Mitä pieni Rubikin kuutio opetti minulle tietorakenteista, informaatioteoriasta ja satunnaisuudesta

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

1 Johdanto

Keväällä 2001 etsiskelin esimerkkiä, jonka avulla voisin opettaa tila-avaruuden muodostamisalgoritmeja ilman, että ensin tarvitsisi pitää luentosarja esimerkin maailmasta. Perinteinen "ruokailevat filosofit" ei tuntunut tarpeeksi motivoivalta eikä haasteelliseltakaan.

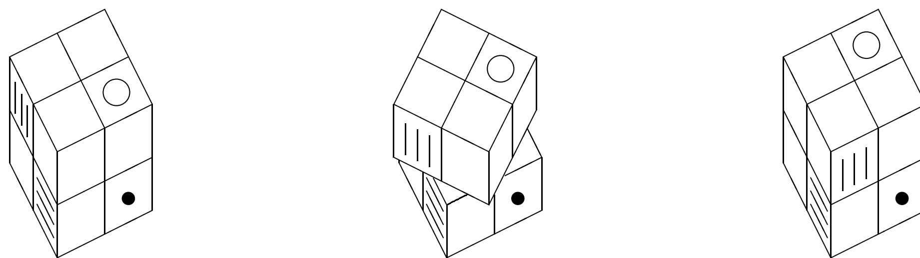
Mieleeni juolahti, että lapsuudestani tutun *Rubikin kuution*[1] $2 \times 2 \times 2$ versiolla saattaisi olla sopiva määrä tiloja. Pian esitettävä pieni laskelma osoittaa, että sillä on 3 674 160 tilaa. Se on niin vähän, että arvelin sen tila-avaruuden mahtuvan silloisiin tietokoneisiin, mutta kuitenkin niin paljon, että huonojen ja hyvien tila-avaruuden muodostamisalgoritmien erot tulisivat esille.

$2 \times 2 \times 2$ Rubikin kuutio osoittautui hedelmällisemmäksi esimerkiksi kuin ikinä olisin osannut arvata. Sen parissa puuhailu tuotti yllätyksiä niin C++ standardikirjastosta kuin informaatioteoreettisista alarajoista, ja johti lopulta julkaisuun, jossa esitellään ja perustellaan äärimmilleen tiivistetty hajautustaulu. Tässä kirjoituksessa käyn uudelleen läpi tuon matkan kohokohdat.

2 Kuution tila-avaruus

$2 \times 2 \times 2$ Rubikin kuutio koostuu kahdeksasta palasta. Sen jokainen tahko on jaettu keskeltä kulkevilla jakolinjoilla neljäksi yhtäsuureksi neliöksi, ja kukin pala käsittää kolme neliötä, jotka kohtaavat samassa nurkassa. Kuution sisäisen nerokkaan ra-

[1] "Rubik" on Seven Towns Ltd -nimisen tahon tavaramerkki.



Kuva 1: $2 \times 2 \times 2$ Rubikin kuution ylätahkon pyöräytys

kenteen ansiosta kutakin tahkoa voi pyörittää ilman, että kuutio hajoaa kappaleiksi. Tällöin neljä palaa liikkuu toisten neljän palan suhteen. Tätä on havainnollistettu kuvassa 1.

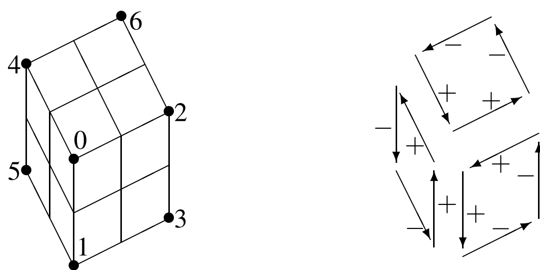
Kuution kukin sivu on alunperin väritetty yhdellä värillä, mutta tahkoja pyöriteltäessä värit menevät pian iloisesti sekaisin. Tahkojen pyörittely siten, että värit saadaan taas kohdalleen, vaatii hieman harjaantumista, vaikkei $2 \times 2 \times 2$ tapauksessa olekaan ylettömän vaikeaa. Kuution "standardikokohan" on $3 \times 3 \times 3$.

Kuution tila-avaruuden muodostamiseksi kannattaa vakioida kuution avaruudellinen asento. Asia on kenties helpoin hahmottaa ajattelemalla, että kuutio asetetaan pöydälle aina siten, että puna-sini-keltainen nurkka on vasemmalla takana alhaalla punainen puoli pöytää vasten. Näin toimien kuution tilat muodostuvat siitä, että muut seitsemän nurkkaa vaihtavat keskinäistä järjestystään ja pyörivät kuution keskustasta nurkkaan kulkevan akselin ympäri.

Nurkat saadaan kaikkiaan $7! = 5040$ järjestykseen. Koska nurkka voi olla kolmessa asennossa, saadaan kuution tilojen määrälle yläraja $7! \cdot 3^7$. Jokainen Rubikin kuutiota tarpeeksi paljon pyöritellyt tietää, että nurkkien asentoja rajaa kuitenkin eräs säännönmukaisuus. Nimittäin viimeisen nurkan asento määräytyy muiden nurkkien asennoista. Sen ansiosta kuutiolla on kaikkiaan $7! \cdot 3^6$ eli mainitut 3 674 160 tilaa.

Kuution tilan voi esittää esimerkiksi luettelemalla eri suunnissa näkyvät värit. Eri värejä on kuusi, joten värin esittämiseen riittää kolme bittiä. Tahkojen neljänneksiä on 24 kappaletta, mutta vakioidun nurkan kolmen neljänneksen värejä ei tarvitse esittää, koska ne eivät muutu. Niinpä tilan esittämiseen tällä tavalla tarvitaan $21 \cdot 3 = 63$ bittiä.

63 bitillä voi muodostaa $2^{63} \approx 10^{19}$ bittiyhdistelmää. Tämä on valtavasti enemmän kuin todellisuudessa mahdolliset 3 674 160 tilaa. Osalla "ylimääräisistä" bittiyhdistelmistä on mielekäs tulkinta: noin seitsemän miljoonaa niistä voidaan tuottaa purkamalla kuutio osiin ja kokoamalla se uudelleen, ja huomattavasti enemmän saadaan maalaamalla sivuja



Kuva 2: Vasen: palojen paikkojen numerointi
Oikea: asennon muutos pyöräytyksessä

uudelleen. Niitä bittiyhdistelmiä, joissa jonkin tahkon neljänneksen värin koodi ei ole mikään käytössä olevista kuudesta, ei voi tulkita ollenkaan — tai ehkä voi ajatella, että niissä käytetään seitsemättä ja mahdollisesti kahdeksatta väriä.

Ylimääräisten bittiyhdistelmien tulkinta ei ole tämän tarinan kannalta olennaista. Mutta se on tärkeää, että niitä on olemassa. Niitä bittiyhdistelmiä, joihin tutkittava järjestelmä voi joutua, kutsutaan usein *saavutettaviksi* tiloiksi. Kaikkien bittiyhdistelmien joukolla ei ole vakiintunutta nimitystä, mutta kutsun sitä *syntaktisesti mahdollisten* tilojen joukoksi. On hyvin tavallista tila-avaruussovelluksissa, että syntaktisesti mahdollisten tilojen joukko on paljon suurempi kuin saavutettavien tilojen joukko.

Syntaktisesti mahdollisten tilojen joukko riippuu tilan esitystavasta. $2 \times 2 \times 2$ Rubikin kuution tila voidaan esittää huomattavasti tiiviimmin kuin yllä tehtiin. Paloille mahdolliset paikat voidaan numeroida $0, \ldots, 6$ ja asennot $0, \ldots, 2$. Nämä voidaan yhdistää yhdeksi, välillä $0, \ldots, 20$ olevaksi koodiksi esimerkiksi lausekkeella $3 \cdot \mathit{paikka} + \mathit{asento}$. Kun näin saatu koodi jaetaan kolmella, ilmaisee osamäärä paikan ja jakojäännös asennon. Ne saadaan siis helposti selville koodista.

Kuvan 2 vasen puoli näyttää paikoille valitsemani numerot. Vakioitu nurkka on takana piilossa.

Asentojen koodaus on hankalampi selittää. Järjestyksessä olevan kuution jokainen pala on asennossa numero 0. Kuvan 2 oikea puoli näyttää, miten asennon numero muuttuu, kun pala siirtyy vastapäivään pyöräytyksessä paikasta toiseen. “$+$” tarkoittaa, että asennon numeroa kasvatetaan yhdellä, ja “$-$” tarkoittaa vastaavasti vähennystä yhdellä. Lopputulokseen lisätään tarvittaessa 3 tai -3, jotta tulos olisi aina välillä $0, \ldots, 2$. Myötäpäivään pyöräytettäessä plussat tulee vaihtaa miinuksiksi ja päinvastoin.

Koska jokaisessa pyöräytyksessä kahden palan asento kasvaa yhdellä ja kahden vähenee yhdellä, pysyy asentojen summa

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4	5	3	11	9	10	1	2	0	8	6	7	12	13	14	15	16	17	18	19	20
13	14	12	1	2	0	6	7	8	9	10	11	17	15	16	5	3	4	18	19	20
7	8	6	3	4	5	20	18	19	9	10	11	1	2	0	15	16	17	14	12	13

Taulukko 1: Kolme pyöräytystaulukkoa:
etutahko, vasen tahko ja ylätahko vastapäivään.

aina kolmella jaollisena. Tämä osoittaa todeksi sen aiemmin mainitsemani seikan, että viimeisen nurkan asento määräytyy muiden nurkkien asennoista.

Seitsemästä liikkuvasta palasta saatavat koodit $k_0, \ldots, k_6$ voidaan yhdistää yhdeksi luvuksi kaavalla $\sum_{i=0}^{6} k_i \cdot 21^i$. Näin saatava luku on enintään noin $1{,}8 \cdot 10^9$, ja on esitettävissä 31 bitillä. Jos tila esitään näin ja sille varataan 31 bittiä, on syntaktisesti mahdollisia tiloja enää $2^{31} \approx 2{,}1 \cdot 10^9$ kappaletta.

Jälkimmäisellä esitystavalla on kaksi etua edelliseen nähden. Jos yksittäinen muistin tarve ylittää 16 bittiä eli kaksi tavua, kuten tässä on tilanne, niin muistia kannattaa tavallisissa nykyisissä tietokoneissa jakaa neljän tavun ryhmissä. Edellisellä esitystavalla kaikkien tilojen esittämiseen tarvittaisiin lähes 30 miljoonaa tavua, mutta jälkimmäisellä esitystavalla riittää puolet siitä. (Tämä laskelma ei muuttuisi olennaisesti, vaikka muistia jaettaisiin biteittäinkin.) Tulemme tosin jatkossa huomaamaan, että tällaisen laskelman antama kuva muistin tarpeesta ei ole yleispätevä; mutta silti jatkossakin pätee, että pienempi esitystapa säästää muistia.

Jälkimmäisen esitystavan toinen etu on, että se sallii pyöräytysten vaikutusten laskemisen hyvin tehokkaasti. Tämä on tärkeää, koska tila-avaruuden muodostamisen aikana lasketaan yli 22 miljoonaa pyöräytystä.

Erilaisia “peruspyöräytyksiä” on kuudenlaisia: voidaan pyörittää etutahkoa, ylätahkoa tai oikean puoleista tahkoa neljänneskierroksen verran myötä- tai vastapäivään. Muita tahkoja ei pyöritetä, koska vasen alatakanurkka pidetään paikallaan. Puolen kierroksen mittaiset pyöräytykset saadaan toistamalla peruspyöräytys kahdesti. (Ne voitaisiin haluttaessa mallittaa erikseen.)

Muistamme, että nurkan asento ja sijainti esitetään välillä 0, …, 20 olevalla koodilla. Kukin peruspyöräytys voidaan esittää 21-alkioisella taulukolla, joka ilmoittaa nurkan pyöräytyksen jälkeistä asentoa ja sijaintia kuvaavan koodin samat tiedot ennen pyöräytystä ilmoittavan koodin funktiona. Pyöräytyksen vaikutus lasketaan muuntamalla jokaisen liikkuvan nurkan koodi pyöräytystä esittävän taulukon avulla.

Taulukossa 1 on annettu kolme pyöräytystaulukkoa. Loput kolme saadaan

niiden käänteispermutaatioina. Esimerkiksi ylimmän rivin mukainen pyöräytys siirtää paikassa 1 asennossa 2 olevan nurkan paikkaan 3 asentoon 1, koska $3 \cdot 1 + 2 = 5$, sarakkeessa 5 lukee “10”, ja 10 modulo $3 = 1$ ja $\lfloor 10/3 \rfloor = 3$. Ylin rivi vastaa siis kuvan 2 etutahkon pyöräytystä vastapäivään. Keskimmäinen rivi esittää vasemmanpuoleisen tahkon ja alarivi ylätahkon pyöräytystä vastapäivään.

Tilan 31-bittinenkin esitys sisältää jotakin turhaa, koska vain joka 584:s syntaktisesti mahdollinen tila on saavutettava. Tila-avaruustehtävissä on tavallista, että tilaa ei osata esittää täydellisen tiivisti. Tulemme näkemään, että $2 \times 2 \times 2$ Rubikin kuution tapauksessa se osataan, mutta — yllättävää kyllä — se ei kannata.

3 Ensimmäinen ohjelma

Tavoitteenani oli laatia ohjelma, joka muodostaisi $2 \times 2 \times 2$ Rubikin kuution kaikki 3 674 160 tilaa niin sanotussa *leveyteen ensin* -järjestyksessä. Jokaisesta tilasta tuli tallettaa tieto, minkä suuntaisella pyöräytyksellä se löydettiin ensimmäisen kerran.

Näin syntyvää tietorakennetta voidaan käyttää muun muassa sotketun kuution ratkaisemiseen. Nimittäin, sotkettu kuutio saadaan askeleen verran lähemmäs järjestettyä tilaa etsimällä sotkettua tilaa edustava tila tietorakenteesta (mikä voidaan tehdä tehokkaasti); katsomalla, minkä suuntaisella pyöräytyksellä se löydettiin; ja tekemällä pyöräytys vastakkaiseen suuntaan. Tämä voidaan toistaa pyöräytyksen lopputuloksena syntyvälle tilalle ja niin edelleen kunnes kuutio on järjestyksessä. Tällä tavalla löydettävä järjestyksen palauttava pyöräytyssarja on niin lyhyt kuin mahdollista. Tietorakennetta voi käyttää myös tila-avaruuden rakenteen tutkimiseen, mistä on esimerkki luvussa 9.

Leveyteen ensin -järjestyksessä toimivan, kaikki tilat muodostavan ohjelman toimintaperiaate on esitetty kuvassa 3. “Alkutila” on järjestyksessä olevaa kuutiota edustava tila. Ohjelma käyttää kahta tietorakennetta: löydettyjen tilojen joukko sekä jono, jossa ovat ne löydetyt tilat, joista käsin ei vielä ole tutkittu pyöräytyksiä. Löydettyjen tilojen joukon yhteydessä talletetaan myös tieto, minkäsuuntaisella pyöräytyksellä tila löytyi.

Tilat esitetään tietorakenteissa lausekkeen $\sum_{i=0}^{6} k_i \cdot 21^i$ mukaisesti edellä kuvatulla tavalla pakattuina. Pyöräytyksen laskemista varten tila puretaan, eli selvitetään luvut $k_0, \ldots, k_6$. Tämä tapahtuu jakamalla tila toistuvasti 21:llä ja ottamalla jakojäännökset talteen. Pyöräytyksen lopputuloksena syntyvän tilan osat saadaan kaavasta $k'_i := P[k_i]$, missä P on edellä mainittu pyöräytystä kuvaava 21-alkioinen taulukko. Uusi tila pakataan kaavalla $\sum_{i=0}^{6} k'_i \cdot 21^i$.

Jono on tietorakenne, jossa alkiot ovat ja josta ne poistetaan siinä järjestyksessä kuin ne sinne lisättiin. Jonon ensimmäisenä on siis kauimmin jonossa ollut alkio, jota ei ole vielä otettu pois.

On tavallista, että jonoon ei laiteta tiloja sellaisenaan, vaan osoittimia tai viittei-

pakkaa alkutila ja pistä se jonoon sekä löydettyihin tiloihin
toista kunnes jono on tyhjä
    ota jonosta ensimmäinen tila t ja poista se jonosta
    pura t
    jokaiselle pyöräytyssuunnalle
        t' := pyöräytetty t
        pakkaa t'
        jos t' ei ole löydetyissä tiloissa **niin**
            lisää t' löydettyihin tiloihin ja talleta pyöräytyksen suunta
            lisää t' jonon perään

Kuva 3: Tilojen muodostus leveyteen ensin -järjestyksessä

tä niihin kohtiin löydettyjen tilojen joukkoa, joissa jonoon kuuluvat tilat sijaitsevat. Näin tehdään siksi, että yksittäinen tila vie tyypillisesti enemmän muistia kuin osoitin. Tässä tapauksessa kuitenkin tila mahtuu neljään tavuun, minkä myös osoitin tyypillisesti vie, joten osoittimien käytöllä ei saavutettaisi säästöä. Koska tilojen käyttö sellaisenaan on helpompaa ja tällaisessa tilanteessa myös nopeampaa, tein niin.

(Jono voidaan toteuttaa myös esimerkiksi linkittämällä tilatietueita listaksi tätä tarkoitusta varten varatun ylimääräisen osoittimen avulla, mutta se veisi ainakin saman verran muistia.)

Etukäteen tiesin, että jonon maksimipituus jää alle tilojen määrän, joten vajaa 15 miljoonaa tavua riittää varmasti jonon tallettamiseen. Jälkikäteen selvitin, että jonon maksimipituus on 1 499 111, joten hyvä jonon toteutus käyttää alle 6 miljoonaa tavua.

Jono löytyy valmiina tietorakenteena (tai oikeastaan niin sanottuna säiliösovittimena) ohjelmointikielen C++ standardikirjastosta.[2] Käytin laiskuuttani ohjelmani ensimmäisessä versiossa hyväksi C++:n standardikirjaston palveluita missä vain mahdollista. Ohjelmani tarpeet täyttävä, nopea ja muistia säästeliäästi käyttävä jonon toteutus olisi erittäin helppo tehdä itse. Missään vaiheessa ei kuitenkaan ilmennyt selvää syytä tehdä niin, joten käytin C++:n standardikirjaston jonoa myös myöhemmissä versioissa.

Löydettyjen tilojen joukon kohdalla kävi toisin. Ohjelmani ensimmäisessä versiossa käytin C++:n standardikirjaston `map`-rakennetta. Standardi ei määrittele, miten `map` on toteutettava, mutta käytännössä se on niin sanottu puna-musta binäärihakupuu.

Kun koeajoin ohjelmani ensimmäistä versiota läppärissäni, alkoi hetken kulut-

[2] C++:n standardikirjaston tietorakenneosuudesta käytetään historiallisista syistä usein lyhennettä “STL”. Lyhenne on sisällöllisesti harhaanjohtava, eikä standardi itse tunne sitä.

tua kiintolevyasemasta kuulua “rapi–rapi–rapi ...”, ja välitulostusten tahti oli kuin hidastetussa filmissä. Silloisessa läppärissäni oli 42 miljoonaa tavua (eli 40 megatavua, missä “mega” on 2^{20}) keskusmuistia — eikö se riittänytkään?

Puna-mustan puun jokaisessa tietueessa on hyötykuorman lisäksi kaksi osoitinta muihin tietueisiin, sekä yksi bitti, joka ilmaisee väriä (punainen tai musta). Värien avulla estetään puun kasvaminen vinoon, mikä on tarpeen, että puna-mustaa puuta käsittelevät operaatiot olisivat aina nopeita. Tietueessa oli siis ainakin $2 \cdot 4$ tavua osoittimia ja 4 tavua hyötykuormaa ja väriä, yhteensä 12 tavua. Tämä kerrottuna saavutettavien tilojen määrällä tuottaa tulokseksi yli 44 miljoonaa tavua.

Seuraavaksi kokeilin ohjelmaa järeämmässä koneessa työpaikallani. Tulos: taas rapi–rapi–rapi. Vaikka kyseisessä koneessa ei ylettömän paljoa vapaata keskusmuistia ollutkaan, piti sitä kuitenkin olla selvästi enemmän kuin 44 miljoonaa tavua.

Ohjelman varaamien muistipaikkojen osoitteiden analysointi paljasti, että jokainen tietue veikin 24 tavua. C++-guru Matti Rintalan juttusilla käynti paljasti miksi näin on. C++:n `map` on toteutettu niin sanottuna muottina (template), mikä tarkoittaa, että kääntäjä selvittää hyötykuorman tyypin käännösaikana. Tietyntyyppisten tietojen käsittelyä nopeuttaa, jos ne sijoitetaan neljällä tai jopa kahdeksalla jaollisiin muistiosoitteisiin. Koska `map`-rakenteen tekijä joutui varautumaan kaikkiin tietotyyppeihin, hän sijoitti hyötykuorman kahdeksalla jaolliseen osoitteeseen, ja varasi sille muistia kahdeksan tavua kerrallaan.

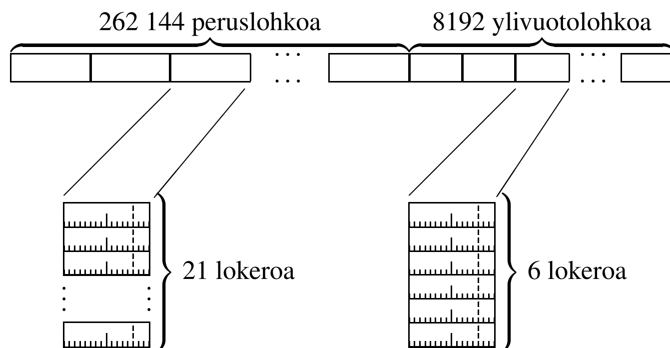
Niinpä pakatun tilan 31-bittinen esitys paisui kahdeksaksi tavuksi. Tietueen lisäksi sisältämät kaksi osoitinta ja yksi väribitti veivät yhteensä kahdeksan tavua ja yhden bitin, joten ne eivät mahtuneet kahdeksaan tavuun. Niille varattiin 16 tavua, ja koko tietueen kooksi tuli 24 tavua.

Löydettyjen tilojen tietorakenne käytti siis lähes 90 miljoonaa tavua muistia. Se tuntui kovin paljolta alle 15 miljoonan tavun hyötykuorman tallettamiseen, joten aloin miettiä tehokkaampaa talletustapaa.

4 Rubikin kuutiolle optimoitu hajautustaulu

Ketjutettu hajautustaulu on tehokas ja usein käytetty tapa esittää joukkoja ja joukon alkioihin liittyviä oheistietoja tietokoneen muistissa. Ketjutettu hajautustaulu muodostuu suurehkosta määrästä linkitettyjä listoja. Listan tietueessa on tyypillisesti yksi alkio, siihen liittyvät oheistiedot sekä linkki listan seuraavaan alkioon.

Lista, johon alkio kuuluu selvitetään *hajautusfunktion* avulla. Hajautusfunktio on hajautustaulua perustettaessa määrätty funktio, joka ottaa alkion arvon, ja tuottaa kokonaisluvun, joka on välillä $0, \ldots, \mathit{listojen\ määrä} - 1$. Listojen määrä on huomattavasti pienempi kuin syntaktisesti mahdollisten alkioiden määrä. (Jos syntaktisten alkioiden määrä on niin pieni, että on mahdollista ottaa käyttöön taulukko, jossa



Kuva 4: $2 \times 2 \times 2$ Rubikin kuutiolle optimoitu hajautustaulu

on jokaista syntaktista alkiota kohti yksi osoitin, niin ei kannata käyttää hajautustaulua.)

Alkio talletetaan listan tietueeseen tavallisesti kokonaisuudessaan. Jo Knuth huomautti, että se ei kuitenkaan ole periaatteessa tarpeen — osa alkiota koskevasta informaatiosta on saman listan alkioilla sama, ja sitä osaa ei tarvitse välttämättä tallettaa [4, luvun 6.4 tehtävä 13, s. 543].

Alkion jako osiin on kuitenkin jossain määrin kömpelöä, ja siten saatava säästö on usein merkityksetön verrattuna tietueessa olevaan muuhun tietoon. Nimittäin, listoja voi tavallisessa tietokoneessa olla enintään miljoonia, joten säästö jää alle 30 bitin. Koska tietueessa on linkki seuraavaan, linkki (teknisesti osoitin) on nykyisin yleensä 32-bittinen, ja tietueessa on jotain hyötykuormaakin, jää säästö vääjäämättä reilusti alle puoleen tietueen koosta.

$2 \times 2 \times 2$ Rubikin kuution tapauksessa hyötykuormaa on kuitenkin vähän, joten alkion jakamisella on mahdollista saavuttaa merkittävää säästöä, varsinkin jos linkeillekin tehdään jotakin. Linkit kuluttavat huomattavasti muistia, mutta sitä on mahdollista vähentää sijoittamalla samaan tietueeseen useita alkioita ja käyttämällä linkkeinä osoittimien sijasta numeroita.

Päädyin lopulta tietorakenteeseen, jota kuvailen seuraavaksi ja havainnollistan kuvassa 4.

Yksi hajautusfunktiolle asetettavista tavoitteista on hajauttaa aineisto eri listoihin mahdollisimman tasaisesti. Tämän varmistamiseksi usein suositellaan, että hajautusfunktion arvo riippuu kaikista alkion biteistä. Siksi tietorakenne aloittaa sotkemalla tilan 31-bittisen esityksen yksinkertaisilla aritmeettisilla ja bittioperaatioilla. Käytin kuvassa 5 esitettyä C++-koodinpätkää. Sitä suunnitellessani varmistin, että se kuvaa eri tilat eri tiloiksi ja jokainen bitti pääsee vaikuttamaan vähiten merkitseviin bitteihin, mutta muuten vedin sen hihasta.

```
tila ^= tila >> bitteja_indeksissa;
tila *= 1234567; tila += 5555555;
tila &= 0x7fffffff;
tila ^= tila >> bitteja_indeksissa;
tila *= 1234567; tila += 5555555;
tila &= 0x7fffffff;
```

Kuva 5: Tilan sotkeminen

Sotkemisen olisi epäilemättä voinut tehdä fiksumminkin. Luvussa 6 tarkastellaan, onko tämä sotkemistapa riittävän hyvä.

Tietorakenne käyttää listan valitsemiseen sotketun esityksen 18 vähiten merkitsevää bittiä. Tätä osaa kutsun *indeksiksi*. Listoja on siis $2^{18} = 262\ 144$ kappaletta. Loput 13 bittiä muodostavat *tilan loppuosan*. Se talletetaan listan tietueeseen. Perustelen, miksi valitsin jaoksi nimenomaan 18+13 sen jälkeen, kun olen esitellyt tietueiden rakenteen.

Linkkien määrän vähentämiseksi lista koostuu kahdenlaisista *lohkoista*: *peruslohkoista*, joihin mahtuu 21 tilan loppuosaa kuhunkin, ja *ylivuotolohkoista* kooltaan 6 tilan loppuosaa. Peruslohkoja on yksi listaa kohti, eli kaikkiaan 262 144. Valitsin peruslohkon ja ylivuotolohkon koot kokeilemalla. Ylivuotolohkojen määräksi asetin 8192, koska se on suurin määrä, minkä alempana kuvailemani rakenne sallii. Hieman vähempi riittäisi, mutta ylivuotolohkojen osuus muistin kulutuksesta on alle prosentin, joten niiden määrää vähentämällä saavutettava muistin säästö olisi merkityksetön.

Linkin koon pienentämiseksi lohkot talletetaan taulukoihin. Näin linkit voidaan esittää lukuina, eikä tarvitse käyttää 32-bittisiä osoittimia. On siis 262 144-alkioinen taulukko peruslohkoja ja 8192-alkioinen taulukko ylivuotolohkoja. Peruslohko valitaan indeksoimalla peruslohkotaulukkoa tilasta edellä kuvatulla tavalla saatavalla 18-bittisellä indeksillä. Ylivuotolohkon indeksi on 13-bittinen, kuten kohta nähdään.

Kukin lohko jakautuu kaksitavuisiin eli 16-bittisiin lokeroihin. Lokeron biteistä kolme käytetään kuvaamaan lokeron tilannetta. (Mieleni olisi tehnyt puhua lokeron “tilasta”, mutta sana “tila” on jo varattu tarkoittamaan Rubikin kuution tilaa.) Tilannekoodilla on kahdeksan arvoa: “tyhjä”, “ylivuoto” ja kuusi erilaista “käytössä”-arvoa.

Jos lokero on käytössä, niin loput 13 sen biteistä tallettavat tilan loppuosan, ja valittu “käytössä”-arvo kertoo, millä peruspyöräytyksellä tila alun perin löytyi. Tyhjän lokeron tapauksessa loppujen 13 bitin sisällöllä ei ole merkitystä. Ylivuotolokeron loput 13 bittiä sisältävät ylivuotolohkon numeron.

Nyt näkyy, miksi ylivuotolohkoja on enintään 8192 kappaletta: ylivuotolinkin tallettamiseen oli kätevästi tarjolla 13-bittinen muistialue, ja 13 bitillä voi esittää enintään 8192 eri arvoa.

(Itse asiassa ohjelmassa ei ole erillisiä taulukoita perus- ja ylivuotolohkoja varten, vaan yksi taulukko, jossa on $262\,144 \cdot 21 + 8192 \cdot 6$ eli 5 554 176 lokeroa. Taulukon jakautuminen perus- ja ylivuoto-osuuteen ja edelleen lohkoihin näkyy vain siinä, miten ohjelmakoodi käyttää taulukkoa.)

Listaan lisättävät tilat talletetaan peruslohkoon sen alusta alkaen kunnes peruslohko täyttyy. Jos sen jälkeen vielä tulee tila, niin otetaan käyttöön ylivuotolohko. Peruslohkon viimeinen tila siirretään ylivuotolohkon ensimmäiseen lokeroon; uusi tila talletetaan ylivuotolohkon toiseen lokeroon; ja peruslohkon viimeinen lokero muutetaan ylivuotolokeroksi, jonne talletetaan ylivuotolohkon numero. Tarvittaessa listaa jatketaan toiseen ylivuotolohkoon ja niin edelleen. Ylivuotolohkot otetaan käyttöön numerojärjestyksessä.

Tilaa etsitään tietorakenteesta valitsemalla sen indeksin mukainen peruslohko, ja tutkimalla sen lokeroita peräjälkeen kunnes etsityn tilan loppuosa löytyy, tai kohdataan tyhjä lokero tai ylivuotolokero. Viimeksi mainitussa tapauksessa siirrytään lokeron ilmoittaman ylivuotolohkon alkuun ja jatketaan etsintää sieltä.

Tässä vaiheessa on helppo nähdä, miksi jaoin sotketun tilan 18-bittiseen indeksiin ja 13-bittiseen loppuosaan. Lokeron koon kannattaa olla tavun monikerta, koska muutoin jouduttaisiin etsimään tilan loppuosia vaihtelevista kohdista tietokoneen tavuja, mikä olisi hidasta ja hankalaa. Kolme bittiä riittää esittämään pyöräytyssuuntatiedon ja jättää juuri sopivasti kaksi ylimääräistä arvoa, joita saatoin käyttää tarkoittamaan tyhjää lokeroa ja ylivuotoa.

Kaksitavuisilla lokeroilla tilan loppuosaa varten jää 13 bittiä. Tällöin listoja tulee mainitut 262 144 kappaletta ja listan keskipituudeksi tulee noin 14, mitkä molemmat ovat hyviä määriä. Jos lokerossa olisi vain yksi tavu, olisi listoja noin 67 miljoonaa, mikä olisi suurta tuhlausta, koska talletettavia tiloja on vain noin 3,7 miljoonaa. Jos taas lokerolle varattaisiin kolme tavua, tulisi listoista hyvin pitkiä (keskimäärin 3588 tilaa), jolloin listan selaaminen tilan löytämiseksi olisi hidasta.

5 Muistin kulutuksen analysointia

Edellä kuvaamani hajautustaulu käyttää hieman yli 11 miljoonaa tavua muistia. Koska `map`-pohjainen tietorakenne kulutti lähes 90 miljoonaa tavua, on parannus huomattava.

Uusi ohjelma toimii moitteettomasti nyt jo ikivanhassa läppärissäni, eikä kiintolevyasema rapise. Kaikkien saavutettavien tilojen muodostaminen kestää läppärissäni noin 105 sekuntia.

Työpaikan järeämpi kone on vaihtunut sitten ensimmäisten kokeiden, joten en enää voi tehdä sillä mittauksia.

Nykyisellä työpaikan koneella olen saanut kahdenlaisia, keskenään odottamattoman erilaisia mittaustuloksia, koska C++-kääntäjäkin vaihtui. Nykyinen kone jaksaa ajaa C++:n map-rakenteeseen perustuvan ohjelman, ja käyttää siihen nykyisellä kääntäjällä 98 sekuntia (aiemmalla kääntäjällä 106 sekuntia). Kiintolevy ei rapise, mutta muista seikoista huomaa, että muisti alkaa käydä vähiin. Sama kone ajaa uuden ohjelmani noin 20 sekunnissa (aiemmin 23).

Kuten edellä kerroin, valitsin peruslohkon ja ylivuotolohkon koot kokeilemalla. En enää muista, kuinka monta koetta tarvitsin ja kauanko siinä meni, mutta muistelen, että löysin pian arvot, joilla tila-avaruuden muodostaminen onnistui. Tämä on helppo uskoa, koska listan keskipituus oli etukäteen tiedossa. Jos peruslohkon kooksi asetetaan se kerrottuna kahdella eli 28, niin valtaosa listoista mahtuu peruslohkoonsa. Koska peruslohkoja on 262 144 kappaletta ja ylivuotolohkoja vain 8192 kappaletta, ei ylivuotolohkon koolla ole suurta merkitystä muistin kulutuksen kannalta. Laitetaan kooksi vaikka sama 28.

Näillä arvoilla muistin kulutus on runsaat 15 miljoonaa tavua. Se mahtuu läppäriini hyvin: ajoaika on 106 sekuntia.

Sen jälkeen oli helppo etsiä haarukoimalla optimaalinen arvo ensin peruslohkon koolle ja sitten ylivuotolohkon koolle.

Laskeskellessani tietorakenteeni muistinkulutusta huomasin hätkähdyttävän seikan. Vaikka $2 \times 2 \times 2$ Rubikin kuution tilojen joukko on kaikkea muuta kuin satunnainen, voidaan talletetun joukon alkiot olettaa satunnaisiksi, koska tietorakenne aloittaa tilan käsittelyn sotkemalla sen. Tietorakenteeni tallettama joukko sisältää siis 3 674 160 kappaletta satunnaisia 31-bittisiä alkioita, mikä tekee yhteensä noin 114 miljoonaa bittiä eli 14,2 miljoonaa tavua informaatiota. Se on enemmän kuin tietorakenteeni käyttämät 11,1 miljoonaa tavua! Sitäpaitsi tietorakenteeni tallettaa joukon lisäksi myös pyöräytyssuuntia.

Miten on mahdollista tallettaa joukko pienemmässä määrässä muistia kuin sen informaatiosisältö on? Ei mitenkään. Talletetun informaation määrä on todellisuudessa paljon vähemmän kuin 14,2 miljoonaa tavua.

Laskelma on kyllä sikäli oikein, että mikä tahansa tietorakenne, joka tallettaa jokaisen alkion sellaisenaan erikseen, käyttää ainakin mainitut 14,2 miljoonaa tavua. Tavalliset hajautustaulut, binäärihakupuut jne. toimivat niin, joten niitä käyttämällä ei voi päästä alle 14,2 miljoonan tavun. Koska ne tarvitsevat muistia hyötykuorman lisäksi osoittimiin, joilla rakenne pidetään koossa, on niiden muistinkulutus itse asiassa paljon suurempi, mistä näimme esimerkin tämän tarinan alkupuolella.

Jos alkiot sijoitetaan peräkanaa isoon taulukkoon, niin osoittimia ei tarvita ja 14,2 miljoonaa tavua muistia riittää. Valitettavasti sellaisesta rakenteesta etsiminen on toivottoman hidasta (paitsi jos taulukko pidetään järjestyksessä, jolloin siihen lisääminen on toivottoman hidasta).

Mutta 14,2 miljoonan tavun alle on siis mahdollista päästä, vieläpä nopeal-

la tietorakenteella. Jos 14,2 miljoonaa tavua ei ole todellinen muistin käytön informaatioteoreettinen alaraja, niin mikä sitten on?

Todellinen alaraja on

$$\log_2 \frac{u!}{n!(u-n)!} \qquad (1)$$

bittiä, missä $u = 2^{31} = 2\ 147\ 483\ 648$ on kaikkien 31-bittisten lukujen määrä, ja $n = 3\ 674\ 160$ on talletettavan joukon koko. Se on sen luvun 2-kantainen logaritmi, joka ilmoittaa, kuinka monta erilaista n-alkioista joukkoa voidaan muodostaa u-alkioisesta perusjoukosta.

2 147 483 648! on sangen epämiellyttävä laskea, mutta hyödyntämällä Abraham de Moivren 1730 julkaisemaa kaavaa — joka tunnetaan Stirlingin kaavana James Stirlingin mukaan [2, s. 599] — saadaan (1):lle sangen tarkka likiarvo

$$nw' + 1{,}44n\ ,$$

missä $w' = \log_2 u - \log_2 n$.

Luvulla w' on mielekäs tulkinta: Merkitään alkion bittien määrää symbolilla w sanasta "width" (siis $w = \log_2 u$). Jotta voisi olla olemassa n erilaista alkiota, on alkiossa oltava ainakin $\log_2 n$ bittiä, eli on oltava $w \geq \log_2 n$. w' on tämän lisäksi alkiossa olevien, ikään kuin "ylimääräisten" bittien määrä. Kaava sanoo, että alkiota kohti on välttämätöntä tallettaa ainakin sen "ylimääräiset" bitit sekä noin 1,44 bittiä. w' on muistin kulutuksen analyysissä parempi parametri kuin w, koska n ja w' voivat saada arvonsa ja lähestyä ääretöntä toisistaan riippumatta, mutta n ei voi kasvaa rajatta ellei w kasva rajatta.

Kaavan antama likiarvo on $2 \times 2 \times 2$ Rubikin kuution tapauksessa noin 4,88 miljoonaa tavua.

Kaava voidaan johtaa seuraavasti.

Tylsä laskelma. Kirjan [1] kaava 6.1.38 (Stirlingin kaavan eräs versio) sanoo, että kun $x > 0$, niin

$$x! = \sqrt{2\pi} x^{x+\frac{1}{2}} e^{-x+\frac{\theta}{12x}}\ , \qquad (2)$$

missä $0 < \theta < 1$. Niinpä $\log_2 n! \geq n\log_2 n - n\log_2 e$, ja edelleen

$$\begin{aligned}
&\log_2 \frac{u!}{n!(u-n)!} \\
=\ &\log_2 \frac{u!}{(u-n)!} - \log_2 n! \\
\leq\ &\log_2 u^n - n\log_2 n + n\log_2 e \\
=\ &n(\log_2 u - \log_2 n) + n\log_2 e. \qquad (3)
\end{aligned}$$

Toisaalta

$$\begin{aligned}
&\log_2 \frac{u!}{n!(u-n)!} \\
=\ &\log_2 \left(\frac{u}{n} \cdot \frac{u-1}{n-1} \cdot \ldots \cdot \frac{u-n+1}{n-n+1} \right) \\
\geq\ &\log_2 \left(\frac{u}{n} \right)^n = n(\log_2 u - \log_2 n).
\end{aligned}$$

Merkitsemällä $w' = \log_2 u - \log_2 n$ ja käyttämällä $\log_2 e$:lle likiarvoa 1,443 saadaan edelliset muotoon

$$nw' \leq (1) \leq nw' + 1{,}443n.$$

Kaavaa (3) johdettaessa tehdyistä likiarvoistuksista merkittävin on lausekkeen $\frac{u!}{(u-n)!}$ korvaaminen ylälikiarvolla

u^n. Sen vaikutusta voi arvioida tekemällä vastaavan likiarvoistuksen toisin päin, eli laskemalla myös lausekkeen $(u-n+1)^n$, ja vertaamalla tulosten 2-kantaisia logaritmeja. Näin nähdään, että käyttämillämme n:n ja u:n arvoilla on virhe alle 10 000 bittiä. Toinen likiarvoistus on kaavan (2) tuottamien vähennettävien $\log_2 \sqrt{2\pi}$, $\frac{1}{2}\log_2 n$ ja $\frac{\theta}{12n}\log_2 e$ poisjättäminen. Tästä aiheutuva virhe on hyvin pieni, noin 12 bittiä.

Kaavalla (3) saatava ylälikiarvo 4,884 miljoonaa tavua poikkeaa siis oikeasta arvosta alle 0,002 miljoonaa tavua.

Pyöräytyssuunnat sisältävät yhteensä $\log_2 6^{3\,674\,160}$ bittiä $\approx 1{,}19$ miljoonaa tavua informaatiota, joten kaiken kaikkiaan tarvitaan vähintään 6,07 miljoonaa tavua. Tämä on todellinen ehdoton alaraja, johon käytettyä 11,1 miljoonaa tavua on mielekästä verrata.

Jos tarkkoja ollaan, niin laskelmassa pitäisi ottaa huomioon myös se, että lopullinen joukko ei ilmesty kerralla valmiina, vaan muodostetaan lisäämällä alkioita yksi kerrallaan. Pitäisi siis laskea, montako *enintään* n-alkioista joukkoa on olemassa, ja ottaa tämän luvun logaritmi:

$$\log_2 \sum_{k=0}^{n} \frac{u!}{k!(u-k)!}\,. \tag{4}$$

Näin saatava luku poikkeaa kuitenkin luvusta (1) vähemmän kuin yhden bitin.

Tylsä laskelma. Nimittäin, tapauksessamme pätee $0 < n < \frac{u+1}{3}$. Niinpä

$$\begin{aligned} & \frac{u!}{(k-1)!(u-(k-1))!} \\ = {} & \frac{k}{u-k+1} \cdot \frac{u!}{k!(u-k)!} \\ < {} & \frac{1}{2} \cdot \frac{u!}{k!(u-k)!} \end{aligned}$$

kun $1 \leq k \leq n$, joten

$$\begin{aligned} & \sum_{k=0}^{n} \frac{u!}{k!(u-k)!} \\ < {} & \left(\cdots + \frac{1}{2^2} + \frac{1}{2^1} + \frac{1}{2^0}\right) \cdot \frac{u!}{n!(u-n)!} \\ = {} & 2 \cdot \frac{u!}{n!(u-n)!}\,. \end{aligned}$$

On ehkä hyvä korostaa, että kaavojen (1) ja (4) ilmaisemat alarajat rajoittavat *keskimääräistä* tapausta. Ne eivät tee mahdottomaksi päästä *yksittäisissä tapauksissa* huomattavasti rajan alle. Ne kuitenkin takaavat, että sellaiset tapaukset ovat äärimmäisen harvinaisia. Oli joukon esitystapa ihan mikä tahansa, niin jos valitaan joukko satunnaisesti, niin muistin kulutus kyseisellä esitystavalla on ainakin teoreettisen alarajan suuruinen hyvin suurella todennäköisyydellä vaikka ei täysin varmasti.

Mainittakoon kuriositeettina, että on olemassa hyvin yksinkertainen mutta toivottoman hidas tietorakenne, jolla päästään varsin lähelle teoreettista alarajaa. Sen muistinkulutus on välillä $nw' + 1{,}91n$, $\ldots$, $nw' + 2n$ riippuen siitä, miten w' katkeaa kokonaisluvuksi.

Alkio jaetaan indeksiin ja loppuosaan kuten $2 \times 2 \times 2$ Rubikin kuutiolle optimoidussa hajautustaulussakin. Indeksin

koko on $\lceil \log_2 n \rceil$ ja loppuosan $\lfloor w' \rfloor$ bittiä. Merkitään $c = w' - \lfloor w' \rfloor$, joten $0 \leq c < 1$ ja $\lfloor w' \rfloor = w' - c$.

Tietorakenne koostuu bittijonosta, jossa on sikin sokin kahdenlaisia yksiköitä: "1" ja "$0b_1b_2 \cdots b_{w'-c}$", missä b_i:t ovat bittejä. 0-alkuinen yksikkö tarkoittaa, että tietorakenteessa on luku $x2^{w'-c} + b$, missä $b = \sum_{i=1}^{w'-c} b_i 2^{w'-c-i}$ on kyseisen yksikön sisältämä binääriluku, ja x on kyseistä yksikköä edeltävien muotoa "1" olevien yksiköiden määrä.

Kun bittijonoa selataan alusta alkaen, voidaan yksikön laji ja siksi myös pituus aina tunnistaa sen ensimmäisen bitin perusteella. Jonon loppuminen tunnistetaan siitä, että on kohdattu n kappaletta 0-alkuisia yksiköitä. Vaihtoehtoisesti jonon perään voidaan lisätä niin monta ylimääräistä "1-yksikköä, että "1-yksiköiden kokonaismäärä on $n2^c$. Tällöin jonon loppuminen tunnistetaan kohdattujen "1"-yksiköiden määrästä, ja rakenne pystyy esittämään myös alle n-alkioisia joukkoja.

0-alkuisten yksiköiden etunollia on kaikkiaan n kappaletta ja b_i-bittejä $n(w' - c)$ kappaletta. "1"-yksiköiden määräksi riittää $n2^c - 1$, koska siten ylletään lukuun $(n2^c - 1)2^{w'-c} + 2^{w'-c} - 1 = n2^{w'} - 1 = u - 1$ asti, kuten pitääkin. Kaikkiaan siis tarvitaan $n + n(w' - c) + n2^c - 1 = nw' + n(1 + 2^c - c) - 1$ bittiä. Lauseke $1 + 2^c - c$ saa arvon 2 välin $0 \leq c \leq 1$ päissä. Välin sisällä arvo on tätä pienempi, minimin ollessa $1 + \log_2 e - \log_2 \log_2 e = 1 + \frac{1}{\ln 2} + \frac{\ln \ln 2}{\ln 2} \approx 1{,}91$ kun $c = \log_2 \log_2 e \approx 0{,}53$.

6 Listojen pituuksien jakauma

Loistavasta menestyksestään huolimatta tietorakenteeni vaikuttaa eräässä suhteessa sangen tehottomalta. Nimittäin, se sisältää tilaa yli 5,5 miljoonalle alkiolle. Siis lähes 1,9 miljoonaa eli kolmasosa sen lokeroista on tyhjiä.

Suurin osa tyhjistä lokeroista on peruslohkoissa. Koska tiloja on 3 674 160 ja listoja 262 144, on listan keskipituus noin 14,02. Kuitenkin edellä todettiin, että ylivuotomekanismista huolimatta saavutettavat tilat eivät mahdu tietorakenteeseen, ellei peruslohkon koko ole vähintään 21. Miksi peruslohkon on oltava näin iso?

Vastaus paljastuu tutkimalla taulukon 2 kahta ensimmäistä saraketta "pit." (pituus) ja "tod." (todellinen). Sarakkeessa "pit." oleva luku ilmoittaa listan pituuden, ja sarakkeen "tod." luku kertoo, kuinka monta niin pitkää listaa tietorakenne sisältää kun kuution koko tila-avaruus on muodostettu. Havaitaan, että 4971 + 3162 + ··· + 1 = 12 574 listaa on pitempiä kuin 20 tilaa. Siis ylivuotolohkot loppuvat kesken, jos peruslohkon koko on 20 tai vähemmän.

Taulukosta 2 näkyy, että aika moni lista on pituudeltaan huomattavasti yli keskiarvon. Pisimmän listan pituus on peräti 33. Tämä havainto voi houkutella epäilemään, että käytetty sotkemisalgoritmi ei satunnaista tiloja kunnolla. Tästä ei kuitenkaan ole kyse.

Todellista syytä voi havainnollistaa pienellä ajatusleikillä. Kuvitellaan, että

pit.	tod.	teor.	ei s.
0	0	0,2	7621
1	3	3,0	16878
2	22	20,9	20427
3	82	97,7	19144
4	344	343,0	16106
5	969	962,6	13740
6	2247	2251,0	11556
7	4404	4511,5	9132
8	7948	7910,9	6682
9	12297	12328,8	4498
10	17436	17290,5	3005
11	22088	22041,7	2038
12	25380	25753,9	1864
13	28057	27773,2	2022
14	28004	27808,0	2622
15	26055	25983,5	3444
16	22399	22758,5	4425
17	18753	18758,9	5441
18	14783	14601,4	6305
19	10810	10765,8	6948

pit.	tod.	teor.	ei s.
20	7489	7540,0	7492
21	4971	5028,6	8119
22	3162	3200,9	8387
23	1997	1948,7	8420
24	1156	1136,8	7972
25	600	636,5	7617
26	360	342,7	7470
27	168	177,6	6742
28	89	88,8	6149
29	42	42,8	5442
30	18	20,0	4734
31	9	9,0	4110
32	1	3,9	3396
33	1	1,7	2887
34		0,7	2327
35		0,3	1928
36		0,1	1403
37		0,0	1047
38		0,0	779
39		0,0	559

Taulukko 2: Listojen pituuksien jakaumia

listoja olisikin 367 416 kappaletta, niin että listan keskimääräiseksi pituudeksi tulee tasan 10. Ainoa tapa saada tilat jakautumaan listoihin aivan tasan on, että juuri ennen viimeisen tilan lisäämistä yhdessä listassa on yhdeksän tilaa ja jokaisessa muussa kymmenen, ja sen jälkeen viimeinen tila osuu ainoaan vajaamittaiseen listaan. Mikä on todennäköisyys, että viimeinen tila osuu sinne minne pitää? Se on $\frac{1}{367\,416}$ eli noin 0,0003%. On siis lähes varmaa, että kaikista listoista ei tule samanpituisia.

Tämä ilmiö vaikuttaa tietenkin myös muita kuin viimeistä tilaa lisättäessä. Se vaikuttaa huomattavasti listojen pituuksien jakaumaan. Jakauman tarkka kaava on (kuten kohta esitettävä tylsä laskelma näyttää) hankala ryöpsähdys kertomia, mutta jakauma noudattaa varsin tarkasti seuraavaa, helposti taskulaskimellakin laskettavaa lakia, joka tunnetaan *Poissonin jakaumana*:

$$m_k \approx \frac{n}{h}\frac{h^k}{k!}e^{-h} \qquad (5)$$

Laissa $n = 3\ 674\ 160$ kuten ennenkin, h

on listojen keskimääräinen pituus eli noin 14,02, e on luonnollisen logaritmijärjestelmän kantaluku, ja m_k ilmoittaa, kuinka monen listan pituus on k. h:n määritelmästä johtuen listojen määrä on $\frac{n}{h}$ = 262 144.

Tylsä laskelma. Olkoon $u = 2^{31}$ kuten tähänkin asti, ja merkitään $l = u\frac{h}{n}$. Siis $l = 8192$ eli niiden erilaisten bittiyhdistelmien määrä, jotka joutuisivat samaan listaan, jos kaikki syntaktisesti mahdolliset tilat lisättäisiin tietorakenteeseen.

Jos johonkin listaan on osunut i tilaa ja muihin listoihin yhteensä j tilaa, niin seuraava tila osuu kyseiseen listaan todennäköisyydellä $\frac{l-i}{u-i-j}$ ja muualle todennäköisyydellä $\frac{u-l-j}{u-i-j}$. On olemassa kaikkiaan $\frac{n!}{k!(n-k)!}$ tapaa heittää n tilaa listoihin siten, että tarkasteltavaan listaan tulee tasan k tilaa. Yksittäisen tällaisen tavan todennäköisyyden ilmaisee tulo, jonka osoittajassa on luvut $l, l-1, \ldots, l-k+1, u-l, u-l-1, \ldots, u-l-(n-k)+1$ ja nimittäjässä $u, u-1, \ldots, u-n+1$, vaikka ei välttämättä tässä järjestyksessä. Niinpä luvun m_k tarkka arvo on

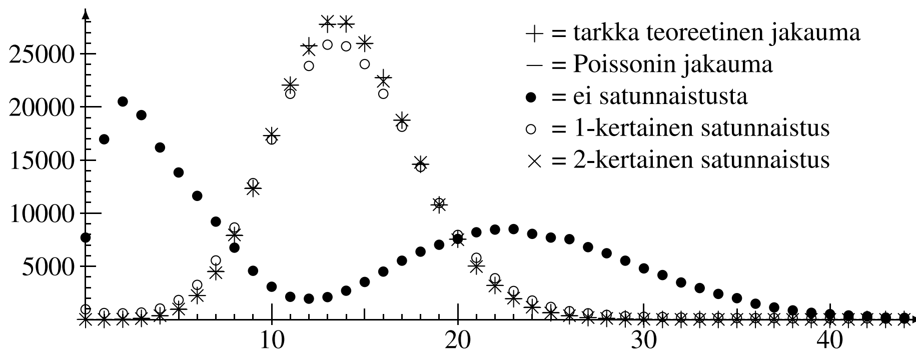
$$\frac{n}{h}\frac{n!}{k!(n-k)!}\frac{l!(u-l)!(u-n)!}{(l-k)!(u-l-n+k)!u!}$$

Meitä kiinnostavat vain pienet k:n arvot. Koska $k \ll l \ll n \ll u$, pätee $\frac{n!}{(n-k)!} \approx n^k$, $\frac{l!}{(l-k)!} \approx l^k$, $\frac{(u-l)!}{u!} \approx u^{-l}$ ja $\frac{(u-n)!}{(u-l-n+k)!} \approx (u-n)^{l-k} = (u-n)^l u^{-k}\left(1-\frac{n}{u}\right)^{-k}$. Niiden tulo on $x := \left(\frac{nl}{u}\right)^k \left(1-\frac{nl/u}{l}\right)^l \left(1-\frac{n}{u}\right)^{-k}$. Siis $m_k \approx \frac{n}{h}\frac{1}{k!}x$. Koska listan keskipituus on $h = \frac{nl}{u}$, on $x = h^k\left(1-\frac{h}{l}\right)^l\left(1-\frac{n}{u}\right)^{-k} \approx h^k e^{-h}$, koska $\left(1-\frac{n}{u}\right)^{-k} \approx 1$ ja $\left(1-\frac{h}{l}\right)^l \approx e^{-h}$.

Taulukon 2 sarakkeessa "teor." (teoreettinen) on tarkalla kaavalla numeerisesti laskettu teoreettinen ennuste listojen määrille pituuden funktiona. (Likimääräisen kaavan (5) antamat luvut poikkeavat tarkoista luvuista alle 2,3% kun pituus < 35.) Ainakin silmämääräisesti tarkasteltuna teorian ja todellisuuden välinen vastaavuus on erinomainen.

Samaa aineistoa kuin taulukossa 2 on kuvassa 6. "2-kertainen satunnaistus" on sama kuin taulukon 2 "tod." (nimi "2-…" saa kohta selityksen). Kuvassa on mukana myös Poissonin jakauma vaakaviivoina, mutta se poikkeaa tarkasta teoreettisesta jakaumasta niin vähän, että se erottuu vain muutamassa kohdassa jos niissäkään.

Epäilemättä on olemassa jokin tilastotieteellinen testi, jolla voisi vahvistaa sen johtopäätöksen, että mitattu jakauma täsmää teoriaan. Meille on kuitenkin tarjolla helpompi keino: lasketaan teorian ennustama muistin kulutus ja verrataan sitä todelliseen. Listojen pituusjakaumasta on helppo laskea, kuinka isoja perus- ja ylivuotolohkojen on oltava, jotta kaikki tilat mahtuisivat tietorakenteeseen. (Laskelmassa täytyy ottaa huomioon, että jokaista ylivuotolohkoa kohti menee yksi lokero "hukkaan" ylivuotolinkin tallettamiseen. Pyöristin teoreettisen jakauman luvut kokonaisluvuiksi siten, että listojen ja tilojen kokonaismäärät olisivat pyöristysvirheistä huolimatta mahdollisimman tarkoin oikeat.)



Kuva 6: Listojen pituusjakaumia kuvana

Teoria suosittelee niitä lohkokokoja mitä todellisuudessa käytetään, ja sen ennustama muistin kulutus on 11,106 miljoonaa tavua. Todellinen luku on 11,105 miljoonaa tavua. (Näitä lukuja laskiessani otin huomioon vain käytössä olevat ylivuotolohkot, koska sillä tavalla mahdolliset erot näkyvät varmemmin. Jos ylivuotolohkojen määränä pidetään aina 8192, kuten aiemmin tein, ei teorian ja käytännön välille synny tavuakaan eroa.)

Muistin kulutus on siis erittäin tarkasti teorian mukainen. Tämä ei suoranaisesti osoita, että todellinen pituusjakaumakin on teorian mukainen, mutta osoittaa, että se on ainakin yhtä hyvä: vaihtamalla teorian mukaiseen jakaumaan ei muistin kulutusta voitaisi pienentää. Kuvan 5 satunnaistamisalgoritmia ei siis tarvitse hylätä.

Uteliaisuuttani laskin jakauman ja muistin kulutuksen myös ilman tilojen satunnaistamista, sekä satunnaistamisalgoritmin muutamalla muunnelmalla. Ilman satunnaistamista saatava jakauma on taulukon 2 sarakkeessa "ei s." (ei satunnaistamista), ja se näkyy myös kuvassa 6. Listanpituudet 40, ..., 57 on jätetty pois tilan säästämiseksi. Se on silmin nähden aivan toisenlainen kuin teoreettinen jakauma. Peruslohkon koko olisi 34, ylivuotolohkon 7, ja muistia kuluisi 17,9 miljoonaa tavua.

Jos satunnaistamisalgoritmi tekisi vain kerran sen, minkä kuvan 5 algoritmi tekee kahdesti, olisi muistin kulutus yhä 11,9 miljoonaa tavua. Kolmella toistolla kulutus olisi 11,1 miljoonaa tavua, samoin neljällä. Yksi satunnaistamiskierros on siis vielä puutteellinen, mutta kaksi tai enemmän satunnaistaa tämän käyttötarkoituksen kannalta täydellisesti.

Kokeilin senkin, mitä tapahtuu, jos tilan bittiesitys käännetään takaperin ennen käyttöä. Silloin indeksiin poimitaan eri bitit kuin tähän asti. Ilman satunnaistamista jakauma muuttui entistäkin omituisem-

maksi, ja edellytti yli 28 miljoonan tavun käyttöä (178 400 tyhjää listaa, suurin listan pituus on 60, jakauma pomppii ylös alas kaoottisen näköisesti: esimerkiksi toiseksi ja kolmanneksi korkeimmat piikit 28 062 ja 8764 ovat kohdissa 54 ja 42). Kaksinkertaisella satunnaistamisella jakauma palasi hyvin lähelle teoreettista vaikka ei identtiseksi aiemman kokeellisen jakauman kanssa, tuottaen taas muistinkulutukseksi noin 11,1 miljoonaa tavua.

Joskus on mahdollista käyttää jakauman vinoutta hyväksi tietorakenteen suunnittelussa muistin säästämiseksi. Jos esimerkiksi minkään listan pituus ei ylittäisi 16, niin silloin valitsemalla peruslohkon kooksi 16 vältyttäisiin ylivuodoilta kokonaan, ja noin 8,4 miljoonaa tavua muistia riittäisi. Vinouden hyödyntäminen on kuitenkin mahdollista vain jos tila-avaruudesta pystytään tunnistamaan jokin sopiva rakenteellinen säännöllisyys. Se onnistuu joskus mutta usein ei, ja kun se epäonnistuu, voi tulos olla todella huono, kuten edellä nähtiin. Sen sijaan satunnaistamiseen perustuva lähestymistapa toimii riippumatta siitä, minkälainen alkuperäinen jakauma on.

7 Yleinen hyvin tiivis hajautustaulu

Taulukon 2 perusteella on selvää, että muistin kulutusta voisi vieläkin vähentää pienentämällä peruslohkon kokoa ja kasvattamalla ylivuotolohkojen määrää. Ylivuotolinkin pituutta pitäisi silloin kasvattaa. Tämän voi tehdä ottamalla käyttöön uuden taulukon, jossa on jokaista lohkoa kohti kyseisen lohkon lopussa (mahdollisesti) olevan ylivuotolinkin täydennysbitit.

En ole tehnyt tällaista ohjelmaa, mutta laskin jakauman avulla, kuinka paljon se kuluttaisi muistia, jos ylivuotolinkki kasvatettaisiin 17-bittiseksi. Peruslohkon kooksi tulisi 15, ylivuotolohkon 4 ja muistin kulutukseksi 8,98 miljoonaa tavua (tai 8,94 miljoonaa tavua, jos vain tarpeelliset ylivuotolohkot toteutetaan). Satunnaistamaton ohjelma kuluttaisi edelleen 12,6 miljoonaa tavua ja kertaalleen satunnaistettu 9,2 miljoonaa.

En tehnyt tällaista ohjelmaa, koska se vapauttaisi tietorakenteeni vain yhdestä tietokoneiden sanapituuden asettamasta keinotekoisesta rajoitteesta. Toinen rajoite on indeksin ja tilan loppuosan välisen rajan sijainti: mikään ei takaa, että jakosuhde 18+13 olisi paras. Valitsin sen vain siksi, että siten sain aineiston kätevästi sovitettua 8-bittisiin tavuihin. Sen sijaan esittelin tietorakenteen jatko-opiskelijalleni Jaco Geldenhuysille ja pyysin häntä kehittämään yleiskäyttöistä tietorakennetta sen pohjalta.

Erinäisten vaiheiden jälkeen Jaco ja minä päädyimme seuraavaan ratkaisuun, jota kutsumme nimellä "very tight hash table" eli "hyvin tiivis hajautustalu" [3]. Tyhjää lokeroa ja ylivuotoa ei enää ilmaista talletettavaan alkioon liitettävän oheistiedon käyttämättömillä arvoilla, koska oheistietoa ei enää välttämättä ole, ja vaikka olisikin, ei sillä välttämättä ole käyttä-

mättömiä arvoja. Sen sijaan jokaiseen lohkoon liitetään laskuri, joka kertoo, kuinka monta alkiota lohkossa on, ja jolla on yksi ylimääräinen arvo tarkoittamaan ylivuotanutta lohkoa. Ylivuotolohkon laskurin ei tarvitse kyetä ilmaisemaan arvoa nolla, koska ylivuotolohkoa ei oteta käyttöön, ellei sinne tule jotakin.

Ylivuotolinkille varataan erikseen tilaa ylivuotolohkoissa, mutta ei peruslohkoissa. Jos peruslohko vuotaa yli, niin ylivuotolinkki sijoitetaan aivan lohkon alkuun, ja sen alle jäävät bitit siirretään listan viimeisen lohkon ylivuotolinkkikenttään. Näin saadaan hyödynnettyä viimeisen lohkon muutoin tarpeettomaksi jäävä ylivuotolinkille varattu tila.

Saadaksemme selville, minkä kokoisia perus- ja ylivuotolohkojen kannattaa olla ja mistä kohti alkio kannattaa jakaa indeksiin ja loppuosaan, teimme ison joukon erilaisia simulaatiokokeita sekä mittauksia Jacon tekemällä prototyypillä. Hyväksi nyrkkisäännöksi osoittautui, että peruslohkon kooksi kannattaa valita listan keskipituus ja ylivuotolohkon kooksi kolme.

Listojen keskipituudesta ei kannata tehdä kovin suurta, koska pitkien listojen selaaminen on hidasta. Sitäpaitsi mitä pitemmät listat, sitä vähemmän indeksissä on bittejä, joten sitä vähemmän indeksien käyttö säästää muistia. Toisaalta kovin lyhyillä listoilla linkkien sekä käyttämättömien lokeroiden osuus muistinkulutuksesta kasvaa. Listan keskipituudeksi kannattaa valita muutama kymmenen.

Analysoimme tietorakenteemme muistin kulutusta teoreettisesti, ja laskimme teoreettisen kaavan vakioille arvoja numeerisesti. Saimme tulokseksi muun muassa, että jos listan pituus on keskimäärin 20, niin muistin kulutus satunnaisella joukolla jää alle

$$1{,}13nw' + 0{,}04n\log_2 n + 5{,}05n$$

bitin, ja keskipituudella 50 alle luvun

$$1{,}07nw' + 0{,}02n\log_2 n + 6{,}12n\ .$$

Kaavoista voi kokeilemalla havaita, että ellei talletettavien alkioiden määrä ole hyvin pieni verrattuna syntaktisesti mahdollisten alkioiden määrään, jää tietorakenteen muistinkulutus alle luvun nw eli (alkioiden määrä) $\cdot$ (alkion pituus) ja siten reilusti alle tavallisen hajautustaulun muistinkulutuksen. Jos talletettavien alkioiden määrä on pienehkö suhteessa syntaktisesti mahdollisten alkioiden määrään ($\leq 5\%$, jos talletettavia alkioita on enintään miljardi), jää tietorakenteen muistinkulutus alle 2 kertaa informaatioteoreettisen alarajan.

Hyvin tiivis hajautustaulu on siis selvä parannus esimerkiksi tavalliseen hajautustauluun nähden, mutta ei enää omasta puolestaan jätä tilaa kovin suurille parannuksille ennen kuin informaatioteoreettinen alaraja tulee vastaan.

Kaavojen perusteella ei voi suoraan laskea, kuinka paljon hyvin tiivis hajautustaulu kuluttaisi muistia $2 \times 2 \times 2$ Rubikin kuution tapauksessa, koska ne eivät ota huomioon pyöräytyssuuntien esittämisen tarvitsemaa muistia. Tulkitsemalla pyöräytyssuuntabitit osaksi alkiota saadaan kummastakin kaavasta tulokseksi

bittejä indeksissä	listan pituus	muisti 10^6 tavua	aika sek läppäri	aika sek työasema
15	112,13	9,2	1160	125
16	56,06	9,0	664	74
17	28,03	8,9	410	48
18	14,02	9,0	286	36
19	7,01	9,4	229	30

Taulukko 3: Mittaustuloksia hyvin tiiviillä hajautustaululla

noin 9,0 miljoonaa tavua, mikä sopii hyvin yhteen tämän luvun alussa esitetyn laskelman kanssa.

Kokeilimme Rubikin kuutio -ohjelmaa hyvin tiivistä hajautustaulua käyttäen siten, että indeksin bittien määrä vaihteli välillä 15, ..., 19. Pyöräytyssuunnat talletettiin alkion yhteyteen. Taulukossa 3 on tuloksia. Ne täsmäävät erinomaisesti edellä annettuun teoreettiseen arvioon. Indeksin koon kasvaessa muistin kulutus aluksi vähenee ja sitten kasvaa, kuten edellä todettiin. Ajan kulutus kasvaa listan pituuden myötä. Ajan kulutuksen osalta hyvin tiivis hajautustaulu ei pysty kilpailemaan aiemman tietorakenteen kanssa, mutta muistin käytön osalta se on selvästi parempi.

Hyvin tiivis hajautustaulu ja siihen liittyviä teoreettisia, simulaatio- ja mittaustuloksia on esitetty julkaisussa [3].

8 Permutaatioiden indeksointi

Tilan *täydellisellä pakkaamisella* tai *indeksoinnilla* tarkoitetaan pakkausmenetelmää, jonka tuottamat pakatut tilat ovat binääriluvuiksi tulkittuina välillä 0, ..., $n-1$, missä n on saavutettavien tilojen määrä. Pakattu tila vie siis juuri ja juuri niin paljon muistia, että jokaiselle tilalle riittää oma bittiyhdistelmä.

Jos tila voidaan pakata täydellisesti, niin saavutettavien tilojen joukko voidaan esittää hyvin yksinkertaisella, nopealla ja muistin käytöltään tehokkaalla tietorakenteella: bittitaulukolla, jota indeksoidaan tilan pakatulla esityksellä, ja jonka sisältö kertoo, onko kyseinen tila löydetty vai ei. Jos tilaan liittyy oheistietoa, se voidaan tallettaa tehokkaasti omaan taulukkoonsa. Jos oheistiedolla on käyttämättömiä arvoja kuten $2 \times 2 \times 2$ Rubikin kuution tapauksessa, niin jokin niistä voidaan valita edustamaan tyhjää, jolloin erillistä bittitaulukkoa ei tarvita.

Tällainen ratkaisu tarvitsee $2 \times 2 \times 2$ Rubikin kuution tilojen esittämiseen vain 459 270 tavua muistia. Jos pyöräytyssuunnatkin esitetään, niin muistin tarve on noin 1,4 miljoonaa tavua. Nämä luvut ovat reippaasti alle aiemmin johdetun informaatioteoreettisen alarajan. Tässä ei kuitenkaan ole ristiriitaa, sillä kyseinen alaraja perustuu oletukseen, että tilat esitetään 31 bitillä, mikä ei enää pidä paikkaansa.

Käyttökelpoista täydellistä pakkausalgoritmia ei tila-avaruussovelluksiin voi yleensä löytää, koska se edellyttäisi parempaa ymmärrystä tutkittavasta järjestelmästä kuin on tarjolla. Rubikin kuution tapauksessa tilojen rakenne kuitenkin ymmärretään erinomaisesti. Tilojen vaihtelu muodostuu siitä, että seitsemää liikkuvaa nurkkaa permutoidaan eri järjestyksiin, ja kuusi nurkkaa voi olla missä tahansa kolmesta asennosta. Seitsemännen nurkan asento määräytyy muiden nurkkien asennoista.

Asennot on helppo pakata täydellisesti tulkitsemalla ne 3-järjestelmän luvun numeroiksi. Nurkkien järjestyksen täydellinen pakkaaminen vastaa kykyä indeksoida permutaatio. Sen tekevä tehokas algoritmi julkaistiin artikkelissa [5].

Kuvissa 7 ja 8 on esitetty näihin perustuva $2 \times 2 \times 2$ Rubikin kuution tilan täydellinen pakkaus- ja purkualgoritmi. Algoritmin pakkaama tila on luku muotoa $3^6 \cdot p + a$, missä $a = \sum_{i=0}^{5} a_i \cdot 3^i$ tallettaa kuuden nurkan asennot, ja p esittää permutaation. Se on muotoa $\sum_{i=1}^{6} i!p_i$.

Luku p_6 on alkion 6 alkuperäisen paikan numero, ja on välillä 0, ..., 6. Luku p_5 on alkion 5 paikan numero siinä taulukossa, joka saadaan edellisestä taulukosta vaihtamalla luku 6 ja viimeinen luku keskenään. Näin ollen $0 \leq p_5 \leq 5$. Näin jatketaan kunnes luku p_1 on saatu. Tämän jälkeen alkio 0 on paikassa 0. Niinpä vastaavalla tavalla määritelty luku p_0 olisi aina nolla, joten sitä ei tarvitse tallettaa.

Kuvan 7 algoritmi vastaa yllä kuvattua, mutta siitä on jätetty pois sijoituslauseita, joiden kohteena olevaa taulukon lokeroa ei jatkossa käytetä. Samoin kuvan 8 algoritmista on jätetty turhia toimintoja pois. Viimeisen nurkan asennon selvittäminen perustuu siihen, että kaikkien liikkuvien nurkkien asentojen summa on aina jaollinen kolmella, kuten kuvan 2 yhteydessä todettiin.

Tilojen täydelliseen pakkaamiseen perustuva ohjelma on muistinkulutukseltaan tietysti ylivertainen muihin nähden: 1,4 miljoonaa tavua (minkä lisäksi jono tarvitsee vajaan 6 miljoonaa tavua saavutettujen tilojen joukon esitystavasta riippumatta). Suureksi yllätykseksi se ei kuitenkaan ollut nopein.

Se kulutti läppärissäni aikaa 169 sekuntia, mikä on 60% enemmän kuin optimoituun hajautustauluun perustuvan ohjelman ajan kulutus. Työasemassa aika oli 28 sekuntia eli 40% hitaampi kuin optimoituun hajautustauluun perustuva ohjelma. (Aiemmalla kääntäjällä tämä aika oli yllättävän paljon: 67 sekuntia.) Nopeusmittauksissa varasin tilataulukolle yhden tavun tilaa kohti, jotta kolmebittisen tilainformaation sullominen keskelle tavua ei hidastaisi ohjelmaa. Sellainen tilataulukko kuluttaa 3,67 miljonaa tavua, mikä on edelleen reilusti alle muiden ohjelmaversioiden tarpeen.

Permutaatioiden indeksointiin perustuvan ohjelman selvä häviö optimoitua hajautustaulua käyttävälle oli niin odottamatonta, että etsin selitystä lopulta tietokoneiden liukuhihna-arkkitehtuurien yksityiskohdista ja hain apua Matti Rintalalta. Loppujen lopuksi selitys on yksinkertainen.

```
pakattu_tila pakkaa()const{
  pakattu_tila tulos = 0;

  /* Muodosta suora- ja käänteispermutaatio */
  int A[ 7 ], P[ 7 ];
  for( int i1 = 0; i1 < 7; ++i1 ){
    A[ i1 ] = sisalto[ i1 ] / 3; P[ A[ i1 ] ] = i1;
  }

  /* Pakkaa nurkkien sijaintipermutaatio */
  for( int i1 = 6; i1 > 0; --i1 ){
    tulos = ( i1 + 1 ) * tulos + P[ i1 ];
    A[ P[ i1 ] ] = A[ i1 ]; P[ A[ i1 ] ] = P[ i1 ];
  }

  /* Pakkaa nurkkien asennot viimeistä nurkkaa lukuunottamatta.
     Sen asento määräytyy muiden asennoista. */
  for( int i1 = 5; i1 >= 0; --i1 ){
    tulos *= 3; tulos += sisalto[ i1 ] % 3;
  }

  return tulos;
}
```

Kuva 7: Permutaatioiden indeksointiin perustuva tilanpakkausalgoritmi

Kumpikin ohjelma käyttää suurimman osan ajastaan tilojen pakkaamiseen ja purkamiseen sekä pakattujen tilojen etsimiseen saavutettujen tilojen tietorakenteesta. Vaikka etsiminen hajautustaulusta edellyttää listojen selaamista ja tilan loppuosan kaivamista esiin tavupareista, ei siihen mene kohtuuttomasti aikaa, koska listat eivät ole kovin pitkiä ja aina samassa kohdassa tavuparia olevan tiedon käsittely on nopeaa.

Toisaalta kuvien 7 ja 8 algoritmit yksinkertaisuudestaan huolimatta tekevät useita kymmeniä operaatioita enemmän kuin hajautustaulun yhteydessä käytettävät pakkaus- ja purkualgoritmi, jotka toimivat samoin kuin asentojen käsittely kuvien algoritmeissa, paitsi että jokaisen nurkan asento talletetaan. Siis täydelliseen pakkaamiseen perustuva ohjelma tekee kaiken kaikkiaan huomattavasti enemmän työtä.

```
void pura( pakattu_tila pt ){

  /* Puretaan nurkkien asennot viimeistä lukuunottamatta */
  for( int i1 = 0; i1 < 6; ++i1 ){
    sisalto[ i1 ] = pt % 3; pt /= 3;
  }
  sisalto[ 6 ] = 0;

  /* Puretaan nurkkien sijainnit */
  int A[ 7 ]; A[ 0 ] = 0;
  for( int i1 = 1; i1 < 7; ++i1 ){
    int paikka = pt % ( i1 + 1 ); pt /= i1 + 1;
    A[ i1 ] = A[ paikka ]; A[ paikka ] = i1;
  }
  for( int i1 = 0; i1 < 7; ++i1 ){ sisalto[ i1 ] += 3 * A[ i1 ]; }

  /* Selvitetään viimeisen nurkan asento muiden funktiona
     (Paikka selvisi automaattisesti edellä) */
  int jk = 120;  // niin iso 3:n kerrannainen, että jk säilyy >= 0
  for( int i1 = 0; i1 < 6; ++i1 ){ jk -= sisalto[ i1 ]; }
  sisalto[ 6 ] += jk % 3;

}
```

Kuva 8: Permutaatioiden indeksointiin perustuva tilanpurkualgoritmi

9 Lopuksi

Tavoitteiltaan alun perin vaatimaton ohjelmointikokeeni johti siis moniin havaintoihin, entistä paremman tietorakenteen kehittämiseen ja (tähän mennessä) yhteen julkaisuun.

Havainnoista ensimmäinen oli, että laadukkaatkin valmiit tietorakenteet voivat olla yllättävän tehottomia. Tilannetta varten optimoidulla omalla tietorakenteella voi päästä samanaikaisesti sekä muistin että ajan kulutuksen osalta huomattavasti parempaan tulokseen. Satunnaisen n-alkioisen joukon esittämiseen tarvittavan muistin määrässä voidaan päästä jopa alle ensi näkemältä periaatteelliselta alarajalta vaikuttavan luvun nw, missä w on alkion pituus bitteinä — ja ohjelma on silti nopea.

Koska nw ei ole todellinen tarvittavan muistin määrän informaatioteoreettinen alaraja, heräsi kysymys, että mikä sitten on. Vastaus on suunnilleen nw –

$n\log_2 n + 1{,}44n$, kun $w \gg \log_2 n$. Tämä on tilastollinen alaraja — yksittäistapauksissa se voidaan alittaa, mutta keskimääräisessä tapauksessa muistia tarvitaan ainakin tämän verran.

Jos alkioita nakellaan satunnaisesti listoihin niin että jokainen lista on aina yhtä todennäköinen, niin listojen pituuksien jakauma on yllättävän vino. $2 \times 2 \times 2$ Rubikin kuution tapauksessa pisin lista oli pituudeltaan yli kaksinkertainen keskimääräiseen listaan verrattuna. Kolme listaa oli yksialkioisia, vaikka keskipituus oli sentään 14.

$2 \times 2 \times 2$ Rubikin kuution saavutettavien tilojen tallettamiseksi suunniteltu tietorakenne aloittaa tilan käsittelyn satunnaistamalla sen. Jos tiloilla tiedetään olevan säännöllistä rakennetta, niin sitä hyödyntämällä voi säästää muistia. Usein on kuitenkin niin, että käyttökelpoista säännönmukaisuutta ei tunneta. Tällaisessa tilanteessa säännönmukaisuus voi olla haitaksi. Niin todella kävi: tietorakennetta kokeiltiin myös ilman satunnaistamista, jolloin muistin kulutus kasvoi melkoisesti.

Jos käyttökelpoista säännönmukaisuutta ei ole, niin sitä ei voi mitenkään luoda. Satunnaisuuden osalta tilanne on toisin: vaikka aineisto ei olisi alunperin satunnainen, se voidaan satunnaistaa. Tämä tekee satunnaisuuteen luottavista ratkaisuista yleispäteviä. Satunnaistamalla menetetään mahdollisuus hyödyntää säännöllistä rakennetta, mutta samalla vältetään myös vaara, että jokin ennalta arvaamaton säännömukaisuus toimii valittua tietorakennetta vastaan ja pilaa sen tehokkuuden.

$2 \times 2 \times 2$ Rubikin kuution tapauksessa kokeiltiin myös tilojen rakenteen säännöllisyyden hyödyntämistä. Koska tiedettiin tilan olevan yhdistelmä 7-alkioisesta permutaatiosta ja kuudesta yksittäisestä kolmiarvoisesta tiedosta, oli mahdollista laatia nopeat algoritmit, jotka pakkasivat ja purkivat tilan täydellisesti. Täydellinen pakkaaminen tarkoittaa, että jokainen tila sai yksilöllisen numerokoodin, joka on pienempi kuin tilojen määrä. Täydellisesti pakatuissa tiloissa ei siis ole yhtään enempää informaatiota kuin tarvitaan yksilöimään tila.

Tilan täydellinen pakkaaminen mahdollistaa tila-avaruuden esittämisen yksinkertaisena taulukkona, jonka käsittelyoperaatiot ovat erittäin nopeita. Niinpä oli yllätys, kun paljastui, että täydelliseen pakkaamiseen perustuva ohjelma oli olennaisesti hitaampi kuin optimoituun hajautustauluun perustuva.

Vaikka täydellinen pakkaaminen ja purkaminen ovat nopeita operaatioita, ovat alkioiden satunnaistamiseen ja optimoituun hajautustauluun perustuvan ohjelman käyttämät hyvin yksinkertainen pakkaus ja purku vielä nopeampia. Ne ovat niin paljon nopeampia, että pakkaamisen ja purkamisen aikana saavutettu ajan säästö riitti moninverroin hyvittämään satunnaistamisessa ja monimutkaisemman tila-avaruusrakenteen käsittelemisessä hävityn ajan.

Toteutettua ohjelmaa voisi periaatteessa käyttää $2 \times 2 \times 2$ Rubikin kuution ratkaisemisen apuvälineenä. Tilan syöttäminen sille on kuitenkin niin kömpelöä, että moni ratkaisee kuution nopeammin kuin

et.	tiloja	uusia	et.	tiloja	uusia	et.	tiloja	uusia
0	1	1	5	2 944	2 256	10	1 450 216	930 588
1	7	6	6	11 913	8 969	11	2 801 068	1 350 852
2	34	27	7	44 971	33 058	12	3 583 604	782 536
3	154	120	8	159 120	114 149	13	3 673 884	90 280
4	688	534	9	519 628	360 508	14	3 674 160	276

Taulukko 4: Saavutettujen tilojen määrä pyöräytyssarjan pituuden funktiona

syöttää tilan. Suurempi hyöty ohjelmasta saadaan tutkimalla sen avulla tila-avaruuden rakennetta.

Taulukossa 4 on saavutettujen tilojen määrä etäisyyden eli alkutilasta alkavan pyöräytyssarjan pituuden funktiona. Kolmas palsta kertoo uusien tilojen määrän, eli niiden tilojen, jotka voi saavuttaa ensimmäisen palstan luvun pituisella pyöräytyssarjalla mutta ei lyhyemmällä.

Taulukosta näkyy, että etäisin tila on 14 pyöräytyksen päässä järjestetystä tilasta. Yli puolet tiloista on yli 10 pyöräytyksen etäisyydellä, siis reilusti kauempana kuin puolet maksimietäisyydestä. Taulukosta voi myös päätellä, että jos tehtäisiin ohjelma, joka etsii annetulle tilalle ratkaisua tekemällä yhtäaikaa leveyteen ensin -hakua sekä järjestetystä tilasta että annetusta tilasta alkaen, niin sen ei koskaan tarvitsisi tutkia yli 90 000 tilaa!

Mitä seuraavaksi? Luonnolliselta tuntuisi käydä $3 \times 3 \times 3$ Rubikin kuution kimppuun. Valitettavasti sillä on peräti $8! \cdot 12! \cdot 2^{12} \cdot 3^8 / 12 \approx 4 \cdot 10^{19}$ saavutettavaa tilaa, mikä taitaa olla liian iso pala ylihuomisenkin tietokoneille. Vain osan tiloista tutkivia ohjelmia voi toki tehdä, mutta on vaikea tietää, onko sellaisen löytämä ratkaisu lyhyin mahdollinen. Ylipäänsä jonkin ratkaisun löytämisessä ei ole haastetta — siihen on julkaistu ohjeita.

Se tiedetään, että nurkkapalat paikalleen laittava optimaalinen pyöräytyssarja voidaan löytää 20 sekunnissa. Nimittäin, $2 \times 2 \times 2$ kuutio on $3 \times 3 \times 3$ kuution nurkkapalojen joukko.

Kiitokset

Pyysin Jyrki Lahtosta Turun yliopistosta riippumattoman asiantuntijan ominaisuudessa sekä Ari Korhosta lehden puolesta lukemaan tämän tekstin ja antamaan siitä huomautuksia ennen Arin tekemää julkaisupäätöstä. Sain hyviä huomautuksia, joiden mukaan paransin tekstiäni. Kiitokseni Jyrkille ja Arille!

Viitteet

[1] Abramowitz, M. & Stegun, I. (toim.): *Handbook of Mathematical Functions, with Formulas, Graphs, and Mathematical Tables*. Dover Publications Inc., New York 1970.

[2] Boyer, C. *Tieteiden kuningatar, matematiikan historia osa II*. Art House 1994.

[3] Geldenhuys, J. & Valmari, A.: "A Nearly Memory-Optimal Data Structure for Sets and Mappings". Model Checking Software, 10th International SPIN Workshop, Portland, OR, USA, May 9–10, 2003, Proceedings, *Lecture Notes in Computer Science* 2648, Springer 2003, pp. 136–150.

[4] Knuth, D. E.: *The Art of Computer Programming, Vol 3: Sorting and Searching*. Addison-Wesley 1973.

[5] Myrvold, W. & Ruskey, F.: "Ranking and Unranking Permutations in Linear Time". *Information Processing Letters*, 79(6) (2001) 281–284.