Erkki Mäkinen ja Timo Poranen

Algoritmit

INFORMAATIOTIETEIDEN YKSIKKÖ
TAMPEREEN YLIOPISTO

INFORMAATIOTIETEIDEN YKSIKÖN RAPORTTEJA 1/2011

TAMPERE 2011

TAMPEREEN YLIOPISTO
INFORMAATIOTIETEIDEN YKSIKKÖ
INFORMAATIOTIETEIDEN YKSIKÖN RAPORTTEJA 1/2011
KESÄKUU 2011

Erkki Mäkinen ja Timo Poranen

Algoritmit

3. painos

INFORMAATIOTIETEIDEN YKSIKKÖ
33014 TAMPEREEN YLIOPISTO

ISBN 978-951-44-8509-1

ISSN-L 1799-8158
ISSN 1799-8158

Sisältö

Luku 1

Aluksi

Tämä moniste on tarkoitettu käytettäväksi Tampereen yliopiston informaatiotieteiden yksikön opintojaksolla Algoritmit. Monisteen oheislukemistona on syytä käyttää jotain alan oppikirjaa. Suositeltavia yleisesityksiä ovat esimerkiksi Cormenin ja muiden [1], Goodrichin ja Tamassian [3] sekä Kleinbergin ja Tardosin [6] kirjat. Algoritmien analysointiin keskittyvät Grahamin ja muiden [4] ja Sedgewickin ja Flajolet'n [12] esitykset. Erityisesti kannattaa huomioida Penttosen suomenkielinen oppikirja [11]. Muut kirjallisuusluettelon kirjat keskittyvät algoritmiikan eri erityisalueisiin.

Potenssi, logaritmi ja summakaavoja

Lukijalta edellytetään ennakkotietoina matematiikan perusteita, esimerkiksi induktioperiaate, logaritmifunktio, permutaatiot ja todennäköisyyslaskennan alkeet. Alla on kerrattu muutamia potenssi- ja logaritmifunktioiden ominaisuuksia, joita jatkossa toistuvasti käytetään.

Jos a, m ja n ovat reaalilukuja, $a \neq 0$, niin

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

Jos logaritmin kantaluvulla ei ole merkitystä, niin kantaluku jätetään merkitsemättä. Näin tehdään usein esimerkiksi asymptoottisten merkintöjen yhteydessä. Jos a, b ja c ovat

nollaa suurempia reaalilukuja, niin

$$\begin{aligned}
a &= b^{\log_b a} \\
\log 1 &= 0 \\
\log(ab) &= \log a + \log b \\
\log b^a &= a \log b \\
a^{\log b} &= b^{\log a} \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b \frac{1}{a} &= \log_b a^{-1} = -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
\log \log n &= \log(\log n) \\
\log^k n &= (\log n)^k.
\end{aligned}$$

Merkintä $\lfloor a \rfloor$ tarkoittaa suurinta kokonaislukua x, jolla pätee $x \leq a$. Vastaasti $\lceil a \rceil$ tarkoittaa pienintä kokonaislukua $x \geq a$.

Seuraavia summakaavoja tarvitaan myös toistuvasti:

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

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}, \qquad \text{jos } a \neq 1 \text{ , ja}$$

$$\sum_{i=0}^{n} i a^i = \frac{n a^{n+2} - (n+1)a^{n+1} + a}{(a-1)^2}, \qquad \text{jos } a \neq 1.$$

Kirjallisuutta

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to Algorithms*. Third Edition. The MIT Press, 2009.

[2] S. Even, *Graph Algorithms*. Computer Science Press, 1979.

[3] M.T. Goodrich and R. Tamassia, *Algorithm Design*. Wiley, 2001.

[4] R.L. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics*. Addison-Wesley, 1989.

[5] J. Jaja, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[6] J. Kleinberg and E. Tardos, *Algorithm Design*. Addison-Wesley, 2006.

[7] T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. Morgan & Kaufmann, 1991.

[8] U. Manber, *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[9]

[10] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1992.

[11] M. Penttonen, *Johdatus algoritmien suunnitteluun ja analysointiin*. Otatieto, 1997.

[12] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996.

[13] J.A. Storer, *Data Compression. Methods and Theory*. Computer Science Press, 1988.

Luku 2

Algoritmien analysointitekniikoita

Mikäli pyritään täsmällisen määrittelyn sijasta vain intuitiivisesti kuvailemaan algoritmeja, voitaisiin niitä sanoa esimerkiksi "toimintaohjeiden joukoiksi", joilla on ainakin seuraavat ominaisuudet:

1. *Ilmaistavuus*. Algoritmi on voitava ilmaista kiinteällä, äärellisellä aakkostolla.
2. *Yksiselitteisyys*. Jokaisen yksittäisen ohjeen on oltava sellainen, että ei synny erimielisyyttä sen suoritustavasta.
3. *Suoritettavuus*. Jokainen ohje on voitava suorittaa "äärellisessä ajassa ja äärellisin apuvälinein".

Esimerkiksi kaikilla ohjelmointikielillä kirjoitetut ohjelmat täyttävät nämä vaatimukset. Tämän vuoksi tässä esityksessä algoritmit kirjoitetaan ohjelmointikielten ilmauksia jäljittelevällä pseudokoodilla. Haluttaessa tämä pseudokoodi voitaisiin "kääntää" jonkun täsmällisesti määritellyn laskennan mallin mukaiseksi. Tällaisia malleja ovat esimerkiksi Turingin kone ja RAM (Random Access Machine).

Algoritmi on *deterministinen*, jos sen ohjeet jokaisessa tilanteessa määräävät suoritettavan toiminta-askeleen yksikäsitteisesti. Muussa tapauksessa algoritmi on *epädeterministinen*. Luvussa 8 tarkasteltavat satunnaistetut algoritmit ovat epädeterministisiä.

Ominaisuuksista 1 ja 2 seuraa, että erilaisia algoritmeja on vain numeroituva määrä. Toisaalta esimerkiksi funktioita luonnollisten lukujen joukolta sille itselleen on ylinumeroituva määrä. Kaikki funktiot eivät siis ole *laskettavissa* (computable) eli kaikilla funktioilla ei ole algoritmia, joka määräisi funktion arvot kaikilla syötteillä. Samoin on olemassa *päätösongelmia* (ongelmia, joihin vastataan "kyllä" tai "ei"), jotka eivät ole *ratkeavia* (solvable, decidable) vaan *ratkeamattomia* (unsolvable, undecidable).

Tarkastellaan seuraavia ongelmia:

P_1 Olkoon A jokin algoritmi. Pysähtyykö A syötteellä x?
P_2 Olkoon G jokin graafi. Onko G väritettävissä k:lla värillä?
P_3 Olkoon S joukko kokonaislukuja. Etsi joukon S mediaani.

Ongelmiin $P_1 - P_3$ liittyviä tyypillisiä algoritmiteoreettisia kysymyksiä ovat esimerkiksi seuraavat: Onko olemassa algoritmia, joka ratkaisee ongelman P? Paljonko P:n ratkaise-

miseen kuluu aikaa ja tilaa, kun käytetään annettua algoritmia? Paljonko P:n ratkaisemiseen vähintään kuluu aikaa? Onko annettu algoritmi paras mahdollinen? Miten aika- ja tilavaatimukset riippuvat toisistaan?

Ongelma P_1 on ratkeamaton päätösongelma. Ongelmilla P_2 ja P_3 puolestaan on ratkaisualgoritmit. Ongelmasta P_3 tiedetään myös, kuinka paljon aikaa sen ratkaisemiseen vähintään kuluu. Jos löydetään algoritmi, jonka aikavaatimus on juuri tämän alarajan suuruinen, on löydetty optimaalinen algoritmi ongelmalle P_3. Kaikille ongelmille ei tunneta aikavaatimuksen alarajoja. Esimerkiksi ongelmasta P_2 ei edes tiedetä, voidaanko se ratkaista polynomisessa ajassa. Ongelma P_2 kuuluu ns. NP-täydellisten ongelmien luokkaan.

Olkoon P jokin ongelma ja A algoritmi, joka ratkaisee ongelman P. Ongelmalla P on useimmiten erikokoisia *tapauksia* (instance). Merkitään tapauksen kokoa luvulla n. Tilanteesta riippuen n voi olla esimerkiksi tapauksen esittämiseen tarvittavien muistipaikkojen tai bittien lukumäärä tai jokin muu sopivasti valittu suure. Algoritmien aikavaatimuksia mitattaessa käytettäviä yksikköjä voivat olla esimerkiksi

- vertailujen lukumäärä,
- tietyn silmukan suorituskertojen määrä,
- aliohjelman kutsukertojen määrä tai
- "seinäkelloaika" eli algoritmin suorittamiseen todellisuudessa kuluva aika.

Algoritmin A analyysin tarkoituksena on selvittää, kuinka paljon A (luvun n funktiona) tarvitsee aikaa ja tilaa *pahimmassa*, *keskimääräisessä* ja *parhaassa* tapauksessa. Usein tyydytään johtamaan asymptoottisia yläraja-arvioita, jotka ilmaisevat, miten aika- tai tilavaatimus käyttäytyy, kun n kasvaa rajatta.

2.1. Asymptoottiset merkintätavat

Algoritmien aika- ja tilavaatimusten tarkka laskeminen on usein vaikeaa. Yleensä riittää, että tiedetään kasvun tyyppi. Tätä varten määritellään kolme asymptoottista merkintätapaa, joissa tarkkoja kertoimia ei huomioida.

Tarkastellaan funktioita $f : \mathbb{N} \to \mathbb{R}^+$ ja $g : \mathbb{N} \to \mathbb{R}^+$. Tällöin määritellään

- f on $O(g)$, jos on olemassa sellaiset positiiviset vakiot c ja n_0, että $f(n) \leq cg(n)$, kun $n \geq n_0$,
- f on $\Omega(g)$, jos on olemassa sellaiset positiiviset vakiot c ja n_0, että $f(n) \geq cg(n)$, kun $n \geq n_0$,
- f on $\Theta(g)$, jos on olemassa sellaiset positiiviset vakiot c_1, c_2 ja n_0, että $c_1 g(n) \leq f(n) \leq c_2 g(n)$, kun $n \geq n_0$.

Yleensä tarvitaan merkintää $O(g)$, jota käytetään joskus silloinkin, kun myös $\Theta(g)$ sopisi.

Yleisessä tapauksessa asymptoottisten merkintöjen yhteydessä tarkastellaan funktioiden arvojen itseisarvoja. Kun kuitenkin määriteltiin, että funktiot f ja g voivat saada vain positiivisia arvoja, ei itseisarvomerkkejä tarvita. Kun f ja g ilmaisevat algoritmien resurssivaatimuksia, on oletus niiden arvojen positiivisuudesta luonnollinen.

Edellä merkittiin "f on $O(g)$"; usein merkitään myös $f = O(g)$. Tässä ei kuitenkaan ole kyse tavanomaisesta yhtäsuuruudesta, mikä on syytä huomioida käsiteltäessä O-merkintöjä sisältäviä lausekkeita. Sama pätee tietenkin myös Ω- ja Θ-merkinnöille. Tarkkaan ottaen yllä määritellyt kolme merkintätapaa määrittelevät kukin funktioiden joukkoja. Esimerkiksi merkinnällä $O(n^2)$ määritellään sellainen funktioiden joukko, johon kuuluvat kaikki ne funktiot, joille voidaan valita yllä olevat ehdot täyttävät vakiot c ja n_0, kun $g(n) = n^2$. Lausekkeessa

$$2n^3 + 4n^2 + n - 3 = 2n^3 + O(n^2)$$

O-merkintä tulkitaan niin, että $O(n^2)$ voitaisiin tarvittaessa korvata jollain ko. joukkoon kuuluvalla funktiolla, mutta tässä yhteydessä ei olla kiinnostuneita, mikä nimenomainen funktio se olisi.

O-merkinnälle pätevät esimerkiksi säännöt

- $f(n) = O(f(n))$,
- $cO(f(n)) = O(f(n)) = O(cf(n))$, kun c on vakio,
- $O(f(n)) + O(f(n)) = O(f(n))$,
- $O(O(f(n))) = O(f(n))$,
- $O(f(n))O(g(n)) = O(f(n)g(n))$ ja
- $O(f(n)g(n)) = f(n)O(g(n))$.

Joskus kirjallisuudessa käytetään pikku-o:ta tarkoittamaan "ei-tiukkaa O:ta" seuraavasti:

- f on $o(g)$, jos kaikilla vakioilla $c > 0$ on olemassa sellainen positiivinen vakio n_0, että $f(n) < cg(n)$, kun $n \geq n_0$.

Esimerkiksi $2n$ on $o(n^2)$, mutta $2n^2$ ei ole $o(n^2)$. Vastaavasti määritellään ω tarkoittamaan "ei-tiukkaa Ω:a".

Logaritmin kantaluvulla ei O-merkinnän yhteydessä yleensä ole merkitystä, kunhan kantaluku on ykköstä suurempi vakio. Tämä johtuu siitä, että kantaluvun muunnos vastaa vakiolla kertomista. Ykköstä pienempien ja ei-vakioisten kantalukujen lisäksi ei myöskään eksponentissa olevaa logaritmin kantalukua voi kuitenkaan korvata toisella.

Asymptoottiset merkintätavat liittyvät läheisesti raja-arvon käsitteeseen. Voidaan esimerkiksi todistaa, että jos

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

kun a on positiivinen reaaliluku, niin $f(n)$ on $\Theta(g)$. Vastaavasti kun

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$

niin $f(n)$ on $o(g(n))$, ja kun

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

niin $f(n)$ on $\omega(g(n))$.

Esimerkki 2.1 Olkoon $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$. Polynomi $f(n)$ on $O(n^m)$, sillä kun $n \geq 1$, voidaan kirjoittaa

$$\begin{aligned} f(n) &\leq |a_m| n^m + |a_{m-1}| n^{m-1} + \ldots + |a_1| n + |a_0| \\ &= (|a_m| + |a_{m-1}|/n + \ldots + |a_1|/n^{m-1} + |a_0|/n^m) n^m \\ &\leq (|a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|) n^m. \end{aligned}$$

Edellä olevan määritelmän ensimmäisen kohdan vakiot voidaan nyt valita seuraavasti:

$$c = |a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|$$

ja $n_0 = 1$. □

Tavallisia resurssivaatimuksissa esiintyviä funktioiden kasvuvauhteja ovat esimerkiksi seuraavat:

- $O(1)$, vakio eli syötteen koko ei vaikuta resurssitarpeeseen,
- $O(\log n)$, logaritminen (tällöin syötettä ei voida lukea kokonaan),
- $O(n)$, lineaarinen,
- $O(n \log n)$ (esimerkiksi monet lajittelualgoritmit),
- $O(n^2)$, neliöllinen (quadratic),
- $O(n^3)$, kuutiollinen (cubic), käytännöllinen enää pienillä n:n arvoilla,
- $O(2^n)$, eksponentiaalinen, ja
- $O(n!)$.

Vaikka asymptoottiset aikavaatimusten ylärajat ovatkin usein tärkeimmät algoritmien vertailuperusteet, niin on syytä ottaa huomioon, että

- joidenkin algoritmien pahimmat tapaukset esiintyvät käytännön tilanteissa niin harvoin, että voidaan toimia keskimääräisten aikavaatimusten perusteella,
- jos algoritmia tullaan käyttämään harvoin, on ohjelmoinnin helppous tärkeää,
- pienillä n:n arvoilla voi asymptoottisesti tehokas algoritmi toimia heikosti,
- nopea algoritmi voi tarvita enemmän tilaa kuin hitaampi algoritmi ja
- numeerisen algoritmin tarkkuus ja stabiilisuus ovat yhtä tärkeitä kuin nopeus.

Asymptoottisten ylärajojen käyttöön liittyy se selkeä etu, että tulokset ovat riippumattomia toteutuksesta, sillä resurssitarve ilmaistaan jonkun tapaukseen liittyvän suureen avulla. Toisaalta asymptoottinen tulos ei välttämättä kerro mitään algoritmin käyttäytymisestä käytännön tilanteissa, sillä vakio n_0 voi olla niin suuri, että se ylittää kaikki käytännön tilanteissa vastaan tulevat arvot. Asymptoottisille ylärajoille perinteisiä vaihtoehtoisia analysointimenetelmiä ovat

- keskimääräisen tapauksen analyysi,
- suoritettujen operaatioiden tarkan lukumäärän laskeminen ja
- algoritmien kokeellinen analysointi.

Keskimääräinen analyysi tarvitsee tietoa tai oletuksia syötteiden jakaumista, ja se on usein matemaattisesti pahimman tapauksen analyysia selvästi vaativampaa. Suoritettujen operaatioiden tarkka arvo on mielekästä laskea esimerkiksi silloin, kun vertaillaan samaan tarkoitukseen laadittuja toisiaan muistuttavia algoritmeja, joiden toiminta perustuu jonkun yhteisen operaation toistamiseen. Algoritmien kokeellista analyysia voidaan käyttää algoritmien suunnittelun apuna, sillä hyvin suunnitellut kokeet voivat paljastaa algoritmin pullonkaulat. Kokeellista analyysia voidaa käyttää myös silloin, kun muut keinot ovat liian monimutkaisia.

Esimerkki 2.2 Tehtävänä on etsiä n-alkioisesta kokonaislukutaulukosta L arvoa x. Jos x on taulukossa, niin palautetaan sen paikka taulukossa; muutoin palautetaan arvo 0. Tapauksen kooksi voidaan valita alkioiden lukumäärä n. Käytetään kuvan 2.1 algoritmia.

---

(1) $j := 1$
(2) **while** $(j \leq n)$ **and** $(L(j) \neq x)$ **do** $j := j + 1$ **od**
(3) **if** $j > n$ **then** $j := 0$ **end**
(4) **return** j

Kuva 2.1: Etsintä.

Rivit (1) ja (3) suoritetaan kerran, ja molempien suoritukseen kuluu jokin vakioaika. Algoritmin aikavaatimus riippuu siis rivin (2) suorituskerroista. Pahimmassa tapauksessa suorituskertoja on n ja parhaassa tapauksessa 1. Keskimääräinen suorituskertojen

lukumäärä näyttäisi olevan noin $n/2$, josta seuraisi keskimääräisen tapauksen aikavaatimukseksi $O(n)$. Tarkastellaan keskimääräisen tapauksen aikavaatimusta kuitenkin hieman tarkemmin. Tarkoittakoon q todennäköisyyttä, että x on taulukossa L, ja olkoon I_i tapaus, jossa x on taulukon paikassa i. Kun oletetaan, että alkioiden sijainti taulukossa on tasaisesti jakautunut, voidaan vertailujen lukumäärä keskimääräisessä tapauksessa k_{ave} kirjoittaa muodossa

$$\begin{aligned} k_{\text{ave}} &= \sum_{i=1}^{n} p(I_i)i + (1-q)(n+1) = \sum_{i=1}^{n} \frac{q}{n} i + (1-q)(n+1) \\ &= \frac{q}{n} \sum_{i=1}^{n} i + (1-q)(n+1) = \frac{q}{n} \frac{n(n+1)}{2} + (1-q)(n+1) \\ &= \frac{q(n+1)}{2} + (1-q)(n+1) \\ &= \big(1 - \frac{q}{2}\big)(n+1). \end{aligned}$$

Jos tiedetään, että x on taulukossa (ts. $q = 1$), niin $k_{\text{ave}} = (n+1)/2$. Jos taas esimerkiksi $q = 1/2$, niin $k_{\text{ave}} = 3(n+1)/4$. □

Ohjelman osien aikavaatimuksille pätevät yleensä seuraavat yksinkertaiset nyrkkisäännöt:

- sijoitus- ja tulostuskäskyt voidaan suorittaa vakioajassa
- ehtolauseen aikavaatimus on ehdon ja hitaamman haaran aikavaatimusten summa
- silmukan aikavaatimus on suorituskertojen lukumäärä kertaa lopetusehdon testauksen ja silmukan rungon aikavaatimusten summa.

Jos sijoituskäsky tai vaikkapa yhteenlasku ajatellaan suoritettavan vakioajassa riippumatta operaation kohteena olevien lukujen suuruudesta (niiden esittämiseen tarvittavien bittien lukumäärästä), käytetään niin sanottua yksikkökustannusta (unit cost). Näin tehdään tässä esityksessä. Tarkempia tuloksia saataisiin ns. logaritmisella kustannusperiaatteella, jossa jokaisessa operaatiossa huomioidaan se, kuinka paljon bittejä on käsiteltävä operaation toteuttamiseksi. Logaritminen kustannusperiaate estää sellaiset keinotekoiset konstruktiot, joissa rakenteita "koodataan" hyvin suuriksi luvuiksi, joita sitten käsitellään esimerkiksi aritmeettisilla operaatioilla.

2.2. Differenssiyhtälöiden ratkaisemisesta

Tarkastellaan kuvassa 2.2 esitettyä lomituslajittelua. Oletetaan, että funktio merge(L_1, L_2) lomittaa lajitellut listat L_1 ja L_2 yhdeksi lajitelluksi listaksi ajassa $O(|L_1| + |L_2|)$. (Merkintä $|L|$ tarkoittaa alkioiden lukumäärää listassa L.)

```
    Procedure sort (i, j):
(1) if i = j then return x_i
(2) else
(3)         m := (i + j − 1)/2;
(4)         return merge(sort(i, m), sort(m + 1, j))
(5) end
```

Kuva 2.2: Lomituslajittelu.

Listan alkuperäinen sisältö ei vaikuta merge-kutsujen lukumäärään, joten aikavaatimukset pahimmassa, keskimääräisessä ja parhaassa tapauksessa ovat samaa suuruusluokkaa. Aikavaatimus $T(n)$ voidaan esittää differenssiyhtälönä

$$T(n) \leq \begin{cases} c_1, & \text{kun } n = 1, \\ 2T(n/2) + c_2 n, & \text{kun } n > 1. \end{cases}$$

Differenssiyhtälön osaa $T(1) \leq c_1$ sanotaan reunaehdoksi. Alemmassa osassa $2T(n/2)$ ilmoittaa rekursiivisiin kutsuihin kuluvan ajan ja $c_2 n$ kaikkeen muuhun laskentaan (vertailut, listan jakaminen ja yhdistäminen) kuluvan ajan.

Kun oletetaan, että n on kakkosen potenssi, niin ongelman "jako menee tasan". Myöhemmin huomataan, että tämä oletus ei loukkaa tarkastelun yleisyyttä.

Tavoitteena on esittää $T(n)$ "suljetussa muodossa" eli n:n ja vakioiden avulla. Yksinkertainen tapa differenssiyhtälön ratkaisemiseksi on arvata oikea ratkaisu, ja sen jälkeen varmistua arvauksen oikeellisuudesta.

Differenssiyhtälön ratkaisu arvaukseen perustuvalla menetelmällä alkaa yleensä sillä, että lasketaan ratkaisu joillain pienillä tapauksilla, ja pyritään päättelemään ratkaisun yleinen muoto. Arvatun ratkaisun oikeellisuuden todistamiseen käytetään tavallisesti induktiota.

Arvataan nyt, että $T(n) \leq an \log n + b$ joillain vakioilla a ja b, ja otetaan tehtäväksi osoittaa arvaus induktion avulla oikeaksi.

Kun $n = 1$, niin $c_1 \leq a \cdot 1 \cdot \log 1 + b = b$. Vakion b on siis oltava vähintään yhtä suuri kuin c_1. Tehdään induktio-oletus: $T(k) \leq ak \log k + b$, kun $k < n$, ja väitetään, että

$$T(n) \leq an \log n + b.$$

Voidaan kirjoittaa

$$\begin{aligned} T(n) &\leq 2T(n/2) + c_2 n \leq 2(a\frac{n}{2} \log \frac{n}{2} + b) + c_2 n \\ &= an \log \frac{n}{2} + 2b + c_2 n = an(\log n - \log 2) + 2b + c_2 n \\ &= an \log n - an + 2b + c_2 n. \end{aligned}$$

Induktiotodistus olisi valmis, jos viimeksi kirjoitettua lauseketta voitaisiin arvioida ylöspäin lausekkeella $an \log n + b$. Tämä on mahdollista, jos vakio a valitaan sopivasti. Selvitetään, millainen arvo vakiolle a on annettava. Epäyhtälö

$$an \log n - an + 2b + c_2 n \leq an \log n + b$$

on voimassa, jos

$$-an + b + c_2 n \leq 0 \quad \text{eli} \quad a \geq c_2 + b/n.$$

Koska oletettiin, että $n > 1$, niin voidaan valita $a \geq c_2 + b$. Jos siis valitaan $b = c_1$ ja $a = c_1 + c_2$, niin on voimassa

$$T(n) \leq (c_1 + c_2) n \log n + c_1$$

eli lomituslajittelun aikavaatimus on $O(n \log n)$.

Tarkastellaan seuraavana esimerkkinä pikalajittelun (quicksort) keskimääräistä aikavaatimusta. Pikalajittelussahan lajitellaan taulukko $A[1 \ldots j]$ seuraavasti:

1. Valitaan jakoalkioksi jokin lajiteltavista luvuista.
2. Järjestetään taulukko niin, että osataulukko $A[1 \ldots k-1]$ sisältää jakoalkiota pienemmät ja $A[k \ldots j]$ sitä suuremmat tai yhtä suuret alkiot.
3. Jatketaan rekursiivisesti osataulukoista $A[1 \ldots k-1]$ ja $A[k \ldots j]$.

Jako osiin $A[1 \ldots k-1]$ ja $A[k \ldots j]$ voidaan tehdä funktiolla partition, joka esitetään kuvassa 2.3. Kutsun partition(i, j) aikavaatimus on $O(j - i - 1)$.

```
    Function partition (i, j: integer; pivot: alkio): integer
    l, r: integer
(1) l := i, r := j
(2) repeat
(3)         swap(A[l], A[r])
(4)         while A[l].alkio < pivot do l := l + 1 od
(5)         while A[r].alkio ≥ pivot do r := r − 1 od
(6) until l > r
(7) return l
```

Kuva 2.3: Listan ositus.

Jakoalkio voidaan valita monella eri tavalla; oletetaan käytettävän sellaista menettelyä, että etsitään taulukon alusta kaksi eri suurta alkiota ja valitaan niistä suurempi. Jos lajiteltavia lukuja on vähemmän kuin kaksi, niin findpivot palauttaa arvon nolla. Kuvan 2.4 quicksort-aliohjelmassa oletetaan findpivot-funktion palauttavan näin valitun jakoalkion.

Oletetaan, että mikään arvo ei esiinny taulukossa yhtä kertaa useammin ja että kaikki alkujärjestykset ovat yhtä todennäköisiä. Käytetään merkintää $p(i)$ siitä todennäköisyydestä, että partition$(1, n)$ tuottaa osataulukot, joiden koot ovat i ja $n-i$ alkiota. Tällaiset jaot saadaan, kun jakoalkio on suuruusjärjestyksessä $(i+1)$. lajiteltavista luvuista. Todennäköisyys sille, että suuruudeltaan $(i+1)$. luku on taulukon paikassa 1, on $\frac{1}{n}$. Jotta se tulisi valituksi jakoalkioksi, on taulukon toisessa paikassa oltava sitä pienempi luku. Tämän tapahtuman todennäköisyys on $\frac{i}{n-1}$. Todennäköisyys sille, että taulukon ensimmäisessä paikassa on lajiteltavista luvuista suuruudeltaan $(i+1)$. luku ja että siitä tulee

```
     Procedure quicksort (i, j: integer)
     pivot: alkio
     pivotindex: integer              % jakoalkion indeksi
(1)  pivotindex := findpivot(i, j)
(2)  if pivotindex ≠ 0 then
(3)       pivot := A[pivotindex].alkio
(4)       k := partition(i, j,pivot)
(5)       quicksort(i, k − 1)
(6)       quicksort(k, j)
(7)  end
```

Kuva 2.4: Pikalajittelu.

jakoalkio, on $\frac{1}{n}\frac{i}{n-1}$. Toinen mahdollisuus on, että suuruudeltaan $(i+1)$. luku on taulukon paikassa 2, ja paikassa 1 on sitä pienempi luku. Tämänkin tapahtuman todennäköisyys on $\frac{1}{n}\frac{i}{n-1}$. Voidaan siis kirjoittaa

$$p(i) = \frac{1}{n}\frac{i}{n-1} + \frac{i}{n}\frac{1}{n-1} = \frac{2i}{n(n-1)}.$$

Keskimääräiselle ajalle saadaan differenssiyhtälö

$$T(n) = \begin{cases} c_1, & \text{kun } n = 1, \\ \sum_{i=1}^{n-1} p(i)[T(i) + T(n-i)] + c_2 n, & \text{kun } n > 1, \end{cases}$$

eli

$$T(n) = \begin{cases} c_1, & \text{kun } n = 1, \\ \sum_{i=1}^{n-1} \frac{2i}{n(n-1)}[T(i) + T(n-i)] + c_2 n, & \text{kun } n > 1. \end{cases}$$

Muokkaamalla differenssiyhtälöä hieman eteenpäin ja tekemällä sitten arvaus voitaisiin todistaa seuraava lause (todistuksen yksityiskohdat sivuutetaan).

Lause 2.1 *Quicksort*$(1, n)$ *vaatii keskimäärin ajan* $O(n \log n)$.

2.2.1. Differenssiyhtälön purkaminen

Ratkaistaan uudelleen lomituslajittelun aikavaatimusta kuvaava differenssiyhtälö. Nyt oletetaan, että n on muotoa 2^k.

Soveltamalla toistuvasti annettua yhtälöä voidaan kirjoittaa

$$\begin{aligned} T(n) &\leq 2T(n/2) + c_2 n \leq 2(2T(n/4) + c_2 n/2) + c_2 n \\ &\leq 2\big(2(2T(n/8) + c_2 n/4) + c_2 n/2\big) + c_2 n \leq \ldots \end{aligned}$$

Toistamalla tätä i kertaa saadaan

$$T(n) \leq 2^i T(n/(2^i)) + i c_2 n.$$

Koska $n = 2^k$, saadaan erityisesti

$$T(n) \leq 2^k T(1) + kc_2 n \leq c_1 n + c_2 n \log n$$

eli $T(n)$ on $O(n \log n)$.

Yllä olevassa ratkaisussa oletetaan, että $n = 2^k$. Tämä oletus ei loukkaa tarkastelun yleisyyttä. Jos nimittäin oletetaan, että $T(n)$ on kasvava funktio (kuten resurssivaatimusten kohdalla jokseenkin aina on asianlaita), niin voidaan päätellä seuraavasti. Jos n ei ole kakkosen potenssi, niin se on joidenkin kakkosen potenssien välissä: $2^{k-1} < n < 2^k$. Funktion T kasvavuuden perusteella on tällöin $T(2^{k-1}) \leq T(n) \leq T(2^k)$. Edellä todistettiin, että $T(2^k) \leq c2^k \log 2^k$ jollain vakiolla c. Nyt pätee "kakkosen potenssien välissä" olevalle n:lle $T(n) \leq c2^k \log 2^k \leq c(2n)\log(2n) = 2cn \log n + 2cn$. Voidaan siis valita määritelmän mukaiset vakiot myös n:lle eli pätee $T(n) = O(n \log n)$. Koska samanlainen päättely on mahdollinen yleisestikin, voidaan ongelman koon olettaa olevan "sopivaa" muotoa.

Esimerkki 2.3 Tarkastellaan differenssiyhtälöä

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

Sovelletaan toistuvasti annettua yhtälöä. Saadaan

$$\begin{aligned} T(n) &= 2T(n/2) + 2 = 2(2T(n/4) + 2) + 2 = 4T(n/4) + 4 + 2 \\ &= 4(2T(n/8) + 2) + 4 + 2 = 8T(n/8) + 8 + 4 + 2 = \ldots \\ &= 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j. \end{aligned}$$

Koska reunaehto on määritelty arvolla $n = 2$, voidaan purkamista toistaa $k-1$ kertaa. Saadaan

$$\begin{aligned} T(n) &= 2^{k-1} T(n/2^{k-1}) + \sum_{j=1}^{k-1} 2^j = 2^{k-1} T(2) + 2^k - 2 \\ &= 2^{k-1} + 2^k - 2 = 2^k/2 + 2^k - 2 = 2^{\log n}/2 + 2^{\log n} - 2 \\ &= n/2 + n - 2 = 3n/2 - 2. \end{aligned}$$

□

Esimerkki 2.4 Tarkastellaan differenssiyhtälöä

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

Sovelletaan taas toistuvasti annettua yhtälöä. Saadaan

$$\begin{aligned} T(n) &= 2T(n/2) + \log n = 2(2T(n/4) + \log(n/2)) + \log n \\ &= 4T(n/4) + 2\log(n/2) + \log n = 4(2T(n/8) + \log(n/4)) + 2\log(n/2) + \log n \\ &= 8T(n/8) + 4\log(n/4) + 2\log(n/2) + \log n = \ldots \\ &= 2^i T(n/2^i) + \sum_{j=0}^{i-1} 2^j \log(n/2^j). \end{aligned}$$

Kun $i = k = \log n$, saadaan

$$\begin{aligned}
T(n) &= 2^k T(n/2^k) + \sum_{j=0}^{k-1} 2^j \log(n/2^j) = n + \sum_{j=0}^{k-1} 2^j \log 2^{k-j} \\
&= n + \sum_{j=0}^{k-1} 2^j (k-j) = n + k \sum_{j=0}^{k-1} 2^j - j \sum_{j=0}^{k-1} 2^j \\
&= n + k(2^k - 1) - ((k-1)2^{k+1} - k2^k + 2) \\
&= n + k2^k - k - k2^{k+1} + 2^{k+1} + k2^k - 2 \\
&= n + k2^{k+1} - k - k2^{k+1} + 2^{k+1} - 2 = 2^{k+1} - k - 2 + n \\
&= 2n - \log n + n - 2 = 3n - \log n - 2.
\end{aligned}$$

□

Esimerkki 2.5 Tarkastellaan differenssiyhtälöä

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 3T(n/2) + n^2 - n, & \text{kun } n = 2^k \text{ ja } n > 1. \end{cases}$$

Sovelletaan toistuvasti annettua yhtälöä. Saadaan

$$\begin{aligned}
T(n) &= 3T(n/2) + n^2 - n = 3(3T(n/4) + n^2/4 - n/2) + n^2 - n \\
&= 9T(n/4) + 3(n^2/4 - n/2) + (n^2 - n) \\
&= 9(3T(n/8) + n^2/16 - n/4) + 3(n^2/4 - n/2) + (n^2 - n) \\
&= 27T(n/8) + 9(n^2/16 - n/4) + 3(n^2/4 - n/2) + (n^2 - n) = \ldots \\
&= 3^i T(n/2^i) + n^2 \sum_{j=0}^{i-1} (3/4)^j - n \sum_{j=0}^{i-1} (3/2)^j.
\end{aligned}$$

Muokataan ensin kahta jälkimmäistä yhteenlaskettavaa:

$$n^2 \sum_{j=0}^{i-1} (3/4)^j = n^2 \frac{(3/4)^i - 1}{3/4 - 1} = -4n^2((3/4)^i - 1)$$

ja

$$-n \sum_{j=0}^{i-1} (3/2)^j = -2n((3/2)^i - 1).$$

Näiden avulla $T(n)$ voidaan kirjoittaa muodossa

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

Valitaan taas $i = \log n$. Tällöin saadaan

$$\begin{aligned} T(n) &= 3^{\log n} + 4n^2 - 4n^2(3/4)^{\log n} + 2n - 2n(3/2)^{\log n} \\ &= n^{\log 3} + 4n^2 + 2n - 4n^2\frac{n^{\log 3}}{n^{\log 4}} - 2n\frac{n^{\log 3}}{n^{\log 2}} \\ &= n^{\log 3} + 4n^2 + 2n - 4n^{\log 3} - 2n^{\log 3} \\ &= 4n^2 + 2n - 5n^{\log 3}. \end{aligned}$$

□

2.2.2. Erään yhtälöluokan yleinen ratkaisu

Tarkastellaan differenssiyhtälöä

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ aT(n/b) + d(n), & \text{kun } n > 1, \end{cases}$$

kun a ja b ovat vakioita ja d jokin funktio.

Otetaan tehtäväksi todistaa, että tarkasteltavaa muotoa olevan yhtälön ratkaisulla on esitys

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}), \text{ kun } n = b^k.$$

Termiä a^k sanotaan ratkaisun homogeeniseksi osaksi ja summalauseketta epähomogeeniseksi osaksi. Soveltamalla differenssiyhtälön purkamismenettelyä saadaan

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) = a\left(aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right) + d(n) \\ &= a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) = a^2\left(aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right) + ad\left(\frac{n}{b}\right) + d(n) \\ &= a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) = \ldots \\ &= a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right). \end{aligned}$$

Huomioimalla, että $n = b^k$, voidaan viimeinen lauseke kirjoittaa muodossa

$$a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}), \text{ kun } n = b^k.$$

Funktiota d sanotaan *multiplikatiiviseksi*, jos $d(nm) = d(n)d(m)$. Esimerkiksi funktio $d_1(n) = n^\alpha$ on multiplikatiivien, sillä $(nm)^\alpha = (n)^\alpha(m)^\alpha$. Sen sijaan funktio $d_2(n) = \log n$ ei ole multiplikatiivinen, sillä $d_2(nm) = \log nm \neq \log n \log m = d_2(n)d_2(m)$.

Edellä saatu tulos voidaan vielä kirjoittaa yksinkertaisempaan muotoon, jos tiedetään, että d on multiplikatiivinen funktio. Tällöin on voimassa

$$\sum_{j=0}^{k-1} a^j d(b^{k-j}) = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = d(b)^k \frac{\left(\frac{a}{d(b)}\right)^k - 1}{\frac{a}{d(b)} - 1} = \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1},$$

kun $a \neq d(b)$. Tämän esityksen avulla voidaan todistaa seuraava käyttökelpoinen lause.

Lause 2.2 *Olkoon d multiplikatiivinen funktio yhtälössä*

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ aT(n/b) + d(n), & \text{kun } n > 1, \end{cases}$$

kun a ja b ovat vakioita. Tällöin

$$T(n) = \begin{cases} O\big(n^{\log_b a}\big), & \text{jos } a > d(b), \\ O\big(n^{\log_b d(b)}\big), & \text{jos } a < d(b), \\ O\big(n^{\log_b d(b)} \log_b n\big), & \text{jos } a = d(b). \end{cases}$$

Todistus. Kun $a > d(b)$, niin

$$\frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}$$

on $O(a^k)$. Tällöin $T(n)$ on $O(a^k)$ eli $O(n^{\log_b a})$. Kun $a < d(b)$, niin

$$\frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}$$

on $O(d(b)^k)$. Tällöin $T(n)$ on $O(d(b)^k)$ eli $O(n^{\log_b d(b)})$. Kun $a = d(b)$, niin multiplikatiivisuusoletuksen jälkeistä päättelyä ei voida tehdä. Tällöin voidaan kuitenkin kirjoittaa $\sum_{j=0}^{k-1} a^j d(b^{k-j}) = d(b)^k \sum_{j=0}^{k-1} 1 = d(b)^k k = n^{\log_b d(b)} \log_b n$ ja $T(n)$ on $O\big(n^{\log_b d(b)} \log_b n\big)$. □

Esimerkki 2.6 Tarkastellaan yhtälöitä

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 4T(n/2) + n, & \text{kun } n > 1, \end{cases}$$

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 4T(n/2) + n^2, & \text{kun } n > 1 \end{cases}$$

ja

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 4T(n/2) + n^3, & \text{kun } n > 1. \end{cases}$$

Kaikissa yhtälöissä on $a = 4$ ja $b = 2$, joten ratkaisun homogeeninen osa on $a^{\log_b n} = n^{\log_b a} = n^2$.

Ylimmässä yhtälössä on $d(b) = 2 < 4 = a$, joten $T(n) = O(n^{\log_2 4}) = O(n^2)$. Vastaavasti keskimmäisessä yhtälössä on $d(b) = 2^2 = 4 = a$ ja $T(n) = O(n^{\log_2 4} \log_2 n) = O(n^2 \log n)$. Alimmassa yhtälössä on puolestaan $d(b) = 2^3 > 4 = a$ ja $T(n) = O(n^3)$. □

Esimerkki 2.7 Ratkaistaan yhtälö

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 2T(n/2) + n \log n, & \text{kun } n > 1. \end{cases}$$

Tässä yhtälössä on $a = b = 2$. Yhtälön homogeeninen osa on $a^k = n^{\log_b a} = n^{\log_2 2} = n$. Funktio $d(n) = n \log n$ ei ole multiplikatiivinen, joten Lausetta 2.2 ei voida soveltaa. Epähomogeeninen osa on

$$\begin{aligned}\sum_{j=0}^{k-1} a^j d(b^{k-j}) &= \sum_{j=0}^{k-1} 2^j 2^{k-j} \log 2^{k-j} = 2^k \sum_{j=0}^{k-1} (k-j) \\ &= 2^{k-1} k(k+1) = 2^{\log n - 1} \log n(\log n + 1) = \frac{n}{2}(\log^2 n + \log n).\end{aligned}$$

Epähomogeeninen osa määrää siis yhtälön $T(n)$ kasvuvauhdin. $T(n)$ on $O(n \log^2 n)$. □

Tämän alakohdan alussa annetun differenssiyhtälön ratkaisu voidaan edellä esitetyn perusteella antaa homogeenisen ja epähomogeenisen osan summana muodossa

$$a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}).$$

Kun differenssiyhtälö esittää rekursiivisen algoritmin aikavaatimusta, niin homogeeninen osa vastaa osaongelmien ratkaisuun kuluvaa aikaa ja epähomogeeninen osa "muuhun laskentaan" (ongelman jakaminen, ratkaisun kokoaminen osista) kuluvaa aikaa. Kun pyritään tehostamaan algoritmia, on tiedettävä, kumpi osa määrää aikavaatimuksen. Jos esimerkiksi homogeeninen osa on määräävä, ei kannata tehostaa ratkaisun kokoamista osista, vaan algoritmin tehostamiseksi ongelma on jaettava osiin toisella tavalla.

Esimerkki 2.8 Lomituslajittelun aikavaatimusta kuvaa differenssiyhtälö

$$T(n) \leq \begin{cases} c_1, & \text{kun } n = 1, \\ 2T(n/2) + c_2 n, & \text{kun } n > 1. \end{cases}$$

Tämän alakohdan merkinnöillä on siis $a = b = 2$ ja $d(n) = c_2 n$. Homogeeniseksi osaksi saadaan $2^{\log n} = n$ ja epähomogeeniseksi osaksi

$$\sum_{j=0}^{k-1} 2^j c_2 2^{k-j} = 2^k \sum_{j=0}^{k-1} c_2 = c_2 2^k k = O(n \log n).$$

Algoritmin tehostamiseksi pitäisi pystyä tehostamaan lomitusvaihetta. Tämä ei selvästikään ole mahdollista. Myöhemmin huomataankin, että lomituslajittelu on asymptoottisesti optimaalinen eräässä laajassa lajittelumenetelmien luokassa. □

Edellisen esimerkin differenssiyhtälö ei ole Lauseen 2.2 vaatimaa muotoa, sillä $d(n) = c_2 n$ ei ole multiplikatiivinen funktio. Vakiolla c_2 ei kuitenkaan ole vaikutusta epähomogeenisen osan asymptoottiseen käyttäytymiseen. Tämä pitää yleisestikin paikkansa: funktion d vakiokerroin voidaan unohtaa, jos ratkaisuksi riittää asymptoottinen tulos. Lause 2.2 voidaan siis kirjoittaa uudelleen hieman yleisemmässä muodossa seuraavan lauseen tapaan.

Lause 2.3 *Olkoon d multiplikatiivinen funktio yhtälössä*

$$T(n) = \begin{cases} 1, & kun\ n = 1, \\ aT(n/b) + cd(n) + e, & kun\ n > 1, \end{cases}$$

kun a, b, c ja e ovat vakioita. Tällöin

$$T(n) = \begin{cases} O\big(n^{\log_b a}\big), & jos\ a > d(b), \\ O\big(n^{\log_b d(b)}\big), & jos\ a < d(b), \\ O\big(n^{\log_b d(b)} \log_b n\big), & jos\ a = d(b). \end{cases}$$

2.2.3. Lavennusmenetelmä

Tarkastellaan esimerkkinä differenssiyhtälöä

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 2T(n-1) + n, & \text{kun } n > 1. \end{cases}$$

Eri n:n arvoilla saadaan seuraavat yhtälöt

$$\begin{aligned} T(n) &= 2T(n-1) + n \\ T(n-1) &= 2T(n-2) + n - 1 \\ &\dots \\ T(n-i) &= 2T(n-i-1) + n - i \\ &\dots \\ T(2) &= 2T(1) + 2. \end{aligned}$$

Valitaan laventajiksi $2^0, 2^1, \dots, 2^i, \dots, 2^{n-2}$ ja kerrotaan saadut yhtälöt puolittain laventajilla. Saadaan uudet yhtälöt

$$\begin{aligned} T(n) &= 2T(n-1) + n \\ 2T(n-1) &= 2 \cdot 2T(n-2) + 2(n-1) \\ &\dots \\ 2^{n-i}T(n-i) &= 2^{n-i}2T(n-i-1) + 2^{n-i}(n-i) \\ &\dots \\ 2^{n-2}T(2) &= 2^{n-2}2T(1) + 2^{n-2}2. \end{aligned}$$

Laskemalla nämä yhtälöt puolittain yhteen saadaan

$$T(n) = 2^{n-1}T(1) + \sum_{i=0}^{n-2}(n-i)2^i = \sum_{i=0}^{n-1}(n-i)2^i.$$

Sieventämällä tämä lauseke saadaan muotoon $T(n) = 2^{n+1} - n - 2$. (Miten?)

2.3. Tasoitettu vaativuusanalyysi

Tarkastellaan sovellusta, jossa tehdään jono peräkkäisiä operaatioita samaan tietorakenteeseen ja halutaan tietää koko operaatiojonon suoritusaika. Operaatiojonon suoritusaikaa voidaan arvioida yksittäisten operaatioiden pahimpien tapausten perusteella. Näin saatava arvio voi kuitenkin olla liian pessimistinen, sillä pahin tapaus ei ehkä voikaan toistua kerrasta toiseen. Toinen perinteinen tapa on käyttää keskimääräisen tapauksen analyysiä. Ongelmana on tällöin sopivien todennäköisyysjakaumien löytäminen.

Esimerkki 2.9 Tarkastellaan pinoa, jota käsitellään T-operaatioilla, jotka muodostuvat k:sta ($k \geq 0$) *pop*-kutsusta ja yhdestä *push*-kutsusta. Mikä on aikavaatimus, kun tehdään m kappaletta T-operaatioita alunperin tyhjään pinoon? (Yhden pop- tai push-kutsun aikavaatimus on 1 yksikköä.) Yksittäisen T-operaation pahimman tapauksen aikavaatimus on m. Pahimman tapauksen aikavaatimus antaa siis koko operaatiojonon aikavaatimukseksi $O(m^2)$. Sellaisia T-operaatioita, jotka tarvitsevat aikaa m yksikköä, ei m:n operaation jonossa voi kuitenkaan olla kuin yksi kappale. Kun tehdään m kappaletta T-operaatioita, niin pinoon viedään yhteensä m alkiota. Myös pop-kutsuja voi tällöin olla korkeintaan m kappaletta. Operaatiojonon ajantarve on siis korkeintaan $2m$. □

Tasoitettu vaativuusanalyysi pyrkii johtamaan koko jonon vaatiman ajan analysoimalla "sopivasti" yksittäisiä operaatioita. Tasoitettua vaativuusanalyysiä voidaan tarkastella ainakin kahdesta erilaisesta, keskenään samanarvoisesta näkökulmasta.

Pankkiirin näkemys. Kutakin operaatiojonon operaatiota kohti on käytettävissä n yksikköä. Jos operaatio tarvitsee vähemmän aikaa, niin käyttämätön aika jää säästöön. Jos n yksikköä ei riitä, niin käytetään säästöjä tai otetaan lainaa. Jos saldo on lopussa positiivinen, niin yhden operaation tasoitettu aikavaatimus on n.

Esimerkki 2.10 Kullakin T-operaatiolla on käytössään 2 yksikköä. Toisella yksiköllä maksetaan push, toisella maksetaan jokin pop tai pannaan se säästöön. Koska pop-kutsuja on korkeintaan yhtä paljon kuin push-kutsuja, niin säästöt riittävät pop-kutsujen maksamiseen. Yhden T-operaation tasoitettu aikavaatimus on 2. □

Fyysikon näkemys. Ajatellaan, että tietorakenteen jokaiseen tilanteeseen D liittyy reaaliluku $\Phi(D)$, jota sanotaan potentiaaliksi. Jos suoritetaan helppoja operaatioita, niin potentiaali kasvaa; kalliit operaatiot pienentävät potentiaalia. Operaation i tasoitettu aika a määritellään yhtälöllä $a = t + \Phi(D') - \Phi(D)$, kun

t = todellinen aika,
$\Phi(D)$ = potentiaali ennen operaatiota ja
$\Phi(D')$ = potentiaali operaation jälkeen.

Tarkastellaan operaatiojonoa, jossa on m operaatiota. Lasketaan yhteen operaatioiden tasoitetut ajat:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0,$$

kun Φ_0 on potentiaali operaatiojonon alkaessa ja Φ_m sen loppuessa. Todellinen aika voidaan kirjoittaa muodossa

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m.$$

Jos $\Phi_0 - \Phi_m \leq 0$ eli $\Phi_0 \leq \Phi_m$, niin tasoitettua aikaa voidaan käyttää arvioitaessa todellista aikaa ylöspäin. Tämä on mahdollista esimerkiksi silloin, kun valitaan potentiaali niin, että $\Phi_0 = 0$ ja $\Phi_i \geq 0$, kun $i > 0$. Tämä vastaa pankkiirin näkemystä ilman lainanottoa.

Esimerkki 2.11 Valitaan pinoesimerkissä pinon potentiaaliksi, $\Phi(\text{pino})$, pinossa olevien alkioiden lukumäärä. Jos pinossa on r alkiota ja suoritetaan T-operaatio, joka tekee k pop-operaatiota, niin

$$a = t + \Phi(\text{pino}') - \Phi(\text{pino}) = (k+1) + (r - (k-1)) - r = 2.$$

□

Mukautuva lista on esimerkki ns. mukautuvista tietorakenteista, joita tarkastellaan jatkossa tarkemmin. Tarkastelun tarkoituksena on selventää perinteisen vaativuusanalyysin ja tasoitetun vaativuusanalyysin välistä eroa.

Sanakirjaongelmassa on tehtävänä ylläpitää alkiojoukkoa, jota käsitellään operaatioilla

- access(x): etsi alkio x joukosta,
- insert(x): lisää alkio x joukkoon ja
- delete(x): poista alkio x joukosta.

Sanakirjaongelma voidaan ratkaista tehokkaasti käyttämällä erilaisia binaaripuita. Seuraavassa tarkastellaan kuitenkin ongelman ratkaisemista järjestämättömän linkitetyn listan avulla.

Alkio etsitään listasta käymällä listaa läpi alusta lähtien. Alkiota lisättäessä varmistaudutaan aluksi siitä, että alkio ei vielä ole listassa. Alkion poistamiseksi se aluksi etsitään kuten operaatiossa access.

Operaatioiden kustannuksista oletetaan seuraavaa:

- listan i:nnen alkion etsintä tai poisto maksaa i yksikköä,
- alkion lisäys maksaa $i+1$ yksikköä, jos listan pituus ennen lisäystä on i,
- operaatio insert(x) voi lisätä x:n mihin tahansa kohtaan listaa ilman lisäkustannuksia,
- operaation access(x) yhteydessä voidaan x siirtää ilmaiseksi mihin tahansa kohtaan lähemmäksi listan alkua ja
- kaikki muut peräkkäisten alkioiden vaihdot maksavat yhden yksikön.

Jos listan alkioihin kohdistuvat viittaukset jakautuisivat kaikkien alkioiden kesken tasaisesti, ei alkioiden järjestyksellä olisi merkitystä. Tämä on käytännössä kuitenkin hyvin harvinaista. Toisaalta, jos alkioiden viittaustodennäköisyydet tunnettaisiin etukäteen, niin alkiot kannattaisi tietenkin järjestää viittaustodennäköisyyksien mukaan laskevaan järjestykseen. Jatkossa oletetaan, että viittaustodennäköisyyksiä ei tunneta etukäteen.

Kirjallisuudessa on tutkittu mm. seuraavia järjestelyheuristiikkoja:

- *move-to-front* (MF), jossa etsitty ja lisätty alkio siirretään listan alkuun,
- *transpose* (TR), jossa etsityn ja sitä edeltävän alkion paikat vaihdetaan; lisäykset tehdään listan alkuun ja
- *frequency count* (FC), jossa pidetään yllä viittauslaskureita ja pidetään alkiot laskureiden mukaan laskevassa suuruusjärjestyksessä.

Otetaan käyttöön merkintä p_x alkion x viittaustodennäköisyydelle jakaumassa p ja $E_A(p)$ jakaumaan p liittyvälle etsintäkustannusten odotusarvolle, kun käytetään heuristiikkaa A. Järjestelyheuristiikkoja verrataan usein optimaaliseen staattiseen järjestykseen, josta käytetään merkintää DP (*decreasing probability*).

Perinteisesti heuristiikkoja on vertailtu etsintäkustannusten odotusarvojen perusteella; tämä vastaa keskimääräisen tapauksen analyysia). On pystytty todistamaan mm. seuraavat tulokset kaikille jakaumille p:

- $E_{\mathrm{FC}}(p) = E_{\mathrm{DP}}(p)$,
- $E_{\mathrm{MF}}(p) \leq 2E_{\mathrm{DP}}(p)$ ja
- $E_{\mathrm{TR}}(p) \leq E_{\mathrm{MF}}(p)$ ja yhtäsuuruus on voimassa vain, kun listassa on kaksi alkiota tai $p_x = 1/n$ kaikilla alkioilla x.

Käytännössä on kuitenkin huomattu, että MF toimii yleensä paremmin kuin TR. Tämä perustuu siihen, että MF:n kustannus suppenee kohti odotusarvoaan paljon nopeammin kuin TR:n kustannus.

Seuraavassa tarkastellaan listaheuristiikkojen tasoitettuja aikavaatimuksia. Oletetaan aluksi, että alkiojoukko on kiinteä, ts. operaatiojono muodostuu pelkästään access-operaatioista. Operaatiojonon S vaatimasta ajasta sovellettaessa heuristiikkaa A käytetään merkintää $C_A(S)$.

Lause 2.4 *Kaikilla operaatiojonoilla S on $C_{MF}(S) \leq 2C_{DP}(S)$.*

Todistus. Kaikkien heuristiikkojen aikavaatimus muodostuu ei-täsmäävistä ja täsmäävistä vertailuista listan alkioiden ja etsittävän alkion välillä. Täsmääviä vertailuja on kaikilla heuristiikoilla yhtä monta. Riittää siis osoittaa, että MF-heuristiikka tekee korkeintaan kaksi kertaa niin paljon ei-täsmääviä vertailuja kuin DP. Tämä voidaan todistaa

erikseen jokaiselle alkioparille (A, B). MF-heuristiikan tekemät ei-täsmäävät vertailut alkioiden A ja B välillä riippuvat pelkästään A:n ja B:n keskinäisestä järjestyksestä listassa. Alkioiden keskinäinen järjestys puolestaan riippuu siitä, kumpaa on viimeksi etsitty; viittaukset muihin alkioihin eivät siihen vaikuta.

Oletetaan, että operaatiojonossa S on m kappaletta access(A)- ja n kappaletta access(B)-operaatioita. Yleisyyttä loukkaamatta voidaan olettaa, että $m \leq n$.

DP-listassa alkio B on ennen alkiota A; ei-täsmääviä (A, B)-vertailuja tehdään siis m kappaletta. Jokaiseen MF:n ei-täsmäävään vertailuun liittyy A:n ja B:n järjestyksen vaihtuminen. Tällaisia vaihtoja voi olla korkeintaan $2m$. □

Lauseen 2.4 raja on tiukka, ts. sitä ei voi parantaa. Tämän osoittamiseksi tarkastellaan aluksi listaa, jossa on neljä alkiota. Operaatiojonolla $S = (ABCD)^t$ on $C_{\mathrm{DP}}(S) = (1 + 2 + 3 + 4)t$. MF-heuristiikan mukainen kustannus riippuu listan alkusisällöstä. Jos lista on aluksi järjestyksessä $D - C - B - A$, niin $C_{\mathrm{MF}}(S) = (4 + 4 + 4 + 4)t = 16t$. Yleisesti voi olla $C_{\mathrm{DP}}(S) = t \sum_{i=1}^{k} i$ ja $C_{\mathrm{MF}}(S) = tk^2$. Kustannusten suhteeksi saadaan

$$\frac{C_{\mathrm{MF}}(S)}{C_{\mathrm{DP}}(S)} = \frac{tk^2}{t \sum_{i=1}^{k} i} = \frac{k^2}{k(k+1)/2} = \frac{2k}{k+1}.$$

Kun k kasvaa, niin suhde lähestyy kahta, ja valitsemalla k tarpeeksi suureksi päästään mielivaltaisen lähelle kakkosta.

TR-heuristiikalle ei voida todistaa Lauseen 2.4 kaltaista tulosta. Tämä nähdään tarkastelemalla listaa, jonka alkusisältö on $A - B - C - D - E$. Operaatiojonon $(ED)^t$ tuloksena vaihtavat alkiot D ja E toistuvasti paikkojaan, mutta eivät etene kohti listan alkua. Operaatiojonon kustannus on $10t$. Optimaalisessa staattisessa järjestyksessä D ja E ovat listan alussa ja kustannus on $(1 + 2)t$; on siis $C_{\mathrm{TR}}(S) > 2C_{\mathrm{DP}}(S)$.

Seuraavassa tarkastellaan dynaamisia alkiojoukkoja. Otetaan tehtäväksi todistaa Sleatorin ja Tarjanin vahva tulos, joka korvaa DP-heuristiikan lauseessa 2.4 mielivaltaisella heuristiikalla.

Otetaan käyttöön seuraavat merkinnät:

$F_A(S)$	= ilmaisten vaihtojen lukumäärä insert- ja access-operaatioiden yhteydessä,
$X_A(S)$	= maksullisten vaihtojen lukumäärä,
$C_A(S)$	= kokonaiskustannus ilman maksullisia vaihtoja ja
m	= operaatiojonon pituus.

Koska MF-heuristiikka tekee vain ilmaisia vaihtoja, on $X_{\mathrm{MF}}(S) = 0$ kaikilla operaatiojonoilla S.

Lause 2.5 *Lähdettäessä tyhjästä listasta kaikilla edellä määriteltyjen laskutussääntöjen mukaisilla heuristiikoilla A ja kaikilla operaatiojonoilla S, joiden pituus on m, pätee*

$$C_{MF}(S) \leq 2C_A(S) + X_A(S) - F_A(S) - m.$$

Todistus. Todistuksessa tarkastellaan rinnakkain kahta listaa. Toista näistä ylläpidetään MF-heuristiikalla ja toista kilpailevalla heuristiikalla A. Listoihin tehdään aina täsmälleen

samat operaatiot. Käytetään listoista vastaavasti nimityksiä MF-lista ja A-lista. Valitaan potentiaalifunktioksi listoissa olevien inversioiden lukumäärä; inversiolla tarkoitetaan sellaista alkioparia (i, j), jonka alkioiden keskenäinen järjestys on erilainen MF-listassa kuin A-listassa. Alkutilanteessa molemmat listat ovat tyhjiä, joten potentiaali on aluksi 0. Koska potentiaali on lukumäärä, niin se on aina ei-negatiivinen. Tasoitettu aika arvioi siis todellista aikaa ylöspäin.

Olkoon i operaation kohteena olevan alkion indeksi A-listassa ja c_A operaation kustannus A-listassa. Lause tulee todistettua, kun näytetään, että MF-listassa tasoitetut aikavaatimukset eri operaatioille ovat seuraavat:

access: $2i - 1 = 2c_A - 1$
insert: $2(i + 1) - 1 = 2c_A - 1$
delete: $i = c_A \leq 2c_A - 1$
ilmainen vaihto A-listassa: -1
maksettu vaihto A-listassa: 1.

Väitteen epäyhtälön oikean puolen termit $2C_A(S)$ ja $-m$ saadaan kolmesta ensimmäisestä operaatiosta. Termit $X_A(S)$ ja $-F_A(S)$ puolestaan saadaan kahdesta jälkimmäisestä operaatiosta. Tarkastellaan erikseen kutakin operaatiota.

Tarkastellaan aluksi access-operaatiota, joka kohdistuu A-listan siihen alkioon, joka on paikassa i. Oletetaan, että tämä alkio on MF-listassa paikassa k. Oletetaan edelleen, että x_i on niiden alkioiden lukumäärä, jotka edeltävät etsittyä alkiota MF-listassa, mutta ovat sen jäljessä A-listassa. Tästä seuraa, että yhteisiä edeltäjiä etsityillä alkioilla on $k-1-x_i$ kappaletta. MF-listan heuristiikka siirtää etsityn alkion listansa alkuun. Tämä aiheuttaa potentiaalin muutoksen, sillä osa inversioista poistuu, mutta toisaalta uusia inversioita tulee tilalle. Poistuvien inversioiden lukumäärä on x_i, ja uusia inversioita syntyy $k-1-x_i$ kappaletta. Nyt voidaan määrätä tasoitettu aika MF-listassa (merkintä $\Delta\Phi$ tarkoittaa potentiaalin muutosta):

$$a = t + \Delta\Phi = k + (k - 1 - x_i) - x_i = 2k - 2x_i - 1 = 2(k - x_i) - 1 \leq 2i - 1.$$

Viimeinen epäyhtälö pätee, sillä yhteisiä edeltäjiä voi etsityillä alkioilla olla korkeintaan $i - 1$ ja $k - x_i \leq i$.

Seuraavaksi tarkastellaan A-listassa tehtävää ilmaista vaihtoa. Se ei tietenkään maksa mitään MF-listalle, joten todellinen aika on 0. Yleisten laskutussääntöjen mukaisesti A-lista voi tehdä ilmaisen vaihdon vain jonkun insert- tai access-operaation yhteydessä. MF-heuristiikka siirtää kyseisen alkion aina oman listansa ensimmäiseksi. A-listan tekemä ilmainen vaihto poistaa siis yhden inversion, joten potentiaali pienenee yhdellä. Vastaavasti A-listan maksettu vaihto ei maksa mitään MF-listalle, mutta se voi kasvattaa potentiaalia yhdellä.

Insert-operaatiossa on aluksi tutkittava, onko lisättävä alkio jo listassa. Täten todellinen aika molemmissa listoissa on sama. Riippuen siitä, mihin kohtaan A-lista lisää uuden alkion, syntyy jokin määrä uusia inversioita. Suurimmillaan tämä määrä on i (= listojen pituus ennen uuden alkion lisäystä). Tasoitetulle ajalle a siis pätee

$$a \leq i + 1 + i = 2i + 1 = 2(i + 1) - 1 = 2c_A - 1.$$

Vielä on käsittelemättä delete-operaation tapaus. Oletetaan, kuten access-operaation kohdalla, että operaatio kohdistuu siihen A-listan alkioon, joka on paikassa i, ja että tämä alkio on MF-listassa paikassa k. Edelleen (kuten access-operaationkin kohdalla) oletetaan, että x_i on niiden alkioiden lukumäärä, jotka edeltävät kyseistä alkiota MF-listassa, mutta ovat sen jäljessä A-listassa. Alkion poistaminen voi ainoastaan vähentää inversioita; vähennyksen suuruus on x_i. Tasoitettua aikaa voidaan nyt arvioida seuraavasti:

$$a = k - x_i \leq i \leq 2c_A - 1.$$

Näin lauseen todistus on valmis. □

Tarkastellaan johonkin tietorakenteeseen kohdistuvaa operaatiojonoa. Operaatioiden toteutuksen kannalta on ratkaisevaa, tiedetäänkö koko operaatiojono etukäteen vai onko tiedossa kerrallaan vain yksi operaatio, joka on suoritettava loppuun ennen kuin selviää, mikä operaatio on seuraava. Jos koko operaatiojono tiedetään etukäteen, sanotaan jonon toteuttavaa algoritmia offline-algoritmiksi. Jos operaatiot tiedetään yksi kerrallaan, on kyseessä online-algoritmi. Sanakirjaongelman tapauksessa offline-algoritmi voisi toimia esimerkiksi niin, että se laskisi eri alkioihin liittyvien viittausten lukumäärät etukäteen, ja järjestäisi listan kiinteään, viittausfrekvenssien mukaan laskevaan järjestykseen. MF- ja TR-heuristiikat puolestaan toimivat online-periatteella.

Käytetään algoritmin A kustannuksesta ongelmaan P liittyvässä operaatiojonossa I merkintää $c_A(I)$. Olkoon optimaalisen (online- tai offline-algoritmin) kustannus $c_{opt}(I)$. Online-algoritmia A sanotaan *kilpailukykyiseksi* (competitive) ongelmassa P, jos on olemassa sellaiset vakiot a ja b, että kaikilla operaatiojonoilla I on voimassa

$$c_A(I) \leq ac_{opt}(I) + b.$$

Kilpailukykyinen algoritmi toimii siis kaikilla syötteillä korkeintaan a kertaa hitaammin kuin optimaalinen algoritmi, joka voi olla myös offline-algoritmi eli sillä voi olla etukäteen tiedossa koko syöte.

Edellä olevan perusteella tiedetään, että MF-heuristiikka on kilpailukykyinen algoritmi sanakirjaongelman tapauksessa. Kilpailukykyisiä algoritmeja on löytynyt myös monille muille ongelmille, jotka käytännön tilanteissa vaativat online-ominaisuutta. Tällaisia ovat esimerkiksi monet töidenjärjestelyyn ja tietokoneen muistinhallintaan liityvät ongelmat.

Kilpailukykyiset algoritmit ovat käytännössä osoittautuneet tehokkaammiksi kuin perinteisen analyysin perusteella parempina pidetyt algoritmit samaan tapaan kuin MF-heuristiikka on käytännössä yleensä parempi kuin TR-heuristiikka, vaikka odotusarvotarkastelujen perusteella voitaisiinkin olettaa päinvaistaista.

Mielenkiintoinen kilpailukykyisiin algoritmeihin liittyvä piirre on se, että niiden täytyy "oppia" mukautumaan erilaisiin syötteisiin. Muuten ne eivät voisi kaikilla mahdollisilla syötteillä olla vain vakiokertaisesti hitaampia kuin optimialgoritmi.

2.4. Mukautuvista tietorakenteista

Tietorakenteita voidaan yrittää tehostaa uudelleenjärjestelyheuristiikoilla, joita sovelletaan jokaisen alkioviittauksen yhteydessä. Tyypillisesti heuristiikat pyrkivät siirtämään alkion, johon useasti viitataan, helpommin tavoitettavaksi. Tässä kohdassa selvitetään ns. mukautuviin (self-adjusting) tietorakenteisiin liittyviä peruskäsitteitä ja esitetään joitakin esimerkkejä mukautuvista tietorakenteista.

Aikaisemmin pyrittiin kehittämään heuristiikkoja, jotka takaisivat tietorakenteen tehokkuuden keskimääräisessä tai pahimmassa tapauksessa, kun tietorakenteen alkioilla oletetaan olevan kiinteät viittaustodennäköisyydet. Keskimääräistä tapausta käytettäessä on ongelmana sopivan todennäköisyysjakauman löytäminen. Pahimman tapauksen käyttö voi taas johtaa todellista tilannetta huonompaan arvioon, kun tarkastellaan jonoa tietorakenteeseen kohdistuvia operaatioita. Pahimman tapauksen analyysissä saadaan tulokseksi yksityisten operaatioiden pahimpien tapauksien summa, ja huomiotta jätetään ne muutokset, joita operaatiot mahdollisesti tekevät tietorakenteeseen ja näin ehkä helpottavat myöhempiä operaatioita. Erityisesti tämä tulee esiin tutkittaessa mukautuvia tietorakenteita.

Tasoitetussa analyysissä lasketaan keskimääräinen vaativuus yli pahimman tapauksen operaatiojonon. Yksittäisen operaation aikavaatimus voi esimerkiksi olla verrannollinen tietorakenteen alkioiden lukumäärään n, mutta tarkasteltaessa tarpeeksi pitkiä operaatiojonoja operaatioiden keskimääräinen aikavaatimus voikin olla $O(\log n)$. Joissain sovelluksissa on tärkeää, että jokaisen yksittäisen operaation kuluttama aika on pienempi kuin jokin annettu raja. Tällöin mukautuvassa tietorakenteessa joskus tarvittavat "vaikeat" operaatiot voivat olla liian hitaita.

Mukautuville tietorakenteille ovat tyypillisiä seuraavat ominaisuudet:

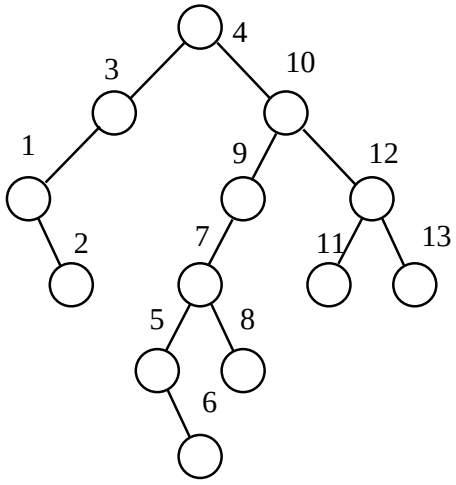
- Ei tallenneta tietoa tietorakenteen tilasta (vrt. AVL-puussa olevat korkeustiedot).
- Jokaisen operaation yhteydessä sovelletaan *yksinkertaista* heuristiikkaa tietorakenteen muuttamiseksi.
- Heuristiikkaa sovelletaan riippumatta tietorakenteen tilasta.
- Koska tilatietoja ei tallenneta, mukautuvat tietorakenteet säästävät tilaa.
- Mukautuvat tietorakenteet ovat usein helppoja toteuttaa.

2.4.1. Levittyvä puu

Move-to-front -heuristiikan vastine järjestyspuita käsiteltäessä on move-to-root -sääntö, joka nostaa juureksi sen solmun, johon viimeksi on viitattu. Se ei kuitenkaan ole tehokas tasoitetussa(kaan) mielessä. On nimittäin olemassa mielivaltaisen pitkiä operaatiojonoja, joiden keskimääräinen aikavaatimus tätä menetelmää käytettäessä on verrannollinen solmujen lukumäärään. Kallis operaatiojono voidaan muodostaa mihin tahansa n-solmuiseen ($n > 3$, n parillinen) järjestyspuuhun seuraavasti: tehdään access-operaatiot järjestyksessä

avaimiin $1, 2, \ldots, \frac{n}{2}, 1, 2, \ldots, \frac{n}{2}$. Tuloksena operaatiojonon alkuosasta on vasemmalle vino puu, joka tekee jonon loppuosan operaatiot kalliiksi. Operaatiojonon kustannus on $> \frac{n^2}{8}$, eli operaatioiden tasoitettu aikavaatimus on $\Omega(n)$.

Levittyvän puun (splay tree) ideana on liittää move-to-root -sääntöön hakupolulla tehtävät rotaatiot. Tarkastellaan esimerkkinä kuvassa 2.5 esitettyä puuta.

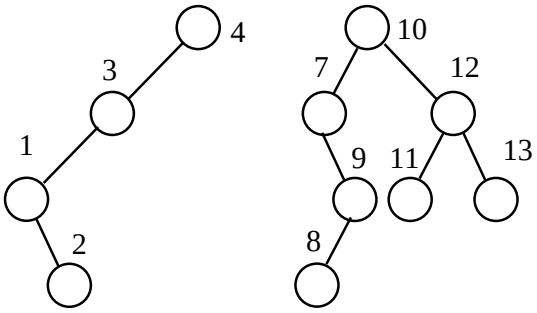


Kuva 2.5: Esimerkkipuu.

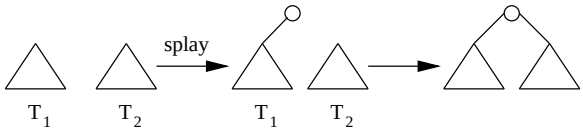
Lähdettäessä juuresta kohti etsittävää solmua (tässä esimerkissä solmu 5) voidaan alkuperäinen puu jakaa kolmeen osapuuhun: etsittävää solmua pienemmät solmut (vasen puu), etsittävää solmua suuremmat solmut (oikea puu) ja toistaiseksi käsittelemättömät solmut. Kuvassa 2.6 on esitetty vasen ja oikea puu siinä tilanteessa, kun ollaan löydetty solmu 5. Hakupolulla edetään kaksi solmua kerrallaan, ja jos molemmat askeleet tehdään oikealle (vastaavasti vasemmalle), suoritetaan rotaatio vasemmalle (vast. oikealle). Esimerkissämme ensimmäiset tavoitettavat solmut ovat 4 ja 10 ja niistä lähtevät linkit hakupolulla ovat eri suuntiin. Seuraavat solmut ovat 9 ja 7, joista kummastakin lähdettäessä seurataan vasempia linkkejä, joten tehdään rotaatio oikealle. Lopuksi hakupolulla saavutetaan etsitty solmu 5. Etsinnän tuloksena olevan puun juuri on etsitty solmu ja sen oikea (vast. vasen) lapsi on oikean (vast. vasemman) puun juuri. Etsityn solmun oikea (vast. vasen) alipuu alkuperäisessä puussa liitetään oikeaan (vast. vasempaan) puuhun "lehtien jatkeeksi" (joten sen paikka on täsmällisesti määrätty). Esimerkissämme siis solmu 6 liitetään oikean puun solmun 7 vasemmaksi lapseksi.

Kaikki levittyvän puun normaalit puuoperaatiot perustuvat access-operaatioon. Puiden T_1 ja T_2 yhdistäminen (join) on mahdollista vain, jos puun T_1 kaikki avaimet ovat pienempiä kuin kaikki puun T_2 avaimet. Tällöin voidaan puussa T_1 tehdä access-operaatio, joka kohdistuu alkioon, jolla on suurin avain. Tällöin kyseinen alkio tulee puun T_1 juureksi, ja sen oikea alipuu jää tyhjäksi. Puu T_2 voidaan lisätä kyseisen juuren oikeaksi alipuuksi. Yhdistäminen on esitetty kuvassa 2.7.

Alkion x suhteen tehtävä puun jakaminen split(x) toteutetaan operaatiolla access(x), joka nostaa alkion x puun juureksi ja mahdollistaa jomman kumman puun alipuun ir-

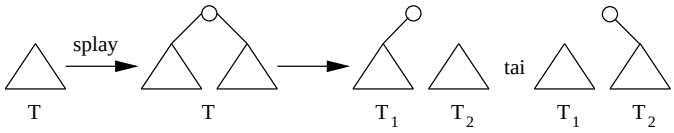


Kuva 2.6: Kuvan 2.5 puuhun liittyvät vasen puu ja oikea puu, kun etsitään alkiota 5.



Kuva 2.7: Levittyvien puiden yhdistäminen.

rottamisen erilliseksi puuksi. Kuvassa 2.8 on esitetty operaation molemmat versiot (x jää joko pienempien tai suurempien avainten muodostaman puun juureksi).



Kuva 2.8: Levittyvän puun operaatio split(x).

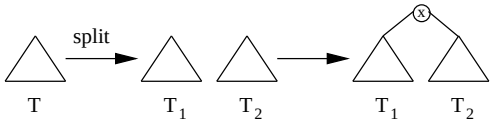
Alkion x lisääminen *insert*(x) aloitetaan tekemällä operaatio access(x). Jos x löytyy puusta, ei sitä voida sinne enää lisätä. Jos x ei ole puussa, päätyy hakupolku x:ää etsittäessä lehteen, josta ei voida enää jatkaa eteenpäin. Kohdistetaan access-operaatio tähän alkioon ja nostetaan se puun juureksi. Puun uusi juuri on alkiota x lähinnä pienempi tai suurempi alkio. Voidaan siis jakaa puu kahteen osaan niin, että osat tulevat sen puun vasemmaksi ja oikeaksi alipuuksi, jonka juureksi tulee x (ks. kuva 2.9).

Alkion x poisto delete(x) aloitetaan tekemällä access(x). Puun juureksi nousseen poistettavan alkion x vasen ja oikea alipuu yhdistetään operaatiolla join. Alkion poisto levittyvästä puusta on esitetty kuvassa 2.10.

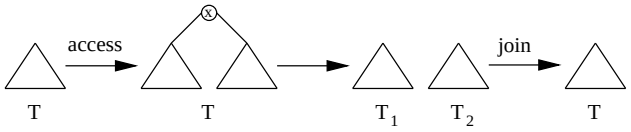
Levitysoperaation tasoitettu aikavaatimus n-solmuisessa puussa on $O(\log n)$. Koska muut operaatiot perustuvat access-operaatioon, riittää tuloksen todistaminen sille.

Todistusta varten kiinnitään levittyvän puun jokaiseen solmuun i kiinteä *paino* $wt(i) > 0$. Solmun i *koko* s_i on alkioiden paino siinä alipuussa, jonka juuri i on. Solmun *aste* $r(i)$ on $\log s(i)$, ja tarkasteltavan levittyvän puun potentiaali on sen solmujen asteiden summa.

Todistuksessa hakupolkua (polku juuresta viitattuun alkioon) tarkastellaan kaksi askelta kerrallaan. Potentiaalin muutoksen laskemiseksi on huomioitava hakupolun solmujen



Kuva 2.9: Levittyvän puun operaatio insert(x).

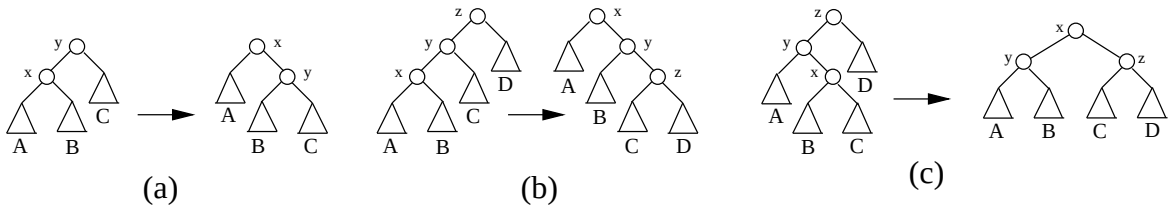


Kuva 2.10: Alkion x poistaminen levittyvästä puusta.

alipuiden muutokset, kun tehdään mukauttamisoperaation kuuluvat rotaatiot.

Jokaiselle kahden askeleen osalle voidaan erikseen todistaa (todistuksen yksityiskohdat löytyvät artikkelista Sleator and Tarjan, Self-adjusting binary search trees, *J. ACM* 32, 3 (1985), 652-686), että jos t on kyseisen hakupolun osan ylin solmu ja v on sen alin solmu, niin tasoitettu aika (todellisen ajan (2 yksikköä) eli hakupolulla edetyn matkan ja potentiaalin muutoksen summa) on korkeintaan

$$3(r(t) - r(v)) + 1 = O(\log \frac{s(t)}{s(v)}).$$



Kuva 2.11: Rotaatiot hakupolulla etsittäessä solmua x. Jokaiselle tapaukselle on olemassa myös symmetrinen vastine.

Tulos saavutetaan, kun esimerkiksi kuvan 2.11 tapauksessa (b) huomioidaan asteiden muutokset solmuissa x, y ja z. Toinen perustapaus on se, jossa kaksi askelta edetessä kuljetaan vuorotellen vasemmalle ja oikealle (kuva 2.11(c)). Erikseen on vielä käsiteltävä se tilanne (kuva 2.11(a)), jossa hakupolun pituus on pariton, ja lopuksi hakupolulla edetään vain yksi särmä. Hakupolkua edetessä solmujen t ja v roolit vaihtuvat niin, että laskettaessa koko hakupolun kustannus mukaan tulevat vain juuren aste $r(t)$ ja etsittävän alkion aste $r(v)$ muiden hakupolun solmujen asteiden supistuessa summasta pois.

Lause 2.6 *Olkoon n-solmuisen levittyvän puun juuri t. Operaation access(v) tasoitettu aikavaatimus on korkeintaan*

$$3(r(t) - r(v)) + 1 = O(\log \frac{s(t)}{s(v)}).$$

Lauseesta 2.6 seuraa, että kaikkien tavanomaisten levittyvään puuhun kohdistuvien operaatioiden (access, insert, delete, join, split) tasoitettu aikavaatimus on $O(\log n)$.

Lause 2.6 on voimassa kaikilla painojen $wt(i) > 0$ valinnoilla. Tarkastellaan tilannetta, jossa m access-operaatiota kohdistetaan levittyvään puuhun, jossa on n solmua ja jokaisen solmun paino on $\frac{1}{n}$. Tällöin solmun suurin mahdollinen koko on $s_+ = 1$ ja pienin mahdollinen koko on $s_- = \frac{1}{n}$. Lauseen 2.6 mukaan yhden operaation tasoitettu aika on korkeintaan

$$\begin{aligned} a &= 3(r(t) - r(v)) + 1 = 3(\log s(t) - \log s(v)) + 1 \\ &\leq 3(\log 1 - \log \frac{1}{n}) + 1 = -3 \log \frac{1}{n} + 1 \\ &= 3 \log n + 1. \end{aligned}$$

Potentiaali pienenee operaatiojonon suorituksen aikana korkeintaan

$$\Delta\Phi = \sum_{i=1}^{n} (\log s_+ - \log s_-) = \sum_{j=1}^{n} \log n = n \log n,$$

joten operaatiojonon todellinen aika on kokeintaan

$$a + \Delta\Phi = m(3 \log n + 1) + n \log n = O((n + m) \log n + m).$$

Jos yhden operaation pahimman tapauksen aikavaatimus olisi $O(\log n)$, kuten on tilanne esimerkiksi AVL-puiden kohdalla, olisi operaatiojonon kustannus $O(m \log n)$. Jos siis tarkastellaan tarpeeksi pitkiä operaatiojonoja, on levittyvä puu yhtä tehokas kuin AVL-puu. Tällöin nimittäin kertojana olevan termin n vaikutus asymptoottiseen aikavaatimukseen häviää.

Valitsemalla solmujen painot eri tavoin voidaan lauseesta 2.6 johtaa muitakin mielenkiintoisia tuloksia.

Levitysoperaatio on analoginen move-to-front -heuristiikan kanssa. Luonnollinen arvaus on siis, että myös levittyvät puut olisivat kilpailukykyisiä. Tätä ns. dynamic optimality -konjektuuria ei kuitenkaan ole pystytty todistamaan.

2.4.2. Vino kasa

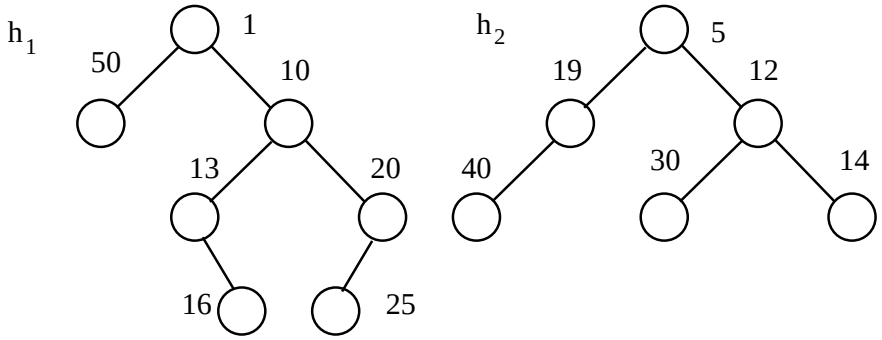
Tässä alakohdassa tarkasteltava tietorakenne, ns. *vino kasa* (skew heap), toteuttaa prioriteettijonon. Tehtävänä on ylläpitää kokoelmaa annetun perusjoukon alkiovieraita osajoukkoja ja suorittaa niille seuraavia operaatioita:

delete_min(h)	palauta osajoukon h pienimmän alkion arvo ja poista pienin alkio joukosta,
insert(x, h)	lisää alkio x osajoukkoon h,
meld(h_1, h_2)	lisää kaikki osajoukon h_2 alkiot osajoukkoon h_1 ja tuhoa h_2 sekä
make_null(h)	muodosta tyhjä osajoukko, jonka nimi on h.

Jokainen joukko esitetään binaaripuuna, jonka jokaisessa solmussa on alkio, joka on arvoltaan pienempi kuin kaikki sen jälkeläiset. Kasan juuressa on aina sen pienin alkio. Binaaripuun *oikeaksi poluksi* kutsutaan juuresta alkavaa oikeita osoittimia seuraavaa polkua; vastaavasti *vasen polku* alkaa juuresta ja seuraa vasempia osoittimia.

Operaatio delete_min voidaan suorittaa niin, että kasasta h poistetaan juuri ja korvataan h operaation meld(h_1, h_2) tuloksella, kun h_1 ja h_2 ovat juuren oikea ja vasen alipuu. Operaatio meld(h_1, h_2) voidaan puolestaan suorittaa etenemällä alipuiden oikeita polkuja juuresta lähtien ja lomittamalla poluilla olevat solmut kasvavaan järjestykseen. Meld-operaatiot hidastuvat oikeiden polkujen pidentyessä. Perinteinen ratkaisu oikeiden polkujen pidentymisen aiheuttamaan ongelmaan on ylläpitää solmuissa tietoa niiden oikeiden ja vasempien lasten etäisyydestä lehtisolmuista. *Vasemmistolaisessa kasassa* (leftist heap) lyhin polku jokaisesta solmusta lehteen kulkee aina oikean lapsen kautta (tässä myös puuttuva alipuu luetaan lehdeksi). Tämän ominaisuuden voimassa pitämiseksi on lomituspoluilla olevien solmujen tilatietoja tietenkin tutkittava meld-operaatioiden jälkeen. Vasemmistolaisen kasan prioriteettioperaatioiden pahimman tapauksen aikavaatimus on $O(\log n)$.

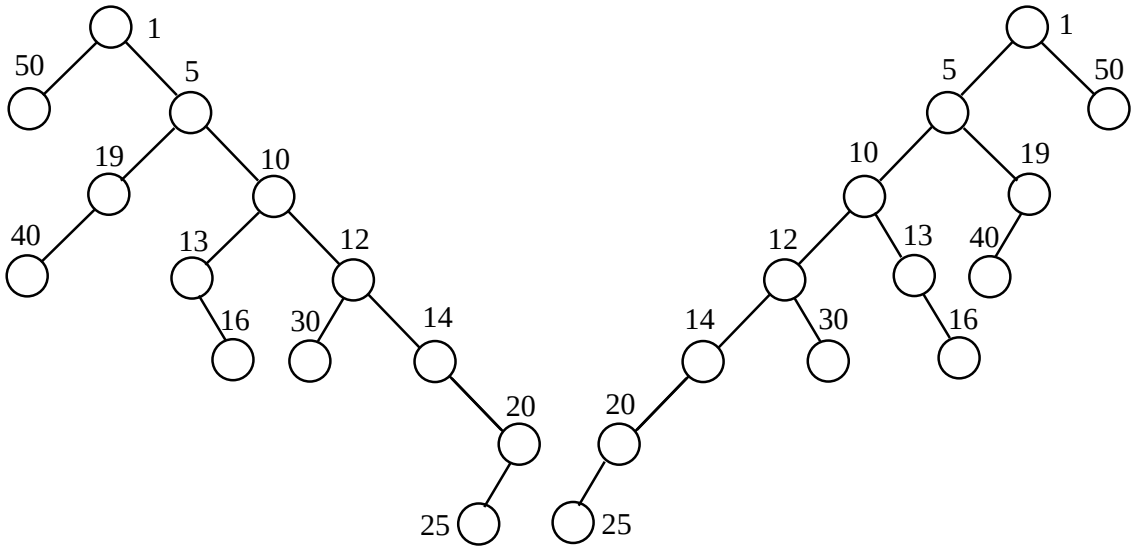
Vastaavassa mukautuvassa tietorakenteessa, vinossa kasassa, samantapainen vaikutus saadaan mukautusoperaatiolla, joka tehdään jokaisen meld-operaation yhteydessä. Mukautusoperaatio vaihtaa lomitetut solmut oikeista lapsista vasemmiksi ja niiden lapset vasemmista oikeiksi, lukuunottamatta viimeisen solmun lasta. Kuten mukautuvien tietorakenteiden periaatteisiin kuuluu, tätä uudelleenjärjestelysääntöä sovelletaan jokaisen meld-operaation yhteydessä riippumatta kasan tilasta. Tarkastellaan esimerkkinä kuvan 2.12 tilannetta. Kuvan vinojen kasojen meld-operaation lopputulos on esitetty kuvassa 2.13 oikealla.



Kuva 2.12: Kaksi vinoa kasaa.

Meld-operaation viimeisen vaiheen tarkoituksena on pitää vinojen kasojen oikeat polut lyhyinä ja nopeuttaa seuraavia meld-operaatioita. Tässä mielessä vino kasa on mukautuva tietorakenne. Meld-operaatio on kasan perusoperaatio, johon perustuen kasan muut operaatiot toteutetaan.

Määritellään vinon kasan solmun x *paino* $wt(x)$ niiden solmujen lukumääränä, jotka ovat siinä alipuussa, jonka juuri x on. Solmua (ei kuitenkaan juurta) sanotaan *ras-*



Kuva 2.13: Kuvan 2.12 vinot kasat lomituksen jälkeen (vasemmalla) ja mukautusoperaation jälkeen (oikealla). Huomaa, että solmu 25 pysyy solmun 20 vasempana lapsena, sillä 20 oli lomituspolun viimeinen solmu.

kaaksi, jos sen painolla $wt(x)$ ja sen vanhemman $p(x)$ painolla $wt(p(x))$ on voimassa $wt(x) > wt(p(x))/2$. Muulloin solmu on *kevyt*. Solmun lapsista vain toinen voi olla raskas. Samoin on helppo todistaa, että millä tahansa n-solmuisen vinon kasan polulla voi olla korkeintaan $\lfloor \log n \rfloor$ kevyttä solmua. Tämän osoittamiseksi merkitään kasan juurta x:llä ja polun viimeistä solmua y:llä. Jos polulla on k kevyttä solmua, niin pätee $wt(y) \leq wt(x)/2^k$. Kertomalla puolittain 2^k:lla, jakamalla puolittain $wt(y)$:lla ja ottamalla molemmista puolista logaritmit saadaan

$$k \leq \log(\frac{wt(x)}{wt(y)}) = \log n,$$

sillä $wt(x) = n$ (x on juuri) ja $wt(y) = 1$ (y on lehti). Millä tahansa n-solmuisen vinon kasan polulla on siis korkeintaan $\lfloor \log n \rfloor$ kevyttä solmua.

Vinon kasan potentiaaliksi määritellään nyt niiden solmujen lukumäärä, jotka ovat raskaita oikeita lapsia. Kuvassa 2.12 kasan h_1 potentiaali on 1, sillä solmu 10 on sen ainoa raskas oikea lapsi. Kasan h_2 potentiaali on 0. Potentiaalifunktion määrittelyn järkevyydestä voidaan nopeasti vakuuttautua seuraavalla päättelyllä. Jos meld-operaation suoritukseen kuluu aikaa enemmän kuin $O(\log n)$, niin oikeissa poluissa on ollut paljon raskaita solmuja, sillä kevyitä solmuja on millä tahansa polulla korkeintaan $\lfloor \log n \rfloor$. Koska oikeilla poluilla olleet raskaat solmut lopputilanteessa ovat vasempia lapsia, vähenee potentiaali juuri niin paljon kuin aika ylitti $O(\log n)$:n.

Oletetaan, että tehdään meld-operaatio vinoille kasoille, joiden solmujen lukumäärät ovat n_1 ja n_2. Oletetaan edelleen, että näiden vinojen kasojen oikeilla poluilla on raskaita solmuja vastaavasti k_1 ja k_2 kappaletta. Raskaiden solmujen lisäksi oikeilla poluilla on yhteensä kaksi juurta ja edellä tehdyn huomion perusteella kevyitä solmuja korkeintaan $\lfloor \log n_1 \rfloor$ ja $\lfloor \log n_2 \rfloor$ kappaletta. Yhteensä kevyitä solmuja on oikeilla poluilla korkeintaan

$2\lfloor \log n \rfloor - 1$, kun $n = n_1 + n_2$. Meld-operaation todellista aikaa voidaan siis arvioida ylöspäin lausekkeella

$$1 + 2\lfloor \log n \rfloor + k_1 + k_2.$$

Tasoitettua aikaa määrättäessä on lisäksi huomioitava potentiaalin muutos. Oikeissa poluissa olleet raskaat solmut muuttuvat vasemmiksi lapsiksi; tämän vuoksi potentiaali pienenee $(k_1 + k_2)$:lla. Toisaalta jotkut aikaisemmin vasempina lapsina olleet raskaat solmut muuttuvat oikeiksi. Jokaista oikeaksi lapseksi muuttunutta raskasta solmua vastaa lopputilanteessa vasemmalla polulla oleva kevyt solmu. Koska kaikilla n-solmuisen vinon kasan poluilla on korkeintaan $\lfloor \log n \rfloor$ kevyttä solmua, ei potentiaali voi kasvaa tätä enempää niiden solmujen vaikutuksesta, jotka muuttuvat meld-operaation yhteydessä vasemmista oikeiksi. Täten meld-operaation tasoitettua aikaa voidaan arvioida ylöspäin lausekkeella

$$1 + 2\lfloor \log n \rfloor + k_1 + k_2 + \lfloor \log n \rfloor - k_1 - k_2 \leq 3\lfloor \log n \rfloor + 1.$$

Meld-operaation tasoitettu aika on siis $O(\log n)$.

Perinteisessä mielessä vinon kasan pahimman tapauksen aikavaatimus on $\Omega(n)$. Tämä nähdään tarkastelemalla esimerkiksi operaatiojonoa

make_null(h),
insert($(n/2) + 1, h$),
insert($(n/2) - 1, h$),
insert($(n/2) + 2, h$),
insert($(n/2) - 2, h$),
$\ldots$,
insert($1, h$),
insert(n, h),
delete_min.

Operaatiojonon päättävässä delete_min operaatiossa lomitetaan polut, joiden pituudet ovat 1 ja $\frac{n}{2}$. Kustannus on siis $\Omega(n)$. Tämä osoittaa, että vino kasa ei ole perinteisessä mielessä tehokas, eli on olemassa operaatiojonoja, joiden yksittäiset operaatiot ovat huomattavan kalliita. Tarkasteltaessa riittävän pitkiä operaatiojonoja, yksittäisten kalliiden operaatioiden kustannus jakautuu jonon muille operaatioille.

2.5. Esimerkki alarajoista

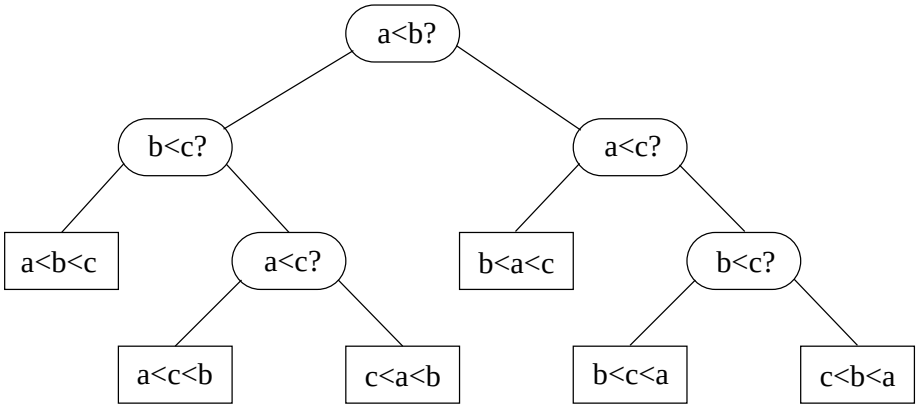
Ongelman vaativuusanalyysissä on tehtävänä selvittää, kuinka paljon aikaa ja tilaa ongelman P ratkaisu vähintään vaatii, käytettiinpä mitä tahansa algoritmia. Yleensä alaraja pyritään osoittamaan jossain "luonnollisessa" algoritmiluokassa C.

Esimerkki 2.12 Lajitteluongelmaa tarkasteltaessa voidaan rajoittua sellaisiin algoritmeihin, jotka perustuvat lajiteltavien alkioiden välisiin, muotoa $a < b$, $a = b$ ja $a > b$ oleviin vertailuihin. Tällöin tarkastelujen ulkopuolelle jäävät esimerkiksi sellaiset lajittelualgoritmit, jotka perustuvat lajiteltavien lukujen binaariesitysten bittikuvioihin. □

Oletetaan, että kaikilla luokkaan C kuuluvilla ongelman P ratkaisevilla algoritmeilla A pätee seuraava: jos $T_{\max}(n) = T(n)$ on A:n aikavaatimus, niin $T(n) = \Omega(f(n))$. Silloin sanotaan, että $T(n)$ on P:n aikavaatimuksen alaraja algoritmiluokassa C pahimmassa tapauksessa. Olkoon $f(n)$ ongelman P jonkin resurssivaatimuksen alaraja luokassa C. Ongelman P ratkaisualgoritmi A on optimaalinen ko. resurssivaatimuksen suhteen luokassa C, jos A:n resurssivaatimus on $O(f(n))$.

Monissa alarajatodistuksissa käytetään algoritmiluokan rajaamiseen päätöspuita. Päätöspuu on rakenne, joka esittää algoritmin sisältämien mahdollisten suoritusten kokoelmaa. Tarkastellaan lähemmin lajitteluongelmaan liittyviä päätöspuita. Tehtävänä on lajitella n alkiota muotoa $a < b$, $a = b$ ja $a > b$ olevilla vertailuilla. Saatavassa päätöspuussa T_n on ainakin yksi lehti kutakin lajiteltavien alkioiden permutaatiota kohti. Lehtien lukumäärä on siis ainakin $n!$.

Esimerkki 2.13 Olkoon $n = 3$ ja olkoon alkutilanne on $A[1] = a$, $A[2] = b$, $A[3] = c$. Tulokseksi saadaan kuvan 2.14 mukainen päätöspuu. □



Kuva 2.14: Esimerkkiin liittyvä päätöspuu.

Lause 2.7 *Jokainen vertailuihin perustuva lajittelualgoritmi tekee pahimmassa tapauksessa ainakin* $\lceil \log n! \rceil = \Omega(n \log n)$ *vertailua.*

Todistus. Jos binaaripuussa on k lehteä, niin puun korkeus on vähintään $\lceil \log k \rceil$. Lajitteluun liittyvässä päätöspuussa T_n lehtien lukumäärä on vähintään $n!$, joten puun korkeus on vähintään $\lceil \log n! \rceil$. Koska puun korkeus on sama kuin vertailujen lukumäärä pahimmassa tapauksessa, niin todistus on valmis, kun vielä näytetään, että $\lceil \log n! \rceil$ on $\Omega(n \log n)$. Voidaan kirjoittaa

$$n! = n(n-1)\ldots\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)\ldots 1 \geq \left(\frac{n}{2}\right)^{n/2} 1^{n/2},$$

joten

$$\log n! \geq \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}\log n - \frac{n}{2}.$$

□

Aikaisemmin on näytetty, että lomituslajittelun aikavaatimus on $O(n \log n)$. Lauseesta 2.7 seuraa nyt, että lomituslajittelu on pahimmassa tapauksessa optimaalinen.

On huomattava, että Lauseen 2.7 rajaa ei välttämättä voida saavuttaa. Esimerkiksi 12 alkion lajittelemiseen tarvitaan aina vähintään 30 vertailua, mutta $\lceil \log 12! \rceil = 29$. Vaikka lauseen raja saavutettaisiinkin, sitä ei yleensä saavuteta tavallisilla lajittelualgoritmeilla.

Lause 2.8 *Jos kaikki syötepermutaatiot ovat yhtä todennäköisiä, niin jokainen vertailuihin perustuva lajittelumenetelmä tekee keskimäärin ainakin* $\Omega(n \log n)$ *vertailua.*

Todistus. Voidaan osoittaa (vaikka tässä todistus sivuutetaan), että k-lehtisessä binaaripuussa lehtien keskimääräinen syvyys on vähintään $\log k$. Väite saadaan, kun valitaan $k = n!$. □

Lauseesta 2.8 seuraa, että pikalajittelu on keskimääräisessä tapauksessa optimaalinen.

Luku 3

Algoritmien suunnittelutekniikoita

Tässä luvussa esitellään erilaisia menetelmiä algoritmien laatimiseksi. Yleisesti ei ole olemassa menetelmää, jolla voitaisiin valita kuhunkin ongelmaan sopiva "oikea" algoritmin suunnittelumenetelmä, ja monet ongelmat voidaan ratkaista tehokkaasti useammalla kuin yhdellä tavalla. Mitä useampia algoritmien suunnittelumenetelmiä ongelman ratkaisija hallitsee, sitä varmemmin hän löytää tehokkaan ratkaisun annetulle ongelmalle.

3.1. Hajota ja hallitse

Hajota ja hallitse -menetelmässä ongelma jaetaan osaongelmiksi, ratkaistaan ne ja kootaan tuloksista alkuperäisen ongelman ratkaisu. Koska osaongelmat usein ovat alkuperäisen ongelman kaltaisia mutta pienemmässä mittakaavassa, voidaan käyttää rekursiota. Hajota ja hallitse -periaate voidaan esittää algoritmina kuvan 3.1 tavalla.

```
    Function HH(s: ongelman tapaus): ongelman ratkaisu
(1) if pienitapaus(s) then
(2)         return ratkaisu(s)
(3) else
(4)         (s_1, s_2, ..., s_p) := jaa_ongelma(s)      % jako samankokoisiin osiin
(5)         return kokoa_ratkaisu(HH(s_1), HH(s_2), ..., HH(s_p))
(6) end
```

Kuva 3.1: Hajota ja hallitse.

Aika-analyysiä varten merkitään tapauksen kokoa n:llä. Kun jako tehdään p:hen yhtä suureen osaan, joiden koko on n/m (osien ei tarvitse olla pistevieraita), niin aikavaatimus voidaan esittää differenssiyhtälönä

$$T(n) = \begin{cases} g(n), & \text{kun } n \text{ on pieni,} \\ pT(n/m) + f(n), & \text{muulloin.} \end{cases}$$

Jos tapaus on pieni, niin suoritetaan algoritmin rivit (1) ja (2). Tähän oletetaan kuluvan aikaa $g(n)$. Rivien (1) ja (4) sekä kokoa_ratkaisu-kutsun oletetaan vievän aikaa

$f(n)$.

Esimerkki 3.1 Otetaan tehtäväksi etsiä n-alkioisen lukujoukon maksimi ja minimi. Käymällä lukujoukko läpi kaksi kertaa voidaan tehtävä ratkaista tekemällä $2n-3$ vertailua. Tehtävän ratkaiseva hajota ja hallitse -algoritmi esitetään kuvassa 3.2.

```
    Function MAXMIN(S: lista): lukupari
(1) if |S| ≤ 2 then      % olkoon S = {a, b}
(2)     return (max(a, b), min(a, b))
(3) else
(4)     Jaa S yhtä suuriin osiin S1 ja S2
(5)     (max1,min1):= MAXMIN(S1)
(6)     (max2,min2):= MAXMIN(S2)
(7)     return (max(max1,max2), min(min1,min2))
(8) end
```

Kuva 3.2: Lukujoukon maksimin ja minimin etsintä hajoita ja hallitse -menetelmällä.

MAXMIN-algoritmin vertailujen määrälle pätee

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

Ratkaisuksi saadaan $T(n) = 3n/2 - 2$ purkamismenettelyllä seuraavasti:

$$\begin{aligned} T(n) &= 2^{k-1}T(\frac{n}{2^{k-1}}) + \sum_{i=1}^{k-1} 2^i = 2^{k-1}T(2) + \sum_{i=0}^{k-1} 2^i - 1 \\ &= 2^{k-1} + 2^k - 1 - 1 = \frac{2^k}{2} + 2^k - 2 = \frac{2^{\log n}}{2} + 2^{\log n} - 2 \\ &= \frac{n}{2} + n - 2 = \frac{3n}{2} - 2. \end{aligned}$$

Hajota ja hallitse -algoritmi voidaan todistaa vertailujen lukumäärän suhteen optimaaliseksi. Käytännössä alkeellinen ratkaisu on kuitenkin tehokkaampi. □

Esimerkki 3.2 (Karatsuban ja Ofmanin kertolasku). Tarkastellaan pitkien lukujen kertolaskua. Jos luvut kerrotaan allekkain peruskoulussa opittuun tapaan, niin numeroiden kerto- ja yhteenlaskujen lukumäärä on $O(n^2)$, kun n on kerrottavien lukujen yhteinen pituus. Lisäämällä kerrottavien eteen tarvittaessa nollia, voidaan olettaa, että n on kakkosen potenssi. Nyt voidaan kerrottavat luvut toistuvasti katkaista keskeltä ja laskea tulo hajotelmia $x = sk^{n/2} + t$ ja $y = uk^{n/2} + v$ käyttäen:

$$\begin{aligned} xy &= (sk^{n/2} + t)(uk^{n/2} + v) = suk^n + (sv + tu)k^{n/2} + tv \\ &= suk^n + \big((s + t)(u + v) - su - tv\big)k^{n/2} + tv. \end{aligned}$$

Viimeksi kirjoitetun lausekkeen mukaan tulo voidaan siis laskea kolmesta $n/2$-numeroisten lukujen tulosta yhteen- ja vähennyslaskuilla. Aikavaatimukselle on voimassa differenssiyhtälö

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ 3T(n/2) + cn, & \text{kun } n > 1. \end{cases}$$

Ratkaisuksi saadaan $T(n) = O(n^{\log 3}) \approx O(n^{1.59})$. Huomiotta on jätetty se mahdollisuus, että summissa $(s + t)$ ja $(u + v)$ voi olla $\frac{n}{2} + 1$ numeroa. Tämä ei vaikuta edellä löydetyn ratkaisun suuruusluokkaan. □

Ongelman hajoittaminen osaongelmiin kannattaa yleensä tehdä niin, että osat ovat mahdollisimman samankokoisia. Esimerkiksi lisäyslajittelu voidaan ymmärtää hajota ja hallitse -tyyppisenä algoritmina, jossa n:n kokoinen ongelma jaetaan osiin, joiden koot ovat 1 ja $(n - 1)$. Algoritmin aikavaatimus on $O(n^2)$. Suorittamalla jako tasapainoisesti, kuten lomituslajittelussa, saadaan aikavaatimukseksi $O(n \log n)$.

3.2. Dynaaminen ohjelmointi

Hajota ja hallitse -menetelmää sovellettaessa saattaa syntyä tilanne, jossa sama osaongelma ratkaistaan monta kertaa. Tämä on tietenkin tehotonta. Esimerkiksi Fibonaccin lukujen laskeminen kuvan 3.3 funktiolla on varsin tehotonta (tarvitaan eksponentiaalinen aika luvun n suhteen), sillä sama Fibonaccin luku joudutaan laskemaan moneen kertaan. Hajota ja hallitse -menetelmä ratkaisee tämän ongelman tehottomasti *osittavalla* (top-down) lähestymistavalla.

```
    Function fib(n): integer
(1) if n < 1 then
(2)         return 1
(3) else
(4)         return fib(n − 1) + fib(n − 2)
(5) end
```

Kuva 3.3: Fibonacin luvut.

Etenemällä "alhaalta ylöspäin"eli soveltaen ongelmaan *kokoavaa* (bottom-up) ratkaisutapaa ja tallentamalla saavutetut välitulokset voidaan n. Fibonaccin luku laskea ajassa $O(n)$. Dynaamiselle ohjelmoinnille on oleellista juuri välitulosten taulukointi. Aloittamalla pienistä osaratkaisuista ja taulukoimalla saadut tulokset voidaan edetä kohti koko ongelman ratkaisua ratkaisematta samaa osaongelmaa yhtä kertaa useammin.

Dynaamista ohjelmointia sovellettaessa noudatetaan yleensä seuraavia vaiheita:

1. Etsitään rekursiivinen ominaisuus, jonka avulla saadaan ongelman ratkaisu.

2. Ratkaistaan ongelma kokoavalla tavalla pienet tapaukset ensin.

Esimerkki 3.3 Tehtävänä on laskea matriisien tulo $M = M_1 M_2 \cdots M_n$. Matriisien kertolasku on assosiatiivinen; laskut voidaan laskea missä järjestyksessä tahansa. Tarvittavien laskuoperaatioiden lukumäärä voi kuitenkin vaihdella huomattavasti käytetystä laskujärjestyksestä riippuen. Esimerkiksi jos tulossa $M_1 M_2 M_3 M_4$ matriisien koot ovat vastaavasti 10×20, 20×50, 50×1 ja 1×100, niin järjestys $M_1(M_2(M_3 M_4))$ vaatii 125000 operaatiota, kun järjestyksessä $(M_1(M_2 M_3))M_4$ riittää 2200 operaatiota.

Tässä voisi käyttää ratkaisua, joka luettelee kaikki laskujärjestykset tulolle $M_1 M_2 \cdots M_n$ ja laskee kuhunkin laskujärjestykseen liittyvät kertolaskujen määrät. Esimerkiksi yllä käsitellyn tulon $M_1 M_2 M_3 M_4$ muut laskujärjestykset ovat

$$((M_1 M_2)M_3)M_4, \;\; M_1((M_2 M_3)M_4) \text{ ja } (M_1 M_2)(M_3 M_4).$$

Eri laskujärjestysten määrä kasvaa eksponentiaalisesti kerrottavien matriisien lukumäärän suhteen, joten kaikkien vaihtoehtojen tarkistaminen yleisessä tapauksessa ei ole käytännöllistä.

Lähdetään johtamaan dynaamisen ohjelmoinnin periaatteilla kertomisjärjestykselle ratkaisua. Olkoon matriisin M_i koko $r_{i-1} \times r_i$, kun $i = 1, \ldots, n$. Oletetaan, että optimaaliseen kertomisjärjestykseen liittyy osatulojen $M_1 \cdots M_i$ ja $M_{i+1} \cdots M_n$ laskeminen. Osatuloista tiedetään ainakin sen verran, että jos ne saadaan laskettua optimaalisesti, on saatu ratkaistua koko ongelma optimaalisesti. Tämän osoittamiseksi voidaan tarkastella tilannetta, jossa tuloa $M_1 \cdots M_i$ ei kerrotakaan optimaalisessa järjestyksessä. Nyt koko ongelma $M_1 \cdots M_n$ ei myöskään ratkea optimaalisesti, sillä osa tuloista voidaan laskea paremmassa järjestyksessä. Sama huomio pätee loppuosaan $M_{i+1} \cdots M_n$.

Muodostetaan taulukko, johon lasketaan osaongelmien ratkaisuja, pienistä tapauksista isoihin. Ideana on dynaamisen ohjelmoinnin periaatteiden mukaan laskea isommat tapaukset pienien tapauksien avulla. Merkitään taulukon alkioita m_{ij}, kun $1 \leq i \leq j \leq n$, ja tulkitaan kukin alkio m_{ij} osatulon $M_i \cdots M_j$ optimaaliseksi vastaukseksi. Koko ongelman vastauksen antaa siis alkio m_{1n}.

Jos $i = j$, on tulossa vain yksi matriisi, joten niissä tilanteissa ei tarvita yhtään kertolaskua. Muiden arvojen m_{ij} $(i < j)$ laskemisessa käytetään apuna edellä esitettyä ongelman rekursiivista rakennetta. Oletetaan, että optimaalisessa järjestyksessä osatulo $M_i \cdots M_j$ jakaantuu matriisien M_p ja M_{p+1} välistä, kun $i \leq p < j$. Aiemman huomion perusteella tässä luvun m_{ij} arvoksi tulee osatulojen $M_i \cdots M_p$ ja $M_{p+1} \cdots M_j$ kertolaskujen lukumäärä sekä näiden kahden osatulon alkioiden kertolaskujen lukumäärä. Kyseinen osatulon kertolaskujen lukumäärä on $r_{i-1} r_p r_j$. Toisin sanoen

$$m_{ij} = m_{ip} + m_{p+1,j} + r_{i-1} r_p r_j.$$

Koska ei tiedetä, millä luvulla p löytyy paras vastaus, joudutaan tutkimaan kaikki $j - i$ vaihtoehtoa. Siis

$$m_{ij} = \begin{cases} 0, & \text{jos } i = j, \\ \min\limits_{i \leq p < j} \left(m_{ip} + m_{p+1,j} + r_{i-1} r_p r_j \right), & \text{jos } i < j. \end{cases}$$

Yllä oleva voidaan laskea rekursiivisesti, mikä on tehotonta, sillä silloin ratkaistaan usea aliongelma moneen kertaan (suoraviivainen rekursio johtaa eksponentiaaliseen algoritmiin). Tässä kohtaa käytetään luonnollisesti dynaamisen ohjelmoinnin kokoavaa tapaa. Kukin aliongelma tarkoittaa lukujen i ja j valintaa ehdolla $1 \leq i \leq j \leq n$ eli käsiteltävänä on yhteensä $O(n^2)$ eri aliongelmaa. Mitään aliongelmaa ei ratkaista kahta kertaa.

```
(1) for i := 1 to n do m_ii := 0 od
(2) for t := 1 to n − 1 do
(3)       for i := 1 to n − t do
(4)             j := i + t
(5)             m_ij := min_{i≤p<j}(m_ip + m_{p+1,j} + r_{i−1} r_p r_j)
(6)  od od
```

Kuva 3.4: Lukujen m_{ij} laskeminen.

Lasketaan "diagonaaleittain" luvut m_{ij} $(1 \leq i < j \leq n)$ kuvan 3.4 algoritmilla: ensin yhden matriisin tulon kertolaskujen määrä (rivi (1)), sitten järjestyksessä (rivit (2)–(6)) kahden matriisin tulojen optimaaliset kertolaskujen määrät, kolmen matriisin tulojen optimaaliset kertolaskujen määrät jne. kunnes vihdoin "yhden mittaiselle diagonaalille" lasketaan kaikkien matriisien tulon kertolaskujen optimaalinen määrä.

Lasketaan yllä olevaan esimerkkiin liittyvät luvut m_{ij}. Esimerkissä oli $r_0 = 10$, $r_1 = 20$, $r_2 = 50$, $r_3 = 1$ ja $r_4 = 100$. Saadaan

$$
\begin{array}{llll}
m_{11} = 0 & & & \\
m_{12} = 10000 & m_{22} = 0 & & \\
m_{13} = 1200 & m_{23} = 1000 & m_{33} = 0 & \\
m_{14} = 2200 & m_{24} = 3000 & m_{34} = 5000 & m_{44} = 0.
\end{array}
$$

Esimerkiksi luku m_{14} saadaan miniminä seuraavista kolmesta luvusta:

$$
\begin{aligned}
m_{11} + m_{24} + r_0 \times r_1 \times r_4 &= 0 + 3000 + 10 \times 20 \times 100 = 23000 \\
m_{12} + m_{34} + r_0 \times r_2 \times r_4 &= 10000 + 5000 + 10 \times 50 \times 100 = 65000 \\
m_{13} + m_{44} + r_0 \times r_3 \times r_4 &= 1200 + 0 + 10 \times 1 \times 100 = 2200.
\end{aligned}
$$

Luvut liittyvät vastaavasti laskujärjestyksiin $M_1((M_2M_3)M_4)$, $((M_1M_2)(M_3M_4))$ ja $(M_1(M_2M_3))M_4$. Viimeksi mainittu on siis paras laskujärjestys.

Algoritmia pitää vielä täydentää niin, että se tulostaa myös laskujärjestyksen eikä pelkästään operaatioiden lukumäärää. Tämä tulee hoidetuksi, kun jokaiseen lukuun m_{ij} liitetään se p:n arvo, jolla kyseinen minimi on saavutettu. Optimaalisen laskujärjestyksen etsivässä algoritmissa on kaksi silmukkaa ja minimin laskeminen sisäkkäin, joten se toimii ajassa $O(n^3)$. □

Esimerkki 3.4 Olkoon annettuna graafi, jonka jokaiseen särmään on liitetty paino (= etäisyys). Tehtävänä on etsiä jokaiselle solmuparille niitä yhdistävän lyhimmän polun pituus. Jos oletetaan, että tarkasteltava graafi on yhtenäinen, niin tällainen lyhin polku on olemassa kaikille solmupareille.

Olkoot graafin solmut $v_1, v_2, \ldots, v_n$ ja d_{ij} solmuja v_i ja v_j yhdistävän särmän paino. Ongelma ratkeaa kuvan 3.5 algoritmilla. Muuttujaan m_{ij} tallennetaan jo löydetyn lyhimmän polun pituus solmusta v_i solmuun v_j. Muuttuja k ilmoittaa suurimman solmujen indeksin, joka polulla sallitaan. Aluksi rajoitutaan polkuihin, joissa voi välisolmuna olla vain v_1, sitten polkuihin, joissa voi välisolmuna olla myös v_2, jne. Samoin kuin matriisien tulon laskevassa algoritmissa, niin nytkin voidaan osatuloksiin liittää parhaan tuloksen antava järjestys. Algoritmissa on kolme sisäkkäistä silmukkaa, joiden sisällä oleva lause voidaan suorittaa vakioajassa. Aikavaatimus on siis $O(n^3)$.

```
     Procedure paths
(1)  for i := 1 to n do
(2)       for j := 1 to n do m_ij := d_ij od od
(3)  for k := 1 to n do
(4)       for i := 1 to n do
(5)            for j := 1 to n do
(6)                 m_ij := min(m_ij, m_ik + m_kj)
(7)  od od od
```

Kuva 3.5: Lyhyimmät polut (Floydin-Warshallin algoritmi).

Tarkastellaan vielä graafia, jonka etäisyysmatriisi on annettu taulukossa 3.1 vasemmalla.

d	1	2	3	4	5
1	0	4	2	6	∞
2	2	0	∞	∞	5
3	∞	∞	0	1	1
4	∞	2	∞	0	∞
5	∞	∞	4	2	0

d	1	2	3	4	5
1	0	4	2	3	3
2	2	0	4	5	5
3	5	3	0	1	1
4	4	2	6	0	7
5	6	4	4	2	0

Taulukko 3.1: Graafin etäisyysmatriisi ja siihen liittyvät lyhimmät etäisyydet.

Kun $k = 1$, sallitaan yhteydet solmun 1 kautta. Tällöin saadaan yhteydet solmusta 2 solmuihin 3 ja 4. Näiden pituudet ovat vastaavasti 4 ja 8. Kun sallitaan yhteydet solmun 2 kautta, saadaan yhteydet solmusta 4 solmuihin 1, 3 ja 5. Vastaavat etäisyydet ovat 4, 6 ja 7. Lisäksi saadaan yhteys solmusta 1 solmuun 5. Kun $k = 3$, lyhenevät yhteydet solmusta 1 solmuihin 4 ja 5 (uudet etäisyydet ovat molemmat 3) ja solmusta 2 solmuun 4. Solmun 4 kautta saadaan yhteydet solmuista 3 ja 5 solmuihin 1 ja 2. Ratkaisuksi saadut etäisyydet ovat taulukossa 3.1 oikealla.

□

Tässä kohdassa annetut yleiset ohjeet dynaamisen ohjelmoinnin soveltamiseksi ja edellinen esimerkki nojautuvat *optimaalisuuden periaatteeseen*, jonka mukaan *optimaalisessa sarjassa päätöksiä tai valintoja on kukin osasarja myös optimaalinen*. Kaikkiin ongelmiin ei optimaalisuuden periaatetta voi soveltaa.

Esimerkiksi jos lyhyin reitti Helsingin ja Tampereen välillä kulkee Hämeenlinnan kautta, niin silloin Helsingin ja Hämeenlinnan välillä tulee käyttää lyhyintä reittiä, aivan kuten Hämeenlinnan ja Tampereen välillä. Tässä siis optimaalisuuden periaate toimii hyvin.

Sen sijaan, jos nopein reitti kulkee Hämeenlinnan kautta, niin se ei tarkoita, että Helsingistä tulisi ajaa nopeinta reittiä Hämeenlinnaan. Sillä jos Helsingin ja Hämeenlinnan välillä käytetään liikaa polttoainetta ja joudutaan pysähtymään Hämeenlinnan ja Tampereen välillä tankkaamaan, saattaa tuloksena oleva reitti olla hitaampi kuin jokin toinen Hämeenlinnaa välietappina käyttävä reitti. Tämä johtuu siitä, että osareitit eivät ole toisistaan riippumattomia, sillä ne käyttävät yhteistä resurssia, polttoainetta. Siis yhden osamatkan optimaalinen vastaus voi estää löytämästä jotain toista optimaalista osamatkaa.

3.3. Ahneet algoritmit

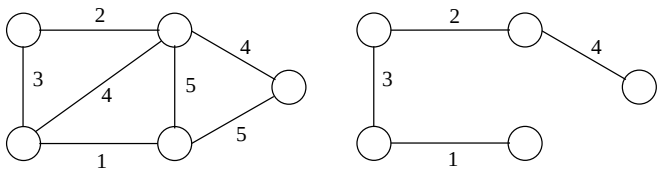
Ahneissa algoritmeissa edetään tekemällä "lokaalisti optimaalisia" valintoja. Eri toimintavaihtoehdot järjestetään paremmuusjärjestykseen kullakin hetkellä käytössä olevien tietojen perusteella ja valitaan (näennäisesti) paras vaihtoehto. Näin toimimalla ei välttämättä päädytä koko ongelman optimaaliseen ratkaisuun.

Esimerkki 3.5 Olkoon $G = (V, E)$ suuntaamaton graafi. Graafin G aligraafia $H = (V, F)$ sanotaan G:n *virittäväksi puuksi*, jos se on puu. Jos graafin G särmiin on liitetty painot, niin pienin virittävä puu on se virittävä puu, jonka särmien yhteenlaskettu paino on pienin. Kruskalin algoritmi muodostaa oikein graafin pienimmän virittävän puun käyttäen kuvan 3.6 ahnetta algoritmia.

```
(1)  Järjestä särmät painon mukaan nousevaan järjestykseen
(2)  Alkutilanteessa jokainen solmu v muodostaa oman aligraafinsa {v}
(3)  while aligraafeja on enemmän kuin yksi do
(4)      Olkoon e painoltaan pienin vielä käsittelemätön särmä
(5)      if e yhdistää kaksi aligraafia W1 ja W2 then
(6)          Muodosta e:n yhdistämä uusi aligraafi W1 ∪ W2
(7)          Poista W1 ja W2 aligraafien kokoelmasta
(8)          Lisää e pienimpään virittävään puuhun
(9)      end
(10) od
```

Kuva 3.6: Kruskalin algoritmi.

Kuvassa 3.7 vasemmalla on painotettu graafi ja oikealla siitä Kruskalin algoritmilla muodostettu pienin virittävä puu. □



Kuva 3.7: Esimerkkigraafi ja siitä Kruskalin algoritmilla muodostettu pienin virittävä puu.

Jos ongelman täsmälliseen ratkaisemiseen kuluu liiaksi aikaa, voidaan etsiä likimääräinen ratkaisu nopealla algoritmilla. Likimääräinen algoritmi voi perustua erilaisiin heuristiikkoihin. Ahne algoritmi sopii joskus tällaiseksi heuristiikaksi.

Esimerkki 3.6 (Yleistetty reppuongelma). Reppuun, jonka tilavuus on T, on pakattava mahdollisimman arvokas lasti tavaroita. Tarjolla olevista tavaroista tunnetaan niiden tilavuudet (t_i) ja arvot (a_i). On siis löydettävä sellainen tarjolla olevien tavaroiden osajoukko I, että $\sum_{i \in I} a_i$ on suurin mahdollinen ehdolla $\sum_{i \in I} t_i \leq T$.

Ahne ratkaisu voidaan nyt toteuttaa kuvan 3.8 tavalla. Näin saatava ratkaisu ei välttämättä ole optimaalinen. Esimerkiksi jos repun koko on 10 ja tavaroiden arvot ovat 9, 5 ja 5 ja koot vastaavasti 7, 5 ja 5, niin ahne algoritmi ei anna optimaalista ratkaisua. □

(1) Lajittele parit (t_i, a_i) yksikköhinnan mukaiseen laskevaan järjestykseen listaan L.
(2) Käy lista L läpi ja ota reppuun kaikki sinne vielä mahtuvat tavarat.

Kuva 3.8: Yleistetyn reppuongelman ahne ratkaisu.

Esimerkki 3.7 (Kauppamatkustajan ongelma). Tarkastellaan seuraavaksi ns. kauppamatkustajan ongelmaa. Sen tapaus muodostuu suuntaamattomasta, painotetusta, täydellisestä graafista $G = (V, E, d)$, $E = V \times V$. Funktio $d : E \to \mathbb{R}$ määrää solmujen väliset etäisyydet. Tehtävänä on löytää graafista mahdollisimman lyhyt (annettujen painojen suhteen) Hamiltonin silmukka eli sellainen polku, joka lähtee jostain solmusta, käy täsmälleen kerran kaikissa muissa solmuissa ja palaa lähtösolmuun.

Ahne algoritmi kauppamatkustajan ongelman ratkaisemiseksi on esimerkiksi kuvassa 3.9 esitetty Kruskalin algoritmin muunnos.

(1) Käsittele särmät kasvavassa pituusjärjestyksessä.
(2) Hyväksy särmä kauppamatkustajan reittiin, jos särmän molemmista päistä alkaa korkeintaan yksi reittiin aikaisemmin valittu särmä ja särmä ei muodosta aikaisemmin valittujen särmien kanssa silmukkaa, ellei samalla saada kaikki solmut sisältävä silmukka.

Kuva 3.9: Kauppamatkustajan ongelman ahne ratkaisu.

	a	b	c	d	e	f
a	0	5.0	7.1	16.6	15.5	18.0
b		0	5.0	11.7	11.0	14.3
c			0	14.0	14.3	18.4
d				0	3.0	7.6
e					0	5.0
f						0

Taulukko 3.2: Kauppamatkustajan ongelman tapaus.

Taulukossa 3.2 on erääseen kauppamatkustajan ongelman tapaukseen liittyvät solmujen etäisyydet. Ahneen algoritmin mukaisesti kauppamatkustajan reittiin tulevat mukaan neljä lyhintä yhteyttä ((d,e), (a,b), (b,c) ja (e,f)). Tämän jälkeen reitistä on valmiina kaksi erillistä osaa, joista toisessa on solmut a, b ja c ja toisessa solmut d, e ja f. Koska solmujen b ja e asteluku on jo kaksi, reitti on mahdollista sulkea särmillä (c,d) ja (a,f) tai särmillä (c,f) ja (a,d). Koska näistä neljästä särmästä lyhin on (c,d), määrää se viimeiseksi mukaan tulevaksi särmäksi särmän (a,f). Saatavan kauppamatkustajan reitin pituus on 50, kun optimireitin pituus on 47.3. □

Joskus voidaan osoittaa, että ahne algoritmi antaa optimaalisen vastauksen. Todistus (ja erityisesti todistuksen tekniikka) voi liittyä vain ko. algoritmiin, mutta usein myös itse ongelmalla on sellaisia kombinatorisia ominaisuuksia, "matroideja", joista tiedetään, että ahne algoritmi antaa optimaalisen vastauksen. Matroideihin ja niiden suhteeseen ahneisiin algoritmeihin voi perehtyä esimerkiksi Cormenin ja muiden oppikirjan [1] avulla.

3.4. Peruuttavat algoritmit

Monissa ongelmissa on ratkaisun löytämiseksi käytävä läpi kaikki ratkaisuvaihtoehdot. Eräs systemaattinen etsintämenetelmä on *peruutus* (backtracking). Ongelmasta muodostetaan puu, jota käydään läpi yleensä syvyyssuuntaisella etsinnällä. Puun lehtisolmut ovat lopullisia ratkaisuja ja polku juuresta lehteen kuvaa tapaa, jolla lehtisolmuun päästään. Peruuttavien algoritmien yhteydessä käytetään usein menetelmiä, joilla etsintäaluetta pyritään rajoittamaan.

3.4.1. Pelipuut ja $\alpha - \beta$ -karsinta

Tarkastellaan esimerkkinä 3×3 ruudukolla pelattavaan jätkänshakkiin liittyvää pelipuuta, jossa puun solmut ovat pelitilanteita. Solmun arvo on

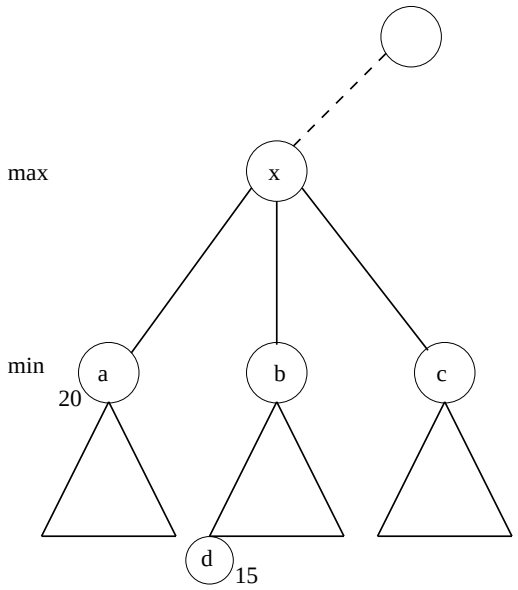
- 1, jos X:llä on solmusta alkava voittostrategia,
- -1, jos 0:llä on solmusta alkava voittostrategia ja
- 0, jos kummallakaan pelaajista ei ole solmusta alkavaa voittostrategiaa.

Lehtisolmut vastaavat pelin lopputilanteita, joten niiden arvot voidaan määrätä suoraan. Sisäsolmujen kohdalla toimitaan seuraavasti: Jos X on siirtovuorossa, niin solmun arvo on sen lasten arvojen maksimi; pelaajan 0 ollessa siirtovuorossa solmun arvo on lasten arvojen minimi. Solmuja kutsutaan vastaavasti *maksimi- ja minimisolmuiksi*.

Etenemällä lehtisolmusta juurta kohti saadaan lopuksi juuren arvoksi 0. Tämä tarkoittaa sitä, että 3×3-ruudukolla pelattavassa jätkänshakissa ei kummallakaan pelaajalla ole voittostrategiaa.

Jätkänshakin pelipuussa solmun arvo voi olla $1, 0$ tai -1. Monimutkaisemmissa tilanteissa on solmujen arvot voitava valita yleisemmin. Näitä arvoja kutsutaan *payoff*-arvoiksi.

Peruuttavassa etsinnässä puu käydään läpi jälkijärjestyksessä. Läpikäyntiä voidaan tehostaa ns. $\alpha - \beta$ *-karsinnalla* (alpha-beta pruning). Yleisesti $\alpha - \beta$ -karsinta soveltuu sellaisten pelipuiden yhteyteen, jossa kaksi pelaajaa tekevät valintoja vuorotellen.



Kuva 3.10: $\alpha - \beta$ -karsinta.

Kuvan 3.10 puussa x on max-solmu ja sen lapset a, b ja c min-solmuja. Oletetaan, että lasten läpikäynti on aloitettu a:sta ja että sen payoff-arvoksi on saatu 20. Koska x on max-solmu, niin lasten b ja c payoff-arvot voivat vaikuttaa x:ään vain jos ne ovat suurempia kuin 20. Oletaan vielä, että keskimmäistä alipuuta läpikäytäessä saadaan solmun b jonkun lapsen payoff-arvoksi 15. Koska b on min-solmu, niin voidaan päätellä, että b:n payoff-arvoksi tulee korkeintaan 15. Keskimmäisen alipuun läpikäynti voidaan siis lopettaa.

Tässä esimerkissä on käytetty solmun a payoff-arvoa 20 solmun x väliaikaisena arvona. Solmulle a arvo 20 oli lopullinen, sillä oletettiin, että koko se alipuu, jonka juurena a on, oli jo läpikäyty. Lopullisten ja väliaikaisten arvojen laskemisessa käytetään seuraavia sääntöjä:

1. Jos solmun x kaikki lapset on jo käsitelty tai karsittu, niin x:n väliaikainen arvo

	A	B	C
1	3	4	5
2	7	2	1
3	6	5	3
4	2	5	7

	A	B	C
1	3	7	12
2	10	12	13
3	16	21	24
4	18	26	33

Taulukko 3.3: Työtehtävien suoritusajat ja vapautumishetket työjärjestyksellä $1-2-3-4$.

muuttuu lopulliseksi.

2. Jos solmulla x on väliaikainen arvo v_1 ja lapsi, jonka lopullinen arvo on v_2, niin uusi väliaikainen arvo on $\max(v_1, v_2)$, jos x on max-solmu ja $\min(v_1, v_2)$, jos x on min-solmu.

3. Olkoot solmun p ja sen vanhemman q väliaikaiset arvot v_1 ja v_2. Jos $v_1 \leq v_2$ ja p on min-solmu (q on siis max-solmu), niin kaikki p:n käsittelemättömät lapset voidaan karsia. Samoin voidaan toimia, jos $v_1 \geq v_2$ ja p on max-solmu.

3.4.2. Branch-and-bound

Branch-and-bound-menetelmässä hakupuusta karsitaan sellaisia haaroja, jotka eivät voi sisältää optimaalista ratkaisua. Optimointitehtävässä etsitään sellaista ratkaisupolkua $x_1, \ldots, x_n$, jonka kustannus $c(x_1, \ldots, x_n)$ on pienin. Oletetaan, että käytössä on rajafunktio b, joka liittää jokaiseen polkuun $x_1, \ldots, x_r$ sellaisen arvon $b(x_1, \ldots, x_r)$, että jos $x_1, \ldots, x_r$ voidaan täydentää koko ongelman ratkaisuksi $x_1, \ldots, x_r, x_{r+1}, \ldots, x_n$, niin $c(x_1, \ldots, x_r, x_{r+1}, \ldots, x_n) > b(x_1, \ldots, x_r)$. Tällaista rajafunktiota voidaan käyttää hakupuiden sellaisten haarojen karsintaan, joilla ei voi olla optimaalista ratkaisua.

Esimerkki 3.8 Tarkastellaan töidenjärjestelyongelmaa, johon liittyy neljä eri työtä, joiden tekemiseen tarvitaan koneita A, B ja C tässä järjestyksessä. Kone voi tehdä vain yhtä työtä kerrallaan. Jos kone ei ole vapaa, jää työ odottamaan vuoroaan. Missä ajassa voidaan suorittaa ne työt, joiden suoritusajat on annettu vasemmalla taulukossa 3.3?

Jos työt tehdään järjestyksessä $1-2-3-4$, niin aikaa kuluu yhteensä 33 aikayksikköä (taulukkossa 3.3 oikealla). Erilaisia töidentekojärjestyksiä on kaikkiaan $4! = 24$. Branch and bound -tekniikalla voidaan tutkittavien tapausten määrää huomattavasti rajoittaa.

Määritellään rajafunktio b yhden, kahden ja kolmen mittaisille poluille seuraavasti:

$$b(i) = A_i + \sum_{m=1}^{4} B_m + \min\{C_n | n \neq i\}$$

$$b(i,j) = A_i + A_j + \sum_{m=1, m\neq i}^{4} B_m + \min\{C_n | n \neq i, j\}$$

$$b(i,j,k) = A_i + A_j + A_k + \sum_{m=1, m\neq i,j}^{4} B_m + C_n, n \neq i, j, k.$$

Arvoiksi $b(i)$ saadaan

$b(1) = 3 + (4 + 2 + 5 + 5) + 1 = 20$

$b(2) = 7 + (4 + 2 + 5 + 5) + 3 = 26$

$b(3) = 6 + (4 + 2 + 5 + 5) + 1 = 23$

$b(4) = 2 + (4 + 2 + 5 + 5) + 1 = 19.$

Koska $b(4)$ oli $b(i)$-arvoista pienin, jatketaan 4-haaraa:

$b(4, 1) = 2 + 3 + (4 + 2 + 5) + 1 = 17$

$b(4, 2) = 2 + 7 + (4 + 2 + 5) + 3 = 23$

$b(4, 3) = 2 + 6 + (4 + 2 + 5) + 1 = 20.$

Jatketaan 41-haaraa, sillä $b(41)$ oli $b(4j)$-arvoista pienin:

$b(4, 1, 2) = 2 + 3 + 7 + (2 + 5) + 3 = 22$

$b(4, 1, 3) = 2 + 3 + 6 + (2 + 5) + 1 = 19.$

Järjestys $4 - 1 - 3 - 2$ kuluttaa 23 aikayksikköä. Nyt voidaan karsia ne osaratkaisut, joiden b-arvo on 23 tai enemmän. Esimerkiksi mitään kakkosella tai kolmosella alkavia ratkaisuja ei tarvitse kehittää pitemmälle.

Huomaa, kuinka tässä esimerkissä rajafunktiota on käytetty myös valitsemaan se polku, joka edustaa ensimmäistä loppuun asti kehitettyä ratkaisua $(4-1-3-2)$. Tämän ratkaisun arvoa (23) on sitten käytetty vertailtaessa rajafunktion arvoja eri pisteissä. Jos algoritmin suorituksen aikana löydettäisiin kokonaissuoritusajaltaan parempi ratkaisu, ryhdyttäisiin rajafunktion arvoja tietenkin vertailemaan siihen. Tässä esimerkissä ensimmäinen valmiiksi kehitetty ratkaisu oli sattumalta koko ongelman optimiratkaisu (vaikkakaan ei yksikäsitteinen).

□

3.5. Paikalliseen etsintään perustuvat algoritmit

Tarkastellaan ongelmaa, jossa etsitään parasta mahdollista ratkaisua lukuisten vaihtoehtojen joukosta. *Hakuavaruudeksi* kutsutaan tällöin kelvollisten ratkaisukandidaattien joukkoa. Esimerkiksi kauppamatkustajan ongelmassa annettuun syötegraafiin liittyvän hakuavaruuden muodostavat graafin kaikki Hamiltonin silmukat. Ongelmana on löytää tästä joukosta se, jossa mukaan otettujen särmien painojen summa minimoituu. Hakuavaruuden läpikäyntiin liittyy *naapuruuden* käsite. Hakuavaruuden alkioita sanotaan toistensa naapureiksi, jos ne voidaan muuntaa toisikseen yksinkertaisella muunnoksella. Esimerkiksi kauppamatkastajan ongelmassa tällainen muunnos voisi olla ratkaisukandidaatin määrittelevän solmujen permutaation muuntaminen jollakin yksinkertaisella menetelmällä, vaikkapa vaihtamalla kahden peräkkäisen solmun järjestystä.

(1) Aloita satunnaisesti valitusta ratkaisusta.
(2) Sovella ratkaisuun sopivaa muunnosta, jotta saataisiin parempi ratkaisu.
(3) Toista kohtaa (2) niin kauan kuin ratkaisua voidaan parantaa.

Kuva 3.11: Paikalliset muunnokset.

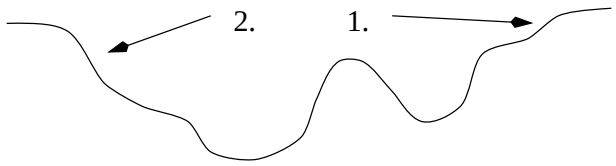
Joidenkin ongelmien ratkaisu löydetäänkin kuvan 3.11 tavalla. Sallitut muunnokset ovat luonteeltaan paikallisia: ratkaisua muunnetaan "lähellä" olevaksi paremmaksi ratkaisuksi, eli etsitään jo löydetyn ratkaisukandidaatin naapurustosta nykyistä parempaa ratkaisua. Tällä menetelmällä ei välttämättä löydetä optimaalista ratkaisua. Pienimmän virittävän puun etsintä on esimerkki ongelmasta, jolle optimi kuitenkin löydetään. Saatava kuvan 3.12 algoritmi on kuitenkin selvästi tehottomampi kuin aikaisemmin esitetty Kruskalin algoritmi, joka toimii ajassa $\theta(e \log e)$, kun e on alkuperäisen graafin särmien lukumäärä.

(1) Valitse jokin virittävä puu T.
(2) Valitse joku puuhun T kuulumaton särmä ja lisää se puuhun T. Puussa T on nyt silmukka. Poista silmukasta se särmä, jonka paino on suurin. T on jälleen virittävä puu.
(3) Toista kohtaa (2), kunnes T ei enää muutu.

Kuva 3.12: Pienin virittävä puu.

Yleisessä tapauksessa optimiratkaisun löytyminen paikallisella etsinnällä riippuu hakuavaruuden rakenteesta ja valituista alkuratkaisuista. Kuvassa 3.13 oleva käyrä esittää ratkaisujen arvoja. Oletetaan, että kyseessä on minimointiongelma eli mitä alempana käyrän piste on, sitä parempaa ratkaisua se vastaa (maksimointiongelman tapauksessa kirjallisuudessa käytetään termiä hill climbing).

Kuvan 3.13 alkuratkaisun 1 avulla voidaan paikallisilla muunnoksilla päästä vain lokaaliin optimiin; alkuratkaisusta 2 voidaan edetä globaaliin optimiin. Seuraavaksi käsitellään etsintämenetelmiä, joita voidaan pitää paikallisen etsinnän yleistyksinä. Niissä voidaan tilapäisesti edetä kohti huonompia ratkaisuja ja näin mahdollisesti välttää huonon alkuratkaisun vaikutus lopputulokseen.



Kuva 3.13: Minimointiongelman ratkaisujen arvoja.

Paikallinen etsintäalgoritmi voidaan muuntaa esimerkiksi *tabuetsinnäksi*. Siinä ylläpidetään listaa niistä naapuruston ratkaisuista, joihin ei ole lupa siirtyä. Tabulistassa

voidaan pitää esimerkiksi niitä ratkaisuja, joita ollaan hiljattain tutkittu. Näin voidaan pakottaa etsintä etenemään aikaisemmin tutkimattomiin hakuavaruuden osiin.

3.5.1. Simuloitu jäähdytys

Simuloidussa jäähdytyksessä jäljitellään metallien valmistuksessa käytettyä menettelyä, jossa pyritään estämään valmistettavan metallin liian nopea jäähtyminen. Sopiva jäähtymisnopeus antaa metallille paremmat ominaisuudet. Paikallisessa etsinnässä liian nopea jäähtyminen vastaa kulkemista aina suoraan kohti lokaalisti parempaa ratkaisua ja jäähtymisen hidastaminen puolestaan vastaa mahdollisuutta valita tietyin ehdoin myös lokaalisti huonoja ratkaisuja.

Simuloitu jäähdytys aloitetaan valitsemalla satunnaisesti jokin alkuratkaisu ja alkulämpötila T. Algoritmin jokaisessa askeleessa tehdään kulloiseenkin ratkaisuun paikallinen muunnos. Jos muunnos parantaa ratkaisua, niin hylätään vanha ratkaisu ja jatketaan uudella ratkaisulla. Vaikka paikallisella muunnoksella löydetty uusi ratkaisu ei olisikaan parempi kuin entinen ratkaisu, voidaan se valita uudeksi ratkaisuksi, jos tietty lämpötilasta riippuva ehto on voimassa: lasketaan entisen ratkaisun ja uuden ratkaisun välinen erotus Δ ja verrataan väliltä [0,1) saatua satunnaislukua arvoon $e^{-\Delta/T}$. Jos saatu satunnaisluku on tätä pienempi, niin löydetty huonompi ratkaisu kelpuutetaan uudeksi ratkaisuksi. Algoritmin edetessä lämpötilaa T lasketaan, jolloin todennäköisyys sille, että huonompi ratkaisu syrjäyttäisi nykyisen ratkaisun, pienenee. Lämpötilan laskiessa simuloitu jäähdytys alkaa siis muistuttaa enemmän ja enemmän tavallista paikallista etsintää. Systeemin toiminta riippuu tavasta laskea lämpötilaa, joten menetelmää voi parametroida valitsemalla erilaisia tapoja lämpötilan laskemiseksi.

3.5.2. Geneettiset algoritmit

Algoritmeja, jotka jäljittelevät luonnossa tapahtuvaa evoluutiota, kutsutaan geneettisiksi algoritmeiksi. Tarkastellaan "populaatiota" (jokin hakuavaruuden osajoukko), jota muunnellaan "mutaatioilla" (paikalliset muunnokset) ja "risteytyksillä" (kahden ratkaisuvaihtoehdon ominaisuuksien vaihtaminen). Oletetaan aluksi yksinkertaisuuden vuoksi, että populaation yksilöt esitetään bittivektoreina (vrt. kromosomin geenit).

Aluksi muodostetaan lähtöpopulaatio joko satunnaismenetelmällä tai jollakin sopivalla heuristiikalla. Populaatiossa olevien yksilöiden hyvyyttä mitataan evaluointifunktiolla, jonka avulla parhaat yksilöt valitaan risteytyksiä varten. Toinen geneettinen operaatio, mutaatio, valitsee satunnaisesti populaation yksilöiden biteistä ne, jotka muunnetaan. Risteytysten ja mutaatioiden avulla muodostetaan uusia sukupolvia, kunnes haluttu "yksilö on syntynyt" (tarpeeksi hyvä ratkaisu on löytynyt) tai kunnes huomataan, että ratkaisut eivät enää parane. Menetelmä voidaan esittää kuvan 3.14 algoritmilla.

Jotta geneettisiä algoritmeja voitaisiin soveltaa, on hakuavaruuden alkiot esitettävä bittijonoina tai muina risteytyksiin soveltuvina tietorakenteina. Lisäksi on kiinnitettävä seuraavat parametrit:

Procedure evoluutio
(1) $t := 0$
(2) Muodosta lähtöpopulaatio $P(0)$
(3) Evaluoi populaatio $P(0)$
(4) **while** pysähtymisehto ei ole voimassa **do**
(5) $\quad t := t + 1$
(6) $\quad$ Muodosta risteytyksillä ja mutaatioilla uusi populaatio $P(t)$
(7) $\quad$ Evaluoi $P(t)$
(8) **od**

Kuva 3.14: Geneettinen algoritmi.

- populaation koko
- risteytymätodennäköisyys
- mutaatiotodennäköisyys.

Esimerkki 3.9 Tarkastellaan esimerkkinä funktion

$$f(x) = x \sin(10\pi x) + 1.0$$

suurimman arvon etsintää välillä $[-1 \ldots 2]$. Tehtävänä on löytää se $x \in [-1 \ldots 2]$, jolla suurin arvo saadaan. Funktion kuvaajan tarkastelu annetulla välillä osoittaa, että paikallinen etsintä olisi suurella todennäköisyydellä hyödytöntä: se luultavasti päätyisi johonkin lukuisista paikallisista maksimeista. (Ongelma olisi tietenkin järkevintä ratkaista analyyttisesti, mutta tässä yhteydessä tyydytään ratkaisua etsimään muilla keinoin.)

Jotta alkiot voidaan esittää bittijonoina, on aluksi sovittava käytettävästä laskutarkkuudesta. Sovitaan, että kuusi desimaalia on sopiva tarkkuus. Koska arvoalueen pituus on kolme, on hakuavaruudessa 3000000 alkiota. Bittijonojen pituudeksi tulee 22, sillä $2097152 = 2^{21} < 3000000 \leq 2^{22} = 4194304$. Esimerkiksi bittijono

$$1000101110110101000111$$

tarkoittaa arvoalueen alkiota 0.637197, sillä $(1000101110110101000111)_2 = 2288967$ ja $-1.0 + 2288967\frac{3}{4194303} = 0.637197$.

Tarkastellaan populaatiota, jossa on yksilöt

$$\begin{aligned} v_1 &= 1000101110110101000111 \\ v_2 &= 0000001110000000010000 \\ v_3 &= 1110000000111111000101. \end{aligned}$$

Evaluointifunktioksi voidaan valita tutkittava funktio f itse. Populaation alkiot vastaavat x:n arvoja 0.637197, -0.958973 ja 1.627888. Vastaavat funktion arvot ovat

$$\begin{aligned} f(v_1) &= f(0.637197) &&= 1.586345 \\ f(v_2) &= f(-0.958973) &&= 0.078878 \\ f(v_3) &= f(1.627888) &&= 2.250650. \end{aligned}$$

Tämän populaation paras yksilö on siis v_3.

Mutaatiossa valitaan satunnaisesti jokin bitti ja muutetaan se toiseksi. Jos esimerkiksi v_3:ssa muutettaisiin viides bitti ykköseksi, saataisiin yksilö 11101000001111111000101, joka vastaa x:n arvoa 1.721638 ja vastaava funktion arvo on -0.082257. Tämä mutaatio johtaisi siis huonompaan yksilöön. Jos mutaatio muuttaisi v_3:n kymmenennen bitin ykköseksi, saataisiin funktion arvoksi 2.343555 eli olisi löydetty aikaisempaa parempi ratkaisu.

Oletetaan seuraavaksi, että v_2 ja v_3 risteytetään keskenään. Valitaan satunnaisesti kohta, josta bittijonot katkaistaan, esimerkiksi viidennen bitin jälkeen,

$$\begin{aligned} v_2 &= 00000|01110000000010000 \\ v_3 &= 11100|00000111111000101 \end{aligned}$$

ja muodostetaan uudet bittijonot vaihtamalla osia bittijonojen välillä:

$$\begin{aligned} v_2^* &= 00000|00000111111000101 \\ v_3^* &= 11100|01110000000010000. \end{aligned}$$

Näihin liittyvät funktion arvot ovat $f(v_2^*) = 0.940865$ ja $f(v_3^*) = 2.459245$. Yksilö v_3^* on siis parempi kuin kumpikaan "vanhemmistaan".

Tässä ongelmassa voitaisiin valita esimerkiksi seuraavat parametrit:

populaation koko = 50
risteytymätodennäköisyys = 0.25
mutaatiotodennäköisyys = 0.01.

Valittu mutaatiotodennäköisyys 0.01 tarkoittaa, että satunnaisesti valitaan populaation joka sadas bitti ja muutetaan se. Risteytymätodennäköisyys 0.25 ilmoittaa populaation "hyvyydeltään" keskimääräisen yksilön todennäköisyyden tulla valituksi mukaan risteytyksiin. Jos evaluointifunktio antaa yksilölle keskimääräistä paremman arvon, sen todennäköisyys tulla valituksi risteytyksiin kasvaa; keskimääräistä huonommilla yksilöillä todennäköisyys on vastaavasti pienempi. Näillä parametreilla päästään hyvin lähelle etsittyä maksimia noin $100 - 150$ sukupolven jälkeen. □

Esimerkki 3.10 Kauppamatkustajan ongelmassa yksilöitä ei kannata esittää bittivektoreina. Jos kauppamatkustajan reittiä esittävä bittivektori katkaistaan mielivaltaisesta paikasta ja yhdistetään toiseen samalla tavalla katkaistuun vektoriin tai mielivaltainen bitti muutetaan, eivät tulokset välttämättä ole enää kauppamatkustajan luvallisia reittejä. Parempi tapa on esittää populaation yksilöt (eli mahdolliset kauppamatkustajan reitit) kokonaislukuvektoreina. Kauppamatkustajan reitti on jokin graafin solmujen järjestysnumeroiden permutaatio. Jos tarkasteltavassa graafissa on n solmua, niin hakuavaruuden muodostavat lukujen $1, 2, \ldots, n$ permutaatiot. Evaluointifunktion määritteleminen on helppoa, sillä annettuun permutaatioon liittyvä kustannus (kauppamatkustajan reitin pituus) voidaan laskea suoraan graafin painoista.

Permutaatioista muodostuvien yksilöiden mutaatio voisi olla esimerkiksi kahden valitun luvun järjestyksen vaihtaminen. Risteytyksessä voidaan toisesta permutaatiosta valita osapermutaatio, joka pidetään muuttumattomana ja pitämällä muiden numeroiden

keskinäinen järjestys samana kuin toisessa permutaatiossa. Esimerkkinä tarkastellaan yksilöiden (eli permutaatioiden) $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)$ ja $(7, 3, 1, 11, 4, 12, 5, 2, 10, 9, 6, 8)$ välistä risteytystä. Valitaan ensiksi mainitusta osa $(4, 5, 6, 7)$, joka pidetään muuttumattomana. Tällöin jälkimmäiseen jäävät luvut $3 - 1 - 11 - 12 - 2 - 10 - 9 - 8$ tässä järjestyksessä. Saadaan siis jälkeläinen $(3, 1, 11, 4, 5, 6, 7, 12, 2, 10, 9, 8)$. □

Geneettisten algoritmien taustalla on ns. *building block -hypoteesi*, jonka mukaan optimaalinen tai lähes optimaalinen ratkaisu voidaan muodostaa pienistä osaratkaisuista, ns. skeemoista, jotka säilyvät risteytyksissä sukupolvesta toiseen. Yllä olevassa esimerkissä osapermutaatio $(4, 5, 6, 7)$ voisi olla tällainen skeema, jonka halutaan säilyvän. Risteytyksen jälkeen se esiintyy nyt uudessa, toivottavasti entistä paremmassa ympäristössä. Sukupolvien saatossa useat tällaiset skeemat toivottavasti yhdistyvät optimiratkaisuksi.

3.6. Muita luontolaskennan etsintämenetelmiä

Luontolaskennalla (natural computing, nature inspired computing) tarkoitetaan luonnossa tapahtuvia laskentaprosesseja tai laskentamenetelmiä, jotka tavalla tai toisella jäljittelevät luonnonilmiöitä. Geneettisten algoritmien lisäksi luontolaskentaan luettavia etsintämenetelmiä ovat esimerkiksi muurahaispopulaatioalgoritmit ja parvialgoritmit.

Muurahaispopulaatioalgoritmit perustuvat muurahaisyhteisöjen ruoanhakuprosessin jäljittelyyn. Kun muurahaiset löytävät ravintoa, ne merkisevät takaisin keolle palatessaan kulkemansa reitin feromoni-nimisellä aineella. Muut muurahaiset voivat feromonijälkiä seuraten löytää saman ravintolähteen. Kun useampi muurahainen kulkee samaa reittiä, niin feromonijälki vahvistuu ja huokuttelee lisää muurahaisia käyttämään samaa reittiä. Kun alun perin löydetty ruokavaranto loppuu, muurahaiset jatkavat ruoan etsimistä muualta, ja ehtyneeltä ruokapaikalta keolle kulkevan reitin feromonijälki haihtuu hiljalleen pois. Näin muurahaisten liikkuminen tehostuu, kun ne seuraavat feromonijälkiä: mitä voimakkaampi jälki on kyseessä, sitä vilkkaampi liikenne ja sitä parempi ravintopaikka.

Etsintäalgoritmeihin sovellettuna muurahaisten toimintaa voidaan jäljitellä niin, että ratkaisukandidaattien osiin (esimerkiksi kauppamatkustajan reitillä oleviin särmiin) lisätään "feromonia" riippuen kandidaatin hyvyydestä. Näin ne osat, jotka ovat mukana useissa keskimääräistä paremmissa ratkaisuissa tulevat merkittyä keskimääräistä voimakkaammin, ja näistä osista voidaan rakentaa uusia, toivottavasti parempia, ratkaisuja.

Parvialgoritmien perusajatuksena on jäljitellä parvissa liikkuvien eläinten, kuten lintujen ja kalojen, käyttäytymistä. Voidaan myös ajatella, että parvi muodostuu elottomista hiukkasista. Parvialgoritmissa osa hakuavaruuden alkioista muodosta parven (vrt. geneettisen algoritmin populaatio), jossa kullakin yksilöllä on kullakin ajanhetkellä sijainti ja nopeus (sijainnin muutos). Tämän lisäksi kukin yksilö muistaa parhaan ratkaisun, jossa se on ollut (yksilökohtainen optimi). Yksilöiden sijannit muuttuvat nopeuden, aikaisemman löydetyn parhaan sijainnin ja naapuruston perusteella. Naapuruston (eli yksilön seuraavaan sijaintiin vaikuttavien muiden yksilöiden) määrittely vaikuttaa huomattavasti algoritmin toimintaan. Karkealla tasolla voi sanoa, että algoritmin toiminta perustuu

siihen, että sekä yksilökohtainen optimi että naapurustossa olevat hyvät ratkaisut vetävät yksilöä puoleensa. Tuloksena on parvi, joka liikkuu kuin yhdessä ruokaa etsivä lintuparvi lähelle hyvyysfunktion optimiarvoa.

3.7. Induktioon perustuva algoritmien suunnittelu

Matematiikasta tuttu induktioperiaate voidaan esittää esimerkiksi seuraavasti: Olkoon $T(n)$ jokin väite, joka halutaan todistaa kaikilla arvoilla $n = 1, 2, 3, \ldots$ Jos on voimassa

- $T(1)$ on tosi,
- Jos $T(i)$ on tosi kaikilla arvoilla $i < k$, niin myös $T(k)$ on tosi,

niin $T(n)$ on tosi kaikilla luonnollisilla luvuilla.

Samantapaista periaatetta voidaan soveltaa myös algoritmeja suunniteltaessa. Tällöin se voidaan kirjoittaa muodossa: yritä ratkaista ongelman tapaus olettaen, että saman ongelman pienemmät tapaukset osataan ratkaista.

Seuraavassa pyritään valaisemaan induktioperiaatteen käyttöä algoritmien suunnittelussa esittelemällä sen soveltamista erilaisissa tilanteissa.

Esimerkki 3.11 (Lajittelu). Yhden alkion lajittelu onnistuu aina. Miten voitaisiin pienentää tapauksen kokoa n:stä $(n-1)$:een? Jos ajatellaan, että $n-1$ alkiota on jo lajiteltu ja n. alkio lisätään omalle paikalle valmiiksi lajiteltujen alkioiden suhteen, menetellään kuten lisäyslajittelussa. Toinen tapa ajatella asiaa on asettaa yksi alkio omalle paikalleen. Tämän jälkeen loput alkiot pannaan joko tämän alkion edelle tai sen jälkeen kokonsa mukaisesti. Molempiin ryhmiin tulee korkeintaan $n-1$ alkiota. "Induktio-oletuksen" mukaan $n-1$ alkiota osataan lajitella. Näin on päädytty pikalajitteluun. □

(1) Pienennä taulukon indeksiä $c[f(i)]$ yhdellä.
(2) Jos se tulee nollaksi, poista $f(i)$ joukosta ja lisää jonoon sitä vastaava elementti.
(3) Toista kohtia (1) ja (2) kunnes jono on tyhjä.

Kuva 3.15: Bijektion etsintä.

Esimerkki 3.12 (Bijektion etsintä). Tarkastellaan äärellisen joukon kuvausta f joukolle itselleen. Otetaan tehtäväksi etsiä tarkasteltavan joukon suurin sellainen osajoukko, että f on tämän osajoukon bijektiivinen kuvaus osajoukolle itselleen. Tarkasteltaessa n-alkioista joukkoa voidaan tehdä "induktio-oletus": ongelma osataan ratkaista $(n-1)$-alkioisilla joukoilla.

Ongelman tapausta voidaan pienentää etsimällä sellainen alkio, joka ei ole minkään alkion kuva kuvauksessa f. Tällainen alkio ei voi kuulua etsittyyn osajoukkoon. Mikäli ei löydy alkiota, joka ei ole minkään alkion kuva, niin kyseessä on bijektio.

Ratkaisun tehostamiseksi otetaan käyttöön taulukko c, jonka i. alkio ilmoittaa, monenko alkion kuva alkio i on. Jos $c[i] = 0$, niin alkio i ei kuulu etsittyyn osajoukkoon.

Tällaiset alkiot poistetaan joukosta, ja viedään jonoon poistettua alkiota vastaava elementti. Jokaista jonossa olevaa alkiota i kohti toimitaan kuvan 3.15 osoittamalla tavalla. □

Esimerkki 3.13 (Intervallien sisältymisongelma). Tarkastellaan erillisiä lukusuoran intervalleja $I_1 = (L_1, R_1)$, $I_2 = (L_2, R_2)$, ..., $I_n = (L_n, R_n)$. Tehtävänä on etsiä ne intervallit, jotka sisältyvät johonkin toiseen intervalliin. Intervalli I_j sisältyy intervalliin I_k, jos $L_k \leq L_j$ ja $R_k \geq R_j$.

Jos yritetään ratkaista ongelmaa siten, että oletetaan ratkaisu tunnetuksi $(n-1)$:n intervallin tapauksessa, niin lisättäessä joukkoon uusi intervalli joudutaan tekemään $n-1$ vertailua. Näin saadaan algoritmi, jonka aikavaatimus on $O(n^2)$.

Aikavaatimusta voidaan pienentää, jos intervalleja tarkastellaan sopivassa järjestyksessä. Oletetaan, että intervallit on lajiteltu nousevaan järjestykseen vasempien päätepisteiden mukaan; jos vasemmat päätepisteet ovat samat, niin intervallit järjestetään laskevaan järjestykseen oikeiden päätepisteiden mukaan.

Tehdään nyt induktio-oletus: ongelma osataan ratkaista, kun järjestettyjä intervalleja on $n-1$ kappaletta. Tarkastellaan n:nnen intervallin lisäämistä. Koska intervallien järjestyksen perusteella tiedetään, että $L_j \leq L_n$, $j = 1, \ldots, n-1$, niin uusi intervalli sisältyy johonkin entiseen intervalliin täsmälleen silloin, kun on sellainen oikea päätepiste R_j, $j < n$, että $R_j \geq R_n$. Ratkaisu yksinkertaistuu, jos tunnetaan suurin jo lisätty oikea päätepiste MaxR. Aikaisempaa induktio-oletusta voidaan nyt täsmentää kirjoittamalla se muotoon: ongelma osataan ratkaista, kun järjestettyjä intervalleja on $n-1$ kappaletta, ja tiedetään niiden suurin oikea päätepiste.

Uuden induktio-oletuksen perusteella saadaan kuvan 3.16 algoritmi, joka merkitsee ne intervallit, jotka kuuluvat johonkin toiseen intervalliin. Algoritmin aikavaatimus on $O(n \log n)$. □

```
(1) Järjestä intervallit tekstissä esitetyllä tavalla
(2) MaxR := R_1
(3) for j := 2 to n do
(4)         if R_j ≤ MaxR then
(5)                 Merkitse intervalli j
(6)         else
(7)                 MaxR := R_j
(8)         end
(9) od
```

Kuva 3.16: Intervallien sisältyminen.

Esimerkki 3.14 (Julkkisongelma). Tarkastellaan n:stä henkilöstä muodostuvaa ryhmää. Henkilö on *julkkis*, jos hänet tuntee ryhmän kaikki muut $n-1$ jäsentä, mutta hän itse ei tunne ketään muuta. Ongelmana on selvittää, onko tarkasteltavana olevassa ryhmässä julkkista, esittämällä ryhmän jäsenille kysymyksiä "Tunnetko tuon henkilön?". Kuinka

monta kysymystä on esitettävä? (Oletetaan, että kaikki vastaukset ovat oikeita ja mahdollinen julkkiskin suostuu vastaamaan.)

Tuttavuussuhteet voidaan esittää suunnattuna graafina, jossa on solmu jokaista henkilöä kohti ja suunnattu kaari (u, v), jos henkilö u tuntee henkilön v. Julkkista vastaa solmu, johon tulee kaari kaikista muista solmuista, mutta josta ei lähde yhtään kaarta. Tällaista solmua sanotaan graafin *nieluksi* (sink).

Jos ongelma on jo ratkaistu $(n-1)$:n kokoiselle ryhmällä, niin miten tilanne muuttuu, kun lisätään tarkasteltavaan ryhmään yksi henkilö? Erilaisia vaihtoehtoja on kolme:

1. Julkkis on $(n-1)$:n ensimmäisen henkilön joukossa. (Ryhmässä voi olla korkeintaan yksi julkkis.)

2. Viimeksi lisätty henkilö on julkkis.

3. Julkkista ei ole lainkaan.

Tapauksessa 1 on tarkastettava, että julkkisehto on voimassa myös lisäyksen jälkeen. Tähän tarvitaan kaksi kysymystä. Tapauksissa 2 ja 3 kysymyksiä tarvitaan $2(n-1)$ kappaletta, joten aikavaatimukseksi tulee $\sum_{i=1}^{n} 2(i-1) = O(n^2)$.

Parempi tulos saadaan, kun huomioidaan, että jokainen kysymys pienentää tarkasteltavaa joukkoa yhdellä. Jos nimittäin tiedetään, että henkilö u tuntee henkilön v, niin ainakaan u ei voi olla julkkis. Jos taas henkilö u ei tunne henkilöä v, niin v ei voi olla julkkis. Näin jatkamalla päädytään tilanteeseen, jossa vain yksi henkilö voi olla julkkis; tarkastetaan lopuksi erikseen, onko jäljelle jäänyt henkilö julkkis vai ei.

Saadaan kuvan 3.17 algoritmi, joka käyttää tuttavuusgraafin yhteysmatriisia, johon viitataan nimellä Know.

```
(1)  i := 1
(2)  j := 2
(3)  next := 2
(4)  while next< n + 1 do
(5)        next := next + 1
(6)        if Know[i, j] then
(7)              i := next
(8)        else
(9)              j := next
(10)       end
(11) od
(12) if i = n + 1 then ehdokas:= j else ehdokas:= i end
(13) Tarkasta, onko ehdokas julkkis
```

Kuva 3.17: Julkkisongelman ratkaisu.

Algoritmi tarvitsee vain $3(n-1)$ kysymystä, sillä julkkisehdokkaan löytämiseksi tarvitaan korkeintaan $n-1$ kysymystä ja löydetyn ehdokkaan tutkimiseksi korkeintaan $2(n-1)$ kysymystä. □

3.8. Approksimointialgoritmeista

Monien ongelmien täsmällinen ratkaiseminen on niin työlästä, että kannattaa käyttää approksimointialgoritmia, joka tarkan ratkaisun sijasta antaa jonkun riittävän lähellä oikeaa ratkaisua olevan ratkaisun.

Esimerkki 3.15 Boolen lausekkeiden toteutuvuusongelmassa tarkastellaan vakioista 1 (tosi) ja 0 (epätosi), muuttujista $(x_1, x_2, \ldots)$ ja konnektiiveista ($\wedge$, $\vee$, $\neg$) muodostuvia lausekkeita. Lauseke on *toteutuva*, jos siinä esiintyvillä muuttujilla on totuusjakelu, joka antaa koko lausekkeen arvoksi 1.

Esimerkiksi lauseke $F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \wedge x_3)$ on toteutuva, sillä totuusjakelu $x_1 = 1$, $x_2 = 0$ ja $x_3 = 1$ antaa sen arvoksi toden.

Toteutusvuusongelma voidaan tietenkin aina ratkaista käymällä läpi kaikki mahdolliset 2^n totuusjakelua. Menettely vaatii kuitenkin eksponenttiaalisen ajan, eikä ongelmalle tunneta polynomista ratkaisualgoritmia. Toisaalta jokaisesta ratkaisukandidaatista (eli totuusjakelusta) voidaan polynomisessa ajassa tarkistaa, toteuttaako se annetun Boolen lausekkeen. Tämä on mahdollista kuvan 3.18 epädeterministisellä algoritmilla.

(1) arvaa totuusjakelu
(2) **if** arvattu jakelu toteuttaa lausekkeen **then** lauseke on toteutuva
(3) **else** lauseke ei ole toteutuva **end**

Kuva 3.18: Toteutuvuusongelman epädeterministinen ratkaisualgoritmi.

Ongelman sanotaan kuuluvan luokkaan **NP**, jos se voidaan ratkaista kuvan 3.18 kaltaisella arvausmenetelmällä polynomisessa ajassa. Luokkaan **NP** kuuluminen tarkoittaa käytännössä siis sitä, että ratkaisun oikeellisuus voidaan tarkistaa polynomisessa ajassa. Yllä olevan perusteella toteutuvuusongelma kuuluu luokkaan **NP**. □

Luokan **NP** tärkeä osaluokka on niin sanotut **NP**-täydelliset ongelmat. **NP**-täydelliset ongelmat ovat läheisessä suhteessa toisiinsa: ne voidaan "palauttaa" toisikseen polynomisella kuvauksella. Tästä seuraa, että jos yhdelle niistä löydettäisiin polynomisessa ajassa toimiva ratkaisualgoritmi, oltaisiin samalla löydetty polynominen ratkaisualgoritmi niille kaikille. Määritelmän mukaisesti **NP**-täydellisyys tarkoittaa vielä enempää, nimittäin sitä, että mikä tahansa luokan **NP** ongelma on palautettavissa **NP**-täydelliseen ongelmaan polymisella kuvauksella.

NP-täydellisten ongelmien olemassaolo todistetaan siten, että mielivaltaisen polynomisessa ajassa toimivan algoritmin kuvaus palautetaan ko. ongelmaan. Tässä käytetään ns. epädeterministisiä Turingin koneita, joita tarkastellaan opintojaksolla *Automaatit*.

Tässä yhteydessä riittää todeta, että toteutuvuusongelma ja muut tämän kohdan ongelmat (esim. solmupeitteen minimointi, klikkiongelma ja kauppamatkustajan ongelma) ovat **NP**-täydellisiä.

3.8.1. Optimointiongelmat

Optimointiongelma on kolmikko $p = (D, S, c)$, kun

D on ongelman *tapausten joukko*,
S on *mahdollisten ratkaisujen joukko*; kuhunkin tapaukseen $x \in D$ liittyy osajoukko $S(x) \subseteq S$, ja
c on ratkaisujen *kustannusfunktio* $c : D \times S \to \mathbb{R}$.

Minimointiongelman tapauksen $x \in D$ ratkaisu $s^* \in S(x)$ on optimaalinen, jos

$$c(x, s^*) \leq c(x, s)$$

kaikilla $s \in S(x)$ ja maksimointiongelmalle vastaavasti $c(x, s^*) \geq c(x, s)$ kaikilla $s \in S(x)$.

Käytetään tapauksen x optimiratkaisun kustannuksesta merkintää $c^*(x)$. Ratkaisun $s \in S(x)$ *hyvyys* $r(x, s)$ määritellään kaavalla

$$r(x, s) = \frac{c(x, s) - c^*(x)}{c^*(x)}, \text{ kun } c^*(x) \neq 0.$$

Maksimointiongelmalle määritellään vastaavasti

$$r(x, s) = \frac{c^*(x) - c(x, s)}{c(x, s)}, \text{kun } c(x, s) \neq 0.$$

Minimointiongelmalla on aina $0 \leq r(x, s) < \infty$, ja $r(x, s) = 0$ jos ja vain jos s on optimiratkaisu.

Algoritmi A on *approksimointialgoritmi*, jos se antaa ongelman jokaiselle tapaukselle x jonkin mahdollisen ratkaisun, ts. $A(x) \in S(x)$ kaikilla ongelman tapauksilla x. A on *ϵ-approksimointialgoritmi*, $\epsilon \geq 0$, jos kaikki sen tuottamat ratkaisut ovat ϵ-hyviä eli $r(x, A(x)) \leq \epsilon$, kaikilla $x \in D$.

Esimerkki 3.16 Graafin solmupeite on sellainen solmujoukko, että siihen kuuluu ainakin toinen jokaisen särmän päätepisteistä. Solmupeiteongelmassa on tehtävänä löytää pienin mahdollinen solmupeite annetulle graafille. Tarkastellaan solmupeiteongelmaa optimointiongelmana $VC = (D, S, c)$, kun D on kaikkien mahdollisten graafien joukko, $S(G)$ on graafin G solmupeitteiden joukko C ja $c(G, C)$ on solmupeitteen C koko $|C|$.

Yksinkertainen approksimointialgoritmi graafin $G = (V, E)$ solmupeitteen minimoimiseksi saadaan kuvassa 3.19 esitetyllä tavalla.

Optimipeitteeseen on kuuluttava ainakin toinen päätesolmu jokaisesta algoritmin valitsemasta särmästä, joten voidaan kirjoittaa $|C^*| \geq |C|/2$ ja $r(G, C) = (|C| - |C^*|)/|C^*| \leq 1$. Kuvan 3.19 algoritmi on siis 1-approksimointialgoritmi. □

Solmupeiteongelman läheinen sukulaisongelma on ns. klikkiongelma. Siinäkin ongelman tapauksen muodostaa suuntaamaton graafi. Klikkiongelmassa kysytään, mikä on annetun graafin suurimman täydellisen aligraafin (eli klikin) koko. (Täydellisellä graafilla tarkoitetaan graafia, jossa mukana on kaikki mahdolliset särmät, eli kaikilla solmupareilla on välitön yhteys.)

(1) C on aluksi tyhjä joukko
(2) **while** E on ei-tyhjä **do**
(3) Valitse joukosta E särmä (u, v)
(4) Lisää solmut u ja v joukkoon C
(5) Poista joukosta E kaikki särmät, jotka päättyvät solmuun u tai solmuun v
(6) **od**
(7) **return** C

Kuva 3.19: Solmupeitteen minimointi.

Kolmas samaan "ongelmaperheeseen" kuuluva ongelma on riippumattoman solmujoukon etsintä. Siinä tehtävänä on etsiä suurin sellainen solmujoukko, että mitään sen solmuparia ei yhdistä graafin särmä. Helposti huomataan, että solmupeiteen etsintä on sama ongelma kuin riippumattoman solmujoukon etsintä komplementtigraafista.

Esimerkki 3.17 Graafi $G = (V, E)$ on *puolittuva*, jos V voidaan jakaa kahteen osaan niin, että jokainen särmä yhdistää eri osiin kuuluvia solmuja. Graafin piirto-ongelmassa tavoitteena on piirtää graafi tasolle (tai muulle pinnalle) optimaalisesti jonkun ns. estetiikkakriteerin suhteen. Usein käytetty estetiikkakriteeri on särmien leikkausten minimointi, ja sitä on luonnollista käyttää myös puolittuvien graafien piirtämisessä.

Olkoon graafi G puolittuva ja olkoot U ja D ne joukon V osat, joita kaikki joukon E särmät yhdistävät. Sovitaan, että joukon U solmut (ns. ylätason solmut) sijoitetaan suoralle erillisiin pisteisiin ja joukon D solmut (ns. alatason solmut) toiselle samansuuntaiselle suoralle erillisiin pisteisiin. Nyt piirrokseen tulevien särmien risteämien lukumäärä riippuu ainoastaan solmujen järjestyksistä suorilla.

Puolittuvien graafien piirto-ongelmassa on kaksi perustyyppiä: solmuja voidaan järjestää joko molemmilla suorilla tai olettaa, että toisella suoralla olevien pisteiden järjestys on kiinteä, jolloin ongelmana on vain toisen suoran solmujen järjestäminen. Tässä tarkastellaan jälkimmäistä ongelmaa.

Oletetaan, että ylätasolla on n solmua, joiden keskinäinen järjestys on kiinnitetty. Tällöin alatason astetta d olevaan solmuun liittyy numerosarja $(x_1, x_2, \ldots, x_d)$, joka kertoo kyseiselle solmulle vierekkäisten ylätason solmujen sijainnit. Tehtävänä on nyt näiden numerosarjojen perusteella järjestää alatason solmut niin, että särmien leikkausten määrä minimoituu.

Ns. *keskiarvoheuristiikassa* lasketaan alatason jokaiseen solmuun liittyvien ylätason solmujen sijaintien keskiarvo ja järjestetään alatason solmut keskiarvojen mukaan nousevaan järjestykseen. (Jos keskiarvot ovat samat, solmujen järjestys on mielivaltainen.) Keskiarvoheuristiikka toimii käytännössä erittäin hyvin, mutta sen tuottamat ratkaisut eivät välttämättä ole ϵ-hyviä millään ϵ:n arvolla. Tämä voidaan näyttää seuraavasti: Tarkastellaan alatason solmuja u ja v. Oletetaan, että u on rinnakkainen vain ylätason paikassa $d+2$ olevan solmun kanssa. Oletetaan lisäksi, että v on rinnakkainen ylätason paikoissa $1, 2, \ldots, d$ ja $(d+2)^2$ olevien solmujen kanssa. Solmuun u liittyy keskiarvo $d+2$ ja solmuun v keskiarvo $(\sum_{j=1}^{d} i + (d+2)^2)/(d+1) > d+2$. Keskiarvoheuristiikka sijoittaa siis solmun

u solmun v vasemmalle puolelle. Tuloksena on $d+1$ risteämää. Solmujen päinvastaiseen järjestykseen liittyy vain yksi risteämä. Kun d kasvaa, kasvaa myös heuristiikan tekemä suhteellinen virhe.

Ns. *mediaaniheuristiikassa* käytetään keskiarvojen sijasta mediaaneja. Jos solmu on rinnakkainen parillisen määrän ylätason solmuja kanssa, niin voidaan sopia käytettäväksi ylämediaania. Mediaaniheuristiikan voidaan todistaa olevan 2-approksimointialgoritmi eli sen antama tulos on aina korkeintaan kolme kertaa niin huono kuin solmujen optimaalinen järjestys. Tulos voidaan todistaa tarkastelmalla mielivaltaisen solmuparin järjestämistä ja käymällä läpi eri vaihtoehdot solmujen parillisuuden ja parittomuuden suhteen. Tässä todistus kuitenkin sivuutetaan. Vaikka keskiarvoheuristiikan suhteelliselle virheelle ei ole ylärajaa, toimii heuristiikka käytännössä yleensä paremmin kuin mediaaniheuristiikka. □

3.8.2. ΔTSP

Kauppamatkustajan ongelma on esimerkki sellaisista ongelmista, joilla ei ole olemassa ϵ-approksimointialgoritmia millään ϵ:n arvolla. Tässä alakohdassa tarkastellaan sellaista kauppamatkustajan ongelman erikoistapausta, jossa särmiin liittyvä kustannusfunktio d toteuttaa kolmioepäyhtälön, ts. kaikilla solmukolmikoilla u, v ja w on voimassa $d(u,v) \leq d(u,w) + d(w,v)$. Tästä ongelmasta käytetään lyhennystä ΔTSP. Alkuperäisestä kauppamatkustajan ongelmasta poiketen ΔTSP:llä on ϵ-approksimointialgoritmeja. Annetaan ΔTSP:lle aluksi yksinkertainen 1-approksimointialgoritmi (kuva 3.20).

Syöte: $G = (V, E, d)$

(1) Määrää graafille G pienin virittävä puu T.

(2) Muodosta T:stä silmukka P' kiertämällä se.

(3) Muodosta kauppamatkustajan silmukka P oikaisemalla toiseen kertaan käytävien solmujen ohitse.

Kuva 3.20: ΔTSP:n approksimointialgoritmi.

Kaikista Hamiltonin silmukoista, siis myös optimisilmukasta P^*, saadaan virittävä puu poistamalla yksi särmä. Koska T on pienin virittävä puu, niin siihen liittyvä kustannus $c(T)$ on pienempi kuin optimisilmukan kustannus $c(P^*)$. Reitin P' kustannus $c(P')$ on kaksi kertaa niin suuri kuin pienimpään virittävään puuhun liittyvä kustannus. Koska d toteuttaa kolmioepäyhtälön, niin algoritmin askeleessa (3) tehtävä muutos ei ainakaan pidennä kuljettavaa matkaa. Näiden huomioiden perusteella voidaan kirjoittaa $c(P) \leq c(P') = 2c(T) \leq 2c(P^*)$. Kyseessä on siis 1-approksimointialgoritmi.

Tarkastellaan vielä toista ΔTSP-ongelman approksimointialgoritmia, ns. Christofidesin algoritmia (kuva 3.21). Rivillä (3) tarvitaan pariutuksia: solmujoukon *täydelliseksi pariutukseksi* sanotaan särmäjoukkoa, jossa jokaiseen solmuun liittyy täsmälleen yksi joukon särmä. (Pariutuksia käsitellään tarkemmin kohdassa 6.4.) Paria $G = (V, E)$ sanotaan *monigraafiksi*, jos se eroaa graafista vain siinä, että kahden solmun välillä voi olla useampia kuin yksi särmä. Seuraavassa käytetään Eulerin vuonna 1736 todistamaa lausetta, jonka

mukaan suuntaamattomassa yhtenäisessä monigraafissa on Eulerin silmukka, jos ja vain jos monigraafin kaikkien solmujen asteluku on parillinen. (Graafin tai monigraafin Eulerin silmukka on polku, joka kulkee täsmälleen kerran graafin kaikkien särmien kautta ja palaa lähtöpisteeseensä.)

Syöte: $G = (V, E, d)$

(1) Määrää graafin G pienin virittävä puu T.

(2) Olkoon V' niiden graafin G solmujen joukko, joiden asteluku puussa T on pariton (joukossa V' on parillinen määrä solmuja).

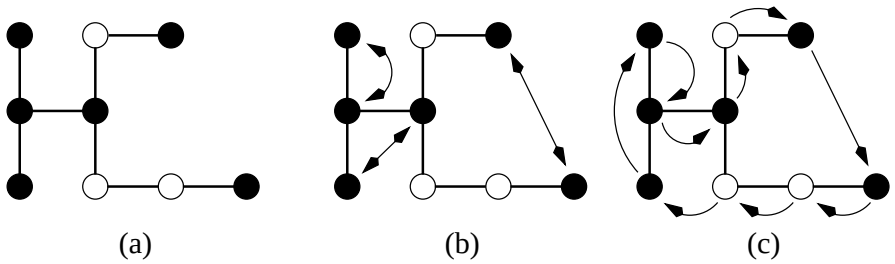
(3) Muodosta joukon V' solmujen täydellinen pariutus E, jonka kustannus on pienin mahdollinen. (Tämä voidaan tehdä tehokkaasti, vaikkakin melko hankalalla algoritmilla.)

(4) Muodosta Eulerin silmukka P' monigraafissa $(V, T \cup E)$.

(5) Muodosta silmukasta P' kauppamatkustajan reitti P oikaisemalla niiden solmujen ohi, joissa on jo käyty.

Kuva 3.21: Cristofidesin approksimointialgoritmi.

Tarkastellaan esimerkkinä kuvan 3.22 (a) virittävää puuta, jonka paritonta astetta olevat solmut on väritetty mustiksi. Algoritmin eteneminen riippuu siitä, mikä on mustien solmujen kustannukseltaan pienin pariutus. Oletetaan, että kustannukseltaan pienin pariutus on se, joka on merkitty nuolilla kuvaan 3.22 (b). Lähtemällä liikkeelle vasemmassa yläkulmassa olevasta solmusta ja kulkemalla myötäpäivään saadaan oikaisemalla kuvassa 3.22 (c) esitetty kauppamatkustajan reitti (nuolet).



Kuva 3.22: Kauppamatkustajan ongelmaan liittyvä virittävä puu (a), sen parittomien solmujen pariutus (b) ja oikaisemalla saatu optimireitti (c).

Tutkitaan vielä Christofidesin algoritmin hyvyyttä. Jos optimisilmukasta P^* muodostetaan oikaisemalla vain joukon V' solmut kiertävä silmukka, niin tämä määrittelee kaksi joukon V' solmujen pariutusta. Lyhyempi näistä on kustannukseltaan pienempi kuin $c(P^*)/2$. Koska kuitenkin E on joukon V' minimaalinen pariutus, niin $c(P) \leq c(T) + c(E) \leq c(P^*) + c(P^*)/2 = 3c(P^*)/2$. Christofidesin algoritmi on siis 1/2-approksimointialgoritmi.

Luku 4

Joukkojen algoritmisesta käsittelystä

Monet käytännön ongelmat voidaan esittää erilaisten alkiojoukkojen avulla. Joukkojen käsittelyssä käytetyt tavallisimmat operaatiot ovat seuraavat (a_i:t ovat alkioita ja S_i:t joukkoja):

member(a, S):	$a \in S$?
insert(a, S):	$S \leftarrow S \cup \{a\}$
delete(a, S):	$S \leftarrow S \setminus \{a\}$
replace(a_i, a_j, S):	$S \leftarrow \big(S \setminus \{a_i\}\big) \cup \{a_j\}$
union(S_i, S_j):	$S_i \leftarrow S_i \cup S_j$
find(a):	jos $a \in \bigcup_i S_i$, niin se i, jolla $a \in S_i$, muutoin määrittelemätön.

Jos alkioiden välillä on määritelty järjestys, niin voidaan käyttää lisäksi seuraavia operaatioita:

min(S):	sellainen alkio a, että $a \leq b$, kaikilla alkioilla $b \in S$
max(S):	vastaavasti kuten min(S)
split(S_i, a, S_j):	jos $a \in S_i$, niin $S_j \leftarrow \{b \in S_i \mid b > a\}$ ja $S_i \leftarrow \{b \in S_i \mid b \leq a\}$
concatenate(S_i, S_j):	jos $\max(S_i) < \min(S_j)$, niin $S_i \leftarrow S_i \cup S_j$.

Esimerkki 4.1 Joukko-operaatioilla voidaan ratkaista useita ongelma:

(a) Etsintä- ja päivitystehtävät muodostuvat jonosta operaatioita member, insert, delete ja replace.

(b) Kruskalin algoritmi käyttää operaatioita insert, min, delete, find ja union.

(c) Prioriteettijono toteutetaan operaatioilla insert, delete ja min. □

4.1. Erillisten joukkojen käsittelyongelma

Oletetaan, että alkutilanteessa tarkasteltavat n alkiota muodostavat kukin oman joukkonsa. Tehtävänä on suorittaa mielivaltainen jono näihin joukkoihin kohdistuvia operaa-

tioita union ja find. Koska kukin tarkasteltavista alkioista kuuluu aina täsmälleen yhteen joukkoon, käytetään tästä ongelmasta nimeä *erillisten joukkojen käsittelyongelma*.

Erillisten joukkojen käsittelyongelmalle on helppo löytää yksinkertaisia (mutta hitaita) ratkaisuja:

- Käytetään taulukkoa $R[1..n]$, jonka alkio $R[i]$ sisältää sen joukon nimen, johon alkio i kuuluu. Alkutilanteessa on $R[i] = i$ kaikilla alkioilla i. Operaatio find(i) voidaan suorittaa vakioajassa lukemalla $R[i]$. Sen sijaan union-operaatio on tehoton, sillä on käytävä läpi koko taulukko ja päivitettävä niiden alkioiden tietoja, jotka kuuluvat yhdistettäviin joukkoihin.

- Samantapainen ratkaisu saadaan, kun linkitetään samaan joukkoon kuuluvat alkiot. Tässä toteutuksessa union voidaan toteuttaa vakioajassa, mutta find on tehoton.

Ongelman tehokkaat ratkaisut perustuvat joukkojen esittämiseen puina. Puun juurena on se alkio, jonka mukaan joukko on nimetty (ns. kanoninen alkio). Muina solmuina puussa on joukon muut alkiot. Puun särminä on kohti juurta suunnatut osoittimet. Find-operaatio suoritetaan paikallistamalla etsittävä alkio jostain puusta ja seuraamalla osoittimia puun juureen, josta löydetään joukon nimi.

Merkinnällä union(a, b, c) tarkoitetaan operaatiota, jossa joukot a ja b yhdistetään uudeksi joukoksi c. Puutoteutuksessa union(a, b, c) suoritetaan niin, että tehdään toisen puun juuresta toisen puun juuren lapsi. Union-operaation aikavaatimus on nyt $O(1)$ ja find-operaation aikavaatimus riippuu läpikäytävän polun pituudesta.

Tarkastellaan operaatiojonoa union$(1, 2, 2)$, union$(2, 3, 3), \ldots,$ union$(n-1, n, n)$. Tuloksena on listamainen puu, jonka juuri on alkio n. Tehdään nyt find-operaatiot find(1), find(2), ..., find(n). Find-operaatioiden aikavaatimus on $\sum_{i=1}^{n} i = O(n^2)$.

Puutoteutusta voidaan huomattavasti tehostaa sekä union- että find-operaatioiden osalta. Sanotaan, että union-operaatio tehdään koon perusteella, jos pienemmän puun juuresta tehdään suuremman puun juuren lapsi. (Puun koolla tarkoitetaan sen solmujen lukumäärää.) Induktiolla voidaan helposti todistaa, että kun union-operaatiot tehdään koon mukaan, niin mielivaltaisen union-operaatioista koostuvan jonon suorituksen jälkeen voi puun korkeus olla h tai enemmän vain, jos puussa on vähintään 2^h solmua.

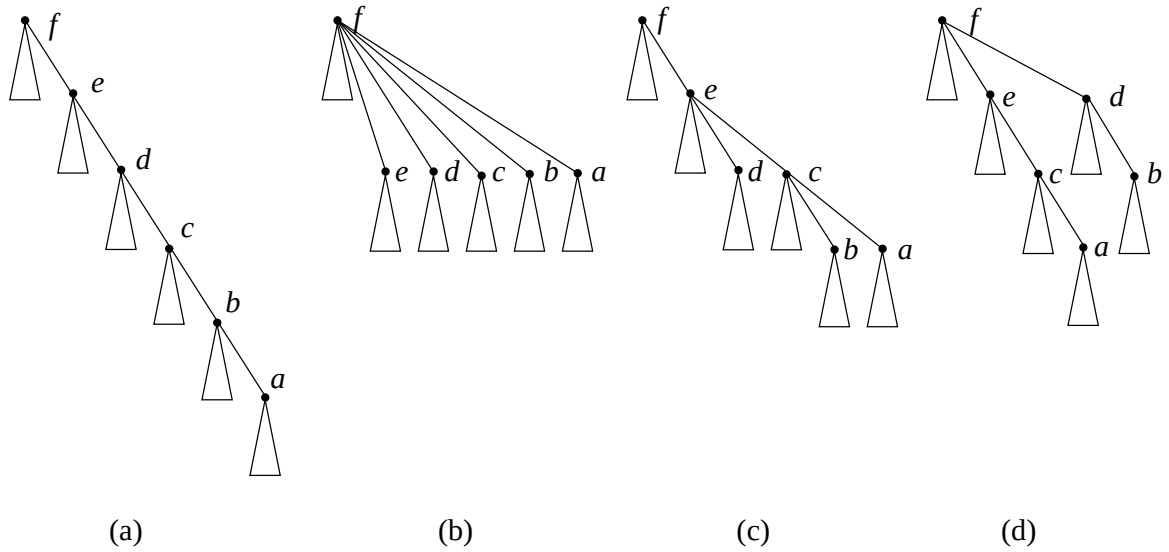
Jos operaatiot tehdään koon perusteella, niin mikään find-operaatio ei vaadi aikaa enempää kuin $O(\log n)$. Samaan tulokseen päästään, jos union-operaatiot tehdään korkeuden perusteella (matalamman puun juuresta tehdään korkeamman puun juuren lapsi). Korkeuden perusteella yhdistettäessä puiden korkeus kasvaa vain, kun yhdistetään yhtä korkeat puut.

Myös find-operaatioita voidaan tehostaa. Tavoitteena on hakupolkujen lyhentäminen. Seuraavassa esitellään kolme tavallisinta tapaa find-operaatioiden tehostamiseksi. (Kuvassa 4.1 kolmiot esittävät mielivaltaisia alipuita.)

1. *Tiivistäminen* (compression). Tehdään kaikista hakupolulla olevista solmuista juuren lapsia (kuva 4.1 (b)).

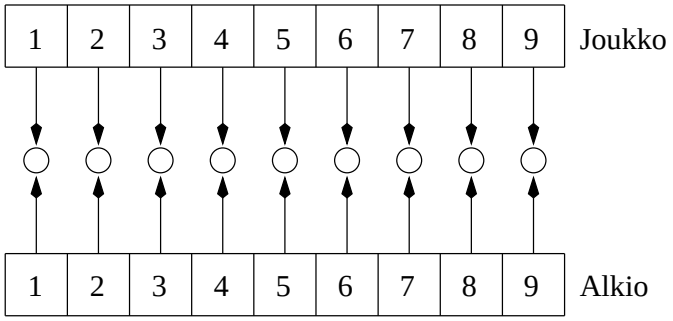
2. *Puolitus* (halving). Tehdään joka toisesta hakupolulla olevasta solmusta (ei kuitenkaan viimeisestä eikä toiseksi viimeisestä) isovanhempansa lapsia (kuva 4.1 (c)).

3. *Halkaisu* (splitting). Tehdään jokaisesta hakupolun solmusta (paitsi viimeisestä ja toiseksi viimeisestä) isovanhempansa lapsi (kuva 4.1 (d)).



Kuva 4.1: (a) Alkuperäinen, (b) tiivistetty, (c) puolitettu ja (d) halkaisu puu.

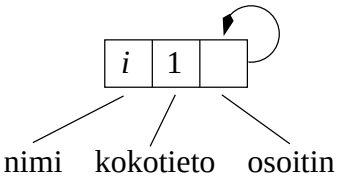
Tarkastellaan seuraavaksi erillisten joukkojen käsittelyongelman puutoteutusta hieman tarkemmin. Union-operaatioiden suorittamiseksi on oltava käytettävissä osoittimet puiden juuriin. Toisaalta find-operaatiot tarvitsevat osoittimet kuhunkin alkioon. Alkutilanteessa jokainen alkio muodostaa oman puunsa. Alkutilanne on esitetty kuvassa 4.2.



Kuva 4.2: Erillisten joukkojen käsittelyalgoritmin alkutilanne.

Kun käytetään yhdistämistä koon mukaan (tai korkeuden perusteella), on jokaisessa solmussa oltava tilaa koko- tai korkeustiedon tallentamista varten. Solmuun on tietenkin varattava tilaa myös sen nimeä (numero väliltä $1..n$) varten. Lisäksi tarvitaan osoitin hakupoluilla etenemistä varten. Puun juuri tunnetaan siitä, että sen osoitin osoittaa solmuun

itseensä. Kun yhdistäminen tehdään koon perusteella, niin solmu i on aluksi kuvassa 4.3 esitetyn kaltainen.



Kuva 4.3: Solmu alkutilanteessa, kun käytetään yhdistämistä koon mukaan.

Union-operaatiossa yhdistettäviä joukkoja edustavien puiden juuret löydetään joukkotaulukon avulla. Tämän jälkeen verrataan solmujen kokotietoja. Pienemmän puun juuren osoitin muutetaan osoittamaan suuremman puun juurta. Uuden puun kokotietoa on päivitettävä lisäämällä siihen pienemmän puun koko. Pienemmän puun kokotiedolla ei tämän jälkeen ole mitään merkitystä.

Erillisten joukkojen käsittelyongelman aikavaatimus esitetään tarkasteltavissa operaatiojonoissa olevien union-operaatioiden lukumäärän n ja find-operaatioiden lukumäärän m funktiona. Oletetaan aluksi, että käytetään sitä puutoteutuksen perusmuotoa, jossa yhdistäminen tehdään puita tutkimatta eikä hakupolkuja pyritä lyhentämään. Aikaisemmin ollaan huomattu, että tapauksessa voi muodostua listaksi kutistunut puu. Kun tarkastellaan operaatiojonoa, jossa on aluksi n union-operaatiota ja niiden jälkeen m kappaletta muodostetun puun ainoaan lehtisolmuun kohdistuvia find-operaatioita, niin huomataan, että pahimmassa tapauksessa perusmuodon aikavaatimus on $n + mn$ eli $\theta(mn)$.

Jos yhdistäminen tehdään koon (tai korkeuden) perusteella, mutta find-operaatiot kuten perusmuodossa, niin pahimman tapauksen aikavaatimukseksi saadaan $n + m \log n$. Jos $m \geq n$, niin aikavaatimus on $\theta(m \log n)$. Se tapaus, jossa samanaikaisesti käytetään sekä union- että find-operaatioiden tehokkaampia versioita, on edellisiä huomattavasti monimutkaisempi. Voidaan todistaa, että toteuttamalla molemmat tehostuskeinot yhdessä saadaan algoritmi, jonka aikavaatimus kaikissa käytännön tilanteissa on lineaarinen suoritettujen find-operaatioiden lukumäärän suhteen. Tällöin aikavaativuuden voidaan todistaa olevan muotoa $\theta(m\alpha(m,n))$, kun $m \geq n$ ja α on ns. Ackermannin funktion eräänlainen käänteisfunktio.

Ackermannin funktio määritellään muodossa

$$A(i,j) = \begin{cases} 2^j, & \text{kun } i = 1, j \geq 1, \\ A(i-1,2), & \text{kun } i \geq 2, j = 1, \\ A(i-1, A(i,j-1)), & \text{kun } i,j \geq 2. \end{cases}$$

Funktio α määritellään nyt A:n avulla seuraavasti:

$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor \frac{m}{n} \rfloor) > \log n\}.$$

A kasvaa räjähdysmäisen nopeasti, mistä seuraa, että α kasvaa hyvin hitaasti.

Koska esimerkiksi $A(3,1) = A(2,2) = A(1,A(2,1)) = A(1,A(1,2)) = A(1,4) = 2^4 = 16$ ja $2^{16} = 65536$, niin $\alpha(m,n) \leq 3$, kun $n < 65536$. Kaikissa käytännön tilanteissa voidaan olettaa, että $\alpha(m,n) \leq 5$.

Kun siis tehdään yhdistämiset koon tai korkeuden perusteella ja find-operaatiot tiivistäen, halkaisten tai puolittain, saadaan algoritmi, joka aikavaativuudeltaan on lähes lineaarinen. Lisäksi voidaan todistaa, että saatu algoritmi on optimaalinen ns. separoituvien algoritmien luokassa.

4.2. Determinististen äärellisten automaattien ekvivalenttisuus

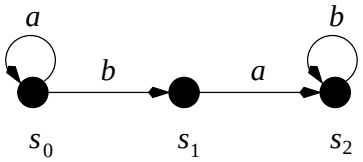
Äärellinen automaatti on laskulaite, jolla on äärellinen tilasysteemi mutta jolta puuttuu muisti. Äärellisen automaatin toiminta voi haarautua tilan ja syötemerkin perusteella. Koska tilajoukko on äärellinen, voi äärellinen automaatti "muistaa" vain lukuja, joiden koolle saadaan yläraja, kun tiedetään automaatin tilojen lukumäärä. Äärellisten automaattien tunnistamien kielten luokkaa kutsutaan säännöllisiksi kieliksi. Esimerkiksi kielet $\{a^i b^j | i = j\}$ ja $\{a^i b^j | i \geq j\}$ eivät kuuluu tähän luokkaan.

Täsmällisemmin äärellinen automaatti määritellään järjestelmänä $M = (S, \Sigma, \delta, s_0, F)$, kun

- S on äärellinen tilojen joukko,
- Σ on syötemerkkien joukko,
- δ on siirtymärelaatio $S \times \Sigma \to S$,
- s_0 on alkutila ja
- F on lopputilojen joukko.

Äärellinen automaatti on deterministinen, jos jokaisella parilla $(q,x)(\in (S \times \Sigma))$ siirtymärelaatio δ määrää yksikäsitteisesti automaatin seuraavan tilan tai $\delta(q,x)$ on määrittelemätön. Automaatin M hyväksymä kieli on joukko $L(M) = \{w \in \Sigma^* \mid \delta(s_0, w) \in F\}$.

Esimerkki 4.2 Kuvassa 4.4 on äärellinen automaatti, joka tunnistaa kielen $\{a^i bab^j \mid i,j \geq 0\}$, kun lopputilojen joukko on $F = \{s_2\}$. □



Kuva 4.4: Äärellinen automaatti, joka tunnistaa kielen $\{a^i bab^j | i,j \geq 0\}$.

Äärellisten automaattien M_1 ja M_2 sanotaan olevan *ekvivalentit*, jos $L(M_1) = L(M_2)$. Tarkastellaan deterministisiä äärellisiä automaatteja $M_1 = (S_1, \Sigma, \delta_1, s_1, F_1)$ ja $M_2 = (S_2, \Sigma, \delta_2, s_2, F_2)$, joilla on yhteinen syötemerkkien joukko Σ. Automaatin M_1 tila p_1 ja

automaatin M_2 tila p_2 ovat ekvivalentit, jos kaikilla syötemerkeistä muodostuvilla sanoilla x on voimassa seuraava ehto: sanan x lukeminen alkaen tilasta p_1 johtaa joukkoon F_1 kuuluvaan lopputilaan, jos ja vain jos sanan x lukeminen alkaen tilasta s_2 johtaa joukkoon F_2 kuuluvaan lopputilaan.

Helposti huomataan, että jos p_1 ja p_2 ovat ekvivalentit, niin myös tilat $\delta_1(p_1, a)$ ja $\delta_2(p_2, a)$ ovat keskenään ekvivalentit, kun a on mikä tahansa syötemerkki. Samoin, jos p_1 ja p_2 ovat ekvivalentit tilat, niin joko ne molemmat ovat lopputiloja tai kumpikaan niistä ei ole lopputila. Deterministiset äärelliset automaatit M_1 ja M_2 ovat siis ekvivalentit, jos ja vain jos niiden alkutilat ovat ekvivalentit.

Soveltamalla äärellisten joukkojen käsittelyongelman ratkaisualgoritmia saadaan tehokas algoritmi determinististen äärellisten automaattien ekvivalenttisuuden tutkimiseksi (kuva 4.5). Algoritmin aikavaatimus riippuu käytetystä äärellisten joukkojen käsittelyalgoritmin toteutustavasta.

Syöte: Äärelliset automaatit $M_1 = (S_1, \Sigma, \delta_1, s0_1, F_1)$ ja $M_2 = (S_2, \Sigma, \delta_2, s0_2, F_2)$.
Tuloste: Tieto siitä, ovatko automaatit M_1 ja M_2 ekvivalentit.

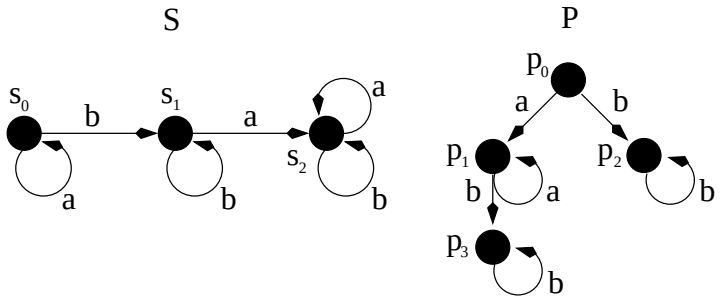
(1) LIST := $(s0_1, s0_2)$
(2) COLLECTION := { { s } | s on jomman kumman automaatin jokin tila }
(3) **while** LIST ei ole tyhjä **do**
(4) ota joukosta LIST pari (s_1, s_2)
(5) A_1 := find(s_1) ja A_2 := find(s_2)
(6) **if** $A_1 \neq A_2$ **then**
(7) union(A_1, A_2, A_1)
(8) lisää joukkoon LIST parit $(\delta_1(s_1, a), \delta_2(s_2, a))$ kaikilla $a \in \Sigma$
(samaa paria ei kuitenkaan lisätä toiseen kertaan listaan)
(9) **end**
(10) **od**
(11) Tutki, kuuluuko kokoelmaan COLLECTION sellaista joukkoa, jossa on sekä lopputiloja että ei-lopputiloja. Jos kuuluu, niin automaatit eivät ole ekvivalentteja, muulloin ovat.

Kuva 4.5: Automaattien ekvivalenttisuus.

Selvitetään kuvan 4.5 algoritmin aikavaatimus tilojen lukumäärän $n = |S_1| + |S_2|$ funktiona. Algoritmin while-silmukassa suoritetaan erillisten joukkojen käsittelyongelman union- ja find-operaatioita. Union-operaatioita tehdään korkeintaan $n - 1$ kertaa. Find-operaatioiden lukumäärä puolestaan riippuu $LIST$aan vietyjen parien lukumääristä. Jokaisen union-operaation yhteydessä pareja voidaan lisätä korkeintaan $|\Sigma|$ kappaletta. Find-operaatioita voidaan suorittaa siis korkeintaan $(n-1)|\Sigma|$ kappaletta.

Koska while-silmukan ulkopuolelle jäävät algoritmin osat tarvitsevat yhteensä aikaa $O(n)$, niin algoritmin aikavaatimuksen määrää while-silmukassa tehtävät erillisten joukkojen käsittelyoperaatiot. Algoritmin aikavaatimukseksi saadaan siis $O(n\alpha(n|\Sigma|, n))$. Kun ajatellaan, että merkkien määrä $|\Sigma|$ on vakio, niin aikavaatimukseksi saadaan $O(n\alpha(n, n))$.

Esimerkki 4.3 Esimerkkinä kuvan 4.5 algoritmin soveltamisesta tarkastellaan kuvan 4.6 automaatteja S ja P, joiden lopputilojen joukot olkoot vastaavasti $\{s_0, s_1\}$ ja $\{p_0, p_1, p_2, p_3\}$ ja siirtymärelaatiot δ_S ja δ_P.



Kuva 4.6: Esimerkkiautomaatit.

Aluksi $LIST$aan otetaan alkutilojen muodostama pari (s_0, p_0). Joukot $\{s_0\}$ ja $\{p_0\}$ yhdistetään joukoksi $\{s_0, p_0\}$ ja $LIST$aan lisätään uudet parit $(\delta_S(s_0, a), \delta_P(p_0, a)) = (s_0, p_1)$ ja $(\delta_S(s_0, b), \delta_P(p_0, b)) = (s_1, p_2)$.

Otetaan seuraavaksi käsittelyyn pari (s_0, p_1). Koska s_0 kuuluu joukkoon $\{s_0, p_0\}$ ja p_1 joukkoon $\{p_1\}$, saadaan uusi joukko $\{s_0, p_0, p_1\}$. Koska tilapari $(\delta_S(s_0, a), \delta_P(p_1, a)) = (s_0, p_1)$ on jo aikaisemmin lisätty $LIST$aan, ei lisäystä enää tehdä. Sen sijaan uusi pari $(\delta_S(s_0, b), \delta_P(p_1, b)) = (s_1, p_3)$ lisätään $LIST$aan.

Seuraavaksi käsitellään tilapari (s_1, p_2). Aluksi yhdistetään joukot $\{s_1\}$ ja $\{p_2\}$ joukoksi $\{s_1, p_2\}$. Huomataan, että $\delta_S(s_1, a) = s_2$, mutta $\delta_P(p_2, a)$ on määrittelemätön. Tämä tarkoittaa, että automaatti P päätyy tässä tapauksessa syöteen hylkäämiseen. Koska s_2 ei ole hyväksymislopputila, niin samaan päädytään myös automaatissa S. Jatkamalla merkillä b tilaparista (s_1, p_2) päädytään samoihin tiloihin.

$LIST$assa on käsittelemättä enää pari (s_1, p_3). Yhdistetään joukot $\{s_1, p_2\}$ ja $\{p_3\}$ joukoksi $\{s_1, p_2, p_3\}$. Samoin kuin edellisen parin kohdalla, syötemerkki a tuottaa tilanteen, jossa toimintaa ei ole määritelty automaatissa P, ja toisaalta saadaan sama pari (tässä siis (s_1, p_3) uudelleen).

Näin $LIST$ on tyhjentynyt pareista, ja siirrytään tutkimaan joukkojen kokoelmaa $COLLECTION$. Siihen kuuluvat joukot $\{s_0, p_0, p_1\}$, $\{s_1, p_2, p_3\}$ ja $\{s_2\}$. Huomataan, että mihinkään joukkoon ei kuulu sekä lopputiloja että ei-lopputiloja. Tulokseksi saadaan siis tieto, että automaatit S ja P ovat ekvivalentit.

□

Luku 5

Merkkijonon sovitusalgoritmeista

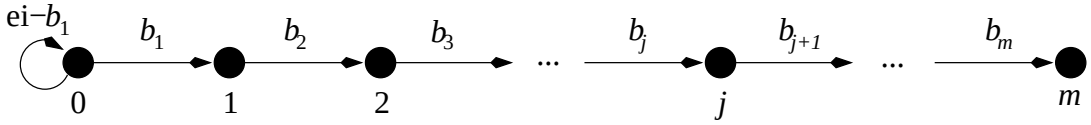
Tarkastellaan merkkijonoja $S = a_1 a_2 \ldots a_n$ (*kohde*) ja $P = b_1 b_2 \ldots b_m$ (*malli*). Tarkasteltavissa merkkijonoissa on yleensä $m \ll n$. Ongelmana on selvittää, voidaanko S kirjoittaa muodossa $S = S'PS''$ eli sisältääkö merkkijono S merkkijonon P. Yksinkertainen mutta hidas ratkaisu on verrata merkkijonoa P kaikkiin merkkijonoihin $a_1 \ldots a_m$, $a_2 \ldots a_{m+1}$, $\ldots$, $a_{n-m+1} \ldots a_n$. Näin tulee jokaisesta kohdasta $i = 1, \ldots, n-m+1$, alkaen tehdyksi korkeintaan m yksittäisten merkkien vertailua. Vertailujen kokonaismäärä on siis $O(mn)$. Pahin tapaus on se, jossa P on muotoa $P = aa \ldots ab$ ja S muotoa $S = aa \ldots aa$.

5.1. Knuthin-Morrisin-Prattin algoritmi

Otetaan tehtäväksi kehittää eräänlaiseen äärelliseen automaattiin perustuva sovitusmenetelmä, jonka aikavaatimus on $O(n+m)$.

Merkkijono Q on merkkijonon R *etuosa*, jos $R = QR'$ jollakin (mahdollisesti tyhjällä) merkkijonolla R', ja *loppuosa*, jos vastaavasti $R = R'Q$. Jatkossa oletetaan, että kohde S ja malli P muodostuvat äärellisen aakkoston I merkeistä.

Muodostetaan deterministinen merkkijononsovitusautomaatti MP, joka tunnistaa kielen $L(MP) = \{x \in I^*P \mid x$ sisältää merkkijonon P ainoastaan loppuosanaan $\}$. Automaatti MP muodostuu perussiirtymistä ja korjaussiirtymistä. Jos $P = b_1 b_2 \ldots b_m$, niin perussiirtymiksi otetaan kuvassa 5.1 esitetyt siirtymät.



Kuva 5.1: Merkkijononsovitusautomaatin perussiirtymät.

Oletetaan, että luettuaan kohteen S etuosan $a_1 a_2 \ldots a_k$ automaatti MP on tilassa j. Tällöin viimeksi luetut j merkkiä ovat $b_1 b_2 \ldots b_j$ eikä mikään merkkijonon $a_1 a_2 \ldots a_k$ pitempi loppuosa ole mallin etuosa. Syötemerkkijonon seuraava merkki on a_{k+1}. Jos nyt $a_{k+1} = b_{j+1}$, niin automaatti siirtyy tilaan $j+1$. Korjaussiirtymää tarvitaan, jos $a_{k+1} \neq$

b_{j+1}. Tällöin automaatti siirtyy sellaiseen tilaan $s (s < j)$, että s on suurin mahdollinen indeksi, jolla $b_1 b_2 \ldots b_s$ on merkkijonon $a_1 a_2 \ldots a_k$ loppuosa.

Korjaussiirtymät voidaan määritellä korjausfunktion f avulla. Jokaisella tilalla j asetetaan

$$f(j) = \begin{cases} \text{suurin indeksi } s < j, \text{ jolle } b_1 b_2 \ldots b_s = b_{j-s+1} b_{j-s+2} \ldots b_j, \\ 0, \text{jos tällaista indeksiä ei ole.} \end{cases}$$

Jos merkit eivät täsmää ja korjausfunktion arvo on 0, niin mallia siirretään oikealle niin paljon, että mallin ensimmäistä merkkiä verrataan viimeksi tutkittuun kohteen merkkiin. Jos korjausfunktion arvo on nollaa suurempi, niin mallia siirretään oikealle niin, että korjausfunktion arvo ilmoittaa sen merkin mallissa, joka tulee viimeksi tutkitun kohteen merkin kohdalle.

Tarkastellaan seuraavaksi, miten korjausfunktion arvot voidaan tehokkaasti laskea. Määritelmän perusteella $f(1) = 0$. Oletetaan, että arvot $f(1), f(2), \ldots, f(j)$ on määrätty ja että $f(j) = i$. Nyt on määrättävä $f(j+1)$. Tätä varten verrataan merkkejä b_{j+1} ja b_{i+1}. Jos merkit ovat samat, niin asetetaan $f(j+1) = f(j) + 1$, sillä tällöin on $b_1 \ldots b_i b_{i+1} = b_{j-i+1} \ldots b_j b_{j+1}$. Jos taas $b_{j+1} \neq b_{i+1}$, niin on etsittävä pienin sellainen korjausfunktion f soveltamiskertojen lukumäärä r, että $f^r(j) = n$ ja $b_{j+1} = b_{n+1}$. Tällöin asetetaan $f(j+1) = n + 1$. Jos kuitenkin päädytään tilaan 0 ($f^r(j) = 0$) ja $b_{j+1} \neq b_1$, niin asetetaan $f(j+1) = 0$. Saadaan kuvan 5.2 algoritmi.

```
(1)  f(1) := 0
(2)  for j := 2 to m do
(3)       i := f(j − 1)
(4)       while b_j ≠ b_{i+1} and i > 0 do i := f(i) od
(5)       if b_j ≠ b_{i+1} and i = 0 then
(6)            f(j) := 0
(7)       else
(8)            f(j) := i + 1
(9)       end
(10) od
```

Kuva 5.2: Knuthin-Morrisin-Prattin korjausfunktio.

Lause 5.1 *Knuthin-Prattin-Morrisin algoritmin korjausfunktio f voidaan muodostaa ajassa, joka on verrannollinen mallin P pituuteen.*

Todistus. Tarkastellaan kuvan 5.2 algoritmia. Jokaisella **for**-silmukan suorituskerralla rivit (3) ja (5) suoritetaan yhden kerran. Rivillä (4) olevan **while**-lauseen suorituskertojen yhteinen aikavaatimus riippuu muuttujan i arvojen muutoksista. Muuttujan i arvo kasvaa peräkkäisillä sijoituksilla

$$f(j) := i + 1; j := j + 1; i := f(j - 1);$$

Aluksi $i = 0$ ja sijoitus $f(j) := i + 1$ rivin (5) **if**-lauseen **else**-osassa voidaan suorittaa vain $m - 1$ kertaa. Muuttujan i arvo kasvaa aina ykkösen kerrallaan. Tästä seuraa, että muuttujan i arvoa voidaan rivillä (4) pienentää korkeintaan $m - 1$ kertaa ja rivin (4) aikavaatimus on $O(m)$. Tämä on myös koko algoritmin aikavaatimus. □

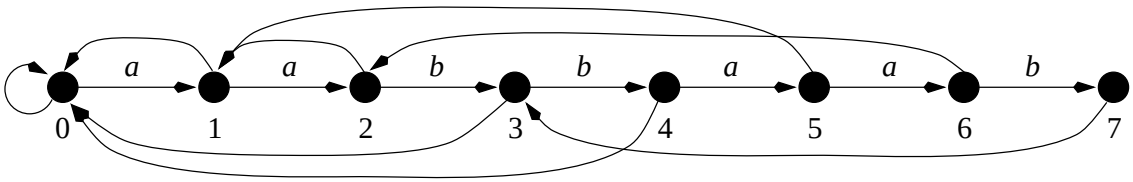
Lause 5.2 *On olemassa algoritmi, joka päättää ajassa $O(|S|+|P|)$, esiintyykö merkkijono P osana merkkijonossa S.*

Todistus. Toimimalla samaan tapaan kuin Lauseen 5.1 todistuksessa voidaan todistaa, että MP tekee korkeintaan $2\,|S|$ siirtymää käsitellessään syötteen S. Toisaalta automaatin MP muodostaminen vaatii ajan $O(|P|)$. Yhteensä aikaa kuluu siis $O(|S| + |P|)$. □

Esimerkki 5.1 Muodostetaan Knuthin-Morrisin-Prattin menetelmän automaatti mallille $P = aabbaab$. Määrätään aluksi korjausfunktio f:

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

Koska mallin pituus on 7, niin automaatissa on 8 tilaa. Automaatti on esitetty kuvassa 5.3. □



Kuva 5.3: Esimerkkiin liittyvä merkkijononsovitusalgoritmi.

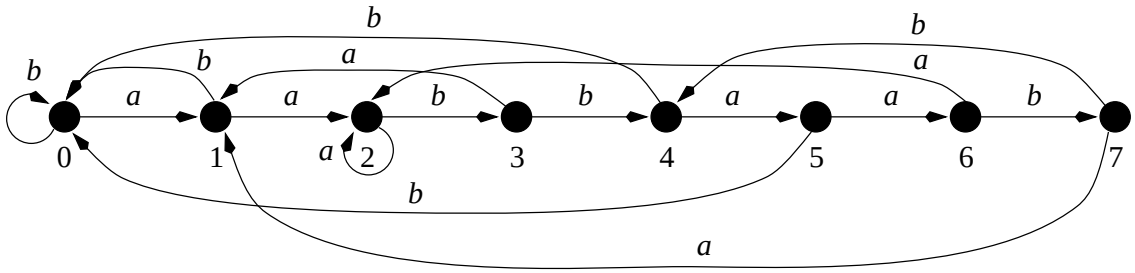
Lauseessa 5.2 ei ole viittausta syöteaakkoston I kokoon. Tämä on mahdollista, koska automaatti MP voi tehdä useita siirtymiä yhtä syötemerkkiä kohti. Otetaan seuraavaksi tehtäväksi muuttaa automaattia MP niin, että se lukee yhden syötemerkin jokaisen siirtymän yhteydessä. Uuden automaatin muodostamiseen kuluu enemmän aikaa kuin MP:n muodostamiseen ja tuloksena olevassa automaatissa on enemmän siirtymiä kuin automaatissa MP. Lisätyö kuitenkin kannattaa, jos syöteaakkostossa I on vähän merkkejä.

Olkoon S automaatin tilajoukko ja olkoon b_{m+1} sellainen merkki, joka ei kuulu aakkostoon I. Tarkastellaan uuden siirtymäfunktion $\delta : S \times I \to S$ muodostamista mallille $P = b_1 b_2 \ldots b_m$. Korjausfunktio f määritellään kuten ennenkin. Sen avulla saadaan uudet siirtymät kuvan 5.4 algoritmilla.

Esimerkki 5.2 Tarkastellaan uudelleen automaatin muodostamista mallille $P = aabbaab$. Nyt halutaan määritellä sellainen automaatti, joka ei tee tyhjiä siirtymiä. Kun käytetään edellisen esimerkin korjausfunktiota, saadaan tulokseksi kuvan 5.5 automaatti. □

(1) **for** $j := 1$ **to** m **do** $\delta(j-1, b_j) := j$ **od**
(2) **for** $b \in I$, $b \neq b_1$ **do** $\delta(0, b) := 0$ **od**
(3) **for** $j := 1$ **to** m **do**
(4) **for** $b \in I$, $b \neq b_{j+1}$ **do** $\delta(j, b) := \delta(f(j), b)$ **od**
(5) **od**

Kuva 5.4: Merkinsovitus.



Kuva 5.5: Esimerkkiin liittyvä merkkijononsovitusalgoritmi.

5.2. Boyerin-Mooren algoritmi

Edellisessä alakohdassa tarkasteltu algoritmi tekee vähintään n (= kohteen pituus) vertailua. Tässä alakohdassa tarkasteltava algoritmi, jonka aikavaatimus pahimmassa tapauksessa on sama ($O(n)$ vertailua) kuin Knuthin-Morrisin-Prattin algoritmilla, mutta joka usein selviää vähemmällä kuin n:llä vertailulla. Parhaassa tapauksessa tämän ns. Boyerin-Mooren algoritmin aikavaatimus on $O(m + m/n)$.

Boyerin-Mooren algoritmissa merkkejä verrataan alkaen mallin lopusta. Jos löydetään merkkipari, joka ei täsmää, siirretään mallia korjausfunktion ilmoittama määrä oikealle ja ryhdytään uudelleen tutkimaan merkkejä mallin lopusta alkaen.

Esimerkki 5.3 Olkoot $P =$ AT_THAT ja

$$S = \text{WHICH\_FINALLY\_HALTS.\_AT\_THAT\_POINT.}$$

Aluksi malli asetetaan kohteen päälle niin, että niiden ensimmäiset merkit ovat kohdakkain. Vertailu aloitetaan mallin lopusta. Ensimmäisenä verrataan siis merkkejä F ja T. Vertailu ei täsmää, ja lisäksi tiedetään, että merkkiä F ei ole koko mallissa. Niinpä mallia voidaan siirtää oikealle merkin F yli.

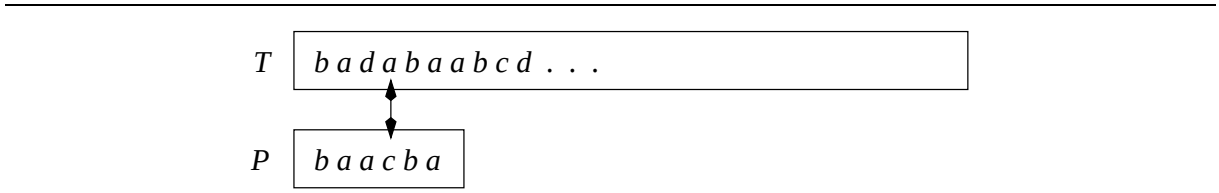
Mallin viimeinen kirjain on nyt välilyönnin (_) kohdalla. Vertailu ei nytkään täsmää. Siirretään mallia oikealle niin, että mallissa oleva välilyönti tulee sen kohteessa olevan välilyönnin kohdalle, joka äsken oli vertailtavana. Mallin viimeinen merkki on nyt kohteessa olevan HALTS-sanan T:n kohdalla; aloitetaan taas vertailu. Ensimmäinen vertailu täsmää, mutta toinen ei. Kohteessa on nyt kirjain L, jota ei ole lainkaan mallissa. Voidaan taas siirtää mallia oikealle tällaisen merkin yli. Mallin loppupää tulee nyt välilyönnin kohdalle, joten ensimmäinen vertailu ei täsmää. Kun nyt siirretään mallia niin, että siinä oleva välilyönti täsmää tähän kohteesta löydettyyn välilyöntiin, niin löydetään koko mallin

esiintymä kohteesta. □

Edellisessä esimerkissä sovellettiin toistuvasti Boyerin-Mooren algoritmin kahdesta heuristiikasta toista, ns. bad-character -heuristiikkaa. Sen mukaan ei-täsmäävän merkin löytyessä voidaan mallia siirtää niin paljon oikealle, että seuraava mallissa oleva tähän kohtaan sopiva merkki tulee kohteen viimeksi tutkitun merkin kohdalle. Jos tällaista merkkiä ei mallissa ole, voidaan malli siirtää kokonaan viimeksi tutkitun kohteen merkin oikealle puolelle. Paras tapaus algoritmin toiminnan kannalta on se, että kussakin mallin sijoituskohdassa jo ensimmäinen merkinsovitus epäonnistuu ja kohteessa on sellainen merkki, jota mallissa ei lainkaan ole. Tällöin mallia voidaan toistuvasti siirtää pituutensa verran oikealle.

Käytännössä bad character -heuristiikkaa sovelletaan yleensä niin, että tallennetaan kustakin merkistä sen oikean puoleisimman esiintymän paikka mallissa. Tällainen paikka saattaa olla siinä osassa mallia, joka on jo tutkittu, ja tällöin heuristiikka siirtäisi mallia vasemmalle! Tällaisia siirtoja ei kuitenkaan koskaan tehdä. Parempaan tulokseen päästäisiin tietenkin niin, että jokaisesta mallin paikasta erikseen ilmoitettaisiin jokaisen merkin seuraava esiintymä vasemmalla. Tämä veisi kuitenkin liian paljon tilaa.

Esimerkki 5.4 Tarkastellaan kuvan 5.6 tilannetta.



Kuva 5.6: Boyerin-Mooren algoritmin soveltamiseen liittyvä tilanne.

Mallin lopusta lukien kaksi ensimmäistä merkkiä täsmäävät, mutta kolmas merkki (c) ei täsmää. Koska kohteessa on tällä kohdalla 'a', bad character -heuristiikka siirtäisi mallia yhden askeleen oikealle. Tällöin kohteen 'a' ja mallin 'a' olisivat kohdakkain. Tämä menettely jättää huomiotta informaation, joka liittyy jo tutkittuihin merkkeihin. Tiedossahan on, että ei-täsmäävän kohdan oikealla puolella kohteessa on 'ba'. Bad character -heuristiikkaa parempaan tulokseen (tässä nimenomaisessa tilanteessa) päästään, kun mallissa oleva osa 'ba' sijoitetaan kohteessa olevan samanlaisen osan kohdalle. Tällöin mallia voidaan siirtää yhden askeleen sijasta neljä askelta oikealle. □

Edellisessä esimerkissä sovellettua menettelyä kutsutaan Boyerin-Mooren algoritmin yhteydessä good suffix -heuristiikaksi. Itse asiassa se on aivan sama kuin Knuthin-Morrisin-Prattin algoritmin korjausfunktio. Jokaisen ei-täsmäävän vertailun yhteydessä tutkitaan, kumpi heuristiikoista siirtäisi mallia pitemmälle oikealle ja siirto tehdään tämän heuristiikan mukaan. Kun malli on annettu, voidaan heuristiikkoihin liittyvät siirtomäärät laskea valmiiksi taulukkoihin.

5.3. Rabinin-Karpin algoritmi

Rabinin-Karpin algoritmissa ajatellaan merkkijonot luvuiksi siinä lukujärjestelmässä, jonka kantaluvun käytetyn aakkoston koko määrää. Yksinkertaisuuden vuoksi oletetaan aluksi, että käytössä on 10-merkkinen aakkosto. Kun vielä käytetään aakkostossa merkkejä $0, 1, \ldots, 9$, niin tarkasteltavat merkkijonot voidaan ajatella tavallisina 10-järjestelmän lukuina. (Rabinin- Karpin menetelmä on tietenkin käyttökelpoinen millaisilla aakkostoilla tahansa; tässä on vain asian yksinkertaistamiseksi valittu tuttu aakkosto.)

Käytetään tässä kohdassa mallista merkintää $P[1..m]$ ja kohteesta merkintää $T[1..n]$. Merkintää p käytetään mallista desimaaliluvuksi tulkittuna ja merkintää t_s siitä desimaaliluvusta, joka vastaa kohteen osaa $T[s+1..s+m]$, $s = 0, 1, \ldots, n-m$. Ensimmäinen ongelma on laskea arvot p ja t_0 tehokkaasti sekä pystyä laskemaan tehokkaasti t_{s+1}, kun tunnetaan t_s. Luku p voidaan laskea ajassa $O(m)$ ns. Hornerin säännöllä:

$$p = P[m] + 10\big(P[m-1] + 10\big(P[m-2] + \ldots + 10(P[2] + 10P[1])\ldots\big)\big).$$

Samoin voidaan tietenkin laskea t_0 kohteen alkuosasta $T[1..m]$. Arvo t_{s+1} voidaan laskea t_s:stä kaavalla $t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$. Luvun $10^{m-1}T[s+1]$ vähentäminen pudottaa t_s:n merkitsevimmän numeron pois ja luvun $T[s+m+1]$ lisääminen antaa uuden vähiten merkitsevän numeron.

Lukujen p ja t_s käytössä on kuitenkin se hankaluus, että ne tulevat kaikissa käytännön tilanteissa liian suuriksi. Sen vuoksi ne korvataan sopivilla jakojäännöksillä. Lasketaan jakojäännökset sellaisen alkuluvun q suhteen, että luku $10q$ vielä mahtuu yhteen muistisanaan (tai on muuten mahdollista käsitellä tehokkaasti).

Oletetaan esimerkiksi, että malli on 31415 ja kohde on 2359023141526739921 ja että käytetään jakojäännöksiä luvun 13 suhteen. Koska mallin jakojäännös mod 13 on 7, etsitään mallista sellaisia viiden merkin pituisia jaksoja, joiden jakojäännös mod 13 on myös 7.

Jakojäännökset on voitava laskea tehokkaasti. Jos esimerkiksi tiedetään ensimmäisen viiden merkin mittaisen jakson (23590) jakojäännös 8 (merkitään $23590 \equiv 8 \pmod{13}$), niin seuraavan jakson jakojäännös saadaan nopeasti jakojäännösaritmetiikan avulla:

$$\begin{aligned} 35902 &\equiv (23590 - 2 \cdot 10000) \cdot 10 + 2 \pmod{13} \\ &\equiv (8 - 2 \cdot 3) \cdot 10 + 2 \pmod{13} \\ &\equiv 2 \cdot 10 + 2 \pmod{13} \equiv 22 \pmod{13} \equiv 9 \pmod{13}. \end{aligned}$$

Huomaa, että tulosta $10000 \equiv 3 \pmod{13}$ käytetään kaikkien "viisikkojen" jakojäännöstä laskettaessa. Koska myös edellinen jakojäännös on käytettävissä, niin uuden jakojäännöksen laskeminen on todella yksinkertaista; yllä olevassa esimerkissä lasketaan $(8 - 2 \cdot 3) \cdot 10 + 2 \pmod{13}$. Samaan tapaan jatkamalla saadaan jakojäännökset kaikille viiden merkin mittaisille kohteen osille. Nämä jakojäännökset ovat $8-9-3-11-0-11-7-8-4-5-10-11-7-9-11$.

Huomataan, että oikea jakojäännös löytyi kaksi kertaa, mutta mallilla on vain yksi esiintymä kohteessa. Oikea jakojäännös ei siis vielä takaa mallin löytymistä. Jos jakojään-

nökset ovat samat, on mallia sovitettava merkki kerrallaan sen varmistamiseksi, että mallin esiintymä tosiaan on löytynyt. Jakojäännösten käyttö on siis eräänlainen heuristiikka, joka karsii mahdollisia siirtymiä. Jos q voidaan valita tarpeeksi suureksi, niin on oletettavaa, että tarpeettomia siirtymiä (jakojäännökset samat, mutta ei mallin esiintymää) tehdään hyvin harvoin.

Rabinin-Karpin algoritmi esitetään kuvassa 5.7. Siinä oletetaan, että käytettävän aakkoston koko on d.

```
     Procedure RK(T, P, d, q)
 (1) n := kohteen pituus
 (2) m := mallin pituus
 (3) h := d^(m-1) mod q
 (4) p := 0
 (5) t_0 := 0
 (6) for i := 1 to m do
 (7)         p := (dp + P[i]) mod q
 (8)         t_0 := (dt_0 + T[i]) mod q
 (9) od
(10) for s := 0 to n - m do
(11)         if p = t_s then
(12)                 if P[1..m] = T[s + 1..s + m] then
(13)                         Mallin esiintymä löytyi
(14)                 end
(15)                 if s < n - m then
(16)                         t_{s+1} := (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
(17)                 end
(18)         end
(19) od
```

Kuva 5.7: Rabinin-Karpin algoritmi.

Luku 6

Graafialgoritmeista

Tässä luvussa tarkastellaan kaikkein keskeisimpiä graafien käsittelyyn liittyviä algoritmeja. Graafeihin liittyvät peruskäsitteet oletetaan tunnetuiksi.

6.1. Syvyyssuuntainen etsintä

Graafin solmujen läpikäynti on osatehtävänä monessa algoritmissa. Yleisin menettely solmujen läpikäyntiin on *syvyyssuuntainen etsintä* (df-etsintä). Tässä kohdassa tarkastellaan df-etsintää suuntaamattomassa graafissa. Suuntaamattoman graafin df-etsintä jakaa graafin särmät *puusärmiksi* ja *takautuviksi särmiksi*. Puusärmiä pitkin df-etsintä etenee solmusta toiseen ja takautuvat särmät yhdistävät etsinnässä myöhemmin vastaantulevia solmuja jo käsiteltyihin solmuihin. Df-etsintä voidaan tehdä algoritmilla 6.1. Aliohjelma search määritellään kuvassa 6.2.

Syöte: Suuntaamattoman graafin $G = (V, E)$ kaikki vieruslistat $L(v), v \in V$
Tuloste: Joukon E ositus puusärmien joukoksi T ja takautuvien särmien joukoksi B

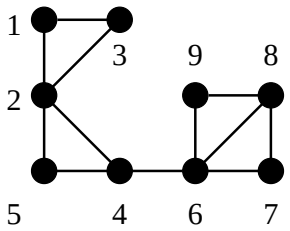
(1) Alusta joukko T tyhjäksi
(2) **while** joukossa V on merkitsemätön solmu v **do** search(v) **od**
(3) Ota joukkoon B ne särmät, jotka eivät ole joukossa T

Kuva 6.1: Df-etsintä.

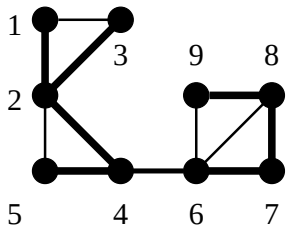
Procedure search(v)
(1) Merkitse(v)
(2) **for all** ($w \in L(v)$) **and** (w ei merkitty) **do**
(3) Ota särmä (v, w) joukkoon T
(4) search(w)
(5) **od**

Kuva 6.2: Aliohjelma search.

Kuvan 6.1 algoritmilla saatava aligraafi (V, T) on graafin G virittävä metsä. Jos graafi



(a)



(b)

1	2, 3
2	1, 3, 4, 5
3	1, 2
4	2, 5, 6
5	2, 4
6	4, 7, 8, 9
7	6, 8
8	6, 7, 9
9	6, 8

(c)

Kuva 6.3: (a) Esimerkkigraafi, (b) sen puusärmät ja (c) sen vieruslistat.

G on yhtenäinen, niin aligraafi on graafin G virittävä puu. Joukkoon T tulevat särmät riippuvat lähtösolmusta ja solmujen järjestyksestä vieruslistoissa.

Lause 6.1 *Suuntaamattoman graafin $G = (V, E)$ syvyyssuuntaisen etsinnän aikavaatimus on $O(\max(|V|, |E|))$.*

Todistus. Search-aliohjelman ulkopuolella algoritmi tutkii kaikki solmut; tämän kustannus on $O(|V|)$. Lisäksi search-aliohjelman sisäpuolella algoritmi tutkii kaikki särmät; kustannus on nyt $O(|E|)$. Koko algoritmin aikavaatimuksen suuruusluokan määrää suurempi arvoista $O(|V|)$ ja $O(|E|)$. □

Jos (v, w) on df-etsintään liittyvä takautuva särmä, niin syntyvässä syvyyssuuntaisessa virittävässä metsässä solmu v on solmun w esivanhempi tai solmu w on solmun v esivanhempi. Jos solmuihin liitetään numerot df-etsinnän määräämässä järjestyksessä, niin saatu numerointijärjestys on virittävän metsän esijärjestys.

6.2. 2-yhtenäisyys

Olkoon $G = (V, E)$ yhtenäinen suuntaamaton graafi. Solmu u on *irrotussolmu*, jos on olemassa sellaiset solmut v ja w, että jokainen näiden solmujen välinen polku kulkee solmun u kautta. Graafi G on 2-*yhtenäinen*, jos sillä ei ole irrotussolmua.

Määritellään graafin G särmien joukossa E ekvivalenssirelaatio seuraavasti: pari (e_1, e_2) kuuluu relaatioon, jos ja vain jos graafissa on silmukka, johon e_1 ja e_2 kuuluvat, tai $e_1 = e_2$. Tämän ekvivalenssirelaation ekvivalenssiluokista muodostettuja aligraafia kutsutaan graafin 2-*yhtenäisiksi komponenteiksi*.

Esimerkki 6.1 Kuvan 6.3 (a) graafin irrotussolmut ovat 2, 4 ja 6. Graafin 2-yhtenäiset solmujoukot ovat { 1, 2, 3 }, { 2, 4, 5 }, { 6, 7, 8, 9 } ja { 4, 6 }. □

Kuvassa 6.3 (b) on edellisen esimerkin graafin puusärmät piirretty vahvalla viivalla ja takautuvat särmät ohuella viivalla.

Seuraavassa luettelossa on tärkeimpiä 2-yhtenäisten komponenttien ominaisuuksia:

- Graafin 2-yhtenäisessä komponentissa ei ole irrotussolmuja.
- Graafin 2-yhtenäiset komponentit ovat samat kuin sen maksimaaliset 2-yhtenäiset aliverkot.
- Kahdella eri 2-yhtenäisellä komponentilla on korkeintaan yksi yhteinen solmu.
- Solmu on irrotussolmu, jos ja vain jos se on yhteinen kahdelle 2-yhtenäiselle komponentille.
- Jos verkolla on k kappaletta 2-yhtenäisiä komponentteja, niin sillä on korkeintaan $k-1$ irrotussolmua.

Kaikilla kuvan 6.3 graafin irrotussolmuilla on df-etsinnän määräämässä virittävässä puussa lapsi, jonka jälkeläisellä ei ole takautuvaa särmää irrotussolmun aitoon esivanhempaan. Tämä ominaisuus on voimassa yleisestikin: Olkoon $G = (V, E)$ yhtenäinen suuntaamaton graafi ja olkoon $H = (V, T)$ sen syvyyssuuntainen virittävä puu. Solmu s on irrotussolmu, jos ja vain jos s on puun H juuri ja sillä on enemmän kuin yksi lapsi tai solmulla s on lapsi t, jonka mistään jälkeläisestä (t itse mukaan lukien) ei johda takautuvaa särmää solmun s aitoon esivanhempaan.

Yllä olevissa esimerkeissä on graafin solmut nimetty niiden järjestysnumerolla df-etsinnässä; näin oletetaan tehtävän jatkossakin. Määritellään syvyyssuuntaiseen etsintään liittyvät solmujen *low-arvot* seuraavasti:

> low[v] = min($\{v\} \cup \{w \mid$ graafissa on sellainen takautuva särmä (x, w), että x on solmun v jälkeläinen ja w on v:n esivanhempi syvyyssuuntaisessa virittävässä puussa $\}$).

Määritelmä voidaan kirjoittaa myös muodossa low[v] = min($\{v\} \cup \{$low[s] $\mid$ s on v:n lapsi virittävässä puussa $\}$ $\cup$ $\{w \mid (v, w) \in B\}$).

Syöte: Yhtenäinen, suuntaamaton graafi G = (V, E).
Tuloste: Luettelo graafin G kunkin 2-yhtenäisen komponentin särmistä.

(1) Aseta aluksi $T := \emptyset$ ja count := 1, alustetaan pino Stack tyhjäksi
(2) Merkitse graafin jokainen solmu "uudeksi"
(3) Valitse jokin solmu $v_0 \in V$
(4) Suorita aliohjelma searchb(v_0).

Kuva 6.4: 2-yhtenäiset komponentit.

Määritelmistä seuraa, että solmun low-arvo voidaan määrätä, kun df-etsinnässä kyseisen solmun vieruslista on käyty läpi. Näiden arvojen perusteella löydetään irrotussolmut ja edelleen 2-yhtenäiset komponentit. Saadaan kuvan 6.4 algoritmi, jossa käytetään seuraavia taulukkoja:

dfnumber: solmun järjestysnumero syvyyssuuntaisessa etsinnässä
parent: solmun vanhempi
low: solmun low-arvo

ja pinoa Stack. Aliohjelma search on kuvassa 6.5 ja se muodostaa syvyyssuuntaisen virittävän puun $S = (V, T)$ ja laskee arvot low(v) kaikille solmuille.

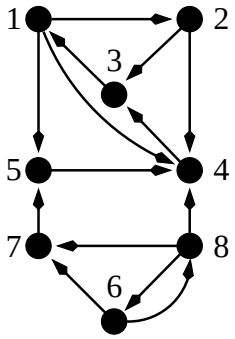
```
     Procedure searchb(v)
(1)  Merkitse solmu v "vanhaksi"
(2)  dfnumber[v] := count
(3)  count := count + 1
(4)  low[v] := dfnumber[v]
(5)  for all w ∈ L[v] do
(6)       Vie särmä (v,w) pinoon Stack, jos se ei vielä ole ollut siellä
(7)       if w on "uusi"then
(8)            Lisää (v,w) joukkoon T
(9)            parent[w] := v
(10)           searchb(w)
(11)           if LOW[w] ≥ dfnumber[v] then          % 2-yht. komp. on löytynyt
(12)                Poista pinon Stack pinnalta kaikki särmät särmään
                     (v,w) asti (se mukaanlukien) ja tulosta ne
(13)           end
(14)           low[v] := min(low[v], low[w])
(15)      else {solmu ei ole "uusi"}
(16)           if w ≠ parent[v] then
(17)                low[v] := min(low[v], dfnumber[w])
(18)           end
(19)      end
(20) od
```

Kuva 6.5: searchb.

6.3. Suunnatun graafin df-etsintä ja vahvasti yhtenäisyys

Suunnatussa graafissa df-etsintä voi edetä vain kaarien suuntien mukaisesti. Suunnatun graafin df-etsintä jakaa graafin kaaret neljään osajoukkoon. Tarkastellaan kuvan 6.6 suunnattua graafia.

Syvyyssuuntainen etsintä etenee *puukaaria* pitkin. Kuvan 6.6 graafissa näitä ovat kaaret (1,2), (2,3), (2,4), (1,5), (6,7) ja (6,8). *Etenevät kaaret* eivät ole puukaaria, mutta ne johtavat solmusta sen jälkeläiseen syvyyssuuntaisen etsinnän määräämässä puussa. Esimerkkigraafin ainoa etenevä kaari on (1,4). *Takautuvat kaaret* johtavat jälkeläisistä



1	2, 4, 5
2	3, 4
3	1
4	3
5	4
6	7, 8
7	5
8	4, 6, 7

Kuva 6.6: Suunnattu graafi ja sen vieruslistat.

Kuva 6.7: Esimerkkigraafin vahvasti yhtenäiset komponentit.

niiden edeltäjiin. Tällaisia ovat kaaret (3,1) ja (8,6). *Poikittaisten kaarten* välillä ei ole syvyyssuuntaisen etsinnän määräämässä puussa edeltäjäsuhdetta kumpaankaan suuntaan. Poikittaisia kaaria ovat kaaret (4,3), (5,4), (7,5), (8,4) ja (8,7).

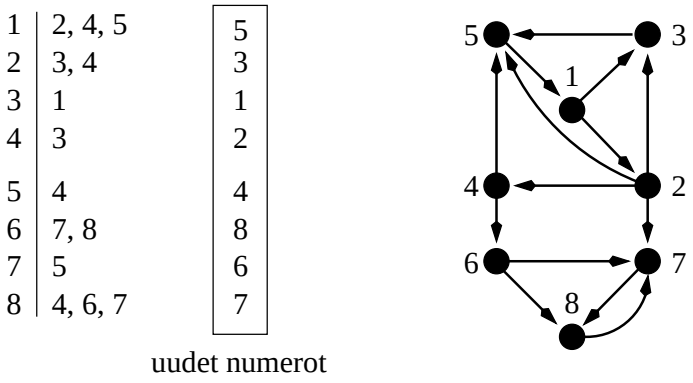
Olkoon $G = (V, E)$ suunnattu graafi. Määritellään solmujen joukossa V ekvivalenssirelaatio seuraavasti: solmupari (v, w) kuuluu relaatioon, jos ja vain jos graafissa G on polku solmusta v solmuun w ja solmusta w solmuun v. Ekvivalenssiluokkien määräämiä aliverkkoja sanotaan graafin *vahvasti yhtenäisiksi komponenteiksi*. Jos graafissa on vain yksi vahvasti yhtenäinen komponentti, niin graafi on *vahvasti yhtenäinen*. Kuvan 6.6 graafin vahvasti yhtenäiset komponentit on esitetty kuvassa 6.7.

Vahvasti yhtenäisten komponenttien etsintä onnistuu kuvan 6.8 algoritmilla, jota jatkossa kutsutaan nimellä VY.

(1) Numeroi solmut df-etsinnän päättymisjärjestyksessä.
(2) Muodosta graafi G_r vaihtamalla kaarien suunnat.
(3) Tee df-etsintä graafissa G_r alkaen solmusta, joka sai kohdassa (1) suurimman numeron. Mikäli kaikkia solmuja ei saavuteta, jatka etsintää aina numeroltaan suurimmasta vielä saavuttamattomasta solmusta.
(4) Askeleen (3) määräämät aliverkot ovat graafin G vahvasti yhtenäiset komponentit.

Kuva 6.8: Vahvasti yhtenäiset komponentit (VY).

Kuvassa 6.9 on esitetty tässä kohdassa tarkasteltua esimerkkigraafia vastaava graafi G_r.



Kuva 6.9: VY-algoritmin soveltaminen esimerkkigraafiin.

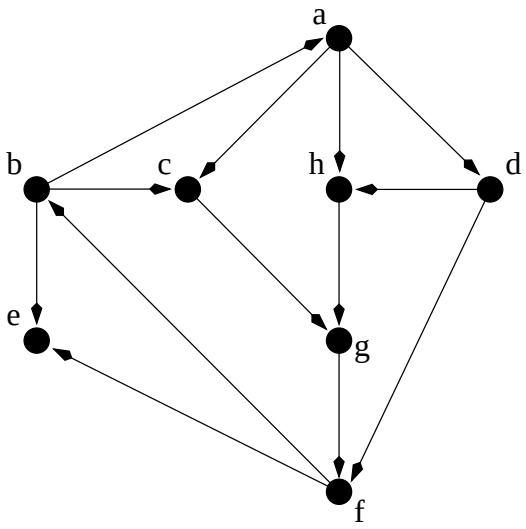
Lause 6.2 *Algoritmi VY toimii oikein ja sen aikavaatimus on* $O(\max(|V|, |E|))$.

Todistus. Oikeellisuuden osoittamiseksi riittää näyttää, että solmut v ja w kuuluvat samaan graafin G_r df-etsinnän määräämään puuhun, jos ja vain jos ne ovat samassa graafin G vahvasti yhtenäisessä komponentissa. Oletetaan aluksi, että graafissa G on polut solmusta v solmuun w ja solmusta w solmuun v. Koska yhteys on molempiin suuntiin, niin kaarien suuntien vaihtaminen ei vaikuta solmujen saavutettavuuteen graafissa G_r. Riippumatta siitä, kumpi solmuista saavutetaan graafin G_r df-etsinnässä ensiksi, saavutetaan ensiksi saavutetusta myös toinen solmu. Ne kuuluvat siis samaan df-etsinnän virittämään puuhun graafissa G_r.

Oletetaan nyt, että solmut v ja w kuuluvat samaan df-etsinnän virittämään puuhun graafissa G_r. Olkoon x sen G_r:n virittävän puun juuri, johon v ja w kuuluvat. Koska solmu v on solmun x jälkeläinen graafin G_r läpikäynnissä, on graafissa G_r polku solmusta x solmuun v. Graafissa G on siis polku solmusta v solmuun x. Koska solmu v kuuluu puuhun, jonka juuri on x, niin solmun x numero on suurempi kuin solmun v numero. Graafin G df-etsinnässä solmun v kutsu päättyi siis ennemmin kuin solmun x kutsu. Koska toisaalta graafissa G on polku solmusta v solmuun x, ei solmun v kutsu voi alkaa ennen solmun x kutsua. Solmua v kutsutaan siis solmun x kutsun aikana eli v on solmun x jälkeläinen graafin G virittävässä metsässä. Graafissa G on siis polku solmusta x solmuun v. Näin ollaan todistettu, että graafissa G on polut solmusta v solmuun x ja solmusta x solmuun v. Sama voidaan todistaa solmuille x ja w. On siis oltava myös polut solmusta v solmuun w ja solmusta w solmuun v. Algoritmin aikavaatimus on sama kuin df-etsinnän aikavaatimus. □

6.4. Palautesolmujen etsintä

Tässä kohdassa tarkastellaan erästä syvyyssuuntaisen etsinnän sovellusta suunnatussa graafissa. Vahvasti yhtenäisen suunnatun graafin solmua sanotaan *palautesolmuksi* (feedback vertex), jos graafin kaikki suunnatut silmukat kulkevat sen kautta.



Kuva 6.10: Esimerkkigraafi.

Esitetään kaksivaiheinen algoritmi, joka löytää syötteenä saadun suunnatun vahvasti yhtenäisen graafin $G = (V, E)$ kaikki palautesolmut ajassa $O(|V| + |E|)$.

Ensimmäisessä vaiheessa graafissa tehdään syvyyssuuntainen etsintä ja graafin solmuihin liitetään ns. post-numerot syvyyssuuntaisen etsinnän päättymisjärjestyksessä. (Sen solmun post-numeroksi tulee 1, jossa etsintä päättyy ensimmäiseksi, jne.) Samalla voidaan määrätä solmujen ns. high-arvot: solmun high-arvo on suurin post-arvo niiden solmujen joukossa, jotka saavutetaan kyseisestä solmusta polulla, jossa aluksi on muita kuin takautuvia kaaria ja lopuksi korkeintaan yksi takautuva kaari.

Jos oletetaan, että kuvan 6.10 graafin solmut on lueteltu vieruslistoissa aakkosjärjestyksessä ja syvyyssuuntainen etsintä aloitetaan solmusta a, niin solmuille saadaan seuraavat post-arvot: post$(a) = 8$, post$(b) = 1$, post$(c) = 5$, post$(d) = 7$, post$(e) = 2$, post$(f) = 3$, post$(g) = 4$ ja post$(h) = 6$.

Huomaa, että takautuville kaarille (v, w) pätee $post(v) < post(w)$ ja että esimerkkigraafin kaikkien solmujen high-arvoksi tulee 8, sillä kaikista solmuista on määritelmän mukainen polku solmuun a, jonka post-arvo on 8.

Palautesolmu voidaan nyt määritellä post- ja high-arvojen avulla: Solmu x on palautesolmu täsmälleen silloin, kun jokaisen takautuvan kaaren (v, w) kohdalla pätee $post(v) \leq post(x) \leq post(w)$, ja jokaisen sellaisen ei-takautuvan kaaren (v, w), jolla $high(w) \geq post(v)$, kohdalla pätee $post(x) \leq post(w)$ tai $post(x) \geq post(v)$. (Todistus palautesolmujen karakterisoinnille post- ja high-arvojen avulla löytyy R.E. Tarjanin artikkelista "Two streamlined depth-first search algorithms", *Fundamenta Informaticae* IX (1986) 85–94.)

Intuitiivisesti palautesolmun määrittely post- ja high-arvojen avulla voidaan selittää seuraavasti. Jos (v, w) on takautuva kaari, joka siis sulkee graafin silmukan, niin palautesolmun on oltava esi-isän w ja jälkeläisen v "välissä". Ei-takautuvien kaarten ehto liittyy tilanteeseen, jossa ei-takautuva kaari (v, w) on mukana silmukassa, sillä muuten solmun w high-arvo ei voi olla suurempi tai yhtä suuri kuin solmun v post-arvo. Ehto

$post(x) \leq post(w)$ liittyy tilanteeseen, jossa x on polulla solmun w high-arvon määräävästä solmusta u solmuun v. Ehto $post(x) \geq post(v)$ puolestaan liittyy tilanteeseen, jossa x on polulla solmusta w solmuun u.

Palautesolmut etsivä algoritmi voidaan nyt antaa kuvan 6.11 muodossa.

---

(1) Alustetaan pino tyhjäksi ja muuttuja lowest arvoon, joka on yhtä suurempi kuin graafin solmujen lukumäärä.
(2) Käsitellään solmut post-numeroiden mukaisessa järjestyksessä.
(3) Jokaisen solmun v kohdalla käydään vieruslista läpi.
(4) Jos kaari (v, w) on takautuva, asetetaan muuttujan lowest arvoksi pienempi arvoista lowest ja post(w) ja tyhjennetään pino.
(5) Jos kaari (v, w) ei ole takautuva ja $high(w) \geq post(v)$, niin poistetaan pinosta kaikki solmut, joiden post-numero on suurempi kuin post(w).
(6) Solmun v vieruslistan käsittelyn jälkeen se viedään pinoon, jos sen post-arvo on korkeintaan lowest.
(7) Kaikkien solmujen käsittelyn jälkeen pinossa on palautesolmut.

Kuva 6.11: Palautesolmujen etsintä.

Kuvan 6.10 tapauksessa pinoon jäävät solmut f ja g, jotka ovat siis kyseisen graafin palautesolmut.

6.5. Pariutusongelma

Tarkastellaan suuntaamatonta graafia $G = (V, E)$. Särmäjoukko $M \subseteq E$ on *pariutus*, jos jokaiseen solmuun liittyy korkeintaan yksi joukon M särmä. Pariutus on täydellinen, jos joukon V koko on täsmälleen puolet joukon M koosta. *Pariutusongelmassa* on tehtävänä määrätä mahdollisimman suuri pariutusjoukko M.

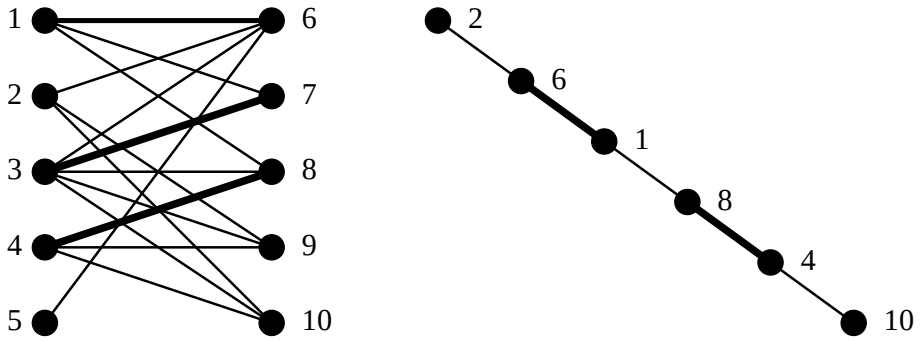
Solmu v on *vapaa* pariutuksen M suhteen, jos mikään joukon M särmä ei liity solmuun v. Polku $(v_1, v_2, \ldots, v_{2k})$ on *täydennyspolku* pariutuksen M suhteen, jos v_1 ja v_{2k} ovat vapaita ja joka toinen särmä polulla kuuluu pariutukseen M. (Täydennyspolku on aina yksinkertainen eli mikään solmu ei esiinny siinä yhtä kertaa useammin)

Esimerkki 6.2 Kuvassa 6.12 on vasemmalla esimerkkigraafi, jossa pariutukseen kuuluvat särmät on piirretty vahvemmalla viivalla kuin muut särmät. Oikealla on eräs tätä pariutusta vastaava täydennyspolku. □

Olkoot M ja N joukkoja. Otetaan käyttöön merkintä $M \oplus N = (M \setminus N) \cup (N \setminus M)$. Merkin $\oplus$ voi lukea esimerkiksi "symmetrinen erotus" tai "poissulkeva tai".

Lause 6.3 *Jos M on pariutus ja särmäjoukon P päätepisteet muodostavat täydennyspolun pariutuksen M suhteen, niin $M \oplus P$ on pariutus ja $|M \oplus P| = |M| + 1$.*

Todistus. Merkitään $P = P_1 \cup P_2$, kun joukkoon P_1 kuuluvat täydennyspolun parittomat särmät ja joukkoon P_2 täydennyspolun parilliset särmät. Joukko P_2 on siis joukon M



Kuva 6.12: Esimerkkigraafi ja sen eräs täydennyspolku.

osajoukko ja joukko P_1 on joukon M komplementtijoukon osajoukko. Voidaan kirjoittaa $M \oplus P = (M \setminus P_2) \cup P_1$. Joukko $(M \setminus P_2)$ on pariutus, jonka suhteen kaikki joukon P solmut ovat vapaita. Tästä seuraa, että $(M \setminus P_2) \cup P_1$ on pariutus, sillä joukossa P_1 olevat särmät yhdistävät joukon $(M \setminus P_2)$ suhteen vapaita solmuja. Koska $|P_1| = |P_2| + 1$, niin $|M \oplus P| = |(M \setminus P_2) \cup P_1| = |M| + 1$. □

Lause 6.4 *(Berge 1957). Pariutus M on suurin mahdollinen, jos ja vain jos graafissa ei ole täydennyspolkua sen suhteen.*

Todistus. Oletetaan, että pariutus M on suurin mahdollinen ja väitetään, että graafissa ei ole täydennyspolkua sen suhteen. Tehdään vastaoletus: P on täydennyspolku pariutuksen M suhteen. Lauseen 6.2 menetelmällä voidaan muodostaa pariutus, jossa on enemmän särmiä kuin pariutuksessa M. Se ei siis ole suurin mahdollinen. Vastaoletuksen on oltava väärä.

Oletetaan seuraavaksi, että graafissa ei ole täydennyspolkua pariutuksen M suhteen ja väitetään, että M on suurin mahdollinen pariutus. Tehdään vastaoletus: tarkasteltavassa graafissa G on sellainen pariutus N, että $|N| > |M|$. Tarkastellaan joukkoon $M \oplus N$ kuuluvia särmiä. Nämä särmät määräävät graafin G aligraafin $G' = (V, M \oplus N)$, joka ei välttämättä ole yhtenäinen. Koska pariutuksessa ei mihinkään solmuun voi liittyä kuin yksi särmä, niin aliverkolla G' on se ominaisuus, että sen kaikkien solmujen asteluku on korkeintaan 2. Jos asteluku on 2, niin toinen särmistä kuuluu pariutukseen M ja toinen pariutukseen N. Kaikki graafin G' yhtenäiset komponentit ovat polkuja tai sellaisia silmukoita, joiden pituus on parillinen. Kaikissa silmukoissa on yhtä monta väliä pariutuksista M ja N. Koska $|N| > |M|$, niin jossakin polussa on oltava enemmän särmiä joukosta N kuin joukosta M. Tämä polku on täydennyspolku pariutuksen M suhteen. Näin on päädytty ristiriitaan vastaoletuksen kanssa. □

Lauseen 6.4 tulosta voidaan soveltaa pariutusongelman ratkaisemiseen. Saadaan kuvan 6.13 algoritmi.

Ongelmana on askeleen (2) toteuttaminen. Se voidaan kuitenkin toteuttaa tehokkaasti, jos graafi on puolittuva. Seuraavassa tarkastellaankin puolittuvia graafeja. Muodostetaan annetulle puolittuvalle graafille G *täydennyspolkugraafi* G_M pariutuksen M suhteen kuvan 6.14 algoritmilla.

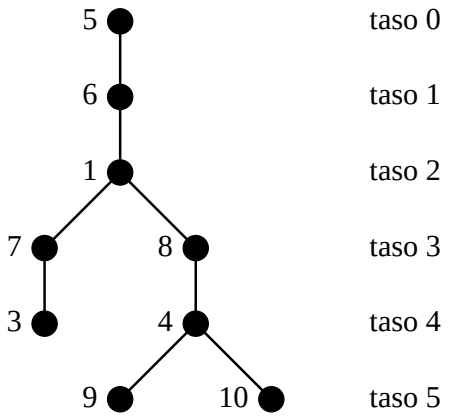
(1) Alusta M tyhjäksi.
(2) Etsi täydennyspolku M:n suhteen.
(3) Korvaa M joukolla $M \oplus P$.
(4) Toista askelia (2) ja (3) niin kauan kuin täydennyspolkuja löytyy.

Kuva 6.13: Pariutus.

(1) Tasoksi 0 valitaan jokin pariutuksen M suhteen vapaa solmu.
(2) Toiminta tasolla i, kun i on pariton: lisää graafiin G_M solmut, joihin johtaa pariutukseen kuulumaton särmä jostain tason $i-1$ solmusta.
(3) Toiminta tasolla i, kun i on parillinen: lisää graafiin G_M solmut, joihin johtaa pariutukseen M kuuluva särmä jostain tason $i-1$ solmusta.
(4) Jos parittomalle tasolle lisätään vapaa solmu, on täydennyspolku löytynyt.
(5) Jos uusia solmuja ei voida enää lisätä graafiin G_M ja vapaita solmuja on vielä jäljellä, aloitetaan graafin G_M uusi komponentti jostain vapaasta solmusta.

Kuva 6.14: Täydennyspolkugraafi.

Esimerkki 6.3 Tarkastellaan kuvan 6.12 graafia ja sen pariutusta $M = \{(1,6),(3,7),(4,8)\}$. Pariutuksen M suhteen vapaita solmuja on neljä: 2, 5, 9 ja 10. Kuvassa 6.15 on graafin eräs täydennyspolkugraafi. Koska saatavassa täydennyspolkugraafissa vapaat solmut 9 ja 10 ovat parittomalla tasolla, ollaan löydetty kaksi täydennyspolkua. □



Kuva 6.15: Täydennyspolkugraafi.

Lause 6.5 *Puolittuvalle graafille saadaan maksimaalinen pariutus ajassa* $O(|V||E|)$.

Todistus. Täydennyspolkugraafi G_M voidaan muodostaa ajassa $O(E)$. Koska jokainen löydetty täydennyspolku vähentää vapaiden solmujen lukumäärää kahdella, niin maksimaalisen pariutuksen löytämiseksi täydennyspolkugraafi on muodostettava korkeintaan $|V|/2$ kertaa. □

Esimerkki 6.4 Osastolla on U erilaista tekemätöntä työtä ja V työntekijää. Muodostetaan puolittuva graafi, jossa on särmä (v, u), jos työntekijä v osaa työn u. Etsimällä maksimaalinen pariutus saadaan selville, montako työtä voidaan aloittaa. □

6.6. Vakaat pariutukset

Vakaat avioliitot -ongelmassa (stable marriage problem) tarkastellaan joukkoa, johon kuuluu n miestä ja yhtä monta naista. Jokainen mies on järjestänyt kaikki naiset mieleiseensä paremmuusjärjestykseen, ja naiset ovat tehneet samoin miehille. Jos nainen q edeltää naista r miehen p listassa, niin sanotaan, että p *pitää enemmän* naisesta q kuin naisesta r. (Samaa sanontaa käytetään myös naisten listoissa olevista miehistä.)

Tehtävänä on muodostaa sellainen pariutus M, että jokaiseen pariin tulee yksi mies ja yksi nainen. Jos mies m ja nainen w muodostavat parin pariutuksessa M, niin heitä sanotaan toistensa *partnereiksi* (pariutuksen M suhteen). Merkintä $m = p_M(w)$ tarkoittaa, että mies m on naisen w partneri pariutuksessa M, ja vastaavasti merkintä $w = p_M(m)$ tarkoittaa, että nainen w on miehen m partneri.

mies 1	2	4	1	3		nainen 1	2	1	4	3
mies 2	3	1	4	2		nainen 2	4	3	1	2
mies 3	2	3	1	4		nainen 3	1	4	3	2
mies 4	4	1	3	2		nainen 4	2	1	4	3

Taulukko 6.1: Miesten ja naisten listat.

Sanotaan, että miehellä m ja naisella w on *suhde* pariutuksessa M, jos m ja w eivät ole toistensa partnereita pariutuksessa M, mutta m pitää w:stä enemmän kuin partneristaan $p_M(m)$ ja w pitää m:stä enemmän kuin partneristaan $p_M(w)$. Jos pariutuksessa on ainakin yksi suhde, niin se on *epävakaa*. Muutoin pariutus on *vakaa*.

Esimerkki 6.5 Tarkastellaan taulukon 6.1 listoja. Pariutukset $\{(1,4),(2,3),(3,2),(4,1)\}$ ja $\{(1,4),(2,1),(3,2),(4,3)\}$ ovat vakaita, mutta kaikki muut pariutukset ovat epävakaita (järjestetyissä pareissa ensimmäisenä komponenttina on mies ja toisena nainen). Esimerkiksi pariutuksessa $\{(1,1),(2,3),(3,2),(4,4)\}$ on suhde (1,4), sillä mies 1 pitää enemmän naisesta 4 kuin partneristaan 1 ja nainen 4 pitää enemmän miehestä 1 kuin partneristaan 4.

□

Vakaat avioliitot -ongelman tapauksella voi siis olla useampia kuin yksi ratkaisu. Toisaalta voidaan todistaa, että ratkaisuja on aina vähintään yksi. Annetun pariutuksen vakaus voidaan selvittää ajassa $O(n^2)$ tutkimalla jokaisen miehen kohdalla kyseisen miehen listassa partneria edeltävät naiset. Kun miehiä on n kappaletta ja partneria listassa edeltäviä naisia korkeintaan $n-1$ kappaletta, saadaan aikavaatimus $O(n^2)$.

Kuvassa 6.16 esitetty Galen-Shapleyn algoritmi löytää aina vakaan pariutuksen. Algoritmista on kaksi versiota, "miesversio" ja "naisversio". Miesversiossa miehet tekevät kosintoja ja naisversiossa naiset. Kosintoja tekevä osapuoli saa omalta kannaltaan parhaan mahdollisen tuloksen, joka on toisen osapuolen kannalta huonoin mahdollinen. Jos ongelman tapauksella on vain yksi ratkaisu, päätyvät algoritmin molemmat versiot luonnollisesti siihen.

Tarkastellaan seuraavassa Galen-Shapleyn algoritmin miesversiota, jossa siis miehet kosivat. Kosijat ovat koko ajan joko vapaita tai varattuja. Kosintojen vastaanottajat (tässä tapauksessa naiset) ovat vapaita ensimmäiseen kosintaan saakka; sen jälkeen he ovat varattuja, mutta mahdollisesti eri aikoina eri miehelle. Huomaa, että algoritmissa ei mitään ole mainintaan siitä, missä järjestyksessä miehet kosivat. Voidaan todistaa (vaikka tässä todistus sivuutetaan), että kosintojen järjestyksellä ei ole merkitystä lopputuloksen kannalta.

(1) Aluksi kaikki miehet ja naiset ovat vapaita.
(2) Vapaat miehet kosivat naisia (vapaita ja varattuja) listansa mukaisessa järjestyksessä.
(3) Vapaa nainen suostuu aina kosintaan; molemmista tulee tämän jälkeen varattuja.
(4) Varattu nainen vertailee kosijaa ja nykyistä partneriaan ja ottaa sen, josta pitää enemmän.
(5) Jos varattu nainen pitää kosijasta enemmän, tulee tästä varattu ja entisestä partnerista vapaa.
(6) Kosintoja jatketaan, kunnes kaikki miehet ovat varattuja.

Kuva 6.16: Galen-Shapleyn algoritmin miesversio.

Algoritmin pysähtyminen kaikilla syötteillä on helppo osoittaa. Algoritmin pysähtymisehtona on se, että jäljellä ei ole vapaita miehiä. Oletetaan, että olisi mies, jonka listan viimeinenkin nainen hylkäisi kyseisen miehen. Tämä on mahdollista vain silloin, kun listan kaikilla naisilla on jo partneri. Kun naisia ja miehiä on yhtä monta, ei tämä ole mahdollista. Algoritmin pysähtymisehto tulee siis aina voimaan.

Samoin on helppo todistaa, että tuloksena on aina vakaa pariutus. Algoritmi voi pysähtyä vain silloin, kun tuloksena on pariutus. Oletetaan nyt, että mies m pitää enemmän naisesta w kuin saadun pariutuksen M mukaisesta partneristaan $p_M(m)$. Algoritmin toimintaperiaatteesta seuraa, että nainen w on algoritmin suorituksen aikana kieltäytynyt miehen m kosinnasta. Nainen w siis pitää enemmän partneristaan $p_M(w)$ kuin miehestä m. Algoritmin tulokseksi saadussa pariutuksessa ei siis voi olla suhteita, joten se on vakaa.

Esimerkki 6.6 Sovelletaan Galen-Shapleyn algoritmin miesversiota taulukon 6.2 listoihin.

Tässä esimerkissä kosintajärjestys on sellainen, että numeroltaan pienin vapaa mies kosii. Aluksi siis mies 1 kosii suosikkiaan, ja muodostuu pari (1,4). Seuraavaksi mies 2 kosii naista 2, ja tuloksena on pari (2,2). Mieluisin nainen miehelle 3 on nainen 2, joka on jo varattu. Nyt nainen 2 vertailee uutta kosijaa (mies 3) ja nykyistä partneriaan (mies

mies 1	4	1	2	3		nainen 1	4	1	3	2
mies 2	2	3	1	4		nainen 2	1	3	2	4
mies 3	2	4	3	1		nainen 3	1	2	3	4
mies 4	3	1	4	2		nainen 4	4	1	3	2

Taulukko 6.2: Vakaat avioliitot -ongelman esimerkkitapaus.

2). Koska nainen 2 pitää enemmän miehestä 3, purkautuu pari (2,2) ja syntyy uusi pari (3,2). Samalla mies 2 tulee uudelleen vapaaksi ja kosii listansa toista naista (nainen 3). Tuloksena on pari (2,3).

Viimeisenä kosii mies 4, jonka listan kärjessä on nainen 3, joka on jo varattu. Nainen 3 vertailee miehiä 2 ja 4, joista mies 2 on mieluisampi. Täten mies 4 pysyy vapaana. Mies 4 kosii listassaan toisena olevaa naista, ja muodostuu pari (4,1).

Nyt kaikki miehet ovat varattuja ja algoritmin toiminta päättyy. Tuloksena on vakaa pariutus $\{(1,4),(2,3),(3,2),(4,1)\}$.

□

Annetulla vakaat avioliitot -ongelman tapauksella voi olla lukuisia ratkaisuja eli vakaita pariutuksia. Niistä kaksi on aina erikoisasemassa: miesoptimaalinen ratkaisu, joka saadaan Galen-Shapleyn algoritmin miesversiolla (miehet kosivat), ja naisoptimaalinen ratkaisu, joka saadaan algoritmin naisversiolla (naiset kosivat). Miesoptimaalisessa ratkaisussa jokaisella miehellä on paras mahdollinen partneri kaikkien vakaiden pariutusten suhteen ja jokaisella naisella on huonoin mahdollinen partneri kaikkien vakaiden pariutusten suhteen. Naisoptimaalinen versio puolestaan antaa naisille mahdollisimman mieluisat partnerit. Vakaat avioliitot -ongelman tapaukseen liittyvät ratkaisut muodostavat hilan, jonka suurin alkio on miesoptimaalinen ratkaisu ja pienin alkio on naisoptimaalinen ratkaisu. Ratkaisusta toiseen päästään tietynlaisella "rotaatiolla", jonka tarkempi muoto tässä kuitenkin sivuutetaan.

Vakaat avioliitot -ongelmaa voidaan yleistää monella tavalla. Voidaan esimerkiksi sallia, että henkilöt voivat jättää epämiellyttävinä pitämiään henkilöitä kokonaan pois listoistaan. Toinen yksinkertainen tapa ongelman yleistämiseksi on sallia, että henkilöt saavat ilmoittaa pitävänsä joistain henkilöistä täsmälleen yhtä paljon. Voidaan myös tutkia, miten joidenkin henkilöiden liittoutuminen tai valehteleminen (todellisten prefenssien salaaminen) muuttaa tilannetta.

Vakaat kumppanit (stable roommates) -ongelmassa tarkastellaan kooltaan parillista joukkoa ihmisiä, jotka kaikki järjestävät kaikki muut henkilöt suosituimmuusjärjestykseen. Pariutus jakaa taas joukon alkiot alkiovieraisiin pareihin. Nyt parit kuitenkin ovat järjestämättömiä, ja niitä merkitään muodossa $\{x,y\}$. Samoin kuin edellä sanotaan, että pariutus on epävakaa, jos on olemassa sellaiset henkilöt, jotka pitävät toisistaan enemmän kuin pariutuksen heille määräämästä partnerista. Jos tällaisia henkilöitä ei ole, niin pariutus on vakaa.

Vakaat kumppanit -ongelman tapauksella ei välttämättä ole ratkaisua. Tämä huoma-

taan tarkastelemalla taulukon 6.3 tilannetta.

1	3	2	4
2	1	3	4
3	2	1	4
4	1	2	3

Taulukko 6.3: Vakaat kumppanit -ongelman tapaus, jolla ei ole ratkaisua.

Taulukon 6.3 tilanteessa henkilölle 4 ei löydy kumppania. Mahdolliset parit, joissa henkilö 4 on mukana, ovat $\{1, 4\}$, $\{2, 4\}$ ja $\{3, 4\}$. Suhteet $\{1, 2\}$, $\{2, 3\}$ ja $\{3, 1\}$ osoittavat vastaavat pariutukset epävakaiksi. Taulukon 6.3 tilanteessa kannattaa vielä huomata, että tapauksella ei ole ratkaisua, olipa henkilön 4 listan järjestys mikä tahansa.

Sen lisäksi, että vakaat kumppanit -ongelman yhteydessä voidaan tutkia samanlaisia ongelmia kuin vakaat avioliitot -ongelman yhteydessä, voidaan lisäksi tutkia ratkaisujen olemassaolokysymyksiä ja esimerkiksi sitä, miten löytyy mahdollisimman suuri vakaa osapariutus.

Luku 7

Tiedon tiivistämisestä

Tiedon tiivistämisen tärkeimmät tavoitteet ovat tallennustilan säästäminen ja tiedonsiirtokustannusten pienentäminen. Tiivistettävä tieto *koodataan* lyhempään muotoon koodausmenetelmää (tiivistämismenetelmää) käyttäen. Tiivistetty informaatio voidaan vastaavasti *tulkita* eli *dekoodata* alkuperäiseen muotoonsa. Mikäli tulkinta palauttaa informaation täsmälleen alkuperäiseen muotoonsa, niin sanotaan, että käytetty koodausmenetelmä on *säilyttävä* (lossless). Jos informaatiota katoaa, sanotaan koodausmenetelmän olevan *hukkaava* (lossy). Hukkaavia koodausmenetelmiä käytetään esimerkiksi kuva- ja ääni-informaatiota tiivistettäessä. Koska tällaisessa informaatiossa voi olla paljon sellaista, jota ihmisaistit eivät kuitenkaan pysty erottamaan, voidaan koodausmenetelmien yhteydessä tällaista informaatiota karsia. Tässä luvussa keskitytään pelkästään tekstimuotoisen tiedon tiivistämiseen säilyttävillä menetelmillä.

Tiivistettävää informaatiota kutsutaan tässä *tekstiksi*. Teksti jakautuu *sanoiksi*. Sanalla tarkoitetaan tässä sellaista tekstin peräkkäisistä biteistä muodostuvaa kokonaisuutta, jota käytetty koodausmenetelmä pitää tekstin pienimpänä yksikkönä. ("Sanalla" ei siis tässä välttämättä ole mitään tekemistä luonnollisen kielen sanojen kanssa.) Koodausmenetelmä voi olla sellainen, että sanat määräytyvät vasta koodauksen aikana ja ne voivat jopa vaihdella tekstin sisällön mukaan. *Kiinteäsanaisissa* (defined-word) koodausmenetelmissä mahdollisten sanojen joukko on kuitenkin tiedossa jo ennen koodauksen aloittamista.

Tiivistettyä informaatiota kutsutaan *koodiksi*; se jakautuu *koodisanoiksi*. Myös koodisanojen pituus voi joissain koodausmenetelmissä vaihdella. Koodisanojen on kuitenkin oltava yksiselitteisesti *tulkittavissa* (eli dekoodattavissa). Tämä vaatimus on täytetty, jos koodisanojen joukolla on *prefix-ominaisuus*. Tämä tarkoittaa sitä, että mikään koodisana ei saa olla toisen koodisanan etuosa.

7.1. Entropia

Oletetaan, että jokin *lähde* tuottaa *lähdeaakkoston* $\{s_1, s_2, \ldots, s_k\}$ merkkejä yksi kerrallaan. Oletetaan lisäksi, että merkkien tuottamistodennäköisyydet ovat toisistaan riippumattomat ja että merkin s_i todennäköisyys esiintyä lähteen tuottamassa merkkivirrassa seuraavana merkkinä on p_i. Mikäli oletus todennäköisyyksien riippumattomuudesta

on voimassa, sanotaan lähteen olevan *ensimmäistä astetta*. Mahdollisuus tiivistää ensimmäistä astetta olevan lähteen tuottamaa merkkivirtaa riippuu ainoastaan merkkien todennäköisyysjakautumasta, joka määrää merkkien *keskimääräisen informaatiosisällön*. Tarkastellaan esimerkkinä kolmea lähdetyyppiä: vakiolähdettä, satunnaislähdettä ja vinoa lähdettä. Vakiolähteen lähdeaakkoston jollakin merkillä s_i on ominaisuus $p_i = 1$. Tällöin tietenkin kaikilla muilla merkeillä s_j, $j \neq i$, on oltava $p_j = 0$. Vakiolähteen merkkivirta muodostuu siis yhden ainoan merkin esiintymistä. Satunnaislähteen merkkiaakkoston kaikilla merkeillä s_i on ominaisuus $p_i = 1/k$. Vinon lähteen merkkiaakkoston jollakin merkillä s_i on ominaisuus $p_i = 1/2$ ja kaikilla muilla merkeillä p_j, $j \neq i$, on $p_j = 1/\big(2(k-1)\big)$.

Tarkastellaan nyt esimerkkilähteiden tuottamien merkkivirtojen informaatiosisältöä ja merkkien koodaamiseen tarvittavaa bittien määrää. Vakiolähteen tapauksessa ei informaatiota ole lainkaan, eikä sen koodaamiseen siis tarvita yhtään bittiä. Satunnaislähteen merkkivirralla puolestaan on korkea informaatiosisältö. Tämä voidaan sanoa myös niin, että seuraavaksi tulevaa merkkiä ei voida ennustaa. Yhden merkin koodaamiseen tarvitaan $\lceil \log k \rceil$ bittiä. Vinon lähteen merkkien informaatiosisältö on kahden edellisen tapauksen väliltä. Vinon lähteen tuottamat merkit kannattaa koodata niin, että se merkki s_i, jolla $p_i = 1/2$, koodataan nollaksi. Muita merkkejä vastaavissa koodisanoissa on alussa ykkönen ja sen jälkeen $\lceil \log(k-1) \rceil$ muuta bittiä. Bittejä tarvitaan siis keskimäärin $1 + \lceil \log(k-1) \rceil/2$. Huomataan, että mitä "vakioisempi" lähde on, sitä vähemmän bittejä tarvitaan. Juuri tätä ominaisuutta kuvaa lähteen *entropia*. Informaatiolähteen entropian käsitteen on kehittänyt Shannon 1940-luvulla.

Edellä olleita merkintöjä käyttäen ensimmäistä astetta olevan lähteen entropiaksi H määritellään

$$H = \sum_{i=1}^{k} p_i \log_r(1/p_i).$$

Logaritmin kantaluku r on sama kuin koodauksessa käytetyn aakkoston koko eli yleensä $r = 2$. Vakiolähteen entropia on 0, satunnaislähteen entropia on $\log k$ ja vinon lähteen entropia on $\frac{1}{2}\log 2 + \sum_{i=1}^{k-1} \frac{1}{2(k-1)} \log(2(k-1)) = \frac{1}{2} + (k-1)\frac{1+\log(k-1)}{2(k-1)} = \frac{1}{2} + \frac{1}{2} + \frac{1}{2}\log(k-1) = 1 + \frac{1}{2}\log(k-1)$.

Lähteen entropia määrää "informaatioteoreettisen alarajan" koodauksessa tarvittavien bittien määrälle. Sitä parempaan tulokseen ei mikään koodausmenetelmä voi päästä. Käytännössä ensimmäistä astetta olevat lähteet ovat kuitenkin varsin harvinaisia. Esimerkiksi luonnollisessa kielessä eivät sanat ja merkit ole suinkaan toisistaan riippumattomia. Jos suomenkielisessä tekstissä on merkit 'HEVONE', niin seuraava merkki melko todennäköisesti on 'N'!

Koodauksen hyvyyttä voidaan mitata redundanssin avulla. Redundanssi lasketaan erotuksena

$$\sum_{i=1}^{k} p(a_i) l_i - \sum_{i=1}^{k} p(a_i) \log \frac{1}{p(a_i)},$$

kun $p(a_i)$ on sanan a_i esiintymistodennäköisyys ja l_i vastaavan koodisanan pituus. Redundanssi lasketaan siis keskimääräisen koodisanan pituuden ja entropian erotuksena.

Toinen tavanomainen tapa mitata koodauksen hyvyyttä on laskea tiivistyssuhde eli se, montako prosenttia koodisanojen esittämiseen tarvittava bittimäärä on tekstin esittämiseen tarvittavasta bittimäärästä.

7.2. Staattiset koodausmenetelmät

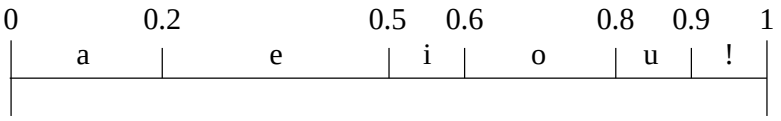
Staattiset koodausmenetelmät ovat kiinteäsanaisia ja sanan koodisana pysyy samana koko koodauksen ajan. Yleinen periaate on, että usein esiintyville sanoille kannattaa varata lyhyet koodisanat. Kun staattisissa koodausmenetelmissä koodisanat pysyvät samoina, olisi jo koodauksen alkaessa tiedettävä tekstissä esiintyvien sanojen frekvenssit. Jotta staattinen menetelmä voisi olla tehokas, on sen siis käytävä teksti kahteen kertaan läpi. Ensimmäisellä kerralla selvitetään sanojen frekvenssit ja toisella kerralla liitetään sanoihin sopivat koodisanat. *Dynaamiset koodausmenetelmät* puolestaan ovat sellaisia, että tiivistettävien sanojen ja koodisanojen välinen kuvaus voi muuttua koodausprosessin kestäessä. Dynaamiset menetelmät pystyvät mukautumaan tekstin sanojen esiintymistiheyksien muutoksiin, ja ne käyvät tekstin läpi vain yhden kerran.

Perinteinen staattinen koodausmenetelmä on *Huffmanin koodaus*. Huffmanin koodausta varten on aluksi selvitettävä sanojen esiintymisfrekvenssit tekstissä ja järjestettävä sanat esiintymisfrekvenssien määräämään järjestykseen. Koodisanojen määräämiseksi muodostetaan binaaripuu, jonka lehtinä koodattavat sanat ovat. Lehtisolmujen painoksi otetaan vastaava frekvenssi ja koko puun painoksi siinä olevien lehtien painojen summa. Alkutilanteessa jokainen sana muodostaa oman puunsa. Puita ruvetaan yhdistämään niin, että painoltaan kaksi pienintä puuta yhdistetään yhdeksi puuksi luomalla sille uusi juuri, jonka kahdeksi alipuuksi yhdistettävät puut tulevat. Uuden puun paino on yhdistettyjen puiden painojen summa. Yhdistämisiä jatketaan kunnes kaikki sanat ovat samassa puussa. Koodisanat määrätään nyt juuresta lehtiin kulkevien polkujen perusteella. Jokainen läpikäyty vasen haara tuo koodisanaan nollan ja oikea haara ykkösen. Huomaa, että puu muodostettiin ahneella algoritmilla.

Aritmeettisessa koodauksessa teksti esitetään sopivana reaalilukuvälinä. Nytkin on tunnettava koodattavien sanojen esiintymistodennäköisyydet. Koodauksen alkaessa on käytössä lukuväli $[0 \ldots 1)$. Käytössä oleva lukuväli kapenee koodattujen sanojen esiintymistodennäköisyyksien suhteessa. Kun väli kapenee, tarvitaan sen määrittelemiseen enemmän bittejä. Usein esiintyvät sanat kaventavat väliä vähemmän kuin harvoin esiintyvät sanat. Yleisten sanojen koodaamiseen tarvitaan siis vähemmän bittejä kuin harvinaisten sanojen koodaamiseen.

Tarkastellaan esimerkkinä tilannetta, jossa sanojen joukko on $\{a, e, i, o, u, !\}$ ja vastaavat esiintymistodennäköisyydet ovat $p(a) = 0.2$, $p(e) = 0.3$, $p(i) = 0.1$, $p(o) = 0.2$, $p(u) = 0.1$ ja $p(!) = 0.1$. Esiintymistodennäköisyydet määräävät sanoihin liittyvät lukuvälit kuvassa 7.1 esitetyllä tavalla.

Jos esimerkiksi tekstin ensimmäinen sana on e, niin alkuperäinen lukuväli $[0 \ldots 1)$ supistuu väliksi $[0.2 \ldots 0.5)$. Jos seuraava sana on a, niin tämä väli puolestaan supistuu väliksi $[0.2 \ldots 0.26)$. Kolmas sana i supistaa tämän välin väliksi $[0.23 \ldots 0.236)$ ja toinen



Kuva 7.1: Esimerkkiin liittyvät lukuvälit.

i-sana edelleen väliksi $[0.233\ldots0.2336)$. Jokaisen tekstin on päätyttävä erityiseen lopetusmerkkiin (tässä !). Tämä määrää lopulliseksi väliksi välin $[0.23354\ldots02336)$. Koodatun tekstin *eaii*! koodisanaksi kelpaa nyt mikä tahansa reaaliluku tältä väliltä. Dekoodatessa on tunnettava käytössä ollut sanajoukko ja sanojen esiintymistodennäköisyydet. Dekoodaus sujuu samaan tapaan kuin koodauskin.

Sekä Huffmanin koodauksesta että aritmeettisesta koodauksesta on olemassa myös dynaaminen versio. Tällöin koodauksen perusteena oleva rakenne (Huffmanin koodauksessa binaaripuu ja aritmeettisessa koodauksessa välin jakosuhde) muuttuu tekstin edetessä. Molempien menetelmien yksityiskohdat sivuutetaan.

7.3. Universaalit koodit ja menetelmät

Universaalikoodaus on staattinen koodausmenetelmä, jossa ei tarvitse tietää koodattavien sanojen tarkkoja esiintymistodennäköisyyksiä. Riittää, että nämä todennäköisyydet osataan järjestää laskevaan suuruusjärjestykseen. Sanat järjestetään vastaavien todennäköisyyksien määräämään järjestykseen. Kaikkein yleisimmälle sanalle annetaan koodisanaksi ykkönen, seuraavaksi yleisimmälle kakkonen ja niin edelleen. Ongelmaksi tulee nyt koodisanoina käytettyjen kokonaislukujen binaariesitysten erottaminen toisistaan. Tässä voidaan käyttää esimerkiksi Eliasin koodeja γ ja δ.

Universaalikoodauksen tuottamien koodisanojen keskipituudelle λ on voimassa

$$\lambda \leq c_1 H + c_2,$$

kun c_1 ja c_2 ovat vakioita ja H tarkoittaa tekstin entropiaa. Koodaus on asymptoottisesti optimaalinen, kun keskimääräinen sanapituus lähestyy entropiaa. Universaalikoodaus on siis asymptoottisesti optimaalinen, kun $c_1 = 1$.

Luvun x Eliasin γ-koodi muodostetaan niin, että koodisanan alkuun pannaan $\lfloor \log x \rfloor$ kappaletta nollia ja sen jälkeen koodattava sana x binaarisena. Jos esimerkiksi $x = 2$, niin koodisanaksi saadaan 010, ja kun $x = 16$, saadaan koodisana 000010000. Eliasin γ-koodin koodisanan pituus on $2\lfloor \log x \rfloor + 1$.

Eliasin δ-koodi muodostetaan siten, että koodisanan alkuun pannaan $\gamma\big(\lfloor \log x \rfloor + 1\big)$ ja sen jälkeen koodattava sana x binaarisena ilman ensimmäistä ykköstä. Jos $x = 2$, niin koodisanaksi saadaan 0100, ja kun $x = 16$, saadaan koodisana 001010000. Eliasin δ-koodin koodisanan pituus on $2\big\lfloor \log\big(\lfloor \log x \rfloor + 1\big)\big\rfloor + 1 + \lfloor \log x \rfloor$. Eliasin δ-koodi on asymptoottisesti optimaalinen.

7.4. Lempelin-Zivin koodaus

Esimerkkinä dynaamisesta koodauksesta tarkastellaan Lempelin-Zivin koodausta. Siinä tekstin sanajoukkoa ei ole kiinnitetty etukäteen, vaan sanat määräytyvät koodauksen aikana, jolloin tekstin peräkkäisistä merkeistä muodostetaan uusia sanoja. Muodostettavien sanojen pituudella on kuitenkin yläraja; samalla tulee määrättyä koodisanojen lukumäärälle yläraja. Oletetaan, että tiivistysmenetelmä on jo löytänyt tekstistä sanan x, jonka se on tallentanut symbolitauluunsa. Jos tekstistä myöhemmin löytyy merkkijono xa, niin se otetaan myös sanaksi (olettaen, ettei sananpituuden ylärajaa ole ylitetty). Koodisanat ovat tällöin muotoa (i, a), kun i on aikaisemman sanan indeksi symbolitaulussa ja a on se merkki, joka lisäämällä saatiin muodostettua uusi sana.

Tarkastellaan esimerkkinä tekstiä *abacababbacadda*. Kaksi ensimmäistä koodisanaa ovat $(0, a)$ ja $(0, b)$. Tämän jälkeen tekstistä löytyy aikaisempi sana a. Yhdessä seuraavan merkin c kanssa siitä muodostuu uusi sana ac, ja sitä vastaava koodisana on $(1, c)$. Samoin muodostetaan sana ab ja sitä vastaava koodisana $(1, b)$. Tässä vaiheessa symbolitaulussa on sanat a, b, ac ja ab tässä järjestyksessä. Seuraavaksi tunnetusta sanasta ab ja sitä tekstissä seuraavasta merkistä b muodostetaan uusi sana abb. Sitä vastaava koodisana on $(4, b)$. Lopuksi löydetään vielä uudet sanat aca, d ja da. Niitä vastaavat koodisanat ovat $(3, a)$, $(0, d)$ ja $(7, a)$. Symbolitaulussa on lopuksi sanat a, b, ac, ab, abb, aca, d ja da tässä järjestyksessä.

Lempelin-Zivin koodaus toimii melko huonosti lyhyiden tekstien tapauksessa, sillä koodauksen alkuvaiheessa joudutaan tuhlaamaan lyhyille sanoille pitkiä koodisanoja.

Lempelin-Zivin koodauksella on lukemattomia toteutusteknisiä variaatioita. UNIXin compress-käsky perustuu juuri Lempelin-Zivin menetelmään. Sen tiivistyssuhteeksi mainitaan yleensä $50-60\%$. Koodausmenetelmien tiivistyssuhteet vaihtelevat kuitenkin huomattavasti syötetekstien koosta ja muodosta riippuen.

7.5. Mukautuviin listoihin perustuva koodausmenetelmä

Tämän luvun lopuksi tarkastellaan luvussa 2 tutkittuihin listan käsittelyheuristiikkoihin perustuvaa koodausmenetelmää. Luvussa 2 huomattiin move-to-front -heuristiikka tietyssä mielessä parhaaksi tavaksi ylläpitää listaa; tässä kohdassa tarkastellaan tämän vuoksi juuri siihen perustuvaa koodausmenetelmää. Move-to-front -heuristiikka mukautuu nopeasti syötetekstin sanojen frekvenssimuutoksiin. Lokaalisti usein esiintyvät sanat ovat yleensä lähellä listan alkua. Sanojen koodisanoina kannattaa täten käyttää niiden etäisyyttä listan alusta.

Tekstin koodaus ja dekoodaus voidaan esittää sähkösanomaviestien lähettämisenä ja vastaanottamisena: lähettäjä (koodaaja) lähettää sähkösanomia vastaanottajalle (dekoodaaja). Molemmat osapuolet ylläpitävät samanlaista listaa move-to-front -heuristiikalla. Listoihin tallennetaan tekstistä löytyneet sanat. Mikäli tekstissä tulee vastaan sana, jota

vielä ei ole listoissa, lähetään ensin tieto siitä, että on tulossa uusi sana, ja sen jälkeen uusi sana sellaisenaan. Uuden sanan merkkinä voidaan käyttää listaindeksiä 0. Muutoin riittää lähettää pelkkä sanan listaindeksi. Sanan etsimiseen listasta liittyy aina etsityn sanan siirtäminen listan alkuun käytetyn heuristiikan mukaisesti. Samoin uudet sanat lisätään listan alkuun.

Listaheuristiikkoja käyttäviä koodausmenetelmiä sovelletaan lähinnä luonnollista kieltä sisältävien tiedostojen ja ohjelmatiedostojen tiivistämiseen. Tällöin tiivistettävä teksti voidaan jakaa sanoiksi luonnollisen kielen tai ohjelmointikielen tavanomaisten sääntöjen mukaisesti. Sanojen erottimina ovat siis välilyönti, piste, puolipiste jne.

Tarkastellaan esimerkkinä pelkistä isoista kirjaimista (ja välilyönneistä) muodostuvaa tekstiä. Välilyönnit jakavat tekstin sanoiksi. Olkoon tekstinä 'THE CAR ON THE LEFT HIT THE CAR I LEFT'. Kolme ensimmäistä sanaa ovat uusia, joten niitä vastaavat koodisanat ovat '0 THE', '0 CAR' ja '0 ON'. Tämän jälkeen listan sisältönä ovat sanat ON CAR THE tässä järjestyksessä. Tekstin seuraava sana on ensimmäinen, joka löytyy listasta. Sitä vastaava koodisana on 3, ja lista muuttuu muotoon THE ON CAR. Seuraavat sanat LEFT ja HIT ovat uusia, ja ne lisätään listan alkuun. Seuraavia sanoja THE ja CAR vastaavat koodisanat ovat 3 ja 5. Kokonaisuudessaan esimerkkitekstiä vastaava koodi on

0 THE 0 CAR 0 ON 3 0 LEFT 0 HIT 3 5 0 I 5 •.

Huomaa, että koodi päättyy loppumerkkiin •.

Koodaus- ja dekoodausalgoritmien tehokas toteuttaminen edellyttää operaatioiden

position(w): palauta sanan w listaindeksi ja
word(p): palauta sana, jonka listaindeksi on p

toteuttamista tehokkaammin kuin suoraviivaisesti läpikäymällä listoja niiden alusta.

Move-to-front -heuristiikkaan perustuvaa koodausmenetelmää voidaan parametroida esimerkiksi listan pituutta rajoittamalla. Jos listan sallitaan kasvaa rajattomasti, tulevat jotkut listaindekseinä esitetyt koodisanat pitkiksi. Tehokkaampaa voi tällöin olla sanojen koodaaminen uusien sanojen tapaan.

Luku 8

Satunnaistetuista algoritmeista

Satunnaistetuissa algoritmeissa on muotoa

$$x := \text{joku arvo joukosta } \{1, 2, \ldots, k\}$$

olevia käskyjä. Satunnaistetut algoritmit ovat siis epädeterministisiä (vrt. luku 2). Aidon satunnaisuuden aikaansaaminen tietokoneella ei ole ongelmatonta, mutta tässä yhteydessä oletetaan, että käytössä on riittävän hyviä satunnaislukugeneraattoreita. Satunnaistetulle algoritmille on tyypillistä, että se voi eri ajokerroilla tuottaa samoista syötteistä eri tuloksia.

Kun eri ajokerroilla algoritmi yleensä tuottaa erilaisia tuloksia ja myös algoritmin ajoaika vaihtelee samoilla syötteillä tehtyjen satunnaisten valintojen perusteella, voidaan puhua satunnaistetun algoritmin *aikavaatimuksen odotusarvosta*. Tämä käsite liittyy siis algoritmin toimintaan samalla syötteellä (ongelman samaa tapausta ratkotaan uudelleen ja uudelleen). Sitä ei pidä sekoittaa deterministisen algoritmin keskimääräisen tapauksen aikavaatimukseen, jossa tarkastellaan algoritmin aikavaatimuksen käyttäytymistä kaikilla kysymykseen tulevilla tapauksilla.

Satunnaistettuja algoritmeja voidaan luokitella esimerkiksi seuraavasti:

- *Numeerinen satunnaistettu algoritmi* ratkaisee likimääräisesti jonkin numeerisen ongelman.
- *Monte Carlo -algoritmi* voi tuottaa väärän tuloksen pienellä todennäköisyydellä, mutta on yleensä determinististä algoritmia nopeampi.
- *Las Vegas -algoritmi* ei tuota koskaan väärää tulosta ja on keskimäärin nopea, mutta ei välttämättä löydä ratkaisua ollenkaan.
- *Sherwood-algoritmi* tuottaa aina oikean ratkaisun. Sherwood-algoritmeja käytetään nopeuttamaan hankalien tapausten käsittelyä, kun deterministinen algoritmi toimii huomattavasti nopeammin keskimäärin kuin pahimmassa tapauksessa.

Numeerisista satunnaistetuista algoritmeista esitetään vain yksi esimerkki. Siinä käytetään merkintää uniform(a, b) väliltä $[a, b)$ olevasta satunnaisluvusta.

Esimerkki 8.1 (Numeerinen integrointi). Lasketaan funktion $f(x)$ integraali välillä $[0,1]$, kun $0 \leq f(x) \leq 1$, kuvan 8.1 algoritmilla.

```
    Function hitormiss(f, n)
(1) k := 0
(2) for i := 1 to n do
(3)         x := uniform(0, 1)
(4)         y := uniform(0, 1)
(5)         if y ≤ f(x) then k := k + 1
(6) od
(7) return k/n
```

Kuva 8.1: Satunnaistettu integrointi.

Kuvan 8.1 satunnaistetulla integroinnilla on käytännön merkitystä vain moniulotteisessa integroinnissa. Tavallisessa integroinnissa saadaaan yleensä parempi likiarvo helpommin jollain deterministisellä menetelmällä. Kuitenkaan mikään deterministinen menetelmä ei toimi hyvin kaikilla mahdollisilla funktioilla, kun taas satunnaistettu integrointi toimii aina.

□

8.1. Sherwood-algoritmit

Olkoon A deterministinen algoritmi, jolla T_{ave} ja $T_{\max}$ (keskimääräisen ja pahimman tapauksen aikavaatimukset) ovat eri kertaluokkaa. Merkitään $t_A(x)$:lla tapauksen x ratkaisemiseen tarvittavaa aikaa ja X_n:llä n:n kokoisten tapausten joukkoa. Olettaen, että jokainen $x \in X_n$ on yhtä todennäköinen, A:n käyttämä keskimääräinen aika n:n kokoisella tapauksella on

$$\bar{t}_A(n) = \sum_{x \in X_n} t_A(x)/|X_n|.$$

(Merkintä $|X|$ tarkoittaa joukon X kokoa.) Algoritmin A ominaisuuksista seuraa, että on olemassa tapaus $y \in X_n$, jolle $t_A(y) \gg \bar{t}_A(n)$. Tavoitteena on muodostaa sellainen satunnaistettu algoritmi B, että $t_B(x) \approx \bar{t}_A(n) + s(n)$ jokaiselle $x \in X_n$, kun $t_B(x)$ on algoritmin B tarvitsema aika x:n ratkaisemiseen ja $s(n)$ kustannusten tasauksen aiheuttama lisäkustannus.

Algoritmi B voi joskus viedä enemmän aikaa kuin $\bar{t}_A(n) + s(n)$, mutta tämä riippuu vain B:n tekemistä satunnaisista valinnoista eikä tapauksesta x. Nyt $\bar{t}_B(n) \approx \bar{t}_A(n) + s(n)$, joten B on keskimäärin $s(n)$:n verran hitaampi kuin A.

Esimerkki 8.2 Jokainen lajittelualgoritmi, joka toimii keskimäärin paremmin kuin pahimmassa tapauksessa (esim. pikalajittelu), voidaan helposti muuntaa Sherwood-algoritmiksi sekoittamalla lajiteltava aineisto riittävällä määrällä satunnaisia vaihtoja ennen lajittelua. □

Esimerkki 8.3 Tarkastellaan linkitettyjä järjestettyjä listoja, joissa next-indeksejä seuraamalla löydetään listan alkiot suuruusjärjestyksessä kunnes next-indeksin arvo 0 ilmoittaa listan loppuneen (ks. alla olevaa taulukkoa). Pienin alkio on val(head) eli tässä tapauksessa val(head) = 1, sillä head osoittaa neljättä alkiota (ks. kuva 8.2).

i	1	2	3	4	5	6	7
val(i)	2	3	13	1	5	21	8
next(i)	2	5	6	1	7	0	3

Kuva 8.2: Järjestetty lista.

Tehtävänä on etsiä alkio x listasta, jossa on n alkiota. Oletaan, että x todella esiintyy listassa ja että kukin alkio esiintyy listassa vain kerran. Jokainen deterministinen algoritmi vie pahimmassa tapauksessa ajan $O(n)$. Listan läpikäynnissä tarvitaan kuvan 8.3 funktiota. Keskimääräisessä tapauksessa deterministinen algoritmi saadaan toimimaan kuitenkin ajassa $O(\sqrt{n})$ (ks. kuva 8.4).

```
    Function search(x, i)
(1) while x > val(i) do i := next(i) od
(2) return i
```

Kuva 8.3: Search.

```
    Function D(x)
(1) i := head
(2) max := val(i)
(3) for j := 1 to ⌊√n⌋ do
(4)         y := val(j)
(5)         if max < y ≤ x then
(6)                 i := j
(7)                 max := y
(8)         end
(9) od
(10) return search(x, i)
```

Kuva 8.4: D.

Kuvan 8.4 algoritmi voidaan satunnaistaa. Tulokseksi saadaan algoritmi, joka toimii keskimäärin ajassa $O(\sqrt{n})$ riippumatta alkiosta x ja listan tallennusjärjestyksestä (merkitään uniform($a \ldots b$):lla satunnaista kokonaislukua väliltä $[a, b]$) (ks. kuva 8.5). □

Esimerkki 8.4 (Universaali hajautus). Oletetaan, että käytössä on kiinteä hajautusfunktio. Vaikka hajautusfunktio toimisi keskimääräisessä tapauksessa hyvin, voidaan muodostaa syötteitä, jotka kiinteä hajautusfunktio "hajauttaa" samaan osoitteeseen. Sherwood-tyyppinen ratkaisu tähän ongelmaan on tasoittaa hajautuksen hyvyys eri syötteillä. Tämä

```
    Function S(x)
(1) i := head
(2) max := val(i)
(3) for k := 1 to ⌊√n⌋ do
(4)         j := uniform(1 ... n)
(5)         y := val(j)
(6)         if max < y ≤ x then
(7)                 i := j
(8)                 max := y
(9)         end
(10) od
(11) return search(x, i)
```

Kuva 8.5: Satunnaistettu etsintä listasta.

on mahdollista, kun luovutaan kiinteästä hajautusfunktiosta, ja sen sijaan käytetään hajautusfunktioiden kokoelmaa, josta hajautusfunktio satunnaisesti valitaan. Näin hajautus tulee riippumattomaksi syötteestä (eli hajautettavasta avainten joukosta).

Olkoon tehtävänä hajauttaa avainten joukko U välille $[0, \ldots, m-1]$, ja olkoon $\mathbf{H}$ jokin tämän hajautuksen suorittavien hajautusfunktioiden kokoelma. Kokoelmaa $\mathbf{H}$ sanotaan universaaliksi, jos jokaisella erillisistä joukon U avaimista muodostuvalla parilla (x, y) sellaisten hajautusfunktioiden $h \in \mathbf{H}$ lukumäärä, jotka aiheuttavat näiden avainten suhteen yhteentörmäyksen, on korkeintaan $|\mathbf{H}|/m$, ts.

$$\big|\{h \in \mathbf{H} \mid h(x) = h(y)\}\big| \leq |\mathbf{H}|/m.$$

Jos siis h valitaan satunnaisesti universaalien hajautusfunktioiden kokoelmasta $\mathbf{H}$, niin minkä tahansa avainparin (x, y) yhteentörmäyksen todennäköisyys on korkeintaan $1/m$.

Universaali hajautusfunktioiden kokoelma voidaan muodostaa esimerkiksi seuraavasti: Olkoon p, $p < m$, alkuluku, ja olkoot s ja t kokonaislukuja. Määritellään funktiot $h_{s,t} : U \to \{0, 1, \ldots, m-1\}$ säännöllä $h_{s,t}(x) = \big((sx + t) \mod p\big) \mod m$. Nyt

$$\mathbf{H}^* = \{h_{s,t} \mid 1 \leq s < p, \quad 0 \leq t < p\}$$

on universaali hajautusfunktioiden kokoelma. □

8.2. Las Vegas -algoritmit

Vaikka Sherwood-algoritmi on käyttäytymiseltään tasapainoisempi kuin deterministinen algoritmi, josta se johdettiin, se ei kuitenkaan ole alkuperäistä algoritmia nopeampi. Las Vegas -algoritmi sen sijaan tarjoaa jossain tapauksissa mahdollisuuden nopeuttaa determinististä algoritmia. Toisaalta haittana on se, että ratkaisun löytymiseen tarvittavalle ajalle ei ole ylärajaa. Las Vegas -algoritmi ei edes välttämättä aina löydä ratkaisua lainkaan.

Las Vegas -algoritmilla on tavallisesti ulostuloparametri *success*, joka on tosi, jos ratkaisu on löytynyt. Tyypillinen Las Vegas -algoritmin kutsu tapauksen x ratkaisemiseksi on muotoa $LV(x, y, success)$, kun y on tulosparametri, johon mahdollinen ratkaisu tallennetaan. Olkoon $p(x)$ ratkaisun löytymistodennäköisyys, kun algoritmia sovelletaan tapaukseen x. Vaaditaan, että $p(x) > 0$ kaikilla tapauksilla x. Olkoot $s(x)$ ja $e(x)$ suoritusaikojen odotusarvot onnistumisen ja epäonnistumisen tapauksessa. Tarkastellaan kuvan 8.6 algoritmia.

```
    Function A(x)
(1) repeat
(2)         LV(x, y, success)
(3) until success
(4) return y
```

Kuva 8.6: Algoritmi A.

Olkoon $t(x)$ algoritmin A suoritusajan odotusarvo. Välittämättä silmukan hallintaan menevästä ajasta saadaan palautuskaava

$$t(x) = p(x)s(x) + (1 - p(x))(e(x) + t(x)),$$

josta ratkaisemalla saadaan

$$t(x) = s(x) + \frac{1 - p(x)}{p(x)} e(x).$$

Kun tavoitteena on arvon $t(x)$ minimointi, on löydettävä sopiva kompromissi arvojen $p(x)$, $s(x)$ ja $e(x)$ välillä. Ei esimerkiksi ole välttämättä järkevää pyrkiä kasvattamaan arvoa $p(x)$, jos se aiheuttaa arvon $e(x)$ kasvamisen.

k	p	s	e	t
0	1.00	114.0		114.0
1	1.00	39.6		39.6
2	0.88	22.5	36.7	25.2
3	0.49	13.5	15.1	29.0
4	0.26	10.3	8.8	35.1
5	0.16	9.3	7.3	46.9
6	0.14	9.1	7.0	53.5
7	0.13	9.0	7.0	56.0
8	0.13	9.0	7.0	56.0

Kuva 8.7: Kuningatarten sijoittelu shakkilaudalle Las Vegas -algoritmilla.

Esimerkki 8.5 (Kahdeksan kuningattaren ongelma). Tehtävänä on sijoittaa kahdeksan kuningatarta shakkilaudalle niin, etteivät ne uhkaa toisiaan. Ongelma voidaan ratkaista

deterministisesti peruutusta käyttäen läpikäymällä eri sijoitusvaihtoehtoja, kunnes vaadittu kuningatarten sijoittelu löytyy. Algoritmi voidaan satunnaistaa ja samalla nopeuttaa sijoittamalla aluksi k, $0 \leq k \leq 8$, kuningatarta laudalle satunnaisesti, ja sen jälkeen sijoittaa loput $8-k$ kuningatarta deterministisellä algoritmilla. Koska kaikkia alkusijoituksia ei voida täydentää koko ongelman ratkaisuksi, on kyseessä Las Vegas -algoritmi. Kuvassa 8.7 on annettu eri k:n arvoilla onnistumistodennäköisyys (p), tutkittavien asemien lukumäärän odotusarvo onnistumisen tapauksessa (s), epäonnistumisen tapauksessa (e) ja näistä edellä esitetyllä palautuskaavalla lasketut arvot (t). Kun $k = 0$, niin kyseessä on deterministinen algoritmi. Huomataan, että yhden kuningattaren satunnaisen sijoittamisen jälkeen löydetään aina ratkaisu. □

Esimerkki 8.6 Tarkastellaan identtisten prosessorien rengasta, jossa kaikki prosessorit ovat yhteydessä kahteen naapuriinsa ja jokaisessa prosessorissa on sama ohjelma ja data. Prosessorirenkaan hyödyntämiseksi olisi usein tärkeää valita renkaalle johtaja, joka jakaa tehtäviä muille prosessoreille. Deterministinen, kaikissa prosessoreissa samalla tavalla toimiva algoritmi ei voi rikkoa renkaan symmetriaa ja löytää johtajaa.

Prosessorirenkaan johtaja voidaan löytää Las Vegas -tyyppisellä satunnaistetulla algoritmilla, jos kullakin prosessorilla on oma satunnaislukugeneraattiorinsa. Jokainen prosessori valitsee satunnaisen kokonaisluvun väliltä $1 \ldots n$. Ykkösen valinneet prosessorit lähettävät tästä tiedon muille prosessoreille. Jos mikään prosessori ei valinnut ykköstä, toistetaan algoritmi. Jos ykkösen valitsi täsmälleen yksi prosessori, tulee siitä renkaan johtaja. Jos ykkösen valitsi useampi kuin yksi prosessori, jatkavat nämä satunnaislukujen valintoja, kunnes johtaja on yksikäsitteisesti määrätty.

□

8.3. Monte Carlo -algoritmit

Monte Carlo -algoritmi voi tuottaa väärän tuloksen, mutta se löytää kuitenkin oikean ratkaisun suurella todennäköisyydellä riippumatta tapauksesta. Monte Carlo -algoritmi on p-oikea, jos se antaa oikean ratkaisun vähintään todennäköisyydellä p.

Esimerkki 8.7 Otetaan tehtäväksi selvittää, onko taulukossa $T[1 \ldots n]$ majoriteettialkiota eli alkiota, jonka esiintymiä on enemmän kuin $n/2$ kappaletta. (Tälle ongelmalle on olemassa tehokas deterministinen algoritmi. Tässä esitettävät algoritmit siis ainoastaan havainnollistavat Monte Carlo -algoritmeihin liittyviä käsitteitä.)

Esitetään ensin yksinkertainen 1/2-oikea algoritmi Maj (ks. kuva 8.8). Jos joukossa T ei ole majoriteettialkiota, Maj(T) antaa aina oikean vastauksen epätosi. Koska algoritmi ei aina löydä oikeaa vastausta myönteisessä tapauksessa, sanotaan, että se on *tosiharhainen* (true-biased).

Virhetodennäköisyyttä 1/2 voidaan pienentää. Kuvan 8.9 algoritmi perustuu seuraavaan päättelyyn. Jos majoriteettialkiota ei ole, niin jokainen algoritmin Maj kutsu palauttaa varmasti arvon epätosi. Näin tekee siis myös Maj2. Jos majoriteettialkio on olemassa,

```
    Function Maj(T[1...n])
(1) i := uniform(1...n)
(2) x := T[i]
(3) k := 0
(4) for j := 1 to n do
(5)       if T[j] = x then k := k + 1 end
(6) od
(7) return (k > n/2)
```

Kuva 8.8: Majoriteettialkion etsivä 1/2-oikea Monte Carlo -algoritmi.

palauttaa Maj arvon tosi todennäköisyydellä p joka on vähintään $1/2$, samoin tekee Maj2 ensimmäisellä kutsulla Maj. Jos ensimmäinen kutsu palauttaa epätoden (todennäköisyys tälle on $1-p$), niin toinen kutsu voi vielä palauttaa oikean vastauksen tosi todennäköisyydellä p. Maj2 palauttaa siis oikean vastauksen tosi todennäköisyydellä

$$p + (1-p)p = 1 - (1-p)^2 \geq 3/4.$$

Oikean vastauksen todennäköisyys kasvaa, koska peräkkäiset kutsut ovat toisistaan riippumattomia.

```
    Function Maj2(T)
(1) if Maj(T) then return true end
(2) else return Maj(T)
(3) od
```

Kuva 8.9: Parannettu versio majoriteettialkiota etsivästä Monte Carlo -algoritmista.

Kuvan 8.9 algoritmin ideaa voidaan yleistää, ja saadaan algoritmi, joka löytää majoriteettialkion ennalta asetetulla virhetodennäköisyydellä $\epsilon > 0$. Tämä algoritmi on esitetty kuvassa 8.10. Sen aikavaatimus on $O(n \log(1/\epsilon))$. □

```
    Function Maj3(T, ε)
(1) k := ⌈log(1/ε)⌉
(2) for i := 1 to k do
(3)       if Maj(T) then return true end
(4)       return false
(5) od
```

Kuva 8.10: Majoriteettialkion etsivä ϵ-oikea Monte Carlo -algoritmi.

Tarkastellaan luonnollisen luvun n jaottomuuden tutkimista. (Onko annettu luku n alkuluku?) Deterministisellä algoritmilla jaottomuuden tutkiminen vaatii ajan $O(\sqrt{n})$, sillä on tutkittava jakojäännöksiä $n \mod i$, kun i saa arvot $2, \ldots, \sqrt{n}$.

Kuvassa 8.11 esitetään ajassa $O(\log^3 n)$ toimiva (todistus sivuutetaan) Monte Carlo -algoritmi MC-prime. Algoritmi perustuu ns. vahva näennäisalkuluku -ominaisuuden tutkimiseen.

Oletetaan, että n on pariton ja $n > 3$. Olkoot s ja t sellaiset kokonaisluvut, että $n-1 = 2^s t$. Olkoon edelleen a jokin kokonaisluku väliltä $[2 \ldots n-2]$. Jos $a^t \equiv 1 \pmod{n}$ tai jos on olemassa sellainen luku i $(0 \leq i < s)$, että $a^{2^i t} \equiv -1 \pmod{n}$, niin n on *vahva näennäisalkuluku kannan a suhteen*. Vahvojen näennäisalkulukujen käyttö perustuu siihen, että alkuluvut ovat vahvoja näennäisalkulukuja kaikkien kantojen suhteen. Toisaalta kaikki vahvat näennäisalkuluvut eivät ole alkulukuja. Tästä syystä kuvassa 8.11 esitettävä algoritmi antaa joskus väärän vastauksen.

```
     Function MC-prime(n)
 (1) a :=uniform(2...n-1)
 (2) s := max{i | (n-1) mod 2^i = 0}
 (3) t := (n-1)/2^s
 (4) b := a^t mod n
 (5) if b = 1 or b = n-1 then return true end
 (6) for i := 1 to s-1 do
 (7)         b := b^2 mod n
 (8)         if b = 1 then return false end
 (9)         if b = n-1 then return true end
(10) od
(11) return false
```

Kuva 8.11: MC-prime.

Kun algoritmi antaa vastauksen false, n on aina jaollinen. Kun algoritmi antaa vastauksen true, n on jaoton todennäköisyydellä $> 3/4$. Algoritmin virhetodennäköisyys on siis pienempi kuin $1/4$.

Esimerkki 8.8 Jos luvun 289 jaollisuutta tutkittaessa tulisi luvuksi a valittua 158, niin algoritmi MC-prime hyväksyisi luvun 289 alkuluvuksi vaikka $289 = 17^2$. Todellisuudessa 289 on siis vain vahva näennäisalkuluku kannan 158 suhteen. □

Jotta edellä esitetty algoritmi olisi käyttökelpoinen, on $a^m \mod n$ voitava laskea tehokkaasti kaikilla positiivisilla kokonaisluvuilla a, m ja n. Käytetään merkintää

$$(m_k, m_{k-1}, \ldots, m_1, m_0)$$

luvun m binaariesityksestä (m_k on merkitsevin bitti). Kuvan 8.12 algoritmi laskee luvut $a^r \mod n$, kun r kasvaa kakkosella kertomisten ja ykkösen lisäysten avulla nollasta m:ään.

Esimerkki 8.9 Lasketaan $7^{560} \mod 561$. Luvun 560 binaariesitys on (1000110000), joten $k = 9$ ja algoritmin **for**-silmukka suoritetaan kymmenen kertaa. Alla olevassa taulukossa on muuttujien r ja d arvot kunkin silmukan suorituskerran jälkeen. Tulokseksi saadaan

Procedure moduloeksponentti (a, m, n)
$\{(m_k, m_{k-1}, \ldots, m_1, m_0)$ on luvun m binaariesitys $\}$
(1) $r := 0, d := 1$
(2) **for** $i := k$ **to** 0 **by** -1 **do**
(3) $\quad r := 2r$
(4) $\quad d := d^2 \mod n$
(5) $\quad$ **if** $m_i = 1$ **then** $r := r + 1$, $d := (d * a) \mod n$ **end**
(6) **od**
(7) **return** d

Kuva 8.12: Jakojäännöksen laskeminen.

i	9	8	7	6	5	4	3	2	1	0
m_i	1	0	0	0	1	1	0	0	0	0
r	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

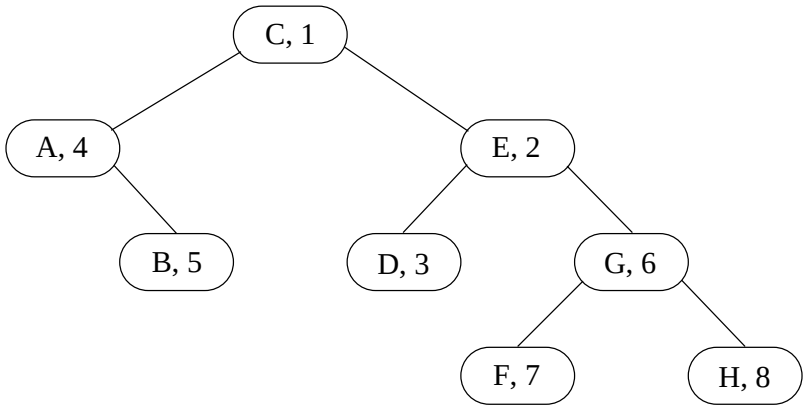
$7^{560} \mod 561 = 1$. Algoritmi laskee jakojäännöksiä $a^{2r} \mod n$ tai $a^{2r+1} \mod n$ kunnes lopuksi lasketaan arvo $a^m \mod n$. □

8.4. Satunnaistettu etsintäpuu

Myös sanakirjaongelma voidaan ratkaista satunnaistetusti. Tarkastellaan aluksi ratkaisua, josta käytetään nimitystä *satunnaistettu etsintäpuu*. Samaan tapaan kuin pikalajittelu toimii keskimääräisessä tapauksessa tehokkaasti, on myös satunnaisesti muodostettu etsintäpuu keskimääräisessä tapauksessa riittävän tasapainoinen. (Pikalajittelun valinta-alkiot itse asiassa määräävät etsintäpuun, sillä jokaista valinta-alkiota voidaan pitää oman alipuunsa juurena, jonka suhteen jäljellä olevat alkiot jaetaan kahteen joukkoon eli juuren vasemmaksi ja oikeaksi alipuuksi.) Ongelmaksi satunnaisesti muodostetun etsintäpuun kohdalla tulevat myöhemmät lisäykset ja poistot, jotka useasti muuttavat puun listamaiseksi, jos puun muotoa ei mitenkään tasapainoiteta. Lisäysten ja poistojen vaikutus puun muotoon voidaan eliminoida, kun tavanomaisten avainten (sanakirjajoukon alkioiden) lisäksi puun jokaiseen solmuun liitetään prioriteetti. Puu muodostetaan sellaiseksi, että se on etsintäpuu avainten suhteen ja kasa prioriteettien suhteen.

Kuvan 8.13 puu on etsintäpuu kirjaintunnusten aakkosjärjestyksen suhteen ja kasa numeroilla ilmaistujen prioriteettien suhteen. Annetut avaimet ja prioriteetit määräävät yksikäsitteisesti puun muodon, joka siis on riippumaton niistä alkioista, jotka on aikaisemmin lisätty puuhun ja myöhemmin sieltä poistettu.

Kun avain lisätään puuhun, valitaan siihen liittyvä prioriteetti satunnaisesti. Uusi solmu on lisättävä puuhun sekä etsintäpuu- että kasaominaisuus säilyttäen. Samoin on toimittava solmuja puusta poistettaessa. Molemmat operaatiot voivat vaatia puussa tehtäviä



Kuva 8.13: Satunnaistettu etsintäpuu.

rotaatioita, mutta voidaan todistaa, että kutakin lisäys- ja poisto-operaatiota kohti riittää keskimääräisessä tapauksessa kaksi rotaatiota.

Toinen satunnaistettu tapa ratkaista sanakirjaongelma on *hyppylista* (skip list). Tarkastellaan aluksi yhteen suuntaan linkitettyä järjestettyä listaa. Alkion etsimiseksi tästä rakenteesta on pahimmassa tapauksessa tutkittava koko lista. Ratkaisua voidaan tehostaa lisäämällä listan joka toiseen alkioon osoitin myös etäisyydelle 2. Nyt riittää pahimmassakin tapauksessa tutkia noin puolet alkioista. Tämä idea voidaan yleistää ja lisätä joka 2^i. alkioon osoitin etäisyydelle 2^i. Näin saadaan listarakenne, joka vastaa täydellistä tai melkein täydellistä binaaripuuta. Ongelmaksi tulee kuitenkin rakenteen ylläpito poisto- ja lisäysoperaatioiden suhteen. Hyppylista on listarakenne, jossa on edellä esiteltyjä "ylimääräisiä" osoittimia, ja jonka tasapainoa ylläpidetään satunnaisesti.

Hyppylistan jokaiselle alkiolle on määrätty taso, joka ilmoittaa, montako osoitinta alkiosta lähtee. Jokaisen alkion taso on vähintään yksi. Alkion i. osoitin osoittaa seuraavaan alkioon, jonka taso on vähintään i. Kun alkio lisätään hyppylistaan, sille määrätään taso satunnaisesti jonkun edeltäkäsin valitun satunnaisfunktion perusteella. Voidaan käyttää esimerkiksi jakaumaa

$$\mathrm{P}\{\text{alkion taso} = 1\} = 0.5,$$
$$\mathrm{P}\{\text{alkion taso} = 2\} = 0.25,$$
$$\mathrm{P}\{\text{alkion taso} = 3\} = 0.125 \text{ jne.}$$

Hyppylistan alkion suurin mahdollinen taso on etukäteen kiinnitetty vakio. Alkion etsintä hyppylistasta tapahtuu seuraavasti: Etsintä aloitetaan käymällä läpi ylimmän tason osoittimia kunnes huomataan, että seuraava alkio on suurempi tai yhtä suuri kuin etsittävä alkio. Tämän jälkeen etsintää jatketaan samalla tavalla mutta yhtä tasoa alempaa. Etsintää jatketaan niin kauan, kunnes tasolla 1 ei voida enää edetä, koska seuraava alkio on joko yhtä suuri tai suurempi kuin etsittävä alkio.Viimeksi tarkistetaan, onko tämä alkio yhtä suuri kuin etsittävä alkio. Lisäysalgoritmi etsii lisäyskohdan edellä esitetyllä etsintäalgoritmilla. Uuden alkion luonnin yhteydessä määrätään sille taso satunnaisesti. Uuden alkion luonnin yhteydessä on myös huolehdittava osoittimien päivityksestä. Tätä

varten on ylläpidettävä tietoa lisäyspaikan ylittävistä eri tasoilla olevista osoittimista. Ne osoittimet, jotka ovat korkeammalla tasolla kuin uuden alkion taso, pysyvät ennallaan. Korkeintaan uuden alkion tasolla oleviin osoittimiin on tehtävä muutokset. Poistoalgoritmi on pääpiirteissään samanlainen kuin etsintä- ja lisäysalgoritmit. Erona lisäysalgoritmin yhteydessä tehtäviin osoitinpäivityksiin on se, että nyt joitakin osoittimia "jatketaan" yli poistokohdan.

Hyppylistarakenteen tärkein parametri on luku p, joka kertoo todennäköisyyden sille, että vähintään tasolla i oleva alkio on tasolla $i+1$. Alkion tason määrääminen tapahtuu toistamalla todennäköisyydellä p onnistuvaa koetta (käytännössä p:stä riippuvan ehdon toteuttavan satunnaisluvun generointia) kunnes koe ei enää onnistu tai on saavutettu rakenteelle määrätty suurin mahdollinen taso. Tyypillinen p:n arvo on 0.5. Se antaa edellä mainitut todennäköisyydet 0.5, 0.25, 0.125 jne. alkioiden tasoille. Jos p pienenee, merkitsee se alkioiden tasojen pienentymistä ja lopulta sitä, että rakenne alkaa muistuttaa tavallista listaa.

Hyppylistan etsintäkustannuksen pahin tapaus on tietenkin $O(n)$. Pahin tapaus sattuu kuitenkin äärimmäisen harvoin. Keskimääräisen tapauksen selvittämiseksi on määrättävä etsintäpolun pituuden odotusarvo, jonka voidaan osoittaa olevan korkeintaan

$$\frac{1}{p}\log_x n + \frac{1}{1-p},$$

kun logaritmin kantaluku x on $1/p$. Tyypillisellä p:n arvolla 0.5 saadaan sekä logaritmin kantaluvuksi että kertoimeksi kakkonen. Lisäys- ja poisto-operaatioiden kohdalla tähän on vielä lisättävä osoitinpäivityksiin kuluva aika. Kaikkien operusoperaatioiden aikavaatimusten odotusarvo eli hyppylistan keskimääräisen tapauksen aikavaatimus on kuitenkin logaritminen.

Luku 9

Rinnakkaisalgoritmeista

Perinteisen ajattelutavan mukaan tietokoneen peruskomponentit ovat (yksi) prosessori, muisti ja oheislaitteet. Rinnakkaistietokoneissa on kuitenkin käytössä monta prosessoria, jotka yhteistyössä ratkaisevat samaa algoritmista ongelmaa. Aluksi on syytä erottaa käsitteet *rinnakkaisuus* (parallelism) ja *samanaikaisuus* (concurrency). Termiä samanaikaisuus käytetään, kun pyritään hallitsemaan yhtäaikaa tapahtuvia toimintoja. Samanaikaisuuden hallintaa tapahtuu esimerkiksi tietokoneverkon ylläpidossa. Rinnakkaisuus puolestaan liittyy useiden prosessorien käyttöön saman yksittäisen ongelman ratkaisemisessa. Rinnakkaisohjelmoinnissa käytettävät prosessorit ovat yleensä keskenään samanlaisia ja ne kommunikoivat yhteisen muistin välityksellä.

9.1. Peruskäsitteitä

Rinnakkaisalgoritmit esitetään usein PRAM:ksi (parallel random acces machine, rinnakkaishajasaantikone) kutsutun teoreettisen mallin avulla. Mallin etuna on, ettei tarvitse sitoutua mihinkään erityiseen konearkkitehtuuriin. Samalla saadaan teoriasta ja algoritmeista yleispäteviä. Käytännössä tehtävän ratkaiseminen tietyssä rinnakkaiskoneessa voi vaatia lisäksi konekohtaista suunnittelua. PRAM-mallilla on seuraavat ominaisuudet:

- Prosessorit on numeroitu $1, \ldots, p$. Jokainen prosessori tietää oman numeronsa.
- Kaikki prosessorit suorittavat samaa ohjelmaa synkronisesti; kukin prosessori P_i ylläpitää omaa ohjelmalaskuria PC_i.
- Kullakin prosessorilla on äärettömän suuri oma muisti; prosessoreilla on myös äärettömän suuri yhteinen muisti. Muisteissa yhteen muistipaikkaan voidaan tallentaa äärettömän suuria lukuja, joiden käsittelyn oletetaan olevan mahdollista yhdessä aikayksikössä.
- Laskenta suoritetaan käyttämällä prosessoreiden omia muistipaikkoja sekä yhteistä muistia.

Riippuen siitä, miten yhteisen muistin muistipaikkaan kohdistuvat samanaikaiset saantioperaatiot käsitellään, saadaan PRAM:ista erilaisia versioita:

- **EREW** (Exclusive Read, Exclusive Write) Ei sallita yhtäaikaista lukua eikä kirjoitusta yhteiseen muistiin.
- **CREW** (Concurrent Read, Exclusive Write) Sallitaan yhtäaikainen luku, mutta ei kirjoitusta.
- **CRCW** (Concurrent Read, Concurrent Write) Sallitaan sekä yhtäaikainen luku että kirjoitus.

Näistä EREW ja CRCW ovat yleisimmät. CRCW PRAM -mallista on vielä useita eri versioita riippuen siitä, miten samanaikainen kirjoitus hoidetaan.

Olkoon PRAM-mallien A ja B välillä relaatio $A \preceq B$, jos mallissa A ei voida suorittaa mitään tehtävää nopeammin kuin mallissa B. Toisin sanoen malli A on heikompi kuin malli B. Tällöin malleille saadaan järjestys EREW $\preceq$ CREW $\preceq$ CRCW.

Rinnakkaisalgoritmit kirjoitetaan samantapaisella pseudokoodilla kuin aikaisemmin on kirjoitettu peräkkäisalgoritmit. Rinnaiskäsittelyn mahdollistaa rakenne **for** $i \in X$ **pardo**, joka tarkoittaa joukon X tehtävien suorittamista rinnakkain. Tässä yhteydessä pitäisi aina olla selvää, miten tehtävät jaetaan eri prosessoreille. Tämä tehtävä jätetään tässä tarkastelussa kääntäjän huoleksi. Ohjelman suoritusta jatketaan **for ... pardo** -lauseen jälkeen, kun kaikki rinnakkaiset tehtävät on saatu suoritettua. Oletetaan, että kääntäjä huolehtii myös synkronoinnista lisäämällä tyhjiä operaatioita niihin ohjelmakohtiin, joiden kesto on muita lyhyempi.

Ensimmäisenä esimerkkinä tarkastellaan ns. *viestin levitys* -ongelmaa, jossa tehtävänä on kopioida annettu tietoalkio x yhteensä n:ksi kopioksi yhteiselle muistialueelle taulukkoon C. Tämä ongelma olisi helppo ratkaista vakioajassa, jos samanaikainen lukeminen olisi sallittua. Tällöin tarvittaisiin $n-1$ prosessoria, jotka lukisivat yhtä aikaa kopioitavan alkion ja kirjoittaisivat sen paikkaan $C[p+1]$, missä p on prosessorin numero. Tarkastellaan tilannetta, jossa laskennan malli on EREW PRAM. Saadaan kuvan 9.1 algoritmi.

```
     C[1] := x
     for i ∈ [1 ... ⌈log n⌉] do
(1)      for j ∈ [2^(i-1) + 1 ... min{2^i, n}] pardo
(2)          C[j] := C[j - 2^(i-1)]
(3)      od
(4)  od
```

Kuva 9.1: Viestin levitys EREW PRAM -mallissa.

.

Algoritmi perustuu kahdentamistekniikkaan, jossa jokaisella askeleella jo olemassa olevista alkioista luodaan uusi kopio. Algoritmin kuvaus on annettu ns. *isäntä-orjat* -tyylillä:

yksi prosessori on isäntä ja se kutsuu orjiaan (muita prosessoreja) ainoastaan **for ... pardo** -lauseessa. Kun käytössä on p prosessoria, niin tehtävien jakaminen prosessoreille käy identiteettikuvausta käyttäen. Algoritmi voidaan esittää myös siten, että kuvataan ainoastaan yksittäisen prosessorin toiminta (kuva 9.2). Tällöin algoritmit ovat normaaleja peräkkäisalgoritmeja, mutta prosessorien välisiä kytkentöjä (synkronointia) on vaikeampi ymmärtää. Jatkossa käytetään ensiksi esitettyä kuvaustapaa.

{# on prosessorin oma numero}
(1) **if** $\# = 1$ **then** $C[1] := x$ **end**
(2) **for** $i \in [1 \ldots \lceil \log N \rceil]$ **do**
(3) **if** $\# \in [2^{i-1} + 1 \ldots min\{2^i, n\}]$ **then** $C[\#] := C[\# - 2^{i-1}]$

Kuva 9.2: Viestin levitys -algoritmi yksittäisen prosessorin toimintana.

Rinnakkaisalgoritmien hyvyyttä analysoitaessa ollaan kiinnostuneita erityisesti seuraavista parametreista, jotka ilmoitetaan syötteen koon n funktiona:

- *suoritusaika* eli suoritettavien rinnakkaisaskelten lukumäärä $T(n)$ pahimmassa tapauksessa,
- *prosessorien lukumäärä* $P(n)$ ja
- *kustannus* $P(n) \cdot T(n)$.

Esimerkiksi edellä kuvattu viestin levitys -ongelman ratkaiseva algoritmi vaatii ajan $O(\log n)$, kun tehtävänä on tehdä annetusta alkiosta n kopiota ja kun käytössä on n prosessoria. Täten kustannus on $O(n \log n)$.

Rinnakkaislaskennan perimmäisenä tarkoituksena on suunnitella algoritmeja, jotka hyödyntävät käytettävissä olevia prosessoreja maksimaalisella tavalla ja jotka toimivat niin nopesti kuin mahdollista. Tavoitteet voidaan täsmällisemmin muotoilla seuraavasti:

1. *Kustannusoptimaalisuus.* Algoritmi on kustannusoptimaalinen, jos $P(n) \cdot T(n) = t(n)$, kun $t(n)$ on parhaan mahdollisen peräkkäisalgoritmin käyttämä aika.
2. Aikaoptimaalisuus.
3. Algoritmien toimiminen mahdollisimman heikossa mallissa.

9.2. Rinnakkaisalgoritmien suunnittelutekniikoita

9.2.1. Raaka voima

Raa'an voiman menetelmän voisi tiivistää seuraavaan lauseeseen:

> Käytä prosessoreita niin paljon kuin pystyt hyödyntämään.

Yleensä raa'an voiman käyttö tuottaa ongelmalle nopean ratkaisun, joka ei ole kuitenkaan kustannusoptimaalinen. Raa'an voiman käyttö esimerkiksi merkkijonon etsinnässä tapahtuu seuraavasti (kuva 9.3): tutkitaan syötteen jokaisesta kohdasta, löytyykö etsittävä malli sieltä. Jos syötteen pituus on n ja etsittävän mallin pituus on m, niin syötteessä on $n - m + 1$ mahdollista aloituskohtaa. Jokaisesta aloituskohdasta tutkitaan m:llä prosessorilla täsmääkö malli. Tarvitaan siis yhteensä $m(n - m + 1)$ prosessoria. Algoritmi toimii CREW mallissa ajassa $O(1)$.

```
    {Syöte taulukossa S ja malli taulukossa M}.
(1) for i ∈ [1 ... n − m + 1] pardo
(2)       parallel if (M[1] = S[i] and ... and M[m] = S[i + m − 1]) then
(3)             Mallin esiintymä löytyi
```

Kuva 9.3: Merkkijonon haku raa'alla voimalla.

Kuvan 9.3 algoritmissa on käytetty myös parallel if -vertailua, jossa kaikki erilliset vertailut suoritetaan rinnakkain.

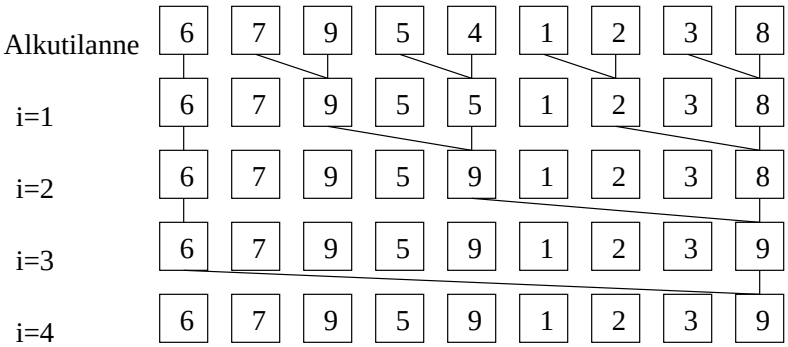
9.2.2. Turnaustekniikka

Tarkastellaan seuraavana esimerkkinä maksimin määräämistä. Olkoon yhteisellä muistialueella annettuna n alkiota. Tehtävänä on etsiä annetuista alkioista suurin. Useimpien rinnakkaisalgoritmien takaa löytyy jonkinlainen puurakenne. Puussa haarautuminen merkitsee jakoa osatehtäviin, jotka voidaan ratkaista samanaikaisesti. Yksinkertaisimmillaan kyseessä on tasapainoinen binaaripuutekniikka. Maksimin määrämisen yhteydessä tätä kutsutaan *turnaustekniikaksi*. Siinä maksimikandidaatit jaetaan pareihin ja pareja verrataan samanaikaisesti. Tätä toistetaan, kunnes jäljellä on yksi alkio, maksimi. Algoritmissa siis tavallaan käydään läpi tasapainoista binaaripuuta tasoittain lehtisolmuista lähtien kohti juurta.

```
(1) for i ∈ [1 ... ⌈log n⌉] do
(2)       for j ∈ [2^(i−1) + 1 ... n] pardo
(3)             if (n − j) mod 2^i = 0 then A[j] := max{A[j − 2^(i−1)], A[j]}
(4)  od od
(5) return (A[n])
```

Kuva 9.4: Maksimin etsintä turnaustekniikalla.

Esimerkki algoritmin toiminnasta on kuvassa 9.5. Algoritmin toiminta on helppo ymmärtää, kun n on kakkosen potenssi. Tällöin taustalla oleva puu on täydellinen. Algoritmi toimii kuitenkin kaikilla n:n arvoilla. Algoritmin suoritusaika on $O(\log n)$, kun käytössä on n prosessoria. Algoritmin kustannus on $O(n \log n)$. Koska samanaikaista lukua eikä kirjoitusta suoriteta, algoritmi toimii EREW PRAM -mallissa. Suurimman osan ajasta useat prosessorit ovat joutilaina, sillä algoritmissa suoritetaan ainoastaan $O(n)$ sijoituslausetta (mutta $O(n \log n)$ vertailua).



Kuva 9.5: Maksimin määrääminen turnaustekniikalla

9.2.3. Lohkomistekniikka

Monet ongelmat voidaan ratkaista kustannusoptimaalisesti käyttämällä *lohkomistekniikkaa*. Ajatuksena on jakaa ongelma ensin lohkoihin, määrätä näiden lohkojen ratkaisut peräkkäisalgoritmilla ja vasta sitten yhdistää lohkojen ratkaisut koko ongelman ratkaisuksi. Tätä kutsutaan myös *katkaisutekniikaksi*. Voidaan nimittäin ajatella, että ongelmaa jaetaan osiin hajota ja hallitse -periaatteella, mutta jakaminen katkaistaan normaalia aikaisemmin, siis ennen kuin päästään vakiokokoisiin osaongelmiin.

Maksimiongelman yhteydessä lohkomistekniikka toimii seuraavasti (oletetaan, että käytössä on p prosessoria):

1. Jaa syöte p:hen likimain yhtä suureen lohkoon; lohkon kooksi tulee enintään $\lceil n/p \rceil$, kun n on syötteen koko.

2. Määrää jokaisen lohkon maksimi rinnakkain. Kukin prosessori ratkaisee oman ongelmansa peräkkäisalgoritmilla.

3. Määrää lohkojen maksimien maksimi esimerkiksi turnaustekniikalla.

Algoritmin suoritusaika on $O(n/p+\log p)$ ja kustannus $O(n+p\log p)$. Jos $p \leq n/\log n$, niin kustannus on $O(n)$, koska ensimmäinen termi dominoi kustannuksia. Täten algoritmi toimii logaritmisessa ajassa, vaikka käytössä olisi vain $n/\log n$ prosessoria.

Tarkastellaan seuraavaksi alkusummien laskemista lohkomistekniikalla, jolloin algoritmista saadaan kustannusoptimaalinen. Alkusummatehtävässä on muodostettava annetun lukuvektorin $A[1,\ldots,n]$ kaikki alkusummat, ts. on muodostettava vektori

$$(A[1], A[1]+A[2], \ldots, \sum_{j=1}^{i} A[j], \ldots, \sum_{j=1}^{n} A[j]).$$

Oletetaan, että käytössä on p prosessoria. Algoritmille saadaan seuraavat vaiheet:

1. Jaa syöte lohkoihin, joiden koko on n/p.

2. Laske kunkin lohkon alkioiden summa samanaikaisesti peräkkäisalgoritmilla.

3. Laske lohkojen osasummille alkusummat ajassa $O(\log p)$ käytössä olevilla p:llä prosessorilla.

4. Lisää kunkin lohkon ensimmäiseen alkioon edellisen lohkon alkusumma.

5. Laske alkusummat kunkin lohkon sisällä peräkkäisalgoritmilla.

Alkusummat voidaan myös laskea turnustekniikalla Algoritmin 9.4 tapaan.

9.2.4. Brentin lause

Viestin levitys -ongelmaa ja maksimiongelmaa ratkaistaessa oletettiin, että prosessorien määrää voitiin aina lisätä ongelman koon kasvaessa (*rajoittamaton rinnakkaisuus*). Käytännössä prosessorien määrä on kuitenkin jokin ennalta kiinnitetty vakio, jonka suuruus riippuu käytössä olevasta rinnakkaiskoneesta (*rajoitettu rinnakkaisuus*).

Algoritmien suunnittelu on helpompaa, jos sallitaan rajoittamaton rinnakkaisuus. Seuraava lause kertoo, kuinka näitä algoritmeja voidaan soveltaa rajoitetun rinnakkaisuuden tapauksessa. Lause sanoo myös, että usein tärkeämpää on analysoida algoritmissa suoritettavien operaatioiden lukumäärää kuin prosessorien määrää.

Lause 9.1 *(Brent) Jos laskenta C vaatii ajan t, jona aikana suoritetaan yhteensä q operaatiota, ja jos käytössä on riittävä määrä prosessoreja, jotka suorittavat operaatiot vakioajassa, niin C voidaan suorittaa p:llä prosessorilla ajassa $t + q/p$.*

Todistus. Oletetaan, että askeleella i suoritetaan s_i operaatiota, kun $i \in [1 \ldots t]$. Tällöin $\sum_{i=1}^{t} s_i = q$. Kun kunkin askeleen operaatiot jaetaan tasan p:lle prosessorille, voidaan askelta i simuloida ajassa $\lceil s_i/p \rceil$. Tällöin laskenta C voidaan suorittaa p:llä prosessorilla ajassa

$$\sum_{i=1}^{t} \lceil s_i/p \rceil \leq \sum_{i=1}^{t} (s_i/p + 1) = t + 1/p \sum_{i=1}^{t} s_i = t + q/p.$$

□

Edellisen alakohdan lohkomistekniikkaa käyttävän algoritmin tarvitsema aika on $O(n/p + \log p)$. Samaan tulokseen päästään kuvan 9.6 rekursiivisella algoritmilla. Algoritmin toimintaa on havainnollistettu taulukossa 9.1.

Algoritmin suoritusaikaa kuvaa differenssiyhtälö

$$T(n) = \begin{cases} 1, & \text{kun } n = 1, \\ T(\lfloor n/2 \rfloor) + O(1), & \text{kun } n > 1. \end{cases}$$

Ratkaisuksi saadaan $T(n) = O(\log n)$. Algoritmin suorittamien operaatioiden lukumäärää puolestaan kuvaa differenssiyhtälö

$$S(n) = \begin{cases} 1, & \text{kun } n = 1, \\ S(\lfloor n/2 \rfloor) + O(n), & \text{kun } n > 1, \end{cases}$$

jonka ratkaisu on $S(n) = O(n)$. Brentin lauseen nojalla algoritmi toimii $O(\log n)$ ajassa, vaikka käytössä olisi vain $n/\log n$ prosessoria. Alkusummat voidaan siis määrätä taulukkomuodossa kustannusoptimaalisesti ajassa $O(\log n)$ käyttäen EREW PRAM -mallia.

```
      function alkusummat(A)
      {syöte taulukossa A, aputaulukot X, Y ja ratkaisu lasketaan taulukkoon S.}
 (1)  n := |A|
 (2)  if n = 1 then S[1] := A[1] end
 (3)  else
 (4)       for i ∈ [1..⌈n/2⌉] pardo X[i] := A[2i − 1] + A[2i] od
 (5)        Y=alkusummat(X)
 (6)       for i ∈ [1..⌈n/2⌉] pardo
 (7)            S[2i − 1] := Y[i]
 (8)            S[2i] := Y[i] + A[2i]
 (9)        od
(10)  end
```

Kuva 9.6: Alkusummien laskeminen rekursiivisesti.

A	5	2	7	1	1	3	2	3
X	7	8	4	5				
Y	7	15	19	24				
S	5	7	14	15	16	19	21	24

Taulukko 9.1: Esimerkki rekursiivisen alkusumma-algoritmin toiminnasta.

9.3. Perusalgoritmeja

9.3.1. Listan rankkaus

Listan rankkaus on tärkeä apualgoritmi, jota tarvitaan monissa sovelluksissa. Ongelman syötteenä on sellainen taulukkoon tallennettu linkitetty lista, jossa jokaiseen alkioon on liitetty osoite sen seuraajaan. Tehtävänä on laskea jokaisen alkion seuraajien lukumäärä. Oletetaan, että lista on tallennettu muistipaikkoihin $L[i]$, missä $1 \leq i \leq n$, ja alkioiden järjestys saadaan taulukosta S, ts. $S[i]$ sisältää listan L alkiota $L[i]$ seuraavan alkion indeksin. Listan rankkaus ratkeaa loppusummat-ongelman erikoistapauksena. Loppusummat-ongelmassa syöte on samanlainen kuin listan rankkauksessa, mutta tehtävänä on laskea

```
(1) for i ∈ [1...n] pardo R[i] := L[i] od
(2)       for k := 1 to ⌈log n⌉ do
(3)             for i ∈ [1...n] pardo
(4)                   if S[i] ≠ 0 then
(5)                         R[i] := R[i] + R[S[i]]
(6)                         S[i] := S[S[i]]
(7)                    end
(8)  od od
```

Kuva 9.7: Listan loppusummien laskenta.

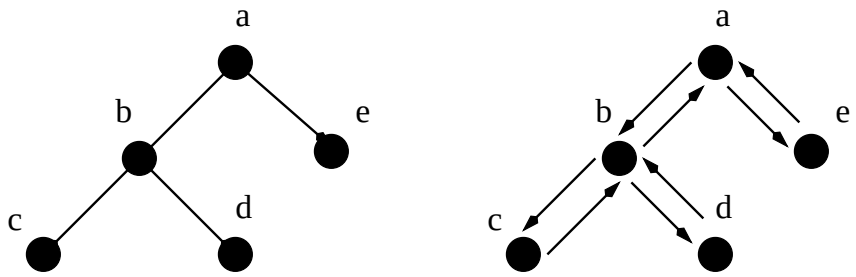
alkioita listassa seuraavien alkioiden summat. Loppusummat-ongelmalle saadaan kuvan 9.7 algoritmi.

Kun listan L alkusisältönä on pelkkiä ykkösiä, saadaan ratkaistua listan rankkaus-ongelma. Algoritmin suoritusaika on $O(\log n)$, koska jokainen for-lause voidaan suorittaa vakioajassa. Prosessoreja tarvitaan n kappaletta, joten algoritmin kustannus on $O(n \log n)$.

9.3.2. Eulerin silmukka -tekniikka

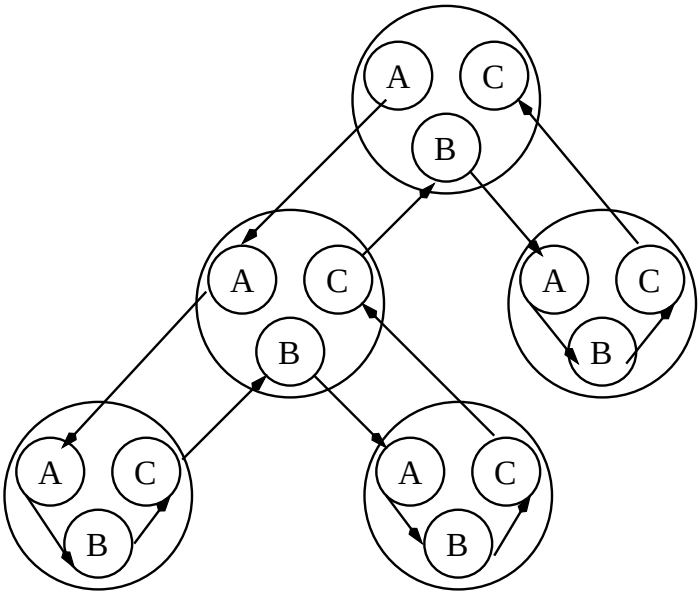
Monet puualgoritmit voidaan muuntaa rinnakkaisiksi käsittelemällä jokaista puun tasoa rinnakkain kuten maksimin etsinnässä turnaustekniikalla. Tällaisen algoritmin vaatima aika riippuu lineaarisesti puun korkeudesta. Jos puu on lähes tasapainoinen ja sen korkeus on $O(\log n)$, kun n on puun solmujen lukumäärä, algoritmi toimii hyvin. Mikäli puu ei ole tasapainoinen, tarvitaan toisenlaista ratkaisua.

Olkoon T puu. Esitetään T suunnattuna graafina T' kahdentamalla T:n kaaret (kuva 9.8) niin, että jokaista T:n kaarta (a, b) vastaa T':ssa suunnatut kaaret (a, b) ja (b, a).



Kuva 9.8: Kaarien kahdentaminen.

Graafin T' Eulerin silmukka on helppo muodostaa peräkkäisalgoritmilla. Graafi käydään läpi syvyyssuuntaisella etsinnällä, jossa peruutus tapahtuu käännettyjä kaaria pitkin. Tämä lähestymistapa voidaan muuntaa rinnakkaiseksi. Rinnakkaisalgoritmilla silmukkaa muodostettaessa voidaan toimia esimerkiksi seuraavasti. Otetaan jokaista puun solmua kohti kolme prosessoria, jotka nimetään prosessoreiksi A, B ja C. Jos solmulla on vasen lapsi, niin solmun A-prosessori osoittaa vasemman lapsen A-prosessoria; muutoin A-

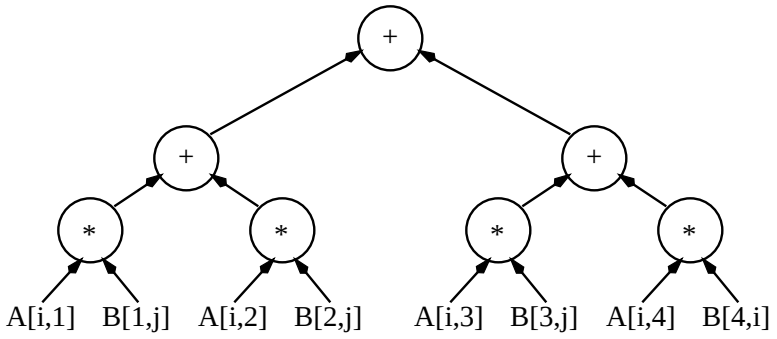


Kuva 9.9: Eulerin silmukka.

prosessori osoittaa saman solmun B-prosessoria. Jos solmulla on oikea lapsi, niin solmun B-prosessori osoittaa oikean lapsen A-prosessoria; muutoin B-prosessori osoittaa saman solmun C-prosessoria. Jos solmu on vasen lapsi, niin sen C-prosessori osoittaa vanhempansa B-prosessoria. Jos taas solmu on oikea lapsi, niin sen C-prosessori osoittaa vanhempansa C-prosessoria. Juuren C-prosessori ei osoita minnekään. Näin saadaan lista, jonka ensimmäinen alkio on juuren A-prosessori ja viimeinen alkio juuren C-prosessori (kuva 9.9). Jatkossa oletetaan, että Eulerin silmukasta on poistettu solmujen sisäiset kaaret.

Muodostettua silmukkaa voidaan käyttää puuongelmien ratkaisemisessa. Tarkastellaan kahta esimerkkiä: solmujen esijärjestyksen määräämistä ja alipuun koon laskemista. Olkoon (i, j) Eulerin silmukan kaari. Kaarta (i, j) sanotaan *eteneväksi* kaareksi, jos solmu i on lähempänä juurta kuin solmu j. Jos (i, j) on etenevä kaari, niin (j, i) on takautuva kaari. Muodostetaan listan rankkaus -algoritmilla kaarten rankkausnumerot $R(i, j)$. Helposti nähdään, että (i, j) on etenevä kaari, kun $R(i, j) > R(j, i)$. Koska kaaren (i, j) kaksi ilmentymää on yhdistetty osoittimilla toisiinsa, voidaan helposti selvittää, kumpi on etenevä kaari. Kun merkitään $f(i, j)$:llä kaarta (i, j) seuraavien etenevien kaarten lukumäärää Eulerin silmukassa, solmun j järjestysnumero esijärjestyksessä on $n - f(i, j)$, kun n on solmujen lukumäärä ja (i, j) solmuun j johtava etenevä kaari. Kaarten f-arvot voidaan laskea rinnakkain esimerkiksi loppusummat-algoritmilla.

Alipuiden koot saadaan myös määrättyä f-arvojen avulla. Voidaan nimittäin osoittaa, että solmun j jälkeläisten määrä on $f(i, j) - f(j, i)$, joten ongelma ratkeaa esijärjestyksen määräämisen sivutuotteena. Siis sekä esijärjestys että alipuiden koot voidaan määrätä $O(\log n)$ ajassa EREW PRAM -mallissa.



Kuva 9.10: Matriisien kertolasku.

9.3.3. Lajittelu taulukossa

Otetaan tehtäväksi lajitella n lukua n:llä prosessorilla, jotka ovat ketjussa, so. prosessori P_i on kytketty prosessoreihin P_{i-1} ja P_{i+1}. Tavoitteena on tilanne, jossa pienin lajiteltavista luvuista on prosessorilla P_1, toiseksi pienin prosessorilla P_2 jne. Alussa kullakin prosessorilla on oma syötelukunsa. Algoritmi toistaa vuorotellen askelia odd ja even, kunnes koko aineisto on lajiteltu:

- odd: P_{2i-1} ja P_{2i} vertaavat lukujaan ja vaihtavat tarvittaessa, kaikilla $i : 1 \leq 2i \leq n$.
- even: P_{2i} ja P_{2i+1} vertaavat lukujaan ja vaihtavat tarvittaessa, kaikilla $i : 1 \leq 2i < n$.

Helposti nähdään, että pahimmassa tapauksessa tarvitaan yhteensä $n - 1$ (odd- tai even-) askelta, sillä pienin luku voi alkutilanteessa olla prosessorilla P_n. Algoritmin kustannus on $O(n^2)$, joten algoritmi ei ole kustannusoptimaalinen.

Ratkaisu voidaan yleistää niin, että kukin prosessori hallitsee k lukua. Tarkastellaan tilannetta, jossa prosessoreita on vain kaksi. Pahimmassa tapauksessa pitää suorittaa $2k$ siirtoa prosessorilta toiselle. Toimintaa voidaan tehostaa seuraavasti: P_1 lähettää suurimman alkionsa P_2:lle ja P_2 lähettää pienimmän alkionsa P_1:lle. Näin jatketaan, kunnes P_1:n suurin alkio on pienempi kuin P_2:n pienin alkio.

9.3.4. Matriisien kertolasku

Tarkastellaan tavallisten neliömatriisien kertolaskua. Jos kerrottavina ovat $n \times n$ -matriisit A ja B, niin tulomatriisin C alkiot ovat muotoa $C[i,j] = \sum_{k=1}^{n} A[i,k]B[k,j]$. Jos jokaisen tuloalkion laskemiseksi on käytössä n prosessoria, niin alkiot voidaan laskea puuksi järjestettyjen prosessorien avulla. Kuvassa 9.10 on esimerkki tilanteesta, jossa prosessoreita on käytössä neljä kappaletta.

Syötteet annetaan lehtisolmuille, jotka suorittavat kertolaskun ja välittävät tulokset ylöspäin puussa. Sisäsolmut tekevät yhteenlaskuja ja lopulta juureen tulee alkion $C[i,j]$ arvo. Jos käytössä on n^3 prosessoria, niin jokainen tulomatriisin C alkio voidaan laskea omalla prosessorien puulla. Aikavaatimus on tällöin $O(\log n)$. Algoritmi ei ole kustannusoptimaalinen.