Tietojenkäsittelytiede **30** Heinäkuu 2010 sivut 33–55
Toimittaja: Jorma Tarhio

Kuinka kiltti mutta tarpeeton pikku algoritmi sai tärkeän tehtävän

Antti Valmari
Tampereen teknillinen yliopisto
Ohjelmistotekniikan laitos
Antti.Valmari@tut.fi

Tiivistelmä

Jos taulukon lokeroista yli puolet sisältää saman arvon, sanotaan sitä enemmistöarvoksi. Enemmistöarvon löytämiseen on olemassa monessa suhteessa esimerkillinen algoritmi: erittäin pieni, elegantti, nopea ja muistin käytöltään säästeliäs. Sen ainoa, mutta sitäkin merkittävämpi, puute on, että sille on melkein mahdoton keksiä järkevää käyttöä. Melkein aina joko tarvitaan tieto eniten esiintyvästä arvosta siinäkin tapauksessa, että se ei ole enemmistönä, ja sitä algoritmi ei tuota; tai eri arvoja on vain kaksi, jolloin tehtävä ratkeaa helpommalla laskemalla jomman kumman arvon esiintymiskerrat. Tästä syystä kirjoittaja on pitkään pitänyt algoritmia mielenkiintoisena mutta käytännössä hyödyttömänä akateemisena kuriositeettina. Yllättäen kirjoittajalle tuli vastaan tarve, johon algoritmi on juuri sopiva. Tämä kirjoitus alkaa algoritmin esittelyllä. Sitten esitellään joukko toistensa varaan rakentuvia algoritmeja, joista viimeinen käyttää enemmistöarvon löytämisalgoritmia apunaan. Matkan varrella kohdataan useita kiehtovia algoritmisia ideoita. Kerrotaan myös, miksi enemmistöarvon löytämisalgoritmi on erityisen sopiva tehtäväänsä viimeisessä algoritmissa.

1 Johdanto

Tämän kirjoituksen päätavoitteena ei ole esitellä uusia tieteellisiä tuloksia, vaikka sitäkin tässä kirjoituksessa tehdään. Algoritmien tutkimus on tuottanut monia yllättäviä ja joskus suorastaan nerokkaita tuloksia, joiden oivaltaminen tuottaa mielihyvää. Tämän kirjoituksen käsittelemien asioiden tutkiminen tuotti kirjoittajalle tällaista mielihyvää tavallista enemmän. Kirjoituksen päätavoitteena on välittää tätä mielihyvää lukijoille.

Kirjoitus on tarkoitettu laajalle lukijakunnalle ohjelmoimaan juuri oppineista algoritmeja hyvin tunteviin. Siksi siinä on vaikeustasoltaan vaihtelevia osia. Osa vaikeimmista asioista on ladottu lauseiden todistuksiksi, joten lukijan on helppo hypätä niiden yli niin halutessaan. Todistuksia ei kuitenkaan kannata hypätä yli rutiininomaisesti, sillä osa kiehtovista ajatuksista on esitetty vain niissä. O- ja Θ-merkintöjen tuntemisesta on apua, mutta välttämätöntä se ei liene.

Tieteellisten kirjoitusten tyyliin kuuluu, että lukija pidetään koko ajan tietoisena siitä, mihin tähdätään. Salapoliisikertomukseen sovellettuna se tarkoittaisi, että syyllinen paljastetaan heti alussa ja loppuosa kirjasta perustelee, miksi juuri hän

on syyllinen. Tämän kirjoituksen tavoitteeseen sellainen sopii huonosti. Siksi tässä kirjoituksessa enimmäkseen annetaan vain vihjeitä siitä, minne ollaan menossa. Tavallisen tasoista ennakkotietoa kaipaava voi aloittaa lukemalla luvun 10, jossa kirjoituksen pääasiat on koottu yhteen.

Tässä kirjoituksessa on myös tavallista tieteellistä kirjoitusta enemmän pohdittu, miksi ensimmäisenä mieleen tuleva tapa tehdä puheena oleva asia ei ole hyvä. Jollei tiedä, että suoraviivainen tapa tehdä jokin asia ei toimi, niin yllättävä tapa saattaa tuntua omituiselta kikkailulta. Jos on tietoinen asiaan liittyvistä ongelmista, niin yllättävän tavan nerokkuus on helpompi huomata.

Aloittakaamme.

2 Enemmistöarvon löytämisalgoritmi

Enemmistöarvon löytämisalgoritmi ratkaisee seuraavan tehtävän. On annettu arvot $A[1]$, $A[2]$, …, $A[n]$, missä $n \geq 1$. Jos jokin arvo esiintyy aineistossa yli $n/2$ kertaa, niin tehtävänä on ilmoittaa tämä arvo. Jos mikään arvo ei esiinny aineistossa yli $n/2$ kertaa, niin algoritmi saa palauttaa minkä tahansa arvon, joka esiintyy aineistossa. Jos enemmistöarvo on olemassa, on se yksikäsitteinen.

Algoritmin palauttamasta arvosta ei voi välittömästi nähdä, onko se enemmistöarvo vai onko niin, että millään arvolla ei ole yksinkertaista enemmistöä. Tämän voi kuitenkin selvittää helposti ja nopeasti laskemalla, montako kertaa palautettu arvo esiintyy aineistossa.

Tällaisen algoritmin suunnitteleminen on hauska tehtävä niille, jotka pitävät pikkunäppärien algoritmien suunnittelemisesta eivätkä tunne ratkaisua ennalta. Jos kuulut heihin, älä lue kohta tulevien kolmen tähden ohi ennen kuin olet yrittänyt! Paras ratkaisu on niin nopea, että aineistoa ei ehdi laittaa suuruusjärjestykseen siinä ajassa.

Tehtävään ei kannata suhtautua pelkkänä akateemisena kuriositeettina. Marraskuussa 2009 julkaistiin uutispalstalla `comp.theory` otsikolla ”Math/CompSci Interview Question — Thoughts?” viesti, jossa ohjelmistotuotannon alan työpaikkaa isosta monikansallisesta yrityksestä hakenut kertoi saaneensa työhönottohaastattelussa ratkaistavakseen seuraavan tehtävän (suomennos minun):

> On annettu 1 000 000 32-bittistä kokonaislukua sisältävä taulukko. Yksi kokonaislukuarvo x esiintyy siinä ainakin 500 001 kertaa. Laadi algoritmi x:n löytämiseksi. Tehokkuudesta saa lisäpisteitä.

Viestistä virinneessä keskustelussa esitettiin muutama erilainen ratkaisu ja pohdittiin, mihin haastattelija pyrki kysymyksellään.

Netti ei olisi netti, ellei siellä olisi myös sivua, johon on koottu vastauksia työpaikkahaastatteluissa esiintyneisiin kysymyksiin. Enemmistöarvon löytäminen löytyy sieltäkin [7].

* * *

Parhaan tunnetun enemmistöarvon löytämisalgoritmin keksivät Robert S. Boyer ja J Strother Moore[1] 1980. He saivat kirjoituksensa julkaistua vasta 1991 [4], mutta tieto heidän keksinnöstään levisi sitä ennen ja löytyi oppikirjasta jo 1986 [2].

Algoritmi on esitetty kuvassa 1. Se käy aineiston kerran läpi ylläpitäen arvausta ja laskuria. Ensimmäiseksi arvaukseksi otetaan aineiston ensimmäinen arvo. Jos vastaan tuleva arvo on sama kuin arvaus, niin laskuria kasvatetaan yhdellä. Muussa tapauksessa laskuria vähennetään yhdellä, paitsi sen ollessa nolla sitä ei vähennetä,

[1] J:n perässä ei ole pistettä, koska J ei ole lyhenne vaan ensimmäinen etunimi kokonaisuudessaan.

```
laskuri := 0
for i := 1 to n do
    if laskuri = 0 then arvaus := A[i]; laskuri := 1
    else if A[i] = arvaus then laskuri := laskuri + 1
    else laskuri := laskuri − 1
```

Kuva 1: Enemmistöarvon löytämisalgoritmi.

vaan vastaan tuleva arvo otetaan uudeksi arvaukseksi ja laskuri kasvatetaan ykköseen.

Algoritmin toiminnan oikeellisuus ei ole aivan itsestään selvä, joten se ansaitsee tulla todistetuksi.

Lause 1 Olkoon a muuttujan *arvaus* arvo kuvan 1 algoritmin lopetettua. Joko a esiintyy taulukossa A yli $n/2$ kertaa tai mikään arvo ei esiinny taulukossa A yli $n/2$ kertaa. Jos $n \geq 1$, niin a esiintyy taulukossa A.

Todistus. Koska laskuria ei vähennetä sen ollessa nolla, on laskurin arvo aina nolla tai positiivinen.

Tarkastellaan **for**-silmukan jokaisen kierroksen alussa funktiota $f(x)$, joka määritellään seuraavasti:

$$f(x) = \begin{cases} \mathit{laskuri}, & \text{jos } x = \mathit{arvaus} \\ -\mathit{laskuri}, & \text{muutoin.} \end{cases}$$

Ensimmäisellä kerralla *arvaus* on määrittelemätön, mutta se ei haittaa, koska silloin $\mathit{laskuri} = 0$ ja funktion määritelmän molemmat vaihtoehdot tuottavat jokaisella x saman tuloksen 0.

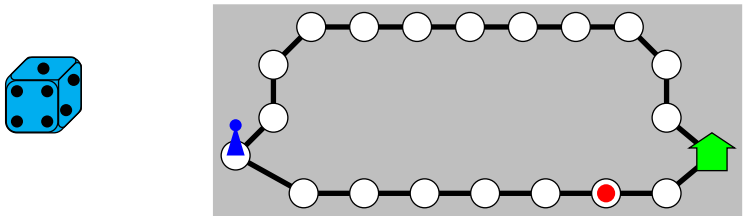
Jos suoritetaan ensimmäinen **then**-haara, niin $f(A[i])$ muuttuu arvosta 0 arvoon 1 ja muilla x $f(x)$ muuttuu arvosta 0 arvoon -1. Keskimmäisen haaran tapauksessa $f(A[i])$ kasvaa yhdellä ja muut $f(x)$ vähenevät yhdellä. Myös viimeisessä haarassa $f(A[i])$ kasvaa yhdellä, koska silloin $A[i] \neq \mathit{arvaus}$ ja $f(A[i]) = -\mathit{laskuri}$. Lisäksi $f(\mathit{arvaus})$ vähenee ja muut $f(x)$ kasvavat yhdellä.

Yhteenvetona saadaan, että jos $A[i] = x$, niin $f(x)$ kasvaa yhdellä riippumatta siitä, mikä **if**-lauseen kolmesta haarasta suoritetaan; ja jos $A[i] \neq x$, niin $f(x)$:n arvo voi kasvaa yhdellä tai vähentyä yhdellä. Jos x on enemmistöarvo, niin kasvukertoja ovat ainakin ne yli $n/2$ kertaa, jolloin $A[i] = x$. Kasvukertoja on siis enemmän kuin vähenemiskertoja. Näin ollen, ja koska $f(x)$ aloittaa nollasta, on algoritmin lopussa $f(x) > 0$. Jos silloin olisi $f(x) = -\mathit{laskuri}$, niin olisi $\mathit{laskuri} < 0$. Mutta näin ei voi olla, koska aikaisemmin todettiin, että aina $\mathit{laskuri} \geq 0$. Jäljelle jää vain f:n määritelmän vaihtoehto $f(x) = \mathit{laskuri}$ ja $x = \mathit{arvaus}$.

Olemme osoittaneet, että jos x on enemmistöarvo, niin algoritmin lopussa $\mathit{arvaus} = x$. Lisäksi algoritmilta vaadittiin, että jos enemmistöarvoa ei ole ja $n \geq 1$, niin lopussa *arvaus* on jokin aineistossa esiintyvä arvo. Tämä on helppo nähdä todeksi kuvasta 1. □

Enemmistöarvon löytämisalgoritmi ei tee muuta kuin selaa aineiston läpi täsmälleen kerran ja jokaisen arvon kohdalla se tekee kolme yksinkertaista toimintoa. Apumuistia se tarvitsee kaksi kokonaislukumuuttujaa vastauksen esittämiseen käytettävän muuttujan lisäksi. Se on siis melkein niin yksinkertainen, nopea ja muistipihi kuin taulukkoa käsittelevä algoritmi voi ylipäätään olla. Silti se ratkaisee haastavan tehtävän. Näissä suhteissa se on algoritmisuunnittelijan unelma.

Enemmistöarvon löytämisalgoritmista ei välittömästi näe, että se tuottaa tulokseksi sen mitä pitää. Tämä ei ole pelkäs-



Kuva 2: Noppapeli, jossa on pitkä turvallinen sekä lyhyt vaarallinen reitti.

tään myönteinen asia, mutta toisaalta se on merkki siitä, että algoritmi ei ole itsestään selvä. Koska se on yksinkertainen ja tehokas mutta ei itsestään selvä, saavat jotkut tutkijat, kuten minä, siitä älyllistä mielihyvää ja kutsuvat sitä elegantiksi.

Näin ollen sitä on käytetty esimerkkinä oppikirjoissa [2, kohta 4.3.3], [11, luku 18]. Se on eräs ensimmäisiä koneellisesti oikeaksi todistettuja algoritmeja, sillä sen keksijät olivat myös erään kuuluisan varhaisen automaattisen teoreemantodistimen laatijoita ja todistivat sillä algoritminsa Fortran-kielisen toteutuksen oikeaksi vuonna 1980 [4]. Algoritmia on käytetty esimerkkinä jopa Helsingin yliopistossa tehdyssä filosofian väitöskirjassa [12]. Ja, kuten edellä todettiin, enemmistöarvon löytämistehtävällä on testattu työnhakijoita.

Kun näiden kehujen jälkeen aletaan miettiä, mihin enemmistöarvon löytämisalgoritmia voitaisiin käytännössä hyödyntää, niin tunnelma lässähtää kuin pannukakku. Tehtävän englanninkielinen nimi ”majority vote/voting” luo mielikuvan vaaleista. Algoritmin tuottama informaatio ei kuitenkaan riitä todellisissa vaaleissa, vaan tarvitaan myös kunkin ehdokkaan äänimäärä, ja ehdokkaat on asetettava niiden mukaan järjestykseen. Nämä on helppo laskea muilla keinoin, jos ehdokkaita on ennalta tunnettu pienehkö joukko. Niinhän todellisissa vaaleissa on.

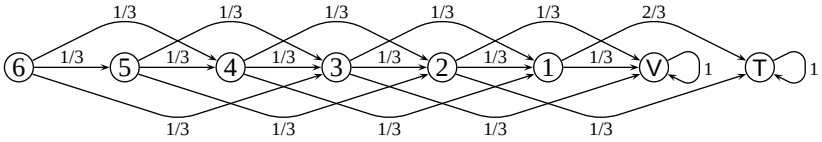
Laskenta muuttuu millään lailla vaikeaksi vasta kun aineisto on poimittu suuresta perusjoukosta. Silloinkaan ei yleensä riitä löytää enemmistöarvoa, vaan halutaan tietää eniten ääniä saanut arvo siinäkin tapauksessa, että se ei yksinään muodosta enemmistöä. Toisaalta, jos aineisto järjestetään suuruusjärjestykseen, niin on helppo selvittää kunkin arvon esiintymiskertojen määrä laskemalla monestiko se toistuu peräkkäin. Vaikka järjestäminen on hitaampaa kuin enemmistöarvon löytämisalgoritmin suoritus, on se riittävän nopeaa useimpiin käytännön tarpeisiin.

Näistä syistä enemmistöarvon löytämisalgoritmille on jokseenkin mahdoton löytää hyödyllistä käyttöä. Niin ainakin uskoin 21. elokuuta 2009 asti.

3 Markovin ketjut

Noppapelissä syntyy silloin tällöin kuvan 2 kaltainen tilanne. Nykyisestä paikasta maaliin on kaksi vaihtoehtoista reittiä: pitkä mutta turvallinen sekä lyhyt, jossa on vaarallinen paikka, johon pelaaja ei halua osua.

Voidakseen tehdä viisaan valinnan pelaaja haluaa tietää, millä todennäköisyydellä hän osuu vaaralliseen paikkaan, jos hän valitsee lyhyen reitin. Jos pelaaja on esimerkiksi neljän askeleen päässä vaarallisesta paikasta, niin seuraavalla heitolla hän pääsee sen ohi todennäköisyydellä $\frac{1}{3}$ (silmäluvut 5 ja 6). Todennäköisyydellä $\frac{1}{6}$ hän osuu siihen seuraavaksi (silmäluku 4). Muutoin hän jää yhden, kahden tai kol-



Kuva 3: Yksinkertaistettua noppapelin tilannetta vastaava Markovin ketju.

men askeleen päähän siitä, missä kunkin vaihtoehdon todennäköisyys on $\frac{1}{6}$.

Olkoon vaaralliseen paikkaan n askelta, ja tarkoittakoon p_n todennäköisyyttä osua siihen ennemmin tai myöhemmin. Kun $n < 0$, on vaarallinen paikka ohitettu, joten $p_n = 0$. Kun $n = 0$, ollaan vaarallisessa paikassa, joten $p_0 = 1$. Kun $n > 0$, niin p_n lasketaan summana niistä kuudesta tapauksesta, jotka vastaavat seuraavan nopanheiton silmälukuja 1, 2, ..., 6:

$$p_n = \frac{1}{6}\sum_{i=1}^{6} p_{n-i} \quad .$$

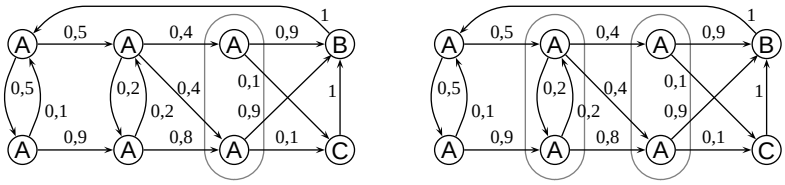
Nämä säännöt voi esittää kuvan 3 kaltaisella piirroksella. Oikein piirrettynä siitä tulisi kamalan sotkuinen, joten sitä on yksinkertaistettu kuvittelemalla, että arpanopassa olisi vain kolme tahkoa silmäluvuiltaan 1, 2 ja 3. Piirrosta on yksinkertaistettu myös esittämällä negatiiviset askelmäärät (eli $n < 0$) yhdellä ympyrällä, johon on kirjoitettu ”T” tarkoittamaan turvallista paikkaa. Vaarallinen paikka on merkitty kirjaimella ”V”, ja ympyröihin kirjoitetut numerot tarkoittavat etäisyyttä vaaralliseen paikkaan. Kaarten varrelle kirjoitetut luvut ilmaisevat siirtymien todennäköisyydet. Jos todennäköisyys on nolla, niin vastaava kaari jätetään piirtämättä.

Meillä on siis järjestelmä, jossa on *tiloja* (kuvan ympyrät) sekä tunnetulla todennäköisyydellä tapahtuvia *siirtymiä* tilasta toiseen (kuvan nuolet). Tällaista järjestelmää sanotaan (diskreetin ajan) *Markovin ketjuksi* venäläisen matemaatikon Andrei Markovin mukaan, joka eli 1856–1922 [3, s. 873]. On olemassa myös jatkuvan ajan Markovin ketjuja. Ne ovat hankalampia selittää, mutta onneksi tässä kirjoituksessa tarvitaan niistä vain kahta tietoa: aivan kohta käsiteltävä samanveroisten tilojen yhdistäminen koskee niitäkin, ja niiden siirtymiin merkityt luvut voivat saada myös negatiivisia arvoja.

Markovin ketjuista voi ratkaista numeerisesti monenlaisia asioita koskien tutkittavaa järjestelmää. Isojen Markovin ketjujen käsittely on valitettavasti työlästä. Tästä syystä on kehitetty keinoja pienentää Markovin ketjuja tunnistamalla tietyssä mielessä samanveroisia tiloja ja sulauttamalla ne yhteen. Kuvan 3 tila T on niiden tilojen yhteensulautuma, jotka vastaavat etäisyyden negatiivisia arvoja.

Kuva 4 havainnollistaa tilojen yhdistämistä yleisemmin. Sen Markovin ketjussa on kolmenlaisia tiloja. Ne on merkitty kirjaimilla ”A”, ”B” ja ”C”. Ajatuksena on, että tilat on merkitty eri kirjaimilla jos ja vain jos niillä on jokin välittömästi merkityksellinen ero. Esimerkiksi noppapelissä vaarallinen tila on välittömästi erilainen kuin muut tilat: pelätty uhka on toteutunut, kun muissa tiloissa on ainakin toivoa, että se vältetään. Tilojen tällaista luokittelua kutsutaan *alkujaoksi*.

Kuvan 4 vasemmanpuoleisessa piirroksessa on huomattu, että kummastakin harmaalla viivalla ympyröidystä tilasta päästään seuraavaksi B-merkittyyn tilaan todennäköisyydellä 0,9 ja C-merkittyyn tilaan todennäköisyydellä 0,1, ja muualle kummastakaan ei pääse. Niillä on siis sa-



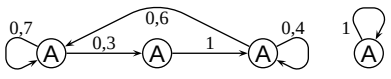
Kuva 4: Tilojen yhdistäminen Markovin ketjussa.

mat mahdolliset tulevaisuudet samoilla todennäköisyyksillä. Niillä ei myöskään ole välitöntä eroa, koska ne on merkitty samalla kirjaimella ”A”. Ne ovat siis kaikkien olennaisten asioiden kannalta samanveroiset, joten ne voi yhdistää yhdeksi tilaksi.

Kuvan oikeanpuoleisessa piirroksessa on kaksi muuta tilaa todettu samanveroisiksi. Niistä toisesta lähtee kaksi siirtymää todennäköisyydellä 0,4 ja toisesta yksi todennäköisyydellä 0,8, mutta se ei haittaa, koska ne vievät samanveroisiksi jo todettuihin tiloihin. Siksi kummastakin päästään samalla todennäköisyydellä 0,8 samanlaiseen vaihtoehtoisten tulevaisuuksien kokonaisuuteen. Lisäksi kummastakin pääsee toiseensa samalla todennäköisyydellä 0,2. Mitään eroa ei voi havaita heti eikä myöhemmin.

Sen sijaan vasemmanpuoleisia tiloja ei voi yhdistää. Ne eroavat toisistaan sen todennäköisyyden suhteen, jolla päästään äsken yhdistettyihin tiloihin. Toiselle niistä tämä todennäköisyys on 0,5 ja toiselle 0,9.

Edellä aloitimme alkuperäisellä Markovin ketjulla, tunnistimme samanveroisia tiloja ja yhdistimme ne. Tämä toimintaperiaate joutuu vaikeuksiin oheisen Markovin ketjun tapauksessa. Sen kaikki tilat voi yhdistää, mutta mistä nähdään, että niin on?



Tämän vuoksi tiloja yhdistävät algoritmit toimivat toisella tavalla. Luvussa 5 katsotaan, miten asia on hoidettu eräässä toisessa yhteydessä. Ennen sitä on tutustuttava erääseen tietorakenteeseen.

4 Hienonnettavan osituksen tietorakenne

Joukon *ositus* on sen jako osiksi siten, että jokainen alkio kuuluu tasan yhteen osaan, eikä mikään osa ole tyhjä. Ositus on sopiva käsite esittämään tilannetta, jossa käsiteltävät kohteet on jaettu ryhmiin jonkin yhdistävän tai erottavan tekijän mukaan.

Joukon $\{1,2,\ldots,n\}$ ositus voidaan esittää kahden taulukon avulla: *luku* ja *alku*. Samaan osaan kuuluvat luvut ovat peräkkäin taulukossa *luku*, ja *alku*$[i]$ ilmaisee kohdan, josta osan *i* luvut alkavat. Ositus $\{2,4\}$, $\{6,1,5,8\}$, $\{3\}$, $\{7,9\}$ voidaan esittää seuraavilla taulukoilla:

luku = [2, 4, 6, 1, 5, 8, 3, 7, 9]
alku = [1, 3, 7, 8]

Olettakaamme, että osa $\{6,1,5,8\}$ halutaan halkaista kahdeksi osaksi $\{1,5\}$ ja $\{6,8\}$. Tämä onnistuu ryhmittelemällä osan alue taulukossa *luku* uudelleen ja lisäämällä uusi osan alkukohta:

luku = [2, 4, 1, 5, 6, 8, 3, 7, 9]
alku = [1, 3, 5, 7, 8]

Tämän toteuttamisessa tehokkaasti on ongelma. Uuden osan alkukohta lisättiin keskelle taulukkoa *alku*, jolloin sen loppuosan sisältö [7, 8] jouduttiin siirtämään askeleen verran oikealle. Jos loppuosa on

iso, menee siirtämiseen paljon aikaa. Olisi nopeampaa lisätä uuden osan alkukohta taulukon *alku* loppuun seuraavasti:

luku	=	[2, 4, 1, 5, 6, 8, 3, 7, 9]
alku	=	[1, 3, 7, 8, 5]

Valitettavasti tämä tekee osien loppukohtien löytämisen hankalaksi. Aikaisemmin osan ℓ loppukohta voitiin selvittää katsomalla *alku*$[\ell + 1]$, paitsi viimeisen osan tapauksessa, mutta sen loppukohdaksi tiedetään koko taulukon loppukohta. Nyt osan 2 loppukohtaa ei voi löytää siten. Tämän ratkaisemiseksi otetaan käyttöön kolmas taulukko *loppu*, jonka sisältö ilmaisee osien loppukohdat. Loppukohdaksi voitaisiin antaa osan viimeisen luvun paikka, mutta myöhemmin esitettävien asioiden kannalta on luontevampaa, että loppukohdaksi annetaan sitä ykkösen verran suurempi luku.

luku	=	[2, 4, 1, 5, 6, 8, 3, 7, 9]
alku	=	[1, 3, 7, 8, 5]
loppu	=	[3, 5, 8, 10, 7]

Osan halkaisemiseksi on tavalla tai toisella ilmaistava, mitkä luvut menevät ensimmäiseen ja mitkä toiseen puolikkaaseen. Riittää kertoa ensimmäiseen puolikkaaseen menevät luvut. Otamme tätä varten käyttöön toiminnot *merkitse*(i) ja *halkaise*(ℓ), missä *i* on luku ja ℓ on osan numero. Aikaisemman esimerkin osa 2, joka sisältää $\{6, 1, 5, 8\}$, saadaan halkaistua suorittamalla *merkitse*(1), *merkitse*(5) ja *halkaise*(2).

Toiminnon *merkitse*(i) on löydettävä luku *i* taulukosta *luku*. Tätä varten otetaan käyttöön neljäs taulukko *paikka*. Jotenkin on pidettävä kirjaa, mitkä luvut on merkitty. Tämä voidaan tehdä jakamalla osa alkupuolikkaaseen, joka sisältää merkityt luvut ja loppupuolikkaaseen, joka sisältää muut. Niiden välisen rajan ilmaisee uusi taulukko *keski*. Viimeinen uusi taulukko on *osa*, joka ilmoittaa kullekin luvulle sen osan numeron, johon luku kuuluu.

Kuva 5 näyttää toimintojen toteutukset. Toiminnossa *merkitse* oleva testi estää merkitsemästä uudelleen lukua, joka on jo merkitty. Luku merkitään vaihtamalla sen paikkaa alkupuolikkaan ja loppupuolikkaan välisen rajan kohdalla olevan luvun kanssa ja kasvattamalla rajaa yhdellä. Lukujen uudet paikat täytyy päivittää taulukkoon *paikka*. Jos siinä osassa, johon luku *i* kuuluu, ei aikaisemmin ollut merkittyjä lukuja, niin *merkitse*(i) palauttaa sen osan numeron. Muussa tapauksessa se palauttaa nollan. Paluuarvosta tulee olemaan hyötyä jatkossa.

Merkitse ei sisällä silmukoita eikä kutsu muita toimintoja eikä itseään, joten se toimii vakioajassa.

Koska *merkitse* vaihtaa lukujen järjestystä, on tärkeää, että merkitsemisiä ei tehdä kesken osan sisällön selaamisen. Muutoin selaaminen saattaa tuottaa joitakin lukuja monesti ja jättää toisia tuottamatta. Tietorakenteen käyttäjien on otettava tämä huomioon.

Toiminnon *halkaise* kaksi ensimmäistä riviä huolehtii, että toiminto ei tee tyhjää osaa. Jos osan kaikki luvut on merkitty, sijoitus *keski*$[\ell]$:= *alku*$[\ell]$ muuttaa ne merkitsemättömiksi. *Halkaise* pyyhkii aina kaikki vanhat merkinnät, jotta ne eivät häiritsisi myöhempiä merkitsemisiä ja halkaisemisia.

Kolmas rivi kasvattaa osien määrää ja ottaa käyttöön uuden osanumeron *u*. Neljäs rivi valitsee pienemmän puolikkaan. Numero *u* annetaan sille, ja suurempi puolikas säilyttää vanhan numeron ℓ. Suurin osa jäljellä olevista riveistä päivittää osien alku-, keski- ja loppukohdat vastaamaan uutta ositusta. **For**-silmukka merkitsee uuteen osaan kuuluville luvuille, että ne kuuluvat siihen.

Halkaise palauttaa joko toisen puolik-

```
merkitse(i)
ℓ := osa[i];  k := keski[ℓ]
p := paikka[i]
if p ≥ k then
    luku[p] := luku[k]
    paikka[ luku[p] ] := p
    luku[k] := i
    paikka[i] := k
    keski[ℓ] := k + 1
    if k = alku[ℓ] then return ℓ
return 0
```

```
halkaise(ℓ)
if keski[ℓ] = loppu[ℓ] then keski[ℓ] := alku[ℓ]
if keski[ℓ] = alku[ℓ] then return 0
osia := osia + 1;  u := osia
if keski[ℓ] − alku[ℓ] ≤ loppu[ℓ] − keski[ℓ] then
    alku[u] := alku[ℓ];  loppu[u] := keski[ℓ]
    alku[ℓ] := keski[ℓ]
else
    alku[u] := keski[ℓ];  loppu[u] := loppu[ℓ]
    loppu[ℓ] := keski[ℓ];  keski[ℓ] := alku[ℓ];  ℓ := u
keski[u] := alku[u]
for p := alku[u] to loppu[u] − 1 do
    osa[ luku[p] ] := u
return ℓ
```

Kuva 5: *Merkitse*(i) ja *halkaise*(ℓ).

kaan numeron tai luvun 0 merkiksi siitä, että ei halkaistu, koska toisesta puolikkaasta olisi tullut tyhjä. Palautettu puolikkaan numero on sen puolikkaan numero, johon merkitsemättä jääneet luvut kuuluvat. Tästä paluuarvosta on hyötyä luvun 8 algoritmin toteutuksessa. Käyttäjä tietää molempien puolikkaiden numerot ilman paluuarvoakin, koska ne ovat kutsussa annettu numero ja *osia*, mutta ilman paluuarvoa hän ei tietäisi, kumpi puolikas sisältää ennen kutsua merkityt luvut.

Toiminnon *halkaise* suoritusaika on **for**-silmukan vuoksi suoraan verrannollinen pienemmän puolikkaan kokoon.

Halkaise olisi ollut yksinkertaisempi toteuttaa siten, että jos tehdään uusi osa, niin se tehdään aina merkityistä luvuista. Silloin suoritusaika olisi suoraan verrannollinen merkittyjen lukujen määrään eikä pienemmän puolikkaan kokoon. Tämä ei ole olennainen ero, koska merkitsemiset ja halkaiseminen vievät joka tapauksessa yhteensä aikaa vähintään verrannollisesti merkittävien lukujen määrään. Sen sijaan olennaista on, että kummallakaan toteutuksella aikaa ei kulu verrannollisesti isomman puolikkaan kokoon siinä tapauksessa, että se koostuu merkitsemättömistä luvuista, koska se voi olla olennaisesti suurempi kuin merkittyjen lukujen määrä.

Monimutkaisemman toteutuksen valitsemisen syy ei siis ole suoritusajassa. Todellinen syy on, että se mahdollistaa luvussa 5 kerrottavan parannuksen eräisiin algoritmeihin.

Sain suuren osan tämän tietorakenteen ideoista Timo Knuutilan kirjoituksesta [10].

5 Deterministisen äärellisten automaattien minimointi

Deterministinen äärellinen automaatti on yksi teoreettisen tietojenkäsittelytieteen tärkeimpiä peruskäsitteitä. Se muistuttaa edellä käsiteltyä Markovin ketjua, mutta siinä on muutamia olennaisia eroja. Siirtymiin ei ole liitetty todennäköisyyksiä vaan nimiä, jotka on poimittu joukosta, jota kutsutaan *aakkostoksi*. ”Deterministinen” tarkoittaa, että samasta tilasta ei voi olla useampia kuin yksi siirtymä samalla nimellä. Tiloista yksi on asetettu erikoisasemaan *alkutilana* ja nolla tai useampia tiloja on nimetty *lopputiloiksi*. Kuvassa 6 on esimerkki, jossa alkutila on osoitettu tyh-

jästä alkavalla nuolella ja lopputilat kaksinkertaisilla ympyröillä.

Siirtymien nimiä on mielekästä ajatella kirjaimina. Jokainen alkutilasta siirtymiä pitkin johonkin lopputilaan kulkeva reitti määrittelee sanan, joka saadaan luettelemalla siirtymien nimet peräkkäin siinä järjestyksessä kuin ne kohdattiin. Tällä tavalla jokainen deterministinen äärellinen automaatti määrittelee jonkin joukon sanoja.

Klassinen deterministisiin äärellisiin automaatteihin liittyvä tehtävä on löytää mahdollisimman pieni automaatti, joka määrittelee täsmälleen saman joukon sanoja kuin annettu automaatti. Sitä kutsutaan deterministisen äärellisen automaatin minimoimiseksi.

Sellaiset tilat ja siirtymät, joihin alkutilasta ei pääse siirtymiä pitkin, eivät vaikuta deterministisen äärellisen automaatin määrittelemien sanojen joukkoon. Minimoinnissa ne tulee poistaa. Sama pätee tiloihin ja siirtymiin, joista ei pääse mihinkään lopputilaan, paitsi alkutilaa ei saa poistaa. Poistaminen onnistuu tavallisilla graafihakualgoritmeilla. Poistovaihe on niin ilmeinen, että osa kirjoittajista ei edes mainitse sitä [8, 9].

Varsinainen haaste minimoinnissa on siihen sisältyvä tilojen yhdistämistehtävä. Se on melko samanlainen kuin edellä esitelty Markovin ketjun tilojen yhdistämistehtävä. John E. Hopcroftin tulos vuodelta 1971 [9] on keskeinen tällä alalla.[2] Siihen on hiljattain saatu parannuksia, joiden vuoksi asiaa kannattaa selostaa laajemmin kuin kiltin pikku algoritmin tarina välttämättä vaatisi.

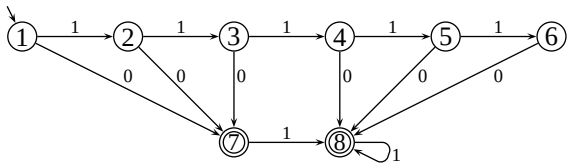
Hopcroftin minimointialgoritmi perustuu tilojen ryhmittelyyn epätyhjiksi lohkoiksi, joita halkaistaan tietyn säännön mukaan pienemmiksi lohkoiksi kunnes ne lakkaavat pilkkoutumasta. Hän käytti lohkojen esittämiseen kaksisuuntaisiin linkitettyihin listoihin perustuvaa tietorakennetta. Siihen voidaan yhtä hyvin, ellei paremminkin, käyttää luvussa 4 kuvattua tietorakennetta.

Aluksi on korkeintaan kaksi lohkoa: lopputilat ja muut tilat. Alkuperäisiä lohkoja on yksi siinä tapauksessa, että kaikki tilat ovat lopputiloja sekä siinä tapauksessa, että lopputiloja ei ole ollenkaan.

Tavoitteena on saavuttaa mahdollisimman vähällä pilkkomisella tilanne, jossa jokainen lohko on *yhteensopiva* jokaisen aakkosen *a* suhteen. Yhteensopivuus tarkoittaa, että lohkon tilat ovat keskenään samaa mieltä siitä, lähteekö niistä siirtymää, jonka nimi on *a*, ja jos lähtee, niin mihin lohkoon kuuluvaan tilaan se vie. Kun yhteensopivuus on saavutettu, saadaan minimoitu automaatti yhdistämällä kunkin lohkon tilat yhdeksi tilaksi. Halkaisemisia halutaan tehdä mahdollisimman vähän, jotta minimoidussa automaatissa olisi mahdollisimman vähän tiloja. Voidaan todistaa, että tällä tavalla saatu automaatti määrittelee saman joukon sanoja kuin alkuperäinen automaatti ja on pienin mahdollinen.

Tarkastellaan esimerkkinä kuvan 6 determinististä äärellistä automaattia. Alkuperäiset lohkot ovat $\{1,2,3,4,5,6\}$ ja $\{7,8\}$. Lohko $\{7,8\}$ on yhteensopiva sekä aakkosen 0 että aakkosen 1 suhteen. Lohko $\{1,2,3,4,5,6\}$ on yhteensopiva 0:n suhteen, mutta ei 1:n suhteen, koska tilasta 6 ei ole mutta muista on 1-siirtymä lohkoon itseensä. Niinpä $\{1,2,3,4,5,6\}$ on halkaistava osiksi $\{1,2,3,4,5\}$ ja $\{6\}$. Koska tila 6 muodostaa nyt oman lohkonsa, on $\{1,2,3,4,5\}$ yhteensopimaton 1:n suhteen, joten se on halkaistava lohkoiksi $\{1,2,3,4\}$ ja $\{5\}$. Samasta syystä tilat 4, 3 ja 2 on erotettava omiksi lohkoikseen

[2] On myös väitetty, että [9] olisi vuodelta 1970. Löysin sen skannattuna netistä ja sen kannessa lukee "January, 1971".



Kuva 6: Deterministinen äärellinen automaatti, jota ei kannata minimoida ”etuperin”.

kukin vuorollaan. Tilat 7 ja 8 pysyvät loppuun asti yhdessä.

Tuntuisi luonnolliselta halkaista lohko käymällä sen tilat läpi ja katsomalla, pääseekö niistä samalla aakkosella samaan lohkoon. Esimerkissä se tarkoittaa, että käydään läpi tilat 1, 2, 3, 4, 5 ja 6; vähän myöhemmin tilat 1, 2, 3, 4 ja 5; sitten 1, 2, 3 ja 4; ja niin edelleen. Jos ylärivissä on $n-2$ tilaa, käydään läpi $(n-2)+\ldots+2+1=\frac{1}{2}(n^2-3n+2)$ tilaa. Näin toimivan algoritmin ajan kulutuksen kertaluokka aakkoston koolla kaksi ei siis voi olla parempi kuin $O(n^2)$, missä n on tilojen määrä.

Hopcroft huomasi, että jos työ järjestetään toisin, voidaan ajan kulutuksen kertaluokaksi aakkoston koolla kaksi saada $O(n\log n)$. Se on selvästi edellistä parempi. Gries selkeytti Hopcroftin algoritmia ja laski, että jos aakkoston koko on α, on sen ajan kulutus $O(\alpha n\log n)$ [8]. Algoritmi jäi Griesinkin jäljiltä vaikeatajuiseksi ja Knuutila on selkeyttänyt sitä edelleen [10]. Tässä luvussa esitetään algoritmista muunnos, jonka ajan kulutus on parempi kuin minkään ennen vuotta 2008 esitetyn ja joka on luultavasti yksinkertaisempi kuin mikään koskaan aikaisemmin esitetty. Sen ajan kulutus on $O(m\log n)$, missä m on siirtymien määrä. Tämä on parannus, koska $m\leq\alpha n$ ja usein m on paljon pienempi kuin αn.

Griesin algoritmissa valitaan yksi lohko L ja aakkonen *a pilkkojaksi*. Jokaisen lohkon tilat jaetaan kahteen ryhmään: niihin, joista on a-niminen siirtymä L:ään kuuluvaan tilaan ja niihin, joista ei ole. Jos kumpikin ryhmä on epätyhjä, halkaistaan lohko ryhmien mukaisesti. Sitten valitaan toinen pilkkoja ja sama toistetaan. Tätä jatketaan, kunnes mikään lohko ei enää halkea pienemmiksi osiksi.

Pilkkojaa käytettäessä ei käydä läpi halkaistavan lohkon tiloja vaan pilkkojan lohkon tilat. Niihin saapuvat pilkkojan aakkosella nimetyt siirtymät kuljetaan takaperin. Siirtymien alkupäiden tilat merkitään luvun 4 toiminnolla *merkitse* tai jollakin sen tapaisella. Niiden lohkojen numerot, joiden tiloja merkittiin, kootaan johonkin tietorakenteeseen H. Luvussa 4 todettiin, että *merkitse* palauttaa lohkon numeron vain kun lohko kohdataan ensimmäisen kerran. Tämän ansiosta on helppo välttää saman numeron lisääminen H:hon moneen kertaan, ja H voidaan toteuttaa hyvin yksinkertaisesti esimerkiksi pinona.

Kun merkitsemisvaihe on valmis, halkaistaan ne lohkot, joiden numerot ovat H:ssa, toiminnolla *halkaise* tai vastaavalla, sikäli kuin ne halkeavat. Luvun 4 *halkaise* huolehtii, että jos lohkon kaikki tilat merkittiin, niin lohkoa ei halkaista. Tavoitellun nopeushyödyn kannalta on tärkeää, että *halkaise* ei käy läpi enempää tiloja kuin merkittiin. Luvussa 4 huolehdittiin siitäkin.

Edellä olleessa esimerkissä lohkon $\{1,2,3,4,5,6\}$ läpikäynti aakkosella 1 löytää tilat 1, 2, 3, 4 ja 5 mutta ei tilaa 6, joten lohko halkeaa osiksi $\{1,2,3,4,5\}$ ja $\{6\}$. Sen jälkeen lohkon $\{6\}$ läpikäynti aakkosella 1 löytää vain tilan 5, joka

näin ollen erotetaan omaksi lohkokseen. Lohkon $\{5\}$ ja aakkosen 1 käyttö pilkkojana erottaa tilan 4 omaksi lohkokseen ja niin edelleen, kunnes ylärivi on pilkkoutunut yhden tilan lohkoiksi. Ylärivi pilkkoutui erillisiksi tiloiksi käymällä läpi vain $2n-5$ tilaa.

Äskeisen esimerkin menestys on kiinni siitä, että isoja lohkoja, kuten $\{1,2,3,4,5\}$, ei alun jälkeen tarvinnut käyttää pilkkomiseen ennen kuin ne olivat itse pilkkoutuneet pieniksi. Oliko tämä vain hyvää tuuria? Ei ollut, vaan takana on yleinen periaate. Hopcroft huomasi, että jos lohkoa L on jo käytetty pilkkomiseen ja se jakaantuu lohkoiksi L_1 ja L_2, niin jatkossa riittää käyttää toista niistä pilkkomiseen.

Ilmiöstä saa käsityksen vertaamalla lohkojen $\{1,2,3,4,5\}$ ja $\{6\}$ halkaisemisvaikutuksia aakkosella 1. Lohko $\{6\}$ aiheuttaa tilan 5 merkitsemisen ja sitä kautta lohkon $\{1,2,3,4,5\}$ jakaantumisen osiksi $\{1,2,3,4\}$ ja $\{5\}$. Vastaavasti $\{1,2,3,4,5\}$ aiheuttaa tilojen 1, 2, 3 ja 4 merkitsemisen ja sitä kautta lohkon $\{1,2,3,4,5\}$ jakaantumisen osiksi $\{1,2,3,4\}$ ja $\{5\}$. Lopputulos on sama, vaikka merkityt tilat olivat erit.

On helppo arvata, että jos jatkossa käytettäväksi valitaan osista L_1 ja L_2 pienempi, niin saadaan säästöä. Säästön määrä voi kuitenkin olla yllätys. Jos samaa siirtymää käytetään halkaisemisessa monta kertaa, niin jokaisella kerralla sen loppupään tila kuuluu lohkoon, jonka koko on enintään puolet koosta edellisellä kerralla. Tästä seuraa, että samaa siirtymää voidaan käyttää enintään noin $\log_2 n$ kertaa. Koska siirtymiä on enintään αn kappaletta, on tästä aiheutuva työmäärä kaiken kaikkiaan $O(\alpha n \log n)$. Jos ei valittaisi pienempää osaa, niin siirtymän käyttökertojen määrä voisi olla lähes n ja työmäärä $\Theta(\alpha n^2)$.

Hopcroftilla oli jokaista aakkosta kohti lista niiden lohkojen numeroista, joita täytyy käyttää jatkossa pilkkojina kyseisen aakkosen kanssa. Aina kun lohko halkesi, selvitettiin kaksiosaisella testillä, kumman osan numero lisätään listaan. Myöhemmillä kirjoittajilla on ollut vaihtelevia tietorakenteita ja testejä samaan tarkoitukseen, myös muissa pilkkomissovelluksissa kuin deterministisen äärellisen automaatin minimoinnissa [1, 5, 6, 8, 10, 13, 14, 15, 16]. Minkäänlaista tietorakennetta ja siihen liittyviä testejä ei kuitenkaan tarvita! Tämä osoitetaan seuraavaksi ja se lienee ennen julkaisematon havainto.

Kutsuttakoon jatkossa käytettävien lohkojen, pilkkojien tms. numeroiden joukkoa työjoukoksi, ja puhukaamme yksinkertaisuuden vuoksi vain lohkoista. Jos haljenneen lohkon numero ei ollut työjoukossa, niin työjoukkoon täytyy lisätä pienemmän osan numero. Jos se oli työjoukossa, niin molempien osien numeroiden kuuluu päätyä työjoukkoon. Vanhan numeron perineen osan numero — eli vanha numero — on siellä jo, joten täytyy lisätä uuden osan numero. Hopcroft ja muut tarvitsivat testejä sen selvittämiseksi, lisätäänkö pienemmän osan vai uuden osan numero, ja kumpi osa on pienempi. Niin nytkin tarvittaisiin, jos luvun 4 *halkaise* antaisi uuden numeron aina merkityistä tiloista koostuvalle osalle. Mutta koska se antaa uuden numeron aina pienemmälle osalle, sulautuvat eri tapaukset yhdeksi, eikä testejä tarvita.

Uudeksi lohkonumeroksi on helpointa antaa yhtä suurempi kuin edellinen uusi numero. Lohkot saa ottaa pilkkomiskäyttöön työjoukosta missä järjestyksessä tahansa. Jos ne otetaan vanhin ensin, niin työjoukossa on aina peräkkäiset numerot alkaen ensimmäisestä käyttämättömän lohkon numerosta ja loppuen suurimpaan lohkon numeroon. Tällaisen joukon tallet-

tamiseen ei tarvita erityistä tietorakennetta, vaan riittää, että kumpikin numero on kokonaislukumuuttujassa.

Vielä on ratkaistava, miten löydetään tehokkaasti ne siirtymät, jotka kulloinkin on kuljettava takaperin. Tyypillinen keino on ollut liittää jokaiseen tilaan ja aakkoseen lista, joka sisältää siihen tilaan päättyvien, sillä aakkosella nimettyjen siirtymien alkupäiden tilat. Nämä listat on helppo muodostaa. Ne vievät $\Theta(\alpha n)$ muistia. Tämä on harmillista, sillä siirtymien todellinen määrä m on usein paljon pienempi kuin αn.

Vuonna 2008 julkaistiin, että muistin käyttö voidaan pudottaa kertaluokkaan $O(m+\alpha)$ ja ajan käyttö kertaluokkaan $O(m \log n)$ ottamalla käyttöön toinen kappale luvun 4 tietorakenteesta [16]. Siihen talletettava ositus muodostuu siirtymistä eikä tiloista. Kutsuttakoon sen osia *kimpuiksi* erotukseksi lohkoista. Lisäksi jokaisella tilalla on kaikkien siihen saapuvien siirtymien numeroiden lista. Se saa olla missä järjestyksessä tahansa, joten se on helppo toteuttaa.

Aluksi kimput muodostetaan siten, että kaikki samannimiset siirtymät kuuluvat samaan kimppuun ja erinimiset eri kimppuihin. Tämä on vaikeampaa kuin voisi luulla, sillä järjestämiseen nimien mukaan menee tavallisilla algoritmeilla $O(m \log m)$ aikaa, joka on enemmän kuin luvattu kokonaissuoritusaika $O(m \log n)$. Koska tämän eron käytännön merkitys on pieni ja nopeampi osituskeino on varsin eksoottinen, emme esitä sitä tässä. Se löytyy julkaisusta [16].

Algoritmin edetessä kimppuja pilkotaan siten, että siirtymät, joiden loppupään tilat kuuluvat eri lohkoihin, joutuvat eri kimppuihin. Lohkoja pilkotaan siten, että tilat, joista toisesta alkaa ja toisesta ei ala siirtymä pilkkomiseen käytettävässä kimpussa, joutuvat eri lohkoihin. Pilkkomisjärjestyksellä ei ole väliä oikean lopputuloksen kannalta. Yksi lohko voidaan jättää kokonaan käyttämättä pilkkomiseen, sillä jokainen kimppu, jossa on sekä siihen päättyviä että muita siirtymiä, tulee pilkotuksi muiden siirtymiensä loppupäiden lohkojen vaikutuksesta. Algoritmi lopettaa, kun mikään lohko ja kimppu ei pilkkoudu.

Kuva 7 esittää tällä tavalla toimivan algoritmin. Lohkotietorakenteen taulukoissa ja toiminnoissa on käytetty etuliitettä *l_* ja kimpuille vastaavasti *k_*. Niinpä *l_luku*$[i]$ on selattavan tilan ja *k_luku*$[i]$ siirtymän numero. Muuttuja k sisältää pilkkomiseen käytettävän kimpun ja ℓ lohkon numeron. Koska yhden lohkon saa jättää käyttämättä, aloittaa ℓ arvosta 2. **While**-silmukoiden ehdoissa esiintyvät *l_osia* ja *k_osia* sisältävät lohkojen ja kimppujen kokonaismäärän kunakin ajanhetkenä, joten kumpikin tyypillisesti kasvaa algoritmin suorituksen aikana.

Pääsilmukan vartalo pilkkoo ensin yhden kimpun mukaan ja sitten kaikkien sillä hetkellä jäljellä olevien lohkojen mukaan. Näin saadaan vähillä testeillä varmistettua, että jos aluksi on ainakin yksi kimppu, niin, kun algoritmi lopettaa, on pilkottu kaikkien kimppujen ja yhtä vaille kaikkien lohkojen mukaan. Jos aluksi ei ole yhtään kimppua, niin lohkoilla pilkkominen jää pois, mutta silloin lohkoilla ei ole mitä pilkkoa.

Julkaisuissa [16, 14] pilkottiin lohkolla aina heti kun se oli haljennut. Siksi niissä tarvittiin erilliset H lohkoille ja kimpuille. Nyt lohkoilla pilkkominen alkaa vasta kun H on tyhjennetty lohkojen numeroista, joten selvitään yhdellä H.

Algoritmin tähänastinen esittely keskittyi sen eroihin aikaisempiin verrattuna, joten toiminnan virheettömyys ja nopeus eivät välttämättä tulleet selviksi. Virheettömyys perustuu seuraavaan tulokseen.

```
// Aloitetaan ensimmäisestä kimpusta ja toisesta lohkosta
k := 1;  ℓ := 2;  H := ∅

// Pääsilmukka
while k ≤ k_osia do

    // Pilko lohkoja vuorossa olevan kimpun mukaan
    for i := k_alku[k] to k_loppu[k] − 1 do
        h := l_merkitse( alkupään_tila[ k_luku[i] ] )
        if h > 0 then H := H ∪ {h}
    for h ∈ H do l_halkaise(h)
    H := ∅;  k := k + 1

    // Pilko kimput vastaamaan äsken pilkottuja lohkoja
    while ℓ ≤ l_osia do
        for i := l_alku[ℓ] to l_loppu[ℓ] − 1 do
            for j ∈ tulosiirtymät( l_luku[i] ) do
                h := k_merkitse(j)
                if h > 0 then H := H ∪ {h}
        for h ∈ H do k_halkaise(h)
        H := ∅;  ℓ := ℓ + 1
```

Kuva 7: Deterministisen äärellisen automaatin minimoinnin pilkkomisvaihe.

Apulause 2 Seuraavat asiat pätevät aina kuvan 7 algoritmin pääsilmukan ehdon kohdalla:

1. Jos kaksi siirtymää on eri kimpuissa, niin niillä on eri nimet tai ne päättyvät eri lohkoihin.
2. Erinimiset siirtymät ovat eri kimpuissa.
3. Jos kaksi siirtymää päättyy eri lohkoihin, niin ne ovat eri kimpuissa tai ainakin toisen lohkon numero on vähintään ℓ.
4. Jos kaksi tilaa on eri lohkoissa, niin ne olivat eri lohkoissa alussa tai toisesta alkaa siirtymä, joka kuuluu kimppuun, jossa ei ole toisesta alkavaa siirtymää.
5. Alussa eri lohkoissa olleet tilat ovat aina eri lohkoissa.
6. Jos tilasta alkaa siirtymä, joka kuuluu kimppuun, jossa ei ole toisesta tilasta alkavaa siirtymää, niin tilat kuuluvat eri lohkoihin tai kimpun numero on vähintään k tai toisesta tilasta alkaa samanniminen siirtymä, ja se kuuluu kimppuun, jonka numero on vähintään k.

Todistus. Väitteiden 1 ja 4 ”niin”-osat eivät voimaan astuttuaan voi enää kumoutua, koska lohkoja ja kimppuja ei yhdistetä eikä siirtymien nimiä ja kytkentöjä muuteta. Kokonaisuudessaan väitteet 1 ja 4 pätevät, koska ”niin”-osissa on lueteltu kaikki tilanteet, joissa algoritmi alustus mukaan lukien voi sijoittaa tiloja eri lohkoihin ja siirtymiä eri kimppuihin. Väite 2 saatetaan voimaan alustuksessa eikä voi kumoutua jälkeenpäin. Väite 5 on ilmeinen.

Vain yhden lohkon numero on alle 2. Väite 3 pätee alussa, koska silloin $\ell = 2$. Myöhemmin ”jos”-osa voi astua voimaan vain lohkon haljetessa. Silloin toinen osalohko saa upouuden lohkonumeron, joka

on suurempi kuin mikään siihenastinen ja siten suurempi kuin ℓ. Siirtymän loppupään tilan lohkon numero voi muuttua algoritmin suorituksen aikana, mutta vain upouudeksi lohkonumeroksi. Kun ℓ kasvaa, on algoritmi juuri huolehtinut, että missään kimpussa ei ole sekä lohkoon ℓ että muualle päättyviä siirtymiä.

Väite 6 pätee samankaltaisista syistä kuin väite 3. Se pätee alussa, koska silloin jokaisen kimpun numero on vähintään k. Jos kimpun halkeaminen saattaa ”jos”-osan voimaan, niin kimpun jompi kumpi osa saa riittävän suuren numeron. Siirtymän kimpun numero voi kasvaa mutta ei vähetä. k:n kasvaessa kaikki lohkot on juuri halkaistu kimpun k mukaisesti. □

Seuraava lause ilmaisee, että algoritmi saavuttaa aiemmin asetetun yhteensopivuustavoitteen. Tästä voidaan todistaa, että jos algoritmiin lisätään edellä mainittu poistovaihe, niin sen lopputulos on pienin mahdollinen automaatti, joka määrittelee saman joukon sanoja kuin syötteeksi annettu automaatti. Jätämme todistuksen esittämättä, koska se on puhdasta matematiikkaa eikä mielenkiintoinen tämän kirjoituksen kannalta.

Lause 3 Kuvan 7 algoritmin lopetettua jokainen lohko on yhteensopiva jokaisen aakkosen suhteen, jokainen lohko on osajoukko jostakin alkuperäisestä lohkosta, ja jokainen lohko on niin suuri, kuin edellä mainitut seikat sallivat.

Todistus. Väitteiden 4 ja 1 ansiosta kaksi tilaa joutuu eri lohkoihin vain, jos ne ovat eri lohkoissa alussa tai toisesta lähtee siirtymä johonkin lohkoon siten, että toisesta ei lähde saman nimistä siirtymää samaan lohkoon. Algoritmi ei siis pilko lohkoja enempää kuin on välttämätöntä yhteensopivuuden saavuttamiseksi.

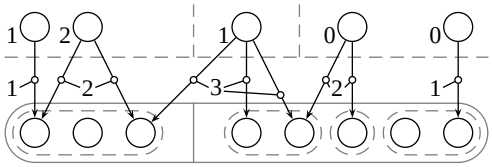
Erikoistapausta $k\_osia = 0$ lukuunottamatta algoritmin lopettaessa minkään kimpun ja lohkon numero ei ole vähintään k ja ℓ. Väitteet 2, 3, 5 ja 6 supistuvat silloin siten, että ne tuottavat väitteille 1 ja 4 vastakkaissuuntaiset väitteet. Kuulukoon kaksi tilaa samaan lohkoon ja olkoon toisesta jonkin niminen siirtymä johonkin lohkoon. Väitteen 4 negaation vuoksi toisestakin on samaan kimppuun kuuluva siirtymä. Sillä on väitteen 1 negaation vuoksi sama nimi ja määränpäälohko kuin ensin mainitulla. Pilkkominen on siis saatettu loppuun eli yhteensopivuus on saavutettu. □

Vielä täytyy osoittaa, että algoritmi on niin nopea kuin luvattiin.

Lause 4 Kuvan 7 algoritmin suoritusaika on $O(m \log n)$, missä n on tilojen ja m siirtymien määrä.

Todistus. Väitteen 2 vuoksi ja koska automaatti on deterministinen, on kussakin kimpussa korkeintaan n siirtymää. Muuttujien k ja ℓ juoksutus tekee ilmeiseksi, että jos samaa siirtymää tai tilaa käytetään pilkkomiseen monta kertaa, se kuuluu joka kerta kimppuun tai lohkoon, jonka numero on suurempi kuin edellisellä kerralla. Koska suuremman eli uuden numeron saa halkaistun kimpun tai lohkon pienempi osa, on uudelleen käytettävän siirtymän kimppu tai tilan lohko kooltaan korkeintaan puolet edelliskertaisesta. Siksi samaa siirtymää tai tilaa käytetään korkeintaan noin $\log_2 n$ kertaa.

Tilojen tulosiirtymien yhteismäärä on sama kuin siirtymien määrä. Näistä, luvun 4 tehokkuusanalyyseista sekä oletuksesta, että H, *alkupään_tila* ja tulosiirtymien selaus on toteutettu asianmukaisesti seuraa, että algoritmin suoritusaika on $O(m \log n)$. □



Kuva 8: Paigen ja Tarjanin algoritmin havainnollistus.

6 Paigen ja Tarjanin algoritmi

Äärellisen automaatin deterministisyys takaa, että jos tilasta on siirtymä johonkin lohkoon, niin siitä ei ole samannimistä siirtymää johonkin toiseen lohkoon. Edellisen luvun algoritmi ja päättely hyödynsivät tätä. Tilanne monimutkaistuu huomattavasti, jos sallitaan monta samannimistä siirtymää samasta tilasta, jopa siinä tapauksessa, että eri nimiä on kaikkiaan käytössä vain yksi. Yhtäpitävästi voidaan olettaa, että siirtymillä ei ole nimiä, ja samasta tilasta voi lähteä monta siirtymää. Paige ja Tarjan käsittelivät tätä tilannetta tiiviissä ja työläästi luettavassa julkaisussa [13].

Vaikeus on siinä, että jos lohkoa L on käytetty pilkkomiseen ja L halkeaa lohkoiksi L_1 ja L_2, niin L_1:n käyttö pilkkomiseen edellä kuvattuun tapaan ei erota niitä tiloja, joista on siirtymä sekä L_1:een että L_2:een niistä, joista on siirtymä pelkästään L_1:een. Tarvitaan keino tehdä tämä erottelu. Sen seurauksena lohko pilkkoontuu kerralla enintään kahden osan sijasta enintään kolmeen osaan. Tämäkin on hallittava jotenkin.

Paige ja Tarjan ratkaisivat nämä *koostelohkojen* ja nerokkaasti käytettyjen laskurien avulla. Koostelohko on tietynlainen epätyhjä joukko lohkoja. Aluksi kaikki lohkot kuuluvat samaan koostelohkoon, ja myöhemmin koostelohkoja syntyy vain jäljempänä kerrottavalla tavalla. Voidaan osoittaa, että koostelohkon lohkojen unionin aiheuttama pilkkomistarve on toteutettu. Jotta tämä pätisi alussa, aivan aluksi tilat pitää jakaa niihin, joista lähtee siirtymiä ja niihin, joista ei lähde.

Tekemättömästä työstä pidetään kirjaa joukolla, jossa ovat niiden koostelohkojen numerot, joissa on ainakin kaksi lohkoa. Pilkkominen aloitetaan poistamalla jostain tällaisesta koostelohkosta yksi lohko ja julistamalla se sekä pilkkojaksi että itsessään koostelohkoksi. Kuvan 8 alarivissä on koostelohko, jonka vasemmasta reunasta on erotettu kolmetilainen pilkkoja.

Pilkkomisen tuloksena jokainen lohko, jonka ainakin yhdestä tilasta on siirtymä alkuperäiseen koostelohkoon, jakaantuu enintään kolmeksi lohkoksi. *Vasen lohko* koostuu niistä tiloista, joiden kaikki alkuperäiseen koostelohkoon vievät siirtymät vievät pilkkojaan; *oikean lohkon* tilojen siirtymistä mikään ei vie pilkkojaan; ja *keskilohkon* tiloista on siirtymiä sekä pilkkojaan että muualle alkuperäiseen koostelohkoon. Kuvan 8 ylärivin lohkon pilkkoutuminen on osoitettu katkoviivoilla. Oikea lohko erottuu muista sillä, että sen tiloja ei kohdata kuljettaessa pilkkojaan päättyvät siirtymät takaperin, mutta vasemman lohkon erottaminen keskilohkosta on vaikeaa.

Jokaiseen tilaan ja koostelohkoon liittyy kuvitteellinen *päälaskuri*, joka kertoo, kuinka monta siirtymää vie kyseessä olevasta tilasta kyseessä olevaan koostelohkoon. Oikeasti toteutetaan ne päälaskurit, joissa oleva arvo on suurempi kuin nolla. Jokaiseen siirtymään liittyy linkki, jonka avulla löydetään se päälaskuri, jonka ti-

la on siirtymän alkutila ja jonka koostelohkoon siirtymä päättyy. Sanomme sitä siirtymän päälaskuriksi, vaikka se voi olla monen siirtymän yhteinen. Siirtymä itse kuuluu päälaskurinsa laskemiin siirtymiin, joten päälaskurin arvo on aina positiivinen. Kuvassa 8 linkit on esitetty näin: ”◦—”.

Lisäksi jokaiseen tilaan liittyy *apulaskuri* lyhytaikaista käyttöä varten. Kuvan 8 ylärivin tilojen apulaskurit on esitetty tilojen vieressä. Pilkkomisen aluksi pilkkojaan tulevat siirtymät käydään läpi ja jokaisen niistä alkupään tilan apulaskuria kasvatetaan yhdellä. Näin saadaan selville, kuinka monta siirtymää kustakin tilasta vie pilkkojaan.

Sitten pilkkojaan tulevat siirtymät käydään läpi uudelleen. Vertaamalla siirtymän alkupään tilan apulaskurin arvoa siirtymän päälaskurin arvoon saadaan selville, vievätkö kaikki tilasta alkuperäiseen koostelohkoon vievät siirtymät pilkkojaan, vai viekö osa niistä muualle koostelohkoon. Toisin sanoen, saadaan selville, kuuluuko tila vasempaan vai keskilohkoon. Tämän perusteella tilat merkitään yhdellä tai toisella tavalla myöhempää halkaisemista varten. Oikeaan lohkoon kuuluvat ne tilat, joita ei kohdata tässä prosessissa ja jotka jäävät siksi merkitsemättä.

Keskilohkon jokaista tilaa varten joudutaan perustamaan uusi päälaskuri, jotta olisi olemassa päälaskuri sekä pilkkojaa että alkuperäisen koostelohkon loppuosaa varten. Pilkkojan päälaskurin arvoksi asetetaan tilan apulaskurin arvo. Sama arvo vähennetään alkuperäisen koostelohkon päälaskurista, joka jatkaa koostelohkon loppuosan päälaskurina. Siirtymän linkki käännetään osoittamaan uutta päälaskuria. Kaikki tämä kannattaa tehdä, kun tila kohdataan ensimmäisen kerran jälkimmäisellä selauksella. Tieto siitä, että tämä on tehty, laitetaan talteen yhdessä uuteen päälaskuriin vievän linkin kanssa. Kun tila kohdataan selauksen edetessä uudelleen, riittää päivittää siirtymän linkki.

Apulaskurit täytyy nollata tulevaa käyttöä varten. Tämän voisi tehdä heti kun tilan laji on selvinnyt. Toisaalta, juuri siksi, että apulaskuria ei sen jälkeen tarvita, sen paikalle voidaan tarvittaessa tallettaa uuteen päälaskuriin vievä linkki, kunhan yksi bitti (esimerkiksi etumerkki) uhrataan kertomaan, kumpi tieto muistipaikassa on. Näin säästetään muistia ja suututetaan ohjelmien ylläpidettävyydestä huolta kantavat. Apulaskurit voi myöhemmin nollata tehokkaasti selaamalla pilkkojaan tulevat siirtymät kolmannen kerran.

Kun kaikki tämä on tehty, niin halkaistaan ne lohkot, joiden tiloja merkittiin edellä. Tämä tapahtuu muuten kuten luvussa 5, paitsi tiloja on merkitty kahdella tavalla ja lohkoja voidaan joutua halkaisemaan kolmeen osaan.

Paigen ja Tarjanin algoritmi voidaan yleistää nimetyille siirtymille edellisen luvun kimppujen avulla [14]. Myös muut edellisen luvun parannukset voidaan ottaa käyttöön. Itse asiassa olen ohjelmoinut ja testannut edellisen luvun uudet ajatukset tässä yhteydessä enkä deterministisillä äärellisillä automaateilla.

7 Derisavin, Hermannsin ja Sandersin algoritmi

Pääsemme vihdoin takaisin Markovin ketjuihin. Kirjallisuudessa on huolettomasti väitetty, että soveltamalla Paigen ja Tarjanin algoritmia voidaan Markovin ketjun tilojen yhdistämistehtävä ratkaista $O(m \log n)$ ajassa, missä n on yhä tilojen ja m siirtymien määrä. Derisavi, Hermanns ja Sanders huomauttivat vuonna 2003, että asia ei ole niin yksinkertainen [5].

Markovin ketjun tilojen yhdistämisel-

lä ja Paigen ja Tarjanin algoritmilla on paljon yhteistä. Tärkein ero on seuraava. Paigen ja Tarjanin algoritmissa olennaista on, onko tilasta siirtymää pilkkojaan. Markovin ketjun tilojen yhdistämisessä olennaista on, kuinka paljon on tilasta pilkkojaan vievien siirtymien todennäköisyyksien summa. Summien laskemiseksi Derisavi työtovereineen käytti edellisen luvun apulaskurien kaltaista menetelmää.

Koska alkuperäisen koostelohkon pilkkomisvaikutus on jo tehty, on kaikilla vasemman, oikean ja keskilohkon tiloilla sama alkuperäiseen koostelohkoon vievien siirtymien todennäköisyyksien summa. Vasemman lohkon tiloilla koko tämä summa kohdistuu pilkkojaan. Muualle alkuperäiseen koostelohkoon kohdistuva summa on nolla. Oikean lohkon tiloilla tilanne on päinvastainen. Keskilohkon tiloilla osa summasta kohdistuu pilkkojaan ja osa muualle alkuperäiseen koostelohkoon. Jakauma voi olla erilainen eri tiloilla. Esimerkiksi yhdestä tilasta voidaan siirtyä todennäköisyydellä 0,1 pilkkojaan ja 0,6 muualle koostelohkoon, ja toisella tilalla vastaavat todennäköisyydet voivat olla 0,4 ja 0,3.

Näin ollen vasen ja oikea lohko ovat yhteensopivia sekä pilkkojan että alkuperäisen koostelohkon loppuosan suhteen, mutta keskilohko ei välttämättä ole. Keskilohkon tilat täytyy ryhmitellä ja jakaa lohkoiksi sen mukaan, mikä on niistä pilkkojaan (tai vaihtoehtoisesti muualle alkuperäiseen koostelohkoon) vievä summa. Kun tällainen keino on otettu käyttöön, sillä voi myös erottaa vasemman lohkon tilat keskilohkon tiloista. Tämä on etu, sillä, kuten luvussa 6 nähtiin, Paigen ja Tarjanin algoritmin keino tehdä sama asia on niin monimutkainen, että siitä eroon pääsy olisi suotavaa.

Luonnollisin ryhmittelykeino on järjestää keski- ja vasemman lohkon tilat summan mukaan, jolloin samaan ryhmään kuuluvat tilat asettuvat peräkkäin. Derisavi työtovereineen huomautti, että koska tyypilliset järjestämisalgoritmit eivät vie $O(k)$ vaan $O(k \log k)$ aikaa, missä k on järjestettävien tilojen määrä, tulee pilkkomisalgoritmin suoritusaikaan ylimääräinen logaritmitermi. Siis suoritusaika on $O(m \log^2 n)$. Niiden aikaisempien kirjoittajien, jotka väittivät, että $O(m \log n)$ riittää, olisi tullut kertoa, miten ylimääräinen logaritmitermi voidaan välttää. Tätä he eivät olleet tehneet.

Derisavi työtovereineen osoitti myös, että jos keski- ja vasemman lohkon unioni järjestetään niin sanottujen splay-puiden avulla, niin pilkkomisalgoritmin suoritusaika putoaa tavoiteltuun kertaluokkaan $O(m \log n)$. Splay-puut ovat kuitenkin vähän tunnettu, monimutkainen ja (kuten binääripuut yleensä) melko paljon muistia kuluttava tietorakenne. Ylimääräinen logaritmitermi ei ole edes teoriassa iso hidastus. Voi olla, että splay-puihin perustuva ratkaisu on monimutkaisuutensa vuoksi käytännön tilanteissa yleensä hitaampi kuin tavallisen yksinkertaisen järjestämisalgoritmin käyttö. Siksi tämän tuloksen käytännön merkitys vaikuttaa pieneltä.

Julkaisu [5] keskittyy nopeuteen splay-puiden kanssa ja ilman. Algoritmin yleisen oikeellisuuden se kuittaa epämääräisillä viittauksilla Paigen ja Tarjanin algoritmiin sekä lähteisiin, joissa puhutaan Markovin ketjun tilojen yhdistämisestä yleisellä mutta ei julkaisun algoritmin tasolla. Tämä ei riitä, sillä julkaisun algoritmi poikkeaa Paigen ja Tarjanin algoritmista huomattavasti. Se muun muassa ei käytä koostelohkoja.

Lukijan on vaikea saada julkaisusta [5] selville, mikä on sen algoritmin oikeellisuuden kannalta olennaista. Lisäksi algoritmissa sellaisena kuin se julkaisussa esi-

tetään on virhe. Kun lohko halkeaa, niin sen osia pitää lisätä vuoroaan odottavien pilkkojien joukkoon. Julkaisussa sinne lisätään aina kaikki muut osat paitsi suurin. Suurimman osan saa kuitenkin jättää pois vain siinä tapauksessa, että haljennut lohko ei itse ollut vuoroaan odottavien pilkkojien joukossa. Virhe ei kumoa algoritmin perusajatuksia, mutta johtaa siihen, että julkaisun mukaan toteutettu ohjelma laskisi väärin. Lukijalla on oltava huomattavat taustatiedot voidakseen havaita ja korjata virheen.

8 Eroon splay-puista

Kesäkuussa 2009 pidin esitelmää julkaisusta [14], joka esittää Paigen ja Tarjanin algoritmin yleistyksen tilanteeseen, jossa siirtymillä on vapaasti valittavat nimet. (Senkin kohdalla kirjallisuudessa on huolettomasti mutta virheellisesti väitetty, että yleistämiseen ei kätkeydy erityisiä ongelmia.) Yleisössä ollut Giuliana Franceschinis kysyi, voiko algoritmiani soveltaa Markovin ketjun tilojen yhdistämistehtävään, sillä algoritmini vaikutti hänestä yksinkertaisemmalta kuin Derisavin ja työtovereiden.

Paljastui, että julkaisun [14] uusista tuloksista ei ole apua, mutta Markovin ketjun tilojen yhdistämistehtävä oli muuten kiinnostava. Tästä syntyi julkaisu [15]. Siinä osoitimme ja korjasimme luvussa 7 mainitun virheen, korvasimme splay-puut yksinkertaisemmalla mekanismilla, ja annoimme perusteelliset algoritmikuvaukset ja todistukset.

Splay-puiden korvaamisessa tarvitaan seuraavaa tulosta.

Apulause 5 Jos Markovin ketjun tilojen osituksen pilkkomisen aikana käsiteltyjen keskilohkojen koot ovat $k_1, k_2, \ldots, k_K$, niin $\sum_{j=1}^{K} k_j \log k_j \leq m \log n$.

Todistus. Tarkoittakoon $\kappa(i)$ niiden koostelohkojen määrää, joihin tilasta i on ainakin yksi siirtymä. Käymällä kaikki tilat läpi saadaan $\sum_{i=1}^{n} \kappa(i) \leq m$.

Keskilohkon tilasta on siirtymä sekä pilkkojaan että muualle alkuperäiseen koostelohkoon. Pilkkojasta tulee uusi koostelohko. Tästä seuraa, että joka kerta kun i on keskilohkossa, $\kappa(i)$ kasvaa yhdellä. Niinpä, jos $\lambda(i)$ on niiden kertojen määrä, jotka tila i on ollut keskilohkossa, niin $\lambda(i) \leq \kappa(i)$. Kun $\lambda(i)$ summataan kaikkien tilojen yli saadaan käsiteltyjen keskilohkojen kokojen summa, eli $\sum_{j=1}^{K} k_j = \sum_{i=1}^{n} \lambda(i)$. Yhdistämällä tähänastiset huomiot saadaan $\sum_{j=1}^{K} k_j \leq m$.

Koska lohkon koko on enintään tilojen määrä, on $\sum_{j=1}^{K} k_j \log k_j \leq \sum_{j=1}^{K} k_j \log n = (\sum_{j=1}^{K} k_j) \log n$. Edellä saadun tuloksen kanssa tämä tuottaa $\sum_{j=1}^{K} k_j \log k_j \leq m \log n$. □

Jos keskilohkot järjestetään $O(k \log k)$ aikaa kuluttavalla algoritmilla, niin luku $\sum_{j=1}^{K} k_j \log k_j$ kuvaa järjestämisten yhteensä kuluttamaa aikaa. Apulause sanoo siis, että algoritmin tavoiteltu suoritusaika riittää keskilohkojen järjestämiseen tavallisella järjestämisalgoritmilla. Näin ollen, jos vasemman lohkon tilat saadaan erotettua keskilohkon tiloista ennen järjestämistä, niin tavallinen järjestämisalgoritmi riittää eikä monimutkaisia splay-puita tarvita.

Vasemman lohkon tilat saadaan erotettua Paigen ja Tarjanin algoritmin laskurikeinolla. Ikävä kyllä se on niin monimutkainen, että ajatus tyytymisestä $O(m \log^2 n)$ suoritusaikaan tuntuu houkuttelevammalta.

Diskreetin ajan Markovin ketjujen tapauksessa siirtymiin liittyvät luvut ovat todennäköisyyksiä ja niin ollen positiivisia. Siitä seuraa, että joko vasen lohko on tyhjä tai se koostuu niistä tiloista, joiden pilkkojaan vievien siirtymien todennäköi-

syyksien summa on suurin. Summan suurin arvo on helppo selvittää selaamalla vasemman ja keskilohkon unionin tilat läpi, jonka jälkeen sen tuottavat tilat voidaan ottaa erilleen selaamalla tilat uudelleen läpi. Jos vasen lohko on tyhjä, tämä keino erottaa keskilohkon jonkin osan, mutta se ei haittaa. Järjestämisen tarkoitus on ryhmitellä keskilohkon tilat summien mukaan, eikä tätä häiritse, jos yksi ryhmä otetaan etukäteen erilleen.

Valitettavasti tämä keino ei toimi jatkuvan ajan Markovin ketjuille, sillä niiden siirtymissä esiintyy myös negatiivisia lukuja. Tarvitaan toinen keino. Äskeinen toteamus yleistäen, ei haittaa, jos keino erottaakin väärän ryhmän, kunhan jäljelle jäävissä ryhmissä on yhteensä tarpeeksi vähän tiloja, ja keino on nopea ja yksinkertainen. On helppo osoittaa, että aika riittää, vaikka luvut k_j kaksinkertaistettaisiin. Joko lukija arvasi?

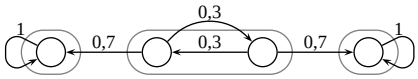
9 Tarvitaanko enemmistöarvon löytämisalgoritmia varmasti?

Enemmistöarvon löytämisalgoritmi on ihanteellinen keino valita yksi ryhmä erotettavaksi vasemman ja keskilohkon unionista ennen järjestämistä. Jos vasen lohko on suurempi kuin keskilohko, niin se löytää vasemman lohkon tiloihin liittyvän summan. Päinvastaisessa tapauksessa vasen ja keskilohko ovat yhteensä niin pienet, että aika riittää niiden unionin järjestämiseen $O(k \log k)$ aikaa kuluttavalla algoritmilla, joten ei ole väliä, mikä ryhmä erotetaan. Enemmistöarvon löytämisalgoritmi on melkein yhtä nopea ja yksinkertainen kuin suurimman summan etsiminen, mutta toimii silloinkin kun siirtymiin liittyvät luvut voivat olla negatiivisia. Näistä syistä julkaisussa [15] käytetään enemmistöarvon löytämisalgoritmia yhden ryhmän erottamiseen.

Enemmistöarvon löytämisalgoritmi on täten saanut mielekkään tehtävän. Sen ansiosta yhdistämisalgoritmin suoritusajaksi saadaan $O(m \log n)$ ilman monimutkaisten splay-puiden käyttöä. Kohta tullaan näkemään, että algoritmien käytännön nopeuksia on vaikea verrata kovin tarkasti. Silti on helppo uskoa, että julkaisun [15] algoritmi on yksinkertaisuutensa ja pienemmän muistin kulutuksensa ansiosta käytännössä yleensä tehokkaampi kuin julkaisun [5] algoritmi.

Vaikeampi on selvittää, tuoko yhden ryhmän erottaminen enemmistöalkion löytämisalgoritmilla käytännössä hyötyä verrattuna siihen, että koko vasemman ja keskilohkon unioni järjestetään yhdessä. Se pudottaa ajan kulutuksen kertaluokasta $O(m \log^2 n)$ kertaluokkaan $O(m \log n)$. Kuitenkaan O-merkinnän tasolla paras algoritmi ei aina ole käytännössä paras, sillä O-merkintä ilmaisee suoritusajalle ylärajan, ja algoritmi voi olla suurimmassa osassa tapauksia paljon sitä nopeampi. Esimerkiksi Quicksort katsotaan yleensä käytännössä nopeammaksi kuin Heapsort, vaikka Quicksortin ajan kulutus on $O(k^2)$ ja Heapsortin $O(k \log k)$.

Enemmistöalkion löytämisalgoritmi on itsessään niin halpa, että se ei missään oloissa kasvata järjestämiseen käytettävää aikaa merkittävästi. Olisi houkuttelevaa päätellä tästä, että sen käytöllä ei koskaan voi hävitä merkittävästi mutta voi voittaa merkittävästi, joten sitä varmasti kannattaa käyttää. Valitettavasti asia ei ole näin yksinkertainen. Yhden osan erottaminen muuttaa järjestystä, jossa uudet lohkot saavat numeronsa. Siten se muuttaa järjestystä, jossa lohkoja käytetään pilkkomiseen. Kuten seuraavaksi perustellaan, sillä voi olla vaikutusta kokonaistyömäärään.



Oheisen kuvan jomman kumman reunalohkon käyttö pilkkomiseen halkaisee keskellä olevan lohkon, eikä muita halkeamisia tapahdu. Kuvasta 7 näkyy, että lohkoa numero 1 ei käytetä pilkkomiseen. Jos keskimmäinen lohko saa alun perin numeron 2, niin sitä käytetään pilkkomiseen ennen kuin se itse halkeaa. Sen molemmat tilat käydään läpi kertaalleen ja myöhemmin vielä jompi kumpi tila kertaalleen. Jos se saa numeron 3, niin sitä käytetään pilkkomiseen vasta kun se on itse haljennut, joten sen kumpikin tila käydään läpi vain kerran. Erolla ei ole muuta vaikutusta algoritmin toimintaan. Jälkimmäisessä tapauksessa tehdään kaiken kaikkiaan vähemmän työtä.

Esimerkki havainnollistaa, että tarkka suoritusaika voi riippua seikoista, jotka eivät määräydy algoritmista vaan siitä, miten se on toteutettu ohjelmana, missä järjestyksessä syöte annetaan ja niin edelleen. Sellaisia ei voi ottaa huomioon algoritmeja vertailtaessa. Ohjelmien paremmuusjärjestystä ei läheskään aina voi selvittää mittaamalla suoritusaikoja, sillä yhdellä syöteaineistolla nopeampi ohjelma saattaa olla hitaampi toisella syöteaineistolla. Tekemissäni mittauksissa ei ilmennyt satunnaisvaihtelua suurempaa nopeuseroa kumpaankaan suuntaan.

On siis erittäin vaikeaa selvittää sitovasti, onko yhden osan erottaminen ennen järjestämistä käytännössä parempi, yhtä hyvä vai huonompi ratkaisu kuin erottamatta jättäminen. Ei kuitenkaan ole mitään erityistä syytä olettaa, että se olisi huonompi, ja erottamisen halpuus ja sen tuoma parannus *O*-merkinnän tasolla ovat syitä olettaa, että se on parempi. Sen käyttö on siis järkevää.

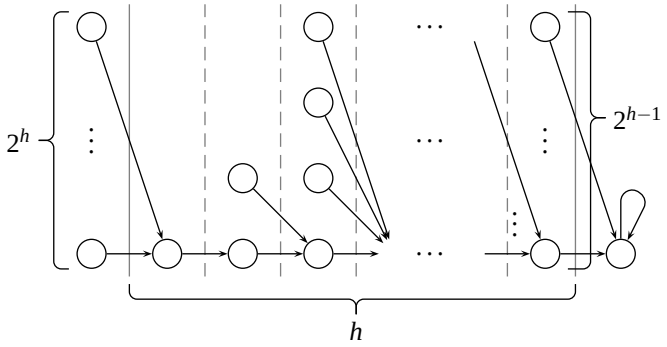
Tässä vaiheessa on ainakin periaatteessa mahdollista, että uusi tieto muuttaa tilannetta. Splay-puut joutuivat syrjään, koska julkaisussa [15] onnistuttiin osoittamaan, että tavoiteltu aika riittää keskilohkon järjestämiseen tavallisella algoritmilla. Kenties tavoiteltu aika riittää myös vasemman ja keskilohkon unionin järjestämiseen tavallisella algoritmilla. Derisavin ja muiden huomautus ei ollut, että se ei riitä, vaan että puuttuu todistus, että se riittää. Jos sellainen todistus löytyy, niin mitään ryhmää ei tarvitse erottaa ennen järjestämistä ja enemmistöarvon löytämisalgoritmi muuttuu taas tarpeettomaksi. Seuraava apulause kertoo, voiko näin käydä.

Apulause 6 Keski- ja vasemman lohkon unionien järjestämiset $\Theta(k \log k)$ aikaa vievällä algoritmilla vievät hitaimmassa tapauksessa yhteensä $\Theta(m \log^2 n)$ aikaa.

Todistus. Kuvassa 9 on esitetty perhe Markovin ketjuja, jossa on yksi ketju jokaiselle luvun h arvolle. Alkujako on osoitettu harmailla viivoilla.

Jos keskimmäistä alkuperäistä lohkoa käytetään ensimmäisenä pilkkojana, se halkaisee itsensä oikeanpuoleisinta katkoviivaa pitkin. Katkoviivan vasemmalle puolelle jää $2^{h-1}-1$ ja oikealle puolelle 2^{h-1} sen tilaa, joten jatkossa käytettäväksi pilkkojaksi valitaan vasen puoli. Kun sitä käytetään pilkkojana, se erottaa seuraavan palstan tilat muista. Tätä jatkuu, kunnes lohko on pilkottu kuvan kaikkia katkoviivoja pitkin.

Jokaisessa pilkkomisessa kuvan vasemmassa reunassa olevat 2^h tilaa muodostavat vasemman lohkon, vastaavan keski- ja oikean lohkon ollessa tyhjiä. Jos vasemman ja keskilohkon unioni järjestetään jokaisen pilkkomisen yhteydessä $\Theta(k \log k)$ aikaa vievällä algoritmilla, niin kuvan vasen reuna aiheuttaa h järjestämistä joista kukin käsittelee 2^h alkiota. Ne



Kuva 9: Esimerkki, jossa sama iso vasen lohko tulee käsiteltäväksi useasti [15]. Jokaisen siirtymän todennäköisyys on 1.

vievät yhteensä aikaa lukuun $h(2^h \log 2^h)$ verrannollisesti. Kuvassa $m = n = 2^{h+1}$, joten ajan kulutus on verrannollinen lukuun $(\log_2 \frac{n}{2})\frac{m}{2} \log \frac{n}{2} = \Theta(m \log^2 n)$. □

Tavoiteltu suoritusaika ei siis riitä vasemman ja keskilohkon unionien järjestämiseen tavallisella algoritmilla. Kyse ei ole siitä, että puuttuu todistus, että se riittää, vaan siitä, että se todellakaan ei riitä. Vasemman lohkon ottaminen erilleen silloin kun se on suuri on siis välttämätöntä. Enemmistöarvon löytämisalgoritmin käyttö ei ole turhaa.

10 Yhteenveto

Tässä kirjoituksessa tarkasteltiin osituksen pilkkomiseen perustuvia algoritmeja kolmen tehtävän ratkaisemiseen. Tehtävät olivat klassinen äärellisen automaatin minimointi; sen muunnos, jossa siirtymillä ei ole nimiä, mutta samasta tilasta voi silti lähteä monta siirtymää; sekä Markovin ketjun samanveroisten tilojen yhdistäminen. Markovin ketjut ovat hyödyllisiä todennäköisyyslaskentaan liittyvien tehtävien numeerisessa ratkaisemisessa. Tavoitteena oli ratkaista tehtävät $O(m \log n)$ ajassa, missä n on tilojen ja m siirtymien määrä.

Markovin ketjun tapauksessa on tarpeen luokitella tiloja tilasta pilkkojana käytettävään lohkoon vievien kaarten painojen summien mukaan. Luokittelua ei voi toteuttaa tavallisella järjestämisalgoritmilla koska, kuten kirjoituksessa todistettiin, tavoiteltu suoritusaika ei riitä siihen. Onneksi järjestettävät tilat ovat peräisin kahdesta lähteestä, joista toinen tuottaa niin vähän tiloja, että aika riittää niiden järjestämiseen, ja toisen tuottamat tilat kuuluvat keskenään samaan luokkaan, joten niitä ei tarvitse järjestää. Ongelmaksi jää erotella eri lähteistä tulevat tilat toisistaan.

Tilat voi erotella Paigen ja Tarjanin esittämällä laskurikeinolla. Valitettavasti se on monimutkainen. Helpommalla päästään erottelemalla yksi ryhmä tiloja enemmistöarvon löytämisalgoritmin avulla. Se on erittäin yksinkertainen ja tehokas algoritmi, joka löytää eniten esiintyvän arvon, jos se esiintyy useammin kuin muut arvot yhteensä. Muussa tapauksessa se palauttaa mielivaltaisen aineistoon kuuluvan arvon.

Kyvyttömyys tuottaa informatiivinen vastaus silloin, kun mikään arvo ei yksinään muodosta enemmistöä, on heikkous, jonka vuoksi enemmistöarvon löytämisalgoritmilla on niukasti todellista hyöty-

käyttöä. Markovin ketjujen sovelluksessa siitä ei kuitenkaan ole haittaa. Jos jälkimmäisen lähteen tuottamia tiloja on niin vähän, että ne eivät muodosta enemmistöä, niin aika riittää kaikkien tilojen järjestämiseen, joten ei ole väliä, mikä ryhmä tiloja erotetaan.

Enemmistöarvon löytämisalgoritmi on siis yksinkertainen ja tehokas keino nopeuttaa Markovin ketjujen tehtävässä esiintyvää järjestämistä sen verran, että tavoiteltuun suoritusaikaan $O(m \log n)$ päästään.

Tarkasteltavia algoritmeja käsiteltiin melko laajasti. Muun muassa esiteltiin tietorakenne, jolla pilkottavan osituksen voi esittää tehokkaasti. Matkan varrella kuvailtiin muutamia viime vuosina keksittyjä parannuksia pilkkomiseen perustuviin algoritmeihin. Kerrottiin, miten deterministisen äärellisen automaatin minimoinnin ajan kulutus voidaan pudottaa kertaluokasta $O(\alpha n \log n)$ kertaluokkaan $O(m \log n)$, missä α on aakkoston koko. Kerrottiin, miten voidaan päästä eroon tietorakenteesta, jota aikaisemmin on tarvittu pitämään kirjaa pilkkomisesta, joka vielä tulee tehdä. Kerrottiin, miten eräästä toisestakin tietorakenteesta päästään eroon. Kirjoituksessa esitetty nopea deterministisen äärellisten automaattien minimointialgoritmi lienee yksinkertaisempi kuin mikään aikaisemmin esitetty.

Viitteet

1. Aho, A.V., Hopcroft, J.E. & Ullman, J.D.: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA (1974)
2. Backhouse, R.C.: *Program Construction and Verification*. Prentice-Hall International Series in Computer Science, UK (1986)
3. Boyer, C.: *Tieteiden kuningatar, matematiikan historia osa II*. Suomentanut Kimmo Pietiläinen. Art House (1994)
4. Boyer, R.S. & Moore, J S.: MJRTY: A Fast Majority Vote Algorithm. Boyer, R.S. (toim.): *Automated Reasoning: Essays in Honor of Woody Bledsoe*, 105–117, Kluwer Academic Publishers, Dordrecht, The Netherlands (1991)
5. Derisavi, S., Hermanns, H. & Sanders, W.H.: Optimal State-space Lumping in Markov Chains. *Information Processing Letters* 87(6), 309–315 (2003)
6. Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming* 13, 219–236 (1989/90)
7. Freshers Interviews: Majority Element. http://placementsindia.blogspot.com/2007/09/majority-element.html Luettu 14.5.2010.
8. Gries, D.: Describing an Algorithm by Hopcroft. *Acta Informatica* 2, 97–109 (1973)
9. Hopcroft, J.: An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. Technical Report STAN-CS-71-190, Stanford University (1971)
10. Knuutila, T.: Re-describing an Algorithm by Hopcroft. *Theoretical Computer Science* 250, 333–363 (2001)
11. Morgan, C.: *Programming from Specification, 2nd ed.*. Prentice-Hall International Series in Computer Science (1994) Myös http://web2.comlab.ox.ac.uk/oucl/publications/books/PfS/ (1998)
12. Mäenpää, P.: *The Art of Analysis: Logic and History of Problem Solving*. Väitöskirja, Helsingin yliopisto, Filosofian laitos (1993)
13. Paige, R. & Tarjan, R.: Three Partition Refinement Algorithms. *SIAM Journal of Computing* 16(6), 973–989 (1987)
14. Valmari, A.: Bisimilarity Minimization in $O(m \log n)$ Time. Franceschinis, G. & Wolf, K. (toim.) *Petri Nets 2009, Lecture Notes in Computer Science* 5606, 123–142, Springer, Heidelberg (2009)
15. Valmari, A. & Franceschinis, G.: Simple $O(m \log n)$ Time Markov Chain Lumping. Esparza, J. & Majumdar, R. (toim.)

TACAS 2010, Lecture Notes in Computer Science 6015, 38–52, Springer, Heidelberg (2010)

16. Valmari, A. & Lehtinen, P.: Efficient Minimization of DFAs with Partial Transition Functions. Albers, S. & Weil, P. (toim.) *STACS 2008, Symposium on Theoretical Aspects of Computer Science*, Bordeaux, France, 645–656. http://drops.dagstuhl.de/volltexte/2008/1328/ (2008)