)}, 1]$ csúcsából az $[u, u^{(d(u))}, 6]$ csúcsába, amely „lefedi” az u-hoz csatlakozó élekhez tartozó segédgráfokat. Ez azt jelenti, hogy bármely ilyen $W_{uu^{(i)}}$ segédgráfra az út tartalmazza vagy mind a 12 csúcsot

(ha $u^{(i)}$ nincs benne a lefedő csúcshalmazban), vagy pontosan az $[u, u^{(i)}, 1]$, $[u, u^{(i)}, 2]$, …, $[u, u^{(i)}, 6]$ csúcsokat (ha $u^{(i)}$ is benne van a lefedő csúcshalmazban).

A másik típusú élek, amiket hozzáveszünk E'-höz az $[u, u^{(1)}, 1]$ és az $[u, u^{(d(u))}, 6]$ csúcsokat kötik össze az összes választó csúccsal, minden $u \in V$-re. Az E' élhalmazt tehát bővítjük az

$$(s_j, [u, u^{(1)}, 1]) \text{ és } (s_j, [u, u^{(d(u))}, 6])$$

élekkel minden $u \in V$-re és $j = 1, 2, \ldots, k$-ra.

Először is megmutatjuk, hogy G' mérete polinomiális G méretében, amiből már következik, hogy G'-t polinom időben meg tudjuk konstruálni G-ből, hiszen az egyes elemek (csúcsok, illetve élek) konstrukciója nyilván végrehajtható polinom időben. G minden éléhez létrehoztunk egy 12 csúcsú segédgráfot, G'-ben ezeken kívül szerepelnek még a választó csúcsok, így G' csúcsainak száma pontosan $12|E|+k \leq 12|E|+|V|$, ami polinomiális G méretében. A G'-beli élek háromfélék lehetnek: a segédgráfokon belüliek, ezeknek a száma $14|E|$, segédgráfokat összekötők, ezekből minden $u \in V$ csúcsra éppen $d(u) - 1$ van, végül olyanok, melyek a segédgráfokat a választó csúcsokkal kötik össze, ilyen élből minden csúcsra $2k$ darab van. Így

$$\begin{aligned} |E'| &= 14|E| + \left(\sum_{u \in V} d(u) - 1\right) + 2k|V| \\ &= 14|E| + (2|E| - |V|) + 2k|V| \\ &= 16|E| + (2k-1)|V| \leq 16|E| + (2|V| - 1)|V|, \end{aligned}$$

ami szintén polinomiális G méretében.

Most azt mutatjuk meg, hogy az átalakítás, aminek segítségével G'-t kaptuk G-ből, visszavezetés. Ehhez azt kell belátnunk, hogy G-ben pontosan akkor létezik k csúcsú lefedő halmaz, ha G'-ben létezik Hamilton-kör.

Tegyük fel, hogy a $G = (V, E)$ gráfnak létezik egy $V^* \subseteq V$ k méretű lefedő csúcshalmaza. Legyen $V^* = \{u_1, u_2, \ldots, u_k\}$. A 34.17. ábrán látható módon megadunk egy Hamilton-kört G'-ben, melyet a következő élek alkotnak.[8] Minden u_j-re az $\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \leq i \leq d(u_j)\}$ élhalmaz, vagyis az u_j-hez csatlakozó éleknek megfelelő segédgráfokat összekötő élek, emellett az ezen segédgráfokban szereplő élek a 34.16(b)–(d) ábrákon látható módok valamelyike szerint, attól függően, hogy az élnek egy vagy két végpontja van benne V^*-ban, végül az

$$\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \leq j \leq k\},$$

$$\{(s_{j+1}, [u_j, u_j^{(d(u_j))}, 6]) : 1 \leq j \leq k-1\} \text{ és}$$

$$\{(s_1, [u_k, u_k^{(d(u_k))}, 6])\}$$

élhalmazok, minden $u_j \in V^*$-ra.

Nem nehéz ellenőrizni (és a 34.17. ábrán ezt érdemes is megtenni), hogy ezek az élek csakugyan kört alkotnak. A kör (mondjuk) s_1-ben kezdődik, végigmegy az u_1-hez csatlakozó élekhez tartozó segédgráfokon, ezt követően elmegy s_2-be, végigmegy az u_2-höz csatlakozó élekhez tartozó segédgráfokon, és így tovább, egészen addig, míg vissza nem tér

[8] A köröket valójában csúcsok, és nem élek segítségével definiáljuk (lásd a B4. alfejezetet). Az egyszerűség kedvéért a Hamilton-kört most az éleivel adjuk meg.

s_1-be. A kör minden segédgráfot egyszer vagy kétszer látogat meg, aszerint, hogy a megfelelő élnek egy vagy két végpontja van benne V^*-ban. Mivel V^* lefedő csúcshalmaz G-ben, G minden éle csatlakozik V^* valamelyik csúcsához, vagyis a kör az összes segédgráf összes csúcsán átmegy. Minthogy körünk a választó csúcsok mindegyikén is átmegy, valóban Hamilton-köre G'-nek.

Tegyük fel most, hogy a $G' = (V', E')$ gráfban létezik egy $C \subseteq E'$ Hamilton-kör. Azt állítjuk, hogy a k elemű

$$V^* = \{u \in V \ : \ \text{létezik } j \leq k, \ \text{hogy } (s_j, [u, u^{(1)}, 1]) \in C\} \tag{34.4}$$

csúcshalmaz lefedi G éleit. Ennek bizonyításához osszuk fel C-t olyan maximális utakra, melyek valamely s_i választó csúcsban kezdődnek, átmennek az $(s_i, [u, u^{(1)}, 1])$ élen valamely $u \in V$-re, és egy s_j választó csúcsban érnek véget anélkül, hogy bármely más választó csúcson átmennének. Nevezzük az ilyen utakat „fedőutaknak”. A G' gráf konstrukciójából látható, hogy az s_i választó csúcsban kezdődő fedőútnak, amely az $(s_i, [u, u^{(1)}, 1])$ élen megy át, át kell mennie az u csúcshoz G-ben csatlakozó összes élnek megfelelő segédgráfokon. Jelöljük ezt a fedőutat p_u-val. A V^* halmaz definíciója alapján $p_u \in V^*$. Tekintsünk egy tetszőleges, p_u által meglátogatott segédgráfot. Ez vagy W_{uv} vagy W_{vu} lesz, valamely $v \in V$-re. Az ennek megfelelő E-beli (u, v) élt a V^*-beli u és v csúcs is lefedi. Mivel minden segédgráfot meglátogat egy (vagy két) fedőút, a V^* csúcshalmaz valóban lefedi G minden élét. ■

34.5.4. Az utazóügynök probléma

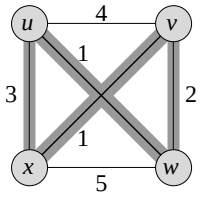
Az utazóügynök problémában (traveling salesman problem – TSP) – mely szoros kapcsolatban áll a Hamilton-kör problémával – szereplő ügynöknek n várost kell meglátogatnia tetszőleges sorrendben, de minél kisebb utazási költség mellett. (Ez negatív is lehet, ekkor az ügynöknek haszna van az útból, néha ez is előfordul.) Pontosabban: adott egy n csúcsú teljes gráf, melynek éleihez egész számokat rendelünk. (Ezek lesznek az utazási költségek az egyes városok között.) A feladat olyan Hamilton-kör megtalálása, melynek összköltsége (vagyis az érintett élekhez rendelt számok összege) minimális. A 34.18. ábrán látható esetben például (u, w, v, x, u) minimális (7) költségű Hamilton-kör. A megfelelő optimalizálási problémából nyert döntési problémához tartozó formális nyelv:

$$\begin{aligned} \text{TSP} = \{\langle G, c, k\rangle : \ & G = (V, E) \text{ teljes gráf,} \\ & c : V \times V \longrightarrow \mathbf{Z}, \\ & k \in \mathbf{Z} \\ & \text{és } G\text{-nek van legfeljebb } k \text{ költségű Hamilton-köre.}\} \end{aligned}$$

A következő tétel azt mutatja, hogy valószínűleg nincs gyors algoritmus az utazóügynök probléma megoldására.

34.14. tétel. TSP NP-*teljes.*

Bizonyítás. Először TSP $\in$ NP-t igazoljuk. Tanúnak egy megfelelő költségű Hamilton-kört használunk. Az ellenőrző algoritmus megnézi, hogy a megadott csúcssorozat valóban



34.18. ábra. Az utazóügynök probléma egy esete. A vastagított élek minimális költségű Hamilton-kört alkotnak.

Hamilton-kör-e (ilyen polinomiális algoritmust már láttunk), majd összeadja az élek költségeit és összehasonlítja k-val. Mindez nyilván végrehajtható polinom időben.

TSP NP-nehézségének bizonyításához megmutatjuk, hogy HAM $\leq_P$ TSP. Legyen $G = (V, E)$ HAM egy esete. Az ehhez tartozó TSP-beli eset konstrukciója a következő. Tekintsük a $G' = (V, E')$ teljes gráfot ($E' = \{(i, j) : i, j \in V \text{ és } i \neq j\}$). Legyenek az élek költségei

$$c(i, j) = \begin{cases} 0, & \text{ha } (i, j) \in E, \\ 1, & \text{ha } (i, j) \notin E. \end{cases}$$

(Mivel G irányítatlan, ezért nem tartalmaz hurkot, és így $c(v, v) = 1$ minden $v \in V$ csúcsra.) Ekkor TSP egy esete $\langle G', c, 0\rangle$, ami könnyen előállítható polinomiális idő alatt.

Legyen ezek után a G-hez rendelt TSP-eset $\langle G, c, 0\rangle$. Ennek kiszámítása nyilván polinomiális idejű. Könnyen látható, hogy G-ben akkor és csak akkor van Hamilton-kör, ha G'-nek létezik legfeljebb 0 költségű Hamilton-köre. ■

34.5.5. A részletösszeg probléma

Ez az NP-teljes probléma aritmetikai jellegű. A ***részletösszeg problémánál*** adott egy $S \subseteq \mathbf{N}$ véges halmaz és egy $t \in \mathbf{N}$ szám. A kérdés az, hogy létezik-e olyan $S' \subseteq S$ halmaz, melyben az elemek összege t. Például ha $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409\,2793, 16808, 17206, 117705, 117993\}$ és $t = 138457$, akkor az $S' = \{1, 2, 7, 98, 343, 686\,2409, 17206, 117705\}$ részhalmaz ilyen.

A problémát szokás szerint nyelv formájában definiáljuk.

$$\text{RÉSZ-ÖSSZEG} = \{\, \langle S, t\rangle \;:\; \text{létezik } S' \subseteq S : \sum_{s \in S'} s = t\}.$$

Mint minden aritmetikai problémánál, itt is fontos észben tartani, hogy a szabványos kódolás során az input egész számokat kettes számrendszerben adjuk meg. Ezt figyelembe véve meg tudjuk mutatni, hogy a részletösszeg problémára valószínűleg nincs gyors algoritmus.

34.15. tétel. RÉSZ-ÖSSZEG NP-*teljes.*

Bizonyítás. RÉSZ-ÖSSZEG $\in$ NP igazolásához egy $\langle S, t\rangle$ esethez az $S' \subseteq S$ részhalmazt választjuk tanúnak. $\sum_{s \in S'} s = t$ teljesülését egy alkalmas ellenőrző algoritmus nyilván meg tudja vizsgálni polinom időben.

Most megmutatjuk, hogy 3-SAT $\leq_P$ RÉSZ-ÖSSZEG. Legyen ϕ tetszőleges formula az $x_1, x_2, \ldots, x_n$ változókon, a $C_1, C_2, \ldots, C_k$ klózokkal, melyek mindannyian pontosan három literált tartalmaznak. A visszavezető algoritmus ϕ-hez megadja a RÉSZ-ÖSSZEG probléma egy $\langle S, t\rangle$ esetét úgy, hogy ϕ pontosan akkor lesz kielégíthető, ha létezik S-nek olyan részhalmaza, melyben az elemek összege k. Két feltételezéssel fogunk élni a ϕ formulát illetően,

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

34.19. ábra. 3-SAT visszavezetése RÉSZ-ÖSSZEG-re. A felhasznált 3-CNF-beli formula $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, ahol $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3), C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3), C_3 = (\neg x_1 \vee \neg x_2 \vee x_3), C_4 = (x_1 \vee x_2 \vee x_3)$. ϕ egy kielégítő behelyettesítése $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. A visszavezetés során kapott S halmaz a következő tízes számrendszerbeli számokból áll: 1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2. A t szám értéke 1114444. Az $S' \subseteq S$ halmaz elemei világosabb színnel vannak kiemelve.

anélkül, hogy ez az általánosság rovására menne. Először is feltesszük, hogy egyetlen klózban sem szerepel egy változó a negáltjával együtt, hiszen az ilyen klózok minden behelyettesítésre 1-et vesznek fel, így nem befolyásolják a formula kielégíthetőségét. Másodszor, feltesszük, hogy minden változó megjelenik legalább egy klózban, ellenkező esetben az értéke nem befolyásolja a formula kielégíthetőségét.

A visszavezető algoritmus két számot hoz létre minden változóhoz és minden klózhoz is, ezek együttesen alkotják majd az S halmazt. A számokat tízes számrendszerben fogjuk létrehozni úgy, hogy minden számnak $n + k$ jegye legyen és minden számjegy vagy egy változóhoz, vagy egy klózhoz tartozzon.

Az S halmaz és a t szám konstrukciója az alábbi módon történik (lásd 34.19. ábra). Az $n + k$ darab helyi értéket (vagyis a számjegyek pozícióit) megcímkézzük a klózokkal vagy a változókkal: az utolsó k helyi értéket (vagyis a k legkisebb helyi értéket) a klózokkal, az első n helyi értéket a változókkal címkézzük.

- A t számban a változókhoz tartozó helyi értékekre 1-est írunk, a klózokhoz tartozó helyi értékekre 4-est.
- Minden x_i változóhoz két számot, v_i-t és v_i'-t tesszük S-be. Mindkettőnél az 1-es számjegy áll az x_i címkéjű helyi értéken és 0 a többi változóval címkézett helyi értéken. Ha az x_i literál megjelenik a C_j klózban, akkor a v_i szám C_j címkéjű helyi értékére az 1-es számjegyet írjuk és 0-t írunk a v_i szám minden más, klózokkal címkézett helyi értékére. Ha a $\neg x_i$ literál megjelenik a C_j klózban, akkor a v_i' szám C_j címkéjű helyi értékére az 1-es számjegyet írjuk és 0-t írunk v_i' minden más, klózokkal címkézett helyi értékére.

Az ily módon kapott összes v_i és v_i' számok mind különbözők. Miért? Ha $l \neq i$, akkor a v_i és v_i' számok különböznek a v_l és v_l' számoktól, hiszen eltérés van köztük az első n számjegyben. Másrészt v_i sem lehet egyenlő v_i'-vel egyetlen i-re sem, hiszen ellenkező esetben az x_i és $\neg x_i$ literálok pontosan ugyanazokban a klózokban jelennének meg, holott feltettük, hogy egyetlen klóz sincs, amelyben mindketten megjelennének és legalább egyiküknek valamelyik klózban meg kell jelennie, mivel ezt is feltettük.

- Minden C_j klózhoz is két számot, s_j-t és s_j'-t tesszük S-be. A C_j-vel címkézett helyi érték kivételével minden számjegy 0 mindkettőjük esetében, míg az s_j szám C_j címkéjű helyi értékén 1-es, az s_j' szám C_j címkéjű helyi értékén 2-es áll. Ezek az egészek „felesleg" változók, amelyeket arra használunk, hogy a klózokkal címkézett pozíciókhoz hozzáadva megkapjuk a 4 célértéket.

  A 34.19. ábra átnézése mutatja, hogy az s_j és s_j' számok nemcsak egymástól különböznek, hanem az s_i, s_i' számoktól is, ha $i \neq j$.

Vegyük észre, hogy minden egyes helyi értékre az összes S-beli szám ilyen helyi értékű számjegyeinek összege legfeljebb 6 lehet (a klózokkal címkézett helyi értékek esetén három 1-es számjegy lesz a v_i és v_i' számok megfelelő helyi értékén és egy 1-es és egy 2-es az s_i és s_i' számok megfelelő helyi értékén). Következésképp akárhány S-beli számot adunk is össze, soha nem kell maradékot átvinni egyik helyi értékről a másikra az összeadás elvégzésekor.[9]

A visszavezetés nyilván végrehajtható polinomiális időben, hiszen az S halmaz $2n + 2k$ elemet tartalmaz, melyek mind $n + k$ jegyűek, s egy számjegy előállítása polinom időben végrehajtható, a t szám pedig $n + k$ konstans időben előállítható számjegyből áll.

Most megmutatjuk, hogy a 3-CNF-beli ϕ formula pontosan akkor kielégíthető, ha létezik olyan S' részhalmaza S-nek, melyben az elemek összege t. Tegyük fel először, hogy ϕ-nek létezik kielégítő behelyettesítése. Ha $x_i = 1$ ebben a behelyettesítésben, akkor vegyük be v_i-t az S' halmazba, ellenkező esetben vegyük be v_i'-t, $i = 1, 2, \ldots, n$-re. v_i és v_i' közül pontosan az egyiket vettük be tehát S'-be, így a változókkal címkézett helyi értékeken az S'-ben lévő elemek összege 1, ami azonos a t szám ilyen helyi értékein lévő számjegyekkel. Mivel a behelyettesítésben minden klóz értéke 1 kell legyen, minden klóz tartalmaz olyan literált, melynek az értéke 1. Így minden klózhoz egy, kettő vagy három olyan literál lesz, mely szerepel benne és az értéke 1. Mivel épp az ilyen literáloknak megfelelő v_i-k vagy v_i'-k kerülnek S'-be, a v_i és v_i' számok konstrukciója alapján minden klózzal címkézett helyi értéken 1,2 vagy 3 lesz az S'-ben lévő elemek összege. (A 34.19. ábrán például a $\neg x_1$, $\neg x_2$ és x_3 literálok értéke 1 a megadott kielégítő behelyettesítésben. A C_1 és C_4 klózok pontosan egyet tartalmaznak ezek közül, így v_1', v_2' és v_3 összege 1 a C_1 és C_4 helyi értékeken. A C_2 klóz pontosan kettőt tartalmaz az említett literálok közül, így a C_2 helyi értéken a v_1', v_2' és v_3 számok összege 2, míg a C_3 klóz a $\neg x_1$, $\neg x_2$ és x_3 literálok mindegyikét tartalmazza, tehát a C_3 helyi értéken a v_1', v_2' és v_3 számok összege 3.) Ahhoz, hogy a t számban szereplő 4-es számjegyet minden, klózokkal címkézett helyi értéken elérjük, már csak annyit kell tennünk, hogy minden klózzal címkézett helyi értékre a meglévő 1-es, 2-es vagy 3-as számjegyet az s_j és s_j' számok közül a megfelelő(k) bevételével 4-re egészítjük ki. Precízebben: ha a C_j klózban az 1-es értéket felvevő literálok száma egy, akkor bevesszük S'-be

[9] Természetesen tízes számrendszer helyett bármilyen, legalább hetes alapú számrendszer alkalmas lenne. A 34.19. ábrán látható számok hetes számrendszerben épp a részfejezet elején leírt S halmaz elemei és az ott megadott t szám lesznek.

az s_j és s_j' számokat, ha ez a szám kettő, akkor bevesszük S'-be s_j'-t, de s_j-t nem, ha pedig az 1-es értéket felvevő literálok száma három, akkor bevesszük S'-be s_j-t, de s_j'-t nem (a 34.19. ábrán S' a klózokhoz tartozó számok közül s_1-et, s_1'-t, s_2'-t, s_3-at, s_4-et és s_4'-t tartalmazza). Ily módon az S'-beli elemek összege a változókkal címkézett helyi értékeken 1 lesz, a klózokkal címkézett helyi értékeken pedig 4, azaz éppen a t számot kapjuk összegként.

Tegyük fel most, hogy létezik olyan $S' \subseteq S$ halmaz, melyben az elemek összege t. Az S' részhalmaz a v_i és v_i' számok közül pontosan egyet tartalmaz, minden $i = 1, 2, \ldots, n$ esetén, ellenkező esetben a változókkal címkézett helyi értékeken nem kaphatnánk 1-et összegként. Ha $v_i \in S'$, akkor legyen az x_i változó értéke 1, ha $v_i' \in S'$, akkor legyen x_i értéke 0. Azt állítjuk, hogy az így kapott behelyettesítés kielégíti ϕ-t. Ehhez be kell látnunk, hogy minden klóz értéke 1. Ennek bizonyításához vegyük észre, hogy a C_j-vel címkézett helyi értéken csak úgy tudjuk elérni a 4-es számjegyet, ha az S' halmaz tartalmaz olyan v_i vagy v_i' számot, amelynek C_j-vel címkézett helyi értékén 1-es áll, hiszen az s_j és s_j' számok együttes hozzájárulása ehhez a számjegyhez legfeljebb 3 lehet. Ha S' tartalmaz olyan v_i-t, melynek C_j címkéjű helyi értékén 1-es áll, akkor az x_i literál megjelenik a C_j klózban. Mivel $x_i = 1$ (hiszen $v_i \in S'$), a C_j klóz értéke 1. Hasonlóképp, ha S' tartalmaz olyan v_i'-t, melynek C_j címkéjű helyi értékén 1-es áll, akkor a $\neg x_i$ literál jelenik meg a C_j klózban. Mivel most $x_i = 0$ (hiszen $v_i' \in S'$), a C_j klóz értéke ismét 1. Beláttuk tehát, hogy minden klóz értéke 1 lesz, a bizonyítást ezzel befejeztük. ■

Gyakorlatok

34.5-1. Az ***izomorf részgráf probléma:*** adottak a G_1 és G_2 gráfok, kérdés, hogy G_1 izomorf-e G_2 egy részgráfjával. Mutassuk meg, hogy ez a probléma NP-teljes.

34.5-2. A ***0-1 egészértékű programozási probléma:*** adott az A $m \times n$-es egész mátrix és a b m-dimenziós egész vektor. Kérdés, hogy létezik-e $x \in \{0, 1\}^n$, melyre $Ax \leq b$. Bizonyítsuk be, hogy ez a probléma is NP-teljes. (*Útmutatás.* Vezessük vissza rá 3-SAT-ot.)

34.5-3. Az ***egészértékű lineáris programozási probléma*** az előző feladatban megadotthoz hasonló, a különbség annyi, hogy az x vektor koordinátái ezúttal nemcsak 0 és 1 lehetnek, hanem tetszőleges egész számok. Mutassuk meg, hogy ha a 0-1 egészértékű programozási probléma NP-nehéz, akkor az egészértékű lineáris programozási probléma is NP-nehéz.

34.5-4. Mutassuk meg, hogy a részletösszeg probléma unáris kódolás esetén polinom időben megoldható.

34.5-5. A ***halmaz-partíció probléma:*** adva van számok egy S halmaza, kérdés, hogy létezik-e $A \subseteq S$, melyre $\sum_{x \in A} x = \sum_{x \in S - A} x$. Mutassuk meg, hogy a halmaz-partíció probléma NP-teljes.

34.5-6. Mutassuk meg, hogy a Hamilton-út probléma NP-teljes.

34.5-7. A ***leghosszabb kör probléma:*** határozzuk meg egy gráf leghosszabb körének hosszát. Mutassuk meg, hogy ez a probléma NP-teljes.

34.5-8. A ***fél 3-SAT*** probléma az alábbi: adott egy ϕ 3-CNF-beli formula, melynek n változója és m klóza van, ahol m páros szám. Feladat annak eldöntése, hogy létezik-e olyan behelyettesítés, melyre ϕ klózainak pontosan a fele vesz fel 1-es értéket. Mutassuk meg, hogy ez a probléma NP-teljes.

Feladatok

34-1. Független halmaz

A $G = (V, E)$ gráfban egy $V' \subseteq V$ csúcshalmaz ***független,*** ha V' semelyik két csúcsa sincs összekötve. A ***független halmaz probléma:*** találjunk maximális méretű független csúcshalmazt egy adott G gráfban.

a. Fogalmazzuk meg a megfelelő optimalizálási és döntési problémát, és igazoljuk hogy az utóbbi NP-teljes. (*Útmutatás.* Vezessük vissza rá a KLIKK problémát.)

b. Tegyük fel, hogy rendelkezésünkre áll egy szubrutin, amely megoldja az a. pontban definiált döntési problémát. Adjunk algoritmust egy maximális méretű független ponthalmaz megtalálására. A futási idő legyen polinomiális $|V|$-ben és $|E|$-ben. (A szubrutin hívását egy lépésnek tekintjük.)

Noha a független halmaz döntési probléma NP-teljes, bizonyos speciális esetei polinom időben megoldhatók.

c. Adjunk minél jobb algoritmust a probléma megoldására olyan gráfok esetében, melyekben minden csúcs foka 2.

d. Adjunk minél jobb algoritmust a probléma megoldására páros gráfok esetében. (*Útmutatás.* Használjuk a 26.3. alfejezet eredményeit.)

34-2. Bonnie és Clyde

Bonnie és Clyde éppen most raboltak ki egy bankot. Zsákmányuk egy értékekkel teli táska, melynek tartalmát szeretnék elosztani. Az alábbi eshetőségek mindegyikére vagy adjunk egy polinomiális algoritmust, vagy bizonyítsuk be, hogy a probléma NP-teljes. A bemenet minden esetben a táskában lévő tárgyak n hosszúságú listája, az egyes tárgyak mellett fel van tüntetve azok értéke is.

a. A táska tartalma n pénzérme, kétféle típusból: egy részük x dolláros, a többi y dolláros. Az új tulajdonosok pontosan egyenlően szeretnének osztozni.

b. A táskában most is n érme van, ezek azonban tetszőleges kettő-hatvány dollárt érhetnek (1 dollár, 2 dollár, 4 dollár stb.). Bonnie és Clyde ez esetben is pontosan szeretné kettéosztani a zsákmányt.

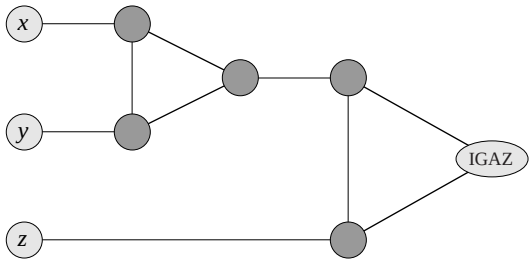
c. A táska n darab csekket rejt, melyeket valami különös véletlen folytán „Bonnie vagy Clyde veheti fel” felirattal láttak el. Az osztozkodás során ismét pontosan szeretnének felezni.

d. Megint ugyanazt az n csekket találják meg, de ezúttal nagyvonalúbb megoldással is beérik: úgy szeretnék elosztani a pénzt, hogy a két rész közti különbség ne legyen nagyobb 100 dollárnál.

34-3. Gráfszínezések

A $G = (V, E)$ gráf k-színezése egy $c : V \longrightarrow \{1, 2, \ldots, k\}$ függvény, amelyre $c(u) \neq c(v)$, ha $(u, v) \in E$. Más szavakkal: minden csúcshoz hozzárendeljük az $1, 2, \ldots, k$ színek valamelyikét úgy, hogy szomszédos csúcsok nem kaphatnak azonos színt. A ***gráfszínezés probléma:*** határozzuk meg a legkisebb k-t, amelyre létezik egy adott G gráfnak k-színezése.

a. Adjunk minél jobb algoritmust egy 2-színezhető gráf 2-színezésére.



34.20. ábra. Az $(x \vee y \vee z)$ klózhoz tartozó segédgráf a 34-3. feladatban.

b. Fogalmazzuk át a gráfszínezés problémát döntési problémává. Mutassuk meg, hogy a kapott döntési probléma pontosan akkor oldható meg polinom időben, ha az eredeti probléma megoldható polinom időben.

c. Legyen 3-SZÍN a 3-színezhető gráfok szabványos kódolásainak nyelve. Mutassuk meg, hogy ha 3-SZÍN NP-teljes, akkor a b. feladat döntési problémája is NP-teljes.

Most bebizonyítjuk, hogy 3-SZÍN NP-teljes. 3-SZÍN $\in$ NP nyilván igaz. Az alábbiakban visszavezetjük 3-SAT-ot 3-SZÍN-re. Az $x_1, x_2, \ldots, x_n$ változókon értelmezett, k klózból álló 3-CNF-beli ϕ formulához megadunk egy $G = (V, E)$ gráfot a következőképp: legyen V-ben egy csúcs minden változóhoz és minden változó negáltjához, 5 csúcs minden klózhoz, és végül 3 speciális csúcs: IGAZ, HAMIS, és PIROS. A gráf élei két típusból kerülnek ki: „literál” élek, melyek nem függnek a klózoktól, és „klóz” élek (ezek – mint sejthető – függenek a klózoktól). A literál élek összekötik a speciális csúcsokat egymással, x_i-t $\neg x_i$-vel és PIROS-sal, továbbá $\neg x_i$-t PIROS-sal, $i = 1, 2, \ldots,$ az $(x_i,\ \neg x_i,$ PIROS) hármasokon.)

d. Igazoljuk, hogy egy, a literál éleket tartalmazó gráf c 3-színezésében egy változó és a negáltja közül az egyik a c(IGAZ), a másik a c(HAMIS) színt kapja. Igazoljuk továbbá, hogy ϕ bármely behelyettesítésére a csak a literál éleket tartalmazó gráf 3-színezhető úgy, hogy az 1-et felvevő változókhoz tartozó szín c(IGAZ), a 0-t felvevőkhöz tartozó pedig c(HAMIS).

A klózok és a változók kapcsolatát most is segédgráfokkal biztosítjuk. Az $(x \vee y \vee z)$ klózhoz tartozó segédgráf látható a 34.20. ábrán. Minden klóz igényli az öt csúcs egy másolatát, melyeket az ábrán sötét színnel jelöltünk. Ezek a literálokat és a speciális IGAZ csúcsot kötik össze.

e. Igazoljuk, hogy ha a 34.20. ábra gráfjának x, y, z csúcsait vagy c(IGAZ)-ra, vagy c(HAMIS)-ra színezzük, akkor segédgráfunk pontosan abban az esetben lesz 3-színezhető, ha x, y, z valamelyikének a c(IGAZ) színt adjuk.

f. Fejezzük be 3-SZÍN NP-teljességének bizonyítását.

34-4. Ütemezés: profitok és határidők

Tegyük fel, hogy van egy gépünk, amivel az $a_1, a_2, \ldots, a_n$ feladatokat szeretnénk elvégezni. Az a_j feladat elvégzéséhez t_j időegység szükséges, befejezésének határideje d_j, a realizálható nyereség pedig p_j. A gépen egyszerre csak egy feladatot tudunk végezni és a feladatokat nem lehet megszakítani. A p_j nyereséget akkor söpörhetjük be, ha az a_j feladatot a d_j határidőre elvégezzük. Ha késünk, akkor nem termelődik nyereség. Feladatunk olyan üte-

mezés elkészítése, amelyben az összes feladatot elvégezzük (nem feltétlenül határidőre), és amely a lehetséges maximális nyereséget termeli.

a. A feladat nyilván egy optimalizálási probléma. Fogalmazzuk meg a megfelelő döntési problémát.

b. Mutassuk meg, hogy a kapott döntési probléma NP-teljes.

c. Adjunk polinomiális algoritmust a döntési problémára abban az esetben, amikor a feladatok elvégzéséhez szükséges idők 1 és n közötti egész számok. (*Útmutatás.* Használjunk dinamikus programozást.)

d. Adjunk polinomiális algoritmust az optimalizálási problémára abban az esetben, amikor a feladatok elvégzéséhez szükséges idők 1 és n közötti egész számok.

Megjegyzések a fejezethez

Garey és Johnson könyve [110] az NP-teljesség elméletének kiváló összefoglalását adja; a téma részletes tárgyalása mellett az 1979-ig NP-teljesnek bizonyult problémák széles skáláját is bemutatja. A 34.13. tétel bizonyítása ebből a könyvből való, csakúgy, mint az NP-teljes problémák előfordulási helyeinek felsorolása a 34.5. alfejezet elején. Johnson 1981 és 1992 között a *Journal of Algorithms* hasábjain tudósított az NP-teljes problémákkal kapcsolatos új fejleményekről. Hopcroft, Motwani és Ullmann [153], Lewis és Papadimitriou [204], Papadimitriou [236], illetve Sipser [279] művei jó bevezetést adnak az NP-teljesség elméletébe a bonyolultságelmélet összefüggésében. Aho, Hopcroft és Ullmann [5] ezen túlmenően jó néhány visszavezetést is ad, köztük a lefedési probléma HAM-ra való visszavezetését.

A P osztályt egymástól függetlenül bevezette 1964-ben Cobham [64] és 1965-ben Edmonds [84] is. Edmonds bevezette az NP osztályt is, és megfogalmazta a $P \neq NP$ sejtést. Az NP-teljesség fogalma Cooktól származik (1971-ből) [67], aki az első NP-teljességi bizonyításokat is adta (SAT-ra és 3-SAT-ra). A fogalmat tőle függetlenül felfedezte Levin is [203], aki egy parkettázási feladat NP-teljességét bizonyította. A visszavezetési módszer Karp [173] nevéhez fűződik (1972). Az ő cikkében szerepel a KLIKK probléma, a lefedési probléma és a Hamilton-kör probléma NP-teljességének első bizonyítása. Azóta problémák százai bizonyultak NP-teljesnek. A Karp hatvanadik születésnapja tiszteletére rendezett összejövetelen Papadimitriou megjegyezte: „Minden évben körülbelül 6000 cikk születik, amelynek címében, összefoglalójában vagy a kulcsszavak között szerepel az *NP-teljes* kifejezés. Ez több, mint ahányszor az *adatbázis*, *operációs rendszer*, *fordító*, *szakértő* vagy *neurális hálózat* kifejezések bármelyike előfordul."

A legújabb bonyolultságelméleti kutatások fényt derítettek számos, közelítő eljárások bonyolultságával kapcsolatos problémára. Új definíció is született az NP osztályra, a „valószínűségi alapon ellenőrizhető bizonyítások" használatával. Kiderült, hogy bizonyos problémák – mint például KLIKK, LEFEDÉS, TSP – esetében jó közelítő megoldások megadása is NP-nehéz. E témába nyerhet betekintést az olvasó Arora disszertációja [19] segítségével vagy az Arora és Lund által írt fejezetből [149], esetleg Mayr, Prömel és Steger könyve [214], illetve Arora [20] és Johnson [167] összefoglaló jellegű cikkei elolvasásával.

35. Közelítő algoritmusok

Számos gyakorlati jelentőségű feladat NP-teljes, de túl fontos ahhoz, hogy pusztán az optimális megoldás megtalálásának kezelhetetlensége miatt ne foglalkozzunk vele. Ha egy feladat NP-teljes, nem valószínű, hogy olyan polinomiális algoritmust találunk, amely pontos megoldást ad, de ez nem jelenti azt, hogy minden remény el van veszve. Három megközelítéssel juthatunk közelebb az NP-teljességhez. Az első, ha a bemenő adatok száma kicsi, akkor egy exponenciális futási idővel rendelkező algoritmus tökéletesen megfelelő lehet. A második lehetőség az, hogy olyan lényeges speciális eseteket találunk, amelyek megoldhatók polinomiális idő alatt. Harmadszor, lehet, hogy polinomiális idő alatt (a legrosszabb vagy átlagos esetben) az *optimálishoz közeli* megoldást találunk. A gyakorlatban az optimálishoz közeli gyakran elég jó. Az optimálishoz közeli megoldást adó algoritmust ***közelítő algoritmusnak*** hívjuk. Ez a fejezet polinomiális közelítő algoritmusokat mutat be néhány NP-teljes feladatra.

A közelítő algoritmusok teljesítménykorlátai

Tételezzük fel, hogy egy optimalizálási feladaton dolgozunk, amelyben mindegyik lehetséges megoldásnak pozitív költsége van, és az optimálishoz közeli megoldást szeretnénk találni. A feladattól függően az optimális megoldást úgy lehetne definiálni, mint a minimális vagy a maximális lehetséges költséggel rendelkezőt: a cél lehet a költség minimalizálása vagy maximalizálása.

Azt mondjuk, hogy $\rho(n)$ egy közelítő algoritmusnak az adott feladatra vonatkozó ***hibakorlát-függvénye,*** ha a közelítő algoritmus által előállított megoldás C költsége – minden n méretű bemenetre – az optimális megoldás C^* költségének legfeljebb $\rho(n)$-szerese és legfeljebb a $\rho(n)$-ed része, azaz

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n). \tag{35.1}$$

Az olyan algoritmust, amely biztosítja a $\rho(n)$ hibakorlát-függvény betartását, ***$\rho(n)$-közelítő algoritmusnak*** nevezzük. A $\rho(n)$ hibakorlát-függvény és a $\rho(n)$-közelítő algoritmus definíciója alkalmazható mind minimalizálási, mind pedig maximalizálási feladatokra. Egy maximalizálási feladatnál $0 < C \leq C^*$ és a C^*/C hányados megadja azt a tényezőt, ahányszor az optimális megoldás költsége nagyobb, mint a közelítő megoldás költsége. Hasonlóképpen, egy minimalizálási feladatnál $0 < C^* \leq C$ és a C/C^* hányados adja meg azt a tényezőt, ahányszor a közelítő megoldás költsége nagyobb, mint az optimális megoldásé. Mivel felté-

telezzük, hogy minden megoldásnak pozitív a költsége, ezek a hányadosok mindig meghatározottak. Egy közelítő algoritmus hibakorlátja soha nem kisebb, mint 1, mivel $C/C^* < 1$ maga után vonja a $C^*/C > 1$ egyenlőtlenséget. Ezért egy 1-közelítő algoritmus[1] optimális megoldást állít elő, viszont a nagy hibakorláttal rendelkező algoritmus az optimálisnál sokkal rosszabb megoldást is eredményezhet.

Számos feladat megoldására ismert kis hibakorlátú, polinomiális futási idejű algoritmus, míg másokra a legjobb ismert polinomiális futási idejű algoritmusok hibakorlátja a bemenet n méretének növekvő függvénye. Az utóbbi esetre példa a 35.3. alfejezetben tárgyalt halmazlefogás.

Az NP-teljes feladatok egy részére ismertek olyan polinomiális futási idejű algoritmusok, amelyek a futási idő növelése árán egyre kisebb hibakorlátot biztosítanak. Azaz szoros összefüggés van a futási idő és a közelítés minősége között. Erre példa a 35.5. alfejezetben tárgyalt részletösszeg feladat. Ez az eset elég fontos ahhoz, hogy saját nevet kapjon.

Egy optimalizálási feladatra a ***közelítő séma*** egy közelítő algoritmus, amelynek bemenete nemcsak a feladat egy konkrét esete, hanem egy $\epsilon > 0$ szám is, úgy, hogy bármely rögzített ϵ-ra a séma egy olyan közelítő algoritmus, amely $(1 + \epsilon)$-közelítő algoritmus. Azt mondjuk, hogy a közelítő séma ***polinomiális idejű közelítő séma*** (röviden: polinomiális séma), ha bármilyen rögzített $\epsilon > 0$ számra a séma futási ideje a konkrét bemenet n méretére nézve polinomiális.

Egy polinomiális közelítő séma futási ideje nagyon gyorsan nőhet, amint ϵ csökken. Például egy polinomiális közelítő séma futási ideje lehet $O(n^{2/\epsilon})$. Ideális esetben, ha ϵ az eredeti érték konstansszorosára csökken, a kívánt közelítés eléréséhez szükséges futási idő nem nő nagyobbra, mint az eredeti konstansszorosa. Más szóval azt szeretnénk, ha a futási idő polinomiális lenne $(1/\epsilon)$-ban és n-ben is.

Egy közelítő sémát ***teljesen polinomiális közelítő sémának*** nevezünk, ha a futási ideje polinomiális $(1/\epsilon)$-ban és a konkrét bemenet n méretében, ahol ϵ a séma relatív hibakorlátja. Például a séma futási ideje lehet $(1/\epsilon)^2 n^3$. Egy ilyen sémával ϵ bármilyen állandószoros csökkenése elérhető a futási idő megfelelő állandószorosra való növelésével.

A fejezet tartalma

A fejezet első négy alfejezete néhány polinomiális közelítő algoritmust mutat be NP-teljes feladatokra, az ötödik alfejezet pedig egy teljesen polinomiális közelítő sémát mutat be. A 35.1. alfejezet a csúcslefedési feladat leírásával kezdődik, ami egy olyan NP-teljes minimalizálási feladat, amelynek megoldására ismerünk $\rho = 2$ hibakorláttal rendelkező közelítő algoritmust. A 35.2. alfejezet egy 2 hibakorlátú közelítő algoritmust mutat be az olyan utazóügynök feladatra, melyben a költségfüggvény kielégíti a háromszög-egyenlőtlenséget. Azt is megmutatja, hogy a háromszög-egyenlőtlenség nélkül ϵ-közelítő algoritmus csak akkor létezhet, ha P = NP. A 35.3. alfejezetben megmutatjuk, hogyan lehet egy mohó eljárást hatékony közelítő algoritmusként alkalmazni a halmazlefogási feladatra, olyan lefogást kapva, melynek költsége legrosszabb esetben egy logaritmikus szorzótényezővel nagyobb az optimális költségnél. A 35.4. alfejezetben két további közelítő algoritmust mutatunk be. Először a 3-CNF kielégíthetőségi feladat optimalizálási változatát elemezzük és bemutatunk egy egyszerű véletlenített algoritmust, amelyre várható hibája legfeljebb 8/7. Azután a

[1] Ha a relatív hibakorlát-függvény független n-től, ennek jelzésére a ρ hibakorlát és a ρ-közelítő algoritmus kifejezéseket fogjuk használni.

csúcslefedési feladat súlyozott változatát vizsgáljuk és megmutatjuk, hogyan alkalmazható a lineáris programozás arra, hogy 2-közelítő algoritmust kapjunk. Végül a 35.5. alfejezet egy teljesen polinomiális közelítő sémát mutat be a részletösszeg feladatra.

35.1. Minimális lefedő csúcshalmaz

A 34.5.2. pontban definiáltuk a minimális lefedő csúcshalmaz feladatot, és bebizonyítottuk annak NP-teljességét. Egy $G = (V, E)$ irányítatlan gráf ***csúcslefedése*** egy $V' \subseteq V$ csúcshalmaz, amelyre igaz, hogy ha (u, v) G éle, akkor vagy $u \in V'$ vagy $v \in V'$ (vagy mindkettő). A lefedés mérete a benne lévő csúcsok száma.

A ***minimális lefedő csúcshalmaz feladat*** az, hogy egy adott irányítatlan gráfban meg kell találni a minimális méretű lefedést. Ezt a lefedést hívjuk ***optimális lefedésnek.*** Ez a feladat egy NP-teljes döntési feladat optimalizálási változata.

Noha egy G gráfban optimális lefedést találni nehéz lehet, egy közel optimálisat találni nem túl nehéz. A következő közelítő algoritmus bemenete egy G irányítatlan gráf, és olyan lefedést ad, amelynek mérete biztosan nem nagyobb, mint az optimális érték kétszerese.

KÖZ-LEFEDÉS(G)

```
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4        do legyen (u, v) E' tetszőleges éle
5           C ← C ∪ {u, v}
6           távolítsunk el E'-ből minden u-ra vagy v-re illeszkedő élt
7  return C
```

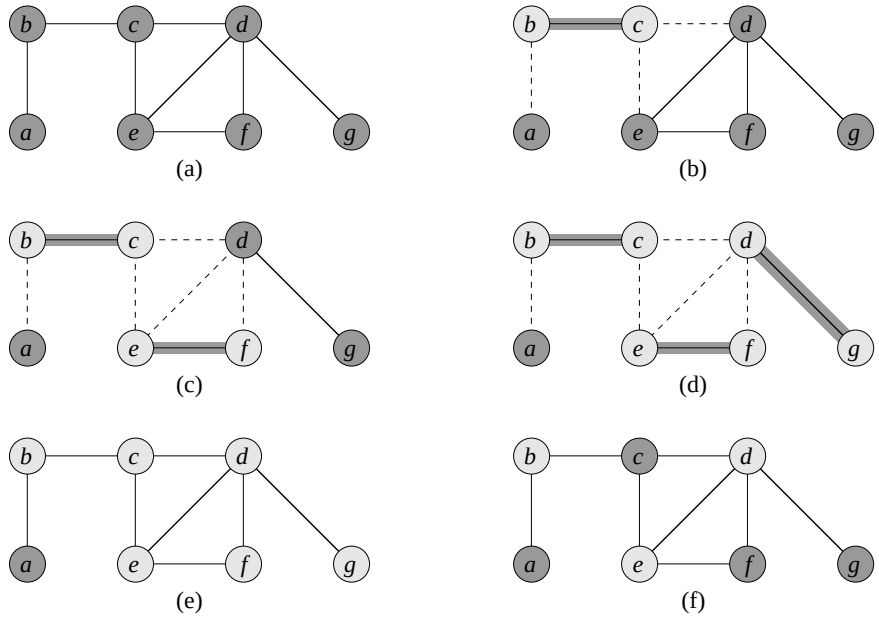
A 35.1. ábra a KÖZ-LEFEDÉS működését mutatja. A C változó tartalmazza a kialakuló lefedést. Az 1. sor C-nek az üres halmazt adja kezdeti értékként. A 2. sor elhelyezi E'-ben a gráf $E[G]$ élhalmazának másolatát. A ciklus a 3–6. sorban ismételten kiválaszt egy (u, v) élt E'-ből, az u és v végpontokat hozzáadja C-hez, és E'-ben kitöröl minden olyan élt, amelyet akár u, akár v lefed. Ennek az algoritmusnak a futási ideje $O(V + E)$, feltéve, hogy E' ábrázolására megfelelő adatszerkezetet használunk.

35.1. tétel. KÖZ-LEFEDÉS *polinomiális 2-közelítő algoritmus.*

Bizonyítás. Azt már megmutattuk, hogy a KÖZ-LEFEDÉS algoritmus polinomiális.

A KÖZ-LEFEDÉS által előállított csúcsok C halmaza egy lefedés, mivel az algoritmus addig ismétli a ciklusmagot, amíg $E[G]$ minden éle lefedődik egy C-ben lévő csúccsal.

Annak belátásához, hogy a KÖZ-LEFEDÉS olyan lefedést ad, amelynek mérete legfeljebb kétszerese az optimális lefedésének, A jelentse az élek azon halmazát, melyet a KÖZ-LEFEDÉS a 4. sorban választott. A éleinek lefedéséhez minden csúcslefedésnek – egy optimális C^* lefedésnek is – tartalmaznia kell A minden élének legalább egy végpontját. A-ban egyik élpárnak sincs közös végpontja, mivel ha az algoritmus egy élt kiválaszt a 4. sorban, akkor minden olyan élt, amelynek végpontjai illeszkednek a kiválasztott él végpontjaira, a 6. sorban kitöröl E'-ből. Ezért A-ban egyik élpár sincs ugyanazon C^*-beli csúccsal lefedve, ezért



35.1. ábra. A KÖZ-LEFEDÉS működése. **(a)** A G bemenő gráf, melynek 7 csúcsa és 8 éle van. **(b)** A vastag vonallal jelölt (b, c) él az első él, melyet a KÖZ-LEFEDÉS választott. A világosan árnyékolt b és c csúcsok az C halmazhoz adódnak hozzá, mely az alakuló lefedést tartalmazza. A szaggatott vonallal jelölt (a, b), (c, e) és (c, d) éleket eltávolítottuk, mivel C-ben már le vannak fedve. **(c)** Az (e, f) élt hozzáadjuk A-hoz. **(d)** A (d, g) élt hozzáadjuk A-hoz. **(e)** Az A halmaz a KÖZ-LEFEDÉS által létrehozott lefedés: 6 csúcsot (b, c, d, e, f, g) tartalmaz. **(f)** Az optimális lefedés ebben az esetben csak 3 csúcsot (b, d, e) tartalmaz.

az optimális csúcslefedés méretére a következő alsó korlátot kapjuk:

$$|C^*| \geq |A|. \tag{35.2}$$

A 4. sor minden végrehajtása pontosan egy olyan élt választ ki, amelynek egyik végpontja sem tartozik C-hez, amiből a csúcslefedés méretére a következő – pontos – felső korlát adódik

$$|C| = 2|A|. \tag{35.3}$$

A (35.2) és (35.3) képletek összekapcsolásával adódik, hogy

$$|C| = 2|A| \leq 2|C^*|,$$

és ezzel a tételt bebizonyítottuk. ■

Elemezzük ezt a bizonyítást. Érdekes kérdés, hogyan lehetséges annak bizonyítása, hogy KÖZ-LEFEDÉS olyan csúcslefedést talált, melynek mérete legfeljebb kétszerese az optimális lefedés méretének, amikor nem is ismerjük az optimális csúcslefedés méretét. A válasz az, hogy felhasználjuk az optimális lefedés méretének egy alsó korlátját. A 35.1-2. gyakorlat célja annak bizonyítása, hogy azoknak az éleknek az A halmaza, melyeket KÖZ-LEFEDÉS a 4. sorban kiválaszt, valójában a G gráf egy maximális párosítását adják. A ***maximális párosítás*** olyan párosítás, amely egyetlen más párosításnak sem valódi részhalmaza. A 35.1. tétel bizonyításában azzal érveltünk, hogy a maximális párosítás mérete az

optimális csúcslefedés méretének alsó korlátja. Az algoritmus olyan csúcslefedést ad meg, melynek mérete legfeljebb kétszere az A maximális párosítás méretének. Ha a megoldás méretét az alsó korláthoz viszonyítjuk, megkapjuk a kívánt relatív hibakorlátot.

Gyakorlatok

35.1-1. Adjunk meg egy olyan gráfot, amelyikre a KÖZ-LEFEDÉS mindig szuboptimális megoldást ad.

35.1-2. Legyen A azoknak az éleknek a halmaza, melyeket a KÖZ-LEFEDÉS algoritmus a 4. lépésben kiválasztott. Bizonyítsuk be, hogy az A halmaz a G gráf egy maximális párosítása.

35.1-3.⋆ Nixon professzor a következő heurisztikát javasolja a lefedési feladat megoldására. Ismételten válasszuk ki a legnagyobb fokszámú csúcsot, és távolítsuk el a rá illeszkedő éleket. Adjunk egy példát, amely megmutatja, hogy a professzor algoritmusának 2 nem hibakorlátja. *Útmutatás.* Próbálkozzunk egy olyan páros gráffal, amelyben a bal oldalon azonos a csúcsok fokszáma, a jobb oldalon viszont különböző.

35.1-4. Adjunk egy hatékony mohó algoritmust, amely egy fára lineáris időben optimális lefedést ad.

35.1-5. A 34.12. tétel bizonyításából tudjuk, hogy a lefedési és az NP-teljes klikk feladat komplementerek abban az értelemben, hogy az optimális lefedés a komplemens gráfban lévő maximális méretű teljes részgráf komplemense. Jelenti-e ez a kapcsolat, hogy létezik közelítő algoritmus állandó hibakorláttal a klikk feladatra? Indokoljuk a választ.

35.2. Az utazóügynök feladat

A 34.5.4. pontban bemutatott utazóügynök feladatban egy teljes irányítatlan $G = (V, E)$ gráfunk van, amelyben minden $(u, v) \in E$ élhez $c(u, v)$ nemnegatív egész költséget rendelünk, és meg kell találni egy minimális költségű H Hamilton-kört (egy körutat). Jelölésünk kiterjesztéseként legyen $c(A)$ az $A \subseteq E$ részhalmaz éleinek összköltsége:

$$c(A) = \sum_{(u,v)\in A} c(u, v).$$

Számos gyakorlati helyzetben mindig az u helyről közvetlenül w helyre menni a legolcsóbb: akármelyik közbeeső v állomás útba ejtése nem lehet olcsóbb. Másképp fogalmazva, egy közbeeső állomás kihagyása soha nem növeli a költséget. Ezt az elképzelést úgy formalizáljuk, hogy azt mondjuk, hogy a c költségfüggvény kielégíti a ***háromszög-egyenlőtlenséget,*** ha minden $u, v, w \in V$ csúcsra

$$c(u, w) \leq c(u, v) + c(v, w).$$

A háromszög-egyenlőtlenség természetes, és sok alkalmazásban automatikusan teljesül. Például, ha a gráf csúcsai a sík pontjai, és a két csúcs közötti utazás költsége a szokásos euklideszi távolság, akkor a háromszög-egyenlőtlenség teljesül. (Vannak más, az euklideszi távolságtól különböző távolságfüggvények is, amelyekre teljesül a háromszög-egyenlőtlenség.)

Ahogy a 35.2-2. gyakorlat mutatja, az utazóügynök feladat akkor is NP-teljes marad, ha megköveteljük, hogy a költségfüggvény kielégítse a háromszög-egyenlőtlenséget. Ezért nem valószínű, hogy olyan polinomiális algoritmust találjunk, amely pontosan megoldja a feladatot. Ezért inkább egy jó közelítő algoritmust keresünk.

A 35.2.1. pontban egy olyan közelítő algoritmust vizsgálunk az utazóügynök feladatra (a háromszög-egyenlőtlenséget megkívánva), melynek 2 hibakorlátja. A 35.2.2. pontban megmutatjuk, hogy háromszög-egyenlőtlenség nélkül állandó hibakorlátú közelítő algoritmus csak akkor létezik, ha P = NP.

35.2.1. Az utazóügynök feladat háromszög-egyenlőtlenséggel

Az előző alfejezet módszerét alkalmazva először meghatározunk egy minimális feszítőfát – melynek súlya alsó korlát az utazóügynök optimális körútjának hosszára. Azután ezt a minimális feszítőfát felhasználjuk arra, hogy olyan körutat állítsunk elő, melynek hossza nem nagyobb, mint a minimális feszítőfa súlyának kétszerese – amennyiben a súlyok kielégítik a háromszög-egyenlőtlenséget. A következő algoritmus ezt a megközelítést alkalmazza, a 23.2. alfejezetben tárgyalt MFF-Prim minimális feszítőfa algoritmust szubrutinként használva.

Köz-uü-körút(G, c)

1 válasszunk ki egy $r \in V[G]$ csúcsot, ami egy „gyökércsúcs” lesz
2 határozzuk meg a G gráf r gyökerű T minimális feszítőfáját
az MFF-Prim (G, c, r) felhasználásával
3 legyen l a már meglátogatott csúcsok listája T preorder fa bejárásának sorrendjében
4 **return** a H Hamilton-kör, amely a csúcsokat L sorrendjében tartalmazza

A 12.1. alfejezet szerint egy preorder fabejárás minden csúcsot rekurzívan látogat meg, minden csúcsot az első találkozáskor felsorolva, mielőtt akármelyik gyerekét meglátogatná.

A 35.2. ábra a Köz-uü-körút működését mutatja. Az ábra (a) része a csúcsok adott halmazát, a (b) része pedig a T minimális feszítőfát mutatja, amint azt az MFF-Prim az a gyökércsúcsból megnövelte. A (c) rész azt mutatja, hogy T preorder bejárása hogyan látogatja a csúcsokat, és a (d) rész a megfelelő körutat jeleníti meg, amelyet a Köz-uü-körút állít elő. Az (e) rész egy optimális körutat mutat be, amely körülbelül 23%-kal rövidebb.

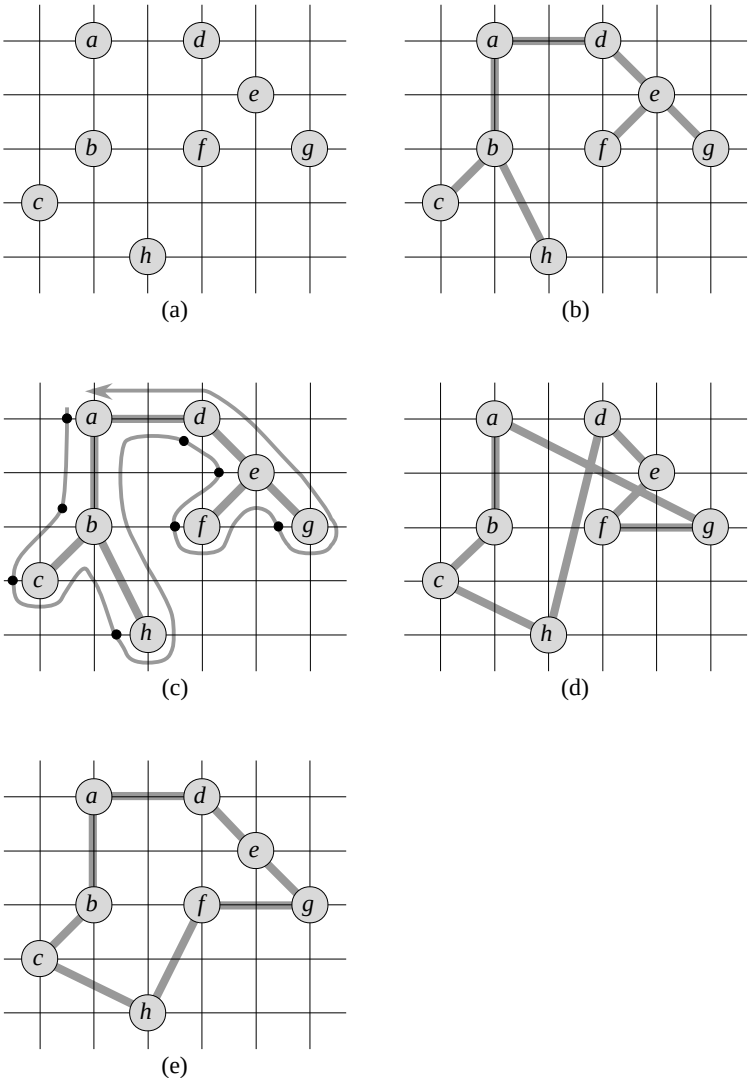
A 23.2-2. gyakorlat szerint a Köz-uü-körút futási ideje $\Theta(E) = \Theta(V^2)$. Megmutatjuk, hogy ha a költségfüggvény az utazóügynök feladat egy konkrét esetére kielégíti a háromszög-egyenlőtlenséget, akkor a Köz-uü-körút olyan körutat ad, amelynek költsége legfeljebb az optimális út költségének kétszerese.

35.2. tétel. *Ha a háromszög-egyenlőtlenség teljesül az utazóügynök feladatnál, akkor a* Köz-uü-körút *polinomiális 2-közelítő algoritmus.*

Bizonyítás. Azt már beláttuk, hogy Köz-uü-körút futási ideje polinomiális.

Legyen H^* egy optimális körút csúcsok adott halmazán. Mivel a körútból egy élt eltávolítva feszítőfát kapunk, a T minimális feszítőfa súlya alsó korlát az optimális körút hosszára, azaz

$$c(T) \leq c(H^*). \tag{35.4}$$



35.2. ábra. A Köz-UÜ-KÖRÚT működése. **(a)** Adott pontok halmaza, amelyek egy egész koordinátájú rács csúcsain fekszenek. Például, f h-tól egy egységgel jobbra és két egységgel feljebb található. Költségfüggvényként a két pont közötti közönséges euklideszi távolságot használjuk. **(b)** Ezen pontoknak az MFF-PRIM által meghatározott T minimális feszítőfája. Az a csúcs a gyökércsúcs. A csúcsok úgy vannak elnevezve, ahogy azokat MFF-PRIM a főfához ábécé-sorrendben hozzáadja. **(c)** A T bejárás, amely a-ban kezdődik. A fa teljes bejárása a csúcsokat a következő sorrendben látogatja meg: $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A T preorder bejárása minden csúcsot az első találkozáskor sorol fel, az a, b, c, h, d, e, f, g sorrendet eredményezve. **(d)** A csúcsoknak a preorder körútnak megfelelő sorrendben való meglátogatásával kapott csúcskörút. Ez a Köz-UÜ-KÖRÚT által adott H bejárás. Összes költsége körülbelül 19,074. **(e)** A H^* optimális körút a csúcsok adott halmazára. Ennek összes költsége körülbelül 14,715.

T ***teljes bejárása*** felsorolja a csúcsokat mind az első látogatáskor, mind pedig egy részfa bejárása utáni visszatéréskor. Jelöljük ezt a bejárást W-vel. Példánk teljes bejárása az

$$a,\ b,\ c,\ b,\ h,\ b,\ a,\ d,\ e,\ f,\ e,\ g,\ e,\ d,\ a$$

sorrendet adja. Mivel a teljes bejárás T minden élén pontosan kétszer halad át, ezért

$$c(W) = 2c(T). \tag{35.5}$$

A (35.4) és a (35.5) sorokból következik, hogy

$$c(W) \leq 2c(H^*), \tag{35.6}$$

és így W költsége legfeljebb az optimális körút költségének kétszerese.

Sajnos, W általában nem egy körút, mivel bizonyos csúcsokat egynél többször is meglátogat. A háromszög-egyenlőtlenség miatt W bármelyik csúcsának látogatását törölhetjük a költség növekedése nélkül. (Ha W-ből töröljük u és w között a v csúcsot, az eredményül kapott sorrend u-ból közvetlenül w-be vezető utat határoz meg.) Ismételten alkalmazva ezt a műveletet, W-ből minden második és későbbi előfordulást eltávolíthatunk. Példánkban ez a következő sorrendet eredményezi:

$$a,\ b,\ c,\ h,\ d,\ e,\ f,\ g.$$

Ez a sorrend ugyanaz, amit a T fa preorder bejárása során kaptunk. H legyen ennek a bejárásnak megfelelő körút. Ez egy Hamilton-kör, mivel minden csúcsot pontosan egyszer látogat meg, és tulajdonképpen ez a KÖZELÍTŐ-UÜ-KÖRÚT által számított körút. Mivel H-t úgy kaptuk, hogy a W teljes bejárásból csúcsokat töröltünk, tehát

$$c(H) \leq c(W). \tag{35.7}$$

A (35.6) és a (35.7) egyenlőtlenségek összevonása teljessé teszi a bizonyítást. ■

A 35.2. tétellel kapott szép aránykorlát ellenére a KÖZ-UÜ-KÖRÚT általában nem a legjobb gyakorlati választás erre a feladatra. Vannak más közelítő algoritmusok, amelyek rendszerint sokkal jobb eredményt adnak a gyakorlatban (a hivatkozásokat lásd ennek a fejezetnek a végén).

Az általános utazóügynök feladat

Ha elvetjük a feltételezést, hogy a c költségfüggvény kielégíti a háromszög-egyenlőtlenséget, nem találhatunk jó közelítő körutakat polinomiális időben, kivéve, ha P = NP.

35.3. tétel. *Ha* P ≠ NP, *akkor minden* $\rho \geq 1$ *állandóra teljesül, hogy nincs polinomiális* ρ*-közelítő algoritmus az általános utazóügynök feladat megoldására.*

Bizonyítás. Indirekt bizonyítást alkalmazunk. Tegyük fel, hogy egy $\rho \geq 1$ számra létezik A polinomiális, ρ aránykorlátú közelítő algoritmus. Az általánosság megszorítása nélkül feltételezzük, hogy ρ egész szám (ha szükséges, kerekítünk). Ekkor megmutatjuk, hogyan használjuk A-t Hamilton-kör feladat (amit a 34.5.5. pontban definiáltunk) konkrét eseteinek polinomiális megoldására. Mivel a Hamilton-kör a 34.14. tétel alapján NP-teljes, polinomiális megoldása a 34.4. tétel szerint azt jelenti, hogy P = NP.

Legyen $G = (V, E)$ példa a Hamilton-kör feladatra. Azt kívánjuk hatékonyan eldönteni a feltételezett A közelítő algoritmus felhasználásával, hogy G tartalmaz-e Hamilton-kört. G-t

az utazóügynök feladat konkrét esetévé tesszük a következőképpen. Legyen $G' = (V, E')$ a teljes gráf V-n, azaz

$$E' = \{(u, v) : u, v \in V \text{ és } u \neq v\}.$$

Jelöljünk ki E'-ben minden élhez egy egész értékű költséget a következőképpen:

$$c(u, v) = \begin{cases} 1, & \text{ha } (u, v) \in E, \\ \rho|V| + 1, & \text{egyébként.} \end{cases}$$

G' és c ábrázolásait $|V|$-ben és $|E|$-ben polinomiális időben létrehozhatjuk G ábrázolásából.

Most nézzük a (G', c) utazóügynök feladatot. Ha az eredeti G gráfnak van H Hamilton-köre, akkor a c költségfüggvény H minden éléhez 1 költséget rendel, így (G', c) tartalmaz egy $|V|$ költségű körutat. Másrészt, ha G nem tartalmaz egy Hamilton-kört, akkor G' bármilyen bejárásának használnia kell nem E-ben lévő élt is. De bármelyik körútnak, amely nem E-ben lévő élt is használ, a költsége legalább

$$\begin{aligned} \rho(|V| + 1) + (|V| - 1) &= \rho|V| + |V| \\ &> \rho|V|. \end{aligned}$$

Mivel a nem G-ben lévő élek ilyen költségesek, legalább $\rho|V|$ a különbség egy olyan körút költsége, amely G-ben Hamilton-kör (a költség $|V|$), és bármely más bejárás költsége (a költség legalább $\rho|V| + |V|$).

Mi történik, ha az A közelítő algoritmust alkalmazzuk a (G', c) utazóügynök feladatra? Mivel biztos, hogy A olyan körutat ad, amelynek költsége nem nagyobb, mint az optimális körút költségének ρ-szorosa, ha G tartalmaz Hamilton-kört, akkor A-nak ezt meg kell találnia. Ha G-ben nincs Hamilton-kör, akkor A olyan körutat ad, amelynek költsége $\rho|V|$-nél nagyobb. Ezért A-t használhatjuk a Hamilton-kör feladat polinomiális megoldására. ■

A 35.3. tétel bizonyítása példa egy olyan általános módszerre, mellyel bizonyítható, hogy egy feladat megoldására nincs jó közelítő algoritmus. Tegyük fel, hogy egy X NP-nehéz feladathoz polinomiális idő alatt meg tudunk adni egy olyan Y minimalizálási feladatot, hogy X „igen” esetei megfelelnek Y legfeljebb k-értékű eseteinek (ahol k adott állandó), X „nem” esetei viszont Y olyan eseteinek felelnek meg, amelyek értéke nagyobb, mint ρk. Ezzel megmutattuk, hogy – kivéve, ha P = NP – az Y feladat megoldására nincs polinomiális idejű ρ-közelítő algoritmus.

Gyakorlatok

35.2-1. Tegyük fel, hogy a $G = (V, E)$ irányítatlan gráfnak legalább 3 csúcsa van és olyan c költségfüggvénye, amely kielégíti a háromszög-egyenlőtlenséget. Bizonyítsuk be, hogy ekkor $c(v, u) \geq 0$ minden $u,\ v \in V$ csúcspárra.

35.2-2. Mutassuk meg, hogyan alakíthatjuk át az utazóügynök feladat konkrét esetét egy másik esetté, melynek költségfüggvénye kielégíti a háromszög-egyenlőtlenséget. A két esetben az optimális körutak halmazainak meg kell egyezniük. Magyarázzuk meg, hogy egy ilyen polinomiális transzformáció miért nem mond ellent a 35.3. tételnek, feltételezve, hogy P ≠ NP.

35.2-3. Vizsgáljuk meg a következő ***legközelebbi-csúcs heurisztikát*** egy közelítő utazóügynök körút építéséhez (tegyük fel, hogy teljesül a háromszög-egyenlőtlenség). Kezdjük egy

triviális körrel, amely egy, tetszőlegesen választott csúcsból áll. Minden lépésnél azonosítsuk az u csúcsot, amely nincs a körön, de távolsága a kör hozzá legközelebb eső csúcsától a minimális. Tegyük fel, hogy az u csúcshoz legközelebbi csúcs a körön a v csúcs. Terjesszük ki a kört úgy, hogy u-t közvetlenül v után beszúrjuk. Ismételjük ezt addig, amíg minden csúcs a körön nem lesz. Bizonyítsuk be, hogy ez a heurisztika olyan körutat ad, amelynek költsége legfeljebb az optimális bejárás költségének kétszerese.

35.2-4. Az ***üvegnyak utazóügynök feladat*** olyan Hamilton-kör keresése, hogy a kör leghosszabb élének hossza a minimális legyen. Feltételezve, hogy a költségfüggvény kielégíti a háromszög-egyenlőtlenséget, mutassuk meg, hogy létezik 3 hibakorlátú polinomiális közelítő algoritmus erre a feladatra. (*Útmutatás.* Mutassuk meg rekurzívan, hogy egy üvegnyak feszítőfán meglátogathatunk minden csúcsot pontosan egyszer, alapul véve a fa teljes bejárását, és átugorva a csúcsokat anélkül, hogy 2 közbülső szomszédos csúcsot ugranánk át.)

35.2-5. Tételezzük fel, hogy az utazóügynök feladat egy konkrét esetének csúcsai pontok a síkban, és a $c(u, v)$ költség az u és v közötti euklideszi távolság. Mutassuk meg, hogy az optimális bejárás soha sem keresztezi önmagát.

35.3. A minimális lefogó részhalmaz

A halmazlefogási feladat egy optimalizálási feladat, amely sok erőforrás-kiválasztási feladatot modellez. Az NP-teljes csúcslefedési feladatot általánosítja, és ezért szintén NP-nehéz. A csúcslefedési feladat kezelésére kifejlesztett közelítő algoritmus azonban itt nem alkalmazható, így más megközelítést kell megpróbálnunk. Egy egyszerű mohó heurisztikát vizsgálunk meg, melynek hibakorlátja logaritmikus. Azaz, ahogy a példa mérete nő, a közelítő megoldás mérete nőhet az optimális megoldás méretéhez képest. Mivel a logaritmusfüggvény elég lassan nő, ez a közelítő algoritmus ennek ellenére adhat hasznos eredményeket.

A ***lefogási feladat*** egy (X, F) konkrét esete egy X véges halmazból és X részhalmazainak olyan $\mathcal{F}$ családjából áll, hogy X minden eleme legalább egy $\mathcal{F}$-beli részhalmazhoz tartozik:

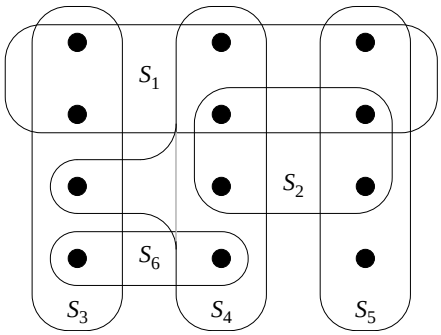
$$X = \bigcup_{S \in \mathcal{F}} S.$$

Azt mondjuk, hogy egy $S \in \mathcal{F}$ részhalmaz ***lefogja*** az elemeit. A feladat az, hogy olyan minimális méretű $C \subseteq \mathcal{F}$ részhalmazt találjunk, amelynek elemei az egész X-et lefedik:

$$X = \bigcup_{S \in C} S. \tag{35.8}$$

Azt mondjuk, hogy bármely C, amely kielégíti a (35.8) egyenletet, ***lefedi*** X-et. A 35.3. ábra szemlélteti a feladatot. C méretét a benne lévő halmazok számával – nem pedig a benne lévő elemek számával – definiáljuk. A 35.3. ábrán a minimális halmazlefogás mérete 3.

A lefogási feladat sok rendszeresen felmerülő kombinatorikai feladat általánosítása. Egyszerű példaként tegyük fel, hogy X olyan képességek halmazát jelöli, amelyek szükségesek egy feladat megoldásához, és adott azon emberek halmaza, akik a feladaton dolgozhatnak. Egy bizottságot szeretnénk létrehozni, amely a lehető legkisebb számú emberből áll



35.3. ábra. A lefogási feladat $(X, \mathcal{F})$ esete, ahol X a 12 fekete pontot tartalmazza, és $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimális méretű halmazlefedés $C = \{S_3, S_4, S_5\}$. A mohó algoritmus 4 méretű lefedést állít elő, a halmazokat S_1, S_4, S_5 és S_3 sorrendben kiválasztva.

úgy, hogy X-ben minden szükséges képességhez legyen egy ember a bizottságban, aki rendelkezik azzal a képességgel. A lefogási feladat döntési változatában azt kérdezzük, hogy létezik-e legfeljebb k méretű lefedés, ahol k kiegészítő paraméter, melyet a konkrét feladatban szintén megadunk. A feladat döntési változata a 35.3-2. gyakorlat szerint NP-teljes.

Egy mohó közelítő algoritmus

A mohó módszer úgy működik, hogy minden lépésnél kiválasztja azt az S halmazt, amelyik a legtöbb, még lefedetlen elemet fedi le.

MOHÓN-HALMAZT-LEFOG$(X, \mathcal{F})$

```
1  U ← X
2  C ← ∅
3  while U ≠ ∅
4        do válasszunk ki egy olyan (S ∈ F)-et, amelyre |S ∩ U| maximális
5           U ← U − S
6           C ← C ∪ {S}
7  return C
```

A 35.3. ábra példájában a MOHÓN-HALMAZT-LEFOG C-hez sorrendben az S_1, S_4, S_5, S_3 halmazokat adja hozzá.

Az algoritmus a következőképpen dolgozik. Az U halmaz minden lépésnél a még lefedetlen elemeket tartalmazza. A C halmaz az alakuló lefedést tartalmazza. A 4. sor a mohó döntéshozó lépés. Egy S részhalmazt választ a lehető legtöbb, még lefedetlen elem lefedésére (a holtversenyt tetszőlegesen dönthetjük el). Miután S-et kiválasztotta, annak elemeit eltávolítja U-ból, és S-et elhelyezi C-ben. Amikor az algoritmus véget ér, a C halmaz magában foglalja $\mathcal{F}$-nek az X-et lefogó részcsaládját.

A MOHÓN-HALMAZT-LEFOG algoritmus könnyen megvalósítható úgy, hogy $|X|$ és $|\mathcal{F}|$ szerint polinomiális időben fusson. Mivel a 3–6. sorokban lévő ciklus végrehajtásainak száma legfeljebb $\min(|X|, |\mathcal{F}|)$, és a ciklusmag megvalósítható úgy, hogy $O(|X||\mathcal{F}|)$ idő alatt lefusson, ezért van olyan megvalósítás, amely $O(|X||\mathcal{F}|)$ min $(|X|, |\mathcal{F}|))$ időben fut. A 35.3-3. gyakorlat lineáris futási idejű algoritmust kér.

Elemzés

Most megmutatjuk, hogy a mohó algoritmus olyan lefogást ad, amely nem sokkal nagyobb, mint az optimális halmazlefogás. Az egyszerűség kedvéért ebben a fejezetben a $H_d = \sum_{i=1}^{d} 1/i$ (lásd az A.1. alfejezetben) d-edik harmonikus számot $H(d)$-vel jelöljük. Peremfeltételként a $H(0) = 0$ definíciót használjuk.

35.4. tétel. MOHÓN-HALMAZT-LEFOG *polinomiális* $\rho(n)$*-közelítő algoritmus, ahol*

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}).$$

Bizonyítás. Azt már beláttuk, hogy MOHÓN-HALMAZT-LEFOG futási ideje polinomiális.

Annak belátásához, hogy MOHÓN-HALMAZT-LEFOG $\rho(n)$-közelítő algoritmus, az algoritmus által kiválasztott halmazokhoz 1 költséget rendelünk, majd az első alkalommal lefogott elemek között elosztjuk ezt a költséget, azután ezeket a költségeket arra használjuk, hogy C^* optimális lefogás mérete és az algoritmus által adott C lefogás mérete közötti kívánt kapcsolatot levezessük. Jelöljük S_i-vel a MOHÓN-HALMAZT-LEFOG által kiválasztott i-edik részhalmazt; az algoritmus költsége 1, amikor S_i-t hozzáadja C-hez. Az S_i kiválasztásának költségét egyenlően elosztjuk az S_i által első alkalommal lefogott elemek között. Legyen c_x az x elemhez rendelt költség, minden $x \in X$ elemre. Minden elemhez csak egyszer rendelünk költséget, amikor először van lefogva. Ha S_i első alkalommal fogja le x-et, akkor

$$c_x = \frac{1}{|S_i - (S_i \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Az algoritmus minden lépésénél egységnyi költség merül fel, ezért

$$|C| = \sum_{x \in X} c_x.$$

Az optimális lefogáshoz rendelt költség

$$\sum_{S \in C^*} \sum_{x \in S} c_x,$$

és mivel minden $x \in X$ esetén van legalább egy $S \in C^*$, így

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x.$$

Az utolsó két egyenlőtlenségből következik, hogy

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x. \tag{35.9}$$

A bizonyítás hátralévő része a következő kulcsegyenlőtlenségen alapszik, amit rövidesen bizonyítunk. Az $\mathcal{F}$ családhoz tartozó bármelyik S halmazra

$$\sum_{x \in S} \leq H(|S|). \tag{35.10}$$

A (35.9) és (35.10) egyenlőtlenségekből következik, hogy

$$\begin{aligned} |C| &\leq \sum_{S \in C^*} H(|S|) \\ &\leq |C^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}), \end{aligned}$$

és ezzel a tételt bebizonyítottuk.

Már csak a (35.10) egyenlőtlenség bebizonyítása maradt hátra. Tekintsünk tetszőleges $S \in \mathcal{F}$ halmazt ($i = 1, 2, \ldots, |C|$), és legyen

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$$

S azon elemeinek száma, amelyek lefedetlenek azután, hogy az algoritmus kiválasztotta az $S_1, S_2, \ldots, S_i$ halmazokat. Legyen $u_0 = |S|$ az S halmaz olyan elemeinek száma, amelyek kezdetben lefedetlenek. Legyen k a legkisebb olyan index, amelyre $u_k = 0$, így S minden eleme le van fedve az $S_1, S_2, \ldots, S_k$ halmazok közül legalább eggyel. Ekkor $u_{i-1} \geq u_i$ és S_i ($i = 1, 2, \ldots, k$) S-nek $u_{i-1} - u_i$ elemét fedi le először. Így

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Vegyük észre, hogy

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| &\geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \\ &= u_{i-1}, \end{aligned}$$

mivel S_i mohó választása biztosítja, hogy S nem fedhet le több új elemet, mint S_i (egyébként S_i helyett S-et választottuk volna). Következésképpen azt kapjuk, hogy

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} .$$

Ennek a mennyiségnek egy felső korlátját a következőképpen kapjuk:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\ &= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\ &\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && \text{(mivel } j \leq u_{i-1}\text{)} \\ &= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\
&= H(u_0) - H(u_k) && \text{(mivel teleszkópos összegzés)} \\
&= H(u_0) - H(0) \\
&= H(u_0) && \text{(mivel } H(0) = 0\text{)} \\
&= H(|S|),
\end{aligned}
$$

és ezzel a (35.10) egyenlőtlenség bizonyítását befejeztük. ■

35.5. következmény. MOHÓN-HALMAZT-LEFOG *polinomiális* $(\ln |X| + 1)$*-közelítő algoritmus.*

Bizonyítás. Használjuk az (A.14) egyenlőtlenséget és a 35.4. tételt. ■

Bizonyos alkalmazásokban $\max\{|S| : S \in \mathcal{F}\}$ egy kis állandó, így a MOHÓN-HALMAZT-LEFOG által adott eredmény az optimálisnak legfeljebb egy kis állandószorosa. Egy ilyen alkalmazás fordul elő, amikor ezt a heurisztikát egy legfeljebb harmadfokú csúcsokkal rendelkező gráf közelítő csúcslefogására használjuk. Ebben az esetben a MOHÓN-HALMAZT-LEFOG által talált megoldás nem nagyobb, mint az optimális megoldás $H(3) = 11/6$-szorosa. Ez a teljesítménygarancia egy árnyalatnyival jobb annál, amit a KÖZ-LEFED esetében kaptunk.

Gyakorlatok

35.3-1. Vegyük a {arid, dash, drain, heard, lost, nose, shun, slate, snare, thread} szavakat betűk halmazának. Mutassuk meg, hogy a MOHÓN-HALMAZT-LEFOG melyik lefogást állítja elő, amikor a holtversenyt annak a szónak a javára döntjük el, amelyik előbb jelenik meg a szótárban.

35.3-2. Mutassuk meg a csúcslefedési feladat visszavezetésével, hogy a lefogási feladat döntési változata NP-teljes.

35.3-3. Mutassuk meg, hogy MOHÓN-HALMAZT-LEFOG megvalósítható úgy, hogy $O(\sum_{S \in \mathcal{F}} |S|)$ időben fusson.

35.3-4. Mutassuk meg, hogy a 35.4. tétel következő, gyengébb formája triviálisan igaz:

$$|C| \le |C^*| \max\{|S| : S \in \mathcal{F}\}.$$

35.3-5. MOHÓN-HALMAZT-LEFOG több különböző megoldást adhat attól függően, hogyan döntjük el a holtversenyt a 4. sorban. Adjunk meg egy olyan ROSSZUL-HALMAZT-LEFOG-ESET(n) eljárást, amely előállítja a halmazlefogási feladat olyan esetét, amelyre – annak függvényében, hogyan döntjük el a holtversenyt a 4. sorban – a MOHÓN-HALMAZT-LEFOG által előállított különböző megoldások száma n exponenciális függvénye.

35.4. Véletlenítés és lineáris programozás

Ebben az alfejezetben két olyan technikát tanulmányozunk, amelyek hasznosak a közelítő algoritmusok tervezésében: a véletlenítést és a lineáris programozást. Először egy egyszerű véletlenített algoritmust mutatunk be a 3-CNF kielégíthetőség optimalizálási változatának megoldására, majd a lineáris programozás segítségével közelítő algoritmust tervezünk a

csúcslefogási feladat súlyozott változatának megoldására. Ez az alfejezet csupán érinti ennek a két hatásos technikának a felületét.

Véletlenített közelítő algoritmus a 3-SAT kielégíthetőségi feladatra

Éppen úgy, ahogy a pontos megoldások előállítására, a közelítő megoldások előállítására is vannak véletlenített algoritmusok. Azt mondjuk, hogy egy véletlenített algoritmusnak egy adott feladat megoldására vonatkozó ***hibakorlátja*** $\rho(n)$, ha a véletlenített algoritmus által szolgáltatott megoldás C *várható* költsége az optimális megoldás C^* költségének legfeljebb a $\rho(n)$-szerese és legalább annak $\rho(n)$-ed része:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n). \tag{35.11}$$

A $\rho(n)$ hibakorlát betartását biztosító véletlenített algoritmust ***$\rho(n)$-közelítő véletlenített algoritmusnak*** is nevezzük. Más szavakkal, a véletlenített közelítő algoritmus hasonlít a determinisztikus közelítő algoritmusra, azzal a különbséggel, hogy a hibakorlát a várható értékre vonatkozik.

A 34.4. alfejezetben definiált 3-SAT kielégíthetőségi probléma egy konkrét esete lehet kielégíthető és nem kielégíthető is. A kielégíthetőséghez szükséges, hogy legyen a változóknak olyan behelyettesítése, amelyre minden klóz értéke 1. Ha egy eset nem kielégíthető, akkor szükségünk lehet arra, mennyire van „közel" a kielégíthetőséghez, azaz kereshetünk olyan behelyettesítést, amely a lehető legtöbb klózt kielégíti. Az így kapott maximalizálási feladatot ***MAX-3-CNF-kielégíthetőségnek*** nevezzük. MAX-3-CNF bemenete ugyanaz, mint 3-CNF bemenete, a cél pedig az, hogy a változók olyan behelyettesítésének a meghatározása, amelyre maximális az 1 értékű klózok száma. Megmutatjuk, hogy a minden változó értékét véletlenül 1/2 valószínűséggel nullára és 1/2 valószínűséggel egyre állító algoritmus véletlenített 8/7-közelítő algoritmus. A 3-CNF-kielégíthetőségnek a 34.4. alfejezetben adott definíciója szerint előírjuk, hogy minden klóz pontosan három különböző literált tartalmazzon. Továbbá feltesszük, hogy egyik klóz sem tartalmaz egyszere egy változót és annak negáltját is. (A 35.4-1. gyakorlat célja ennek a feltételnek a kiküszöbölése.)

35.6. tétel. *Az a véletlenített algoritmus, amelyik az $x_1, x_2, \ldots, x_n$ változókat és m klózt tartalmazó MAX-3-CNF-kielégíthetőségi problémát úgy oldja meg, hogy a változóknak egymástól függetlenül 1/2 valószínűséggel 0 és 1/2 valószínűséggel 1 értéket ad, véletlenített 8/7-közelítő algoritmus.*

Bizonyítás. Tegyük fel, hogy minden változónak 1/2 valószínűséggel a 0 és 1/2 valószínűséggel az 1 értéket adjuk. Definiáljuk az $i = 1, 2, \ldots, n$ indexekre az

$$Y_i = \mathrm{I}\{\text{az } i \text{ klóz ki van elégítve}\}$$

indikátor valószínűségi változót úgy, hogy $Y_i = 1$, ha az i-edik klózban legalább egy literál értéke 1. Mivel ugyanabban a klózban egyik literál sem fordul elő egynél többször, és feltettük, hogy egyik klózban sem fordul elő valamely változó és annak a negáltja is, a 3 literál értéke minden klózban egymástól független. Egy klóz csak akkor nincs kielégítve, ha mindhárom literál értéke 0, ezért $\Pr\{\text{az } i \text{ klóz nincs kielégítve}\} = (1/2)^3 = 1/8$. Ezért

$\Pr\{\text{az } i \text{ ki van elégítve}\} = 1 - 1/8 = 7/8$. Ezért az 5.1. lemma szerint $\mathrm{E}[Y_i] = 7/8$. Legyen Y a kielégített klózok száma – ekkor $Y = Y_1 + Y_2 + \cdots + Y_m$. Ebből következik, hogy

$$\begin{aligned}\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} \mathrm{E}[Y_i] \quad \text{(a várható érték linearitása miatt)} \\ &= \sum_{i=1}^{m} \frac{7}{8} \\ &= \frac{7m}{8}\,.\end{aligned}$$

Mivel m nyilvánvalóan a kielégített klózok számának felső korlátja, a hibakorlát legfeljebb $m/(7m/8) = 8/7$. ■

Közelítő súlyozott minimális csúcslefedés megoldása lineáris programozással

A ***minimális csúcslefedési problémában*** $G = (V, E)$ irányítatlan gráf, melyben minden $v \in V$ csúcshoz hozzárendeljük annak $w(v)$ pozitív súlyát. Minden $V' \subseteq V$ csúcslefedésre $w(V') = \sum_{v \in V'} w(v)$ a csúcslefedés súlya. A cél a minimális súlyú lefedés meghatározása.

A súlyozatlan csúcslefedés meghatározására alkalmazott algoritmust most nem tudjuk felhasználni, és a véletlenített módszert sem; mindkét módszer olyan megoldást ad, amely távol van az optimálistól. Egy lineáris program segítségével azonban alsó korlátot tudunk adni a minimális súlyú csúcslefedésre. Ezután „kerekítjük" ezt a megoldást és felhasználjuk a csúcslefedés előállítására.

Tegyük fel, hogy minden $v \in V$ csúcshoz hozzárendelünk egy $x(v)$ változót, és megköveteljük, hogy $x(v) \in \{0, 1\}$ teljesüljön minden $v \in V$ csúcsra. Az $x(v) = 1$ egyenlőséget úgy értelmezzük, hogy v benne van a csúcslefedésben, az $x(v) = 0$ egyenlőséget pedig úgy, hogy nincs. Ekkor azt a korlátozást, hogy minden (u, v) élre az u és v csúcsok közül legfeljebb egy lehet a csúcslefedésben, úgy írhatjuk fel, hogy $x(u) + x(v) \geq 1$. Ily módon a következő ***0-1 egészértékű programot*** kapjuk a minimális csúcslefedés meghatározására:

$$\text{minimalizálandó} \quad \sum_{v \in V} w(v)x(v) \tag{35.12}$$

feltéve, hogy

$$x(u) + x(v) \geq 1, \qquad \text{ha } (u, v) \in E, \tag{35.13}$$

$$x(v) \in \{0, 1\}, \qquad \text{ha } v \in V \tag{35.14}$$

A 34.5-2. gyakorlat szerint tudjuk, hogy a (35.13) és a (35.14) egyenlőtlenségeket kielégítő $x(v)$ meghatározása NP-nehéz, ezért ez a megfogalmazás nem feltétlenül hasznos. Tegyük fel azonban, hogy elhagyjuk az $x(v) \in \{0, 1\}$ korlátozást, és helyette a $0 \leq x(v) \leq 1$ korlátot alkalmazzuk. Ekkor a következő lineáris programot kapjuk, amely ***lineáris programozási közelítés*** néven ismert:

minimalizálandó $\sum_{v \in V} w(v)x(v)$ (35.15)

feltéve, hogy

$$x(u) + x(v) \geq 1, \quad \text{ha } (u, v) \in E \tag{35.16}$$

$$x(v) \leq 1, \quad \text{ha } v \in V \tag{35.17}$$

$$x(v) \geq 0, \quad \text{ha } v \in V. \tag{35.18}$$

A (35.12)–(35.14) sorokban megfogalmazott 0-1 egészértékű program bármely megengedett megoldása egyúttal megoldása a (35.15)–(35.18) sorokban megfogalmazott lineáris programnak is. Ezért a lineáris program optimális megoldása alsó korlátot ad a 0-1 egészértékű programozási feladat optimális megoldására, és így alsó korlát a minimális csúcslefedési problémára is.

A következő eljárás a fenti lineáris program megoldását használja fel arra, hogy előállítsa a minimális csúcslefedési probléma közelítő megoldását.

KÖZ-MIN-SÚLY-CSÚCSLEFED(G, w)

```
1  C ← 0
2  számítsuk ki a (35.15)–(35.18) sorokban megadott
                                  lineáris program x̄ optimális megoldását
3  for minden v ∈ V csúcsra
4      do if x̄(v) ≥ 1/2
5          then C ← C ∪ {v}
6  return C
```

A KÖZ-MIN-SÚLY-CSÚCSLEFED eljárás a következőképpen működik. Az 1. sor beállítja a lefedő halmaz kezdőértékét, az üres halmazt. A 2. sor megfogalmazza és megoldja a (35.15)–(35.18) sorokban szereplő lineáris programot. Egy optimális megoldás minden v csúcshoz hozzárendel egy $\overline{x}(v)$ értéket, amelyre $0 \leq \overline{x}(v) \leq 1$. Ezt az értéket felhasználjuk annak eldöntésére, hogy melyik csúcsokat adjuk hozzá C-hez a 3–5. sorokban. Ha $\overline{x}(v) \geq 1/2$, akkor v-t hozzáadjuk C-hez, egyébként nem. Ennek eredménye, hogy a lineáris program megoldásában szereplő tört változókat „kerekítjük” 0-ra vagy 1-re, hogy a (35.12)–(35.14) sorokban lévő 0-1 egészértékű program egy megoldását kapjuk. Végül a 6. sorban megkapjuk a C csúcslefedést.

35.7. tétel. KÖZ-MIN-SÚLY-CSÚCSLEFED *polinomiális futási idejű 2-közelítő algoritmus a minimális súlyú csúcslefedési feladat megoldására.*

Bizonyítás. Mivel a 2. sorban megadott lineáris program megoldására van polinomiális algoritmus, és a 3–5. sorokban lévő **for** ciklus polinomiális ideig tart, KÖZ-MIN-SÚLY-CSÚCSLEFED polinomiális algoritmus.

Most megmutatjuk, hogy KÖZ-MIN-SÚLY-CSÚCSLEFED 2-közelítő algoritmus. Legyen $C^\star$ a minimális csúcslefedési probléma egy optimális megoldása, és legyen $z^\star$ a (35.15)–(35.18) sorokban lévő lineáris program egy optimális megoldása. Mivel az optimális csúcslefedés a lineáris program egy megengedett megoldása, $z^\star$ egy alsó korlát $w(C^\star)$-ra, azaz

$$z^* \leq w(C^*). \tag{35.19}$$

Ezután azt állítjuk, hogy az $\bar{x}(v)$ változók történtékének a kerekítésével olyan C halmazt állítunk elő, amely csúcslefedés és kielégíti a $w(C^*) \leq 2z^*$ feltételt. Annak belátásához, hogy C csúcslefedés, tekintsük bármely $(u, v) \in E$ élet. A (35.16) korlát szerint $x(u) + x(v) \geq 1$, amiből következik, hogy legalább egy $\bar{x}(u)$ és legalább egy $\bar{x}(v)$ legalább 1/2. Ezért legalább egy u és legalább egy v benne van a csúcslefedésben, és így minden él le lesz fedve.

Tekintsük a lefedés súlyát. Ekkor

$$\begin{aligned} z^* &= \sum_{v \in V} w(v)\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\ &= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in C} w(v) \\ &= \frac{1}{2} w(C). \end{aligned} \tag{35.20}$$

A (35.19) és (35.20) képleteket felhasználva

$$w(C) \leq 2z^* \leq 2w(C^*) ,$$

ahonnan már adódik, hogy Köz-MIN-SÚLY-CSÚCSLEFED 2-közelítő algoritmus. ■

Gyakorlatok

35.4-1. Mutassuk meg, hogy a változóknak 1/2 valószínűséggel 0 és 1/2 valószínűséggel 1 értéket adva még akkor is véletlenített 8/7-közelítő algoritmust kapunk, ha megengedjük, hogy a klózok egy változóval együtt annak negáltját is tartalmazhassák.

35.4-2. A ***MAX-CNF kielégíthetőségi feladat*** hasonló a MAX-3-CNF feladathoz; a különbség az, hogy itt nem szerepel az a korlátozás, hogy minden klóz legfeljebb 3 literált tartalmazhat. Adjunk meg egy 2-közelítő véletlenített algoritmust a MAX-CNF kielégíthetőségi feladat megoldására.

35.4-3. A MAX-CUT problémában adott egy $G = (V, E)$ súlyozatlan, irányítatlan gráf. Az $(S, V - S)$ vágást úgy definiáljuk, mint a 23. fejezetben, a ***vágás súlyát*** pedig úgy, mint a vágást metsző élek számát. Célunk a maximális vágás meghatározása. Tegyük fel, hogy minden v csúcsot véletlenül és egymástól függetlenül 1/2 valószínűséggel az S és 1/2 valószínűséggel a $V - S$ csúcshalmazba tesszük. Mutassuk meg, hogy ez egy véletlenített 2-közelítő algoritmus.

35.4-4. Mutassuk meg, hogy a (35.17) feltétel redundáns abban az értelemben, hogy ha elhagyjuk ezt a sort a (35.15)–(35.18) sorokban lévő lineáris programból, az így kapott lineáris program minden optimális megoldásának ki kell elégítenie az $x(v) \leq 1$ feltételt minden $v \in V$ csúcsra.

35.5. A részletösszeg feladat

A részletösszeg feladatra példa egy (S, t) pár, ahol S pozitív egész számok $\{x_1, x_2, \ldots, x_n\}$ halmaza és t egész szám. Ebben a feladatban azt kell eldöntenünk, hogy létezik-e S-nek olyan részhalmaza, amelyben az elemek összege pontosan a t célérték. Ez a feladat NP-teljes (lásd a 34.5.5. pontot).

Az ezzel a döntési feladattal kapcsolatos optimalizációs feladat gyakorlati alkalmazásokban is felmerül. Az optimalizációs feladatban olyan $\{x_1, x_2, \ldots, x_n\}$ részhalmazt szeretnénk találni, amelynek összege a lehető legnagyobb, de nem nagyobb, mint t. Például van egy teherautó, amely t kilogrammnál és n különböző doboznál nem tud többet szállítani, az i-edik doboz súlya x_i kilogramm. Szeretnénk a lehető legjobban megtölteni a teherautót, anélkül hogy az adott súlyhatárt átlépnénk.

Ebben az alfejezetben bemutatunk egy exponenciális idejű algoritmust erre az optimalizációs feladatra, azután megmutatjuk, hogyan módosítsuk az algoritmust, hogy teljesen polinomiális közelítő sémává váljon. (Emlékezzünk arra, hogy a teljesen polinomiális közelítő séma futási ideje $(1/\epsilon)$-ban és n-ben is polinomiális.)

Egy exponenciális idejű pontos algoritmus

Tegyük fel, hogy S minden S' részhalmazára kiszámítottuk S' elemeinek összegét, azután pedig azon halmazok közül, melyek összege nem lépte túl t-t, kiválasztottuk azt, amelynek összege a legnagyobb. Ez az algoritmus természetesen optimális megoldást ad, de exponenciális ideig fut. Ennek az algoritmusnak a megvalósításához olyan iteratív eljárást használunk, amely az i-edik iterációban az $\{x_1, x_2, \ldots, x_i\}$ halmaz összes részhalmazának összegét határozza meg, kiindulásként felhasználva az $\{x_1, x_2, \ldots, x_{i-1}\}$ összes részhalmazának összegét. Ha így járunk el, észrevesszük, hogy egy S' részhalmaz összege túllépte t-t, akkor nincs értelme tovább foglalkozni vele, mivel S' szuperhalmaza nem lehet optimális megoldás. Most megadjuk ennek a stratégiának egy megvalósítását.

A PONTOS-RÉSZLETÖSSZEG eljárás bemenete egy $S = \{x_1, x_2, \ldots, x_n\}$ halmaz és egy t célérték. Ez az eljárás iteratív módon számolja ki az $\{x_1, x_2, \ldots, x_i\}$ t-nél nem nagyobb összegű részhalmazainak összegét tartalmazó L_i listákat, majd L_n-ben megadja a maximális értéket.

Ha L pozitív egészek listája és x egy további pozitív egész, akkor $L + x$ jelölje az L-ből L minden elemének x-szel való növelésével nyert egészek listáját. Például, ha $L = \langle 1, 2, 3, 5, 9 \rangle$, akkor $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. Ezt a jelölést használjuk halmazokra is, így

$$S + x = \{s + x : s \in S\}.$$

A LISTÁKAT-ÖSSZEFÉSÜL(L, L') kiegészítő eljárást alkalmazzuk, amely a két rendezett bemenő L és L' lista összefésülését adja. Mint az ÖSSZEFÉSÜL eljárás, amit az összefésülő rendezésben használtunk (1.3.1. pont), a LISTÁKAT-ÖSSZEFÉSÜL is $O(|L| + |L'|)$ időben fut. (Nem adunk meg pszeudokódot a LISTÁKAT-ÖSSZEFÉSÜL-re.)

PONTOS-RÉSZLETÖSSZEG(S, t)

```
1  n ← |S|
2  L_0 ← ⟨0⟩
3  for i ← 1 to n
4      do L_i ← LISTÁKAT-ÖSSZEFÉSÜL(L_{i-1}, L_{i-1} + x_i)
5         távolítsunk el L_i-ből minden olyan elemet, amely t-nél nagyobb
6  return L_n-ben a legnagyobb elem
```

Az algoritmus működésének megértéséhez P_i jelölje azon értékek halmazát, melyeket egy (esetleg üres) $\{x_1, x_2, \ldots, x_i\}$ részhalmaz kiválasztásával és tagjainak összeadásával nyerhetünk. Például, ha $S = \{1, 4, 5\}$, akkor

$$\begin{aligned} P_1 &= \{0, 1\}, \\ P_2 &= \{0, 1, 4, 5\}, \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\}. \end{aligned}$$

A

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \tag{35.21}$$

azonosság segítségével i-re vonatkozó indukcióval (lásd a 35.5-1. gyakorlatot) bizonyítható, hogy az L_i lista P_i minden elemét tartalmazó rendezett lista, melynek értéke nem nagyobb, mint t. Mivel L_i hossza akár 2^i is lehet, PONTOS-RÉSZHALMAZ-ÖSSZEG egy exponenciális algoritmus, noha polinomiális algoritmus azokban a speciális esetekben, amikor t $|S|$-ben polinomiális, vagy S minden eleme $|S|$ polinomjával korlátozható.

Egy teljesen polinomiális közelítő séma

Teljesen polinomiális közelítő sémát kaphatunk a részletösszeg feladatra, ha létrehozása után minden L_i listát „megritkítunk”. Az alapötlet az, hogy ha L két eleme közel van egymáshoz, akkor a közelítő megoldáshoz nincs szükség mind a kettőre. Pontosabban, olyan δ ritkító paramétert használunk, amelyre $0 < \delta < 1$. Az L lista δ-val való ***ritkítása*** azt jelenti, hogy L-ből a lehető legtöbb elemet távolítjuk el úgy, hogy ha L' L ritkításának eredménye, akkor minden L-ből eltávolított y elemre még van egy olyan z elem L'-ben, amely y közelítő értéke, azaz

$$\frac{y}{1+\delta} \le z \le y. \tag{35.22}$$

Ezt a z-t tekinthetjük y „képviselőjének” az új L' listában. Minden y-t képvisel egy olyan z, amelyre teljesül (35.22). Ha például $\delta = 0{,}1$ és

$$L' = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle,$$

akkor L-et megritkíthatjuk, hogy az

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle$$

listát kapjuk, ahol a kitörölt 11 értéket 10 képviseli, a kitörölt 21 és 22 értéket 20 képviseli,

és a kitörölt 24 értéket 23 képviseli. Fontos emlékeznünk arra, hogy a lista ritkított változatának minden értéke eleme a lista eredeti verziójának is. A lista ritkítása jelentős mértékben csökkentheti egy lista elemeinek számát, miközben egy közeli (és némileg kisebb) képviselő értéket megtart minden listából kitörölt elem számára.

A következő eljárás egy $L = \langle y_1, y_2, \ldots, y_m \rangle$ bemeneti listát ritkít $\Theta(m)$ időben, feltételezve, hogy L nemcsökkenő sorrendbe van rendezve. Az eljárás kimenete egy ritkított, rendezett lista.

```
RITKÍT(L, δ)
1  m ← |L|
2  L' ← ⟨y_1⟩
3  utolsó ← y_1
4  for i ← 2 to m
5      do if y_i > utolsó · (1 + δ)          ▷ y_i ≥ utolsó, mivel L rendezve van
6            then adjuk y_i-t L' végéhez
7                 utolsó ← y_i
8  return L'
```

L elemeit növekvő sorrendben pásztázzuk, és egy szám csak akkor kerül a készülő L' listába, ha L első eleme, vagy az L'-be került utolsó szám nem képviselheti.

A RITKÍT eljárással a következőképpen hozhatjuk létre a közelítő sémánkat. Az eljárás bemenete az n egész számból álló $S = \{x_1, x_2, \ldots, x_n\}$ halmaz (tetszőleges sorrendben), a t egész célszám és az ϵ „közelítő paraméter”, ahol

$$0 < \epsilon < 1. \tag{35.23}$$

Az eljárás kimenete egy olyan z érték, amelynek az optimális megoldás legfeljebb $(1 + \epsilon)$-szorosa.

```
KÖZ-RÉSZLETÖSSZEG(S, t, ε)
1  n ← |S|
2  L_0 ← ⟨0⟩
3  for i ← 1 to n
4      do L_i ← LISTÁKAT-ÖSSZEFÉSÜL(L_{i-1}, L_{i-1} + x_i)
5         L_i ← RITKÍT (L_i, ε/n)
6         távolítsunk el L_i-ból minden olyan elemet, amely t-nél nagyobb
7  legyen z* a legnagyobb érték L_n-ben
8  return z*
```

A 2. sor beállítja a lista L_0 kezdeti értékét (a lista ekkor csak a 0 elemet tartalmazza). A 3–6. sorokban lévő ciklus hatása L_i kiszámítása, mint P_i halmaz megfelelően ritkított változatát tartalmazó rendezett lista, melynek összes eleme nagyobb, mint az eltávolított t. Mivel L_i-t L_{i-1}-ből hoztuk létre, meg kell bizonyosodnunk arról, hogy az ismételt ritkítás nem eredményez túl nagy pontatlanságot. Rövidesen látni fogjuk, hogy a KÖZELÍTŐ-RÉSZLETÖSSZEG megfelelő közelítést ad, ha ilyen létezik.

Példaként feltételezzük, hogy az

$$L = \langle 104, 102, 201, 101 \rangle$$

konkrét esetben $t = 308$ és $\epsilon = 0,40$. A δ ritkítási paraméter $\epsilon/8 = 0,05$. A KÖZELÍTŐ-RÉSZLETÖSSZEG algoritmus a jelölt sorokban a következő értékeket számolja:

2. sor: $L_0 = \langle 0 \rangle$,

4. sor: $L_1 = \langle 0, 104 \rangle$,
5. sor: $L_1 = \langle 0, 104 \rangle$,
6. sor: $L_1 = \langle 0, 104 \rangle$,

4. sor: $L_2 = \langle 0, 102, 104, 206 \rangle$,
5. sor: $L_2 = \langle 0, 102, 206 \rangle$,
6. sor: $L_2 = \langle 0, 102, 206 \rangle$,

4. sor: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
5. sor: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
6. sor: $L_3 = \langle 0, 102, 201, 303 \rangle$,

4. sor: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
5. sor: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
6. sor: $L_4 = \langle 0, 101, 201, 302 \rangle$.

Az algoritmus a $z^* = 302$-t adja válaszként, amely az optimális $307 = 104 + 102 + 101$ választól $\epsilon = 40\%$-nál jóval kevesebbel tér el; valójában 2%-on belül van.

35.8. tétel. *A* KÖZELÍTŐ-RÉSZLETÖSSZEG *egy teljesen polinomiális közelítő séma a részletösszeg feladat megoldására.*

Bizonyítás. Az 5. sorban L_i ritkítása és L_i-ből a t-nél nagyobb elemek eltávolítása megtartja azt a tulajdonságot, hogy L_i minden eleme egyúttal P_i tagja is. Ezért a 8. sorban kapott z^* érték valójában S bizonyos részhalmazainak összege. Legyen $y^* \in P_n$ a részletösszeg feladat egy optimális megoldása. Ekkor a 6. sor szerint $z^\star \leq y^\star$. A 35.1. egyenlőtlenség szerint azt kell megmutatnunk, hogy $y^\star/z^\star \leq 1 + \epsilon$. Azt is meg kell mutatnunk, hogy az algoritmus futási ideje mind $(1/\epsilon)$-nak, mind a bemenet méretének polinomiális függvénye.

i szerinti indukcióval megmutatható, hogy minden P_i-beli y-ra, azaz t-nél nem nagyobb elemre van olyan $z \in L_i$, amelyre

$$\frac{y}{(1 + \epsilon/2n)^i} \leq z \leq y \tag{35.24}$$

(lásd a 35.5-2. gyakorlatot). A (35.24) egyenlőtlenségnek teljesednie kell, ha $y^* \in P_n$, és így van olyan $z \in L_n$, amelyre

$$\frac{y^*}{(1 + \epsilon/2n)^n} \leq z \leq y^*,$$

és így

$$\frac{y^*}{z} \leq \left(1 + \frac{\epsilon}{2n}\right)^n. \tag{35.25}$$

Mivel van olyan $z \in L_n$, amelyre fennáll a (35.25) egyenlőtlenség, ennek az egyenlőtlenségnek teljesednie kell L_n legnagyobb elemére, azaz z^*-ra; tehát

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n . \tag{35.26}$$

Már csak azt kell belátni, hogy $y^*/z^* \leq 1 + \epsilon$. Ehhez megmutatjuk, hogy $(1 + \epsilon/2n)^n \leq 1 + \epsilon$. A (3.13) egyenlőség szerint $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$. Mivel belátható, hogy

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0, \tag{35.27}$$

ezért a $(1 + \epsilon/2n)^n$ függvény n növekvő függvénye és a $e^{\epsilon/2}$ értékhez tart, ezért

$$\begin{aligned}\left(1 + \frac{\epsilon}{2n}\right)^n &\leq e^{\epsilon/2} \\ &\leq 1 + \frac{\epsilon}{2} + \left(\frac{\epsilon}{2}\right)^2 && \text{((3.12) egyenlőtlenség)} \\ &\leq 1 + \epsilon && \text{((35.23) egyenlőtlenség).}\end{aligned} \tag{35.28}$$

A (35.26) és (35.28) egyenlőtlenségek összekapcsolása teljessé teszik a hibakorlát elemzését.

Ahhoz, hogy megmutassuk, hogy KÖZELÍTŐ-RÉSZLETÖSSZEG teljesen polinomiális közelítő séma, megadunk egy L_i hosszára vonatkozó korlátot. Ritkítás után L_i két, egymást követő eleme – z és z' – között fenn kell állnia a $z/z' > 1/(1 - \epsilon/n)$ egyenlőtlenségnek. Azaz legalább $1+\epsilon/2n$ tényezőben különbözniük kell egymástól. Minden lista tartalmazza a 0 elemet, esetleg az 1 elemet és esetleg még legfeljebb $\lfloor \log_{1+\epsilon/2n} t \rfloor$ további értéket. Az elemek száma minden L_i-ben legfeljebb

$$\begin{aligned}\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1 + \epsilon/2n)} + 2 \\ &\leq \frac{2n(1 + \epsilon/2n)\ln t}{\epsilon} + 2 && \text{((3.16) egyenlőtlenség)} \\ &\leq \frac{4n \ln t}{\epsilon} + 2 && \text{((35.23) egyenlőtlenség).}\end{aligned}$$

Ez a korlát polinomiális az adott bemenet méretében – ami a t ábrázolásához szükséges bitek $\lg t$ számából és az S halmaz ábrázolásához szükséges bitek számából áll, ami polinomiális n-ben – és így $1/\epsilon$-ban. Mivel KÖZELÍTŐ-RÉSZLETÖSSZEG futási ideje polinomiális L_i hosszában, teljesen polinomiális közelítő séma. ■

Gyakorlatok

35.5-1. Bizonyítsuk be a (35.21) egyenletet. Ezután mutassuk meg, hogy PONTOS-RÉSZLETÖSSZEG 5. sorának végrehajtása után L_i olyan rendezett lista, amely P_i minden olyan elemét tartalmazza, melynek értéke legfeljebb t.

35.5-2. Bizonyítsuk be a (35.24) egyenletet.

35.5-3. Bizonyítsuk be a (35.27) egyenletet.

35.5-4. Hogyan módosítsuk az ebben a fejezetben bemutatott közelítő sémát, hogy a legkisebb, t-nél nem kisebb olyan értékre jó közelítést találjon, amely az adott bemenő lista bizonyos részhalmazában lévő elemek összege?

Feladatok

35-1. Ládapakolás

Tegyük fel, hogy adott egy n tárgyból álló halmaz, ahol az i-edik tárgy s_i méretére $0 < s_i < 1$. Az összes tárgyat szeretnénk bepakolni minimális számú egységnyi méretű ládákba. Minden ládába a tárgyak bármely részhalmaza belefér, amelynek összmérete nem haladja meg az 1-et.

a. Bizonyítsuk be, hogy a szükséges ládák minimális számát meghatározó feladat NP-nehéz. (*Útmutatás.* Vezessük vissza a részletösszeg feladatból.)

A ***first-fit*** közelítő algoritmus sorban vesz minden tárgyat, és belerakja az első olyan ládába, amelyikbe belefér. Legyen $S = \sum_{i=1}^{n} s_i$.

b. Mutassuk meg, hogy a szükséges ládák optimális száma legalább $\lceil S \rceil$.

c. Mutassuk meg, hogy a first-fit heurisztika legfeljebb egy ládát hagy a felénél kevésbé megtöltve.

d. Bizonyítsuk be, hogy a first-fit heurisztika által felhasznált ládák száma soha nem több, mint $\lceil 2S \rceil$.

e. Bizonyítsuk be, hogy a first-fit heurisztikára 2 egy hibakorlát.

f. Valósítsuk meg a first-fit közelítő algoritmust és elemezzük a megvalósított algoritmus futási idejét.

35-2. Maximális klikk méretének közelítése

Legyen $G = (V, E)$ egy irányítatlan gráf. Bármely $k \geq 1$ értékre definiáljuk $G^{(k)}$-t úgy, mint $(V^{(k)}, E^{(k)})$ irányítatlan gráfot, ahol $V^{(k)}$ a V-beli csúcsokból képezett rendezett k-asok halmaza, és E^k-t úgy definiáljuk, hogy $(v_1, v_2, \ldots, v_k)$ akkor és csak akkor szomszédos $(w_1, w_2, \ldots, w_k)$-val, ha valamely v_i csúcs szomszédos w_i-vel G-ben, vagy $v_i = w_i$.

a. Bizonyítsuk be, hogy $G^{(k)}$-ban a maximális klikk mérete egyenlő G-ben a maximális klikk méretének a k-adik hatványával.

b. Bizonyítsuk be, hogy ha van állandó hibakorlátú közelítő algoritmus maximális méretű teljes gráf keresésére, akkor létezik teljesen polinomiális közelítő séma is a feladat megoldására.

35-3. Súlyozott halmazlefogási feladat

Tegyük fel, hogy általánosítjuk a halmazlefogási feladatot úgy, hogy az $\mathcal{F}$ családhoz tartozó minden S_i halmazhoz hozzárendeljük a w súlyt, és a C lefogás súlya $\sum_{S_i \in C} w_i$. A minimális súlyú lefogást szeretnénk meghatározni. (A 35.3. alfejezet foglalkozik azzal az esettel, melyben $w_i = 1$ minden i-re.)

Mutassuk meg, hogy a mohó halmaz lefogási heurisztikát általánosítani lehet természetes módon annak érdekében, hogy közelítő megoldást adjon a súlyozott halmazlefogási feladat bármely esetére. Mutassuk meg, hogy ennek a heurisztikának $H(d)$ hibakorlátja, ahol d az S_i-ben lévő halmazok maximális mérete.

35-4. Legnagyobb párosítás

Emlékezzünk arra, hogy a párosítás egy G irányítatlan gráfban olyan élek halmaza, melyek közül bármelyik kettőre igaz, hogy nincs közös végpontjuk. A 26.3. alfejezetben láttuk,

hogyan határozható meg egy legnagyobb párosítás páros gráfokban. Ebben a feladatban az általános esetet vizsgáljuk (azaz nem tételezzük fel, hogy az irányítatlan gráfok párosak).

a. A ***maximális párosítás*** olyan párosítás, amely nem valódi részhalmaza egyetlen másik párosításnak sem. Azzal bizonyítsuk be, hogy egy maximális párosítás nem szükségképpen legnagyobb párosítás, hogy megadunk egy G irányítatlan gráfot és annak egy M maximális párosítását, amely nem legnagyobb párosítás. (Létezik ilyen 4 csúcsú gráf.)

b. Adjunk meg egy olyan mohó algoritmust, amely $O(E)$ idő alatt meghatározza egy $G(V, E)$ gráf egy maximális párosítását.

Ebben a feladatban a legnagyobb párosítás közelítő meghatározására koncentrálunk. Míg a legnagyobb párosítás meghatározására alkalmas leggyorsabb algoritmus szuperlineáris (bár polinomiális), az itt tárgyalt algoritmus lineáris. Itt meg fogjuk mutatni, hogy a (b) részben szereplő, a maximális párosítást meghatározó algoritmus 2-közelítő algoritmus a legnagyobb párosítás meghatározására.

c. Mutassuk meg, hogy G minden legnagyobb párosításának mérete alsó korlát G bármely csúcslefedésének méretére.

d. Tekintsük $G(V, E)$ egy maximális párosítását. Legyen

$$T = \{v \in V : \text{ van } M\text{-ben } v\text{-re illeszkedő él.}\}$$

Mit mondhatunk G azon részgráfjáról, amelyet G-nek azok a csúcsai indukálnak, amelyek nem elemei T-nek?

e. A (d) rész alapján vezessük le, hogy G csúcslefedésének mérete $2|M|$.

f. A (c) és (e) részeket felhasználva mutassuk meg, hogy a (b) rész szerinti algoritmus 2-közelítő algoritmus a legnagyobb párosítás meghatározására.

35-5. Párhuzamos ütemezés

A ***párhuzamos-ütemezési-problémában*** adott $(J_1, J_2, \ldots, J_n)$ munkákat kell elvégezni az $M_1, M_2, \ldots, M_m$ gépeken. A gépek azonosak, a J_k munka elvégzési ideje t_k $(k = 1, 2, \ldots, n)$. A munkák ***ütemezése*** minden J_k munkára megadja, hogy az adott munkát melyik gépen és milyen időintervallumban kell elvégezni. A J_k munka elvégzéséhez valamely M_i gépen p_k egymás utáni időegységre van szükség, és ebben az időintervallumban az M_i gépen más munka elvégzése nem folyhat. Legyen C_k az az időpont, amelyben a J_k munka végrehajtása befejeződik. Adott ütemezés $C_{\max}$ ***hosszát*** a $C_{\max} = \max_{1 \le i \le n}$ előírással definiáljuk. A cél a minimális hosszúságú ütemezés meghatározása.

Tegyük fel például, hogy adott az M_1 és M_2 gép, továbbá a J_1, J_2, J_3 és J_4 munkák, melyekre $p_1 = 2$, $p_2 = 12$, $p_3 = 4$ és $p_4 = 5$. Akkor az egyik lehetséges ütemezés szerint az M_1 gépen először a J_1, majd a J_2 munka fut, míg az M_2 gépen először a J_4 munkát, azután a J_3 munkát hajtjuk végre. Erre az ütemezésre $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$ és $C_{\max} = 14$. Optimális ütemezést kapunk, ha például a J_2 munkát az M_1 gépen, a J_1, J_3 és J_4 munkákat pedig az M_2 gépen futtatjuk. Erre az ütemezésre $C_1 = 2$, $C_2 = 12$, $C_3 = 6, C_4 = 11$ és $C_{\max} = 12$.

Adott párhuzamos ütemezési problémára legyen $C^{\star}_{\max}$ az optimális ütemezés hossza.

a. Mutassuk meg, hogy az optimális ütemezés hossza legalább akkora, mint a legnagyobb végrehajtási idő, azaz

$$C^{\star}_{\max} \geq \max_{1 \leq k \leq n} p_k.$$

b. Mutassuk meg, hogy az optimális ütemezés hossza legalább akkora, mint a gépek átlagos terhelése, azaz

$$C^{\star}_{\max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k.$$

Tegyük fel, hogy a párhuzamos ütemezési probléma megoldására a következő mohó algoritmust alkalmazzuk: ha egy gép szabad, akkor arra ütemezzük bármelyik ütemezetlen munkát.

c. Írjunk pszeudokódot, amely megvalósítja ezt az algoritmust. Mekkora a megvalósított algoritmus futási ideje?

d. Mutassuk meg, hogy a mohó algoritmus által előállított ütemezésre

$$C^{\star}_{\max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k.$$

Mutassuk meg, hogy ez az algoritmus polinomiális futási idejű 2-közelítő algoritmus.

Megjegyzések a fejezethez

Bár évezredek óta ismertek olyan módszerek, amelyek nem szükségképpen vezetnek pontos eredményre (például a π szám közelítő meghatározása), a közelítő algoritmusok fogalma sokkal fiatalabb. Hochbaum szerint Garey, Graham és Ullman [109], valamint Johnson [167] formalizálták először a polinomiális futási idejű algoritmusok fogalmát. Az első ilyen algoritmust gyakran Grahamnek [129] tulajdonítják. Az algoritmus egy párhuzamos gépekre vonatkozó ütemezési feladatot old meg (lásd 35-5. feladat).

A korai példák nyomán közelítő algoritmusok ezreit tervezték különböző feladatok széles körének a megoldására, és nagyon gazdag szakirodalom jött létre. Számos friss könyv, mint Ausiello és mások [25], Hochbaum [149] és Vazirani [305] ugyanúgy kizárólag a közelítő algoritmusokkal foglalkozik, mint Shmoys [277], valamint Klein és Young [181] összefoglalói. Számos más könyv, mint Garey és Johnson [110], valamint Papadimitriou és Steiglitz [237] jelentős terjedelemben foglalkoznak közelítő algoritmusokkal. Lawler, Lenstra, Rinnooy Kan és Shmoys [197] részletesen tárgyalja az utazóügynök feladat megoldásának közelítő algoritmusait.

Papadimitriou és Steiglitz a KÖZ-CSÚCSLEFOGÁS algoritmust F. Gavrilnak és M. Yannakakisnak tulajdonítják. A halmazlefogási feladatot alaposan elemezték: (Hochbaum [149] 16 különböző közelítő algoritmust sorol fel), a hibakorlát azonban mindegyik esetben legalább $2 - o(1)$.

A KÖZ-UÜ-KÖRÚT algoritmus Rosenkrantz, Stearns és Lewis [261] cikkében jelent meg. Christofides megjavította ezt az algoritmust és 3/2-közelítő algoritmust adott az utazóügynökfeladatra abban az esetben, amikor teljesül a háromszög-egyenlőtlenség. Arora [21] és

Mitchell [223] megmutatták, hogy ha a pontok az euklideszi síkban vannak, akkor van polinomiális idejű közelítő séma. A 35.3. tétel Sahninak és Gonzaleznek [264] köszönhető.

A lefogási feladat mohó közelítő megoldásának elemzését a Chvátal által [61] publikált általánosabb eredmény bizonyítása alapján végeztük; az itt bemutatott fő eredmény Johnsonnak [167] és Lovásznak [206] köszönhető.

A KÖZELÍTŐ-RÉSZLETÖSSZEG algoritmus és elemzése a hátizsák és a részhalmaz-összeg feladat közelítő algoritmusainak Ibarra és Kim [164] által publikált leírására és elemzésére támaszkodik.

A MAX-3-CNF kielégíthetőség közvetve Johnson [167] munkája. A súlyozott csúcslefogási algoritmust Hochbaum [148] javasolta. A 35.4. alfejezet csak éppen érinti a véletlenítésnek és a lineáris programozásnak a közelítő algoritmusokkal kapcsolatos nagy hatóerejét. Ennek a két módszernek az összekapcsolása eredményezi a ***véletlenített kerekítést,*** ahol a feladatot először egészértékű lineáris programozási feladatként fogalmazzuk meg. Ezután megoldjuk a lineáris programozási relaxációt, és a megoldásban lévő változókat valószínűségekként értelmezzük. Ezeket a valószínűségeket azután felhasználjuk az eredeti feladat megoldásához. Ezt a módszert először Raghavan és Thompson [255] ismertették, később számos más alkalmazása is előfordult. (Lásd Motwani, Naor és Raghavan áttekintő cikkét [227].) Számos más, említésre méltó friss gondolat a közelítő algoritmusokkal kapcsolatban tartalmazza a primál-duál módszert (lásd Goemans és Williamson áttekintő cikkét [115]), a ritka vágások alkalmazását az oszd-meg-és-uralkodj algoritmusokban [199] és a szemidefinit programozás alkalmazását [116].

Amint a 34. fejezethez fűzött megjegyzésekben említettük, a valószínűségi módszerekkel ellenőrizhető bizonyítások számos feladat közelítő megoldhatóságával kapcsolatban eredményeztek alsó korlátokat – köztük több, ebben a fejezetben tárgyalt feladatra is. Az itt említett hivatkozásokhoz kiegészítésképpen Arora és Lund fejezete [22] jó leírást tartalmaz a valószínűségi módszerekkel ellenőrizhető bizonyítások és különböző feladatok közelítő megoldása közötti kapcsolatról.

VIII. BEVEZETÉS A MATEMATIKÁBA

Bevezetés

Az algoritmusok elemzésekor gyakran van szükség matematikai eszközök használatára. Ezek némelyike egyszerű algebrai tényeken alapul, mások azonban újabb ismereteket is megkövetelnek. Az első fejezetben az aszimptotikus jelölések használatával, rekurziók kezelésével foglalkoztunk. Ebben a függelékben tömören összefoglaljuk az algoritmusok vizsgálatához szükséges egyéb fogalmakat és módszereket. Ahogy azt az első fejezet bevezetőjében is megjegyeztük, lehetséges, hogy az Olvasó ennek a függeléknek az anyagát már a könyvünk elolvasása előtt is többnyire ismerte (jóllehet bizonyos specifikus jelöléseket esetenként más könyvektől eltérően használunk). Tekinthető tehát a Függelék mintegy hivatkozási anyagként is, azonban – mint a könyv egyéb fejezeteiben is – olyan feladatokat, probléma felvetéseket is szerepeltetünk itt, amelyek révén az Olvasó növelheti a jártasságát ezeken a területeken.

A Függelék A fejezetét olyan összegzésekkel kapcsolatos számításoknak és becsléseknek szenteltük, amelyek gyakran lépnek fel algoritmusok elemzése során. Ennek a fejezetnek számos formulája szerepel minden kalkulus könyvben, de itt mindezt kényelmesen megtalálja az olvasó egy helyen.

A B fejezet a halmazokra, relációkra, függvényekre, gráfokra és fákra vonatkozó alapfogalmakat és jelöléseket, illetve az ezeket érintő alapvető tulajdonságokat tartalmazza.

A Függelék C fejezete olyan elemi kombinatorikus számítási elvekkel kezdődik, mint a permutáció, kombináció és hasonlók. A fejezet további része a valószínűségszámítást megalapozó fogalmakkal és összefüggésekkel foglalkozik. Ezeket a könyvben szereplő algoritmusok nagy részének az elemzése nem igényli, így első olvasatban a fejezet utolsó alfejezetei akár ki is hagyhatók. Később viszont, ha az olvasó olyan valószínűségszámítási vizsgálatokkal találja szemben magát, amelyeket jobban szeretne megérteni, a Függelék C fejezete jól felépített hivatkozási anyagként szolgálhat mindehhez.

A. Összegzések

Ha egy algoritmus olyan iteratív vezérlő szerkezeteket tartalmaz, mint a **while** vagy **or** ciklus, akkor a futási ideje kifejezhető mint a ciklus magjának ismételt végrehajtására fordított idők összege. Így például a 2.2. alfejezetben a beszúrórendezés j-edik iterációjának végrehajtása legrosszabb esetben j-vel arányos időt vesz igénybe, ezért

$$\sum_{j=2}^{n} j$$

összeget kaptuk. Ennek az össszegnek az elemzése alapján az algoritmus futási ideje legrosszabb esetben $\Theta(n^2)$. Ez a példa is alátámasztja annak a fontosságát, hogy jól tudjunk összegeket *kezelni,* ill. becsülni.

Az A.1. alfejezetben összegzésekre vonatkozó alapvető képleteket sorolunk fel, az A.2.-ben pedig összegek becslésével foglalkozunk. Az A.1. alfejezet képletei bizonyítás nélkül szerepelnek, jóllehet a módszerek illusztrációjaképpen néhánynak az igazolását az A.2. alfejezetben megadjuk. A többinek a bizonyítása bármely bevezető analízis tankönyvben megtalálható.

A.1. Összegzések és tulajdonságaik

Véges sok $a_1, a_2, \ldots, a_n$ szám $a_1 + a_2 + \cdots + a_n$ összegét, ahol n nemnegatív egész, az alábbi szimbólummal jelöljük:

$$\sum_{k=1}^{n} a_k.$$

Ha $n = 0$, akkor az összeg legyen definíció szerint 0. Az összegben szereplő tagok sorrendje tetszés szerint megváltoztatható.

Legyen adott most számoknak egy $a_1, a_2, \ldots$ sorozata. Ekkor az $a_1 + a_2 + \cdots$ végtelen összeget a

$$\sum_{k=1}^{\infty} a_k$$

alakban írjuk és ezen az összegen a

$$\lim_{n\to\infty}\sum_{k=1}^{n} a_k$$

(véges) határértéket értjük, feltéve, hogy ez létezik. Ha az előbbi limesz nem létezik, akkor a szóban forgó végtelen sor ***divergens,*** különben ***konvergens.*** Egy konvergens végtelen sor összege általában már nem invariáns a tagok sorrendjére nézve. Azt mondjuk, hogy a $\sum a_k$ végtelen sor ***abszolút konvergens,*** ha létezik a $\sum_{k=1}^{\infty} |a_k|$ (véges) határérték. Ebben az esetben a végtelen sor konvergens is, és sem ez a tény, sem pedig a sor összege nem változik a tagok sorrendjének a megváltoztatásával.

Linearitás

Bármely $a_1, a_2, \ldots, a_n$ és $b_1, b_2, \ldots, b_n$ (véges) sorozat esetén tetszőleges c valós számra igaz a következő egyenlőség:

$$\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

Ezzel a tulajdonsággal a konvergens végtelen sorok is rendelkeznek.

A linearitás alkalmazható aszimptotikus összegzések esetén is. Így például

$$\sum_{k=1}^{n}\Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

Ebben az egyenlőségben a Θ szimbólum a bal oldalon a k-ra, a jobb oldalon az n-re vonatkozik. Hasonló eljárások konvergens végtelen sorokkal kapcsolatban is alkalmazhatók.

Számtani sorok

A

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

összegek egy ún. ***számtani sor*** részletösszegei. Jól ismert, hogy

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) \tag{A.1}$$

$$= \Theta(n^2). \tag{A.2}$$

Négyzetes és köbös összegek

Négyzetszámok, ill. köbszámok összegeire az alábbi képletek igazak:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \tag{A.3}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4} . \tag{A.4}$$

Geometriai sor

Ha az x valós szám nem 1, akkor a

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

összegek egy ún. ***mértani*** (vagy ***geometriai***) sor részletösszegei, amelyeket a következ őképpen kell kiszámítani:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} . \tag{A.5}$$

A szóban forgó mértani sor $|x| < 1$ esetén konvergens és az összege

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} . \tag{A.6}$$

Harmonikus sor

Legyen n egy pozitív egész szám. Az n-edik ***harmonikus számot*** a következ őképpen definiáljuk:

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} \\ &= \ln n + O(1). \end{aligned} \tag{A.7}$$

(Ezt a felső becslést az A.2. alfejezetben bebizonyítjuk.)

Sorok integrálása és differenciálása

A fenti formulákból integrálással és differenciálással újabbakat nyerhetünk. Például az (A.6) mértani sor mindkét oldalát differenciálva és az eredményt x-szel szorozva azt kapjuk, hogy tetszőleges $|x| < 1$ mellett

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2} . \tag{A.8}$$

Teleszkopikus összegek

Ha adottak az $a_0, a_1, \ldots, a_n$ számok, akkor

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0. \tag{A.9}$$

Valóban, a fenti összegben minden $a_1, a_2, \ldots, a_{n-1}$ tag pontosan egyszer fordul elő összeadandóként is és kivonandóként is. Az ilyen összegeket ***teleszkopikus összegeknek*** nevezzük. Hasonlóan kapjuk, hogy

$$\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n.$$

A teleszkopikus összegzés alkalmazásaként számítsuk ki az alábbi összeget:

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} .$$

Mivel

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1} ,$$

ezért

$$\begin{aligned} \sum_{k=1}^{n-1} \frac{1}{k(k+1)} &= \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1} \right) \\ &= 1 - \frac{1}{n} . \end{aligned}$$

Szorzatok

Az $a_1 a_2 \cdots a_n$ véges szorzat jelölésére a

$$\prod_{k=1}^{n} a_k$$

szimbólumot használjuk. Ha $n = 0$, akkor a szorzat értéke definíció szerint legyen 1. Ha az a_k számok valamennyien pozitívok, akkor a fenti szorzat-formulát a logaritmusra vonatkozó azonosságok alapján összeg-formulává transzformálhatjuk:

$$\lg\left(\prod_{k=1}^{n} a_k \right) = \sum_{k=1}^{n} \lg a_k.$$

Gyakorlatok

A.1-1. Mivel egyenlő a $\sum_{k=1}^{n}(2k-1)$ összeg?

A.1-2.⋆ A harmonikus sor felhasználásával bizonyítsuk be, hogy igaz az alábbi egyenl őség: $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$.

A.1-3. Lássuk be, hogy ha $0 < |x| < 1$, akkor $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$.

A.1-4.⋆ Mutassuk meg, hogy $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

A.1-5.⋆ Számítsuk ki a $\sum_{k=1}^{\infty}(2k+1)x^{2k}$ összeget.

A.1-6. A linearitás alkalmazásával igazoljuk, hogy $\sum_{k=1}^{n} O(f_k(n)) = O(\sum_{k=1}^{n} f_k(n))$.

A.1-7. Határozzuk meg a $\prod_{k=1}^{n} 2 \cdot 4^k$ szorzatot.

A.1-8.⋆ Mennyi a $\prod_{k=2}^{n}(1 - 1/k^2)$ szorzat értéke?

A.2. Összegek nagyságrendi becslése

Az algoritmusok futási idejére vonatkozó összegzések eredményének a becslésére számos eljárás ismeretes. Az alábbiakban ezek közül néhány gyakran használatos módszert tárgyalunk.

Teljes indukció

Összegek kiszámítását illetően az egyik legfontosabb módszer a teljes indukció. Illusztratív példaként mutassuk meg, hogy a $\sum_{k=1}^{n} k$ összeg értéke $n(n+1)/2$. Ez az egyenlőség $n = 1$ esetén triviális. Tételezzük fel, hogy valamilyen n-re a szóban forgó egyenlőség igaz, és lássuk be, hogy $(n+1)$-re is az:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1)\\ &= \frac{1}{2}n(n+1) + (n+1)\\ &= \frac{1}{2}(n+1)(n+2).\end{aligned}$$

A teljes indukció alkalmazásához nem kell feltétlenül ismernünk a fellépő (rész)összegek pontos értékét. A módszert az összegek becslésére is használhatjuk, amint azt a következő példa mutatja. Belátjuk, hogy a $\sum_{k=0}^{n} 3^k$ (mértani) összeg nagyságrendje $O(3^n)$, azaz egy alkalmas (n-től független) c konstanssal $\sum_{k=0}^{n} 3^k \le c3^n$. Ha $n = 0$, akkor minden $c \ge 1$ konstansra igaz a $\sum_{k=0}^{0} 3^k = 1 \le c\cdot 1$ becslés. Tegyük fel, hogy a kívánt egyenlőtlenség valamilyen n mellett teljesül, és lássuk be azt n helyett $(n+1)$-re:

$$\begin{aligned}\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1}\\ &\le c3^n + 3^{n+1}\\ &= \left(\frac{1}{3} + \frac{1}{c}\right)c3^{n+1}\\ &\le c3^{n+1},\end{aligned}$$

hacsak $(1/3 + 1/c) \le 1$, vagy ami ezzel ekvivalens, $c \ge 3/2$. Tehát $\sum_{k=0}^{n} 3^k = O(3^n)$, amint azt állítottuk.

Az előbbihez hasonló becslésekben különös gonddal kell ügyelni az aszimptotikus jelölésmód használatára. Példaként tekintsük a hamis $\sum_{k=1}^{n} k = O(n)$ állítás „bizonyítását” jelentő alábbi hibás gondolatmenetet: ha $n = 1$, akkor a $\sum_{k=1}^{1} k = O(1)$ egyenlőség triviálisan fennáll. Feltéve mindezt valamilyen n-re, „lássuk be” n helyett $(n+1)$-re:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1)\\ &= O(n) + (n+1) \qquad \Longleftarrow \textit{hiba!!}\\ &= O(n+1).\end{aligned}$$

A gondolatmenetben az a hiba, hogy a „nagy ordó” mögött meghúzódó „konstans” n növekedtével nő, és így nem állandó. Azt nem mutattuk meg, hogy ugyanaz a konstans érvényes minden n-re.

A tagok becslése

Gyakran kaphatunk megfelelő becslést a szóban forgó összegre az egyes tagok becslése révén. Sőt, néha az is elegendő, ha a legnagyobb tagra vonatkozó becslést alkalmazzuk a többi tagra is. Például az (A.1) számtani sor részletösszegekre egy gyors felső becslés a következőképpen adódik:

$$\begin{aligned}\sum_{k=1}^{n} k &\leq \sum_{k=1}^{n} n \\ &= n^2.\end{aligned}$$

Általában, ha a $\sum_{k=1}^{n} a_k$ összegben szereplő tagokra $a_{\max} = \max_{1\leq k\leq n} a_k$, akkor

$$\sum_{k=1}^{n} a_k \leq n a_{\max}.$$

A legnagyobb taggal való felső becslés persze sok esetben nem elégséges. Ennél erősebb becslés kapható például akkor, ha az illető összeg tagjait egy mértani sor tagjaival tudjuk majorálni. Nevezetesen, tegyük fel, hogy a $\sum_{k=0}^{n} a_k$ összegben $a_k > 0$ $(k = 0, 1, \ldots)$ és valamilyen $0 < r < 1$ számmal $a_{k+1}/a_k \leq r$ teljesül minden $k = 0, 1, \ldots$ indexre. Ekkor az előbbi összeget egy konvergens mértani sor segítségével becsülhetjük, ui. $a_k \leq a_0 r^k$ $(k = 0, 1, \ldots)$, azaz

$$\begin{aligned}\sum_{k=0}^{n} a_k &\leq \sum_{k=0}^{\infty} a_0 r^k \\ &= a_0 \sum_{k=0}^{\infty} r^k \\ &= a_0 \frac{1}{1-r}\,.\end{aligned}$$

Ezt a gondolatmenetet alkalmazhatjuk a $\sum_{k=1}^{\infty}(k/3^k)$ végtelen sorösszeg felső becslésére is. Annak érdekében, hogy az összegzés $k = 0$-val kezdődjön, írjuk át a szóban forgó összeget $\sum_{k=0}^{\infty}((k+1)3^{k+1})$ alakba. Az első tag $1/3$, az egymást követő tagok hányadosa pedig

$$\begin{aligned}\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} &= \frac{1}{3}\frac{k+2}{k+1} \\ &\leq \frac{2}{3}\end{aligned}$$

minden $k = 0, 1, \ldots$ természetes számra. Így

$$\begin{aligned}\sum_{k=1}^{\infty}\frac{k}{3^k} &= \sum_{k=0}^{\infty}\frac{k+1}{3^{k+1}}\\ &= \frac{1}{3}\frac{1}{1-2/3}\\ &= 1.\end{aligned}$$

A fenti módszerrel kapcsolatban hangsúlyozni kell, hogy nem elegendő az egymást követő tagok hányadosának az 1-nél kisebb voltát igazolni. Ebből még nem következtethetnénk arra, hogy a szóban forgó összeg egy mértani sor segítségével becsülhető. Álljon itt figyelmeztető példa gyanánt a harmonikus sor, amely

$$\begin{aligned}\sum_{k=1}^{\infty}\frac{1}{k} &= \lim_{n\to\infty}\sum_{k=1}^{n}\frac{1}{k}\\ &= \lim_{n\to\infty}\Theta(\lg n)\\ &= \infty\end{aligned}$$

miatt divergens. Ugyanakkor a $(k+1)$-edik és a k-adik tag hányadosa ebben a sorban minden $k = 1, 2, \ldots$ esetén 1-nél kisebb, hiszen $k/(k+1) < 1$. A harmonikus sor tagjait azonban nem tudjuk egy konvergens mértani sor tagjaival felülről becsülni. Egy ilyen becsléshez olyan (k-tól független) $0 < r < 1$ konstansra lenne szükség, amelyet egyetlen k esetén sem lép túl a $(k+1)$-edik és a k-adik tag hányadosa. A harmonikus sor esetén ilyen r nem adható meg, ui. az említett hányadosok $k \to \infty$ esetén 1-hez tartanak.

Az összeg felbontása

Bonyolultabb összegek becslésére alkalmazhatjuk az ún. felbontási technikát. Ennek során a szóban forgó összeget (az indextartomány részekre osztásával) két vagy több összegre bontjuk, és ezeket összegezzük. Ha mindegyik részösszegre ismerünk egy alkalmas alsó (felső) becslést, akkor ezeket összeadva alulról (felülről) becsülhetjük az eredeti összeget. Becsüljük meg például ezzel a módszerrel a $\sum_{k=1}^{n} k$ összeget alulról. (Emlékeztetünk arra, hogy korábban ezt felülről becsültük n^2-tel.) Ha minden tagra a triviális módon adódó $k \geq 1 \quad (k = 1, 2, \ldots)$ becslést alkalmaznánk, akkor az előbb említett n^2 felső becsléstől meglehetősen távoli $\sum_{k=1}^{n} k \geq n$ alsó becslést kapnánk.

A felbontási módszerrel azonban ennél lényegesen jobb alsó becslés adható. Tegyük fel az egyszerűség kedvéért, hogy az n index éppen páros. Ekkor

$$\begin{aligned}\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k\\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n}\left(\frac{n}{2}\right)\\ &= \left(\frac{n}{2}\right)^2\\ &= \Omega(n^2),\end{aligned}$$

ami aszimptotikusan egy erős becslés, hiszen $\sum_{k=1}^{n} k = O(n^2)$.

Az algoritmusok elemzése során fellépő összegzések vizsgálatakor gyakran alkalmazhatjuk ezt a felbontási módszert. Nevezetesen, ha a $\sum_{k=0}^{n} a_k$ összegben minden a_k tag nemnegatív és független n-től, akkor bármely $k_0 > 0$ index mellett

$$\begin{aligned}\sum_{k=0}^{n} a_k &= \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k \\ &= \Theta(1) + \sum_{k=k_0}^{n} a_k\end{aligned}$$

írható, mivel az összeg kezdeti tagjai konstansok és konstans számú van belőlük. A $\sum_{k=k_0}^{n} a_k$ összeg becslésére aztán bármilyen célravezető módszer alkalmazható. Példaként adjunk felső becslést a felbontási technika révén a

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k}$$

összegre. Ehhez először is vegyük észre, hogy (a harmadiktól kezdve) az egymás utáni tagok hányadosa legfeljebb 8/9:

$$\begin{aligned}\frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+1)^2}{2k^2} \\ &\le \frac{8}{9}\,.\end{aligned}$$

Következésképpen az összeg felbontásával

$$\begin{aligned}\sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ &\le \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k \\ &= O(1)\end{aligned}$$

adódik, ahol az első összegnek konstans számú tagja van, a második összeget pedig egy konvergens mértani sor segítségével becsültük.

A felbontási módszer jóval bonyolultabb szituációban is alkalmazható aszimptotikus becslések igazolására. A harmonikus sor (A.7) részletösszegeire például ezzel az eljárással juthatunk el az $O(\log n)$ nagyságrendi becsléshez. Ehhez osszuk fel az (A.7)-ben szereplő $1 \to n$ indextartományt $\lfloor \log n \rfloor$ részre, és becsüljük 1-gyel az egyes részösszegeket. Minden részösszeg $1/2^i$-vel kezdődik és $1/(2^{i+1} - 1)$-gyel végződik, ezért

$$\begin{aligned}\sum_{k=1}^{n} \frac{1}{k} &\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\ &\le \sum_{i=0}^{\lfloor \lg 2n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^j}\end{aligned}$$

$$\begin{aligned} &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\ &\leq \lg n + 1. \end{aligned} \tag{A.10}$$

Az összeg becslése integrállal

Ha a kiszámítandó összeg egy alkalmas, monoton növő, nemnegatív értékeket felvevő f függvénnyel $\sum_{k=m}^{n} f(k)$ alakban írható, akkor a kérdéses összeget az f integráljával a következőképpen becsülhetjük:

$$\begin{aligned} \int_{m-1}^{n} f(x)\,dx &\leq \sum_{k=m}^{n} f(k) \\ &\leq \int_{m}^{n+1} f(x)\,dx. \end{aligned} \tag{A.11}$$

Az (A.11) egyenlőtlenségeket az A.1. ábrán szemléltetjük. A szóban forgó összeg az ábrán látható téglalapok területösszege, az aktuális integrál pedig a függvény grafikonja alatti, sötéttel jelzett síkrész területe. Ha az f függvény monoton csökkenő, akkor az

$$\begin{aligned} \int_{m}^{n+1} f(x)\,dx &\leq \sum_{k=m}^{n} f(k) \\ &\leq \int_{m-1}^{n} f(x)\,dx \end{aligned} \tag{A.12}$$

becslések az előzőhöz hasonló módon adódnak.

Az (A.12) integrál-közelítés révén (nagyságrendileg) pontos becslést kaphatunk a harmonikus sor részletösszegeire. Valóban, ami az alsó becslést illeti, a következőt mondhatjuk:

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &\geq \int_{1}^{n+1} \frac{dx}{x} \\ &= \ln(n+1). \end{aligned} \tag{A.13}$$

A felső becsléshez az alábbi módon jutunk:

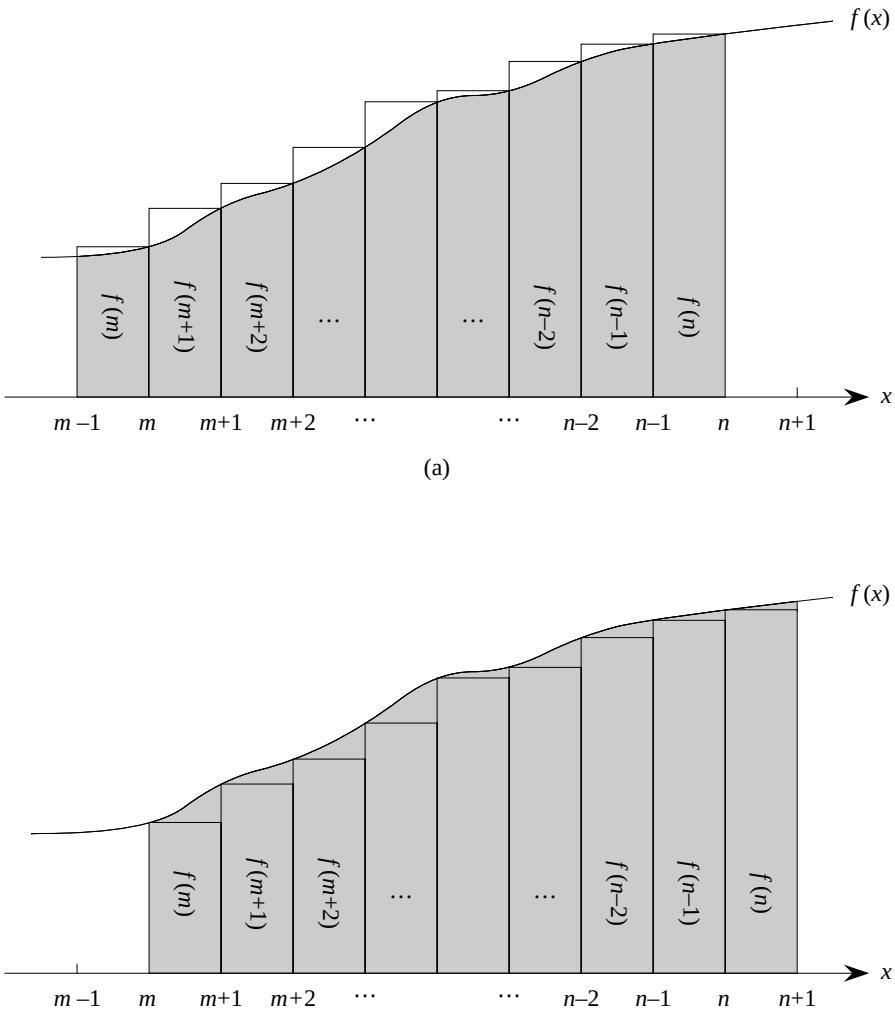
$$\begin{aligned} \sum_{k=2}^{n} \frac{1}{k} &\leq \int_{1}^{n} \frac{dx}{x} \\ &= \ln n, \end{aligned}$$

amiből már következik, hogy

$$\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1. \tag{A.14}$$

Gyakorlatok

A.2-1. Mutassuk meg, hogy $\sum_{k=1}^{n} 1/k^2$ becsülhető felülről egy n-től független konstanssal.
A.2-2. Keressünk aszimptotikus alsó becslést a $\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$ összegre.



A.1. ábra. A $\sum_{k=m}^{n} f(k)$ összeg közelítése integrálokkal. A téglalapok területösszege adja a szóban forgó összeget, a függvény grafikonja alatti terület pedig az integrált. Az említett területek összehasonlításából kapjuk az ábra **(a)** részében az $\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k)$ becslést, míg a **(b)**-ben a téglalapok egységnyi jobbra tolásával a $\sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx$ egyenlőtlenséget.

A.2-3. A felbontási technika alkalmazásával bizonyítsuk be, hogy $\sum_{k=1}^{n} 1/k = \Omega(\lg n)$.
A.2-4. Egy alkalmas integrál alakjában adjunk közelítést $\sum_{k=1}^{n} k^3$-re.
A.2-5. Alkalmazható-e közvetlenül az (A.12) integrál-közelítés a harmonikus sor $\sum_{k=1}^{n} 1/k$ részletösszegeinek a felső becslésére?

Feladatok

A-1. Összegek nagyságrendi becslése
Adjunk aszimptotikusan pontos korlátokat az alábbi összegekre. Feltesszük, hogy r és s nemnegatív konstansok.

a. $\sum_{k=1}^{n} k^r$.

b. $\sum_{k=1}^{n} \lg^s k$.

c. $\sum_{k=1}^{n} k^r \lg^s k$.

Megjegyzések a fejezethez

Az ebben a fejezetben tárgyalt anyagot illetően Knuth[182] könyvét ajánljuk az Olvasó figyelmébe. A sorokkal, összegzésekkel kapcsolatos alapvető ismeretek bármely színvonalas analíziskönyvben megtalálhatók, így pl. Apostol[18] vagy Thomas és Finney[296] művében.

B. Halmazok és más alapfogalmak

A könyvünk számos fejezetében találkozhatunk a diszkrét matematika elemeivel. A B fejezet halmazokkal, relációkkal, függvényekkel, gráfokkal és fákkal kapcsolatos definíciókat, jelöléseket és elemi tulajdonságokat tartalmaz. Ha az Olvasó már járatos ezen a területen, akkor elegendő, ha csak átfutja ezt a fejezetet.

B.1. Halmazok

A ***halmaz*** különböző objektumok együttese, amelyeket az illető halmaz ***elemeinek*** nevezünk. Ha egy x objektum eleme egy S halmaznak, akkor az $x \in S$ szimbólumot használjuk (és úgy olvassuk, hogy x eleme S-nek vagy röviden: x S-ben van, esetleg: x S-hez tartozik). Ha x nem eleme S-nek, akkor ezt az $x \notin S$ jelöléssel juttatjuk kifejezésre. Egy halmazt megadhatunk az elemeinek kapcsos zárójelek közötti felsorolásával. Például definiáljuk az S halmazt úgy, hogy az pontosan az 1, 2, 3 számokat tartalmazza. Ekkor azt írjuk, hogy $S = \{1, 2, 3\}$. Mivel 2 eleme S-nek, ezért $2 \in S$. Hasonlóan kapjuk, hogy $4 \notin S$, ui. 4 nem eleme az S halmaznak. Egy halmaz megadásakor annak minden elemét csak egyszer soroljuk fel és ennek során semmiféle sorrendiség nem játszik szerepet.[1] Két halmazt ***egyenlőnek*** nevezünk (írásban: $A = B$), ha ugyanazok az elemeik. Például $\{1, 2, 3\} = \{3, 2, 1\}$.

A gyakran előforduló halmazokra az alábbi speciális jelöléseket használjuk:

- $\emptyset$ jelöli az ***üres halmazt,*** tehát azt a halmazt, amelynek egyetlen eleme sincs.
- $\mathbf{Z}$ az ***egész számok*** halmaza, azaz a $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ halmaz.
- $\mathbf{R}$-rel jelöljük a ***valós számok*** halmazát.
- $\mathbf{N}$ fogja jelölni a ***természetes számok*** halmazát, amely a $\{0, 1, 2, \ldots\}$ halmaz.[2]

Ha az A halmaz minden eleme a B halmaznak is eleme, azaz $x \in A$ esetén $x \in B$, akkor azt mondjuk, hogy A ***részhalmaza*** B-nek, és azt írjuk, hogy $A \subseteq B$. Az A halmaz ***valódi részhalmaza*** B-nek (jelölésben $A \subset B$), ha $A \subseteq B$, de $A \neq B$. (Egyes szerzők a $\subset$ szimbólumot nem csupán a valódi részhalmaz, hanem a részhalmaz jelölésére is használják.)

---

[1] Az olyan *halmazokat*, amelyek ugyanazt az objektumot egynél többször is tartalmazhatják elemként, ***multihalmazoknak*** nevezik.

[2] Vannak, akik a természetes számokat 1-gyel kezdik a 0 helyett. A mai gyakorlatnak megfelelően mi 0-val kezdünk.

Minden A halmazra $A \subseteq A$. Ha A, B halmazok, akkor $A = B$ akkor és csak akkor teljesül, ha $A \subseteq B$ és $B \subseteq A$. Tetszőleges A, B, C halmazokra igaz a következő állítás: ha $A \subseteq B$ és $B \subseteq C$, akkor $A \subseteq C$. Bármely A halmaz esetén $\emptyset \subseteq A$.

Ha adott egy halmaz, akkor ebből kiindulva újabb halmazokat definiálhatunk. Mondjuk egy A halmaz esetén definiálhatjuk a $B \subseteq A$ halmazt valamely, az A elemeire értelmezett tulajdonság alapján. Például a páros egész számok halmazát az alábbi módon adhatjuk meg: $\{x : x \in \mathbf{Z} \text{ és } x/2 \text{ egész szám}\}$. A kettőspontot ebben a jelölésben a következőképpen olvassuk: „olyan, hogy." (A kettőspont helyett gyakran egy függőleges vonalat írnak.)

Adott A, B halmazok esetén a ***halmazműveletek*** révén is értelmezhetünk újabb halmazokat:

- Az A és a B ***metszete*** (vagy ***közös része***) az
$$A \cap B = \{x : x \in A \text{ és } x \in B\}$$
halmaz.
- Az A és a B ***uniója*** (vagy ***egyesítése***) az
$$A \cup B = \{x : x \in A \text{ vagy } x \in B\}$$
halmaz.
- az A és a B halmaz ***különbsége*** az
$$A - B = \{x : x \in A \text{ és } x \notin B\}$$
halmaz.

A halmazműveletekkel kapcsolatban a következő állítások teljesülnek.

Null-elem tulajdonság:

$$\begin{aligned} A \cap \emptyset &= \emptyset, \\ A \cup \emptyset &= A. \end{aligned}$$

Idempotencia:

$$\begin{aligned} A \cap A &= A, \\ A \cup A &= A. \end{aligned}$$

Kommutativitás:

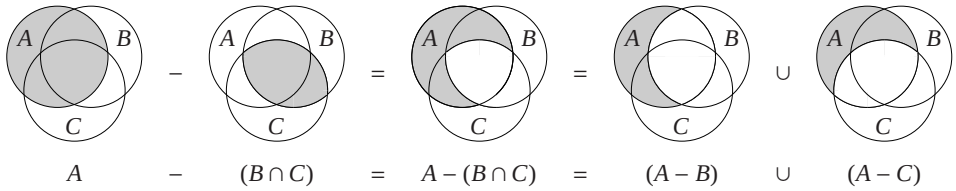
$$\begin{aligned} A \cap B &= B \cap A, \\ A \cup B &= B \cup A. \end{aligned}$$

Asszociativitás:

$$\begin{aligned} A \cap (B \cap C) &= (A \cap B) \cap C, \\ A \cup (B \cup C) &= (A \cup B) \cup C. \end{aligned}$$

Disztributivitás:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C), \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C). \end{aligned} \tag{B.1}$$



B.1. ábra. A (B.2) DeMorgan-azonosságot illusztráló Venn-diagram. Az A, B, C halmazok mindegyikét körökkel szemléltettük.

Elnyelés:

$$\begin{aligned} A \cap (A \cup B) &= A, \\ A \cup (A \cap B) &= A. \end{aligned}$$

DeMorgan-azonosságok:

$$\begin{aligned} A - (B \cap C) &= (A - B) \cup (A - C), \\ A - (B \cup C) &= (A - B) \cap (A - C). \end{aligned} \tag{B.2}$$

Az első DeMorgan-azonosságot a B.1. ábrán a ***Venn-diagramok*** segítségével szemléltetjük, amikor is a halmazokat grafikusan síkbeli tartományok jelenítik meg.

A vizsgált halmazok gyakran egy ***alaphalmaznak*** nevezett U halmaz részhalmazai. Például, ha a szóban forgó halmazok elemei egész számok, akkor a $\mathbf{Z}$ halmaz egy ilyen alaphalmaz. Az U alaphalmaz segítségével a következőképpen definiáljuk egy A halmaz (U-ra vonatkozó) ***komplementerét:*** $\overline{A} = U - A$. Tetszőleges $A \subseteq U$ halmazra a következők igazak:

$$\begin{aligned} \overline{\overline{A}} &= A, \\ A \cap \overline{A} &= \emptyset, \\ A \cup \overline{A} &= U. \end{aligned}$$

A (B.2) DeMorgan-azonosságok kifejezhetők a komplementer halmaz segítségével is. Nevezetesen, bármely $B \subseteq U$ és $C \subseteq U$ halmaz esetén

$$\begin{aligned} \overline{B \cap C} &= \overline{B} \cup \overline{C}, \\ \overline{B \cup C} &= \overline{B} \cap \overline{C}. \end{aligned}$$

Az A, B halmazokat ***diszjunktnak*** nevezzük, ha nincs közös elemük, azaz, ha $A \cap B = \emptyset$. Nem üres halmazoknak egy $\mathcal{S} = \{S_i\}$ együttese az S halmaz egy ***felosztását*** alkotja, ha

- az S_i halmazok ***páronként diszjunktak,*** tehát $S_i, S_j \in \mathcal{S}$, $i \neq j$ esetén $S_i \cap S_j = \emptyset$ és
- az S_i halmazok egyesítése megegyezik S-sel, azaz $S = \bigcup_{S_i \in \mathcal{S}} S_i$.

Más szóval tehát $\mathcal{S}$ az S egy felosztása, ha az S minden eleme pontosan egy $S_i \in \mathcal{S}$ halmaznak az eleme és tetszőleges i-re $\emptyset \neq S_i \subseteq S$.

Egy S halmaz elemeinek a számát az S ***számosságának*** nevezzük, és az $|S|$ szimbólummal jelöljük. Két halmaz számossága pontosan akkor egyezik meg, ha az elemeik kölcsönösen egyértelműen megfeleltethetők egymásnak. Az üres halmaz számossága nulla: $|\emptyset| = 0$.

Ha egy halmaz számossága természetes szám, akkor az illető halmazt ***végesnek,*** egyébként pedig ***végtelennek*** nevezzük. Azt mondjuk, hogy a szóban forgó végtelen halmaz ***megszámlálhatóan végtelen,*** ha kölcsönösen egyértelműen megfeleltethető a természetes számok **N** halmazának; különben a halmaz ***nem megszámlálható.*** Az egész számok **Z** halmaza megszámlálható, ugyanakkor az **R** nem megszámlálható.

Ha A és B véges halmaz, akkor

$$|A \cup B| = |A| + |B| - |A \cap B|, \tag{B.3}$$

amiből rögtön következik, hogy

$$|A \cup B| \leq |A| + |B|.$$

Ha A és B diszjunkt, akkor $|A \cap B| = 0$ és így $|A \cup B| = |A| + |B|$. Nyilvánvaló, hogy $A \subseteq B$ esetén $|A| \leq |B|$.

Egy n elemű véges halmazt gyakran ***n-halmaznak*** is mondanak. Valamely halmaz k elemet tartalmazó részhalmazait ***k-részhalmazoknak*** is nevezik.

Legyen S halmaz. Azt a halmazt, amely az S összes részhalmazából áll, az S ***hatványhalmazának*** nevezzük, és 2^S-sel jelöljük. Például $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Egy véges S halmaz hatványhalmazának a számossága $2^{|S|}$.

A későbbi vizsgálatainkban szerepet kapnak olyan halmazstruktúrák is, amelyekben az elemek rendezett párok. Az a, b elemek által meghatározott ***rendezett párt*** (a, b)-vel jelöljük, és a következő halmazként definiáljuk: $(a, b) = \{a, \{a, b\}\}$. Az a elem az (a, b) rendezett pár első, a b a második komponense. Nyilvánvaló, hogy az (a, b) rendezett pár általában *nem* ugyanaz, mint a (b, a) rendezett pár.

Ha A és B egy-egy halmaz, akkor az $A \times B$ ***Descartes-szorzatuk*** az összes olyan rendezett párból álló halmaz, amelynek az első komponense A-ban, a második B-ben van. Tehát

$$A \times B = \{(a, b) : a \in A \text{ és } b \in B\}.$$

Például $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Ha A és B véges halmaz, akkor a Descartes-szorzatuk számossága

$$|A \times B| = |A| \cdot |B|. \tag{B.4}$$

Az n darab $A_1, A_2, \ldots, A_n$ halmaz Descartes-szorzata a ***rendezett n-esek*** következő halmaza:

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i, i = 1, 2, \ldots, n\};$$

a számossága

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n| \,,$$

ha mindegyik halmaz véges. Egy A halmaz önmagával vett n-szeres Descartes-szorzata az

$$A^n = A \times A \times \cdots \times A$$

halmaz, amelynek a számossága (véges A-ra) $|A^n| = |A|^n$. A rendezett n-esek egy n hosszúságú véges sorozatként is felfoghatók (lásd a B.3. alfejezetet).

Gyakorlatok

B.1-1. Szemléltessük Venn-diagram segítségével a (B.1) disztributivitást.

B.1-2. Bizonyítsuk be a DeMorgan-azonosságok alábbi általánosítását tetsz őlegesen választott véges sok $A_1, A_2, \ldots, A_n$ halmazra:

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A}_1 \cup \overline{A}_2 \cup \cdots \cup \overline{A}_n,$$

$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A}_1 \cap \overline{A}_2 \cap \cdots \cap \overline{A}_n.$$

B.1-3.* Mutassuk meg, hogy igaz a (B.3) egyenl őség következő általánosítása (az ún. ***logikai szita-formula***):

$$\begin{aligned} |A_1 \cup A_2 \cup \cdots \cup A_n| \quad = \quad & |A_1| + |A_2| + \cdots + |A_n| \\ & - |A_1 \cap A_2| - |A_1 \cap A_3| - \cdots \qquad \text{(minden pár)} \\ & + |A_1 \cap A_2 \cap A_3| + \cdots \qquad \text{(minden hármas)} \\ & \qquad \vdots \\ & + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n|. \end{aligned}$$

B.1-4. Igazoljuk, hogy a páratlan természetes számok halmaza megszámlálható.

B.1-5. Bizonyítsuk be, hogy bármely véges S halmaz esetén a 2^S hatványhalmaznak $2^{|S|}$ eleme van. (Tehát S-nek van $2^{|S|}$ darab különböző részhalmaza.)

B.1-6. A rendezett pár halmazelméleti definíciójából kiindulva adjuk meg a rendezett n-es rekurzív értelmezését.

B.2. Relációk

Az A, B halmazok $A \times B$ Descartes-szorzatának valamely R részhalmazát az A, B halmazokon értelmezett ***(bináris) relációnak*** nevezzük. Ha $(a, b) \in R$, akkor azt is írjuk, hogy aRb. Az A halmazon értelmezett (bináris) reláción az $A \times A$ halmaz valamely R részhalmazát értjük. Például a természetes számokon értelmezett „kisebb" reláció a következő halmaz: $\{(a, b) : a, b \in \mathbf{N} \text{ és } a < b\}$. Az $A_1 \times A_2 \times \cdots \times A_n$ halmaz részhalmazait az $A_1, A_2, \ldots, A_n$ halmazokon értelmezett (n-szeres) relációnak nevezzük.

Az $R \subseteq A \times A$ bináris reláció ***reflexív,*** ha minden $a \in A$ esetén

$$aRa.$$

Például az „=" és a „≤" reflexív reláció **N**-en, de a „<" nem az. Az R reláció ***szimmetrikus,*** ha minden $a, b \in A$ esetén

$$aRb\text{-ből következik } bRa.$$

Így például az „=" szimmetrikus, azonban a „<" és a „≤" nem. Az R reláció ***tranzitív,*** ha minden $a, b, c \in A$ esetén

$$aRb \text{ és } bRc\text{-ből következik } aRc.$$

Például a „<, ≤" és az „=" tranzitív relációk, de az $R = \{(a, b) : a, b \in \mathbf{N} \text{ és } a = b - 1\}$ reláció nem, hiszen $3R4$ és $4R5$, de $3R5$ nem igaz.

Azt mondjuk, hogy egy reláció ***ekvivalencia-reláció,*** ha reflexív, szimmetrikus és tranzitív. Például az „=” ekvivalencia-reláció a természetes számokon, viszont a „<” nem az. Ha R egy ekvivalencia-reláció az A halmazon, akkor valamely $a \in A$ elem esetén az a által generált ***ekvivalencia-osztály*** az alábbi halmaz: $[a] = \{b \in A : aRb\}$. Tehát $[a]$ az a-val ekvivalens A-beli elemek halmaza. Definiáljuk például az R relációt úgy, hogy $R = \{(a, b) : a, b \in \mathbf{N}$ és $a+b$ páros szám$\}$. Ekkor R egy ekvivalencia-reláció, ha ui. $a, b, c \in \mathbf{N}$, akkor $a+a$ páros (reflexivitás); ha $a+b$ páros, akkor $b+a$ is páros (szimmetria); ha $a+b$ is és $b+c$ is páros, akkor $a+c$ is páros (tranzitivitás). A 4 által generált ekvivalencia-osztály a $[4] = \{0, 2, 4, 6, \ldots\}$ halmaz, a 3 által meghatározott pedig az $\{1, 3, 5, 7, \ldots\}$. Az ekvivalencia-osztályokkal kapcsolatos alaptétel a következő.

B.1. tétel (az ekvivalencia-relációk és a felosztások kapcsolata). *Az A halmazon értelmezett bármely R ekvivalencia-reláció esetén az R által meghatározott ekvivalencia-osztályok az A egy felosztását alkotják. Fordítva, az A tetszőleges felosztásához megadható egy olyan A-n értelmezett ekvivalencia-reláció, amely által meghatározott ekvivalencia-osztályok halmaza megegyezik a szóban forgó felosztással.*

Bizonyítás. Az állítás első részének az igazolásához azt kell megmutatnunk, hogy az R által meghatározott ekvivalencia-osztályok olyan nem üres, páronként diszjunkt halmazok, amelyeknek az uniója A. Mivel R reflexív, ezért $a \in [a]$, következésképpen az ekvivalencia-osztályok nem üresek. Az $a \in [a]$ tartalmazásból az is világos, hogy az ekvivalencia-osztályok uniója megegyezik A-val. Azt kell már csak belátnunk, hogy az ekvivalencia-osztályok páronként diszjunkt halmazok, azaz, ha az $[a]$, $[b]$ ekvivalencia-osztályoknak van egy közös c elemük, akkor $[a] = [b]$. Valóban, ekkor aRc és bRc, ezért a szimmetria és a tranzitivitás alapján aRb is fennáll. Így bármely $x \in [a]$ elemre xRa, amiből xRb következik. Tehát $[a] \subseteq [b]$. Hasonlóan kapjuk, hogy $[b] \subseteq [a]$, azaz $[a] = [b]$.

A tétel második állításának a bizonyításához legyen $\mathcal{A} = \{A_i\}$ egy felosztása az A-nak és definiáljuk az R relációt az alábbiak szerint: $R = \{(a, b) :$ van olyan i, hogy $a \in A_i$ és $b \in A_i\}$. Azt állítjuk, hogy R egy ekvivalencia-reláció az A halmazon. A reflexivitás világos, hiszen $a \in A_i$ esetén nyilván aRa. Ha aRb, akkor a, b ugyanabban az A_i halmazban vannak, így bRa, azaz a szimmetria is teljesül. Végül tegyük fel, hogy aRb és bRc. Ekkor az itt szereplő mindhárom elem ugyanahhoz az A_i halmazhoz tartozik, amiből aRc, tehát a tranzitivitás adódik. Azt kell már csak megmutatnunk, hogy az $\mathcal{A}$ felosztásban szereplő halmazok nem mások, mint az R által meghatározott ekvivalencia-osztályok. Ehhez vegyük észre, hogy $a \in A_i$ esetén $x \in [a]$-ból $x \in A_i$ következik. Ha viszont $x \in A_i$, akkor $x \in [a]$. ■

Az A halmazon értelmezett R (bináris) relációt ***antiszimmetrikusnak*** nevezzük, ha minden $a, b \in A$ esetén

$$aRb \text{ és } bRa\text{-ból következik } a = b.$$

Például a „$\leq$” reláció a természetes számok halmazán antiszimmetrikus, mivel az $a \leq b, b \leq a$ feltétel teljesülése esetén $a = b$. Ha egy reláció reflexív, antiszimmetrikus és tranzitív, akkor ***parciális rendezésnek*** nevezzük. Azt a halmazt, amelyen egy parciális rendezés definiálva van, ***parciálisan rendezett halmaznak*** mondjuk. Például a „leszármazottja” egy parciális rendezés az emberek halmazán (ha az egyes embereket önmaguk leszármazottjának tekintjük).

Egy parciálisan rendezett A halmaznak nem feltétlenül létezik a szokásos értelemben vett *maximuma*, tehát olyan $a \in A$, amelyre minden $b \in A$ esetén bRa. Lehetnek viszont olyan a (maximális) elemek az A-ban, amelyekhez nem található olyan $a \neq b \in A$, amelyre aRb teljesülne. Vegyük például különböző méretű dobozoknak egy halmazát. Ekkor lehetnek olyan „maximális" dobozok, amelyek nem férnek bele egyetlen más dobozba sem, ugyanakkor nincs olyan doboz, amelybe az összes többi beleférne.[3]

Egy A halmazon értelmezett R parciális rendezést ***teljesnek*** vagy ***lineárisnak*** nevezünk, ha tetszőleges $a, b \in A$ esetén vagy aRb vagy bRa. Más szóval tehát, az A bármely két eleme az R szerint relációba hozható. Például a természetes számok halmazán a „$\leq$" egy teljes rendezés, de a fent említett „leszármazottja" nem az, ui. vannak olyan emberek, akik közül egyik sem leszármazottja a másiknak.

Gyakorlatok

B.2-1. Bizonyítsuk be, hogy a $\mathbf{Z}$ hatványhalmazán értelmezett „$\subseteq$" reláció egy parciális rendezés, amely azonban nem teljes.

B.2-2. Mutassuk meg, hogy tetszőleges n természetes szám esetén az „ekvivalens modulo n" reláció az egész számok halmazán egy ekvivalencia-reláció. (Azt mondjuk, hogy $a \equiv b \pmod{n}$, ha egy alkalmas q egész számmal $a - b = qn$.) Mik lesznek a szóban forgó ekvivalencia-reláció által meghatározott ekvivalencia-osztályok?

B.2-3. Adjunk példát olyan relációra, amely

a. reflexív és szimmetrikus, de nem tranzitív;

b. reflexív és tranzitív, de nem szimmetrikus;

c. tranzitív és szimmetrikus, de nem reflexív.

B.2-4. Legyen S egy véges halmaz, R pedig egy ekvivalencia-reláció $S \times S$-en. Igazoljuk, hogy ha R antiszimmetrikus is, akkor az R által meghatározott ekvivalencia-osztályok egyelemű halmazok.

B.2-5. Narcissus professzor azt sejti, hogy ha egy R reláció szimmetrikus és tranzitív, akkor reflexív is. A sejtésre a következő bizonyítást adja. A szimmetria miatt aRb-ből bRa következik. A tranzitivitás miatt tehát aRa is igaz. Helyes a professzor bizonyítása?

B.3. Függvények

Legyen adott két halmaz, A és B. Az $f \subseteq A \times B$ ***függvény*** egy olyan bináris reláció, amelyre bármely $a \in A$ esetén pontosan egy olyan $b \in B$ létezik, hogy $(a, b) \in f$. Az A halmaz az f ***értelmezési tartománya,*** a B az f ***képtartománya.*** Minderre a következő jelölést használjuk: $f : A \to B$. Ha $(a, b) \in f$, akkor azt írjuk, hogy $b = f(a)$ (ezzel is utalva arra, hogy az a megválasztásával a b-t egyértelműen meghatároztuk).

Azt mondjuk, hogy az $f : A \to B$ függvény az A elemeihez B-beli elemeket rendel, miközben az A egyetlen eleméhez sem rendel két különböző elemet B-ből. Ugyanakkor a B valamely eleme tartozhat a fenti értelemben az A két különböző eleméhez is. Például az

$$\{(a, b) : a \in \mathbf{N} \text{ és } b \equiv a \pmod{2}\}$$

[3] Ahhoz, hogy a *belefér* reláció egy parciális rendezés legyen, minden dobozt önmagába belefér*ő*nek tekintünk.

bináris reláció egy $f : \mathbf{N} \to \{0, 1\}$ függvény, ui. tetszőleges a természetes szám esetén pontosan egy olyan b létezik $\{0, 1\}$-ből, hogy $b \equiv a \pmod 2$. Ebben a példában $0 = f(0), 1 = f(1), 0 = f(2)$ és így tovább. A

$$g = \{(a, b) : a \in \mathbf{N} \text{ és } a + b \text{ páros}\}$$

reláció azonban nem függvény, mivel $(1, 3)$ és $(1, 5)$ egyaránt g-beli, azaz az $a = 1$ választással nemcsak egy b-re igaz, hogy $(a, b) \in g$.

Ha adott az $f : A \to B$ függvény, akkor $b = f(a)$ esetén az a-t ***argumentumnak*** (vagy ***független változónak***), a b-t pedig az f a-beli ***(helyettesítési) értékének*** nevezzük. Egy függvényt definiálhatunk úgy is, hogy az értelmezési tartománya minden pontjában megadjuk az értékét. Például az $f(n) = 2n \quad (n \in \mathbf{N})$ előírással az alábbi függvényt definiáltuk: $f = \{(n, 2n) : n \in \mathbf{N}\}$. Két függvény, f és g, ***egyenlősége*** azt jelenti, hogy az értelmezési tartományaik megegyeznek és ennek bármely a elemére $f(a) = g(a)$.

Egy n hosszúságú ***véges sorozaton*** olyan függvényt értünk, amelynek az értelmezési tartománya $\{0, 1, \ldots, n - 1\}$. Véges sorozatokat gyakran az értékeik felsorolásával jelölünk: $\langle f(0), f(1), \ldots, f(n - 1)\rangle$. A ***végtelen sorozat*** olyan függvény, amelynek az értelmezési tartománya a természetes számok halmaza. Például a (3.21)-ban definiált Fibonacci-sorozat egy végtelen sorozat, nevezetesen: $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Amennyiben az f függvény értelmezési tartománya egy Descartes-szorzat, akkor az f értékeinek a jelölésében gyakran elhagyjuk az argumentumot körülvevő két zárójel pár egyikét. Például, ha $f : A_1 \times A_2 \times \cdots \times A_n \to B$, akkor $f((a_1, a_2, \ldots, a_n))$ helyett azt írjuk, hogy $f(a_1, a_2, \ldots, a_n)$. Az itt szereplő minden egyes a_i-t ***argumentumnak*** nevezzük, dacára annak, hogy formálisan az $(a_1, a_2, \ldots, a_n)$ rendezett n-es az argumentum.

Ha $f : A \to B$ egy függvény és $b = f(a)$, akkor egyúttal azt is mondjuk, hogy b az a f által létesített ***képe.*** Az $A' \subseteq A$ halmaznak az f által létesített képe pedig az

$$f(A') = \{b : b \in B \text{ és } b = f(a) \text{ egy alkalmas } a \in A' \text{ esetén}\}.$$

Az f ***értékkészlete*** az értelmezési tartományának a képe, azaz az $f(A)$ halmaz. Például az $f : \mathbf{N} \to \mathbf{N}$, $f(n) = 2n$ függvény értékkészlete a következő: $f(\mathbf{N}) = \{m : m \in \mathbf{N} \text{ és } m = 2n \text{ egy alkalmas } n \in \mathbf{N} \text{ esetén}\}$.

Egy függvényt ***szürjektívnek*** nevezünk, ha a képhalmaza megegyezik az értékkészletével. Például az $f(n) = \lfloor n/2 \rfloor$ függvény szürjektív $\mathbf{N}$-ről $\mathbf{N}$-re, mivel minden $\mathbf{N}$-beli szám előáll az f egy alkalmas helyen felvett értékeként. Ugyanakkor az $f(n) = 2n$ függvény nem szürjektív $\mathbf{N}$-ről $\mathbf{N}$-re, hiszen a 3-at egyetlen helyen sem veszi fel értékéül. Viszont az $f(n) = 2n$ függvény szürjektív a természetes számok halmazáról a páros természetes számok halmazára. A szürjektív $f : A \to B$ függvényről azt mondjuk, hogy az A halmazt a B-re ***képezi le.*** Amikor azt a kifejezést használjuk, hogy egy f függvény ráképezés, akkor ezen az f szürjektivitását értjük.

Az $f : A \to B$ függvény ***injektív,*** ha az értelmezési tartományának bármely két különböző eleméhez különböző értékeket rendel. Ez tehát azt jelenti, hogy ha $a, b \in A, a \neq b$, akkor $f(a) \neq f(b)$. Így például az $f(n) = 2n$ függvény injektív $\mathbf{N}$-ről $\mathbf{N}$-be, mivel minden páros b természetes szám az értelmezési tartomány egyetlen elemének az f által létesített képe, nevezetesen a $(b/2)$-é. Az $f(n) = \lfloor n/2 \rfloor$ $(n \in \mathbf{N})$ függvény nem injektív, ui. az 1 két argumentumhoz is értékként van hozzárendelve: a 2-höz is meg a 3-hoz is. Az injektív függvényt ***egy-egy értelmű leképezésnek*** is nevezik.

Az $f : A \to B$ függvény egy ***bijekció*** (vagy másképpen mondva az f ***bijektív***), ha injektív és szürjektív. Például az $f(n) = (-1)^n\lceil n/2\rceil$ függvény egy bijekció **N**-ről **Z**-re:

$$\begin{aligned} 0 &\mapsto 0\,, \\ 1 &\mapsto -1\,, \\ 2 &\mapsto 1\,, \\ 3 &\mapsto -2, \\ 4 &\mapsto 2\,, \\ &\vdots \end{aligned}$$

A szóban forgó függvény valóban injektív, mivel **Z**-nek nincs egyetlen olyan eleme sem, amely egynél több **N**-beli elemhez lenne hozzárendelve. Szürjektív is, hiszen a **Z** bármely eleme egy alkalmas **N**-beli elem képe. Az illető függvény tehát egy bijekció. A bijekciót ***kölcsönösen egyértelmű leképezésnek*** is nevezik, mivel az értelmezési tartomány minden elemének az értékkészlet pontosan egy eleme felel meg és fordítva. Egy A halmazon értelmezett, A-ra képező bijekciót ***permutációnak*** nevezünk.

Ha az f függvény bijektív, akkor az f^{-1}-gyel jelölt ***inverzét*** a következőképpen definiáljuk:

$$\text{legyen } f^{-1}(b) = a \text{ pontosan akkor, ha } f(a) = b.$$

Például az $f(n) = (-1)^n\lceil n/2\rceil$ $(n \in \mathbf{N})$ függvény inverze az

$$f^{-1}(m) = \begin{cases} 2m, & \text{ha } m \geq 0, \\ -2m-1, & \text{ha } m < 0. \end{cases}$$

Gyakorlatok

B.3-1. Legyen A is és B is egy-egy véges halmaz, $f : A \to B$ pedig egy függvény. Bizonyítsuk be, hogy

a. ha f injektív, akkor $|A| \leq |B|$;

b. ha f szürjektív, akkor $|A| \geq |B|$.

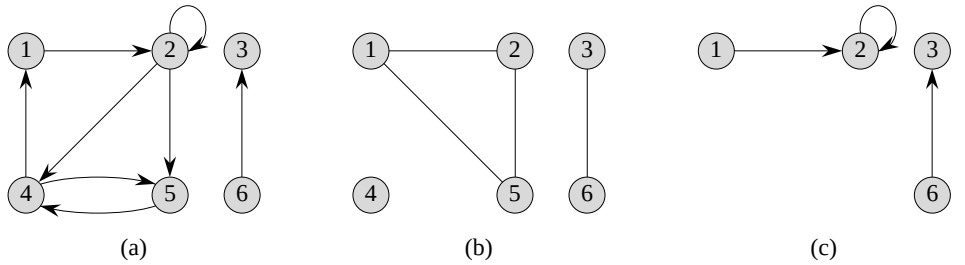
B.3-2. Igaz-e, hogy az $f : \mathbf{N} \to \mathbf{N}$, $f(x) = x+1$ függvény bijektív? Válaszoljunk a kérdésre akkor is, ha **N**-et kicseréljük **Z**-re.

B.3-3. Adjuk meg a bináris reláció inverzének a definícióját oly módon, hogy ha egy reláció bijektív függvény, akkor a definíciónk szerinti inverz relációja egyezzen meg a függvényértelemben vett inverzével.

B.3-4.⋆ Létesítsünk bijekciót **Z**-ről $\mathbf{Z} \times \mathbf{Z}$-re.

B.4. Gráfok

Ebben az alfejezetben irányított és irányítatlan gráfokkal foglalkozunk. Az Olvasó tapasztalni fogja, hogy a szakirodalomban megszokott bizonyos definíciók különböznek az általunk később megadandóktól, azonban a legtöbb esetben ez a különbség csak árnyalatnyi. A 22.1. alfejezetben megmutatjuk, hogyan lehet a gráfokat számítógépen ábrázolni.



B.2. ábra. Irányított és irányítatlan gráfok. **(a)** A $G = (V, E)$ gráf irányított, $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. A $(2, 2)$ él egy hurok. **(b)** A $G = (V, E)$ gráf irányítatlan, $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. A 4 egy izolált csúcs. **(c)** Az 1,2,3,6 csúcsok által meghatározott részgráfja az (a)-beli gráfnak.

Irányított gráfon egy $G = (V, E)$ rendezett párt értünk, ahol V egy véges halmaz, E pedig egy bináris reláció V-n. V-t a G ***csúcshalmazának,*** az elemeit ***csúcsoknak*** nevezzük. E a G ***éleinek a halmaza;*** az E-beli elemeket ***éleknek*** nevezzük. A B.2(a) ábrán egy olyan irányított gráfot szemléltetünk, amelynek a csúcsai a következők: 1, 2, 3, 4, 5, 6. Az ábrán a csúcsokat körökkel, az éleket ívekkel reprezentáltuk. Érdemes megjegyezni, hogy el őfordulhatnak ***hurkok*** is, azaz olyan élek, amelyek egy csúcsból ugyanabba a csúcsba vezetnek.

Egy ***irányítás nélküli*** (vagy ***irányítatlan***) $G = (V, E)$ gráf esetén az élek E halmaza a csúcsokból alkotott rendezetlen párokból áll. Ekkor tehát az élek $\{u, v\}$ alakú halmazok, ahol $u, v \in V$ és $u \neq v$. Állapodjunk meg azonban abban, hogy az élek jelölésére ekkor is az (u, v) szimbólumot fogjuk használni $\{u, v\}$ helyett; (u, v) és (v, u) egyazon élt fogja jelenteni. Irányítás nélküli gráfban hurkok nem fordulnak el ő, így minden él pontosan két különböző csúcsból alkotott (rendezetlen) pár. A B.2(b) ábrán egy irányítatlan gráfot látunk, amelynek a csúcsai: 1, 2, 3, 4, 5, 6.

Az irányított, ill. az irányítatlan gráfok számos, lényegében megegyező definíciója ismert, azonban egyes fogalmak jelentése bizonyos vonatkozásban eltér ő lehet. Ha (u, v) egy él a $G = (V, E)$ irányított gráfban, akkor azt mondjuk, hogy (u, v) az u ***csúcsból kiinduló*** és a v ***csúcsba mutató*** él. Például a B.2(a) ábrán a 2 csúcsból kiinduló élek: $(2, 2), (2, 4), (2, 5)$, a 2-be mutató élek pedig: $(1, 2), (2, 2)$. Ha (u, v) a $G = (V, E)$ irányítás nélküli gráfnak egy éle, akkor azt mondjuk, hogy (u, v) az u-ból és a v-b ől ***kiinduló él.*** A B.2(b) ábrán a 2-b ől kiinduló élek: $(1, 2), (2, 5)$.

Legyen (u, v) egy él a $G = (V, E)$ gráfban, ekkor a v csúcsot az u csúcs ***szomszédjának*** nevezzük. Ha a gráf irányítatlan, akkor a szomszédság relációja szimmetrikus. Ha viszont a gráf irányított, akkor ez a reláció nem feltétlenül tesz eleget a szimmetriának. Azt, hogy egy irányított gráfban v az u szomszédja, a következ őképpen is fogjuk jelölni: $u \rightarrow v$. A B.2. ábra (a) és (b) részében a 2 szomszédja az 1-nek, mivel az $(1, 2)$ él mindkét gráfhoz hozzátartozik. Ugyanakkor a B.2(a) ábrán az 1 *nem* szomszédja a 2-nek, hiszen $(2, 1)$ nem éle a gráfnak.

Irányítatlan gráfban egy csúcs ***fokszámának*** (vagy röviden ***fokának***) nevezzük az illető csúcsból kiinduló élek számát. Például a B.2(b) ábrán a 2 csúcs foka 2. Az olyan csúcsot, amelyiknek a foka nulla, ***izoláltnak*** nevezzük. Ilyen pl. a B.2(b) ábrán a 4 csúcs. Irányított gráfban egy csúcs ***kimenő fokszáma*** (vagy ***foka***) a bel őle kiinduló, a ***bemenő fokszáma (foka)*** pedig a csúcsba mutató élek száma. Egy irányított gráf valamely csúcsának a

fokszáma (vagy ***foka***) az illető csúcs kimenő és bemenő fokszámának az összege. Például a B.2(a) ábrán a 2 bemenő foka 2, a kimenő foka 3, így a fokszáma 5.

A $G = (V, E)$ gráfban az u csúcsot az u' csúccsal összekötő k ***hosszúságú úton*** csúcsoknak egy olyan (véges) $\langle v_0, v_1, \ldots, v_k\rangle$ sorozatát értjük, amelyre $u = v_0, u' = v_k$ és $(v_{i-1}, v_i) \in E \quad (i = 1, 2, \ldots, k)$. Az út hossza az útban szereplő élek száma. A szóban forgó út a $v_0, v_1, \ldots, v_k$ csúcsokat és a $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ éleket ***tartalmazza.*** Ha van olyan p út, amely u-t összeköti u'-vel, akkor azt mondjuk, hogy u' ***elérhető*** u-ból a p út mentén. Irányított gráf esetén mindezt az $u \to^p u'$ szimbólummal is fogjuk jelölni. Legyen továbbá definíció szerint minden csúcs önmagából elérhető. Egy út ***egyszerű,*** ha a benne szereplő csúcsok páronként különbözők. A B.2(a) ábrán $\langle 1, 2, 5, 4\rangle$ egy 3 hosszúságú egyszerű út. A $\langle 2, 5, 4, 5\rangle$ út nem egyszerű út.

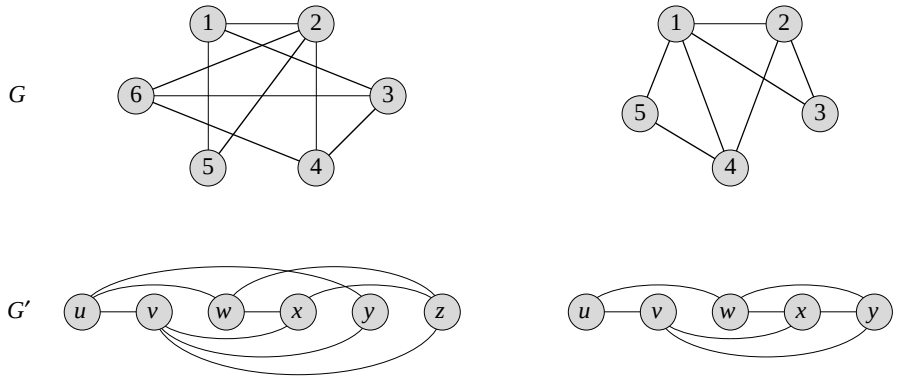
A $p = \langle v_0, v_1, \ldots, v_k\rangle$ út ***részén*** olyan utat értünk, amely a p-t alkotó csúcsoknak a következő alakú részsorozata: $\langle v_i, v_{i+1}, \ldots, v_j\rangle$, ahol $0 \le i \le j \le k$.

Egy irányított gráfban a $\langle v_0, v_1, \ldots, v_k\rangle$ út ***kört*** alkot, ha $v_0 = v_k$ és a szóban forgó út tartalmaz legalább egy élt. A kör ***egyszerű,*** ha még az is teljesül, hogy a $v_1, v_2, \ldots, v_k$ csúcsok különbözők. A hurok egy 1 hosszúságú kör. A $\langle v_0, v_1, \ldots, v_k\rangle$, $\langle v'_0, v'_1, \ldots, v'_k\rangle$ utak ugyanazt az utat alkotják, ha alkalmas j egész számmal $v'_i = v_{(i+j) \pmod k} \quad (i = 0, 1, \ldots, k-1)$. A B.2(a) ábrán az $\langle 1, 2, 4, 1\rangle$ út ugyanazt a kört alkotja, mint a $\langle 2, 4, 1, 2\rangle, \langle 4, 1, 2, 4\rangle$ utak. Ez a kör egyszerű, de az $\langle 1, 2, 4, 5, 4, 1\rangle$ nem az. A $(2, 2)$ által meghatározott $\langle 2, 2\rangle$ kör egy hurok. A hurok nélküli irányított gráfot ***egyszerűnek*** nevezzük. Irányítás nélküli gráfban akkor mondjuk, hogy a $\langle v_0, v_1, \ldots, v_k\rangle$ út (egyszerű) kört alkot, ha $k \ge 3, v_0 = v_k$ és a $v_1, v_2, \ldots, v_k$ csúcsok különbözőek. Például a B.2(b) ábrán az $\langle 1, 2, 5, 1\rangle$ út egy kör. A gráf ***körmentes,*** ha nem tartalmaz kört.

Egy irányítatlan gráfot ***összefüggőnek*** nevezünk, ha bármely két csúcsa összeköthető úttal. A gráf ***összefüggő komponensein*** a csúcsok alkotta ekvivalencia-osztályokat értünk, ahol az ekvivalencia-relációt a csúcsok közötti „elérhetőség” jelenti. A B.2(b) ábrán látható gráfnak három összefüggő komponense van: $\{1, 2, 5\}, \{3, 6\}$ és $\{4\}$. Az $\{1, 2, 5\}$-ben szereplő valamennyi csúcs az $\{1, 2, 5\}$ bármely más csúcsából elérhető. Egy irányítás nélküli gráf akkor összefüggő, ha pontosan egy összefüggő komponense van. Ez azt jelenti tehát, hogy minden csúcs elérhető bármely más csúcsból.

Az irányított gráfot akkor mondjuk ***erősen összefüggőnek,*** ha tetszőleges két csúcs esetén mindegyik elérhető a másikból. A gráf ***erősen összefüggő komponensei*** a csúcsok alkotta azon ekvivalencia-osztályok, amelyeket a kölcsönös elérhetőség mint ekvivalencia-reláció határoz meg. Egy irányított gráf akkor erősen összefüggő, ha pontosan egy erősen összefüggő komponense van. A B.2(a) ábrán látható gráfnak három erősen összefüggő komponense van: $\{1, 2, 4, 5\}$, $\{3\}$ és $\{6\}$. Az $\{1, 2, 4, 5\}$ halmazból bárhogy választunk is ki két csúcsot, ezek kölcsönösen elérhetőek. A 3, 6 csúcsok nem alkotnak erősen összefüggő komponenst, ui. a 6 nem érhető el a 3-ból.

A $G = (V, E), G' = (V', E')$ gráfokat ***izomorfaknak*** nevezzük, ha van olyan $f : V \to V'$ bijekció, hogy $(u, v) \in E$ akkor és csak akkor teljesül, ha $(f(u), f(v)) \in E'$ is fennáll. Más szóval tehát, ha a G csúcsait úgy tudjuk megfeleltetni a G' csúcsainak, hogy közben a G és a G' élei is egymáshoz rendelődnek. A B.3(a) ábrán a csúcsok $V = \{1, 2, 3, 4, 5, 6\}$ halmazához tartozó G gráf és a $V' = \{u, v, w, x, y, z\}$-hez tartozó G' gráf izomorf. Az $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$ előírással meghatározott $f : V \to V'$ leképezés eleget tesz az izomorfia definíciójában szereplő kikötéseknek. A B.3(b) ábrán



B.3. ábra. (a) Izomorf gráfok. A felső gráf csúcsait az $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ leképezés felelteti meg az alsó gráf csúcsainak. **(b)** A két gráf nem izomorf, ui. a felső gráfnak van olyan csúcsa, amelynek a fokszáma 4, míg az alsónak ilyen csúcsa nincs.

szereplő gráfok nem izomorfak. Igaz ugyan, hogy mindkét gráfnak 5 csúcsa és 7 éle van, de a felső gráfnak van olyan csúcsa, amelynek a foka 4, az alsónak viszont ilyen nincs.

Azt mondjuk, hogy a $G' = (V', E')$ gráf ***részgráfja*** a $G = (V, E)$ gráfnak, ha $V' \subseteq V$ és $E' \subseteq E$. Ha adott egy $V' \subseteq V$ halmaz, akkor a G gráf V' ***által meghatározott részgráfja*** a $G' = (V', E')$, ahol

$$E' = \{(u, v) \in E : u, v \in E'\}.$$

A B.2(a) ábrán az $\{1, 2, 3, 6\}$ csúcshalmaz által meghatározott részgráf (lásd B.2(c) ábra) éleinek a halmaza az $\{(1, 2), (2, 2), (6, 3)\}$ halmaz.

Egy irányítatlan $G = (V, E)$ ***gráfhoz tartozó irányított gráf*** az az irányított $G' = (V', E')$ gráf, amelyre $(u, v) \in E'$ akkor és csak akkor igaz, ha $(u, v) \in E$. Ez azt jelenti tehát, hogy bármely G-beli irányítatlan (u, v) élt a G-hez tartozó irányított gráfban két irányított éllel helyettesítünk, nevezetesen (u, v)-vel és (v, u)-val. Valamely irányított $G = (V, E)$ ***gráfhoz tartozó irányítatlan gráf*** az az irányítatlan $G' = (V', E')$ gráf, amelyre $(u, v) \in E'$ akkor és csak akkor áll fenn, ha $u \neq v$ és $(u, v) \in E$. A G' megadásakor tehát elhagyjuk az esetleges hurkokat, ill. az élek irányítását. (Mivel (u, v) és (v, u) az irányítás nélküli gráf ugyanazon éle, ezért az irányított gráfhoz tartozó irányítás nélküli gráf a szóban forgó élt csak „egyszer” tartalmazza, akkor is, ha az irányított gráf (u, v)-t is és (v, u)-t is tartalmazza.) Az irányított $G = (V, E)$ gráfban egy u csúcs ***szomszédjai*** azok a csúcsok, amelyek a G-hez tartozó irányítás nélküli gráfban szomszédjai u-nak. Tehát v akkor és csak akkor szomszédja u-nak, ha vagy $(u, v) \in E$, vagy pedig $(v, u) \in E$.

Egyes speciális tulajdonságokkal rendelkező gráfokat külön elnevezéssel is illetünk. Így például ***teljes gráfnak*** nevezünk egy irányítás nélküli gráfot, ha bármely két csúcsa szomszédos. A ***páros gráf*** egy olyan irányítás nélküli $G = (V, E)$ gráf, amelynél a V halmaz felbontható diszjunkt V_1, V_2 halmazok uniójára úgy, hogy $(u, v) \in E$ esetén vagy $u \in V_1$ és $v \in V_2$, vagy pedig $u \in V_2$ és $v \in V_1$. Ekkor tehát az összes él a V_1-ből a V_2-be, vagy fordítva, a V_2-ből a V_1-be halad. A körmentes, irányítás nélküli gráfot ***erdőnek,*** az összefüggő, körmentes és irányítás nélküli gráfot pedig ***(nyílt fának)*** nevezzük (lásd B.5.).

Tegyünk még röviden említést a gráf fogalom két lehetséges kiterjesztéséről. Ezek közül az egyik a ***többszörös gráf,*** amely lényegében egy irányítás nélküli gráf, de tartalmazhat többszörös éleket és hurkokat. A ***hipergráf*** ugyancsak egy irányítás nélküli gráf, amelyben minden ún. hiperél nem két csúcsot, hanem csúcsok halmazát köti össze. Számos olyan algoritmus, amely közönséges irányított vagy irányítatlan gráfokra íródott, adaptálható ezekre a gráfszerű struktúrákra is.

Egy irányítatlan $G = (V, E)$ gráf valamely $e = (u, v)$ élére vonatkozó ***összehúzásán*** azt a $G' = (V', E')$ gráfot értjük, amelyre $V' = V \setminus \{u, v\} \cup \{x\}$ (ahol x egy új csúcs), az élek E' halmazát pedig E-ből a következőképpen kapjuk: egyrészt elhagyjuk az (u, v) élt és minden olyan (u, w), (v, w) élt, amelyre w szomszédja u-nak vagy v-nek, másrészt minden ilyen w-re E'-höz tartozónak tekintjük az (x, w) élt.

Gyakorlatok

B.4-1. Egy egyetemi fogadás alkalmával a résztvevők kézfogással üdvözlik egymást. Mindenki mindenkivel kezet fog, közben pedig megjegyzi, hogy hány emberrel fogott kezet. A fogadás végén a tanszékvezető összegzi a kézfogások számát. Mutassuk meg, hogy az eredmény páros szám, ill. ennek mintegy általánosításaként bizonyítsuk be az ún. ***kézfogási lemmát:*** ha $G = (V, E)$ egy irányítatlan gráf, akkor

$$\sum_{v \in V} \text{fok}(v) = 2|E|,$$

ahol fok(v) a v csúcs fokszámát jelöli.

B.4-2. Lássuk be, hogy ha egy irányított vagy irányítatlan gráfban az u, v csúcsok úttal összeköthetők, akkor ezek a csúcsok egyszerű úttal is összeköthetők. Mutassuk meg továbbá, hogy ha egy irányított gráf tartalmaz kört, akkor tartalmaz egyszerű kört is.

B.4-3. Legyen $G = (V, E)$ összefüggő, irányítatlan gráf. Igazoljuk, hogy $|E| \geq |V| - 1$.

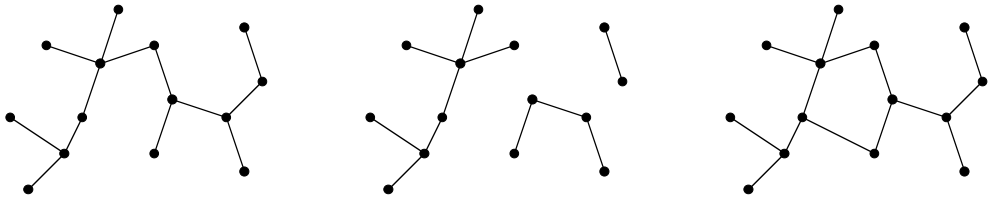
B.4-4. Mutassuk meg, hogy egy irányítatlan gráfban az elérhetőség egy ekvivalencia-reláció a gráf csúcsai között. Az ekvivalencia-reláció három tulajdonsága közül melyek teljesülnek akkor, ha a gráf irányított?

B.4-5. Mi lesz a B.2(a) ábrán látható irányított gráfhoz tartozó irányítatlan gráf, ill. a B.2(b) ábrabeli irányítatlan gráfhoz tartozó irányított gráf?

B.4-6.$\star$ Bizonyítsuk be, hogy egy hipergráf reprezentálható egy páros gráffal, ha a hipergráfban a hiperéllel való összeköthetőségen a páros gráfbeli szomszédságot értjük. (*Útmutatás.* A páros gráf csúcsainak egy halmaza feleljen meg a hipergráf csúcsainak, a csúcsok másik halmaza pedig feleljen meg a hiperéleknek.)

B.5. Fák

Amint azt általában a gráfokkal kapcsolatban már említettük, a fákat illetően is ismertek egymástól kissé eltérő definíciók, jelölések. Ebben az alfejezetben különböző típusú fákra vonatkozóan adjuk meg a fogalmakat, tulajdonságokat. A 10.4. és a 22.1. alfejezetekben láthatjuk, hogyan tárolhatók a fák a számítógép memóriájában.



B.4. ábra. (a) Nyílt fa. **(b)** Erdő. **(c)** Kört tartalmazó gráf, ami következésképpen se nem fa, se nem erdő.

B.5.1. Nyílt fák

A ***nyílt fa*** (lásd B.4. alfejezet) egy összefüggő, körmentes, irányítatlan gráf. A „nyílt” jelzőt a legtöbbször elhagyjuk, és egyszerűen fának nevezzük az előbbi tulajdonságú gráfot. Ha egy irányítatlan gráf körmentes, akkor ***erdőnek*** nevezzük. Számos, a fákra működő algoritmus működik erdőkre is. A B.4(a) ábra egy nyílt fát, a B.4(b) pedig egy erdőt szemléltet. Az utóbbi nem fa, mivel nem összefüggő. A B.4(c) ábrabeli gráf se nem fa, se nem erdő, ui. van benne kör.

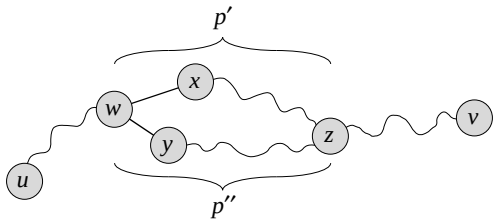
A következő tételben a nyílt fákkal kapcsolatban több fontos állítást fogalmazunk meg.

B.2. tétel (a nyílt fák tulajdonságai). *Legyen $G = (V, E)$ egy irányítatlan gráf. Ekkor az alábbi kijelentések egymással ekvivalensek.*

1. *G egy nyílt fa.*
2. *G bármely két csúcsához egyértelműen létezik egy őket összekötő egyszerű út.*
3. *G összefüggő, de tetszőleges élének az elhagyásával kapott részgráf már nem összefüggő.*
4. *G összefüggő és $|E| = |V| - 1$.*
5. *G körmentes és $|E| = |V| - 1$.*
6. *G körmentes, de akár egyetlen éllel is bővítve E-t, a kapott gráf már tartalmaz kört.*

Bizonyítás. (1) $\Longrightarrow$ (2). Mivel egy fa összefüggő, ezért bármely két G-beli csúcs legalább egy egyszerű úttal összeköthető. Legyen u és v egy-egy olyan csúcs, amelyek két különböző egyszerű úttal is összeköthetők, mondjuk p_1-gyel és p_2-vel (lásd B.5. ábra). Jelöljük w-vel azt a csúcsot, amelyben ezek az utak először eltérnek, azaz legyen w az első olyan csúcs p_1-en és p_2-n, amelyet a p_1 úton x, a p_2 úton y követ és $x \neq y$. Ugyanakkor legyen z az első olyan csúcs, amelyben a szóban forgó utak ismét találkoznak. A z tehát az első olyan w utáni csúcs p_1-en, amely rajta van p_2-n is. Tekintsük a p_1 útnak azt a p' részét, amely w-ből x-en át z-be vezet. Hasonlóan, legyen p'' a p_2 útnak az a része, amely w-ből y-on át z-be vezet. A p', p'' utak a végpontjaikon kívül nem tartalmaznak közös csúcsot. Következésképpen a p' és a p'' megfordításának az egyesítésével kapott út egy kör, amely ellentmond (1)-nek. Tehát, ha G egy fa, akkor bármely két csúcs legfeljebb egy egyszerű úttal köthető össze.

(2) $\Longrightarrow$ (3). Ha a G bármely két csúcsa egyértelműen köthető össze egy egyszerű úttal, akkor G összefüggő. Legyen $(u, v) \in E$ egy él. Ekkor ez az él egy u-ból v-be vezető út, a feltételeink miatt pedig ez az egyetlen ilyen út. Ha tehát (u, v)-t elhagyjuk G-ből, akkor nem lesz u-t v-vel összekötő út, így a visszamaradt részgráf nem összefüggő.



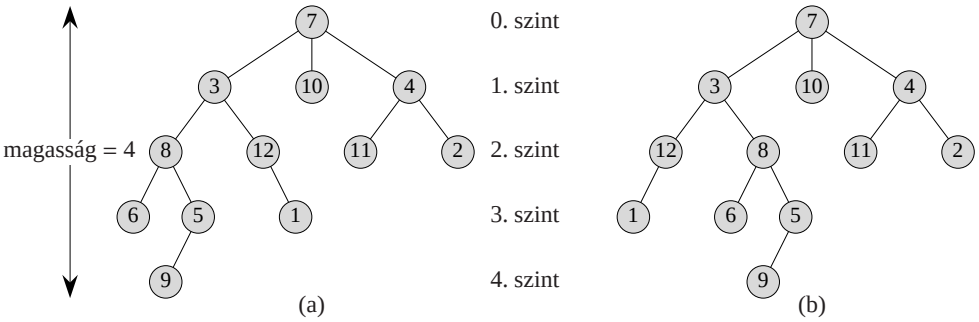
B.5. ábra. A B.2. tétel bizonyításának egy lépése: G egy nyílt fa (lásd (1)), ekkor (lásd (2)) G bármely két csúcsa egyértelműen köthető össze egyszerű úttal. Valóban, tegyük fel indirekt módon, hogy az u, v csúcsok két különböző egyszerű úttal is összeköthetők, p_1-gyel és p_2-vel. Ezek az utak először a w csúcsban ágaznak el, majd ezt követően először a z csúcsban futnak össze. A p_1 út és a p_2 megfordításával kapott út egyesítése kört alkot, ami ellentmond a feltételeknek.

(3) $\Longrightarrow$ (4). A feltételek miatt G összefüggő, a B.4-3. feladat alapján pedig $|E| \geq |V|-1$. Teljes indukcióval megmutatjuk, hogy $|E| \leq |V| - 1$. Ez $n = 1$ vagy $n = 2$ esetén triviális, hiszen ekkor a gráfnak $n - 1$ éle van. Tegyük fel tehát, hogy G-nek legalább 3 csúcsa van (azaz $n \geq 3$) és minden olyan, n-nél kevesebb csúcsot tartalmazó gráf esetén, amely eleget tesz (3)-nak, $|E| \leq |V|-1$ is igaz. Hagyjuk el a G-nek egy tetszőleges élét, ezzel a gráfot $k \geq 2$ összefüggő komponensre bontottuk. Mivel G eleget tesz (3)-nak, ezért minden komponens is rendelkezik a (3) tulajdonsággal. Az indukciós feltétel miatt tehát az élek száma összesen legfeljebb $|V| - k \leq |V| - 2$, azaz az elvett éllel együtt legfeljebb $|V| - 1$.

(4) $\Longrightarrow$ (5). Tegyük fel, hogy G összefüggő és $|E| = |V| - 1$. Azt kell belátnunk, hogy G nem tartalmaz kört. Tegyük fel ehhez indirekt módon azt, hogy G-ben van kör, amelyet mondjuk a k darab $v_1, v_2, \ldots, v_k$ csúcs határoz meg. Legyen $G_k = (V_k, E_k)$ a G gráfnak az ebből a körből álló részgráfja. Ekkor $|V_k| = |E_k| = k$. Ha $k < |V|$, akkor léteznie kell egy olyan $v_{k+1} \in V \setminus V_k$ csúcsnak, amely szomszédos valamely $v_i \in V_k$ csúccsal, ui. G összefüggő. Tekintsük a $G_{k+1} = (V_{k+1}, E_{k+1})$ gráfot mint a G-nek a $V_{k+1} = V_k \cup \{v_{k+1}\}$, $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$ halmazokkal meghatározott részgráfját. Világos, hogy $|V_{k+1}| = |E_{k+1}| = k+1$. Ha $k + 1 < n$, akkor az előző lépéshez hasonlóan definiáljuk G_{k+2}-t és így tovább, egészen addig, amíg olyan $G_n = (V_n, E_n)$ gráfhoz nem jutunk, amelyre $n = |V_n|, V_n = V$ és $|E_n| = |V_n| = |V|$. Mivel G_n részgráfja G-nek, ezért $E_n \subseteq E$, tehát $|E| \geq |V|$, ami viszont ellentmond az $|E| = |V| - 1$ feltételnek. A G gráf tehát valóban körmentes.

(5) $\Longrightarrow$ (6). Legyen a G egy kört nem tartalmazó olyan gráf, amelyre $|E| = |V|-1$. Jelöljük k-val a G összefüggő komponenseinek a számát. Mivel minden összefüggő komponens egy fa és (1)-ből (5) következik, ezért a G összefüggő komponenseiben összesen $|V| - k$ darab él van. Tehát k csak 1 lehet és G valóban egy fa. (2) következik (1)-ből: bármely két G-beli csúcs egyértelműen összeköthető egy egyszerű úttal. Következésképpen egy él hozzávételével kört kapunk.

(6) $\Longrightarrow$ (1). Tegyük fel, hogy G körmentes, de ha E-t bővítjük egy éllel, akkor az új gráf már tartalmaz kört. Azt kell belátnunk, hogy G összefüggő. Legyen ehhez u és v a G két tetszőleges csúcsa. Ha ezek esetleg nem szomszédosak, akkor vegyük hozzá a gráfhoz az (u, v) élt. A feltétel szerint az új gráfban lesz olyan kör, amelynek az élei (legfeljebb az (u, v) kivételével) mind G-hez tartoznak. Következésképpen van u-ból v-be vezető út. Mivel u-t és v-t tetszőlegesen választhattuk, ezért G összefüggő. ■



B.6. ábra. Gyökeres és rendezett fák. **(a)** Gyökeres fa, amelynek 4 a magassága. A fát a szokásos módon ábrázoltuk: felül van a gyökér (a 7-tel jelzett csúcs), a gyerekei (1 szintű csúcsok) egy sorban alatta, azok alatt hasonlóan a gyerekeik (2 szintű csúcsok) és így tovább. Ha a fa rendezett, akkor egy csúcs gyerekeinek a bal vagy jobb jelzője lényeges, egyébként nem. **(b)** Egy másik gyökeres fa. Mint ilyen, azonos az (a)-beli fával, rendezett faként azonban különbözik tőle, mivel a 3 csúcs gyerekeinek más a rendezése.

B.5.2. Gyökeres fák és rendezett fák

Gyökeresnek nevezünk egy olyan fát, amelyben az egyik csúcsnak kitüntetett szerepe van a többihez képest. Ezt a csúcsot a fa ***gyökerének*** vagy ***gyökércsúcsának*** nevezzük.[4] A B.6(a) ábrán egy gyökeres fát láthatunk 12 csúccsal, amelynek a 7-tel jelölt csúcs a gyökere.

Legyen T egy gyökeres fa, amelynek r a gyökere és tekintsük az x csúcsot. Az r-ből x-be vezető (egyértelműen meghatározott) út által tartalmazott bármely y csúcsot az x ***megelőzőjének*** nevezünk. Ha y megelőzője x-nek, akkor x az y ***rákövetkezője.*** (Minden csúcs önmagának egyszerre megelőzője és rákövetkezője is.) Az $x \neq y$ esetben y egy ***valódi megelőzője*** x-nek, x pedig egy ***valódi rákövetkezője*** y-nak. Azt a gyökeres fát, amelynek a gyökere x és maga a fa x-ből, ill. az x rákövetkezőiből áll, az x által meghatározott, ***x-ben gyökeredző részfának*** nevezzük. Például az 5.6(a) ábrán a 8-cal jelölt csúcsban gyökeredző részfa a 8, 6, 5, 9 csúcsokat tartalmazza.

Ha az r gyökerű T fában az r-ből x-be vezető út utolsó éle (y, x), akkor y-t az x ***szülőjének,*** x-et az y ***gyerekének*** nevezzük. A gyökér az egyetlen olyan csúcs T-ben, amelynek nincs szülője. Ha két csúcsnak ugyanaz a szülője, akkor ***testvéreknek*** mondjuk őket. A gyerek nélküli csúcs egy ún. ***külső csúcs*** vagy ***levél.*** Az egyéb csúcsokra azt mondjuk, hogy ***belső csúcsok.***

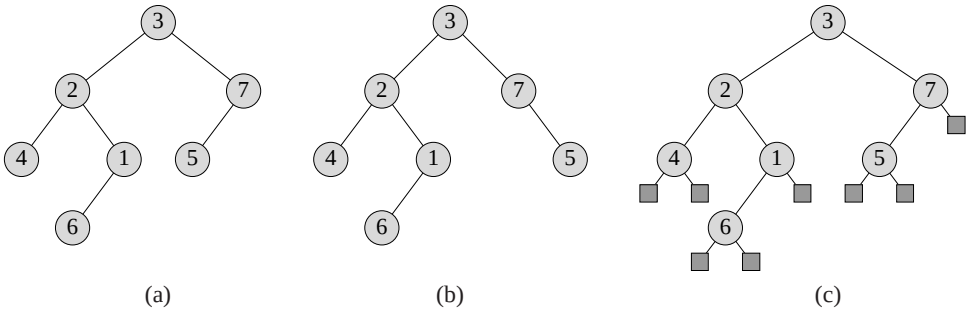
Egy gyökeres T fában valamely x csúcs gyerekeinek a számát x ***fokszámának*** nevezzük.[5] Az r gyökérből az x csúcsba vezető út hossza az x ***szintje.*** A T-beli csúcsok szintjei közül a legnagyobbat T ***magasságának*** nevezzük.

Azt mondjuk, hogy egy gyökeres fa ***rendezett,*** ha benne minden csúcs gyerekei rendezettek. Ez azt jelenti tehát, hogy ha egy csúcsnak k gyereke van, akkor van első, második, ..., k-adik gyerek. A B.6. ábrán szereplő két fa különbözik egymástól, ha őket mint rendezett fákat tekintjük, viszont mint gyökeres fák megegyeznek.

---

[4] Egy ilyen gráf csúcsaira gyakran használják a ***pont*** elnevezést is. (A gráfelmélettel kapcsolatos irodalomban a pont kifejezést többnyire a csúcs szinonimájaként használják.)

[5] Megjegyezzük, hogy a fokszám függ attól, hogy T-t gyökeres fának vagy nyílt fának tekintjük. Nyílt fában, mint általában minden irányítatlan gráfban, egy csúcs fokszáma a vele szomszédos csúcsok száma. Gyökeres fában azonban a fokszám a gyerekek száma, tehát a szülő nem számít bele a fokszámba.



B.7. ábra. Bináris fák. **(a)** A szokásos módon ábrázolt bináris fa. Egy csúcs bal gyerekét a csúcs alatt, tőle balra ábrázoltuk. Hasonlóan, a jobb gyerekét a csúcs alatt jobbra találjuk. **(b)** Az (a)-belitől különböző bináris fa. (a)-ban 5 a 7 csúcsnak bal gyereke, a jobb gyerek hiányzik. (b)-ben a helyzet éppen fordított. Mint rendezett fák ezek a gráfok megegyeznek, de mint bináris fák különböznek. **(c)** Az (a)-beli bináris fában a belső csúcsok kijelölésével egy teljes bináris fához jutunk: egy olyan rendezett fához, amelyben minden belső csúcsnak a fokszáma 2. A fa leveleit négyzetekkel szemléltettük.

B.5.3. Bináris fák és bővített fák

Egy bináris fát a legegyszerűbben rekurzívan adhatunk meg. Nevezetesen, a T ***bináris fa*** egy olyan struktúra csúcsoknak valamely véges halmazán, amely

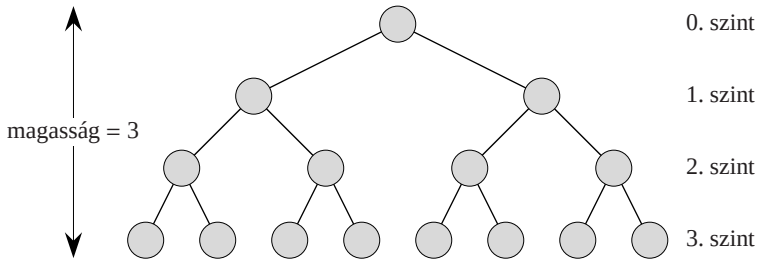
- vagy nem tartalmaz csúcsot,
- vagy pedig az általa tartalmazott csúcsok három diszjunkt halmazba sorolhatók: a ***gyökér,*** egy ***bal részfának*** nevezett bináris fa és egy ***jobb részfának*** nevezett bináris fa.

Az olyan bináris fát, amely nem tartalmaz csúcsot, ***üres*** vagy ***null fának*** nevezzük. Ha a bal részfa nem üres, akkor a gyökere a belső fa gyökerének a ***bal gyereke.*** Hasonlóan, egy nem üres jobb részfa gyökere a belső gyökerének a ***jobb gyereke.*** Ha egy részfa üres, akkor ***hiányzó gyerekről*** beszélünk. A B.7(a) ábra egy bináris fát ábrázol.

Egy bináris fa nem egyszerűen egy olyan rendezett fa, amelyben minden csúcs fokszáma legfeljebb kettő. Például, ha egy bináris fában egy csúcsnak pontosan egy gyereke van, akkor ennek a helyzete szempontjából fontos, hogy ***bal gyerekről*** vagy ***jobb gyerekről*** van-e szó. A rendezett fákban ilyen különbségtevés nincs. A B.7(b) ábrabeli bináris fa éppen ebben különbözik a B.7(a) ábrán láthatótól. Ugyanakkor mint rendezett fák megegyeznek.

A B.7(c) ábra azt mutatja, hogyan lehet egy bináris fával kapcsolatban a bővítést a rendezett fa belső csúcsai révén szemléltetni: a bináris fában minden hiányzó gyereket helyettesítünk egy olyan csúccsal, amelynek nincs gyereke. Ezeket a leveleket az ábrán négyzetekkel szemléltettük. Az eredményül kapott fa egy ***teljes bináris fa:*** minden csúcs vagy egy levél, vagy pedig a fokszáma pontosan kettő. Nincs benne tehát olyan csúcs, amelynek a fokszáma egy.

A bináris fát a rendezett fától megkülönböztető bővítést kiterjeszthetjük olyan fákra is, amelyben a csúcsonkénti gyerekek száma kettőnél nagyobb. Egy ún. ***súlyozott fában*** a csúcsok gyerekeit különböző pozitív egész számokkal indexeljük. Azt mondjuk, hogy egy csúcs i-edik gyereke ***hiányzó,*** ha nincs olyan gyerek, amelynek i az indexe. A ***k-ad rendű fa*** egy olyan súlyozott fa, amelyben minden csúcs k-nál nagyobb indexű fiai hiányoznak. Így például egy bináris fa egy k-adrendű fa, amelyre $k = 2$.



B.8. ábra. Teljes bináris fa: a magassága 3, a leveleinek a száma 8, a belső csúcsainak a száma 7.

***k*-ad rendű teljes fának** nevezünk egy k-adrendű fát, ha benne minden levélnek ugyanaz a szintje és az összes belső csúcs fokszáma k. A B.8. ábrán egy olyan teljes bináris fa látható, amelynek a magassága 3. Hány levele van egy h magasságú teljes k-adrendű fának? A gyökérnek k gyereke van, ezek mindegyikének 1 a szintje. Minden egyes gyereknek van k gyereke, az utóbbiak szintje 2 és így tovább. Tehát a h szintű levelek száma k^h. Következésképpen egy teljes k-adrendű fa magassága $\log_k n$, ahol n a levelek száma. A h magasságú teljes k-adrendű fa belső csúcsainak a száma pedig az (A.5) egyenlőség alapján

$$\begin{aligned} 1 + k + k^2 + \cdots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1}. \end{aligned}$$

Ez azt jelenti, hogy egy teljes bináris fának $2^h - 1$ belső csúcsa van.

Gyakorlatok

B.5-1. Szemléltessük az összes olyan ***a.*** fát, amelynek a csúcsai A, B és C; ***b.*** gyökeres fát, amelynek a csúcsai A, B, C és a gyökere A; ***c.*** rendezett fát, amelynek a csúcsai A, B, C és a gyökere A; ***d.*** bináris fát, amelynek a csúcsai A, B, C és a gyökere A.

B.5-2. Legyen $G = (V, E)$ egy olyan körmentes, irányított gráf, amelyben egy alkalmas $v_0 \in V$ csúccsal bármely $v \in V$ csúcs esetén egyértelműen létezik v_0-ból v-be vezető út. Lássuk be, hogy a G-hez tartozó irányítatlan gráf egy fa.

B.5-3. Igazoljuk teljes indukcióval, hogy bármely bináris fában a másodfokú csúcsok száma kisebb, mint a levelek száma.

B.5-4. Használjunk teljes indukciót annak az igazolására, hogy az n csúcsú bináris fák magassága legalább $\lfloor \lg n \rfloor$.

B.5-5.⋆ Legyen egy teljes bináris fa ***belső úthossza*** a belső csúcsok szintjeinek az összege. Hasonlóan, legyen a ***külső úthossz*** a levelek szintjeinek az összege. Tekintsünk egy olyan teljes bináris fát, amelynek n belső csúcsa van, a belső úthossza i, a külső úthossza pedig e. Bizonyítsuk be, hogy $e = i + 2n$.

B.5-6.⋆ Egy T bináris fa minden d szintű x leveléhez rendeljük hozzá a $w(x) = 2^{-d}$ „súlyt.” Mutassuk meg, hogy $\sum_x w(x) \leq 1$, ahol az összegzés a T összes levelére vonatkozik. (Ez az ún. ***Kraft–egyenlőtlenség***.)

B.5-7.⋆ Bizonyítsuk be, hogy ha egy bináris fának L levele van, akkor tartalmaz olyan részfát, amelyben a levelek száma $L/3$ és $2L/3$ közé esik (a határokat is beleértve).

Feladatok

B-1. Gráfok színezése

Egy irányítás nélküli $G = (V, E)$ gráf ***k-színezésén*** olyan $c : V \to \{0, 1, \ldots, k-1\}$ függvényt értünk, amelyre minden $(u, v) \in E$ esetén $c(u) \neq c(v)$. Más szóval tehát, ha a $0, 1, \ldots, k-1$ számok k színt reprezentálnak, akkor a szomszédos csúcsok különböző színűek.

a. Lássuk be, hogy bármely fa 2-színezhető.

b. Bizonyítsuk be, hogy a következő kijelentések egymással ekvivalensek:

1. G páros gráf.
2. G 2-színezhető.
3. G-ben nincs páratlan hosszúságú kör.

c. Legyen d a G gráfban lévő csúcsok fokszámának a maximuma. Igazoljuk, hogy a G gráf $(d+1)$-színezhető.

d. Tegyük fel, hogy G-nek $O(|V|)$ éle van. Mutassuk meg, hogy a G gráf $O(\sqrt{|V|})$-színezhető.

B-2. Baráti gráfok

Fogalmazzuk át az alábbi állításokat irányítatlan gráfokra vonatkozó tételekké, és bizonyítsuk is be őket. (Feltesszük, hogy a *barátság* szimmetrikus, de nem reflexív reláció.)

a. Bármely, legalább 2 főből álló társaságban van legalább két olyan ember, akiknek a társaság tagjai között ugyanannyi barátja van.

b. Egy 6 fős társaságban mindig találunk három olyan embert, akik vagy kölcsönösen egymás barátai, vagy egyik sem barátja a másik kettő közül senkinek sem.

c. Tetszőleges (legalább 2 fős) társaság két részre osztható úgy, hogy a társaság minden tagjára igaz a következő: az illető barátainak legalább a fele *nem* ahhoz a részhez tartozik, amelyhez ő.

d. Ha egy társaságban mindenkinek legalább annyi barátja van, mint a társaság fele, akkor a társaság leültethető egy asztal köré úgy, hogy mindenki két barátja között üljön.

B-3. Kettéosztott fák

Számos, gráfokon működő algoritmus megköveteli azt, hogy a gráfot (a csúcsainak a felosztása révén) két közel egyenlő nagyságú részgráfra osszuk. Mindez néhány él elhagyásával fák kettéosztásának a feladatához vezet. Megköveteljük, hogy bármely olyan két csúcs, amely az élek elhagyásával egy részfában van, ugyanabba a felosztásba tartozzon.

a. Bizonyítsuk be, hogy egy n-csúcsú bináris fa csúcsait egyetlen él elhagyásával két olyan halmazba, A-ba és B-be sorolhatjuk, amelyekre $|A| \leq 3n/4$ és $|B| \leq 3n/4$ igaz.

b. Mutassuk meg, hogy az előbbi feladatban szereplő 3/4 optimális. Adjunk tehát példát olyan egyszerű bináris fára, hogy egy alkalmas él eltávolításával $|A| = 3n/4$.

c. Lássuk be, hogy ha egy n-csúcsú fa élei közül legalább $O(\lg n)$ élt elhagyunk, akkor a csúcsokat olyan A és B halmazba sorolhatjuk, amelyekre $|A| = \lfloor n/2 \rfloor$ és $|B| = \lceil n/2 \rceil$ teljesül.

Megjegyzések a fejezethez

Már G. Boole mint a szimbolikus logika első művelője, egy 1854-ben megjelent könyvében számos halmazelméleti jelölést vezetett be. A modern halmazelmélet megalapozása – amely G. Cantor nevéhez fűződik – 1874 és 1895 közé tehető. Cantort elsősorban a véges számosságú halmazok érdekelték. A függvényfogalom G. W. Leibniznek tulajdonítható, aki a függvényeket különböző matematikai képletek leírására használta. Az általa adott definíciót aztán nagymértékben általánosították. A gráfelmélet kialakulása 1736-tól számítható, amikor L. Euler a következőt bizonyította be: nem lehet bejárni Königsberg 7 hídját úgy, hogy mindegyik hidat pontosan egyszer érintsük, és a kiindulási ponthoz térjünk vissza.

A gráfelmélet fogalmainak és eredményeinek egy jól használható, tömör összefoglalóját találja az Olvasó Harary[138] könyvében.

C. Leszámlálás és valószínűség

Ebben a fejezetben a kombinatorika és a valószínűségszámítás elemeiről adunk áttekintést. Amennyiben az olvasó kellő háttérrel rendelkezik ezeken a területeken, nyugodtan átlapozhatja a fejezet elejét, és összpontosítsa figyelmét a későbbi alfejezetekre. A legtöbb fejezet nem igényel valószínűségszámításbeli ismereteket, azonban néhány fejezetnél ezek a ismeretek nélkülözhetetlenek.

A C.1. alfejezetben a leszámlálás elméletének elemi eredményeit tekintjük át, beleértve a permutációkra és a kombinációkra vonatkozó alapvető képleteket is. A valószínűség axiómáit és a valószínűségi eloszlásokkal kapcsolatos alapfogalmakat ismertetjük a C.2. alfejezetben. A C.3. alfejezetben bevezetjük a valószínűségi változó fogalmát, és felsoroljuk a várható érték és a szórásnégyzet tulajdonságait. A C.4. alfejezetben a Bernoulli-kísérletsorozat vizsgálatánál felmerülő geometriai és binomiális eloszlás tanulmányozására kerül sor. A C.5. alfejezetben az eloszlás „farkainak” részletesebb elemzésével folytatjuk a binomiális eloszlás vizsgálatát.

C.1. Leszámlálás

A leszámlálás elmélete anélkül próbál meg választ adni a „Mennyi?” kérdésre, hogy az összeszámlálást ténylegesen elvégezné. Megkérdezhetjük például, hogy „Hány különböző n bit hosszúságú szám létezik?”, vagy „Hányféle sorbarendezése lehetséges n különböző elemnek?”. Ebben az alfejezetben a leszámlálás elméletének elemeit tekintjük át. Mivel az anyag egy része a halmazok alapismeretét feltételezi, azt ajánljuk, hogy az olvasó előbb tekintse át a B.1. alfejezetet.

Az összeadás és a szorzás szabálya

Néha az összeszámlálandó dolgok halmazát előtudjuk állítani diszjunkt halmazok uniójaként vagy halmazok Descartes-féle szorzataként.

Az ***összeadás szabálya*** azt mondja ki, hogy két diszjunkt halmazból annyiféleképpen választható ki egy elem, amennyi a halmazok számosságainak az összege. Azaz, ha A és B két olyan véges halmaz, amelyeknek nincs közös elemük, akkor $|A \cup B| = |A| + |B|$, ami a (B.3) egyenletből következik. Például, egy autó rendszámának minden egyes helyére vagy

egy betű, vagy pedig egy számjegy kerülhet. Mivel betűből 26-, számjegyből pedig 10-félét választhatunk, ezért a lehetőségek száma minden egyes hely esetén $26 + 10 = 36$.

A ***szorzás szabálya*** azt mondja ki, hogy egy rendezett pár kiválasztására annyi lehetőség van, amennyi az első és a második elemnél rendelkezésre álló lehetőségek szorzata. Azaz, ha A és B két véges halmaz, akkor $|A \times B| = |A| \cdot |B|$, ami éppen a (B.4) egyenlet. Például, ha egy fagylaltozó 28-féle fagylaltot és 4-féle díszítést kínál, akkor $28 \cdot 4 = 112$ fajta olyan parfé készíthető, amely egy gömb fagylaltból és egy díszítésből áll.

Sztringek

Az S véges halmaz feletti ***sztringen*** egy az S elemeiből képzett sorozatot értünk. Például, 8 darab 3 hosszúságú bináris sztring létezik:

$$000,\ 001,\ 010,\ 011,\ 100,\ 101,\ 110,\ 111.$$

Egy k elemből álló sztringet ***k-hosszú sztringnek*** fogunk nevezni. Az s sztring egymás utáni elemeinek egy részsorozatát s egy s' ***rész-sztringjének*** nevezzük. Egy sztring ***k-hosszú rész-sztringjén*** egy k elemből álló rész-sztringet fogunk érteni. Például, 010 egy 3-hosszú rész-sztringe a 01101001 sztringnek (az a 3-hosszú rész-sztring, amely a 4. számjegynél kezdődik), míg az 111 sztring nem rész-sztringe 01101001–nek.

Az S halmaz elemeiből képzett k-hosszú sztringre gondolhatunk úgy, mint az S^k Descartes-szorzat egy elemére; így a k hosszúságú sztringek száma $|S|^k$. Például, a bináris, k-hosszú sztringek száma 2^k. Ha egy k-hosszú sztringet akarunk konstruálni egy n elemű halmaz elemeiből, akkor n-féleképpen választhatjuk meg az első elemet; minden egyes választás után n-féleképpen választhatjuk meg a második elemet; és így tovább k-szor. Ez a konstrukció mutatja, hogy a k-hosszú sztringek száma megegyezik az $n \cdot n \cdots n = n^k$ k tényezős szorzattal.

Permutációk

Az S véges halmaz egy ***permutációján*** az S összes eleméből képzett olyan rendezett sorozatot értünk, amelyben minden elem pontosan egyszer szerepel. Például, ha $S = \{a, b, c\}$, akkor S-nek 6 darab permutációja létezik:

$$abc,\ acb,\ bac,\ bca,\ cab,\ cba.$$

Egy n elemű halmaz permutációinak száma $n!$, hiszen a sorozat első elemét n-féleképpen választhatjuk meg, a második elemét $n-1$-féleképpen, a harmadik elemét $n-2$-féleképpen és így tovább.

S egy ***k-ad osztályú permutációján*** az S k számú eleméből képzett olyan rendezett sorozatot értünk, amelyben egyetlen elem sem fordul elő egynél többször. (Így a szokásos permutáció pontosan egy n elemű halmaz n-ed osztályú permutációjával egyezik meg.) Az $\{a, b, c, d\}$ halmaz 12 darab másodosztályú permutációja:

$$ab,\ ac,\ ad,\ ba,\ bc,\ bd,\ ca,\ cb,\ cd,\ da,\ db,\ dc.$$

Egy n elemű halmaz k-ad osztályú permutációinak száma

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}\,, \tag{C.1}$$

hiszen n-féleképpen választhatjuk meg az első elemet, $n-1$-féleképpen a második elemet és így tovább, míg k elemet ki nem választunk, ahol utoljára $n-k+1$ elemből választhatunk.

Kombinációk

Az n elemű S halmaz egy ***k-ad osztályú kombinációján*** egyszerűen az S egy k elemű részhalmazát értjük. Például, az $\{a, b, c, d\}$ 4 elemű halmaznak 6 darab másodosztályú kombinációja van:

$$ab,\ ac,\ ad,\ bc,\ bd,\ cd.$$

(Az $\{a, b\}$, és hasonlóan a többi két elemű halmaz jelölésénél is, az ab rövidítést használtuk.) Úgy adhatjuk meg egy n elemű halmaznak egy k-ad osztályú kombinációját, hogy kiválasztjuk az n elemű halmaz k (különböző) elemét.

Egy n elemű halmaz k-ad osztályú kombinációinak száma megadható a halmaz k-ad osztályú permutációinak száma segítségével. Minden k-ad osztályú kombinációnak pontosan $k!$ számú különböző permutációja létezik, amely mindegyike k-ad osztályú permutációja az n elemű halmaznak. Így egy n elemű halmaz k-ad osztályú kombinációinak száma egyenlő a k-ad osztályú permutációk száma osztva $k!$-sal; ez a szám a (C.1) egyenlet alapján pedig

$$\frac{n!}{k!(n-k)!}\ . \tag{C.2}$$

A $k = 0$ esetben ez a képlet azt fejezi ki, hogy 1 (és nem 0) annak a száma, hogy egy n elemű halmazból hányféleképpen választhatunk ki 0 számú elemet, ugyanis $0! = 1$.

Binomiális együtthatók

Egy n elemű halmaz k-ad osztályú kombinációinak számára az $\binom{n}{k}$ (olvasd „n alatt a k”) jelölést fogjuk használni. A (C.2) egyenlet alapján azt kapjuk, hogy

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}\ .$$

Mivel ez a képlet szimmetrikus k-ban és $n-k$-ban, így

$$\binom{n}{k} = \binom{n}{n-k}. \tag{C.3}$$

Ezek a mennyiségek ***binomiális együtthatókként*** is ismertek, mivel megjelennek az

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} \tag{C.4}$$

binomiális tételben. A binomiális tételt az $x = y = 1$ speciális esetben alkalmazva kapjuk, hogy

$$2^n = \sum_{k=0}^{n} \binom{n}{k}.$$

Ez a képlet annak felel meg, hogy úgy számoljuk össze a 2^n számú bináris, n-hosszú sztringet, hogy megnézzük hány egyest tartalmaznak; $\binom{n}{k}$ számú olyan bináris, n-hosszú sztring

van, amely pontosan k db 1-est tartalmaz, ugyanis $\binom{n}{k}$-féleképpen választhatunk ki az n helyből k-t az 1-esek számára.

Számos binomiális együtthatókkal kapcsolatos azonosság ismert. Az alfejezet végén található feladatok alkalmat adnak az Olvasónak arra, hogy egy párat bebizonyítson közülük.

A binomiális együttható becslései

Néha szükségünk lehet arra, hogy becslést adjunk a binomiális együtthatóra. $1 \le k \le n$ esetén az alábbi alsó korlátot kapjuk:

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\ &\ge \left(\frac{n}{k}\right)^k .\end{aligned}$$

A (3.17) Stirling-formulából származó $k! \ge (k/e)^k$ egyenlőtlenség segítségével az alábbi felső korlátot kapjuk:

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &\le \frac{n^k}{k!} \\ &\le \left(\frac{en}{k}\right)^k . \end{aligned} \tag{C.5}$$

Indukcióval bizonyíthatjuk az

$$\binom{n}{k} \le \frac{n^n}{k^k (n-k)^{n-k}} \tag{C.6}$$

becslést minden $0 \le k \le n$ esetén (lásd a C.1-12. gyakorlatot), ahol az egyszerűség kedvéért feltettük, hogy $0^0 = 1$. Ez a korlát $k = \lambda n$ esetén, ahol $0 \le \lambda \le 1$, az

$$\begin{aligned}\binom{n}{\lambda n} &\le \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n \\ &= 2^{nH(\lambda)}\end{aligned}$$

alakot ölti, ahol

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \tag{C.7}$$

a ***(bináris) entrópia függvény,*** és ahol az egyszerűség kedvéért feltettük, hogy $0 \lg 0 = 0$, így $H(0) = H(1) = 0$.

Gyakorlatok

C.1-1. Hány k-hosszú rész-sztringe van egy n-hosszú sztringnek? (Azonos, de különböző helyeken lévő rész-sztringeket különbözőeknek tekintünk.) Hány rész-sztringe van egy n-hosszú sztringnek összesen?

C.1-2. Egy n-változós, m-értékű Boole-függvényen egy, az $\{\text{IGAZ},\text{HAMIS}\}^n$ halmazt az $\{\text{IGAZ}, \text{HAMIS}\}^m$ halmazba képező függvényt értünk. Hány n-változós, 1-értékű Boole-függvény létezik? Hány n-változós, m-értékű Boole-függvény létezik?

C.1-3. Hányféleképpen ültethetünk le n professzort egy kör alakú konferenciaasztal köré? Két ültetést megegyezőnek tekintünk, ha az egyik forgatással átvihető a másikba.

C.1-4. Hányféleképpen választhatunk ki három különböző számot az $\{1, 2, \ldots, 100\}$ halmazból úgy, hogy az összegük páros legyen?

C.1-5. Bizonyítsuk be az

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{C.8}$$

azonosságot, ahol $0 < k \leq n$.

C.1-6. Bizonyítsuk be az

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

azonosságot, ahol $0 \leq k < n$.

C.1-7. k számú elemnek n számú elem közül való kiválasztásánál egy elemet kitüntethetünk a többi közül, és figyelhetjük, hogy vajon ezt a kitüntetett elemet kiválasztjuk-e vagy sem. Ennek az észrevételnek a segítségével bizonyítsuk be, hogy

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

C.1-8. A C.1-7. gyakorlat eredményét felhasználva készítsünk táblázatot a $\binom{n}{k}$ binomiális együtthatókra az $n = 0, 1, \ldots, 6$ és $0 \leq k \leq n$ értékekkel úgy, hogy a táblázat csúcsára a $\binom{0}{0}$ kerüljön, a következő sorába az $\binom{1}{0}$ és az $\binom{1}{1}$ és így tovább. A binomiális együtthatók így nyert táblázatát ***Pascal-háromszögnek*** nevezzük.

C.1-9. Igazoljuk, hogy

$$\sum_{i=1}^{n} i = \binom{n+1}{2}.$$

C.1-10. Mutassuk meg, hogy minden $n \geq 0$ és $0 \leq k \leq n$ esetén az $\binom{n}{k}$ kifejezés maximuma akkor vétetik fel, ha $k = \lfloor n/2 \rfloor$ vagy $k = \lceil n/2 \rceil$.

C.1-11.$\star$ Lássuk be, hogy minden $n \geq 0$, $j \geq 0$, $k \geq 0$ és $j + k \leq n$ esetén

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k}. \tag{C.9}$$

Adjunk mind algebrai, mind pedig $j + k$ elemnek n elemből való kiválasztásán alapuló kombinatorikai bizonyítást. Adjunk példát arra, amikor nem áll fenn egyenlőség.

C.1-12.$\star$ Bizonyítsuk be a (C.6) egyenlőtlenséget k szerinti teljes indukcióval $k \leq n/2$ esetén, majd a (C.3) egyenlőség alapján ezt terjesszük ki az összes $k \leq n$ számra.

C.1-13.$\star$ A Stirling-formula segítségével bizonyítsuk be, hogy

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right). \tag{C.10}$$

C.1-14.⋆ A $H(\lambda)$ entrópia függvény differenciálásával mutassuk meg, hogy az a maximumát a $\lambda = 1/2$ helyen veszi fel. Mennyi $H(1/2)$ értéke?

C.1-15.⋆ Mutassuk meg, hogy minden $n \geq 0$ egész esetén

$$\sum_{k=0}^{n} \binom{n}{k} k = n2^{n-1}. \tag{C.11}$$

C.2. Valószínűség

A valószínűségszámítás nélkülözhetetlen eszköz valószínűségi és véletlenített algoritmusok tervezésénél és elemzésénél. Ebben az alfejezetben a valószínűségszámítás alapjait tekintjük át.

A valószínűséget egy S halmazon, az ún. ***eseménytéren*** definiáljuk, amelynek elemeit ***elemi eseményeknek*** fogjuk nevezni. Minden elemi esemény tekinthető úgy, mint egy kísérlet egy lehetséges kimenetele. Két egymástól megkülönböztethető érme feldobásakor az eseménytér a {F, I} halmaz feletti összes 2-hosszú sztringből áll:

$$S = \{\text{FF}, \text{FI}, \text{IF}, \text{II}\}.$$

Eseményen[1] az S eseménytér egy részhalmazát értjük. Például, két érme feldobásánál az az esemény, hogy egy fejet és egy írást dobunk: {FI, IF}. Az S eseményt ***biztos eseménynek,*** az $\emptyset$ eseményt ***lehetetlen eseménynek*** fogjuk nevezni. Ha $A \cap B = \emptyset$, akkor azt mondjuk, hogy az A és B események ***egymást kizáróak.*** Néha egy $s \in S$ elemi eseményt célszerű az $\{s\}$ eseménnyel azonosítani. A definíció alapján bármely két elemi esemény egymást kizáró.

A valószínűség axiómái

Az S eseménytéren adott $\Pr\{\}$ ***valószínűségeloszlás*** olyan S eseményeihez valós számokat hozzárendelő leképezés, amely eleget tesz a következő ***valószínűségi axiómáknak:***

1. $\Pr\{A\} \geq 0$ minden A esemény esetén.
2. $\Pr\{S\} = 1$.
3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ bármely két egymást kizáró A és B esemény esetén. Általánosabban, az $A_1, A_2, \ldots$ események bármely (véges vagy megszámlálhatóan végtelen) egymást páronként kizáró sorozatára fennáll, hogy

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}.$$

[1] A valószínűségszámítás általános elméletében az S eseménytérnek lehetnek olyan részhalmazai is, amelyek nem események. Ez a helyzet rendszerint akkor fordul elő, amikor az eseménytér nem megszámlálhatóan végtelen halmaz. Egy eseménytér eseményeinek a halmazával kapcsolatban a következmény az, hogy zárt legyen a komplementer-képzésre és a véges vagy megszámlálhatóan végtelen unió-, illetve metszet-képzésre nézve. A legtöbb valószínűségeloszlás, mellyel foglalkozni fogunk, véges vagy megszámlálhatóan végtelen eseménytéren lesz definiálva, és ezért általában az eseménytér összes részhalmazát eseménynek fogjuk tekinteni. Az egyetlen figyelemre méltó kivétel az egyenletes eloszlás lesz, amelyet röviden ismertetni fogunk.

A $\Pr\{A\}$ mennyiséget az A esemény ***valószínűségének*** nevezzük. Itt jegyezzük meg, hogy a 2. axióma egyszerű normálást fejez ki: valójában semmi sem indokolja azt, hogy a biztos esemény valószínűsége 1 legyen, hacsak az nem, hogy ez a választás természetes és kényelmes.

A valószínűség számos tulajdonsága közvetlenül adódik a fenti axiómákból és a halmazelméleti alapismeretekből (lásd a B.1. alfejezetet). A lehetetlen esemény valószínűsége $\Pr\{\emptyset\} = 0$. Ha $A \subseteq B$, akkor $\Pr\{A\} \leq \Pr\{B\}$. Ha $\overline{A}$–sal jelöljük az $S \setminus A$ eseményt (az A ***komplementerét***), akkor $\Pr\left\{\overline{A}\right\} = 1 - \Pr\{A\}$. Bármely két A és B esemény esetén

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \tag{C.12}$$

$$\leq \Pr\{A\} + \Pr\{B\}\,. \tag{C.13}$$

Az érmedobással kapcsolatos példánkban tegyük fel, hogy mind a 4 elemi esemény 1/4 valószínűségű. Ekkor annak valószínűsége, hogy legalább egy fejet dobunk,

$$\begin{aligned}\Pr\{\text{FF,FI,IF}\} &= \Pr\{\text{FF}\} + \Pr\{\text{FI}\} + \Pr\{\text{IF}\} \\ &= \frac{3}{4}\,.\end{aligned}$$

Másképpen, mivel annak a valószínűsége, hogy kevesebb mint egy fejet dobunk $\Pr\{\text{II}\} = 1/4$, így annak a valószínűsége, hogy legalább egy fejet dobunk $1 - 1/4 = 3/4$.

Diszkrét valószínűségeloszlások

Egy valószínűségeloszlást ***diszkrétnek*** nevezünk, ha véges vagy megszámlálhatóan végtelen eseménytéren van definiálva. Jelölje S az eseményteret. Ekkor minden A esemény esetén

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\}\,,$$

hiszen az elemei események, így az A-beliek is, egymást páronként kizáróak. Ha S véges halmaz, és minden $s \in S$ elemi esemény valószínűsége

$$\Pr\{s\} = \frac{1}{|S|}\,,$$

akkor a ***diszkrét egyenletes valószínűségeloszláshoz*** (röviden: egyenletes eloszláshoz) jutunk S-en. Ebben az esetben a megfelelő kísérletet gyakran úgy fogjuk leírni, hogy „válasszunk véletlenszerűen egy elemet az S halmazból”.

Példaként tekintsük egy ***szabályos érmével*** való dobások sorozatát, ahol annak a valószínűsége, hogy fejet kapunk, megegyezik azzal, hogy írást, azaz $1/2$. Ha az érmét n-szer feldobjuk, akkor a 2^n számosságú $S = \{\text{F,I}\}^n$ eseménytéren definiált egyenletes eloszlást kapjuk. S minden egyes elemi eseményét tekinthetjük úgy, mint az $\{\text{F,I}\}^n$ halmaz n hosszúságú sztringjét, és mindegyik elemi esemény bekövetkezésének a valószínűsége $1/2^n$. Az

$$A = \{\text{pontosan } k \text{ fejet és } n - k \text{ írást dobunk}\}$$

esemény az S egy $|A| = \binom{n}{k}$ számosságú részhalmaza, hiszen $\binom{n}{k}$ számú olyan n-hosszú sztringe van a {F,I} halmaznak, amely pontosan k számú F-et tartalmaz. Az A esemény valószínűsége így $\Pr\{A\} = \binom{n}{k}/2^n$.

Az egyenletes eloszlás

Az egyenletes eloszlás olyan példa valószínűségeloszlásra, amelynél nem lehet az eseménytér összes részhalmaza esemény. Az egyenletes valószínűségeloszlást a valós számok egy $[a, b]$ zárt részintervallumán definiáljuk, ahol $a < b$. Heurisztikusan azt szeretnénk elérni, hogy az $[a, b]$ intervallum minden pontja „egyenlően valószínű" legyen. Azonban mivel nem megszámlálhatóan sok pontunk van, az összes pontnak ugyanakkora véges, pozitív valószínűséget adva, nem tehetünk egyszerre eleget a 2. és a 3. axiómának. Emiatt az S csak néhány olyan részhalmazához kívánunk valószínűséget hozzárendelni, amely események már eleget tesznek az axiómáknak.

Minden olyan $[c, d]$ zárt intervallum esetén, melyre $a \le c < d \le b$, az ***egyenletes valószínűségeloszlás*** a $[c, d]$ esemény valószínűségét a

$$\Pr\{[c, d]\} = \frac{d - c}{b - a}$$

módon definiálja. Megjegyezzük, hogy bármely x pont esetén $x = [x, x]$, így az x pont valószínűsége 0. Ha elhagyjuk a $[c, d]$ intervallum végpontjait, akkor a (c, d) nyílt intervallumot kapjuk. Mivel $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, a 3. axióma alapján $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. Általában, az egyenletes valószínűségeloszlás esetén az események halmaza megegyezik az $[a, b]$ eseménytér olyan részhalmazaival, amelyek előállíthatók nyílt és zárt intervallumok segítségével.

Feltételes valószínűség és függetlenség

Bizonyos esetekben rendelkezünk valamilyen részleges, előzetes ismerettel egy kísérlet kimeneteléről. Tegyük fel például, hogy egy barátunk feldobott két szabályos érmét, és megmondta, hogy az érmék közül legalább az egyik fej lett. Mennyi annak a valószínűsége, hogy mindkét érme fejet mutat? A kapott információ kizárja a két írás lehetőségét. Mivel a fennmaradó három elemi esemény egyenlően valószínű, arra következtethetünk, hogy mindegyik 1/3 valószínűséggel fordulhat elő. Mivel ezek közül az elemi események közül csak az egyik szolgáltat két fejet, a kérdésünkre adott válasz: 1/3.

A feltételes valószínűség bevezetésével formalizálhatjuk egy kísérlet kimenetelével kapcsolatos részleges, előzetes ismeretet. Az A eseménynek a B eseményre vonatkozó ***feltételes valószínűségét*** a

$$\Pr\{A|B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \tag{C.14}$$

képlettel definiáljuk, feltéve, hogy $\Pr\{B\} \neq 0$. ($\Pr\{A|B\}$-t úgy olvassuk, hogy „az A esemény valószínűsége a B feltétel mellett.") Mivel a B esemény bekövetkezett, ezért $A \cap B$ az az esemény, hogy az A esemény is bekövetkezik. Azaz, $A \cap B$ azoknak a kimeneteleknek a halmaza, amelyeknél A is és B is bekövetkezik. Mivel a kimenetelek csak B-beli elemi események lehetnek, az összes B-beli elemi esemény valószínűségét lenormálhatjuk $\Pr\{B\}$-vel osztva úgy, hogy az összegük 1 legyen. Az A feltételes valószínűsége a B feltétel mellett

ezért az $A \cap B$ esemény és a B esemény valószínűségeinek a hányadosa. A fenti példában az A esemény az, hogy mindkét érme fej, a B esemény pedig az, hogy legalább az egyik érme fej. Így $\Pr\{A|B\} = (1/4)/(3/4) = 1/3$.

Két eseményt ***függetlennek*** nevezünk, ha

$$\Pr\{A \cap B\} = \Pr\{A\} \Pr\{B\}, \tag{C.15}$$

amely – ha feltesszük, hogy $\Pr\{B\} \neq 0$ – ekvivalens azzal, hogy

$$\Pr\{A|B\} = \Pr\{A\}.$$

Tegyük fel például, hogy feldobtunk két szabályos érmét és a kimenetelek függetlenek egymástól. Ekkor a két fej valószínűsége $(1/2)(1/2) = 1/4$. Tegyük most fel, hogy az egyik esemény az, hogy az első érmén fejet kapunk, a másik pedig az, hogy a két érmével eltérőt dobunk. Mindegyik esemény 1/2 valószínűséggel következik be, és 1/4 annak a valószínűsége, hogy mindkét esemény egyszerre bekövetkezik; így a függetlenség definíciója szerint a két esemény független, habár azt gondolnánk, hogy mindkét esemény függ az első érmétől. Végezetül tegyük fel, hogy az érméket egymáshoz forrasztottuk úgy, hogy egyszerre esnek a fej vagy az írás oldalukra, és ez a két lehetőség egyenlően valószínű. Ekkor 1/2 annak a valószínűsége, hogy az érmékkel egyenként fejet dobunk, azonban annak a valószínűsége, hogy mindkét érmével egyszerre dobunk fejet $1/2 \neq (1/2)(1/2)$. Következésképp az az esemény, hogy az egyikkel fejet dobunk, és az az esemény, hogy a másikkal fejet dobunk, nem független egymástól.

Az $A_1, A_2, \ldots, A_n$ események összességét ***páronként függetlennek*** nevezzük, ha minden $1 \leq i < j \leq n$ esetén

$$\Pr\left\{A_i \cap A_j\right\} = \Pr\{A_i\} \Pr\left\{A_j\right\}.$$

Azt mondjuk, hogy ezek az események ***(teljesen) függetlenek,*** ha bármely k elemű $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ részhalmazukra $2 \leq k \leq n$ és $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ esetén fennáll a

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\} \Pr\{A_{i_2}\} \cdots \Pr\{A_{i_k}\}$$

egyenlőség. Dobjunk fel például két szabályos érmét. Legyen A_1 az az esemény, hogy az első érmével fejet dobunk, A_2 az az esemény, hogy a második érmével fejet dobunk, és A_3 az az esemény, hogy a két érmével különbözőt dobunk. Azt kapjuk, hogy

$$\begin{aligned}
\Pr\{A_1\} &= \frac{1}{2}, \\
\Pr\{A_2\} &= \frac{1}{2}, \\
\Pr\{A_3\} &= \frac{1}{2}, \\
\Pr\{A_1 \cap A_2\} &= \frac{1}{4}, \\
\Pr\{A_1 \cap A_3\} &= \frac{1}{4}, \\
\Pr\{A_2 \cap A_3\} &= \frac{1}{4}, \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0.
\end{aligned}$$

Mivel minden $1 \le i < j \le 3$ esetén $\Pr\left\{A_i \cap A_j\right\} = \Pr\{A_i\}\Pr\left\{A_j\right\} = 1/4$, az A_1, A_2 és A_3 események páronként függetlenek. Azonban ezek az események nem teljesen függetlenek, hiszen $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ és $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$.

A Bayes-tétel

A feltételes valószínűség (C.14) definíciójából és az $A \cap B = B \cap A$ kommutativitási szabályból következik, hogy két nem nulla valószínűségű A és B esemény esetén

$$\begin{aligned} \Pr\{A \cap B\} &= \Pr\{B\}\Pr\{A|B\} \\ &= \Pr\{A\}\Pr\{B|A\}\,. \end{aligned} \tag{C.16}$$

Ezt $\Pr\{A|B\}$-re megoldva kapjuk a

$$\Pr\{A|B\} = \frac{\Pr\{A\}\Pr\{B|A\}}{\Pr\{B\}} \tag{C.17}$$

Bayes-tételt. A $\Pr\{B\}$ nevező egy normáló konstans, amelyet helyettesíthetünk a következővel. Mivel $B = (B \cap A) \cup (B \cap \overline{A})$, és $B \cap A$ és $B \cap \overline{A}$ egymást kizáró események, ezért

$$\begin{aligned} \Pr\{B\} &= \Pr\{B \cap A\} + \Pr\left\{B \cap \overline{A}\right\} \\ &= \Pr\{A\}\Pr\{B|A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B|\overline{A}\right\}. \end{aligned}$$

A (C.17) egyenletbe behelyettesítve a Bayes-tétel következő ekvivalens alakját kapjuk:

$$\Pr\{A|B\} = \frac{\Pr\{A\}\Pr\{B|A\}}{\Pr\{A\}\Pr\{B|A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B|\overline{A}\right\}}\,.$$

A Bayes-tétel egyszerűsítheti a feltételes valószínűség kiszámítását. Tegyük fel például, hogy adott egy szabályos és egy olyan szabálytalan érme, amellyel mindig fejet dobunk. Hajtsunk végre egy három független eseményből álló kísérletet: először válasszunk ki véletlenszerűen egy érmét, ezután a kiválasztott érmét dobjuk fel egyszer, majd még egyszer. Tegyük fel, hogy a kiválasztott érmével mindkét alkalommal fejet dobtunk. Mennyi a valószínűsége annak, hogy az érme szabálytalan volt?

A feladatot a Bayes-tétel segítségével oldjuk meg. Legyen A az az esemény, hogy a szabálytalan érmét választottuk ki, B pedig az az esemény, hogy mindkétszer fejet dobtunk. A $\Pr\{A|B\}$ mennyiséget akarjuk meghatározni. Tudjuk, hogy $\Pr\{A\} = 1/2$, $\Pr\{B|A\} = 1$, $\Pr\left\{\overline{A}\right\} = 1/2$ és $\Pr\left\{B|\overline{A}\right\} = 1/4$, így

$$\begin{aligned} \Pr\{A|B\} &= \frac{1/2 \cdot 1}{1/2 \cdot 1 + 1/2 \cdot 1/4} \\ &= \frac{4}{5}\,. \end{aligned}$$

Gyakorlatok

C.2-1. Bizonyítsuk be a ***Boole-egyenlőtlenséget:*** az $A_1, A_2, \ldots$ események bármely véges vagy megszámlálhatóan végtelen sorozata esetén

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \le \Pr\{A_1\} + \Pr\{A_2\} + \cdots\,. \tag{C.18}$$

C.2-2. Rosencrantz professzor feldob egy szabályos érmét. Guildenstern professzor feldob két szabályos érmét. Mennyi a valószínűsége annak, hogy Rosencrantz professzor több fejet dob, mint Guildenstern professzor?

C.2-3. Alaposan megkeverünk egy 10 lapból álló kártyapaklit, amelyben a lapok különböző, 1 és 10 közötti számokkal vannak megjelölve. Ezután három lapot húzunk egymás után a pakliból. Mennyi a valószínűsége annak, hogy a három lapot nagyság szerint növekvő sorrendben húzzuk ki?

C.2-4. ⋆ Adjunk meg egy olyan eljárást, amely két a és b egész bemenet mellett, ahol $0 < a < b$, egy szabályos érme dobásainak segítségével olyan kimenetet szolgáltat, ahol a fejek valószínűsége a/b, és a írások valószínűsége $(b - a)/b$. Adjunk $O(1)$ rendű korlátot az érmedobások várható számára. (*Útmutatás*. Vegyük a/b bináris el őállítását.)

C.2-5. Bizonyítsuk be, hogy

$$\Pr\{A|B\} + \Pr\left\{\overline{A}|B\right\} = 1.$$

C.2-6. Bizonyítsuk be, hogy az $A_1, A_2, \ldots, A_n$ események bármely családjára

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\} = &\Pr\{A_1\} \cdot \Pr\{A_2|A_1\} \cdot \Pr\{A_3|A_1 \cap A_2\} \cdots \\ &\Pr\{A_n|A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}.\end{aligned}$$

C.2-7. ⋆ Hogyan adhatunk meg egy olyan n eseményb ől álló halmazt, amelynek elemei páronként függetlenek, azonban minden $k > 2$ elemű részhalmaza nem teljesen független?

C.2-8. ⋆ Az A és B eseményeket ***feltételesen függetlennek*** nevezzük a C feltétel mellett, ha

$$\Pr\{A \cap B|C\} = \Pr\{A|C\} \cdot \Pr\{B|C\}.$$

Adjunk egyszerű, de nem triviális példát két olyan eseményre, amelyek nem függetlenek, azonban feltételesen függetlenek egy harmadik esemény mellett.

C.2-9. ⋆ Egy olyan szerencsejáték résztvev ői vagyunk, amelyben a díjat három függöny valamelyike mögé rejtették. A díjat megnyerjük, amennyiben a helyes függönyt választjuk ki. Miután már kiválasztottunk egy függönyt, de még miel őtt az felemelkedne, a műsorvezet ő felemel egy másik függönyt, megmutatván egy üres színpadot, majd megkérdezi, hogy nem kívánjuk-e a harmadik függönyre módosítani a jelenlegi választásunkat. Hogyan változnak meg az esélyeink, ha módosítjuk a választásunkat?

C.2-10. ⋆ Egy börtönigazgató három rab közül véletlenszerűen kiválaszt egyet, akit majd szabadon enged. A másik kettőt kivégzik. Az őr tudja, melyikük lesz szabad, azonban tilos bármilyen, a helyzetüket érint ő információt közölni a rabokkal. Jelöljük a rabokat az X, Y és Z betűkkel. Az X rab bizalmasan megkérdi az őrtől, hogy X és Y közül kit fognak kivégezni, azt gondolván, hogy közülük egynek meg kell halnia, és az őr úgysem adhat információt a saját helyzetéről. Az őr azt feleli X-nek, hogy Y-t fogják kivégezni. Az X rab ennek megörül, úgy számolván, hogy vagy ő, vagy a Z rab szabad lesz, ami azt jelenti, hogy annak a valószínűsége, hogy szabad lesz, most már 1/2. Igaza van-e, vagy az esélye még mindig 1/3? Indokoljuk meg állításunkat.

C.3. Diszkrét valószínűségi változók

Az X ***(diszkrét) valószínűségi változón*** egy olyan függvényt értünk, amely az S véges vagy megszámlálhatóan végtelen eseményteret a valós számok halmazába képezi le. A valószínűségi változó egy kísérlet összes lehetséges kimeneteléhez egy-egy valós számot rendel, ily módon elegendő az így kapott számhalmazon indukált valószínűségeloszlással dolgozni. Valószínűségi változókat definiálhatunk nem megszámlálhatóan végtelen eseménytéren is, azonban ekkor olyan technikai kérdések merülnek fel, amelyek a céljaink szempontjából fölöslegessé teszik, hogy beszéljünk róluk. Ezért a továbbiakban feltételezzük, hogy a valószínűségi változók mindig diszkrétek.

Egy X valószínűségi változó és egy x valós szám esetén az $X = x$ eseményt az $\{s \in S : X(s) = x\}$ módon definiáljuk; így ennek valószínűsége

$$\Pr\{X = x\} = \sum_{s \in S : X(s) = x} \Pr\{s\} .$$

Az

$$f(x) = \Pr\{X = x\}$$

függvényt az X valószínűségi változó ***sűrűségfüggvényének*** nevezzük. A valószínűség axiómáiból következik, hogy $\Pr\{X = x\} \geq 0$ és $\sum_x \Pr\{X = x\} = 1$.

Példaként tekintsük azt a kísérletet, amely során egy pár szabályos, 6 oldalú kockával dobunk. Az eseménytér ekkor 36 lehetséges elemi eseményből áll. Tegyük fel, hogy a valószínűségeloszlás egyenletes, így minden $s \in S$ elemi esemény egyenlően valószínű: $\Pr\{s\} = 1/36$. Definiáljuk az X valószínűségi változót úgy, mint a kockákkal dobott két érték maximumát. Ekkor azt kapjuk, hogy $\Pr\{X = 3\} = 5/36$, hiszen X a 36 lehetséges elemi esemény közül 5 esetén vesz fel 3-t, nevezetesen az $(1, 3)$, $(2, 3)$, $(3, 3)$, $(3, 2)$ és $(3, 1)$ esetekben.

Gyakran előfordul, hogy ugyanazon az eseménytéren több valószínűségi változót definiálunk. Ha X és Y valószínűségi változók, akkor az

$$f(x, y) = \Pr\{X = x \text{ és } Y = y\}$$

függvényt X és Y ***együttes sűrűségfüggvényének*** nevezzük. Rögzített y esetén

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ és } Y = y\} ,$$

és hasonlóan, rögzített x esetén

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ és } Y = y\} .$$

A feltételes valószínűség (C.14) definícióját felhasználva azt kapjuk, hogy

$$\Pr\{X = x | Y = y\} = \frac{\Pr\{X = x \text{ és } Y = y\}}{\Pr\{Y = y\}} .$$

Az X és Y valószínűségi változókat ***függetleneknek*** nevezzük, ha bármely x és y esetén az $X = x$ és $Y = y$ események függetlenek, vagy ekvivalens módon ha $\Pr\{X = x \text{ és } Y = y\} = \Pr\{X = x\} \Pr\{Y = y\}$ minden x és y esetén.

Ugyanazon az eseménytéren definiált valószínűségi változók egy adott halmaza esetén új valószínűségi változókat definiálhatunk: véve az eredeti változók összegét, szorzatát vagy más függvényét.

Valószínűségi változók várható értéke

Egy valószínűségi változó eloszlásának egyik legegyszerűbb és leghasznosabb jellemzése az általa felvett értékek „átlaga”. Az X diszkrét valószínűségi változó ***várható értékén*** (vagy másképpen ***átlagán***) az

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\} \tag{C.19}$$

mennyiséget értjük, amely jól definiált, ha az összeg véges vagy abszolút konvergens. Néha X várható értékét μ_X-szel, vagy ha a szövegkörnyezet alapján a valószínűségi változó nyilvánvaló, egyszerűen μ-vel fogjuk jelölni.

Tekintsük azt a játékot, amely során feldobunk két szabályos érmét. Nyerjünk 3 dollárt minden egyes fej, és veszítsünk 2 dollárt minden egyes írás esetén. Ekkor a nyereségünk nagyságát leíró valószínűségi változó várható értéke

$$\begin{aligned}\mathrm{E}[X] &= 6 \cdot \Pr\{2\mathrm{F}\} + 1 \cdot \Pr\{1\mathrm{F},\ 1\mathrm{I}\} - 4 \cdot \Pr\{2\mathrm{I}\} \\ &= 6 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} - 4 \cdot \frac{1}{4} \\ &= 1.\end{aligned}$$

Két valószínűségi változó összegének a várható értéke megegyezik várható értékeiknek az összegével, azaz

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y], \tag{C.20}$$

amennyiben $\mathrm{E}[X]$ és $\mathrm{E}[Y]$ létezik. Ezt a tulajdonságot a várható érték ***linearitásának*** nevezzük, amely akkor is fennáll, ha X és Y nem független. Ez a tulajdonság átvihető véges vagy abszolút konvergens összegek várható értékére is. A várható érték linearitása az az alapvető tulajdonság, amely képessé tesz bennünket arra, hogy valószínűségi elemzést végezzünk indikátor valószínűségi változók segítségével (lásd az 5.2. alfejezetet).

Ha X tetszőleges valószínűségi változó, akkor bármely $g(x)$ függvény segítségével egy $g(X)$ új valószínűségi változót definiálhatunk. Ha létezik a $g(X)$ várható értéke, akkor

$$\mathrm{E}[g(X)] = \sum_x g(x) \Pr\{X = x\}.$$

A $g(x) = ax$ függvényt választva azt kapjuk, hogy

$$\mathrm{E}[aX] = a\mathrm{E}[X] \tag{C.21}$$

bármely a konstans esetén. Következésképp a várható érték képzése lineáris: bármely két X és Y valószínűségi változó és a konstans esetén

$$\mathrm{E}[aX + Y] = a\mathrm{E}[X] + \mathrm{E}[Y]. \tag{C.22}$$

Amennyiben az X és Y valószínűségi változók függetlenek és mindegyiknek létezik a várható értéke, akkor

$$\begin{aligned}
\mathrm{E}[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ és } Y = y\} \\
&= \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} \\
&= \left(\sum_x x \Pr\{X = x\}\right)\left(\sum_y y \Pr\{Y = y\}\right) \\
&= \mathrm{E}[X]\,\mathrm{E}[Y]\,.
\end{aligned}$$

Általában n teljesen független $X_1, X_2, \ldots, X_n$ valószínűségi változó esetén

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\,\mathrm{E}[X_2] \cdots \mathrm{E}[X_n]\,. \tag{C.23}$$

Abban az esetben, ha az X valószínűségi változó az $N = \{0, 1, 2, \ldots\}$ természetes számok közül veszi fel az értékeit, egy elegáns kifejezés adható a várható értékére

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} \\
&= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i + 1\}) \\
&= \sum_{i=1}^{\infty} \Pr\{X \geq i\}\,,
\end{aligned} \tag{C.24}$$

hiszen mindegyik $\Pr\{X \geq i\}$ tagot i-szer adunk össze és $(i - 1)$-szer vonunk le (kivéve a $\Pr\{X \geq 0\}$ tagot, amelyet 0-szor adunk össze és egyáltalán nem vonunk le).

Amikor egy $f(x)$ konvex függvényt alkalmazunk egy X valószínűségi változóra, a ***Jensen-egyenlőtlenség*** alapján

$$\mathrm{E}[f(X)] \geq f(\mathrm{E}[X]), \tag{C.25}$$

amennyiben a várható értékek léteznek és végesek. (Egy $f(x)$ függvény konvex, ha minden x-re és y-ra bármely $0 \leq \lambda \leq 1$ esetén fennáll, hogy $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y)$.)

A szórásnégyzet és a szórás

Egy valószínűségi változó várható értéke nem mondja meg, hogy mennyire „szóródnak” a változó értékei. Például, ha az X és Y valószínűségi változók olyanok, hogy $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ és $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$, akkor $\mathrm{E}[X]$ és $\mathrm{E}[Y]$ egyaránt $1/2$-del egyenlő, míg az Y által felvett értékek távolabb esnek az átlagtól, mint az X által felvettek.

A szórásnégyzet fogalma matematikailag fejezi ki azt, hogy egy valószínűségi változó értékei átlagosan milyen messze vannak a várható értéktől. Egy $\mathrm{E}[X]$ várható értékű X valószínűségi változó ***szórásnégyzetén*** a

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] \\
&= \mathrm{E}\left[X^2 - 2X\mathrm{E}[X] + \mathrm{E}^2[X]\right]
\end{aligned}$$

$$\begin{aligned} &= \mathrm{E}\left[X^2\right] - 2\mathrm{E}\left[X\mathrm{E}\left[X\right]\right] + \mathrm{E}^2\left[X\right] \\ &= \mathrm{E}\left[X^2\right] - 2\mathrm{E}^2\left[X\right] + \mathrm{E}^2\left[X\right] \\ &= \mathrm{E}\left[X^2\right] - \mathrm{E}^2\left[X\right] \end{aligned} \tag{C.26}$$

mennyiséget értjük. Az $\mathrm{E}\left[\mathrm{E}^2\left[X\right]\right] = \mathrm{E}^2\left[X\right]$ és $\mathrm{E}\left[X\mathrm{E}\left[X\right]\right] = \mathrm{E}^2\left[X\right]$ egyenlőségek igazolásához alkalmaztuk a (C.21) azonosságot ($a = \mathrm{E}\left[X\right]$ választással). A (C.26) egyenletet átrendezve egy valószínűségi változó négyzetének a várható értékére a következő képletet kapjuk:

$$\mathrm{E}\left[X^2\right] = \mathrm{Var}\left[X\right] + \mathrm{E}^2\left[X\right]. \tag{C.27}$$

Egy X valószínűségi változó szórásnégyzete és aX szórásnégyzete között az alábbi összefüggés áll fenn (lásd C.3-10. gyakorlat):

$$\mathrm{Var}\left[aX\right] = a^2\mathrm{Var}\left[X\right].$$

Amennyiben X és Y független valószínűségi változók, úgy

$$\mathrm{Var}\left[X + Y\right] = \mathrm{Var}\left[X\right] + \mathrm{Var}\left[Y\right].$$

Általában, ha az $X_1, X_2, \ldots, X_n$ n számú valószínűségi változó páronként független, akkor

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}\left[X_i\right]. \tag{C.28}$$

Egy X valószínűségi változó ***szórásán*** X szórásnégyzetének a pozitív négyzetgyökét értjük. Néha egy X valószínűségi változó szórását σ_X-szel, vagy ha a szövegkörnyezet alapján az X valószínűségi változó egyértelmű, egyszerűen σ-val fogjuk jelölni. Ennek megfelelően az X szórásnégyzetét σ^2-tel jelöljük.

Gyakorlatok

C.3-1. Dobjunk két szabályos, 6 oldalú kockával. Mennyi a várható értéke a dobott számok összegének? Mennyi a várható értéke a dobott számok maximumának?

C.3-2. Az $A[1\,..\,n]$ tömb n darab olyan különböző számot tartalmaz, amelynek sorrendje véletlenszerű úgy, hogy az n szám mindegyik permutációja egyenlően valószínű. Mennyi a tömb legnagyobb eleme indexének a várható értéke? Mennyi a tömb legkisebb eleme indexének a várható értéke?

C.3-3. Egy farsangi játékot egy dobozzal és a benne lévő három kockával játszanak. A játékos 1 dollárt tehet fel az 1-től 6-ig terjedő számok bármelyikére. A dobozt megrázzák, amely után a kifizetés az alábbi módon történik. Ha a játékos száma egyetlen kockán sem jelent meg, akkor elveszíti a dollárját. Egyébként, ha a szám a három kocka közül pontosan k-n jelent meg, ahol $k = 1, 2, 3$, a játékos visszakapja a dollárját, és nyer még k dollárt. Mennyi a játékos várható nyeresége a farsangi játékban?

C.3-4. Lássuk be, hogy ha X és Y nemnegatív valószínűségi változók, akkor

$$\mathrm{E}\left[\max(X, Y)\right] \leq \mathrm{E}\left[X\right] + \mathrm{E}\left[Y\right].$$

C.3-5.⋆ Legyenek X és Y független valószínűségi változók. Bizonyítsuk be, hogy $f(X)$ és $g(Y)$ függetlenek, bárhogy is válasszuk az f és g függvényeket.
C.3-6.⋆ Legyen X egy nemnegatív valószínűségi változó, és tegyük fel, hogy $\mathrm{E}[X]$ létezik. Bizonyítsuk be a ***Markov-egyenlőtlenséget:***

$$\Pr\{X \geq t\} \leq \frac{\mathrm{E}[X]}{t} \tag{C.29}$$

minden $t > 0$ esetén.
C.3-7.⋆ Legyen S egy eseménytér, és X, illetve X' olyan valószínűségi változók, amelyekre $X(s) \geq X'(s)$ minden $s \in S$ esetén. Bizonyítsuk be, hogy bármely valós t konstans esetén

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\}.$$

C.3-8. Mi a nagyobb: egy valószínűségi változó négyzetének a várható értéke, vagy pedig a várható értékének a négyzete?
C.3-9. Mutassuk meg, hogy bármely olyan X valószínűségi változóra, amely csak a 0 és 1 értékeket veszi fel, fennáll, hogy $\mathrm{Var}[X] = \mathrm{E}[X]\,\mathrm{E}[1 - X]$.
C.3-10. A szórásnégyzet (C.26) definíciója alapján bizonyítsuk be, hogy $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$.

C.4. A geometriai és a binomiális eloszlás

Az érmedobás az ún. ***Bernoulli-kísérlet*** egy speciális esete. Ebben a kísérletben két kimenetel lehetséges: a ***jó eset,*** amely p valószínűséggel következik be, és a ***rossz eset,*** amely $q = 1 - p$ valószínűséggel következik be. Amikor összefoglalóan ***Bernoulli-kísérletsorozatról*** beszélünk, akkor azon olyan teljesen független kísérleteket értünk, amelyeknél, ha mást nem mondunk, a jó esetnek mindig ugyanannyi a valószínűsége. A Bernoulli-kísérletsorozat kapcsán két fontos eloszlás lép fel: a geometriai és a binomiális eloszlás.

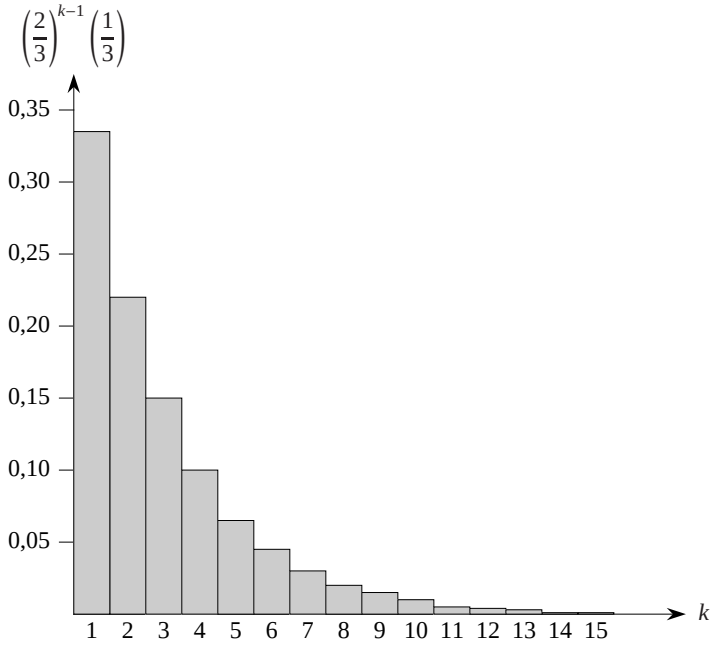
A geometriai eloszlás

Tegyük fel, hogy adott egy olyan Bernoulli-kísérletsorozat, amelynél a jó eset valószínűsége mindig p, a rossz eset valószínűsége pedig $q = 1 - p$. Hány kísérletet kell végrehajtanunk, amíg egy jó eset be nem következik? Legyen az X valószínűségi változó az egy jó eset bekövetkezéséhez szükséges kísérletek száma. Ekkor X az $\{1, 2, \ldots\}$ értékeket veszi fel, és bármely $k \geq 1$ esetén

$$\Pr\{X = k\} = q^{k-1}p, \tag{C.30}$$

ugyanis $k - 1$-számú rossz eset fordul elő, amíg egy jó eset be nem következik. A (C.30) egyenletnek eleget tevő valószínűségeloszlást ***geometriai eloszlásnak*** nevezzük. Egy ilyen eloszlást ábrázoltunk a C.1. ábrán.

Feltéve, hogy $q < 1$, az (A.8) azonosság segítségével kiszámolhatjuk a geometriai eloszlás várható értékét



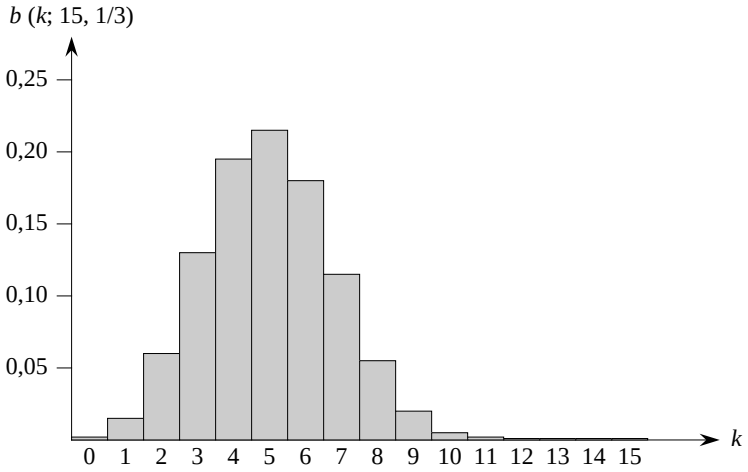
C.1. ábra. Egy olyan geometriai eloszlás, amelynél a jó eset valószínűsége $p = 1/3$, a rossz eset valószínűsége pedig $q = 1 - p$. Az eloszlás várható értéke $1/p = 3$.

$$\begin{aligned} \mathrm{E}[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=0}^{\infty} kq^k \\ &= \frac{p}{q}\cdot\frac{q}{(1-q)^2} \\ &= \frac{1}{p}\,. \end{aligned} \tag{C.31}$$

Így átlagosan $1/p$ számú kísérletet kell végrehajtanunk ahhoz, hogy egy jó eset bekövetkezzék, ami intuitívan is adódik. A szórásnégyzet, amelyet az A.1-3. gyakorlatot felhasználva hasonlóan számolhatunk ki,

$$\mathrm{Var}[X] = \frac{q}{p^2}\,. \tag{C.32}$$

Példaként tegyük fel, hogy addig dobunk többször egymás után két kockával, amíg a dobott számok összegeként hetet vagy tizenegyet nem kapunk. A 36 lehetséges kimenetel közül 6 esetben kapunk hetet és 2 esetben tizenegyet. Így a jó eset valószínűsége $p = 8/36 = 2/9$, és átlagosan $1/p = 9/2 = 4,5$ alkalommal kell dobnunk, hogy hetet vagy tizenegyet kapjunk összegként.



C.2. ábra. A $b(k; 15, 1/3)$ binomiális eloszlás, amely $n = 15$ számú olyan Bernoulli-kísérletből származik, ahol a jó eset valószínűsége mindig $p = 1/3$. Az eloszlás várható értéke $np = 5$.

A binomiális eloszlás

Hányszor következik be a jó eset n számú olyan Bernoulli-kísérlet során, ahol a jó eset valószínűsége p, a rossz eset valószínűsége pedig $q = 1 - p$. Legyen az X valószínűségi változó a jó esetek száma az n kísérlet során. Ekkor X a $\{0, 1, \ldots, n\}$ halmazból veszi fel az értékeit, és minden $k = 0, \ldots, n$ esetén

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \tag{C.33}$$

hiszen $\binom{n}{k}$-féleképpen választhatjuk ki a k számú jó esetet az n kísérletből, és mindegyik sorozat bekövetkezésének a valószínűsége $p^k q^{n-k}$. A (C.33) egyenletnek eleget tevő valószínűségeloszlást ***binomiális eloszlásnak*** nevezzük. A kényelem kedvéért a binomiális eloszlások családjára a

$$b(k; n, p) = \binom{n}{k} p^k (1 - p)^{n-k} \tag{C.34}$$

jelölést fogjuk használni. A C.2. ábrán egy binomiális eloszlást ábrázoltunk. A „binomiális” elnevezés abból ered, hogy a (C.33) kifejezés a $(p + q)^n$ kifejtésének a k-adik tagja. Következésképp

$$\sum_{k=0}^{n} b(k; n, p) = 1, \tag{C.35}$$

hiszen $p + q = 1$, ami éppen a 2. valószínűségi axióma által megkövetelt összefüggés.

A (C.8) és (C.35) egyenletek segítségével kiszámolhatjuk egy binomiális eloszlású valószínűségi változó várható értékét. Legyen X egy $b(k; n, p)$ binomiális eloszlású valószínűségi változó, és legyen $q = 1 - p$. A várható érték definíciója alapján azt kapjuk, hogy

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{n} k\Pr\{X=k\} \\
&= \sum_{k=0}^{n} k\, b(k;n,p) \\
&= \sum_{k=1}^{n} k\binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k;n-1,p) \\
&= np. \qquad \text{(C.36)}
\end{aligned}$$

A várható érték linearitását felhasználva ugyanezt az eredményt lényegesen kevesebb számolással is megkaphatjuk. Legyen az X_i valószínűségi változó a jó esetek száma az i-edik kísérletben. Ekkor $\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$, és a várható érték linearitása alapján (lásd (C.20)) a jó esetek várható száma n kísérlet során

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np. \qquad \text{(C.37)}
\end{aligned}$$

Ugyanezt a módszert alkalmazhatjuk az eloszlás szórásnégyzetének a kiszámolásánál is. A (C.26) egyenlet alapján $\mathrm{Var}[X_i] = \mathrm{E}\left[X_i^2\right] - \mathrm{E}[X_i]$. Mivel X_i csak a 0 és az 1 értékeket veszi fel, ezért $\mathrm{E}\left[X_i^2\right] = \mathrm{E}[X_i] = p$, és így

$$\mathrm{Var}[X_i] = p - p^2 = pq. \qquad \text{(C.38)}$$

Az X szórásnégyzetének kiszámolásához ezután kihasználjuk az n kísérlet függetlenségét; így a (C.28) összefüggés alapján

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{Var}[X_i]
\end{aligned}$$

$$= \sum_{i=1}^{n} pq$$
$$= npq. \tag{C.39}$$

Ahogy a C.2. ábrán látható, a $b(k; n, p)$ binomiális eloszlás nő, amíg a k 0-tól n-ig futva el nem éri az np átlagot, majd pedig csökken. Két egymás utáni tag hányadosát tekintve belátjuk, hogy az eloszlás mindig így viselkedik:

$$\begin{aligned}\frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} \\ &= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\ &= \frac{(n-k+1)p}{kq} \\ &= 1 + \frac{(n+1)p-k}{kq}\,.\end{aligned} \tag{C.40}$$

Ez a hányados pontosan akkor nagyobb mint 1, ha $(n+1)p - k$ pozitív. Következésképp $b(k; n, p) > b(k-1; n, p)$ (az eloszlás nő), ha $k < (n+1)p$, és $b(k; n, p) < b(k-1; n, p)$ (az eloszlás csökken), ha $k > (n+1)p$. Ha $k = (n+1)p$ egész, akkor $b(k; n, p) = b(k-1; n, p)$, így az eloszlásnak két maximuma van: $k = (n+1)p$-ben és $k-1 = (n+1)p-1 = np-q$-ban. Egyébként pedig a maximumát arra az egyértelmű k értékre veszi fel, amely az $np - q < k < (n+1)p$ tartományba esik.

A következő lemmában felső korlátot adunk a binomiális eloszlásra.

C.1. lemma. *Legyen $n \geq 0$, $0 < p < 1$, $q = 1 - p$ és $0 \leq k \leq n$. Ekkor*

$$b(k; n, p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

Bizonyítás. A (C.6) egyenlet alapján kapjuk, hogy

$$\begin{aligned} b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \\ &\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}. \end{aligned}$$

■

Gyakorlatok

C.4-1. Ellenőrizzük a valószínűségi axiómák közül a másodikat a geometriai eloszlásra.

C.4-2. Hányszor kell átlagosan feldobni 6 szabályos érmét, hogy 3 fejet és 3 írást kapjunk?

C.4-3. Mutassuk meg, hogy $b(k; n, p) = b(n-k; n, q)$, ahol $q = 1 - p$.

C.4-4. Mutassuk meg, hogy közelítőleg $1/\sqrt{2\pi npq}$ – ahol $q = 1 - p$ – a $b(k; n, p)$ binomiális eloszlás maximuma.

C.4-5. ⋆ Mutassuk meg, hogy körülbelül $1/e$ a valószínűsége annak, hogy nem következik be a jó eset n számú olyan Bernoulli-kísérletnél, ahol a jó eset valószínűsége mindig $p = 1/n$. Mutassuk meg, hogy egy jó eset bekövetkezésének a valószínűsége szintén körülbelül $1/e$.

C.4-6. ⋆ Rosencrantz és Guildenstern professzorok egyenként n-szer feldobnak egy szabályos érmét. Mutassuk meg, hogy $\binom{2n}{n}/4^n$ a valószínűsége annak, hogy ugyanannyi számú fejet dobnak. (*Útmutatás.* Nevezzük el a fejet jó esetnek Rosencrantz professzornál, az írást pedig rossz esetnek Guildenstern professzornál.) Ennek a gondolatmenetnek a segítségével igazoljuk a

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}$$

azonosságot.

C.4-7. ⋆ Mutassuk meg, hogy minden $0 \le k \le n$ esetén

$$b(k; n, p) \le 2^{nH(k/n)-n},$$

ahol $H(x)$ a (C.7) entrópia függvény.

C.4-8. ⋆ Tekintsünk n számú olyan Bernoulli-kísérletet, ahol a jó eset valószínűsége az i-edik kísérletnél p_i, $i = 1, 2, \ldots, n$, és jelölje az X valószínűségi változó a jó esetek számát. Legyen $p \ge p_i$ minden $i = 1, 2, \ldots, n$ esetén. Bizonyítsuk be, hogy minden $1 \le k \le n$ esetén

$$\Pr\{X < k\} \ge \sum_{i=0}^{k-1} b(i; n, p).$$

C.4-9. ⋆ Legyen az X valószínűségi változó a jó esetek száma n számú Bernoulli-kísérletnek egy olyan A halmazában, ahol a jó eset valószínűsége az i-edik kísérletben p_i, és legyen az X' valószínűségi változó a jó esetek száma n számú Bernoulli-kísérletnek egy olyan A' másik halmazában, ahol a jó eset valószínűsége az i-edik kísérletben $p'_i \ge p_i$. Bizonyítsuk be, hogy minden $0 \le k \le n$ esetén

$$\Pr\{X' \ge k\} \ge \Pr\{X \ge k\}.$$

(*Útmutatás*. Vizsgáljuk meg, hogyan kaphatjuk meg az A' halmazbeli Bernoulli-kísérleteket olyan kísérletekből, amelyek magukban foglalják az A halmazbelieket, és használjuk a C.3-7. gyakorlat eredményét.)

C.5. A binomiális eloszlás farkai

Gyakran fontosabb számunkra annak az eseménynek a valószínűsége, hogy legalább k vagy legfeljebb k számú jó eset következik be n számú olyan Bernoulli-kísérlet során, ahol a jó eset valószínűsége mindegyikben p, mint azé az eseményé, hogy a jó eset pontosan k-szor következik be. Ebben az alfejezetben a binomiális eloszlás ***farkait,*** azaz a $b(k; n, p)$ eloszlásnak azt a két tartományát fogjuk vizsgálni, amelyek messze vannak az np átlagtól. Számos fontos becslést bizonyítunk be a farkakra (a farkakban lévő tagok összegére) vonatkozóan.

Először a $b(k; n, p)$ eloszlás jobb oldali farkára adunk egy becslést. A bal oldali farokra a jó és a rossz eset szerepének a felcserélésével határozhatunk meg becsléseket.

C.2. tétel. *Tekintsük n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset bekövetkezésének a valószínűsége p. Jelölje az X valószínűségi változó a jó esetek számát. Ekkor minden* $0 \le k \le n$ *esetén annak a valószínűsége, hogy legalább k számú jó eset következik be,*

$$\begin{aligned}\Pr\{X \ge k\} &= \sum_{i=k}^{n} b(i; n, p) \\ &\le \binom{n}{k} p^k .\end{aligned}$$

Bizonyítás. Egy $S \subseteq \{1, \ldots, n\}$ részhalmazra jelölje A_S azt az eseményt, hogy az összes i-edik, $i \in S$, kísérlet sikeres volt. Világos, hogy $\Pr\{A_S\} = p^k$, ha $|S| = k$. Így azt kapjuk, hogy

$$\begin{aligned}\Pr\{X \ge k\} &= \Pr\{\text{létezik } S \subseteq \{1, \ldots, n\} : |S| = k \text{ és } A_S \text{ bekövetkezett}\} \\ &= \Pr\left\{\bigcup_{S \subseteq \{1,\ldots,n\}:|S|=k} A_S\right\} \\ &\le \sum_{S \subseteq \{1,\ldots,n\}:|S|=k} \Pr\{A_S\} \\ &= \binom{n}{k} p^k ,\end{aligned}$$

ahol az egyenlőtlenség a (C.18) Boole-egyenlőtlenségből következik. ■

Az alábbi következményben az eloszlás bal oldali farkára fogalmazzuk újra a tételt. Általában a becsléseknek az egyik oldali farokról a másik oldalira való adaptálását az olvasóra bízzuk.

C.3. következmény. *Tekintsük n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset bekövetkezésének a valószínűsége p. Ha az X valószínűségi változó jelöli a jó esetek számát, akkor bármely* $0 \le k \le n$ *esetén annak a valószínűsége, hogy a jó eset legfeljebb k-szor fordul elő,*

$$\begin{aligned}\Pr\{X \le k\} &= \sum_{i=0}^{k} b(i; n, p) \\ &\le \binom{n}{n-k} (1-p)^{n-k} \\ &= \binom{n}{k} (1-p)^{n-k} .\end{aligned}$$

A következő becslésnél a binomiális eloszlás bal oldali farkával foglalkozunk. Amint azt a tételt követő következmény állítja, az átlagtól távol, a bal oldali farok exponenciálisan csökken.

C.4. tétel. *Tekintsük n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset p, a rossz eset pedig* $q = 1 - p$ *valószínűséggel következik be. Jelölje az X valószínűségi változó*

a jó esetek számát. Ekkor minden $0 < k < np$ *esetén annak a valószínűsége, hogy k-nál kevesebb jó eset következik be,*

$$\begin{aligned}\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i; n, p) \\ &< \frac{kq}{np - k} b(k; n, p).\end{aligned}$$

Bizonyítás. A $\sum_{i=0}^{k-1} b(i; n, p)$ sort egy geometriai sorral becsüljük, felhasználva az A.2. alfejezetben alkalmazott módszert. A (C.40) egyenletből azt kapjuk, hogy minden $i = 1, 2, \ldots, k$ esetén

$$\begin{aligned}\frac{b(i-1; n, p)}{b(i; n, p)} &= \frac{iq}{(n - i + 1)p} \\ &< \frac{iq}{(n - i)p} \\ &\leq \frac{kq}{(n - k)p} .\end{aligned}$$

Az utolsó törtet x-szel jelölve

$$\begin{aligned}x &= \frac{kq}{(n - k)p} \\ &< \frac{kq}{(n - np)p} \\ &= \frac{kq}{nqp} \\ &= \frac{k}{np} \\ &< 1,\end{aligned}$$

akkor minden $0 < i \leq k$ esetén

$$b(i-1; n, p) < x\, b(i; n, p)$$

következik. Ezt az egyenlőtlenséget $k - i$-szer megismételve kapjuk, hogy minden $0 \leq i < k$ esetén

$$b(i; n, p) < x^{k-i} b(k; n, p),$$

és ezért $0 \leq i < k$ esetén

$$\begin{aligned}\sum_{i=0}^{k-1} b(i; n, p) &< \sum_{i=0}^{k-1} x^{k-i} b(k; n, p) \\ &< b(k; n, p) \sum_{i=1}^{\infty} x^i \\ &= \frac{x}{1 - x} b(k; n, p) \\ &= \frac{kq}{np - k} b(k; n, p).\end{aligned}$$

■

C.5. következmény. *Tekintsük n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset p, a rossz eset pedig $q = 1 - p$ valószínűséggel következik be. Ekkor, $0 < k \leq np/2$ esetén, annak a valószínűsége, hogy kevesebb mint k jó eset fordul elő, kisebb, mint $1/2$-szer annak a valószínűsége, hogy kevesebb mint $k + 1$ jó eset fordul elő.*

Bizonyítás. A $k \leq np/2$ feltétel alapján

$$\begin{aligned}\frac{kq}{np-k} &\leq \frac{(np/2)q}{np-(np)/2}\\ &= \frac{(np/2)q}{np/2}\\ &\leq 1,\end{aligned}$$

mivel $q \leq 1$. Jelölje az X valószínűségi változó a jó esetek számát. Ekkor a C.4. tételből következik, hogy

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p).$$

Így azt kapjuk, hogy

$$\begin{aligned}\frac{\Pr\{X < k\}}{\Pr\{X < k+1\}} &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^{k} b(i; n, p)}\\ &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^{k-1} b(i; n, p) + b(k; n, p)}\\ &< \frac{1}{2},\end{aligned}$$

mivel $\sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p)$. ■

A jobb oldali farokra hasonlóan adható becslés. A bizonyítás a C.5-2. gyakorlat témája.

C.6. következmény. *Tekintsük n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset bekövetkezésének a valószínűsége p. Jelölje az X valószínűségi változó a jó esetek számát. Ekkor minden $np < k < n$ esetén annak a valószínűsége, hogy k-nál több jó eset következik be,*

$$\begin{aligned}\Pr\{X > k\} &= \sum_{i=k+1}^{n} b(i; n, p)\\ &< \frac{(n-k)p}{k-np} b(k; n, p).\end{aligned}$$

C.7. következmény. *Tekintsük n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset p, a rossz eset pedig $q = 1 - p$ valószínűséggel következik be. Ekkor, $(np + n)/2 < k \leq n$ esetén, annak a valószínűsége, hogy több mint k jó eset fordul elő, kisebb, mint $1/2$-szer annak a valószínűsége, hogy több mint $k - 1$ jó eset fordul elő.*

A következő tételben n számú olyan Bernoulli-kísérletet vizsgálunk, amelyekben a jó eset valószínűsége p_i, ahol $i = 1, 2, \ldots, n$. Amint azt az utána lévő következmény mutatja, a tétel segítségével becslést adhatunk a binomiális eloszlás jobb oldali farkára, ha minden egyes kísérletnél $p_i = p$ választással élünk.

C.8. tétel. *Tekintsük az n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol a jó eset bekövetkezésének a valószínűsége p_i, a rossz eseté pedig $q_i = 1 - p_i$, ahol $i = 1, 2, \ldots, n$. Jelölje az X valószínűségi változó a jó esetek számát, és legyen $\mu = E[X]$. Ekkor minden $r > \mu$ esetén*

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r .$$

Bizonyítás. Mivel bármely $\alpha > 0$ esetén az $e^{\alpha x}$ függvény szigorúan monoton növekvő x-ben, ezért

$$\Pr\{X - \mu \geq r\} = \Pr\left\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\right\}, \tag{C.41}$$

ahol α-nak majd később adunk értéket. A (C.29) Markov-egyenlőtlenséget felhasználva azt kapjuk, hogy

$$\Pr\{X - \mu \geq r\} \leq \mathrm{E}\left[e^{\alpha(X-\mu)}\right] e^{-\alpha r}. \tag{C.42}$$

A bizonyítás során becslést adunk az $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ kifejezésre, és egy megfelelő α értéket helyettesítünk a (C.42) egyenlőtlenségbe. Számoljuk ki az $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ várható értéket. Az 5.2. alfejezet jelölését használva legyen $X_i = I$(az i-edik Bernoulli-kísérlet sikeres), azaz X_i ($i = 1, 2, \ldots, n$) olyan valószínűségi változó, amelynek értéke 1, ha az i-edik Bernoulli-kísérletben a jó eset, és 0, ha a rossz eset következik be. Így

$$X = \sum_{i=1}^{n} X_i,$$

és a várható érték linearitása miatt

$$\mu = \mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p_i,$$

amelyből következik, hogy

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i).$$

$\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ kiszámolásához írjunk $X - \mu$ helyébe $e^{\alpha(X-\mu)}$-t. Így kapjuk, hogy

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \mathrm{E}\left[e^{\alpha \sum_{i=1}^{n}(X_i - p_i)}\right] \\
&= \mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha(X_i - p_i)}\right] \\
&= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i - p_i)}\right],
\end{aligned}$$

ami (C.23)-ból következik, felhasználva, hogy az X_i valószínűségi változók teljes függetlensége maga után vonja az $e^{\alpha(X_i - p_i)}$ valószínűségi változók teljes függetlenségét (lásd C.3-5. gyakorlat). A várható érték definíciója alapján

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X_i - p_i)}\right] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i \\
&= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\
&\leq p_i e^{\alpha} + 1 \\
&\leq \exp(p_i e^{\alpha}),
\end{aligned} \tag{C.43}$$

ahol $\exp(x)$ jelöli az $\exp(x) = e^x$ exponenciális függvényt. (A (C.43) egyenlőtlenség az $\alpha > 0$, $q_i < 1$, $e^{\alpha q_i} \le e^{\alpha}$ és $e^{-\alpha p_i} \le 1$ egyenlőtlenségekből következik, míg az utolsó sort a (3.11) egyenlőtlenségből kapjuk.) Következésképpen

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i-\mu_i)}\right] \\ &\le \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\ &= \exp\left(\sum_{i=1}^{n} p_i e^{\alpha}\right) \\ &= \exp(\mu e^{\alpha}), \end{aligned} \tag{C.44}$$

mivel $\mu = \sum_{i=1}^{n} p_i$. Ezért a (C.41) egyenlőségből, illetve a (C.42) és (C.44) egyenlőtlenségekből következik, hogy

$$\Pr\{X - \mu \ge r\} \le \exp(\mu e^{\alpha} - \alpha r). \tag{C.45}$$

Az $\alpha = \ln(r/\mu)$ választással (lásd C.5-7. gyakorlat) azt kapjuk, hogy

$$\begin{aligned} \Pr\{X - \mu \ge r\} &\le \exp\left(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)\right) \\ &= \exp(r - r\ln(r/\mu)) \\ &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r . \end{aligned}$$

■

Amennyiben a fentieket olyan Bernoulli-kísérletsorozatra alkalmazzuk, ahol a jó esetnek mindegyik kísérletben ugyanannyi a valószínűsége, a C.8. tétel az alábbi következményt szolgáltatja, becslést adván egy binomiális eloszlás jobb oldali farkára.

C.9. következmény. *Tekintsük az n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol mindegyik kísérletben a jó eset valószínűsége p, és a rossz eset valószínűsége $q = 1 - p$. Ekkor minden $r > np$ esetén*

$$\begin{aligned} \Pr\{X - np \ge r\} &= \sum_{k=\lceil np+r \rceil}^{n} b(k; n, p) \\ &\le \left(\frac{npe}{r}\right)^r . \end{aligned}$$

Bizonyítás. A (C.36) egyenletből azt kapjuk, hogy $\mu = E[X] = np$. ■

Gyakorlatok

C.5-1.⋆ Melyik esemény a kevésbé valószínű: egy szabályos érme n-szeri feldobása után egyetlen fejet sem kapunk, vagy az érme $4n$-szeri feldobása után kevesebb, mint n fejet kapunk?

C.5-2.⋆ Bizonyítsuk be a C.6 és a C.7. következményeket.

C.5-3. ⋆ Mutassuk meg, hogy

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b\left(k; n, \frac{a}{a+1}\right)$$

minden $a > 0$ és minden olyan k esetén, amelyre $0 < k < n$.

C.5-4. ⋆ Bizonyítsuk be, hogy ha $0 < k < np$, ahol $0 < p < 1$ és $q = 1 - p$, akkor

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np - k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

C.5-5. ⋆ Mutassuk meg, hogy a C.8. tétel feltételeiből következik, hogy

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r.$$

Hasonlóan mutassuk meg, hogy a C.9. következmény feltételeiből következik, hogy

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r.$$

C.5-6. ⋆ Tekintsük az n számú Bernoulli-kísérletnek egy olyan sorozatát, ahol az i-edik kísérletben a jó eset bekövetkezésének a valószínűsége p_i, a rossz eset bekövetkezésének a valószínűsége pedig $q_i = 1 - p_i$, ahol $i = 1, 2, \ldots, n$. Jelölje az X valószínűségi változó a jó esetek számát, és legyen $\mu = E[X]$. Mutassuk meg, hogy minden $r \geq 0$ esetén

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n}.$$

(*Útmutatás*. Lássuk be, hogy $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{-\alpha^2/2}$. Ezután kövessük a C.8. tétel bizonyítását úgy, hogy a (C.43) egyenlőtlenség helyett ezt az egyenlőtlenséget használjuk.)

C.5-7. ⋆ Mutassuk meg, hogy az $\alpha = \ln(r/\mu)$ választás minimalizálja a (C.45) egyenlőtlenség jobb oldalát.

Feladatok

C-1. Golyók és urnák

Ebben a feladatban azt vizsgáljuk meg különböző feltételek mellett, hogy hányféleképpen helyezhetünk el n golyót b különböző urnában.

a. Tegyük fel, hogy az n golyó megkülönböztethető egymástól, és hogy a golyóknak egy urnán belüli sorrendje nem számít. Bizonyítsuk be, hogy b^n-féleképpen helyezhetjük el a golyókat az urnákba.

b. Tegyük fel, hogy az n golyó megkülönböztethető egymástól, és hogy számít a golyóknak az urnákon belüli sorrendje. Bizonyítsuk be, hogy pontosan $(b + n - 1)!/(b - 1)!$-féleképpen helyezhetjük el a golyókat az urnákba. (*Útmutatás*. Gondoljuk meg, hogy hányféleképpen rendezhetünk sorba n különböző golyót és $b - 1$ egymástól megkülönböztethetetlen gyufaszálat.)

c. Tegyük fel, hogy a golyók megkülönböztethetetlenek egymástól, és ezért egy urnán belüli sorrendjük nem számít. Mutassuk meg, hogy $\binom{b+n-1}{n}$-féleképpen helyezhetjük el a golyókat az urnákba. (*Útmutatás.* Hány ismétlődik a (b) pontbeli elrendezések közül, ha a golyók megkülönböztethetetlenek?)

d. Tegyük fel, hogy a golyók megkülönböztethetetlenek egymástól, és hogy egyetlen urna sem tartalmazhat egynél több golyót. Mutassuk meg, hogy $\binom{b}{n}$-féleképpen helyezhetjük el a golyókat.

e. Tegyük fel, hogy a golyók megkülönböztethetetlenek egymástól, és hogy egyetlen urna sem lehet üres. Mutassuk meg, hogy $\binom{n-1}{b-1}$-féleképpen helyezhetjük el a golyókat.

Megjegyzések a fejezethez

Az első, valószínűségi feladatok megoldására szolgáló, általános módszerek a B. Pascal és P. Fermat közötti, 1654-ben kezdődött híres levelezésben és C. Huygens 1657-ben publikált könyvében jelentek meg. A szigorú valószínűségszámítás J. Bernoulli 1713-ban és A. de Moivre 1730-ban megjelent munkájával kezdődött. Az elmélet további fejlődéséhez hozzájárult még P. S. de Laplace, S.-D. Poisson és C. F. Gauss.

Valószínűségi változók összegeit P. L. Csebisev és A. A. Markov tanulmányozta először. A valószínűségszámítást A. N. Kolmogorov axiomatizálta 1933-ban. Eloszlások farkaira Csernov [59] és Hoeffding [150] adott becsléseket. Erdős Pál jelentős fejlődést elindító munkát végzett a véletlen kombinatorikus struktúrák területén.

Knuth [182] és Liu [205] jó ismertetést ad az elemi kombinatorikáról és leszámlálásról. Az olyan alapvető tankönyvek, mint Billingsley [42], Chung [60], Drake [80], Feller [88] és Rozanov [263], átfogó bevezetést nyújtanak a valószínűségszámításba.

Irodalomjegyzék

[1] M. Abramowitz, I. A. Stegun (szerkesztők). *Handbook of Mathematical Functions*. Dover, 1965. 69

[2] G. M. Adelszon-Velszkij, E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962. 271

[3] L. M. Adleman, C. Pomerance, R. S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117(1):173–206, 1983. 770

[4] A. Aggarwal, J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. 375

[5] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. 30, 51, 325, 375, 397, 454, 551, 656, 792, 862

[6] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. 30, 204, 484

[7] R. K. Ahuja, T. L. Magnanti, J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993. 596, 702

[8] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(1):213–223, 1990. 140, 531

[9] R. K. Ahuja, J. B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989. 596

[10] R. K. Ahuja, J. B. Orlin, R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989. 597

[11] M. Ajtai, J. Komlós, E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983. 617

[12] M. Ajtai, N. Megiddo, O. Waarts. Improved algorithms and analysis for secretary problems and generalizations. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, 473–482. o., 1995. 120

[13] M. Akra, L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998. 93

[14] N. Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35(1):201–204, 1990. 597

[15] A. Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science 709. száma, 60–71. o. Springer-Verlag, 1993. 271

[16] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, 135–141. o., 1996. 171, 379

[17] A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998. 171

[18] T. M. Apostol. *Calculus*, 1. kötet. Blaisdell Publishing Company, 2. kiadás, 1967. 69, 902

[19] S. Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. PhD thesis, University of California, Berkeley, 1994. 862

[20] S. Arora. The approximability of NP-hard problems. In *Proceedings of the 30th Annual ACM Symposium onTheory of Computing*, 337–348. o., 1998. 862

[21] S. Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998. 888

[22] S. Arora, C. Lund. Hardness of approximations. In D. S. Hochbaum (szerkesztő), *Approximation Algorithms for NP-Hard Problems*, 399-446. o. PWS Publishing Company, 1997. 889

[23] J. A. Aslam. A simple bound on the expected height of a randomly built binary search tree. Technical Report TR2001-387, Dartmouth College Department of Computer Science, 2001. 248

[24] M. J. Atallah (szerkesztő). *Algorithms and Theory of Computation Handbook*. CRC Press, 1999. 30

[25] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999. 888

[26] S. Baase A. Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 3. kiadás, 2000. 30

[27] E. Bach. Szóbeli közlés, 1989. 770

[28] E. Bach. Number-theoretic algorithms. In *Annual Review of Computer Science*, 4. kötet, 119–172. o. Annual Reviews, Inc., 1990. 769

[29] E. Bach J., Shallit. *Efficient Algorithms, Algorithmic Number Theory* 1. kötete. The MIT Press, 1996. 769

[30] D. H. Bailey, K. Lee, H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1990. 631

[31] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972. 271

[32] R. Bayer, E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. 271, 397

[33] P. Beauchemin, G. Brassard, C. Crépeau, C. Goutier, C. Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1:53–64, 1988. 770

[34] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957. 325

[35] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. 531

[36] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, 80–86. o., 1983. 171

[37] M. A. Bender, E. D. Demaine, M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 399–409. o., 2000. 397

[38] S. W. Bent, J. W. John. Finding the median requires 2n comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, 213–216. o., 1985. 182

[39] J. L. Bentley. *Writing Efficient Programs*. Prentice Hall, 1982. 30

[40] J. L. Bentley. *Programming Pearls*. Addison-Wesley, 1986. 30, 156

[41] J. L. Bentley, D. Haken, J. B. Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980. 93

[42] P. Billingsley. *Probability and Measure*. John Wiley & Sons, 2. kiadás, 1986. 950

[43] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. 182

[44] B. Bollobás. *Random Graphs*. Academic Press, 1985. 120

[45] J. A. Bondy, U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, 1976. 830

[46] G. Brassard, P. Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, 1988. 30, 69

[47] G. Brassard, P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996. 30, 354

[48] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980. 770

[49] M. R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Computer Science Department, Stanford University, 1977. Technical Report STAN-CS-77-600. 415

[50] M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978. 415

[51] J. P. Buhler, H. W. Lenstra, Jr., C. Pomerance. Factoring integers with the number field sieve. In A. K. Lenstra, H. W. Lenstra, Jr. (szerkesztők), *The Development of the Number Field Sieve*, *Lecture Notes in Mathematics* 1554. kötete, 50–94. o. Springer-Verlag, 1993. 770

[52] J. L. Carter, M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. 231

[53] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000. 498

[54] J. Cheriyan, T. Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2):203–226, 1995. 597

[55] J. Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989. 597

[56] B. V. Cherkassky, A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997. 597

[57] B. V. Cherkassky, A. V. Goldberg, T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996. 532

[58] B. V. Cherkassky, A. V. Goldberg, C. Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999. 140

[59] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952. 950

[60] K. L. Chung. *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974. 950

[61] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. 889

[62] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983. 702

[63] V. Chvátal, D. A. Klarner, D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972. 325

[64] A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, 24–30. o. North-Holland, 1964. 862

[65] H. Cohen, H. W. Lenstra, Jr.. Primality testing and Jacobi sums. *Mathematics of Computation*, 42(165):297–330, 1984. 770

[66] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979. 397

[67] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 151–158. o., 1971. 862

[68] J. W. Cooley, J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. 724

[69] D. Coppersmith. Modifications to the number field sieve. *Cryptology*, 6(1):169–180, 1993. 770

[70] D. Coppersmith, S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990. 550, 655

[71] T. H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992. 375

[72] E. V. Denardo, B. L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979. 140

[73] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994. 231

[74] W. Diffie, M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. 770

[75] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(3):269–271, 1959. 531

[76] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970. 596

[77] B. Dixon, M. Rauch, R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992. 499

[78] J. D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91(6):333–352, 1984. 769

[79] D. Dor, U. Zwick. Selecting the median. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, 28–37. o., 1995. 182

[80] A. W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967. 950

[81] J. R. Driscoll, H. N. Gabow, R. Shrairman, R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988. 434

[82] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989. 378

[83] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, *EATCS Monographs on Theoretical Computer Science* 10. kötete. Springer-Verlag, 1987. 819

[84] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. 862

[85] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971. 354

[86] J. Edmonds, R. M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972. 596

[87] S. Even. *Graph Algorithms*. Computer Science Press, 1979. 484, 596

[88] W. Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, 3. kiadás, 1968. 950

[89] R. W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of ACM*, 5(6):345, 1962. 550

[90] R. W. Floyd. Algorithm 245 (TREESORT). *Communications of ACM*, 7(12):701, 1964. 139

[91] R. W. Floyd. Permuting information in idealized two-level storage. In R. E. Miller, J. W. Thatcher (szerkesztők), *Complexity of Computer Computations*, 105–109. o. Plenum Press, 1972. 375

[92] R. W. Floyd, R. L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975. 182

[93] L. R. Ford, Jr., D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962. 531, 596

[94] L. R. Ford, Jr., S. M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387–389, 1959. 170

[95] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976. 550

[96] M. L. Fredman, J. Komlós, E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. 231

[97] M. L. Fredman, M. E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 345–354. o., 1989. 455

[98] M. L. Fredman, R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. 434, 498

[99] M. L. Fredman, D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993. 140, 171, 378

[100] M. L. Fredman, D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994. 499

[101] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4):107–114, 2000. 484

[102] H. N. Gabow, Z. Galil, T. Spencer, R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986. 498

[103] H. N. Gabow, R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985. 455

[104] H. N. Gabow, R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989. 532

[105] Z. Galil, O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1997. 550

[106] Z. Galil, O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, 1997. 550

[107] Z. Galil, J. Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983. 792

[108] I. Galperin, R. L. Rivest. Scapegoat trees. In *Proceedings of the 4st ACM-SIAM Symposium on Discrete Algorithms*, 165–174. o., 1993. 271

[109] M. R. Garey, R. L. Graham, J. D. Ullman. Worst-case analyis of memory allocation algorithms. In *Proceedings of the Fouth Annual ACM Symposium on Theory of Computing*, 143–150. o., 1972. 888

[110] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 862, 888

[111] S. Gass. *Linear Programming: Methods and Applications*. International Thomson Publishing, 4. kiadás, 1975. 702

[112] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972. 354

[113] A. George, J. W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981. 655

[114] E. N. Gilbert, E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959. 325

[115] M. X. Goemans, D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In D. S. Hochbaum (szerkesztő), *Approximation Algorithms for NP-Hard Problems*, 144–191. o. PWS Publishing Company, 1997. 889

[116] M. X. Goemans, D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995. 889

[117] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987. 596

[118] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995. 532

[119] A. V. Goldberg, S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998. 597

[120] A. V. Goldberg, É. Tardos, R. E. Tarjan. Network flow algorithms. In B. Korte, L. Lovász, H. J. Prömel, A. Schrijver (szerkesztők), *Paths, Flows, and VLSI-Layout*, 101–164. o. Springer-Verlag, 1990. 596

[121] A. V. Goldberg, R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988. 596

[122] D. Goldfarb, M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy-Kan, M. J. Todd (szerkesztők), *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, 73–170. o. Elsevier Science Publishers, 1989. 702

[123] S. Goldwasser, S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984. 770

[124] S. Goldwasser, S. Micali, R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988. 770

[125] G. H. Golub, C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3. kiadás, 1996. 618, 655, 656

[126] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984. 30, 204, 231

[127] R. C. Gonzalez, R. E. Woods. *Digital Image Processing*. Addison-Wesley, 1992. 724

[128] M. T. Goodrich, R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998. 30, 204

[129] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966. 888

[130] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972. 819

[131] R. L. Graham, P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985. 498

[132] R. L. Graham, D. E. Knuth, O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2. kiadás, 1994. 69, 93

[133] D. Gries. *The Science of Programming*. Springer-Verlag, 1981. 52

[134] M. Grötschel, L. Lovász, A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988. 702

[135] L. J. Guibas, R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, 8–21. o. IEEE Computer Society, 1978. 271, 397

[136] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. 30

[137] Y. Han. Improved fast integer sorting in linear space. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, 793–796. o., 2001. 171

[138] F. Harary. *Graph Theory*. Addison-Wesley, 1969. 922

[139] G. C. Harfst, E. M. Reingold. A potential-based amortized analysis of the union-find data structure. *ACM SIGACT News*, 31(3):86–95, 2000. 454, 455

[140] J. Hartmanis, R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965. 829

[141] M. T. Heideman, D. H. Johnson, C. S. Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, 14–21. o., 1984. 724

[142] M. R. Henzinger, V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, 664–672. o., 1995. 379

[143] M. R. Henzinger, V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999. 379

[144] M. R. Henzinger, S. Rao, H. N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 34(2):222–250, 2000. 379

[145] N. J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352–368, 1990. 631

[146] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of ACM*, 4(7):321–322, 1961. 182

[147] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962. 156

[148] D. S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6(3):243–254, 1983. 889

[149] D. S. Hochbaum (szerkesztő). *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997. 862, 888

[150] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27:713–721, 1956. 950

[151] M. Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987. 120

[152] J. E. Hopcroft, R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. 597

[153] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2. kiadás, 2001. 862

[154] J. E. Hopcroft, R. E. Tarjan. Efficient algorithms for graph manipulation. *Journal of the ACM*, 16(6):372–378, 1973. 484

[155] J. E. Hopcroft, J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973. 454

[156] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. 825, 832

[157] E. Horowitz, S. Sahni, S. Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998. 30

[158] E. Horowitz, S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978. 354

[159] T. C. Hu, M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982. 325

[160] T. C. Hu, M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984. 325

[161] T. C. Hu, A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971. 325

[162] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 354

[163] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *SC96 Technical Papers*, 1996. 631

[164] O. H. Ibarra, C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal ot the ACM*, 22(4):463–468, 1975. 889

[165] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973. 819

[166] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977. 550

[167] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974. 862, 888, 889

[168] D. S. Johnson. The NP-completeness column: An ongoing guide the tale of the second prover. *Journal of Algorithms*, 13(3):502–524, 1992.

[169] D. R. Karger, P. N. Klein, R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995. 499

[170] D. R. Karger, D. Koller, S. J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993. 551

[171] H. Karloff. *Linear Programming*. Birkhäuser, 1991. 702

[172] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. 702

[173] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher (szerkesztők), *Complexity of Computer Computations*, 85–103. o. Plenum Press, 1972. 862

[174] R. M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1-3):165–201, 1991. 120

[175] R. M. Karp, M. O. Rabin. Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981. 792

[176] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974. 596

[177] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997. 498

[178] V. King, S. Rao, R. E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17:447–474, 1994. 597

[179] J. H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, 1990. 30

[180] D. G. Kirkpatrick, R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(2):287–299, 1986. 819

[181] P. N. Klein, N. E. Young. Approximation algorithms for NP-hard optimization problems. In *CRC Handbook on Algorithms*, 341–3419. o. CRC Press, 1999. 888

[182] D. E. Knuth. *Fundamental Algorithms*, *The Art of Computer Programming* 1. kötete. Addison-Wesley, 1968. 3. kiadás, 1997. 30, 51, 69, 93, 204, 484, 902, 950

[183] D. E. Knuth. *Seminumerical Algorithms*, *The Art of Computer Programming* 2. kötete. Addison-Wesley, 1969. 3. kiadás, 1998. 30, 51, 769

[184] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971. 320, 325

[185] D. E. Knuth. *Sorting and Searching*, *The Art of Computer Programming* 3. kötete. Addison-Wesley, 1973. 2. kiadás, 1998. 30, 51, 170, 231, 248, 325, 397, 617

[186] D. E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2):18–23, 1976. 69

[187] D. E. Knuth, J. H. Morris Jr., V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. 792

[188] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985. 499

[189] B. Korte, L. Lovász. Mathematical structures underlying greedy algorithms. In F. Gécseg (szerkesző), *Fundamentals of Computation Theory*, *Lecture Notes in Computer Science* 117. kötete, 205–209. o. Springer-Verlag, 1981. 354

[190] B. Korte, L. Lovász. Structural properties of greedoids. *Combinatorica*, 3(3-4):359–374, 1983. 354

[191] B. Korte, L. Lovász. Greedoids – a structural framework for the greedy algorithm. In W. Pulleybank (szerkesztő), *Progress in Combinatorial Optimization*, 221–243. o. Academic Press, 1984. 354

[192] B. Korte, L. Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229–238, 1984. 354

[193] D. C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992. 30, 454

[194] D. W. Krumme, G. Cybenko, K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992. 375

[195] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956. 498

[196] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976. 354, 531, 550, 596

[197] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys (szerkesztők). *The Traveling Salesman Problem*. John Wiley & Sons, 1985. 888

[198] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961. 484

[199] F. T. Leighton, S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, 422–431. o., 1988. 889

[200] D. A. Lelewer, D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987. 354

[201] A. K. Lenstra, Lenstra, Jr., H. W., M. S. Manasse, J. M. Pollard. The number field sieve. In A. K. Lenstra, H. W. Lenstra, Jr. (szerkesztők), *The Development of the Number Field Sieve*, *Lecture Notes in Mathematics* 1554. kötete, 11–42. o. Springer-Verlag, 1993. 770

[202] H. W. Lenstra, Jr.. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987. 770

[203] L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. Oroszul. 862

[204] H. R. Lewis, C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 2. kiadás, 1998. 825, 862

[205] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968. 93, 950

[206] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975. 889

[207] L. Lovász, M. D. Plummer. *Matching Theory*, *Annals of Discrete Mathematics* 121. kötete. North-Holland, 1986. 597

[208] B. M. Maggs, S. A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988. 551

[209] M. Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999. 204

[210] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. 30

[211] C. Martínez, S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998. 248

[212] W. J. Masek, M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. 325

[213] H. A. Maurer, T. Ottmann, H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976. 271

[214] E. W. Mayr, H. J. Prömel, A. Steger (szerkesztők). *Lectures on Proof Verification and Approximation Algorithms*, *Lecture Notes in Computer Science* 1367. kötete. Springer-Verlag, 1998. 862

[215] C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995. 551

[216] M. D. McIlroy. A killer adversary for quicksort. *Software Practice and Experience*, 29(4):341–344, 1999. 156

[217] K. Mehlhorn. *Graph Algorithms and NP-Completeness*, *Data Structures and Algorithms* 2. kötete. Springer-Verlag, 1984. 30

[218] K. Mehlhorn. *Multidimensional Searching and Computational Geometry*, *Data Structures and Algorithms* 3. kötete. Springer-Verlag, 1984. 30

[219] K. Mehlhorn. *Sorting and Searching*, *Data Structures and Algorithms* 1. kötete. Springer-Verlag, 1984. 30

[220] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. 770

[221] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976. 756, 770

[222] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996. 52

[223] J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, *k*-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999. 889

[224] L. Monier. *Algorithmes de Factorisation D'Entiers*. PhD thesis, L'Université Paris-Sud, 1980. 770

[225] L. Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980. 770

[226] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, 285–292. o. Harvard University Press, 1959. 484

[227] R. Motwani, J. (S.) Naor, P. Raghavan. Randomized approximation algorithms in combinatorial optimization. In D. S. Hochbaum (szerkesztő), *Approximation Algorithms for NP-Hard Problems*, 447–481. o. PWS Publishing Company, 1997. 889

[228] R. Motwani, P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 120

[229] J. I. Munro, V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2):151–160, 1996. 171

[230] J. Nievergelt, E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973. 271

[231] I. Niven, H. S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, 4. kiadás, 1980. 727, 747, 769

[232] A. V. Oppenheim, R. W. Schafer, J. R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, 2. kiadás, 1998. 724

[233] A. V. Oppenheim, A. S. Willsky, S. Hamid Nawab. *Signals and Systems*. Prentice Hall, 2. kiadás, 1997. 724

[234] J. B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1):109–129, 1997. 702

[235] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1993. 819

[236] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 862

[237] C. H. Papadimitriou, K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982. 354, 596, 702, 888

[238] M. S. Paterson, 1974. Publikálatlan előadás, Ile de Berder, France. 655

[239] M. S. Paterson. Progress in selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, 368–379. o., 1996. 182

[240] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000. 30

[241] S. Phillips, J. Westbrook. Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium onTheory of Computing*, 402–411. o., 1993. 597

[242] J. M. Pollard. A Monte Carlo method for factorization. *BIT*, 15(3):331–334, 1975. 770

[243] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra, Jr. H. W. Lenstra (szerkesztők), *The Development of the Number Field Sieve*, *Lecture Notes in Mathematics* 1554. kötete, 4–10. o. Springer-Verlag, 1993. 770

[244] C. Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587–593, 1981. 758

[245] C. Pomerance (szerkesztő). *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990. 769

[246] W. K. Pratt. *Digital Image Processing*. John Wiley & Sons, 2. kiadás, 1991. 724

[247] F. P. Preparata, M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 285, 819

[248] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986. 655, 724

[249] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988. 655, 724

[250] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957. 498

[251] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Journal of the ACM*, 33(6):668–676, 1990. 271

[252] P. W. Purdom, Jr., C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985. 30, 93

[253] M. O. Rabin. Probabilistic algorithms. In J. F. Traub (szerkesztő), *Algorithms and Complexity: New Directions and Recent Results*, 21–39. o. Academic Press, 1976. 120

[254] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980. 770

[255] P. Raghavan, C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987. 889

[256] R. Raman. Recent results on the single-source shortest paths problem. *ACM SIGACT News*, 28(2):81–87, 1997. 140, 531

[257] E. M. Reingold, J. Nievergelt, N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, 1977. 30

[258] H. Riesel. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser, 1985. 769

[259] R. L. Rivest, A. Shamir, L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communication of the ACM*, 21(2):120–126, 1978. Lásd még az U.S. Patent 4 405 829 számú szabadalmat. 770

[260] H. Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62(1):26–29, 1955. 69

[261] D. J. Rosenkrantz, R. E. Stearns, P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563–581, 1977. 888

[262] S. Roura. An improved master theorem for divide-and-conquer recurrences. In *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, 449–459. o., 1997. 93

[263] Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969. 950

[264] S. Sahni, T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976. 889

[265] A. Schönhage, M. Paterson, N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976. 182

[266] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986. 702

[267] A. Schrijver. Paths and flows a historical survey. *CWI Quarterly*, 6(3):169–183, 1993. 596

[268] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978. 156

[269] R. Sedgewick. *Algorithms*. Addison-Wesley, 2. kiadás, 1988. 30, 397

[270] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995. 550

[271] R. Seidel, C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996. 271

[272] J. Setubal, J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997. 30, 323

[273] C. A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, 2. kiadás, 2001. 204

[274] J. Shallit. Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4):401–419, 1994. 769

[275] M. I. Shamos, D. Hoey. Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, 208–215. o., 1976. 819

[276] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981. 484

[277] D. B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. In W. Cook, L. Lovász, P. Seymour (szerkesztők), *Combinatorial Optimization*, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 20. kötete. American Mathematical Society, 1995. 888

[278] A. Shoshan, U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 605–614. o., 1999. 551

[279] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997. 862

[280] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998. 30

[281] D. D. Sleator, R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. 271, 378

[282] D. D. Sleator, R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. 378

[283] J. Spencer. *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987. 120

[284] D. A. Spielman, S.-H. Teng. The simplex algorithm usually takes a polynomial number of steps. In *Proceedings of the 33rd Annual ACM Symposium onTheory of Computing*, 2001. 702

[285] G. Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986. 655, 656

[286] G. Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, third kiadás, 1988. 655, 656

[287] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969. 655, 656

[288] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975. 325

[289] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. 484

[290] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. 454

[291] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979. 455

[292] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983. 271, 378, 454, 484, 498, 596

[293] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985. 375

[294] R. E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999. 454

[295] R. E. Tarjan, J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984. 455

[296] G. B. Thomas, Jr., R. L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, 7. kiadás, 1988. 69, 902

[297] M. Thorup. Faster determinstic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 550–555. o., 1998. 171

[298] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999. 532

[299] M. Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000. 140, 531

[300] R. Tolimieri, M. An, C. Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer-Verlag, 2. kiadás, 1997. 724

[301] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 75–84. o. IEEE Computer Society, 1975. 140, 378

[302] J. van Leeuwen (szerkesztő). *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science Publishers és The MIT Press, 1990. 30

[303] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992. 724

[304] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996. 702

[305] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001. 888

[306] R. M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2):568–581, 1997. 93

[307] J. Vuillemin. A data structure for manipulating priority queues. *Journal of the ACM*, 21(4):309–315, 1978. 415

[308] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962. 550

[309] M. S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes*. Chapman & Hall, 1995. 30

[310] M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1994. 204

[311] M. A. Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison-Wesley, 1996. 271

[312] M. A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998. 204

[313] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999. 204

[314] H. Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, 1935. 354

[315] H. S. Wilf. *Algorithms and Complexity*. Prentice Hall, 1986. 30

[316] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7(6):347–348, 1964. 139

[317] S. Winograd. On the algebraic complexity of functions. In *Actes du Congrès International des Mathématiciens*, 3. kötet, 283–288. o., 1970. 656

[318] A. C.-C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780–787, 1981. 819

[319] Y. Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997. 702

[320] D. Zwillinger (szerkesztő). *CRC Standard Mathematical Tables and Formulae*. CRC Press, 30. kiadás, 1996. 69

Ezt az irodalomjegyzéket HBibTEXsegítségével állítottuk össze. A dokumentumokat elsősorban a szerzők neve (első szerző vezetékneve és utóneveinek rövidítései, második szerző vezetékneve és utóneveinek rövidítései stb.), másodsorban a megjelenés éve, harmadsorban pedig a dokumentumok címe alapján rendeztük.

Az aláhúzás azt jelzi, hogy az irodalomjegyzéknek a könyv honlapján lévő PDF változatában az aláhúzott szövegrészre kattintva a megfelelő honlapra juthatunk.

Tárgymutató

Ez a tárgymutató a következő szempontok szerint készült.

Először a matematikai jelöléseket, azután a tárgyszavakat soroljuk fel.

A számokat és görög betűket tartalmazó tárgyszavakat kiejtésük szerint rendeztük: például a„2-3-4 fá”-t „kettőháromnégy fa”-ként, a „*d*-kupac”-ot „dkupac”-ként, az „ϵ-sűrű gráf”-ot „epszilonsűrű gráf”-ként.

Az algoritmusok (pszeudokódok) nevét a betűmérettel is megkülönböztettük a többi névből (függvények, modellek, problémák, nyelvek stb.): például BINOMÁLIS-KUPACBAN-KERES, illetve VAGY-függvény, ÚT, SAT. A kezdőbetűkből álló szavak betűi azonban mindenütt azonos méretűek: pl. ALAP-FFT, LKR-NYOMTAT, illetve FIFO, MFF, MFH.

Ha egy tárgyszó nem a fő szövegre utal, akkor az oldalszámot kiegészítés követi: *gy* gyakorlatot, *fe* feladatot, *áb* ábrát, *lá* lábjegyzetet jelent.

A különböző típusú objektumokat lehetőség szerint tipográfiailag is megkülönböztettük. A matematikai jelöléseket és a programokban használt változók nevét dőlt betűk emelik ki, mint például $\Omega(n \lg n)$ vagy *rang*[*kulcs*]. Az algoritmusok neveit kiskapitális betűkkel írtuk, mint például GYORSRENDEZ. Az algoritmusok kódjában a programozási alapszavakat félkövéren szedtük, mint például **if, then, for**.

Az algoritmusok nevében kiskötőjelet használtunk, mint például UTAZÓ-ÜGYNÖK. Az egyes fogalmak meghatározásának és az algoritmusok pszeudokódjának helyére a tárgymutató dőlt oldalszámmal utal.

Elsősorban az algoritmusokat tárgyaló tankönyvek matematikai jelöléseit alkalmaztuk. Az oldalszámok felsorolásánál nem törekedtünk teljességre.

Névmutató

A névmutatóban a magyar és külföldi szerzők utónevét lehetőleg kiírtuk (csak akkor alkalmaztunk rövidítést, ha nem ismertük a teljes utónevet).