PAIKALLISHAKU KLUSTEROINTIMENETELMÄNÄ

Marko Tuononen

13.04.2006

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Klusterointi on menetelmä saada tietoa aineiston rakenteesta. Klusterointi on tärkeä monilla tietojenkäsittelytieteen osa-alueilla sekä useiden empiiristen tieteiden parissa. Menetelmät parhaan mahdollisen klusteroinnin tuottamiseksi ovat tunnettuja, mutta klusteroinnin ollessa NP-täydellinen ongelma, joudutaan käytännössä ajan säästämiseksi tinkimään ratkaisun laadusta. Tässä tutkielmassa klusterointimenetelmien välisissä vertailuissa käytetään kriteerinä tehokkuutta, joka on menetelmän saavuttama parannus aikayksikköä kohden.

Klusterointi määritellään kombinatorisena optimointiongelmana, ja sen ratkaisemiseksi käytetään paikallishakua, jonka tehostaminen käyttäen PDS-menetelmää sekä karsittujen vertailujen hakua tullaan esittämään. Paikallishaku perustuu naapuruuden käsitteeseen. Naapuriratkaisu muodostetaan deterministisesti poistamalla klusteri, josta on vähiten hyötyä, ja luomalla uusi klusteri sinne, missä sille on eniten tarvetta. Satunnaisessa naapuriratkaisun muodostamisessa sekä poisto että luonti ovat satunnaisia. Naapuriratkaisun muodostamiseksi on tarkasteltu myös vaihtoehtoa, jossa ainoastaan klusterin poisto tai lisäys on deterministinen.

Paikallishaun satunnaista naapuriratkaisun muodostamista käyttävä versio on rakenteeltaan yksinkertainen ja se voittaa tehokkuudessa esimerkiksi laajalti käytetyn K-means –menetelmän. Muodostettaessa naapuriratkaisu täysin satunnaisesti tehdään turhaa työtä arvioitaessa runsaasti ratkaisuja, jotka eivät paranna tulosta. Paikallishaun determinististä naapuriratkaisun muodostamista käyttävät versiot ovat rakenteeltaan hieman monimutkaisempia. Ne kuitenkin pyrkivät deterministisyydellän vähentämään turhien ratkaisujen luomista ja täten saavuttamaan paremman tehokkuuden.

Tässä tutkielmassa saatujen tulosten mukaan deterministisyys naapuriratkaisun muodostamisessa toimii aineistoilla, joissa on havaittavissa selkeitä klustereita. Klusterin satunnaisesti poistava ja luotavan klusterin sijainnin deterministisesti valitseva versio vaikuttaisi olevan hyvä kompromissi täysin satunnaisen ja täysin deterministisen version välillä.

ACM-luokat (ACM Computing Classification System, 1998 version): I.5.3, G.2.1
Avainsanat: klusterointi, kombinatorinen optimointiongelma, paikallishaku, NP-täydellinen, PDS-menetelmä, karsittujen vertailujen haku, K-means

Esipuhe

Tämä tutkielma on syntynyt erikoistyöni ja SIPU-laboratoriossa vuosina 2004–2006 tekemäni työn tuloksena.

Tahdon kiittää työni ohjaajaa professori Pasi Fräntiä hänen antamastaan opastuksesta ja kannustuksesta sekä FT Olli Virmajokea yhteistyöstä determinististen versioiden parissa. Lisäksi tahdon kiittää äitiä ja isää siitä tuesta, jonka he ovat tarjonneet opintoihini.

Joensuussa 13.04.2006

Marko

Lyhenteet ja symbolit

Lyhenteet

ACM	Association for Computing Machinery
IS	Iterative Shrinking
LDC	Linguistic Data Consortium
MAE	Mean Absolute Error
MPS	Mean-distance-ordered Partial Search
MFCC	Mel Frequency Cepstral Coefficents
MSE	Mean Square Error
NIST	National Institute of Standards and Technology
PDS	Partial Distortion Search
RGB	Red, Green, Blue
SIPU	Speech and Image Processing Unit
TIE	Triangular Inequality Elimination
TIMIT	Texas Instruments & Massachusetts Institute of Technology

Symbolit

<<	paljon pienempi kuin
a	naapuriklustereiden keskimääräinen määrä; määritettävä erikseen kullekin aineistolle
C	sentroidit eli klustereiden keskipisteet $c_1, c_2, \ldots, c_M$
c_i^j	klusterin i sentroidin j:s attribuutti.
I	K-means –iteraatioiden lukumäärä
K	aineiston X alkioiden ulotteisuus
M	klustereiden lukumäärä
N	aineiston X koko
O()	asymptoottinen järjestys; ordo-
P	ensisijainen ositus $p_1, p_2, .., p_N$; kuvaa kunkin alkion lähimpään klusteriin
Q	toissijainen ositus $q_1, q_2, \ldots, q_N$; kuvaa kunkin alkion toiseksi lähimpään klusteriin
T	paikallishaku- tai toistettu K-means –iteraatioiden lukumäärä
X	aineisto; koostuu alkioista $x_1, x_2, \ldots, x_N$
x_i^j	aineiston X i:nnen alkion j:s attribuutti

Sisällysluettelo

1. JOHDANTO 1

2. KLUSTEROINTI 2

2.1. PERUSTEET 2
2.2. KLUSTEROINTIONGELMAN MÄÄRITTELY 4
2.3. KLUSTEROINNIN ARVIOINTI 6
2.3. HIERARKKISET MENETELMÄT 7
2.4. OSITTAVAT MENETELMÄT 8
2.5. KLUSTEROITAVAT AINEISTOT 10

3. PAIKALLISHAKU 14

3.1. PAIKALLISHAUN PERUSVERSIO 14
3.2. PAIKALLISHAUN EDISTYNEEMMÄT VERSIOT 17
3.3. PAIKALLISHAKU KLUSTEROINTIMENETELMÄNÄ 20
3.4. SATUNNAISTETTU PAIKALLISHAKU KLUSTEROINTIMENETELMÄNÄ 21
3.4.1. Tietorakenteet 21
3.4.2. Perusoperaatiot 22
3.4.3. Lähtöratkaisun valinta 25
3.4.4. Naapuristofunktio 26

4. SATUNNAISTETTU PAIKALLISHAKU 30

4.1. SATUNNAISTETUN PAIKALLISHAUN HAVAINNOLLISTAMINEN 31
4.2. SATUNNAISTETUN PAIKALLISHAUN ANALYSOINTIA 34
4.2.1. Parametrien vaikutus 35
4.2.2. Paikallishakuiteraatioiden määrä 37
4.2.3. Profilointi 39

5. PAIKALLISHAUN TEHOSTAMINEN 41

5.1. LÄHIMMÄN NAAPURIN HAUN NOPEUTTAMINEN 41
5.2. KARSITTUJEN VERTAILUJEN HAKU 42
5.3. EDISTYNEEN VERSION ANALYSOINTIA 45
5.3.1. K-means –iteraatioiden määrä 45
5.3.2. Edistyneen paikallishaun vertailua 47
5.3.3. Saavutettu nopeutus 48
5.3.4. Nopeutus operaatiotasolla 49
5.3.5. Koodikirjan koon vaikutus 49

6. DETERMINISTINEN SENTROIDIN KORVAUS 51

6.1. DETERMINISTISEN SENTROIDIN KORVAUKSEN IDEA 51
6.2. DETERMINISTISEN SENTROIDIN KORVAUKSEN TOTEUTUS 54
6.2.1. Poistettavan klusterin valinta 54
6.2.2. Luotavan klusterin valinta 55
6.2.3. LBG-U 56
6.2.4. Sentroidin korvauksen aikavaativuus 56
6.3. DETERMINISTISEN SENTROIDIN KORVAUKSEN HAVAINNOLLISTAMINEN 56
6.4. DETERMINISTISEN SENTROIDIN KORVAUKSEN ANALYSOINTIA 58
6.4.1. Lisäys- ja poistomenetelmän vaikutus 58

6.4.2. Determinististen versioiden vertailua kuva- ja Birch-aineistoilla *59*
6.4.3. Determinististen versioiden vertailua puheaineistoilla *62*
6.4.4. Determinististen versioiden suorituskyky *64*
6.5. PAIKALLISHAUN JATKOKEHITYSIDEOITA 66
6.5.1. Nopeuttaminen *66*
6.5.2. Laadun parantaminen *66*

7. YHTEENVETO 68

VIITTEET 69

LIITE 1: PAIKALLISHAUN EDISTYNEEN VERSION TOTEUTUS 75

LIITE 2: DETERMINISTISEN KORVAUKSEN TOTEUTUS 82

1. Johdanto

Kykymme tuottaa ja kerätä tietoa on kasvanut kiivaasti viimeisten vuosikymmenten aikana. Tämän on saanut aikaan muun muassa useiden alojen *tapahtumien* (transaction) kirjauksen tietokoneistuminen sekä edistysaskeleet tiedonkeruussa ja tallennuksessa (Han & Kamber, 2001). Esimerkiksi vakuutusyhtiöt, pankit, lentoyhtiöt, hotelliketjut, matkapuhelinoperaattorit ja monet muut yritykset varastoivat päivittäiset tapahtumansa. Pelkästään yhdysvaltalainen kauppaketju Wal-Mart kirjaa 20 miljoonaa myyntitapahtumaa joka päivä (Cios & al., 1998). Tallennetun tiedon määrän arvioidaankin kaksinkertaistuvan 20–36 kuukauden välein (Frawley & al., 1992; Lyman & Hal, 2003).

Tietomäärien kasvaessa tarvitaan menetelmiä hyödyllisen ja mielenkiintoisen tiedon löytämiseksi. Tällöin puhutaan usein *tiedonlouhinnasta* (*data mining*) tai *tietämyksen keruusta tietokannoista* (*knowledge discovery in databases*; Han & Kamber, 2001). Klusterointi on eräs perustavaa laatua oleva menetelmä saada tietoa käsiteltävän aineiston rakenteesta ja siitä on tullut erittäin aktiivinen tutkimuskohde tiedonlouhinnassa (Han & Kamber, 2001). Klusterointi on tärkeä myös monilla muilla tietojenkäsittelytieteen osa-alueilla sekä useiden empiiristen tieteiden parissa (Jain & al., 1999).

Menetelmät parhaan mahdollisen klusteroinnin tuottamiseksi tunnetaan, mutta klusteroinnin ollessa NP-täydellinen ongelma, joudutaan käytännössä ajan säästämiseksi tinkimään ratkaisun laadusta. Klusterointia onkin tutkittu jo kymmeniä vuosia ja lukuisia klusterointimenetelmiä on esitetty (Jain & al., 1999). Klusteroinnin poikkitieteellisyyden vuoksi klusterointia tarvitsevat useiden eri alojen asiantuntijat, joten tilausta olisi yksinkertaiselle ja yleiskäyttöiselle klusterointimenetelmälle. Tässä tutkielmassa esitellään, alun perin Fräntin ja Kivijärven (2000) esittämä, satunnaistettuun paikallishakuun perustuva klusterointimenetelmä ja sen tehostaminen käyttämällä PDS-menetelmää sekä karsittujen vertailujen hakua. Esitettävä satunnaistettu menetelmä on rakenteeltaan yksinkertainen ollen silti tehokkaampi kuin esimerkiksi tunnettu ja yleisesti käytetty K-means (McQueen, 1967). Tehokkuus on määritelty menetelmän saavuttamana parannuksena aikayksikköä kohden.

Paikallishaun satunnaisuuden havaitaan olevan sekä hyvä että huono asia. Satunnaisuuden etuna on, että toteutus saadaan pidettyä yksinkertaisena mutta silti melko tehokkaana ja että ratkaisuun tehtävien satunnaisten muutosten ansiosta haku ei jää helposti jumiin. Toisaalta satunnaisuus aiheuttaa turhaa työtä, koska tuotetaan paljon ratkaisuja, jotka eivät paranna ratkaisua. Tämä on potentiaalinen tehostuskohde, joten paikallishaun tehokkuutta on tässä tutkielmassa pyritty parantamaan tekemällä paikallishausta deterministisempi. Deterministisyydellä pyritään siis siihen, että paikallishaku löytäisi useammin ja toivon mukaan myös nopeammin parannusta, jolloin saavutettaisiin parempi tehokkuus. Deterministisyys lisää väistämättä toteutuksen kompleksisuutta. Lisäksi riskinä deterministisessä versiossa on, että haku voi mahdollisesti kyetä parantamaan ratkaisua vain tiettyyn pisteeseen saakka.

Tutkielman rakenne on seuraava. Aluksi luvussa 2 kerrotaan yleisesti klusteroinnista ja eri klusterointimenetelmistä sekä esitellään tutkielmassa käytettävät aineistot. Luku 3 esittelee ensin paikallishaun idean yleisellä tasolla ja sen jälkeen paikallishaun käytön klusterointiongelman ratkaisemiseksi. Luvussa 4 tutkitaan paikallishakuun pohjautuvan klusterointimenetelmän toimintaa, luvuissa 5 ja 6 paneudutaan sen tehostamiseen. Tutkielman päättää lyhyt yhteenveto.

2. Klusterointi

Annettujen aineellisten tai abstraktien alkioiden jaottelemista ryhmiin samankaltaisuuden perusteella kutsutaan klusteroinniksi (Han & Kamber, 2001). Asioiden ja ilmiöiden jaottelu ryhmiin on ihmiselle erittäin luonteenomaista toimintaa. Oppiihan ihminen jo aikaisessa lapsuudessaan erottamaan kissan koirasta sekä löytämään eron kasvi- ja eläinkunnan väliltä jatkaen läpi elämänsä tämän alitajuisen jaottelun tarkentamista.

Klusteroinnilla on pitkä historia useissa empiirisissä tieteissä, kuten esimerkiksi biologiassa, psykiatriassa, psykologiassa, lääketieteessä, arkeologiassa, geologiassa, sosiologiassa ja markkinoinnissa, joissa klusterointia käytetään datan analysointiin (Jain & al., 1999). Biologiassa klusterointia on voitu käyttää kasvi- ja eläinluokittelujen muodostamisessa ja populaatioiden sisäisen rakenteen tutkimisessa (Han & Kamber, 2001). Esimerkiksi Fränti ja muut (2000) ovat käyttäneet klusterointia bakteerien luokitteluun.

Psykiatriassa klusterointia on käytetty muun muassa itsemurhaan taipuvaisten mielenterveyspotilaiden luokitteluun. Koska mielenterveysongelmat ovat monimutkaisempia kuin muut terveyteen liittyvät ongelmat, on psykiatriassa yleisemminkin osoitettu mielenkiintoa klusteroinnin soveltamiseksi nykyisten ongelmaluokkien tarkentamiseksi tai jopa uudelleen määrittelemiseksi (Everitt, 1993).

Markkinoinnissa voidaan kuluttajien ostokäyttäytymistä klusteroimalla löytää erilaisia kuluttajaryhmiä. Tätä tietoa voidaan edelleen hyödyntää suunniteltaessa täsmämainontaa (Han & Kamber, 2001). Markkinointitutkimuksissa halutaan yleensä kerätä tietoa erilaisilta kuluttajaryhmiltä. Klusteroimalla voidaan kaupungit tai kaupunginosat jakaa samankaltaisiin ryhmiin, jolloin voidaan paremmin varmistua kyselyiden kohdistumisesta tasapuolisesti kaikkiin kuluttajaryhmiin (Everitt, 1993).

Klusteroinnilla on useita sovelluksia myös tietojenkäsittelytieteen alalla, sillä klusterointi on tärkeä työkalu muun muassa *hahmontunnistuksessa* (*pattern recognition*; Theodoridis & Konstantinos, 2003), *tietoanalyysissa* (*data analysis*; Kiers & al., 2000), *kuvankäsittelyssä* (*image processing*; Petrou, 1999) sekä *tiedonlouhinnassa* (*data mining*; Han & Kamber, 2001). Seuraavaksi luodaan yleiskatsaus klusterointiin. Ensin määritellään klusterointiongelma ja tämän jälkeen esitellään erilaisia lähestymistapoja klusterointiongelman ratkaisemiseksi.

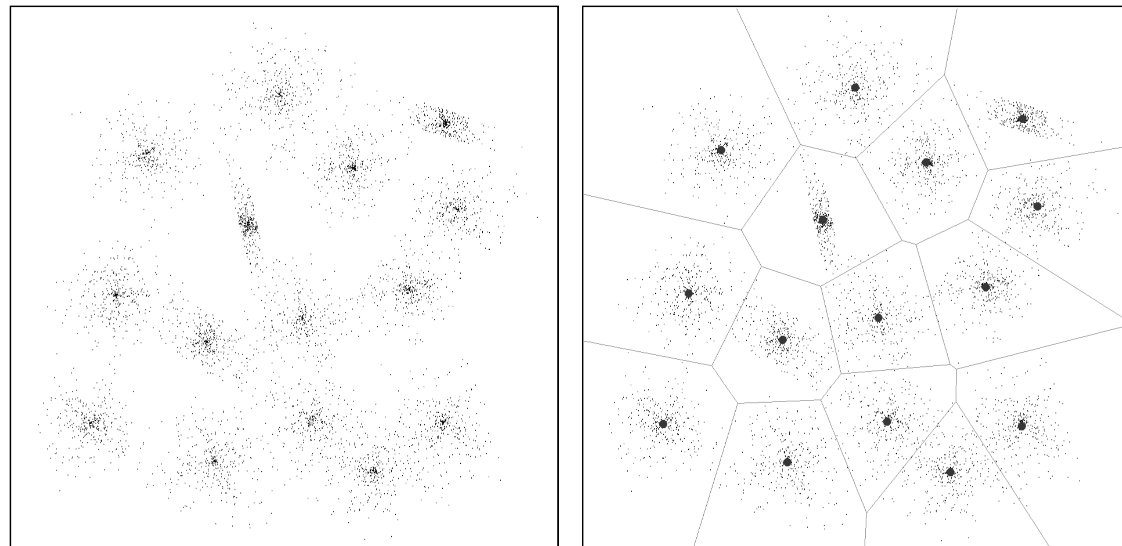
2.1. Perusteet

Klusteroinnissa (*clustering*) pyritään aineiston alkiot jakamaan ryhmiin siten, että kunkin ryhmän sisältämät alkiot ovat mahdollisimman samanlaisia ja että ryhmät keskenään ovat mahdollisimman erilaisia (Kaufman & Rousseeuw, 1990; Everitt, 1993). Muodostettuja ryhmiä kutsutaan klustereiksi. *Klusteri* (*cluster*) määritellään aineiston epätyhjänä osajoukkona. Lisäksi vaaditaan, ettei millään kahdella klusterilla ole yhteisiä alkioita. Poikkeuksena tästä on *sumeaan logiikkaan* (*fuzzy logic*) perustuva klusterointi, jossa kukin alkio voi kuulua useampaan kuin yhteen klusteriin (Jain et al., 1999). Tässä tutkielmassa jätämme kuitenkin huomiotta sumeaan logiikkaan perustuvat klusterointimenetelmät.

On hyvä tehdä ero klusteroinnin ja erotteluanalyysin välille, sillä nämä ovat kaksi eri asiaa. *Erotteluanalyysi* (*discriminant analysis*) on tyypiltään *ohjattua luokittelua* (*supervised classification*) ja siinä aineiston alkiot jaetaan ennalta annettuihin luokkiin (Jain & al., 1999). Erotteluanalyysia kut-

sutaan yleisesti vain *luokitteluksi* (*classification;* Theodoris & Konstantinos, 2003). Klusteroinnissa luokitus syntyy klusteroinnin tuloksena ja näin ollen klusterointi on tyypiltään *ohjaamatonta luokittelua* (*unsupervised classification*; Jain & al., 1999).

Klusteroitava *aineisto* voi olla esimerkiksi joukko kuvia, jotka halutaan järjestää sisältönsä mukaan (Chen & al., 2000), geenien *ekspressiodataa*, jota klusteroimalla halutaan löytää toiminnaltaan samankaltaiset *geenit* (Seal & al., 2005), tai ohjelmakooditiedostoja, joita klusteroimalla halutaan saada käsitys ohjelmiston rakenteesta (Andritsos & Tzerpos, 2005; Shoukoufandeh & al., 2005). Jotta klusterointi olisi mielekästä, oletamme, että klusteroitavalla aineistolla on jonkinlainen rakenne. Kuvassa 2.1 on esitetty esimerkki, jossa klusteroitavana aineistona on 5000 tason pistettä. Muodostetut 15 klusteria on osoitettu rajaamalla syntyneet klusterit viivoilla, jotka on saatu piirtämällä *Voronoin diagrammi* klustereiden keskipisteiden eli sentroidien suhteen (Aurenhammer, 1991).



Kuva 2.1: Eräs aineisto ja sen klusterointi.

Aineiston alkioiden jako M klusteriin määrää aineiston erään *M-osituksen* (*M-partition*). Aineiston, jossa on N alkiota, erilaisten M-ositusten lukumäärän ilmaisee *toisen lajin Stirlingin luku* (*Stirling number of the second kind*; Graham & al., 1994), ja se voidaan Späthin (1985) mukaan laskea kaavalla:

$$\left\{ \begin{matrix} N \\ M \end{matrix} \right\} = \frac{1}{M!} \sum_{j=1}^{M} (-1)^{M-j} \binom{M}{j} j^N .$$

Esimerkiksi 100 alkion erilaisia 7-osituksia on noin 10^{81}, joka on samaa luokkaa kuin havaittavissa olevan universumin atomien lukumäärä (Tyson, 1994). Erilaisia osituksia on siis useimmiten liikaa, jotta optimaalinen klusterointi voitaisiin etsiä kokeilemalla kaikki mahdolliset ositukset. Tehokkaimmatkin optimaalisen ratkaisun tuottavat algoritmit, kuten esimerkiksi Fräntin ja muiden (2002) esittämä algoritmi, ovat aikavaativuudeltaan eksponentiaalisia, sillä klusterointi kombinatorisessa muodossaan on *NP-täydellinen* ongelma (Garey et al., 1982). Näin ollen on käytännössä käytettävä klusterointialgoritmeja, jotka eivät takaa optimaalista ratkaisua.

2.2. Klusterointiongelman määrittely

Jainin ja Dubesin (1988) mukaan klusterointiongelma yleisessä tapauksessa koostuu seuraavista osaongelmista:

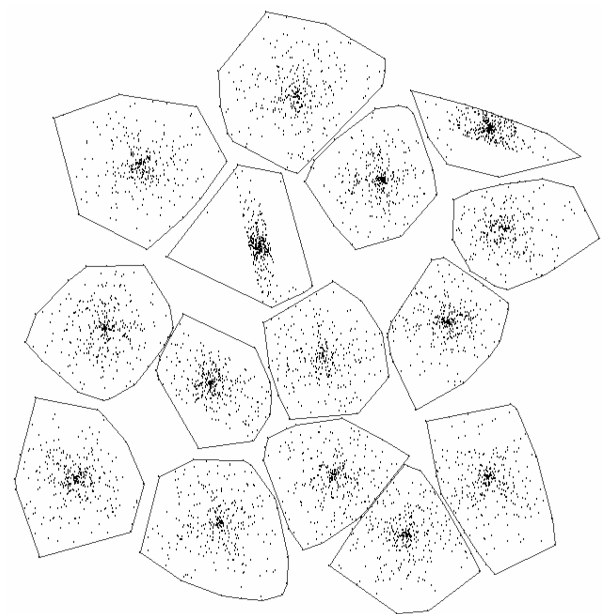
- Klusteroitavan aineiston alkioiden esitysmuodon valinta.
- Klusteroitavan aineiston alkioiden välisen läheisyysmitan määrittely.
- Klusterointialgoritmin valinta ja klusteroinnin suorittaminen koostuen klustereiden lukumäärän ratkaisemisesta ja klusterien muodostamisesta.
- Klusteroinnin tulosten esittäminen.
- Klusteroinnin tulosten arviointi.

Koska klusteroitava aineisto on usein puutteellista, ristiriitaista ja *kohinaista* (*noisy*), joudumme käytännössä lisäksi täyttämään puuttuvat arvot, korjaamaan esiintyvät ristiriitaisuudet sekä havaitsemaan ja poistamaan esiintyvän kohinan (Han & Kamber, 2001).

Määrittelemme jatkossa *klusterointiongelman* seuraavasti. Oletamme aineiston koostuvan N alkiosta ja alkioiden olevan K-ulotteisen *euklidisen avaruuden vektoreita*, joiden läheisyysmittana käytämme euklidista etäisyyttä. Aineisto X koostuu N alkiosta $x_1,\ldots,x_N$ ja kuhunkin alkioon x_i liittyy K *attribuuttia* $x_i^1,\ldots,x_i^K$, jotka ovat *reaalilukuja*. Alkioiden x_i ja x_j välisen *euklidisen etäisyyden* $d(x_i,x_j)$ laskeminen on esitetty kaavassa 2.1.

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{K} \left(x_i^k - x_j^k\right)^2} \qquad \textbf{(2.1)}$$

Olkoon M klustereiden lukumäärä. Klusteroinnin tulos on tällöin aineiston M-ositus eli aineiston alkioiden jako epätyhjiin joukkoihin eli klustereihin $T_1,\ldots,T_M$ siten, että kukin alkio kuuluu yhteen joukoista T_i. Ositus voidaan esittää *kuvauksena* (*mapping*) $P = \{p_1,\ldots,p_N\}$, joka määrittelee kullekin alkiolle, mihin klusteriin kyseinen alkio kuuluu. Toisin sanoen pätee $p_i = j$, jos ja vain jos alkio x_i kuuluu klusteriin T_j. Osituksesta puhuttaessa tarkoitamme jatkossa kuvauksen P avulla ilmaistua ositusta. Kuvan 2.1 aineiston 15-ositus on esitetty kuvassa 2.2 piirtämällä klustereiden *konveksit peitteet* (*convex hulls*; Bentley & al., 1982). Kuvassa 2.2 on siis muodostettu kullekin klusterille reunaviiva, jolloin kaikki kyseiseen klusteriin kuuluvat alkiot ovat reunaviivan rajaaman alueen sisäpuolella.

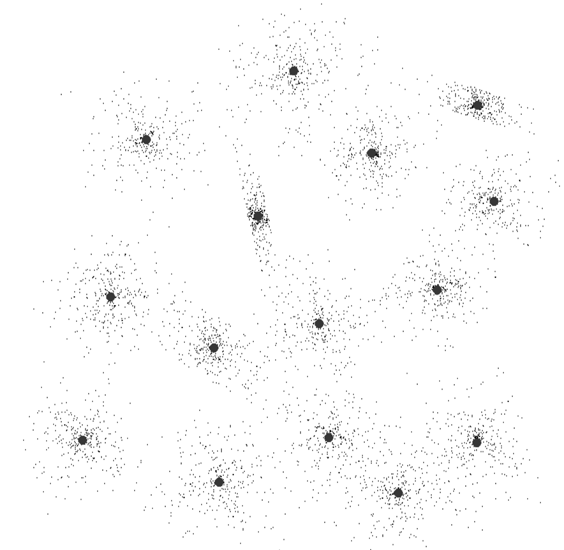


Kuva 2.2: Osituksen esitys konveksien peitteiden avulla.

Vaihtoehtoinen tapa osituksen esittämiseen on käyttää *klustereiden edustajia* (*cluster representatives*) $C = \{c_1,\ldots,c_M\}$. Klusterin edustaja c_j on alkio, joka pyrkii olemaan mahdollisimman hyvä yleistys klusterin T_j alkioista. Euklidista etäisyyttä käytettäessä klusterin edustajaksi on luontevaa valita kunkin *klusterin sentroidi* (*cluster centroid*) eli klusterin sisältämien alkioiden keskiarvo, sillä se minimoi keskineliövirheen käytettäessä euklidista etäisyyttä. Klusterin sentroidi lasketaan kaavalla 2.2.

$$c_m = \frac{1}{|T_m|} \sum_{x_i \in T_m} x_i \qquad \textbf{(2.2)}$$

Kuvassa 2.3 on esitetty kuvan 2.1 aineiston 15-ositus klusterien sentroidien avulla. Sentroideja eli klustereiden keskipisteitä on merkitty mustilla palloilla.



Kuva 2.3: Osituksen esitys klusterien sentroidien avulla.

2.3. Klusteroinnin arviointi

Klusteroinnin hyvyyden arvioimiseksi voimme laskea muodostuvan virheen, kun kukin klusteri ilmaistaan klusterin sentroidilla sen sijaan, että ilmaistaisiin kaikki kyseiseen klusteriin kuuluvat alkiot. Virhe on tällöin kullekin alkiolle sama kuin sen etäisyys klusterinsa sentroidiin. Virheiden summaa jaettuna attribuuttien ja alkioiden lukumäärän tulolla kutsutaan *keskivirheeksi* (*MAE, mean absolute error*; Kivijärvi, 2004). Vastaavasti kun virhe kullekin alkiolle on etäisyyden neliö, virheiden summaa jaettuna attribuuttien ja alkioiden lukumäärän tulolla kutsutaan *keskineliövirheeksi* (*MSE, mean square error*; Kivijärvi, 2004). Keskivirheen laskeminen on esitetty kaavassa 2.3 ja keskineliövirheen laskeminen kaavassa 2.4. Käytämme jatkossa klusteroinnin hyvyyden arviointiin keskineliövirhettä, sillä se painottaa kaukana sentroidista olevia alkioita, mikä on yleensä toivottua.

$$MAE = \frac{1}{NK} \sum_{i=1}^{N} d(x_i, c_{p_i}) \qquad \textbf{(2.3)}$$

$$MSE = \frac{1}{NK} \sum_{i=1}^{N} d(x_i, c_{p_i})^2 \qquad \textbf{(2.4)}$$

Minimoitaessa keskineliövirhettä riippuvat sentroidit ja ositus toisistaan siten, että kun toinen on annettu, voidaan puuttuva muodostaa optimaalisesti (Fränti & Kivijärvi, 2000). Jos aineiston alkioiden ositus on annettu, saadaan optimaaliset sentroidit laskettua ottamalla kunkin klusterin alkioiden keskiarvovektori. Vastaavasti, jos sentroidit on annettu, saadaan optimaalinen ositus muodostettua sijoittamalla kukin alkio siihen klusteriin, jonka sentroidi on lähinnä kyseistä alkiota. Gershon ja Grayn (1992) mukaan näitä saatuja optimaalisuusehtoja kutsutaan sentroidi- sekä lähin naapuri –ehdoksi. *Sentroidiehdon* (*centroid condition*) laskeminen, eli optimaalisten sentroidien selvittäminen osituksen perusteella, on esitetty kaavassa 2.5. *Lähin naapuri –ehdon* (*nearest neighbour condition*) laskeminen, eli optimaalisen osituksen selvittäminen sentroidien perusteella, on esitetty kaavassa 2.6.

$$c_j = \frac{\sum_{p_i = j} x_i}{\sum_{p_i = j} 1} \quad \forall\ j \in [1, M] \tag{2.5}$$

$$p_i = \min_{1 \le j \le M} d(x_i, c_j)^2 \quad \forall\ i \in [1, N] \tag{2.6}$$

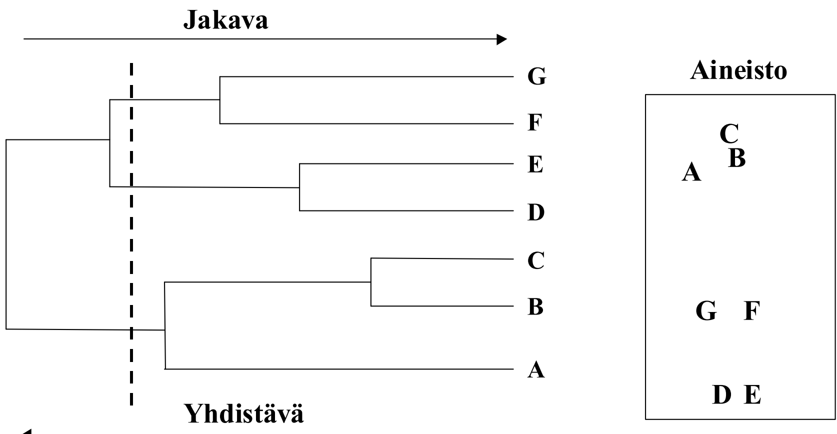
Ryhdymme näiden oletusten vallitessa tarkastelemaan lähemmin eri tapoja ratkaista klusterointiongelma. Kirjallisuudessa on esitetty lukemattomia eri klusterointimenetelmiä. Klusterointimenetelmän valinnassa on otettava huomioon klusteroitavan aineiston tyyppi sekä klusteroinnin käyttötarkoitus (Han & Kamber, 2001). Mikäli klusterointia käytetään *tutkivana menetelmänä* (*exploratory tool*), voidaan aineistoa klusteroida useilla eri menetelmillä, jotta nähdään, mitä aineistosta paljastuu (Han & Kamber, 2001; Jain & al., 1999).

Klusterointimenetelmien luokittelemiseksi ei kirjallisuudesta löydy mitään yhtenäistä linjaa. Esimerkiksi Jain ja muut (1999), Han ja Kamber (2001), Theodoridis ja Konstantinos (2003), Ma ja Chow (2004) sekä Virmajoki (2004) esittävät kukin erilaisen luokittelun klusterointimenetelmille. Tässä tutkielmassa esitettävässä klusterointimenetelmien luokittelussa on käytetty soveltuvin osin edellä mainittuja luokitteluja. Klusterointimenetelmät voidaan karkeasti jakaa hierarkkisiin ja osittaviin menetelmiin.

2.3. Hierarkkiset menetelmät

Hierarkkiset menetelmät tuottavat aineistolle useita osituksia, jotka muodostavat puumaisen hierarkian. Hierarkkiset menetelmät jaetaan *jakaviin* (*divisive*) ja *yhdistäviin* (*agglomerative*; Kaufman & Rousseeuw, 1990; Everitt, 1993). Jakavissa menetelmissä kaikki alkiot ovat aluksi samassa klusterissa ja tämän jälkeen klustereita jaetaan kahtia kunnes saavutetaan haluttu määrä klustereita tai kunnes kukin alkio on omassa klusterissaan. Yhdistävissä menetelmissä jokainen alkio on aluksi omassa klusterissaan ja tämän jälkeen klustereita yhdistellään kunnes saavutetaan haluttu määrä klustereita tai kunnes kaikki alkiot ovat samassa klusterissa. (Jain & al., 1999; Kauffman & Rousseeuw, 1990: Everitt, 1993)

Hierarkkisen klusteroinnin tulokset voidaan esittää *dendrogrammina* (Jain & al., 1999). Kuvassa 2.4 on esitetty aineisto, joka koostuu alkioista A, B, C, D, E, F, G, jotka muodostavat kolme klusteria {A,B,C}, {D,E} ja {F,G}. Kuvassa 2.4 on lisäksi esitetty dendrogrammina aineiston hierarkkinen klusterointi. Jakavan algoritmin tapauksessa kuvan 2.4 dendrogrammia luetaan vasemmalta oikealle ja yhdistävän klusteroinnin tapauksessa oikealta vasemmalle. Katkaisemalla dendrogrammi sopivalta tasolta saadaan aineiston klusterointi halutulla klusterien määrällä. Kuvassa 2.4 on dendrogrammi katkaistu katkoviivalla siten, että saamme aineiston klusteroinnin kolmeen klusteriin, jotka ovat {A,B,C}, {D,E} ja {F,G}. Huomattavaa on, että yhdistävä ja jakava menetelmä eivät välttämättä aina tuota samaa dendrogrammia.



Kuva 2.4: Hierarkkisen klusteroinnin tuloksen esittäminen dendrogrammina (Jainia ja muita (1999) mukaillen).

Jakavien menetelmien tapauksessa on ratkaistava, miten valitaan jaettava klusteri sekä miten valittu klusteri jaetaan. Jaettava klusteri voidaan valita esimerkiksi klusterin koon (Kaufmann & Rousseuw, 1990) tai *hajonnan* (*variance*; Wu & Zhang, 1991; Fränti & al., 1997a) mukaan. Jako voidaan tehdä esimerkiksi valittua ulottuvuutta (Heckbert, 1982) tai *pääakselia* (*principal axis*; Fränti & al., 1997a) pitkin.

Jakavia menetelmiä voidaan jaotella edelleen monoteistisiin ja polyteistisiin menetelmiin sen mukaan, kuinka ne suhtautuvat klusteroitavien alkioiden attribuutteihin (Jain & al., 1999). *Monoteistiset* (*monothetic*) menetelmät suorittavat erottelun yhden attribuutin arvojen perusteella ja *polyteistiset* (*polythetic*) puolestaan perustuvat useiden attribuuttien arvojen perusteella tehtävään jakoon.

Yhdistävien menetelmien tapauksessa yhdistettävien klustereiden valinta on keskeisintä, sillä varsinainen klustereiden yhdistäminen on melkoisen suoraviivaista. Yhdistettävät klusterit voidaan valita esimerkiksi klustereiden lähimpien (*single linkage*; Sneath & Sokal, 1973) tai kauimpien (*complete linkage*; King, 1973) alkioiden perusteella. Yhdistettävät klusterit on myös mahdollista valita siten, että klustereiden yhdistäminen heikentää ratkaisua vähiten (*Ward's method*; Ward, 1963). Hierarkkisista menetelmistä yhdistävät menetelmät ovat yleisemmin käytettyjä ja helpompia toteuttaa kuin jakavat menetelmät (Everitt, 1993; Virmajoki, 2004). Virmajoki (2004) väittää lisäksi yhdistävien menetelmien saavuttavan jakavia menetelmiä parempia tuloksia.

Hierarkkisten menetelmien hyvä puoli on, että menetelmien suorituksen aikana saadaan aineiston ositukset kaikille mahdollisille klustereiden lukumäärille. Hierarkkisten menetelmien heikkous on kuitenkin tietty ahneus, sillä klustereita yhdistellään tai jaetaan sen mukaan, mikä juuri kyseisellä hetkellä vaikuttaa parhaalta ratkaisulta. Tehtyjä huonoja päätöksiä ei kuitenkaan pysty myöhemmin perumaan (Kaufmann & Rousseeuw, 1990).

2.4. Osittavat menetelmät

Osittavat menetelmät tuottavat aineistolle vain yhden osituksen (Kauffman & Rousseeuw, 1990). Osittaville algoritmeille on vaikea löytää hierarkkisten algoritmien tapaan mitään yleisesti hyväksyttyä luokittelua. Esimerkiksi Jainin ja muiden (1999) mukaan osittavat algoritmit jaetaan luokkiin *neliövirhe* (*square error*), verkkoteoreettiset (*graph theoretic*), *sekoitteen etsintä* (*mixture*

resolving) ja *moodin haku* (*mode seeking*). Virmajoen (2004) mielestä osittavat algoritmit jaetaan luokkiin *heuristiset* (*heuristic*), optimointi (optimization) ja verkkoteoreettiset (graph theoretic). Manin ja Chown (2004) mukaan ei-hierarkkiset algoritmit jaetaan luokkiin *osittavat* (*partitioning*), tiheysperustaiset (density-based), ristikkoperustaiset (grid-based) sekä muut (others). Kussakin edellä mainituista lähteistä osittaville algoritmeille esitetään juuri kyseiseen tarkoitukseen sopiva luokittelu. Mielestäni osittavat menetelmät tulee jakaa optimointimenetelmiin sekä verkkoteoreettisiin, tiheysperustaisiin, ristikkoperustaisiin ja malliperustaisiin menetelmiin.

Optimointimenetelmät (*optimization*) jakavat aineiston haluttuun määrään klustereita pyrkien luomaan laadullisesti mahdollisimman hyvän ratkaisun (Jain & al., 1999; Kaufmann & Rousseeuw, 1990; Everitt, 1993). Ratkaisun laatua mitataan yleensä käyttäen aiemmin määriteltyä keskineliövirhettä. Luvussa 3 esitettävät paikallishakuun perustuvat klusterointimenetelmät kuuluvat optimointimenetelmien luokkaan.

K-means (McQueen, 1967) on eräs tunnetuimpia klusterointimenetelmiä ja se kuuluu optimointimenetelmien luokkaan. K-means käyttää kahta optimaalisuuskriteeriä, joita vuorottelemalla sen onnistuu optimoida ratkaisu paikallisesti (Selim & Ismail, 1984). Menetelmä kuvataan useissa lähteissä (Han & Kamber, 2001; Jain & al., 1999; Kaufmann & Rousseeuw, 1990) ja sitä käytetään yleisesti myös toisten menetelmien osana. K-means tulee olemaan tärkeä osa myös luvussa 3 esitettävää satunnaistettuun paikallishakuun perustuvaa klusterointimenetelmää.

Verkkoteoreettiset (*graph-theoretical*) menetelmät muodostavat alkuperäisestä aineistosta verkon ja sen jälkeen ratkaisevat ongelman käyttäen muodostettua verkkoa. Eräs mahdollinen tapa on muodostaa aineistosta *pienin virittävä puu* (*minimal spanning tree*), jossa solmuina on aineiston alkiot ja kaarten painoina alkioiden välimatkat (Jain & al., 1999). Tämän jälkeen poistamalla aina painavin kaari voidaan muodostaa klusterointeja eri klusterien lukumäärille. Näin ollen verkkoteoreettisilla menetelmillä on selkeitä yhtymäkohtia aiemmin esitettyihin hierarkkisiin menetelmiin.

Tiheysperustaiset (*density-based*) menetelmät käsittävät klusteroinnin harvojen alueiden erottamien tiheiden alueiden etsimiseksi (Ma & Chow, 2004). Tiheysperustaisissa menetelmissä aineiston alkiot pyritään jakamaan joukoiksi sellaisia alueita, joiden sisältämät alkiot ovat mahdollisimman tiheässä (Han & Kamber, 2001). Hanin ja Kamberin (2001) mukaan tiheysperustaiset menetelmät löytävät mielivaltaisen muotoisia ja kokoisia klustereita sekä sietävät hyvin kohinaa. Huonona puolena Han ja Kamber (2001) mainitsevat, että tiheyden käsitteen määrittely vaatii tuntemusta aineistosta.

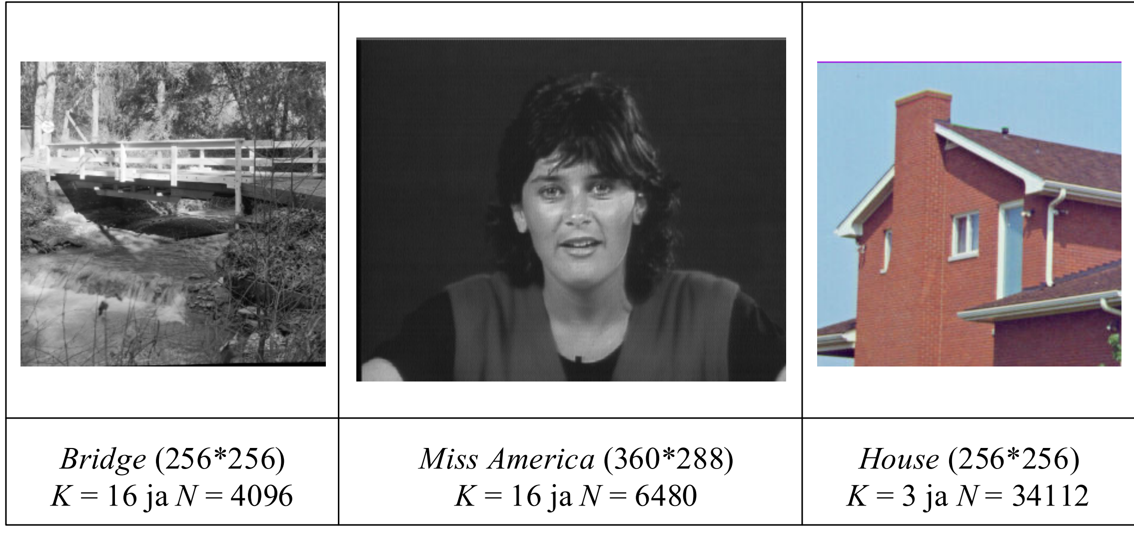
Ristikkoperustaiset (*grid-based*) menetelmät jakavat aineiston muodostaman *vektoriavaruuden* äärelliseen määrään *soluja* (*cell*), jotka yhdessä muodostavat ristikkorakenteen (Han & Kamber, 2001). Tämän jälkeen kaikki klusterointitoimet tehdään ristikkorakenteessa. Han ja Kamber (2001) mainitsevat ristikkoperustaisten menetelmien hyvänä puolena nopean suoritusajan, sillä aikavaativuus ristikkoperustaisissa menetelmissä on tyypillisesti suhteessa solujen määrään, eikä suhteessa aineiston alkioiden määrään.

Malliperustaiset (*model-based*) menetelmät pyrkivät hakemaan aineistolle mahdollisimman hyvän selityksen jonkin matemaattisen mallin avulla (Han & Kamber, 2001). Malli voi olla tilastotieteellinen, jolloin aineiston oletetaan olevan *todennäköisyysjakaumien* (*propability distributions*) tuottamaa, tai *neuroverkkoon* (*neural network*) pohjautuva. Kohosen (1982) esittämä *itseorganisoituva kartta* (*self-organizing map*) on eräs neuroverkkoon pohjautuva malliperustainen menetelmä.

Tässä esitetty klusterointimenetelmien luokitus ei ole sekään mitenkään aukoton, sillä useat klusterointimenetelmät ovat yhdistelmiä edellä esitetyistä lähestymistavoista. Tästä esimerkkinä Kaukorannan ja muiden (1998) esittämä *Split-and-merge* –menetelmä, jossa vuoroin jaetaan ja yhdistellään klustereita hierarkkisten menetelmien tapaan, mutta varsinainen menetelmä on pikemminkin optimointimenetelmä. Vastakkaisen esimerkin tarjoaa Likasin ja muiden (2003) esittämä menetelmä, joka on hierarkkinen jakava menetelmä, jossa on käytetty K-meansia menetelmän osana. Ma ja Chown (2004) esittämä menetelmä on yhdistelmä tiheys- ja ristikkoperustaisista menetelmistä, joten eri osittavien menetelmien luokkiakin on mahdollista yhdistellä.

2.5. Klusteroitavat aineistot

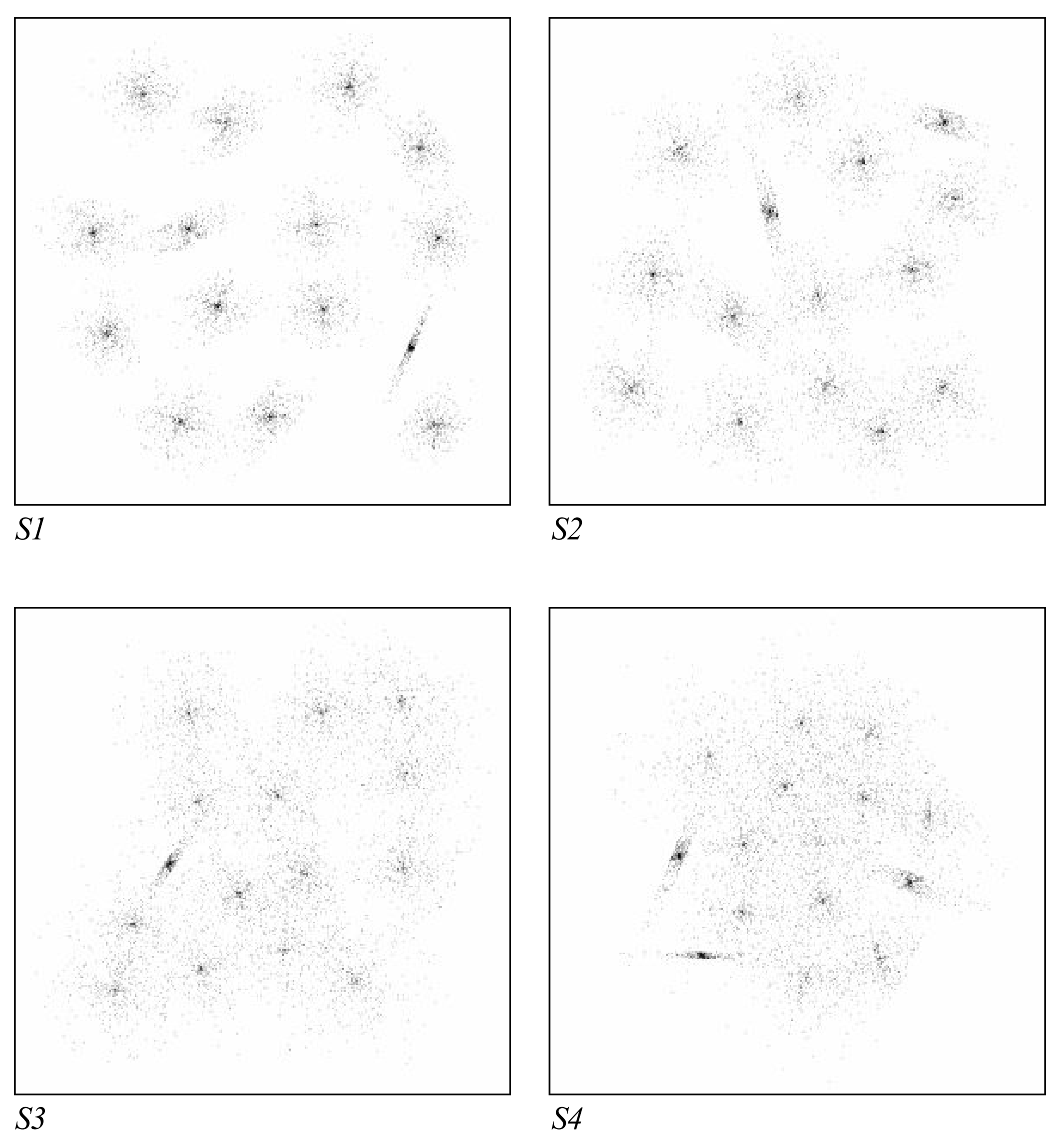
Tässä tutkielmassa käytetään klusteroitavina aineistoina kuvien pohjalta luotuja aineistoja, synteettisia aineistoja sekä puheesta luotuja aineistoja. Kuvassa 2.5 on esitetty käytettävien kuva-aineistojen lähdekuvat. Kuvaan on merkitty kunkin kuvan pohjalta muodostetun aineiston vektoreiden ulottuvuus *K* sekä aineiston vektoreiden lukumäärä *N*. Aineiston *Bridge* alkiot ovat harmaasävykuvan 4x4 *pikselin* kokoisia ei-päällekkäisiä lohkoja ja aineiston *Miss America* alkiot ovat videokuvan kahden peräkkäisen *kehyksen* (*frame*) 4x4 pikselin kokoisia *erotuslohkoja* (*difference blocks*). Aineiston *House* alkiot koostuvat kuvan pikseleiden *RGB-arvoista.* Alkioiden monikertaiset esiintymät on korvattu yhdellä alkiolla ja tiedolla alkion *frekvenssistä*. Kiinnitämme kuva-aineistojen klustereiden määräksi 256.



Bridge (256*256) $K = 16$ ja $N = 4096$	*Miss America* (360*288) $K = 16$ ja $N = 6480$	*House* (256*256) $K = 3$ ja $N = 34112$

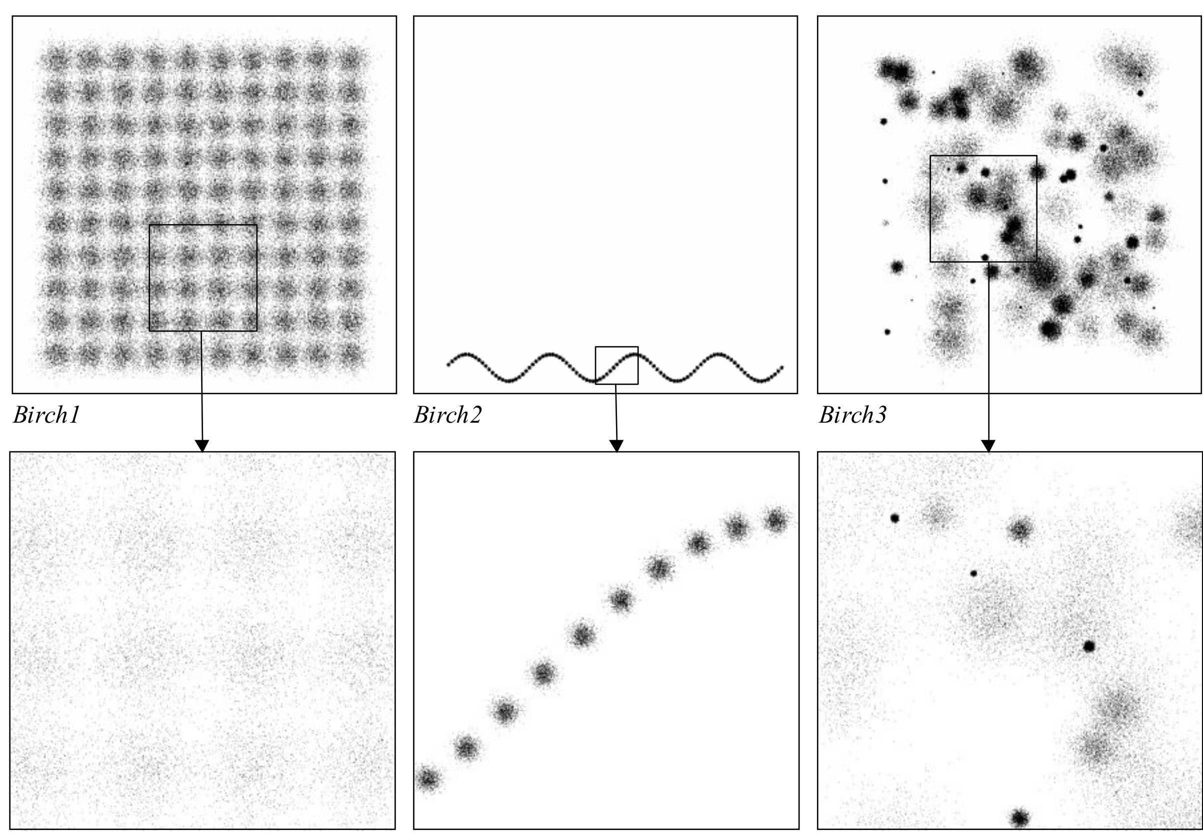
Kuva 2.5: Kuva-aineistojen lähdekuvat.

Kuvassa 2.6 on esitetty *S*-aineistot, jotka ovat synteettisia aineistoja. Kukin aineistoista koostuu 5000 alkiosta, jotka muodostavat 15 klusteria. Aineistot eroavat toisistaan klustereiden päällekkäisyyden ja jakautumisen suhteen.



Kuva 2.6: *S*-aineistot.

Kuvassa 2.7 on esitetty *Birch*-aineistot (Zhang & al., 1997). Kukin aineistoista koostuu 100000 alkiosta, jotka muodostavat 100 klusteria. Aineiston *Birch1* tapauksessa klusterit on sijoitettu säännölliseen taulukkorakenteeseen, aineiston *Birch2* tapauksessa klusterit on sijoitettu sinikäyrälle ja aineiston *Birch3* tapauksessa satunnaisen kokoiset klusterit on sijoitettu satunnaisiin paikkoihin.



Kuva 2.7: *Birch*-aineistot.

Puheaineistojen lähteenä on käytetty *TIMIT* (LDC, 2006) ja *NIST-1999* (NIST, 2006) –puhujatietokantoja, jotka koostuvat useiden henkilöiden puheesta. Kunkin henkilön puheesta *irrotetaan* (*extract*) *piirrevektorit* (*feature vectors*) käyttäen *MFCC* (m*el frequency cepstral coefficents*; Rabiner & Juang, 1993) –menetelmää. Klusteroimalla piirrevektoreiden muodostamaan aineistoa saadaan henkilölle muodostettua *puhujamalli* (*speaker model*), jolle on käyttöä *puhujan tunnistuksessa* (*speaker recognition*; Soong & al., 1987; Kinnunen, 2005).

Taulukossa 2.1 on esitetty yhteenvetona kaikki tutkielmassa käytettävät aineistot. Aineistot on pyritty valitsemaan monipuolisesti, jotta saataisiin realistinen kuva, kuinka tutkittavat menetelmät soveltuvat yleiskäyttöisiksi käytännön klusterointimenetelmiksi.

Taulukko 2.1: Tutkielmassa käytettävien aineistojen yhteenveto.

	Tyyppi	Klustereita (*M*)	Vektoreita (*N*)	Dimensio (*K*)
Bridge	Harmaasävykuva	256	4 086	16
House	RGB-kuva	256	34 112	3
Miss America	Residuaaleja vektoreita	256	6 480	16
Birch1 … *Birch3*	Synteettisesti luotua	100	100 000	2
S1 … *S4*	Synteettisesti luotua	15	5 000	2
TIMIT (630 kappaletta)	Puhetta	Ei tiedossa	5553 … 6497	12
NIST-1999 (536 kappaletta)	Puhetta	Ei tiedossa	1496 … 3518	12

S-aineistoja tullaan käyttämään etupäässä havaintoesimerkeissä, sillä siihen aineistot soveltuvat hyvin kaksiulotteisuutensa ja selkeän rakenteensa ansiosta. *S*-aineistojen selkeästä rakenteesta seuraa, että niille on helppo löytää oikea, optimaalinen klusterointi. *Birch*-aineistot edustavat kokonsa puolesta astetta työläämpää kaksiulotteista klusterointitehtävää. Huomattavaa on, että kaksiulottei-

nen aineisto voidaan luontevasti visualisoida, mutta suurempiulotteinen aineisto voidaan vain korkeintaan projisoida tasoon. Näin ollen kaksiulotteiset aineistot soveltuvat hyvin havaintoesimerkkeihin.

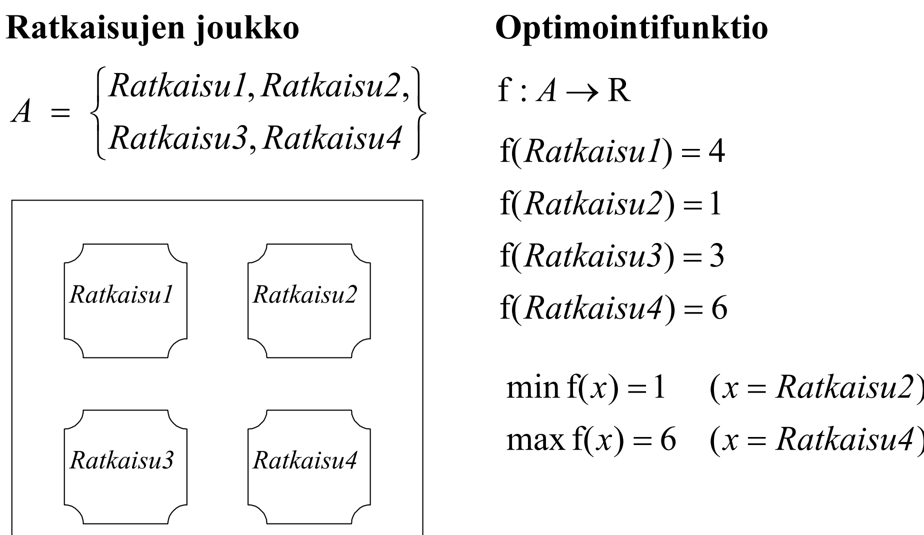
Kuva-aineistot *Bridge*, *House* ja *Miss America* tuovat mukaan lisää ulottuvuuksia. Havaintoesimerkkien aika-laatu –tyyppiset vertailut tullaan tekemään pääosin *Bridge* aineistolle, sillä siinä ei ole havaittavissa selkeitä klustereita toisin kuin esitellyillä kaksiulotteisilla aineistoilla. Kinnusen (2005) mukaan myöskään puheaineistoissa ei ole havaittavissa klustereita, joten klusteroinnin rooli on näillä aineistoilla käytännössä *näytteistää* (*sample*) aineistoa.

Aineistoille suoritettavat testit on ajettu koneella, jossa on neljä *Intel(R) Xeon(TM)* 2.80GHz prosessoria, 1 GB keskusmuistia ja käyttöjärjestelmänä *GNU/Linux*, jossa *ytimen* (*kernel*) versio on *2.6.5-1.358smp*.

3. Paikallishaku

Paikallishakua käytetään kombinatoristen optimointiongelmien ratkaisemiseen. *Kombinatorinen optimointiongelma* (*combinatorial optimization problem*; Papadimitriou & Steiglitz, 1982) voidaan esittää kaikkien mahdollisten ratkaisujen joukkona sekä näihin liittyvänä optimointifunktiona (Aarts & Lenstra, 1997). *Optimointifunktio* on kuvaus ratkaisujen joukosta reaalilukujen joukkoon, jolloin se liittää yhden reaalilukuarvon jokaiseen ratkaisuun. Riippuen siitä, onko kyseessä minimointi- vai maksimointiongelma, ongelman optimaalinen ratkaisu on se ratkaisu, jolla on pienin tai suurin optimointifunktion arvo (Aarts & Lenstra, 1997).

Kuvassa 3.1 on esitetty erään kombinatorisen optimointiongelman ratkaisujen joukko sekä optimointifunktion arvo kullekin ratkaisulle. Kuvasta nähdään, että tässä tapauksessa optimointifunktion pienin arvo on 1 ja suurin arvo on 6. Optimointifunktion arvo on pienin ratkaisulle 2 ja suurin ratkaisulle 4. Näin ollen ratkaisu 2 on optimaalinen ratkaisu, mikäli kyseessä on minimointiongelma, ja vastaavasti ratkaisu 4 on optimaalinen ratkaisu, mikäli kyseessä on maksimointiongelma.



Kuva 3.1: Kombinatorinen optimointiongelma.

Useat kombinatoriset optimointiongelmat ovat *NP-vaikeita* (*NP-hard*; Garey & Johnson, 1979), joten niiden optimaalisen ratkaisun löytämiseksi ei tunneta aikavaativuudeltaan polynomisia algoritmeja. Aarts ja Lenstra (1997) esittävät näiden NP-vaikeiden kombinatoristen optimointiongelmien ratkaisemiseksi kolme tapaa. Voimme ratkaista optimointiongelman käyttämällä *numeroivia* (*enumerative*) menetelmiä, jotka vaativat eksponentiaalisen ajan mutta tuottavat taatusti optimaalisen ratkaisun, tai voimme käyttää *approksimoivia* (*approximation*) menetelmiä, jotka toimivat polynomisessa ajassa ja takaavat ylärajan tuottamansa ratkaisujen etäisyydelle optimaalisesta. Kolmas vaihtoehto optimointiongelman ratkaisemiseksi on käyttää *heuristisia* menetelmiä kuten paikallishakua, jotka yrittävät tuottaa parhaan mahdollisen ratkaisun antamatta kuitenkaan takeita siitä, kuinka hyvän ratkaisun onnistuvat tuottamaan.

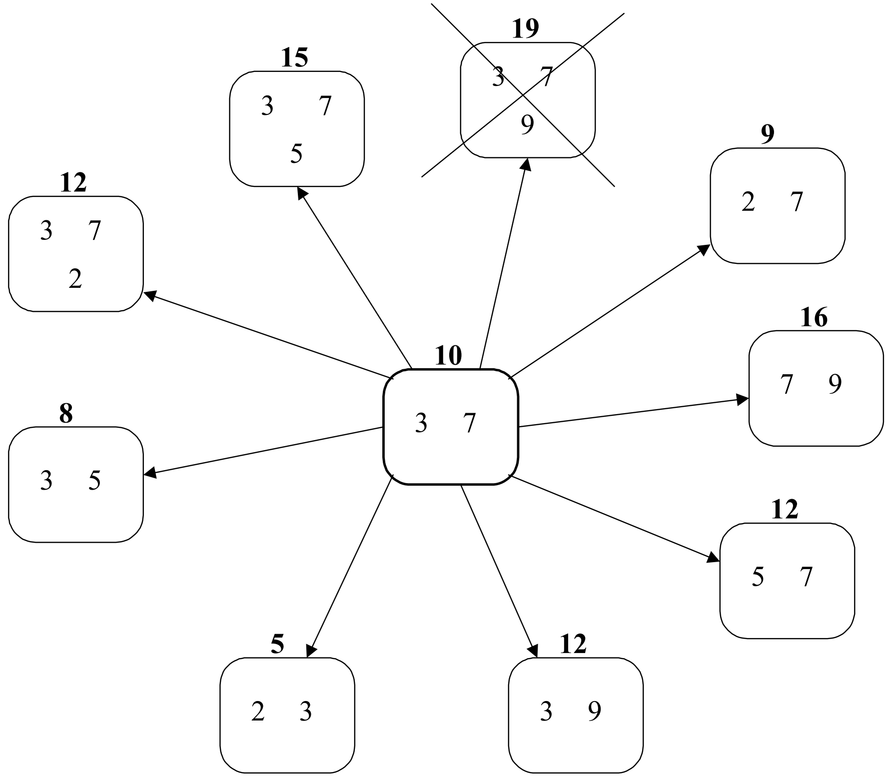
3.1. Paikallishaun perusversio

Paikallishaku (*local search*; Aarts & Lenstra, 1997) on eräs heuristinen menetelmä kombinatoristen optimointiongelmien ratkaisemiseksi. Paikallishaku perustuu naapuruuden käsitteeseen. Kullekin

ratkaisulle on nimittäin määritelty joukko naapuriratkaisuja, jotka ovat jollain tapaa lähellä kyseistä ratkaisua (Aarts & Lenstra, 1997). Naapuriratkaisut voivat olla esimerkiksi helposti johdettavissa kyseisestä ratkaisusta lähtien tai naapuriratkaisuilla ja kyseisellä ratkaisulla voi olla muutoin samankaltainen rakenne.

Havainnollistamme seuraavaksi naapuruuden käsitettä ja optimointifunktiota reppuongelman avulla. R*eppuongelmassa* (*knapsack problem*) on annettu repun koko sekä tavaroiden koot ja tehtävänä on pakata reppu mahdollisimman täyteen tavaroita. Reppuongelman tapauksessa kukin ratkaisu on eräs tavarajoukko. Optimointifunktio puolestaan kertoo kullekin ratkaisulle siihen kuuluvien tavaroiden yhteiskoon. Kysymyksessä on maksimointiongelma sillä rajoituksella, että optimointifunktion arvo eli tavaroiden yhteiskoko ei saa ylittää repun kokoa. Kuvan 3.1 esimerkissä repun koko on 18 ja tavaroiden koot ovat 2, 3, 5, 7 ja 9. Esimerkissä optimointifunktion arvot on merkitty lihavoiduin kirjaimin ja tavaroiden kokoja käytetty tavaroiden niminä, joten repun sisältö ”3 7” tarkoittaa, että repussa on tavarat, joiden koot ovat vastaavasti 3 ja 7.

Kuvan 3.2 esimerkissä ratkaisulle ”3 7” on luotu naapuristo sillä periaatteella, että kaikki ne ratkaisut, jotka saadaan lisäämällä alkuperäiseen ratkaisuun yksi tavara tai korvaamalla yksi tavara toisella, ovat sen naapuriratkaisuja. Huomaamme, että käyttämämme naapuriston luontimenetelmä voi luoda epäkelvollisia ratkaisuja kuten esimerkissä ratkaisu ”3 7 9”, jonka koko ylittää repun koon. Koska epäkelpoja ratkaisuja ei hyväksytä naapuristoon, on kuvassa 2 tämä osoitettu piirtämällä rasti ratkaisun ”3 7 9” päälle.



Kuva 3.2: Reppuongelman ratkaisuja ja niiden optimointifunktion arvot.

Kuvassa 3.3 on esitetty paikallishaun rakenne. Paikallishausssa luodaan ensin lähtöratkaisu, jota tämän jälkeen parannetaan iteratiivisesti. Kullakin iteraatiolla luodaan nykyisestä ratkaisusta lähtien joukko naapuriratkaisuja, joista paras valitaan nykyiseksi ratkaisuksi. Tätä jatketaan joko ennalta

määrätty määrä kertoja tai kunnes parannusta ei enää tapahdu. Esittämäämme versiota paikallishausta, jossa sallitaan ainoastaan ratkaisun paraneminen, kutsutaan tyypillisesti *iteratiivinen parannus* (*iterative improvement*; Aarts & Lenstra, 1997) tai *mäen nousu* (*hill-climbing*; Carson, 2001) –menetelmäksi.

```
Muodosta lähtöratkaisu ja valitse se nykyiseksi ratkaisuksi.
TOISTA
  Muodosta nykyisen ratkaisun pohjalta uudet ratkaisut (naapuristo).
  Arvota uudet ratkaisut.
  Valitse paras ratkaisu nykyiseksi ratkaisuksi.
KUNNES pysähtymisehto saavutettu
```

Kuva 3.3: Paikallishaun rakenne (Fränti & Kivijärvi, 2000).

Paikallishakua toteuttaessa on Fräntin ja Kivijärven (2000) mukaan valittava käytettävä ratkaisun esitysmuoto, naapuristofunktio sekä hakustrategia. Seuraavassa esitettävät näiden vaiheiden kuvaukset ovat Fräntiä ja Kivijärveä (2000) mukaillen, mikäli ei toisin mainita. Ratkaisun esitysmuoto on syytä valita huolella, sillä se määrää haun aikana muokattavat tietorakenteet. Eräs vaihtoehto olisi esittää ratkaisu bittijonona, jolloin naapuriratkaisuja voitaisiin generoida muuttamalla bittijonon satunnaisia bittejä. Tämä ei kuitenkaan ole kovinkaan tehokas tapa ratkaisun parantamiseksi, koska suurin osa bittijonoista ei esitä kelvollista ratkaisua. Lisäksi satunnaisia bittejä muuttamalla luotu ratkaisu ei välttämättä ole mielekkäästi naapuri alkuperäiselle ratkaisulle. Näin luodulla ratkaisulla ei nimittäin välttämättä ole mitään yhteistä alkuperäisen ratkaisun kanssa. Yleensä käytetäänkin ongelmakohtaista ratkaisun esitysmuotoa ja operoidaan suoraan ratkaistavana olevan ongelman käsitteillä.

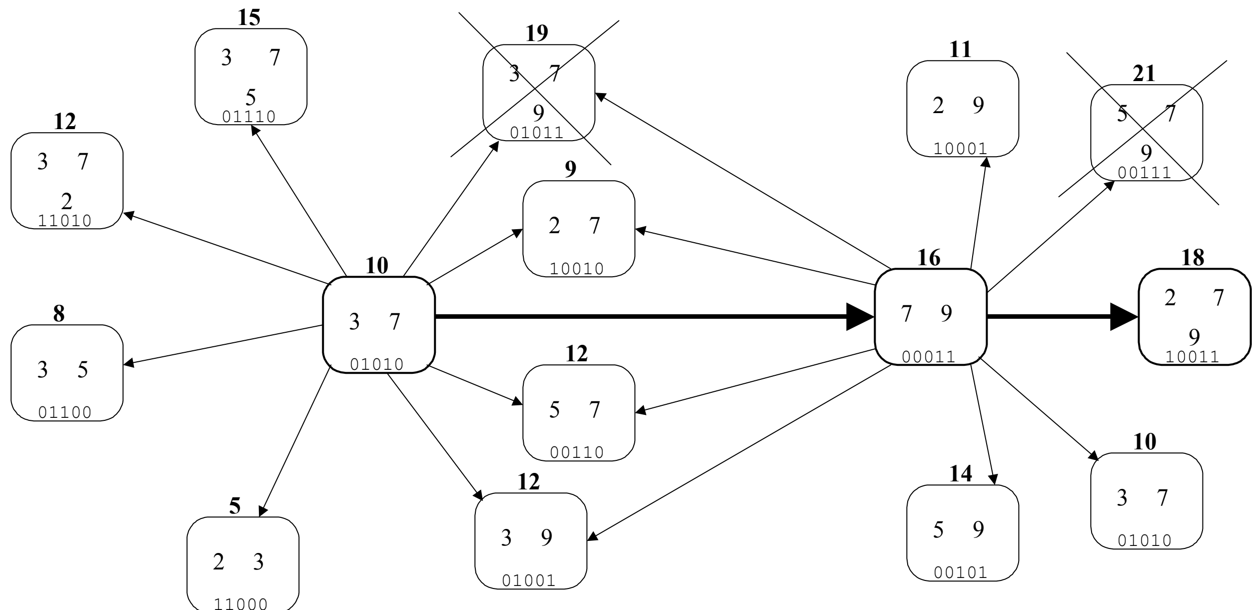
Naapuristofunktio määrää, kuinka uusia ratkaisuja muodostetaan. Naapuristofunktio luo uusia ratkaisuja tekemällä muutoksia nykyiseen ratkaisuun. Muutosten lukumäärän tulee olla toisaalta riittävän pieni, ettei tuhota alkuperäistä ratkaisua kokonaan, mutta toisaalta riittävän suuri, jotta haun olisi mahdollista päästä etenemään. Mikäli naapuriston muodostamisessa on mukana satunnaisuutta, on tällöin kyse *satunnaistetusta paikallishausta* (*randomized local search*; Wegener & Witt, 2005).

Hakustrategia määrää, kuinka uusien ratkaisujen joukosta valitaan paras. Ilmeisin tapa lienee *paras jälkeläinen* (*steepest descent*) –menetelmä, jolloin valitaan ratkaisuista se, jolla on paras optimointifunktion arvo. Vaihtoehtoisesti voidaan esimerkiksi käyttää *ensimmäinen parannus* (*first-improvement*) –menetelmää (Fränti & Kivijärvi, 2000), jolloin valitaan ratkaisuista ensimmäinen, joka parantaa optimointifunktion arvoa, tai *satunnainen jälkeläinen* (*random descent*) –menetelmää (Kivijärvi, 2004), jolloin valitaan satunnainen ratkaisu. Ensimmäinen parannus –menetelmä on olennaisesti sama kuin paras jälkeläinen –menetelmä yhden kokoisella naapuristolla.

Reppuongelman tapauksessa ratkaisu voidaan esittää yksinkertaisesti bittijonona. Käyttämällä aiemmin esitettyä reppuongelmaa, jossa tavaroiden koot olivat 2, 3, 5, 7 ja 9, ja kiinnittämällä tavaroiden järjestykseksi edellä mainitun järjestyksen voimme esittää esimerkiksi ratkaisun ”3 7” bittijonona 01010. Ilmaisemme siis ykkösillä ne tavarat, jotka pakkaamme reppuun. Tässä tapauksessa ratkaisun esittäminen bittijonona on tarkoituksenmukaista, sillä kunkin ratkaisun sisältämä tieto tavaroiden sisältymisestä on koodattavissa mielekkäästi bittijonona. Kuvan 3.2 mukaisesti määrittelemme ratkaisun naapuristoon kuuluviksi kaikki ne ratkaisut, jotka saadaan lisäämällä alkuperäiseen ratkaisuun jokin tavara tai korvaamalla jokin tavara toisella. Naapuriksi ei määritellä ratkaisua,

joka saadaan poistamalla tavara, sillä tavaran poistaminen taatusti huonontaa ratkaisua. Bittijonotasolla tavaran lisääminen vastaa yhden bitin vaihtamista nollasta ykköseksi ja tavaran korvaaminen toisella yhden bitin kääntämistä ykkösestä nollaksi ja toisen bitin kääntämistä nollasta ykköseksi.

Kuvan 3.4 esimerkissä on kuvan 3.2 mukaisesti repun koko 18, tavaroiden koot 2, 3, 5, 7 ja 9 sekä paikallishaun lähtöratkaisuna satunnaisesti valittu lähtöratkaisu ”3 7”. Kuvassa 3.4 käytetyt merkinnät vastaavat kuvassa 3.2 käytettyjä paitsi, että kukin ratkaisu on esitetty sekä bittijonona että luettelemalla pakattavat tavarat. Paikallishaku lähtee siis liikkeelle ratkaisusta ”3 7”, jolle muodostetaan naapuristo. Hakustrategiana on käytetty paras jälkeläinen –menetelmää, joten naapuristosta valitaan ratkaisu, jolla on mahdollisimman suuri optimointifunktion arvo eli tässä tapauksessa ratkaisu ”7 9”. Tämän jälkeen ratkaisulle ”7 9” muodostetaan naapuristo ja tehdään valinta naapuristosta vastaavasti optimointifunktion arvon perusteella. Ratkaisun ”7 9” naapuristosta valitaan täten ratkaisu ”2 7 9”, joka on reppuongelman ratkaisu.



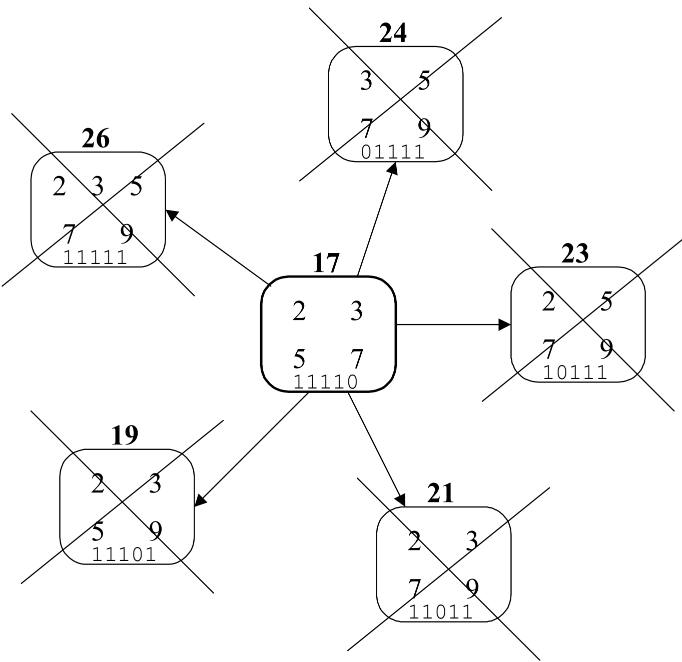
Kuva 3.4: Esimerkki paikallishausta reppuongelman tapauksessa.

Paikallishaku voidaan käsittää *matkaamisena* (*walk*) suunnatussa verkossa, jossa ratkaisut ovat solmuja ja kaaret yhdistävät naapuriratkaisuja (Aarts & Lenstra, 1997). Tällöin kulloisestakin solmusta siirrytään aina optimointifunktiolla mitattuna parhaaseen naapurisolmuun. Kuva 3.4 voidaan käsittää verkkona siten, että pyöristetyt laatikot eli ratkaisut ovat solmuja ja naapuruussuhteen kuvaavat nuolet ovat kaaria. Verkossa paikallishaun aikana kuljettu polku on kuvattu paksummilla nuolilla ja polun varrella olevat solmut on paksummilla reunoilla varustettuja laatikoita.

3.2. Paikallishaun edistyneemmät versiot

Paikallishaun perusversio on ahne, minkä seurauksena haku juuttuu ensimmäiseen paikallisesti optimaaliseen ratkaisuun. Määrittelemme, että ratkaisu on *paikallisesti optimaalinen*, jos se on kaikkia naapuriratkaisujaan parempi optimointifunktion arvolla mitattuna. Vastaavasti määrittelemme, että ratkaisu on *globaalisti optimaalinen*, jos se on kaikkia mahdollisia ratkaisuja parempi optimointifunktion arvolla mitattuna. Aartsin ja Lenstran (1997) mukaan paikallishaku löytää aina paikallisesti optimaalisen ratkaisun, mutta se ei välttämättä ole globaalisti optimaalinen ratkaisu. Nimittäin

käytettäessä hakustrategiana paras jälkeläinen tai ensimmäinen parannus –menetelmää paikallishaku päättyy, kun yksikään uusista ratkaisuista ei ole nykyistä ratkaisua parempi. Saatu ratkaisu on tällöin ainoastaan naapuriratkaisujaan parempi, mutta ei ole mitään takeita siitä, onko saatu ratkaisu paras mahdollinen vai ei.



Kuva 3.5: Paikallishaun juuttuminen paikalliseen maksimiin reppuongelman tapauksessa

Kuvassa 3.5 on esitetty haun juuttuminen paikallisesti optimaaliseen ratkaisuun aiemmin esitetyn reppuongelman tapauksessa. Lähtöratkaisuna kuvassa 3.5 on ”2 3 5 7”, joka on optimointifunktion arvolla mitattuna lähellä optimaalista. Valitsemaamme naapuriston luontimenetelmää käyttäen emme kuitenkaan saa luotua yhtään kelvollista naapuriratkaisua, joten ratkaisu ”2 3 5 7” on tässä tapauksessa paikallisesti optimaalinen ratkaisu.

Kuvien 3.4 ja 3.5 pohjalta havaitsemme, että paikallishaku on herkkä lähtöratkaisulle. Kuvan 3.4 esimerkissä lähtöratkaisulla ”3 7” paikallishaku löytää muutamalla iteraatiolla globaalisti optimaalisen ratkaisun, mutta kuvan 3.5 esimerkissä lähtöratkaisulla ”2 3 5 7” paikallishaku juuttuu heti alkuunsa paikallisesti optimaaliseen ratkaisuun. Ratkaisuna paikallishaun herkkyydellä lähtöratkaisulle voitaisiin tietenkin suorittaa haku useista lähtöratkaisuista lähtien, mutta tästä lähestymistavasta ei ole saatu erityisen hyviä tuloksia (Aarts & Van Laarhoven, 1985; Johnsson, 1990; Aarts & al., 1994). Aarts ja Lenstra (1997) ehdottavat erääksi ratkaisuksi lähtöratkaisun herkkyydelle *monitasomenetelmää* (*multilevel*), jossa käytetään useita eri naapuristoja. Tämä tarkoittaa esimerkiksi sitä, että tekemällä ratkaisuihin suurehkoja muutoksia muodostamme ensin karkean tason naapuriratkaisut, joihin tekemällä pienehköjä muutoksia muodostamme hienomman tason naapuriratkaisut (Gupta & al., 2000). Monitasomenetelmää on sovellettu hyvällä menestyksellä muun muassa *kauppamatkustajan ongelman* ratkaisemisessa (Johnson, 1990; Martin & al., 1992).

Herkkyys lähtöratkaisulle on seurausta paikallishaun taipumuksesta juuttua paikallisesti optimaaliseen ratkaisuun. Esimerkiksi reppuongelmalle esittämämme paikallishaku etsii ahneesti aina vain parannusta. Tällöin voi ymmärrettävästi käydä siten, että optimointifunktiolla mitattuna lähellä globaalisti optimaalista ratkaisua olevalle ratkaisulle ei voida luoda kilpailukykyistä naapuria, jolloin haku juuttuu tähän paikalliseen optimiin. Ratkaisuna paikallisesti optimaaliseen ratkaisuun juuttu-

miseen voimme sallia ei-optimaaliset siirtymiset tai lisätä naapuriston luontiin hieman satunnaisuutta (Aarts & Lenstra, 1997).

Mikäli naapuriston luonnissa on mukana satunnaisuutta, on tällöin kyse satunnaistetusta paikallishausta. Lisättäessä naapuriston luontiin satunnaisuutta on muutosten oltava toisaalta riittävän suuria, jotta on mahdollista päästä paikallisesti optimaalisen ratkaisun yli, mutta toisaalta ne eivät saa olla liian suuria, jotta ei tuhottaisi alkuperäistä ratkaisua (Fränti & Kivijärvi, 2000). Satunnaistettua paikallishakua on käytetty hyvällä menestyksellä muun muassa *toteutuvuusongelman* (*satisfiability problem*) ratkaisemisessa (Selman & al., 1992).

Aartsin ja Lenstran (1997) mukaan ei-optimaalisten siirtymien sallivia menetelmiä on muun muassa simuloitu jäähdytys, tabuhaku sekä geneettiset algoritmit. Carson (2001) mainitsee lisäksi *metropolis* (Metropolis & al., 1953) sekä *go-with-the-winners* (Dimitriou & Impagliazzo, 1996) – menetelmät, joita emme kuitenkaan käsittele jatkossa.

Simuloidussa jäähdytyksessä (*simulated annealing*) matkitaan kappaleen fysikaalista jäähtymistä ja hakeutumista *matalaenergiseen tilaan* (*low-energy state*; Kirkpatrick & al., 1983). Tämä onnistuu siten, että kullakin iteraatiolla valitaan naapuristosta satunnainen ratkaisu. Valittu ratkaisu hyväksytään, jos se on nykyistä parempi. Kuitenkin tietyllä todennäköisyydellä, joka on suhteutettu virheen kasvuun, voidaan myös nykyistä ratkaisua huonompi ratkaisu hyväksyä. Tämä todennäköisyys, jota kutsutaan jäähdytystermein lämpötilaksi, laskee hiljalleen ajan kuluessa. Simuloidussa jäähdytyksessä suuren lämpötilan ansiosta aluksi päästään helposti paikallisten optimien yli. Ajan kuluessa ja lämpötilan laskiessa muuttuu haku yhä enemmän normaaliksi paikallishauksi.

Tabuhaussa (*tabu search*) sallitaan ei-optimaaliset siirtymät ja käytetään hyödyksi tietoa aiemmin vierailluista ratkaisuista (Glover, 1989). Pitämällä yllä *tabulistaa* aiemmin vierailluista ratkaisuista estetään hakua palaamasta aiempiin ratkaisuihin ja pyritään ohjaamaan hakua kohti uusia, aiemmin käsittelemättömiä ratkaisuja. Tabuhaku voi edetä esimerkiksi siten, että kullakin iteraatiolla lisätään nykyinen ratkaisu tabulistaan ja valitaan naapuristosta paras ratkaisu, joka ei vielä kuulu tabulistaan. Koska tabuhaussa ratkaisu useinkin hetkellisesti heikkenee, pidetään haun aikana yllä parasta löydettyä ratkaisua, joka palautetaan lopuksi.

Geneettiset algoritmit (*genetic algorithms*) perustuvat luonnonvalintaan, jossa vain parhaat yksilöt selviävät (Holland, 1975). Geneettisissä algoritmeissa pyritään genetiikkaan ja evoluutioteoriaan pohjautuen optimoimaan populaation *kuntoisuus* (*fitness*) käyttäen geneettisiä operaatioita, joita ovat *risteytys* (*crossover*) ja *mutaatio* (*mutation*). *Geneettisessä paikallishaussa* (Mühlenbein & al., 1988) käsitellään, erona tavallisen paikallishaun yhteen ratkaisuun, joukkoa mahdollisia ratkaisuja eli ratkaisupopulaatiota. Geneettisessä paikallishaussa luodaan aluksi joukko satunnaisia lähtöratkaisuja, jotka muodostavat ratkaisupopulaation. Haun kullakin iteraatiolla luodaan ratkaisupopulaation pohjalta uusia ratkaisuja aiemmin mainittujen geneettisten operaatioiden avulla ja nämä ratkaisut lisätään osaksi populaatiota. Risteytystä käytettäessä uusi ratkaisu muodostetaan yhdistämällä kaksi olemassa olevaa ratkaisua. Mutaation avulla puolestaan tehdään pieniä muutoksia syntyneisiin uusiin ratkaisuihin. Ratkaisupopulaation koko kuitenkin pidetään vakiona, joten ainoastaan parhaimmat ratkaisut selviävät seuraavalle iteraatiolle. Haun päätyttyä palautetaan lopullisen ratkaisupopulaation paras ratkaisu.

3.3. Paikallishaku klusterointimenetelmänä

Seuraavaksi kerrotaan yleisesti paikallishaun käytöstä klusterointimenetelmänä. Määrittelemme klusteroinnin kombinatorisena optimointiongelmana kiinnittämällä kaavassa 2.4 esitetyn keskineliövirheen minimoitavaksi optimointifunktioksi. Paikallishaku soveltuu klusterointimenetelmäksi, sillä esimerkiksi Al-Sultan ja Khan (1996) ovat tutkineet geneettiseen paikallishakuun, tabuhakuun sekä simuloituun jäähdytykseen perustuvia klusterointimenetelmiä ja todenneet kaikkien näiden päihittävän ainakin K-meansin ratkaisun laadussa. Koska tahdomme välttää paikallishaulle tyypillisen paikalliseen optimiin juuttumisen, tarkastelemme seuraavassa ainoastaan paikallishaun edistyneempiä versioita geneettistä paikallishakua, tabuhakua sekä satunnaistettua paikallishakua.

Esimerkkinä geneettisestä paikallishausta klusterointimenetelmänä käynee Fränti ja muiden (1997b) esittämä menetelmä. Heidän menetelmä saavuttaa parhaat tulokset käyttämällä yhdistävää klusterointimenetelmää ratkaisujen risteytykseen. Mutaatioita menetelmä luo korvaamalla toisinaan satunnaisen sentroidin satunnaisella alkiolla. Pääasialliset heikkoudet heidän menetelmässään ovat hidas suoritusaika sekä runsas määrä parametreja. Vaikkakin parametreja säätämällä voidaan löytää hyviä ratkaisuja kohtuullisessa ajassa (Fränti, 2000), ovat parametrit riippuvaisia klusteroitavasta aineistosta ja parametrit on kokeellisesti määritettävä kullekin aineistolle erikseen, mikäli suoritusaika halutaan pitää siedettävänä. Parametrien säätämisestä päästään eroon käyttämällä Kivijärven ja muiden (2003) esittämää menetelmän adaptiivista versiota. Virmajoen (2004) ja Kivijärven (2004) mukaan parhaat klusterointitulokset ratkaisun laadusta puhuttaessa on saavutettu juuri geneettistä paikallishakua käyttäen. Geneettisestä paikallishausta löytyy useita muitakin toteutuksia muun muassa Delportin ja Koschorreckin (1995), Murthyan ja Chowdhurybyn (1996) sekä Tsengin ja Yangin (2000) esittämät versiot.

Tabuhaun käyttämisen klusterointimenetelmänä esitti ensimmäisen kerran Al-Sultan (1995). Tämän jälkeen ovat muun muassa Fränti ja muut (1998) sekä Sung ja Jin (1999) tutkineet tabuhaun mahdollisuuksia klusterointimenetelmänä. Fränti ja muiden (1998) menetelmä muodostaa ratkaisun naapuriston tekemällä satunnaisia muutoksia alkuperäiseen ratkaisuun. Tabulistasta vaikuttaisi olevan melko vähän hyötyä, vaikka hylkäisimme täsmälleen samojen ratkaisujen lisäksi myös tabulistassa olevien ratkaisujen kanssa samankaltaiset ratkaisut. Fränti ja muut (1998) raportoivat menetelmänsä tuottavan hyviä ratkaisuja, mutta olevan melkoisen hidas.

Koska tabuhaku tuottaa hyviä ratkaisuja, vaikka tabulistasta ei vaikuttaisi olevan juurikaan hyötyä, lienee melko luonnollista selvittää, kuinka haku pärjää ilman tabulistaa. Päädymme täten satunnaistettuun paikallishakuun, jossa kunkin ratkaisun naapuristo luodaan tekemällä satunnaisia muutoksia nykyiseen ratkaisuun ja jossa haku etenee kuitenkin aina ensimmäisen parannuksen suuntaan. Fränti ja Kivijärvi (2000) raportoivat esittämänsä satunnaistettua paikallishakua käyttävän klusterointimenetelmän olevan helppo toteuttaa ja saavuttavan hyviä tuloksia kohtuullisessa ajassa. Menetelmä ei ole herkkä lähtöratkaisulle, joten tulosten hajonta on pientä (Kivijärvi, 2004). Lisäksi menetelmän saavuttamat tulokset ovat lähes yhtä hyviä kuin geneettisellä paikallishaulla saavutetut (Virmajoki, 2004).

Fräntin ja Kivijärven (2004) esittämän menetelmän yksinkertainen rakenne, tulosten hyvä laatu sekä nopeus verrattuna esimerkiksi geneettiseen paikallishakuun puoltavat menetelmän soveltamista käytännön klusterointitehtäviin. Menetelmää on käytetty muun muassa bakteerien luokitteluun (Fränti & al., 2000), karttakuvien *segmentointiin* (Kopylov & Fränti, 2004; Rus & Astola, 2005) sekä klusterointimenetelmänä *värekerrointen* (*wavelet coefficent*) mallinnuksessa (Lehtinen & Kivijärvi, 2001). Fräntin ja muiden (2000, 2003) tulokset osoittavat menetelmän toimivan hyvin, vaikka optimointifunktiona käytetään muuta kuin keskineliövirhettä.

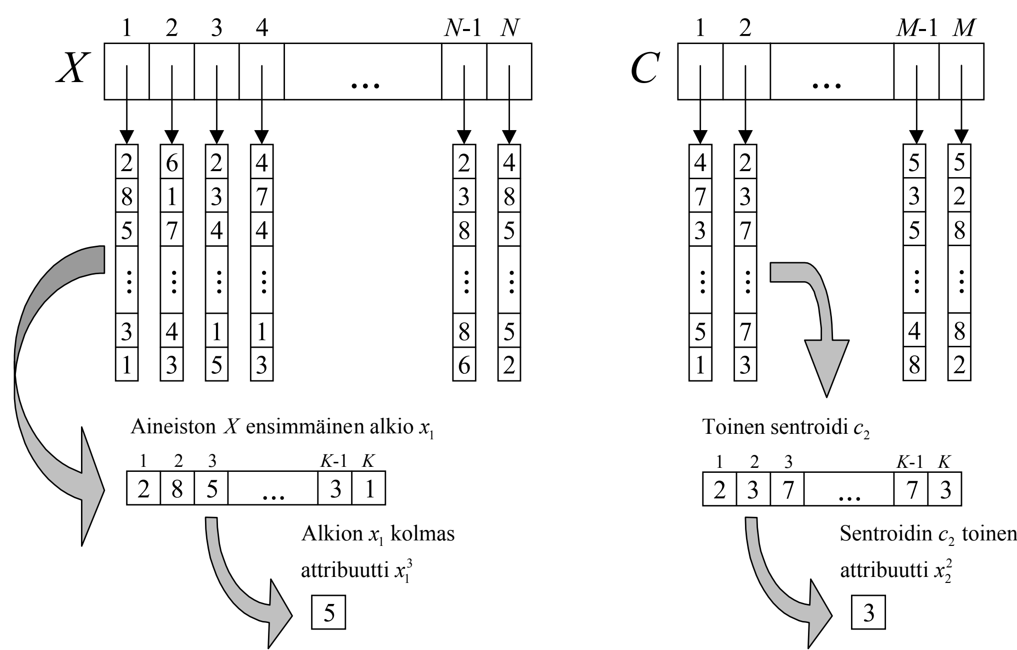
3.4. Satunnaistettu paikallishaku klusterointimenetelmänä

Tässä kohdassa tullaan kuvaamaan Fräntin ja Kivijärven (2000) esittämän satunnaistettuun paikallishakuun perustuvan klusterointimenetelmän *RLS-2* tarvitsemat operaatiot yksityiskohtaisesti pseudokoodin tasolla. Keskitymme vain Fräntin ja Kivijärven (2000) esittämään *RLS-2* menetelmään, sillä sen tuottamien tulosten on raportoitu olevan parempia ja tulosten hajonnan pienempää verrattuna *RLS-1* menetelmään (Fränti & Kivijärvi, 2000).

3.4.1. Tietorakenteet

Kuten luvussa 2 määriteltiin klusteroitava aineisto koostuu N alkiosta, jotka ovat K-ulotteisia vektoreita, ja alkioiden välisenä etäisyysmittana käytetään euklidista etäisyyttä. Edelleen luvun 2 mukaisesti klustereiden lukumäärä on M ja aineiston klusteroinnin tulos voidaan esittää joko klustereiden sentroidien C tai alkioiden osituksen P avulla. Klustereiden sentroidien joukkoa kutsutaan usein *koodikirjaksi* (*codebook*) ja klustereiden sentroideja *koodivektoreiksi* (*code vectors*). Nämä nimitykset tulevat *vektorikvantisoinnissa* (*vector quantization*; Gersho & Gray, 1992) käytettävästä termistöstä. Seuraavaksi esitellään käytettävät tietorakenteet.

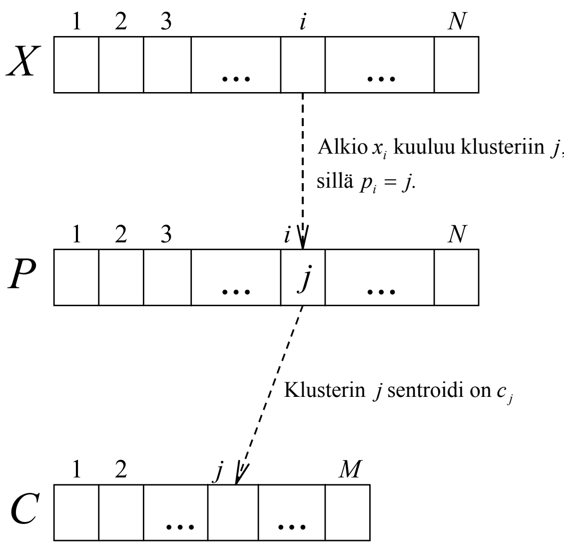
Kuvassa 3.6 on esitetty aineisto X sekä klustereiden sentroidien joukko eli koodikirja C. Aineisto ja koodikirja koostuvat alkioista, jotka ovat K-ulotteisia vektoreita. Sekä aineisto että koodikirja voidaan täten käsittää taulukkona, jonka alkioina ovat vektorit. Jokaista aineiston ja koodikirjan alkiota eli vektoria voidaan käsitellä jälleen omana taulukkonaan. Käyttämässämme notaatiossa aineiston X ensimmäinen alkio on siis x_1 ja tämän kolmas attribuutti on x_1^3. Koodikirjan alkioihin viitataan vastaavasti ja esimerkiksi koodikirjan kolmas alkio c_3 on siis kolmannen klusterin sentroidi.



Kuva 3.6: Aineiston alkiot ja klusteiden sentroidit.

Kuvassa 3.7 on esitetty aineiston X ositus P. Ositus määrittelee kullekin alkiolle, mihin klusteriin kyseinen alkio kuuluu. Ositus voidaan täten käsittää aineiston alkioiden mukaan järjestettynä taulukkona, jonka alkioina ovat alkioiden klustereiden indeksit. Alkio x_i kuuluu siis klusteriin j, jos

osituksen P i:s alkio p_i on j. Sama pätee myös kääntäen, eli jos osituksen P i:s alkio p_i on j, kuuluu alkio x_i tällöin klusteriin j.



Kuva 3.7: Aineiston ositus.

3.4.2. Perusoperaatiot

Kuvassa 3.8 on esitetty pseudokoodina alkioiden x_i ja x_j välisen euklidisen etäisyyden laskeminen kaavan 2.4 mukaan. Etäisyyden laskemisen aikavaativuus on O(K), sillä etäisyyttä laskettaessa täytyy käydä läpi kaikki annettujen alkioiden attribuutit. Kuvan 8 pseudokoodissa on etäisyyden laskevan operaation nimeksi annettu *Dist* myöhempää viittaamista silmällä pitäen. *Dist*-operaatio saa *arvoparametreina* vektorit x_i ja x_j sekä palauttaa saamiensa vektoreiden välisen etäisyyden. Vakioiden N, M ja K oletetaan olevan globaaleja ja täten operaation käytössä. Myöhemmin tässä tutkielmassa esitettävät pseudokoodit noudattavat samoja edellä mainittuja käytäntöjä ellei toisin mainita.

```
Dist(xi,xj) {
     sum := 0;

     FOR k := 1 TO K DO {
          sum := sum + (xi^k - xj^k)^2;
     }

     RETURN SQRT(sum);
}
```

Kuva 3.8: Alkioiden välinen euklidinen etäisyys.

Kuvassa 3.9 on esitetty pseudokoodina optimointifunktiona käytettävän keskineliövirheen laskeminen. Keskineliövirhe lasketaan ottamalla kunkin alkion etäisyys klusterinsa sentroidiin toiseen potenssiin ja jakamalla tämä alkioiden ja attribuuttien määrän tulolla. Optimointifunktion laskemisen aikavaativuus on O(NK), sillä kullekin alkiolle lasketaan etäisyys sentroidiinsa.

```
ObjectiveFunction(P,C,X) {
    sum := 0;

    FOR i := 1 TO N DO {
        j   := pi;
        sum := sum + Dist(xi,cj)^2;
    }

    RETURN sum / (N*K);
}
```

Kuva 3.9: Optimointifunktion arvon laskeminen.

Luvussa 2 todettiin, että minimoitaessa keskineliövirhettä klusterien sentroidit ja alkioiden ositus riippuvat toisistaan siten, että kun toinen annettu, voidaan puuttuva muodostaa optimaalisesti. Pseudokoodi optimaalisten klustereiden sentroidien muodostamiseksi osituksen pohjalta on esitetty kuvassa 3.10. Optimaalinen sentroidi on kaavan 2.5 sentroidiehdon mukaisesti klusterin alkioiden keskiarvovektori. Optimaalisten sentroidien laskemisen aikavaativuus on $O(NK+MK)$, sillä täytyy laskea N kertaa kahden K-ulotteisen vektorin alkioittainen summa ja M kertaa jakaa kukin K-ulotteisen vektorin attribuutti tulolla NK. Aikavaativuus on täten $O(NK)$, koska pätee $M \ll N$.

```
OptimalRepresentatives(P,X) {
    /* Taulukon Sum[1..M] alkioina on vektoreita ja
       taulukon Count[1..M] alkioina on kokonaislukuja.
       Molemmat taulukot on alustettu nollaksi. */

    FOR i := 1 TO N DO {
        j      := pi;
        Sumj   := Sumj + xi;  /* vektoreiden alkioittainen summa */
        Countj := Countj + 1;
    }

    FOR i := 1 TO M DO {
        IF Counti <> 0 THEN {
            ci := Sumi / Counti;
        }
    }

    RETURN C;
}
```

Kuva 3.10: Optimaalisten sentroidien muodostaminen, kun ositus on annettu.

Pseudokoodi optimaalisen osituksen muodostamiseksi sentroidien pohjalta on esitetty kuvassa 3.11. Optimaalinen ositus saadaan muodostettua kaavan 2.6 lähin naapuri –ehdon mukaisesti sijoittamalla kukin alkio siihen klusteriin, jonka sentroidi on lähinnä kyseistä alkiota. *FindNearestRepresentative*-operaatio etsii annettua alkiota lähinnä olevan sentroidin. Operaation aikavaativuus on täten $O(MK)$, sillä sen on laskettava annetun alkion etäisyys kaikkiin sentroidei-

hin. Optimaalisen osituksen aikavaativuus on näin ollen O(*NMK*), sillä se kutsuu *FindNearestRepresentative*-operaatiota jokaiselle aineiston alkiolle.

```
FindNearestRepresentative(C,x) {
    nrst := 1;
    dst  := Dist(x,c_nrst);

    FOR i := 2 TO M DO {
        d := Dist(x,c_i);

        IF d < dst THEN {
            nrst := i;
            dst  := d;
        }
    }

    RETURN nrst;
}


OptimalPartition(C,X) {
    FOR i := 1 TO N DO {
        p_i := FindNearestRepresentative(C,x_i);
    }

    RETURN P;
}
```

Kuva 3.11: Optimaalisen osituksen muodostaminen, kun sentroidit on annettu.

Mikäli tiedetään ainoastaan yhden sentroidin muuttuneen, voidaan ositus palauttaa optimaaliseksi vähemmällä työllä käyttämällä *paikallista uudelleenositusta* (*local repartition*; Fränti ja Kivijärvi, 2000), jonka vaiheet ovat alkioiden hylkäys sekä alkioiden houkuttelu.

Alkioiden hylkäyksessä (*object rejection*) sijoitamme kunkin alkion muuttuneesta klusterista alkiota lähinnä olevaan klusteriin. Menetelmää kutsutaan alkioiden hylkäykseksi, sillä sentroidin muututtua osa klusterin alkioista todennäköisesti siirtyy johonkin toiseen klusteriin, eli muuttunut sentroidi hylkää siihen sopimattomat alkiot. Kaavassa 3.1 on esitetty alkioiden hylkäyksen periaate. Kukin muuttuneen klusterin *j* alkio sijoitetaan siihen klusteriin, jonka sentroidi on kyseistä alkiota lähinnä. Mikäli alkiot ovat jakautuneet tasaisesti klustereihin, on muuttuneessa klusterissa *N*/*M* alkiota ja kutakin alkiota kohden lasketaan etäisyys jokaiseen sentroidiin. Näin ollen alkioiden hylkäyksen aikavaativuus on keskimäärin O((*N*/*M*)·*MK*) = O(*NK*).

$$p_i = \min_{1 \le k \le M} d(x_i, c_k)^2 \quad \forall\, i \in [1, N] \mid p_i = j \qquad \textbf{(3.1)}$$

Alkioiden houkuttelussa (*object attraction*) sijoitamme muuttuneeseen klusteriin ne aineiston alkiot, jotka ovat lähempänä muuttuneen klusterin sentroidia kuin nykyisen klusterinsa sentroidia. Menetelmää kutsutaan alkioiden houkutteluksi, koska muuttunut klusteri voi houkutella alkioita naapuriklustereistaan. Kaavassa 3.2 on esitetty alkioiden houkuttelun periaate. Jokaiselle alkiolle lasketaan etäisyys sekä muuttuneeseen klusteriin *j* että alkion nykyiseen klusteriin ja alkio sijoitetaan näistä

lähempään. Jokaista aineiston alkiota kohden lasketaan siis etäisyys kahteen sentroidiin, joten alkioiden houkuttelun aikavaativuus on O(2·*NK*) = O(*NK*).

$$p_i = \min_{k=j \vee k=p_i} d(x_i, c_k)^2 \quad \forall\, i \in [1, N] \tag{3.2}$$

Paikallinen uudelleenositus on esitetty pseudokoodina kuvassa 3.12. Paikallinen uudelleenositus saa parametrinaan muuttuneen klusterin *j* ja tämän jälkeen tekee parametrina annetulle klusterille alkioiden hylkäyksen ja houkuttelun. Edellä esitettyjen alkioiden hylkäyksen ja houkuttelun aikavaativuusanalyysin nojalla koko *LocalRepartition*-operaation aikavaativuus on O(*NK*), mikä on vähemmän verrattuna *OptimalPartition*-operaation jokaisen alkion ja sentroidin välisen etäisyyden laskemiseksi vaadittavaan O(*NMK*) nähden.

```
LocalRepartition(P,C,X,j) {
    /* alkioiden hylkäys */
    FOR i := 1 TO N DO {
        IF pi = j THEN {
            pi := FindNearestRepresentative(C,xi);
        }
    }

    /* alkioiden houkuttelu */
    FOR i := 1 TO N DO {
        k = pi;
        IF Dist(xi,cj) < Dist(xi,ck) THEN {
            pi := j;
        }
    }

    RETURN P;
}
```

Kuva 3.12: Paikallinen uudelleenositus.

Klusteroinnin tuloksen esittämiseen riittää siis joko klustereiden sentroidien tai alkioiden osituksen antaminen. Ratkaisun esittämiseksi on siten kolme vaihtoehtoa: voimme joko käyttää ositusta, sentroideja tai molempia yhdessä. Fränti ja Kivijärvi (2000) ovat päätyneet esittämään ratkaisun sekä osituksen että sentroidien avulla, sillä molempia tarvitaan optimointifunktiona käytettävän keskineliövirheen laskemiseksi, joten mekin pitäydymme tässä esitysmuodossa.

3.4.3. Lähtöratkaisun valinta

Paikallishaun lähtöratkaisu valitaan tyypillisesti satunnaisesti (Kivijärvi, 2004). Seuraavaksi tarkastellaan mahdollisuuksia satunnaisen lähtöratkaisun muodostamiseen. Satunnaisen lähtöratkaisun käyttö on tässä tapauksessa soveliasta, koska esiteltävä menetelmä ei ole herkkä lähtöratkaisun valinnalle (Fränti & Kivijärvi, 2000; Kivijärvi, 2004). Eräs menetelmä muodostaa satunnainen lähtöratkaisu on sijoittaa jokainen alkio satunnaisesti valittuun klusteriin (Kivijärvi, 2004). Tällöin kuitenkin samassa klusterissa olevilla alkioilla ei yleisesti ottaen ole mitään yhteistä, sillä kukin klusteri voi sisältää alkioita joka puolelta aineistoa. Näin muodostetussa ratkaisussa klustereiden sentroidit pyrkivät sen vuoksi keskittymään koko aineiston sentroidin läheisyyteen, vaikka yleensä hyvässä ratkaisussa klusterit ovat aineiston selkeästi erillisiä osia. Näin ollen hyvän ratkaisun saavut-

tamiseksi tällä tavoin muodostetusta lähtöratkaisusta lähtien tulisi vaatimaan mittavia muutoksia (Kivijärvi, 2004).

Toinen menetelmä muodostaa lähtöratkaisu on valita klustereiden sentroidit satunnaisesti. Sentroidit voitaisiin valita aineiston koko arvoalueelta, mutta tämä lähestymistapa ei ota huomioon aineiston alkioiden jakautumaa (Kivijärvi, 2004). Tällöin voi käydä niin, että sentroidi sijoitetaan paikkaan, jossa ei ole alkioita lähimaillakaan. Näin ollen on parempi valita aineistosta satunnaiset alkiot sentroideiksi, koska tämä menetelmä jakaa sentroidit tasaisesti ympäri aineistoa ja täten tuottaa paremman lähtöratkaisun (Kivijärvi, 2004). Tulemmekin muodostamaan paikallishaun lähtöratkaisu valitsemalla aineiston satunnaiset alkiot sentroideiksi. Lähtöratkaisun muodostava *SelectRandomRepresentatives*-operaatio on esitetty pseudokoodina kuvassa 3.13. Lähtöratkaisun muodostamisen aikavaativuus on O(*MK*), sillä jokaista sentroidia kohden on sijoitettava yksi aineiston alkio sentroidiksi.

```
SelectRandomRepresentatives(X) {
    k := N / M;  /* kokonaislukujen jakolasku */

    FOR i := 1 TO M DO {
        /* Väli [1,N) jaetaan M osaan ja valitaan
           satunnainen luku jokaisesta osasta. */
        IF i = M THEN {
            j := RANDOM(i*k, N);  /* Satunnaisluku väliltä [i*k,N]
        } ELSE {
            j := RANDOM(i*k, (i+1)*k);
        }

        c_i := x_j;
    }

    RETURN C;
}
```

Kuva 3.13: Lähtöratkaisun muodostaminen.

3.4.4. Naapuristofunktio

Naapuristoa muodostettaessa nykyiseen ratkaisuun täytyy tehdä sekä globaaleja että paikallisia muutoksia. Globaaleja muutoksia tulee tehdä, jotta olisi mahdollista päästä paikallisen minimin yli, ja paikallisia muutoksia tulee tehdä, jotta muodostettu naapuriratkaisu olisi kilpailukykyinen nykyisen ratkaisun kanssa (Fränti & Kivijärvi, 2000). Strategiana on siis muuttaa ensin ratkaisua globaalisti, jolloin ratkaisu ei välttämättä ole enää paikallisesti optimaalinen, ja tämän jälkeen paikallisilla muutoksilla palauttaa ratkaisua lähemmäksi paikallista minimiä.

Eräs vaihtoehto ratkaisun muuttamiseksi olisi satunnaisen alkion siirtäminen nykyisestä klusteristaan satunnaisesti valittuun klusteriin (Kivijärvi, 2004). Tässä kuitenkin törmätään jälleen samaan ongelma kuin lähtöratkaisun muodostamisen parissa, eli satunnaiseen klusteriin siirretyllä alkiolla ei mitä todennäköisimmin ole mitään yhteistä sen klusterin kanssa, johon se siirrettiin. Edistyneempi versio tästä menetelmästä sallisi vain järkevät siirrot. Esimerkiksi sallittaisiin ainoastaan klusterinsa sentroidista kaukana olevien alkioiden siirtäminen tai alkioiden siirto ainoastaan naapuriklustereihinsa (Kivijärvi, 2004). Nämäkin versiot tekevät kuitenkin ainoastaan paikallisia muutok-

sia, eivätkä sen vuoksi ole kovin varteenotettavia vaihtoehtoja naapuriston generointiin (Fränti & Kivijärvi, 2000).

Toinen vaihtoehto ratkaisun muuttamiseksi on korvata satunnaisen klusterin sentroidi satunnaisella alkiolla (Fränti & al., 1998). Sentroidin korvaus muuttaa ratkaisua globaalisti ja sen jälkeen ositus ei ole enää optimaalinen sentroideihin nähden. Käyttämällä aiemmin esiteltyä paikallista uudelleen-ositusta voidaan ositus kuitenkin palauttaa optimaaliseksi kohtuullisella vaivalla.

K-means (McQueen, 1967) vuorottelee optimaalisten sentroidien ja optimaalisen osituksen muodostavia *OptimalRepresentatives*- ja *OptimalPartition*-operaatioita ja onnistuu tällä tavalla löytämään käyttämämme optimointifunktion paikallisen minimin (Han & Kamber, 2001). Näin ollen K-means soveltuu ratkaisun paikalliseen optimointiin. Kuvassa 3.14 on esitetty K-meansin pseudokoodi, jossa I on iteraatioiden lukumäärä ja parametrina saadun osituksen oletetaan olevan sentroideihin nähden optimaalinen. K-meansin aikavaativuutta dominoi *OptimalPartition*-operaatio, joten K-meansin aikavaativuus on $O(INMK)$.

```
KMeans(P,C,X) {
    FOR i := 1 TO I DO {
        C := OptimalRepresentatives(P,X);
        P := OptimalPartition(C,X);
    }

    RETURN (P,C);
}
```

Kuva 3.14: K-means –algoritmi.

Fräntin ja Kivijärven (2000) mukaisesti nykyisen ratkaisun naapuriratkaisu muodostetaan korvaamalla satunnaisesti valittu sentroidi satunnaisesti valitulla alkiolla. Näin saadun ratkaisun ositus palautetaan optimaaliseksi käyttämällä paikallista uudelleenositusta. Kuvassa 3.15 on esitetty pseudokoodina sentroidin korvaus. *RandomSwap*-operaatio saa parametrinaan sentroidit ja aineiston, korvaa satunnaisesti valitun sentroidin j satunnaisesti valitulla aineiston alkiolla ja palauttaa muutetun koodikirjan ja muutetun sentroidin indeksin j. *SelectRandomDataObject*-operaatio valitsee aineistosta satunnaisen alkion, jota ei ole vielä koodikirjassa. Operaation aikavaativuus on keskimäärin $O(MK)$, sillä satunnaisesti valitulle aineiston alkiolle tarkistetaan, ettei sitä jo löydy koodikirjasta. *SelectRandomDataObject*-operaation on toki teoriassa mahdollista jäädä arpomaan alkioita pidemmäksi aikaa, mikäli satunnaislukugeneraattori arpoo aina sellaisen alkion, joka on jo koodikirjassa. Tämä ei kuitenkaan käytännössä ole ongelma, sillä alkioiden suurempi määrä verrattuna sentroidien määrään pitää huolen, että todennäköisyys jäädä jumiin pidemmäksi aikaa on erittäin pieni. *RandomSwap*-operaation aikavaativuutta dominoi *SelectRandomDataObject*-operaatio, joten sen aikavaativuus on $O(MK)$.

```
SelectRandomDataObject(C,X) {
    REPEAT {
        i := RANDOM(1,N);
        ok := True

        /* Duplikaatteja ei sallita. */
        FOR j := 1 TO M DO {
            IF c_j = x_i THEN {
                ok := False;
            }
        }

    } UNTIL ok = True;

    RETURN x_i;
}


RandomSwap(C,X) {
    j := RANDOM(1,M);  /* Satunnaisluku väliltä [1,M] */
    c_j := SelectRandomDataObject(C,X);

    RETURN (C,j);
}
```

Kuva 3.15: Sentroidin korvaus.

Koska sentroidin korvaus ja uudelleenositus muuttavat ratkaisua globaalisti, tarvitaan niiden jälkeen ratkaisun paikallista parantelua. Fräntin ja Kivijärven (2000) mukaisesti ratkaisua parannetaan paikallisesti tekemällä kaksi K-means –iteraatiota. Iteraatioita ei tehdä enempää, sillä kahden iteraation jälkeen ei yleensä saavuteta enää merkittävää parannusta (Fränti & al., 1997b; Fränti & al., 1998).

Esitetty naapuristofunktio voisi pahimmassa tapauksessa luoda vain satunnaisia lähtöratkaisuja K-meansin parannettavaksi. Tällöin sama voitaisiin tehdä paljon yksinkertaisemmin käyttämällä toistettua K-meansia. *Toistettu K-means* (*repeated K-means*; Tuononen, 2004) suorittaa K-meansia toistuvasti satunnaisista lähtöratkaisuista lähtien ja valitsee lopuksi saaduista ratkaisuista parhaan.

Kuvassa 3.16 on esitetty toistettu K-means pseudokoodina. Kullakin iteraatiolla muodostetaan ratkaisu valitsemalla satunnaiset alkiot sentroideiksi (*SelectRandomRepresentatives*), minkä jälkeen ajetaan K-meansia (OptimalPartition- ja *KMeans*-operaatiot) kunnes sentroidit eivät enää muutu. Näin saatua uutta ratkaisua verrataan nykyiseen ja uusi ratkaisu korvaa sen, mikäli uusi ratkaisu on optimointifunktiolla (*ObjectiveFunction*) mitattuna parempi. Menetelmä palauttaa lopuksi T iteraation jälkeen parhaan siihen mennessä löydetyn ratkaisun. Menetelmän aikavaativuutta dominoi aiemmin esitettyjen operaatioiden aikavaativuusanalyysien nojalla *KMeans*-operaatio ja tarkemmin ottaen sen *OptimalPartition*-operaatio, joten menetelmän aikavaativuus on $O(TNMK)$.

```
RepeatedKMeans(X) {

    FOR i := 1 TO T DO {
        C'      := SelectRandomRepresentatives(X);
        P'      := OptimalPartition(C',X);
        (P',C') := KMeans(P',C',X);

        IF i == 1 THEN (P,C) := (P',C');

        IF ObjectiveFunction(P',C',X) < ObjectiveFunction(P,C,X) {
            (P,C) := (P',C');
        }
    }

    RETURN (P,C);
}
```

Kuva 3.16: Toistettu K-means.

4. Satunnaistettu paikallishaku

Hakustrategiana Fräntin ja Kivijärven (2000) esittämässä satunnaistetussa paikallishaussa käytetään ensimmäinen parannus –menetelmää, sillä Andersonin (1996) tulokset osoittavat sen soveltuvan paremmin käytännön ongelmien hakustrategiaksi. Ensimmäinen parannus –menetelmä nimittäin jää epätodennäköisemmin jumiin paikalliseen minimiin ja todennäköisesti löytää optimin nopeammin, koska sillä on joka iteraatiolla vähemmän ratkaisuja arvotettavanaan. Fränti ja Kivijärvi (2000) lisäksi raportoivat, että he eivät omissa testeissään havainneet suuremman naapuriston käytöstä merkittävää etua ja että suuremman naapuriston käyttö hidasti hakua.

Kuvassa 4.1 on esitetty Fräntin ja Kivijärven (2000) esittämän *RLS-2* menetelmän pseudokoodi. Aluksi luodaan koodikirja valitsemalla satunnaiset alkiot sentroideiksi (*SelectRandomRepresentatives)*, minkä jälkeen luodaan sentroideihin nähden optimaalinen ositus (*OptimalPartition)*. Näin muodostettu koodikirja ja ositus ovat menetelmän lähtöratkaisu. Tämän jälkeen suoritetaan *T* paikallishakuiteraatiota.

Kullakin iteraatiolla luodaan uusi ratkaisu korvaamalla nykyisen ratkaisun satunnaisesti valittu sentroidi satunnaisesti valitulla alkiolla (*RandomSwap*), suorittamalla paikallinen uudelleenositus (*LocalRepartition*) sekä kaksi K-means –iteraatiota (*Kmeans*). Näin saatua uutta ratkaisua verrataan nykyiseen ja uusi ratkaisu korvaa sen, mikäli se on optimointifunktiolla (*ObjectiveFunction*) mitattuna parempi. Menetelmä palauttaa lopuksi *T* paikallishakuiteraation jälkeen parhaan siihen mennessä löydetyn ratkaisun. Menetelmän aikavaativuutta dominoi aiemmin esitettyjen operaatioiden aikavaativuusanalyysien nojalla *KMeans*-operaatio ja tarkemmin ottaen K-meansin *OptimalPartition*-operaatio, joten menetelmän aikavaativuus on O(*TNMK*).

```
RandomizedLocalSearch(X) {

    C := SelectRandomRepresentatives(X);
    P := OptimalPartition(C,X);

    FOR i := 1 TO T DO {
        (C',j)  := RandomSwap(C,X);
        P'      := LocalRepartition(P,C',X,j);
        (P',C') := KMeans(P',C',X);

        IF ObjectiveFunction(P',C',X) < ObjectiveFunction(P,C,X) THEN {
            (P, C):= (P',C');
        }
    }

    RETURN (P,C);
}
```

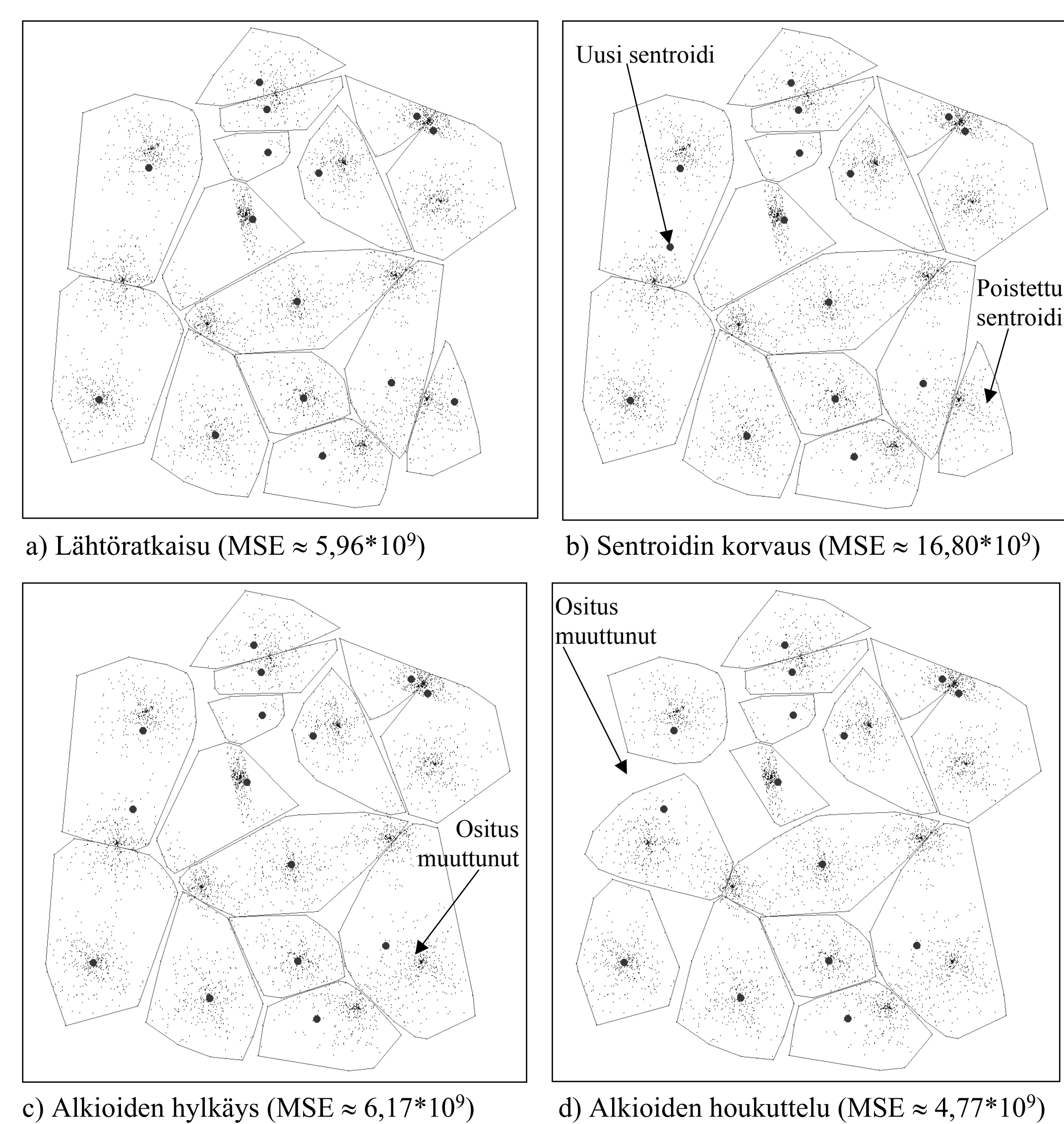
Kuva 4.1: Satunnaistettu paikallishaku.

Paikallishakuiteraatioiden määrä *T* on vapaasti valittavissa. Koska satunnaistettu paikallishaku ei juutu paikalliseen minimiin, pystyy se parantamaan ratkaisua sitä enemmän, mitä enemmän iteraatioita suoritetaan. Toisaalta ratkaisuun löydetään yhä harvemmin parannusta iteraatioiden määrän kasvaessa (Fränti & Kivijärvi, 2000). Iteraatioiden määrä vaikuttaa paikallishaun käyttämään aikaan, joten iteraatioiden määrän valinnassa täytyy käytännössä tehdä kompromissi käytettävän ajan

ja halutun ratkaisun laadun välillä. Nyrkkisääntönä voi pitää, että iteraatioiden määrän on oltava suunnilleen samaa suuruusluokkaa aineiston alkioiden määrän kanssa. Tällöin nimittäin tulee yritettyä uuden klusterin perustamista joka puolelle aineistoa.

4.1. Satunnaistetun paikallishaun havainnollistaminen

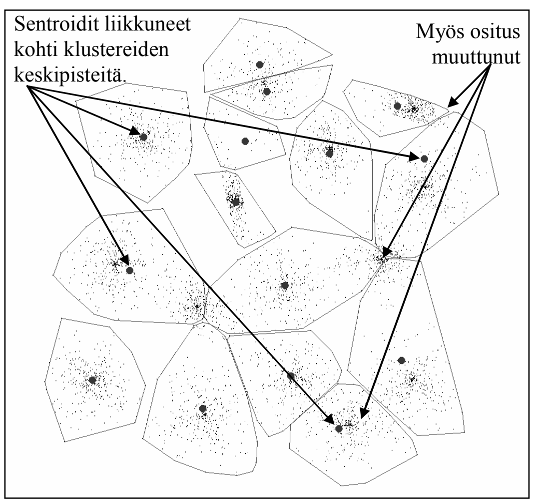
Klusteroitavana aineistona tässä kohdassa tullaan käyttämään *S2* aineistoa. Kuvassa 4.2a on esitetty paikallishaun luoma lähtöratkaisu ja tilanne sentroidin korvauksen jälkeen on esitetty kuvassa 4.2b. Kuvia vertaamalla havaitaan, että alaoikealla ollut sentroidi on poistettu ja uusi on lisätty aineiston vasemmalle puolelle. Huomattavaa on, että nyt on muutettu ainoastaan koodikirjan rakennetta, joten ositus ei ole optimaalinen koodikirjaan nähden. Nimittäin korvatun sentroidin alkiot alavasemmalla kuuluvat yhä korvatun sentroidin edustamaan klusteriin, vaikka sentroidi on aivan eri puolella aineistoa. Osituksen epäoptimaalisuus havaitaan myös esimerkiksi minimoitavana optimointifunktiona käytettävän keskineliövirheen arvon kasvusta, sillä keskineliövirhe on enemmän kuin kaksinkertainen sentroidin korvauksen jälkeen kuin mitä se oli sitä ennen.



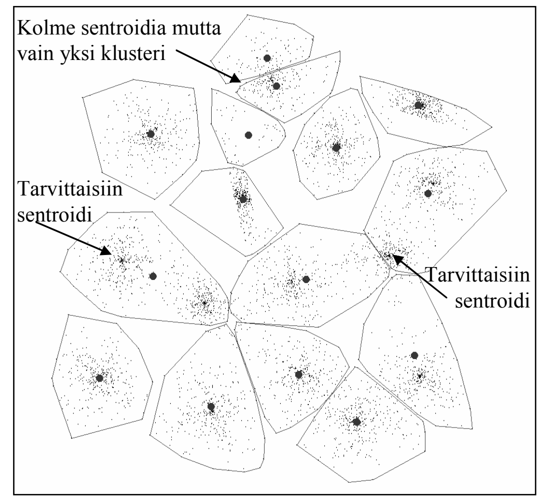
Kuva 4.2: Uuden ratkaisun muodostaminen lähtöratkaisua muokkaamalla.

Ratkaisu alkioiden hylkäyksen jälkeen on esitetty kuvassa 4.2c. Vertaamalla kuvia 4.2b ja 4.2c nähdään, että muuttuneeseen klusteriin kuuluneet alkiot alaoikealla on nyt sijoitettu lähimpiin klustereihinsa. Parannusta ratkaisun laadussa voidaan havaita tapahtuneen jo pelkän alkioiden hylkäyksen jälkeen, sillä keskineliövirhe on pienentynyt suunnilleen lähtöratkaisun tasolle. Ratkaisu alkioiden hylkäyksen ja houkuttelun eli paikallisen uudelleenosituksen jälkeen on esitetty kuvassa 4.2d. Vertaamalla kuvia 4.2c ja 4.2d nähdään, että muuttunut klusteri vasemmalla on nyt saanut omia alkioita. Myös ratkaisun laatu keskineliövirheellä mitattuna on parantunut.

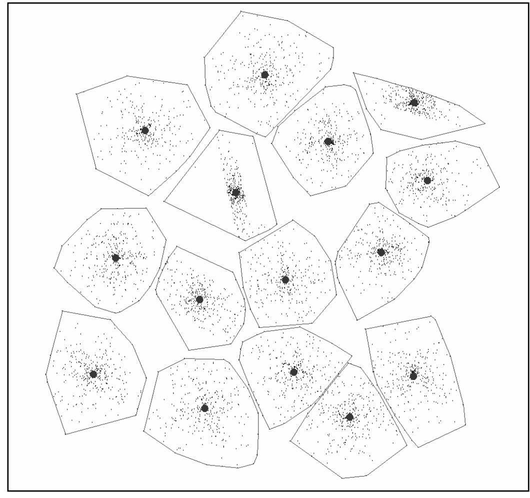
Sentroidin korvaus muuttaa ratkaisua niin paljon, että pelkkä paikallinen uudelleenositus ei välttämättä takaa uuden ratkaisun olevan kilpailukykyinen nykyiseen ratkaisuun nähden. Näin ollen ratkaisua hienosäädetään tekemällä kaksi K-means –iteraatiota. Ratkaisu ensimmäisen K-means –iteraation jälkeen on esitetty kuvassa 4.3a ja ratkaisu kahden iteraation jälkeen kuvassa 4.3b. Kuvista voidaan silmämääräisesti havaita ratkaisun parantuneen, sillä klusterit ovat tulleet tiiviimmiksi ja siistimmiksi. Ratkaisun laatu on myös parantunut keskineliövirheellä mitattuna.



a) Kuvan 4.2d ratkaisu yhden K-means-iteraation jälkeen (MSE ≈ $3{,}29 \cdot 10^9$)



b) Kuvan 4.2d ratkaisu kahden K-means-iteraation jälkeen (MSE ≈ $2{,}58 \cdot 10^9$)



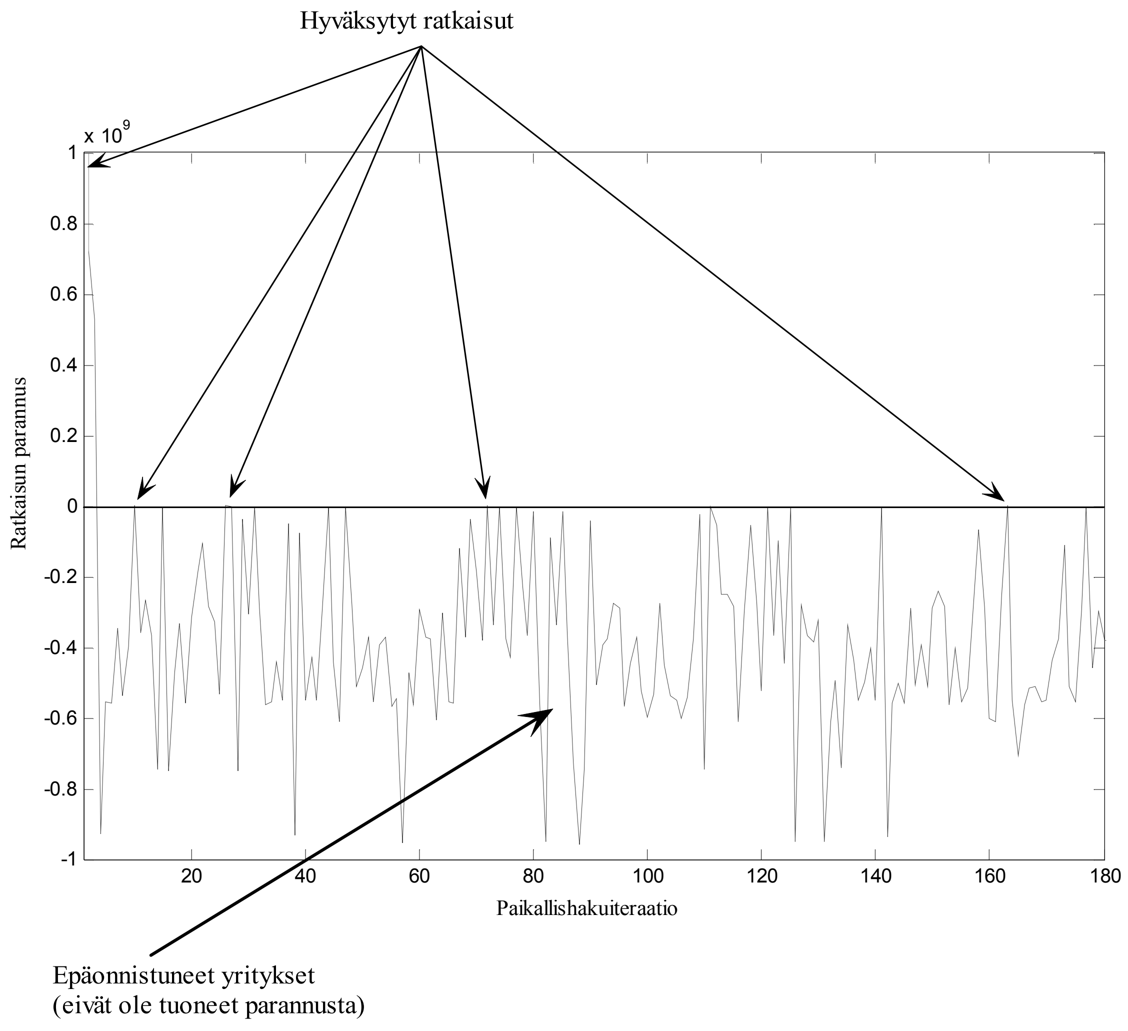
c) Lähtöratkaisu 162 paikallishakuiteraation jälkeen (MSE ≈ $1{,}33 \cdot 10^9$)

Kuva 4.3: Kuvassa 4.2d esitetyn ratkaisun kehittyminen.

Kuvan 4.2a lähtöratkaisu on nyt satunnaistetun paikallishaun nykyinen ratkaisu ja kuvan 4.3b ratkaisu on sen pohjalta muodostettu uusi ratkaisu. Muodostettu uusi ratkaisu on nykyistä parempi, joten uudesta ratkaisusta tehdään nykyinen ratkaisu seuraavalle paikallishakuiteraatiolle.

Kuvan 4.3b ratkaisusta löydetään helposti vielä lisää paranneltavaa. Esimerkiksi ylhäällä keskellä on kolme sentroidia yhden klusterin läheisyydessä ja keskeltä puuttuu yksi sentroidi sekä vasemmalta että oikealta. Kuvassa 4.3c on esitetty ratkaisu 162 paikallishakuiteraation jälkeen. Kuvasta 4.3c havaitaan silmämääräisesti, että satunnaistettu paikallishaku on onnistunut löytämään klusterit. Lisäksi havaitaan ratkaisun parantuneen huomattavasti myös keskineliövirheellä mitattuna verrattuna kuvan 4.3b ratkaisuun.

Paikallishaku ei kuitenkaan onnistu parantamaan ratkaisua läheskään jokaisella iteraatiolla. Kuvassa 4.4 on esitetty paikallishaun luomien ratkaisuehdokkaiden aiheuttama ratkaisun parannus eli virheen pienennys kullakin iteraatiolla. Parannus on mitattuna sen hetkiseen parhaimpaan ratkaisuun nähden.



Kuva 4.4: Ratkaisuehdokkaat ja ratkaisun parannus.

Kuvien 4.2 ja 4.3 esimerkissä vain seitsemän ratkaisuehdokasta 180 ratkaisuehdokkaasta on johtanut ratkaisun paranemiseen. Saadun parannuksen määrää on havainnollistettu kuvassa 4.4, josta

havaitaan parannusta saavutettavan sitä harvemmin, mitä enemmän iteraatioita suoritetaan. Lisäksi havaitaan saavutetun parannuksen määrän vähenevän iteraatioiden myötä, sillä kuvaajan alkupäässä olevat piikit ovat loppupään piikkejä korkeampia. Tarkempi selvitys parannukseen johtavien ratkaisuehdokkaiden määristä eri aineistoilla löytyy taulukosta 6.1.

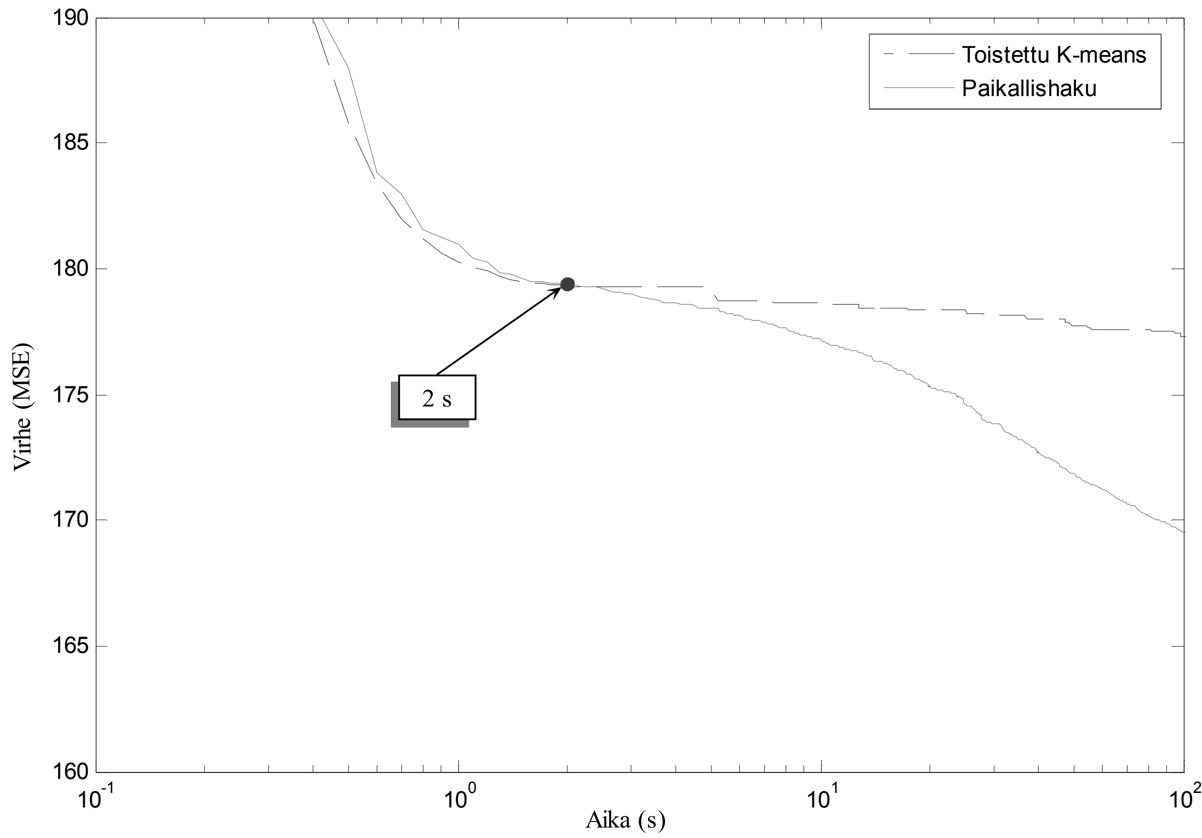
4.2. Satunnaistetun paikallishaun analysointia

Haettaessa klusterointiongelman ratkaisua paikallishaulla on käytännössä tehtävä kompromissi ratkaisun laadun ja käytettävissä olevan ajan välillä. Toisin sanoen pyritään löytämään mahdollisimman hyvä ratkaisu mahdollisimman nopeasti. Käytettävissä oleva aika kuitenkin vaihtelee tilanteen mukaan. Interaktiivisissa ja reaaliaikaisissa sovelluksissa (Leon & al., 2004; Lázaro & al., 2005) voi pari sekuntiakin olla liikaa. Toisaalta sovelluksissa, joissa laskentaa tehdään *erillisenä* (*offline*), voi aikaa olla käytettävissä periaatteessa niin paljon kuin tarpeen.

Esimerkiksi interaktiivisten sovellusten kohtuullisen ajan määrä ei ole muuttunut ajan kuluessa. Laskentatehon kasvu on kuitenkin aiheuttanut sen, että nykyisin voidaan näissäkin sovelluksissa käyttää entistä monimutkaisempia menetelmiä. Esimerkiksi Wu ja Witten (1985) raportoivat yhden *täysvärikuvan* (*full-color image*) klusteroinnin vieneen yli 20 tuntia silloisella *VAX 780* tietokoneella. Vertailukohtana voidaan pitää *House* aineistoa, joka on täysvärikuva ja jonka klusterointiin K-meansilla kului testauksessamme vain kymmenen sekuntia. Luvussa 5 esitettyjen tehostuksien jälkeen klusterointiaika putosi noin kahteen sekuntiin. K-meansia on siis vielä 80-luvulla pidetty ilmeisen hitaana moniin käyttötarkoituksiin, mutta nykypäivänä kaikkine tehostuksineen sitä voitaneen pitää nopeana. Vastaavasti paikallishakuun perustuvat menetelmät, kuten satunnaistettu tai geneettinen paikallishaku, voivat nykypäivänä vaikuttaa hitailta haettaessa laadultaan hyvää ratkaisua, mutta tulevaisuudessa laskentakapasiteetin kehittyessä näitäkin menetelmiä voidaan pitää nopeina.

Tämä kaikki johtaa siihen, että emme voi ainoastaan tuijottaa suoritusaikaan ja katsoa, onko menetelmä mielestämme nopea vai hidas. Haluamme siis klusterointimenetelmän, joka antaa parhaan vastineen käyttämälleen ajalle, parhaan aika-laatu –suhteen. Voidaan määritellä, että menetelmä on sitä *tehokkaampi*, mitä paremman ratkaisun se pystyy tietyssä ajassa tuottamaan. Erityisesti kahdesta menetelmästä tehokkaampi on se, kumman muodostama ratkaisu tietyllä ajanhetkellä on parempi.

Kuvassa 4.5 on verrattu paikallishakua ja toistettua K-meansia. Vertaaminen K-meansiin on relevanttia, sillä paikallishaun voisi pahimmassa tapauksessa vain tuottaa uusia satunnaisia ratkaisuja K-meansille. Kuva 4.5 kuitenkin osoittaa, ettei näin ole, sillä paikallishaku voittaa toistetun K-meansin. Toistettu K-means on aluksi paikallishakua hieman tehokkaampi. Tilanne kuitenkin kääntyy paikallishaun eduksi reilun 2 sekunnin jälkeen ja mitä enemmän aikaa menee, sitä ylivoimaisempi paikallishaku on toistettuun K-meansiin verrattuna. Paikallishaku on tässä tapauksessa toistettua K-meansia tehokkaampi, kun aikaa on käytettävissä enemmän kuin 2 sekuntia.



Kuva 4.5: Toistetun K-meansin ja paikallishaun vertailua virhe-aika –asteikolla *Bridge*-aineistolle. Kuvaajat ovat 10 ajon keskiarvoja.

Taulukossa 4.1 on esitetty toistetun K-meansin ja paikallishaun tuloksia *Bridge*, *House*, *Miss America* sekä *Birch1* aineistoille. Taulukosta nähdään, että paikallishaku voittaa K-meansin aina, jos käytettävissä on riittävästi aikaa. Aineiston *Bridge* tapauksessa aikaa tarvitaan viiden sekunnin luokkaa, aineiston *Miss America* tapauksessa 10 sekunnin luokkaa ja aineistojen *House* sekä *Birch1* tapauksessa jo noin sadan sekunnin luokkaa. Erot tämän ajan suhteen selittynevät aineistojen kokojen eroavaisuuksilla, sillä isoilla aineistoilla kestää pidempään ennen kuin paikallishaun lähestymistavasta saadaan hyötyä.

Taulukko 4.1: Toistetun K-meansin ja paikallishaun MSE-arvojen vertailua.

	Bridge		*House*		*Miss America*		*Birch1*	
	K-means	RLS	K-means	RLS	K-means	RLS	K-means	RLS
2 s	178,93	179,19	6,88	7,27	5,98	6,03	7,69	9,15
5 s	178,20	178,07	6,59	6,77	5,93	5,94	6,01	6,34
20 s	177,40	174,83	6,54	6,58	5,90	5,86	5,39	5,56
100 s	176,64	169,29	6,46	6,25	5,83	5,68	5,26	4,88

4.2.1. Parametrien vaikutus

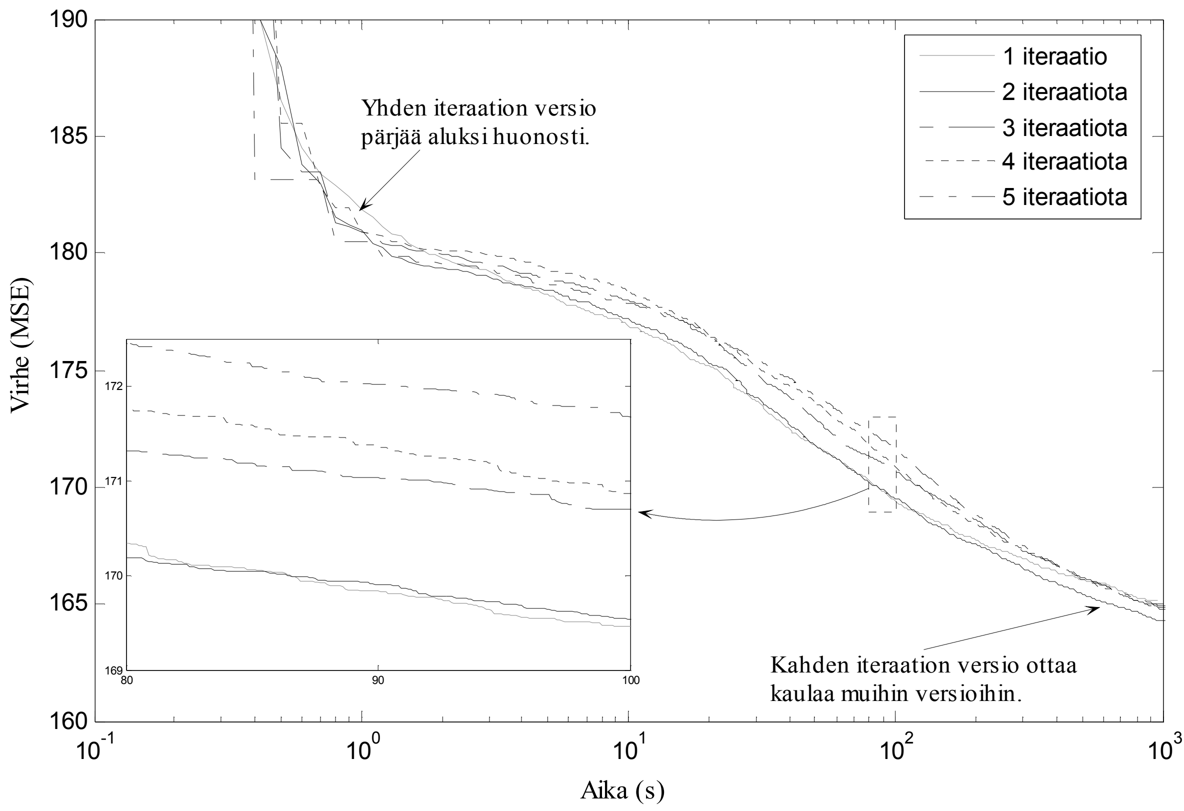
Klustereiden määrän ollessa tiedossa on paikallishaussa kaksi säädettävää parametria, jotka ovat paikallishakuiteraatioiden määrä T sekä K-means –iteraatioiden määrä I. Paikallishaun aikavaativuusanalyysin mukaisesti paikallishakuiteraatioiden määrä vaikuttaa suoraan algoritmin käyttämään aikaan. Toisaalta mitä enemmän iteraatioita suoritetaan, sitä parempi ratkaisu on mahdollista saada,

sillä mitä enemmän iteraatioita suoritetaan, sitä enemmän eri ratkaisuja käydään läpi. Näin ollen paikallishakuiteraatioiden määrällä voidaan säätää ratkaisun laadun ja käytetyn ajan suhdetta. Kohdassa 4.2.2 on yritetty arvioida tarvittavien paikallishakuiteraatioiden määrää todennäköisyyslaskennan keinoin.

Sen sijaan K-means –iteraatioiden määrä on mahdollisesti kriittinen tekijä, sillä sentroidin korvauksella muodostettua ratkaisua on parannettava *LocalRepartition*- ja *KMeans*-operaatioilla riittävästi, jotta se olisi kilpailukykyinen nykyisen ratkaisun kanssa. Toisaalta muodostettu ratkaisu voi olla niin huono, ettei sitä voi parantaa kilpailukykyiseksi, joten ratkaisun paranteluun ei myöskään kannata käyttää liikaa aikaa.

Kuvassa 4.6 on esitetty K-means –iteraatioiden määrän vaikutus paikallishaun tehokkuuteen. Havaitaan, että aineiston *Bridge* tapauksessa K-means –iteraatioita kannattaa ehdottomasti tehdä, sillä tulokset käytettäessä K-meansia ovat huomattavasti paremmat kuin ilman K-meansia. Itse asiassa ilman K-meansia ajetun paikallishaun käyrä menee niin ylhäällä, ettei se ole edes sopinut kuvaan 4.6. Esimerkiksi 10 sekunnin jälkeen virhe ilman K-meansia on noin 211,21 ja virhe K-meansia käytettäessä on pienimmillään yhden iteraation tapauksessa, jolloin virhe on 176,95, ja suurimmillaan neljän iteraation tapauksessa, jolloin virhe on 178,37.

Kuvassa 4.6 ensimmäisen sekunnin ajan viiden iteraation versio pärjää parhaiten, mutta pian tämän jälkeen yhden ja kahden iteraation versiot tuottavat paremman tuloksen. Viiden iteraation version menestys selittyy sillä, että K-means –iteraatioiden määrää kasvatettaessa haku alkaa muistuttamaan yhä enemmän K-meansia ja kuten taulukosta 4.1 havaittiin paikallishaku häviää K-meansille iteroinnin alkuvaiheessa. Noin 200 sekunnin jälkeen kahden iteraation versio alkaa olemaan yhden iteraation versiota parempi ja noin 1000 sekunnin jälkeen kahden iteraation versio on selvästi parempi muihin versioihin verrattuna. Tällöin virhe kahden iteraation tapauksessa on 164,28 ja muilla iteraatiomäärillä välillä 164,78–165,15. Kahden iteraation menetelmä pärjää tasaisen hyvin koko tarkastellulla ajanjaksolla, joten kaksi iteraatiota on sopivin määrä algoritmin tehokkuuden kannalta tällä aineistolla.



Kuva 4.6: K-means –iteraatioiden määrän vaikutus paikallishaun tehokkuuteen *Bridge*-aineistolla. Kuvaajat ovat 10 ajon keskiarvoja.

Edellä esitetyt havainnot yleistyvät melko hyvin myös muillekin aineistoille. Taulukossa 4.2 on esitetty paikallishaun tuottaman ratkaisun virhe eri K-means –iteraatioiden määrillä. Havaitaan, että kaksi K-means –iteraatiota on paras määrä algoritmin tehokkuuden kannalta, sillä se saavuttaa parhaan tuloksen kaikilla tutkituilla aineistoilla. Tulokset paljastavat kahden K-means –iteraation olevan hyvä valinta paikallishaun tehokkuuden kannalta, mikä tukee Fräntin ja Kivijärven (2000) valitsemaa kahta K-means –iteraatiota. Jatkossa käytetäänkin paikallishaussa kahta K-means –iteraatiota.

Taulukko 4.2: K-means –iteraatioiden määrän vaikutus ratkaisun laatuun, kun aikaa on käytettävissä 1000 sekuntia. Luvut ovat 10 ajon keskiarvoja.

	Bridge	*House*	*Miss America*	*Birch1*
1	165,15	6,01	5,36	4,64
2	164,28	6,00	5,33	4,64
3	164,84	6,01	5,42	4,64
4	164,78	6,02	5,45	4,65
5	164,91	6,01	5,49	4,64

4.2.2. Paikallishakuiteraatioiden määrä

Oletetaan, että ratkaisussa on ainoastaan yksi sentroidi väärässä paikassa, jolloin yksi sentroidin korvaus riittää muuttamaan ratkaisun optimaaliseksi. Koska sentroideja on M kappaletta, todennäköisyys valita juuri oikea klusteri poistettavaksi on $1/M$. Koska lisäksi oletamme kaikkien klustereiden olevan samankokoisia, juuri oikean kohdeklusterin valinnan todennäköisyys on myös $1/M$. Täten todennäköisyys tehdä haluttu sentroidin korvaus on vähintään $(1/M)^2$.

Paikallisen uudelleenosituksen ja K-meansin suorittama hienosäätö pystyy kuitenkin korjaamaan tilanteen, vaikka sentroidin poisto tai lisäys tehtäisiin halutun klusterin naapuriklusteriin. Tämän vuoksi ei ole välttämätöntä valita juuri oikeita klustereita, vaan sekä poistettavaksi että lisättäväksi klusteriksi voidaan valita mikä tahansa oikeiden klustereiden naapureista.

Naapuriklustereiden määrä a riippuu klustereiden määrästä M ja aineiston ulotteisuudesta K. Yläraja naapureiden lukumäärälle a on klustereiden lukumäärä. Naapureiden määrä kasvaa luultavasti eksponentiaalisesti ulotteisuuden suhteen, mikä tarkoittaa $a = O(2^K)$. Tämä on kuitenkin liian pessimistinen arvio ainakin klusteroituneelle aineistoille, kuten voidaan havaita *S*- ja *Birch*-aineistojen kohdalla kuvista 2.6 ja 2.7. Voidaan sanoa, että naapuriklustereiden määrä kertoo siis jotain aineiston rakenteesta, sen klusteroituneisuudesta.

Todennäköisyys suorittaa haluttu sentroidin korvaus on

$$p = (a / M)^2 . \qquad \textbf{(4.1)}$$

Onnistuneen korvauksen todennäköisyys siis laskee, kun klusterien määrä kasvaa, ja kasvaa ulotteisuuden kasvaessa. Onnistumisen todennäköisyys on riippumaton aineiston koosta N. Mikäli virheellisiä klustereita on b kappaletta ja oletetaan jokaisella iteraatiolla olevan vain yksi oikea korvaus, on vastaava onnistumisen todennäköisyys

$$1 - \sum_{i=0}^{b-1} \binom{n}{i} p^i (1-p)^{n-i} . \qquad \textbf{(4.2)}$$

Käyttämällä kaavassa 4.1 esitettyä todennäköisyyttä voimme laskea tarvittavien iteraatioiden määrän T. Oletetaan, että ainoastaan yksi klusteri on väärässä paikassa ja että oikea klusterointi tahdotaan löytää todennäköisyydellä $p_2 = 0{,}999$. Tarvittava iteraatioiden määrä on *geometrisesti jakautunut*, joten saadaan

$$(1 - p_2)^T = 1 - p$$

Tarvittava iteraatioiden määrä on näin ollen

$$T = \frac{\log(1 - p_2)}{\log(1 - p)} \qquad \textbf{(4.3)}$$

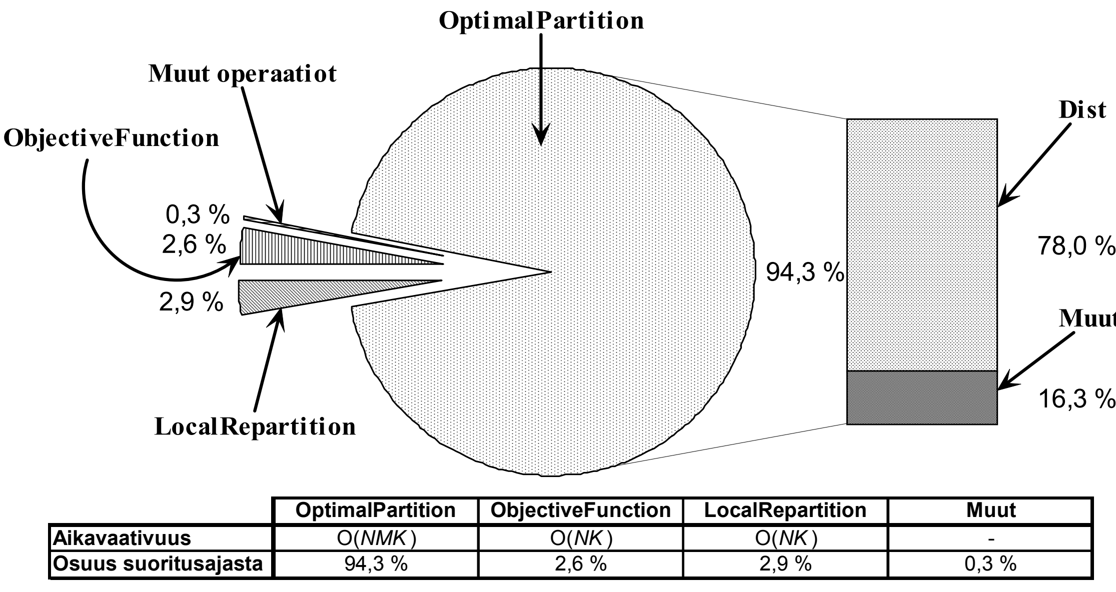
Pienehkö otos *S*-aineistojen satunnaisista lähtöratkaisuista paljastaa, että yksi klusteri väärässä paikassa on itse asiassa hyvä arvio näiden aineistojen tapauksessa. Nimittäin 25 tapauksessa 40:stä oli lähtöratkaisussa vain yksi tai ei yhtään sentroidia väärin ja 15 tapauksessa kaksi väärin. Taulukossa 4.3 onkin tutkittu kaavan 4.3 toimivuutta *S*-aineistoilla. Havaitaan, että kaava toimii suuntaa antavasti. Kaavan antama arvio on nimittäin luokkaa kymmeniä tai korkeintaan sata iteraatiota ja todellisuudessa tarvittava määrä on joitakin kymmeniä iteraatioita.

Taulukko 4.3: Tarvittavien paikallishakuiteraatioiden arvioidut ja todelliset määrät *S*-aineistoilla. Arvioiden laskemiseksi tarvittava naapuriratkaisujen määrä on määritetty empiirisesti tutkimalla satunnaisia lähtöratkaisuja. Todelliset arvot ovat 10 toiston keskiarvoja.

	S1	*S2*	*S3*	*S4*
Empiirisesti laskettu	33	19	22	27
Teoreettinen tulos	108	85	82	72

4.2.3. Profilointi

Tiedämme nyt, kuinka paikallishaun parametrien arvot tulee valita. Seuraavaksi analysoimme paikallishaun tehokkuutta hieman tarkemmin. Kuvassa 4.7 on esitetty yhteenvetona *GNU gprof* (FSF, 2006) –profilointityökalun antamat paikallishaun eri operaatioiden käyttämät ajat, kun klusteroitavana aineistona on ollut *S3*. Kuvan 4.7 ympyräkaaviossa on esitetty kolmen eniten aikaa vaatineen operaation *LocalRepartition*, *ObjectiveFunction* ja *OptimalPartition* suhteelliset ajat prosentteina. Kuvan 4.7 oikealla puolella olevassa palkkikaaviossa on lisäksi esitetty *OptimalPartition*-operaation käyttämän ajan jakaantuminen *Dist*-operaatiolla suoritettujen etäisyyslaskujen ja muiden toimien kesken.



	OptimalPartition	ObjectiveFunction	LocalRepartition	Muut
Aikavaativuus	O(*NMK*)	O(*NK*)	O(*NK*)	-
Osuus suoritusajasta	94,3 %	2,6 %	2,9 %	0,3 %

Kuva 4.7: Paikallishaun eniten aikaa vievät operaatiot.

Kuvasta nähdään, että suurin osa paikallishaun käyttämästä ajasta kuluu *OptimalPartition*-operaation suoritukseen, josta suurin osa kuluu alkioiden välisiä etäisyyksiä *Dist*-operaatiolla laskettaessa. *OptimalPartition*-operaation käyttämä aika on 297,54 sekuntia, mikä on noin 94 % kokonaisajasta. Edelleen *Dist*-operaatio käyttää aikaa 246,26 sekuntia, mikä on noin 78 % kokonaisajasta. *OptimalPartition*-operaation suurehko aikavaativuus on sopusoinnussa paikallishaun aikavaativuusanalyysin kanssa, jonka mukaan se on työläin operaatio aikavaativuuden ollessa O(*NMK*). Muiden operaatioiden aikavaativuuksiksi saatiin joko O(*NK*) tai O(1).

Kuvan alaosan taulukossa on esitetty kullekin operaatiolle sekä aikavaativuusluokka että mitattu suhteellinen suoritusaika. Taulukosta havaitaan *OptimalPartition*-operaation vievän noin 33–37 kertaa enemmän aikaa *ObjectiveFunction*- ja *LocalRepartition*-operaatioihin verrattuna. Tämä havainto selittyy aikavaativuusanalyysin avulla, sillä aikavaativuuksien perusteella *OptimalPartition*-

operaatio vie *ObjectiveFunction*- ja *LocalRepartition*-operaatioihin nähden M kertaa enemmän aikaa. Lisäksi aiemmin päätettiin K-means –iteraatioiden määräksi kaksi, joten *OptimalPartition*-operaatiota suoritetaan kaksi kertaa kutakin *ObjectiveFunction*- ja *LocalRepartition*-operaatiota kohden. Aikavaativuusanalyysimme mukaan *OptimalPartition*-operaation tulee olla $2 \cdot M = 30$ kertaa hitaampi, mikä vastaa melko hyvin saamaamme tulosta.

Seuraavassa luvussa tutkitaan mahdollisuuksia paikallishaun tehostamiseksi edellä mainittuja operaatioita nopeuttamalla.

5. Paikallishaun tehostaminen

Kutsumme jatkossa luvussa 4 esitettyä paikallishakua *paikallishaun perusversioksi* erona tässä luvussa esitettävään *paikallishaun edistyneeseen versioon*. Vastaavasti samoja nopeutusmenetelmiä käyttävää versiota toistetusta K-meansista kutsumme sen edistyneeksi versioksi.

Sekä luvun 3 ja 4 aikavaativuusanalyysin että kuvassa 4.7 esitettyjen tulosten nojalla ilmeinen keino paikallishaun nopeuttamiseen on keskittyä *OptimalPartition*-menetelmän nopeuttamiseen. Koska suurin osa ajasta kuluu alkioiden välisten etäisyyksien laskemiseen, on nopeuttamiseen kaksi mahdollisuutta. Ensimmäinen mahdollisuus on vähentää suoritettavien etäisyyslaskujen määrää, mikä vaatii, että pystytään välttämään tiettyjen alkioiden ja sentroidien välisten etäisyyksien laskeminen. Toinen mahdollisuus on nopeuttaa etäisyyslaskennan suorittavaa *Dist*-operaatiota.

Kaukorannan ja muiden (2000) mukaan osituksen muodostamisen tehostusmenetelmät voidaan jakaa *eksakteihin* (*exact*) ja *approksimoiviin* (*approximate*) menetelmiin. Eksaktit menetelmät löytävät kullekin alkiolle lähimmän sentroidin, mutta approksimoivat menetelmät löytävät kutakin alkiota lähellä olevan sentroidin, mutta ei välttämättä lähintä sentroidia. Haluamme säilyttää osituksen optimaalisena säilyttääksemme edistyneen version vertailukelpoisena perusversion kanssa, joten jatkossa keskitymme vain eksakteihin menetelmiin.

5.1. Lähimmän naapurin haun nopeuttaminen

Optimaalisen osituksen muodostamisessa etsitään kullekin aineiston alkiolle *lähin naapuri* (*nearest neighbour*) sentroidien joukosta. Lähimmän naapurin tehokkaaseen hakuun on esitetty runsaasti erilaisia menetelmiä, esimerkiksi Aghbari (2005), Hjaltason ja Samet (2003), Bandyopadhyay ja Maulik (2002), Ra ja Kim (1993), Chen ja Tsieh (1991) sekä Bei ja Gray (1985). Seuraavassa esitellään näistä kolme viimeksi mainittua.

PDS-menetelmä (*partial distortion search*; Bei & Gray, 1985) nopeuttaa varsinaista etäisyyslaskentaa. Kun optimaalista ositusta muodostettaessa halutaan laskea alkion ja sentroidin välinen etäisyys, halutaan itse asiassa vain tietää, onko kyseinen sentroidi lähempänä alkiota kuin siihen mennessä löydetty lähin sentroidi. PDS-menetelmää käyttävälle etäisyysfunktiolle annetaankin alkio ja sentroidi sekä etäisyys siihen mennessä löydettyyn lähimpään sentroidiin. PDS-menetelmää käyttävä etäisyysfunktio laskee yhteen alkion ja sentroidin komponenttien välisten erotuksien neliöitä ja lopettaa laskennan, jos minimietäisyys ylittyy, sillä tällöin tiedetään, ettei kyseinen sentroidi voi olla lähin. PDS-menetelmän tehokkuus riippuu suuresti nykyisestä lähimmästä alkiosta (Kaukoranta & al. 2000). Näin ollen lähintä sentroidia etsittäessä lasketaankin jatkossa ensin etäisyys aiemman osituksen mukaiseen lähimpään sentroidiin, joka toimii ikään kuin lähtöarvauksena.

TIE-menetelmä (*triangular inequality elimination*; Chen & Tsieh, 1991) hyödyntää tietoa, että etäisyysmitta toteuttaa *kolmioepäyhtälön* (*triangular inequality*), josta johdetun ehdon nojalla pystyy välttämään alkion ja tiettyjen sentroidien välisten etäisyyksien laskemisen. Mikäli alkion x_i tähän mennessä löydetty lähin sentroidi on c_a, niin kaavassa 5.1 annettu ehto kertoo, milloin tutkittava sentroidi c_b ei voi olla lähempänä kuin c_a.

$$d(c_a, c_b)^2 > 4d(x_i, c_a)^2 \qquad \textbf{(5.1)}$$

TIE-menetelmän tehokas toteutus vaatii, että ylläpidetään taulukkoa kaikkien sentroidien välisistä etäisyyksistä. Näiden etäisyyksien laskeminen vaatii $O(M^2K)$ tilan ja ajan, mikä on kuitenkin usein

kohtuullista, sillä osituksen muodostamisen aikavaativuus on O(*NMK*) ja yleensä $M << N$. Lisäksi etäisyystaulukon tehokas käyttö vaatii, että rivit ovat järjestetty etäisyyksien mukaisesti kasvavaan järjestykseen. TIE-menetelmässä lähimmän sentroidin haku alkiolle x_i tapahtuu seuraavasti. Alkion x_i edellisen osituksen mukainen lähin sentroidi c_a valitaan lähtöarvaukseksi. Tämän jälkeen etäisyydet muihin sentroideihin lasketaan etäisyysmatriisin rivin antamassa järjestyksessä käyttäen PDS-menetelmää. Kaavan 5.1 mukaisesti voidaan etäisyyslaskenta lopettaa, kun sentroidit menevät liian kauaksi sentroidista c_a.

MPS-menetelmä (*mean-distance-ordered partial search*; Ra & Kim, 1993) pystyy etäisyysmitan ominaisuuksista johdetun ehdon nojalla välttämään alkion ja tiettyjen sentroidien välisten etäisyyksien laskemisen. Nimittäin mikäli alkion x_i tähän mennessä löydetty lähin sentroidi on c_a, niin kaavan 5.2 ehto kertoo, milloin tutkittava sentroidi c_b ei voi olla lähempänä kuin c_a.

$$\left(\sum_{k=1}^{K} x_i^k - \sum_{k=1}^{K} c_b^k \right)^2 > K\ d(x_i, c_a) \qquad \textbf{(5.2)}$$

MPS-menetelmän tehokas toteutus lajittelee koodikirjan nousevaan järjestykseen sentroidien komponenttien keskiarvojen perusteella. Lähtöarvaus lähimmäksi sentroidiksi c_a saadaan valitsemalla se sentroidi, jonka komponenttien keskiarvo on lähinnä alkion x_i komponenttien keskiarvoa. Tämän jälkeen rajat lähimmän sentroidin haulle saadaan kaavasta 5.2. Etäisyyslaskuissa käytetään PDS-menetelmää. Menetelmä vaatii lajittelun vuoksi O(*M log M*) ajan.

Huomattavaa on, että TIE- ja MPS-menetelmät perustuvat tiettyihin etäisyysmitalle ja optimointifunktiolle tehtäviin oletuksiin. Esimerkiksi TIE-menetelmässä vaaditaan etäisyysmitan toteuttavan kolmioepäyhtälön. Nämä oletukset tosin toteutuvat euklidista etäisyyttä ja keskineliövirhettä käytettäessä, mutta eivät välttämättä muille etäisyysmitoille ja optimointifunktioille. PDS ei puolestaan tee yhtä vahvoja oletuksia, sillä se perustuu oletukseen, että kahden alkion välisen etäisyyden laskeminen koostuu positiivisten alkioiden summaamisesta ja tämä summaaminen voidaan katkaista, kun saatu etäisyys ylittää sen hetkisen minimietäisyyden.

Liitteessä 1 kuvattavassa paikallishaun edistyneessä versiossa tullaan käyttämään PDS-menetelmää. Syynä tähän on PDS-menetelmän yksinkertaisuuden lisäksi se, että PDS on yleisempi kuin muut esitetyt menetelmät. PDS-menetelmän voidaan näin ollen olettaa yleistyvän paremmin myös käytettäessä muita, mahdollisesti ei-metrisiä, *etäisyysmetriikoita*.

PDS, TIE ja MPS ovat yleisiä menetelmiä lähimmän naapurin haun nopeuttamiseksi. Näin ollen ne eivät ota huomioon sitä, että lähimmän naapurin hakua suoritetaan tässä tapauksessa K-meansin osana.

5.2. Karsittujen vertailujen haku

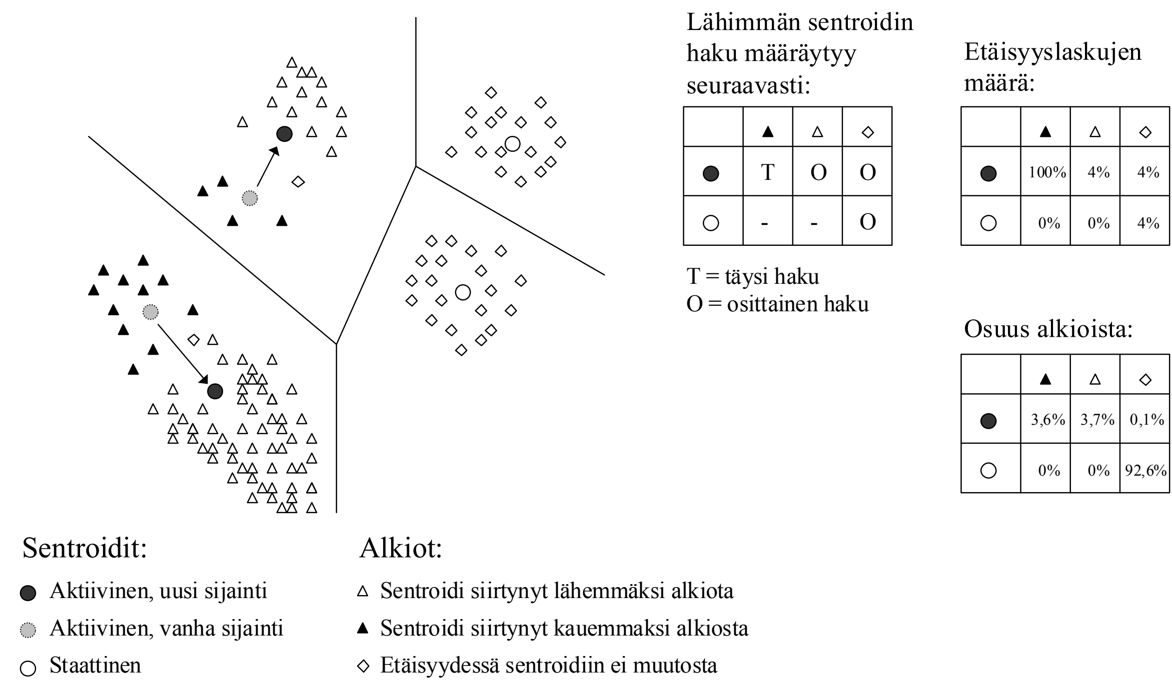
Kaukorannan ja muiden (2000) esittämä *karsittujen vertailujen haku* (*reduced-comparison search*) perustuu havaintoon, että K-means tekee vain paikallisia muutoksia koodikirjaan ja muutosten määrä vaihtelee suuresti klusterista toiseen. Tämän havainnon hyödyntämiseksi karsittujen vertailujen haussa sentroidit jaetaan aktiivisiin ja staattisiin. Sentroidi on *aktiivinen*, jos se on muuttunut edellisellä K-means –iteraatiolla, ja tällöin kyseisen sentroidin edustamaa klusteria kutsutaan *aktiiviseksi klusteriksi*. Vastaavasti muuttumattomia sentroideja kutsutaan *staattisiksi*, ja niiden edustamia klustereita *staattisiksi klustereiksi*. Staattisten sentroidien määrä kasvaa K-means –iteraatioiden

myötä (Kaukoranta & al., 2000). Aluksi sentroidit voivat muuttua runsaastikin, mutta K-means –iteraatioiden myötä yhä useammat klusterit, ja näin ollen myös niiden sentroidit alkavat stabiloitua.

Tietoa sentroidien aktiivisuuksista voidaan käyttää etäisyyslaskujen karsimiseksi. Ajatellaan tapausta, jossa alkio x_i kuuluu staattiseen klusteriin $a = p_i$, jonka sentroidi on c_a. Havaitaan, että tällöin mikään toinen staattinen sentroidi ei voi olla lähempänä kyseistä alkiota. Nimittäin mikäli jokin toinen staattinen sentroidi olisi lähempänä kyseistä alkiota kuin sentroidi c_a, olisi alkion x_i kuuluttava tähän toiseen klusteriin. Näin ollen ainoastaan jokin aktiivisista sentroideista voi olla alkiota x_i lähempänä ja siten riittää laskea alkion x_i etäisyydet vain aktiivisiin sentroideihin.

Tutkitaan seuraavaksi tilannetta, jossa alkio x_i kuuluu aktiiviseen klusteriin $a = p_i$, jonka sentroidi on c_a. Havaitaan tilanteen olevan sama kuin edellä kuvatun staattisen sentroidin tapauksessa, mikäli aktiivinen sentroidi c_a on siirtynyt edellisestä iteraatiosta lähemmäksi alkiota x_i. Tällöin ainoastaan jokin aktiivisista sentroideista voi olla lähempänä ja riittää laskea alkion x_i etäisyydet vain aktiivisiin sentroideihin. Mikäli alkion x_i sentroidi on siirtynyt edellisestä iteraatiosta kauemmaksi, ei vastaavia päätelmiä voida tehdä, joten joudutaan laskemaan etäisyydet kaikkiin sentroideihin.

Kuvassa 5.1 on havainnollistettu karsittujen vertailujen haun suorittamaa alkioiden luokittelua. Valkeilla kolmioilla on merkitty ne alkiot, joiden klusterin sentroidi on siirtynyt lähemmäksi alkiota, ja mustilla kolmioilla ne alkiot, joiden klusterin sentroidi on siirtynyt kauemmaksi alkiosta. Alkiot, joiden etäisyydessä sentroidiin ei tapahtunut muutosta, on merkitty valkeilla vinoneliöillä. Staattiset sentroidit on merkitty valkeilla ympyröillä, ja aktiivisista sentroideista on merkitty sentroidin aiempi sijainti vaaleanharmaalla ympyrällä ja nykyinen sijainti tummanharmaalla ympyrällä.



Kuva 5.1: Karsittujen vertailujen haun suorittama alkioiden luokittelu. Prosenttiluvut ovat *Bridge* aineistolle suoritettujen 10 toiston keskiarvoja.

Karsittujen vertailujen hakua käytettäessä etäisyyslaskujen määrää saadaan vähennettyä, sillä optimaalista ositusta muodostettaessa riittää tehdä $N{\cdot}M_a + N_b{\cdot}(M\text{-}M_a)$ etäisyyslaskua (Kaukoranta & al., 2000), missä M_a on aktiivisten sentroidien lukumäärä ja N_b niiden alkioiden lukumäärä, joiden sent-

roidi on liikkunut kauemmaksi. Toisin sanoen karsittujen vertailujen hakua käytettäessä lasketaan kaikkien alkioiden etäisyydet aktiivisiin sentroideihin, mutta vain sentroideistaan kauemmaksi joutuneiden alkioiden etäisyydet staattisiin sentroideihin.

Karsittujen vertailujen haun täytyy pitää yllä alkioiden ja sentroidien välisiä etäisyyksiä edelliseltä iteraatiolta, jotta saadaan selville sentroidien liikkeet suhteessa alkioihin. Etäisyyksien ylläpito vaatii $O(N)$ tilan. Haun on lisäksi selvitettävä entisiä ja nykyisiä sentroideja vertaamalla, mitkä sentroideista ovat kullakin iteraatiolla aktiivisia. Näiden aktiivisten sentroidien joukon ylläpito vaatii enintään $O(MK)$ ajan ja $O(M)$ tilan. Nämä toimet eivät kuitenkaan vaikuta osituksen muodostamisen aikavaativuuteen, sillä se on yhä $O(NMK)$.

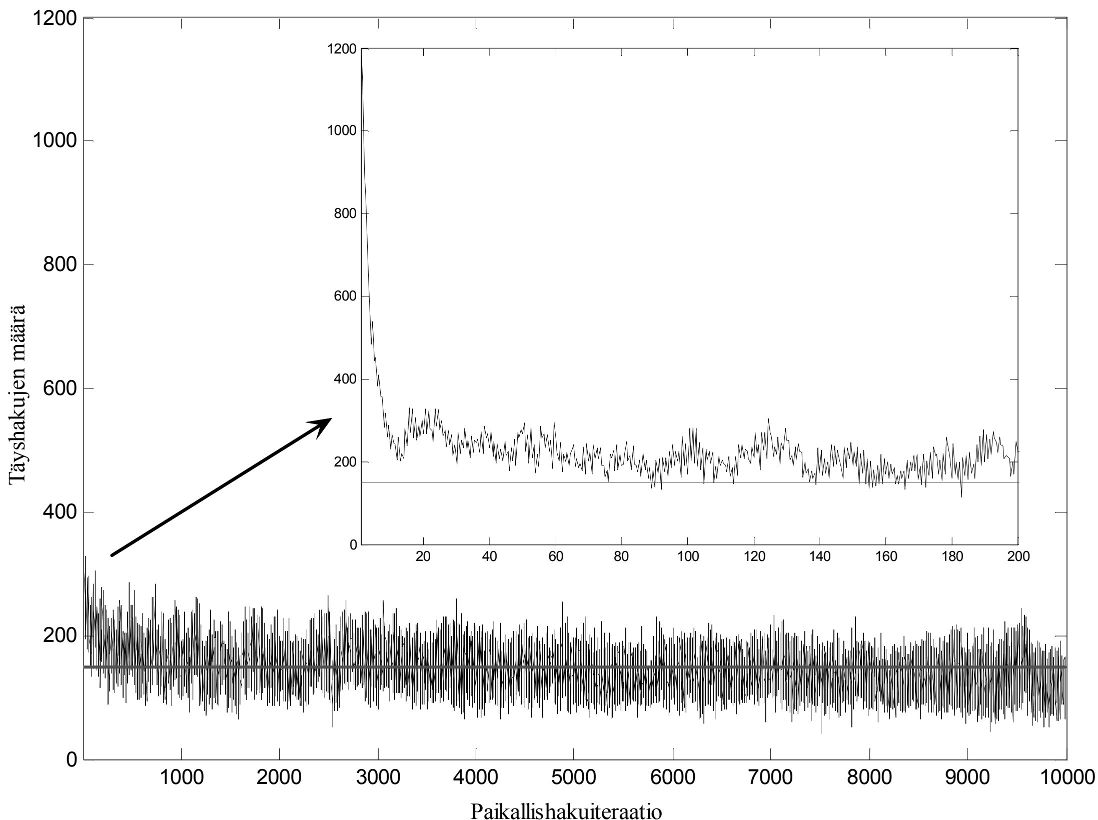
Tiukempi raja osituksen muodostamisen aikavaativuudelle saadaan vedettyä, kun arvioidaan, että ainoastaan poistettavan ja lisättävän klusterin naapuriklusterit tulevat olemaan aktiivisia. Kohdan 4.2.2. mukaisesti a on keskimääräinen naapuriklustereiden määrä. Aktiivisia sentroideja arvioidaan siis olevan $2a$ kappaletta ja niissä alkioita $2a{\cdot}N/M$ kappaletta, kun kaikkien klustereiden oletetaan olevan samankokoisia. Etäisyyslaskujen määräksi saadaan tällöin $N{\cdot}2a + 2a{\cdot}(N/M){\cdot}(M\text{-}2a)$, joka on $O(aN)$. Aikavaativuudeksi saadaan tällöin $O(aNK)$. Osituksen muodostamisen aikavaativuus riippuu siis naapureiden lukumäärästä a, joka taas riippuu klustereiden määrästä M ja ulotteisuudesta K ja yleensäkin aineiston rakenteesta.

Taulukossa 5.1 on esitetty karsittujen vertailujen haun suorittama alkioiden luokittelu eri ryhmiin, kun on ajettu 10000 paikallishakuiteraatiota. Esitetyt luvut ovat 20 toiston keskiarvoja. Taulukosta 5.1 nähdään, että karsittujen vertailujen haun käyttö todella tuo nopeusetua käytettäessä osana paikallishakua. Aineistosta riippuen 87–99 prosenttia alkioista kuuluu staattiseen klusteriin tai sen sentroidi on tullut lähemmäksi. Näissä tapauksissa riittää suorittaa kyseisen alkion lähimmän sentroidin haku ainoastaan aktiivisten sentroidien joukosta.

Taulukko 5.1: Alkioiden jakautuminen klusterien aktiivisuuden mukaan.

	Staattinen	Aktiivinen, tullut lähemmäs	Aktiivinen, siirtynyt kauemmaksi
Bridge	92,6 %	3,7 %	3,6 %
House	97,9 %	1,1 %	1,0 %
Miss America	97,5 %	1,2 %	1,2 %
Birch1	77,9 %	11,2 %	10,8 %
Birch2	94,5 %	2,7 %	2,9 %
Birch3	74,1 %	13,1 %	12,8 %

Kuvassa 5.2 on esitetty täyshakujen määrän kehittyminen. Täyshaulla tarkoitetaan tilannetta, jossa alkion lähimmän sentroidin haku joudutaan tekemään koko koodikirjasta. Kuvasta havaitaan, että aivan ensimmäisten iteraatioiden aikana tehdään todella paljon täyshakuja. Ensimmäisellä iteraatiolla täyshakuja tehdään noin 1200 ja seuraavallakin iteraatiolla 900. Tilanne kuitenkin tasaantuu nopeasti ja noin 10 iteraation jälkeen täyshakujen määrä asettuu noin 200–300 hakuun per iteraatio. Täyshakujen määrä vähenee tasaisesti loppua kohden ollen 6000–10000 iteraation välillä noin 100–200 hakua per iteraatio muutamia piikkejä lukuunottamatta. Verrattuna perusversioon tämä tarkoittaa, että tällöin tehdään 20–40 kertaa vähemmän täyshakuja. Näin ollen havaitaan, että tilanne rauhoittuu iteraatioiden myötä ja karsittujen vertailujen haku tuo sitä enemmän nopeutusta paikallishaulle, mitä enemmän iteraatioita suoritetaan.



Kuva 5.2: Täyshakujen määrän kehittyminen. Aineistona *Bridge* ja arvot 10 toiston keskiarvoja. Harmaalla viivalla on merkitty täyshakujen määrän keskiarvo.

Kaukorannan ja muiden (2000) mukaan esitetty karsittujen vertailujen haun antama nopeutus perustuu K-meansin yleisiin ominaisuuksiin eikä etäisyysmitan ominaisuuksiin kuten kolmioepäyhtälöön. Täten menetelmän odotetaan soveltuvan myös käytettäessä ei-metrisiä etäisyysmetriikoita. Menetelmä soveltuu käytettäväksi myös yhdessä edellä esiteltyjen PDS-, TIE- ja MPS-menetelmien kanssa. Kaukoranta ja muut (2000) raportoivat saavuttaneensa parhaat tulokset kuva-aineistoilla *Bridge*, *Miss America* ja *House* käyttämällä PDS- ja MPS-menetelmää yhdistettynä karsittujen vertailujen hakuun, jolloin suoritusaika on vain 2–4 % täyden haun ajasta.

5.3. Edistyneen version analysointia

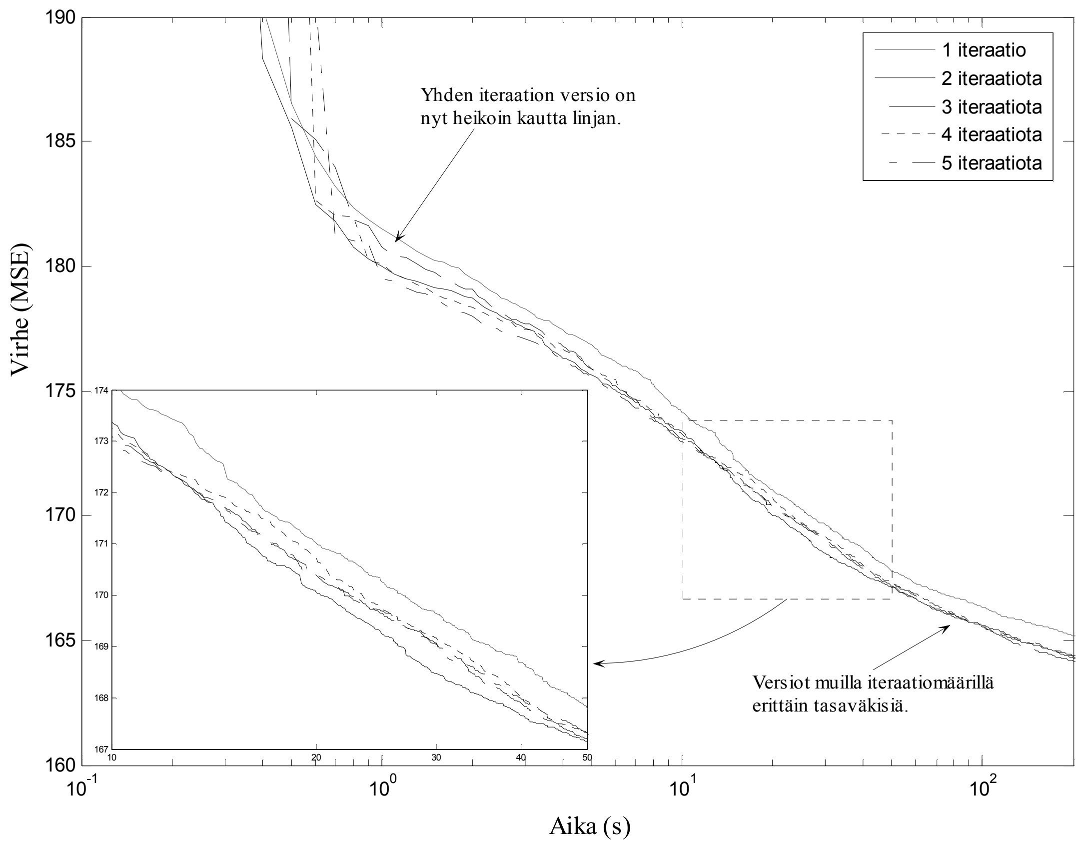
Paikallishaun sekä toistetun K-meansin edistynyt versio on nopeutettu käyttämällä PDS-menetelmää sekä karsittujen vertailujen hakua. Etäisyyslaskennan nopeuttamiseen on käytetty PDS-menetelmää, sillä se on esiteltyjä TIE- ja MPS-menetelmiä yksinkertaisempi ja yleiskäyttöisempi.

5.3.1. K-means –iteraatioiden määrä

Käytetyt nopeutusmenetelmät oletuksemme mukaan nopeuttavat *OptimalPartition*-operaatiota ja siten paikallishaun osana olevaa K-meansia. Näin ollen nopeutuksen myötä paras K-means –iteraatioiden määrä paikallishaulle on saattanut muuttua. Kuvassa 5.3 on esitetty K-means –iteraatioiden määrän vaikutus paikallishaun edistyneen version tehokkuuteen. Kuvasta havaitaan, että yhden iteraation versio on nyt huonoin kautta linjan. Paras määrä iteraatioita lienee kaksi tai kolme. Kahden iteraation versiota puoltaa tasaisen hyvä suoritus. Kolmen iteraation versio sen sijaan näyttäisi ku-

vaajan loppupuolella menestyvän hyvin, vaikka toisaalta se menestyi hieman heikosti suorituksen alkuvaiheilla.

Taulukossa 5.2 on esitetty paikallishaun tuottaman ratkaisun virhe eri K-means –iteraatioiden määrillä. Taulukosta havaitaan, että ei voida mitenkään yksiselitteisesti sanoa jonkin yhden iteraatiomäärän olevan yksiselitteisesti paras määrä algoritmin tehokkuuden kannalta, sillä paras iteraatiomäärä vaihtelee testattujen aineistojen välillä. Iteraatiomäärien saavuttamien tulosten eroavaisuudet ovat kuitenkin melko pieniä, alle 1 %:n luokkaa. Koska kahden iteraation versio pärjää taulukon testauksen mukaan kaikissa tapauksissa vähintäänkin kohtuullisesti, käytetään jatkossa paikallishaun edistyneessä versiossa kahta iteraatiota. Loppujen lopuksi voidaan tehdä johtopäätös, että vain muutama iteraatio riittää ja tarkan arvon valinnalla ei ole kovinkaan suurta merkitystä algoritmin suoritustehoon.



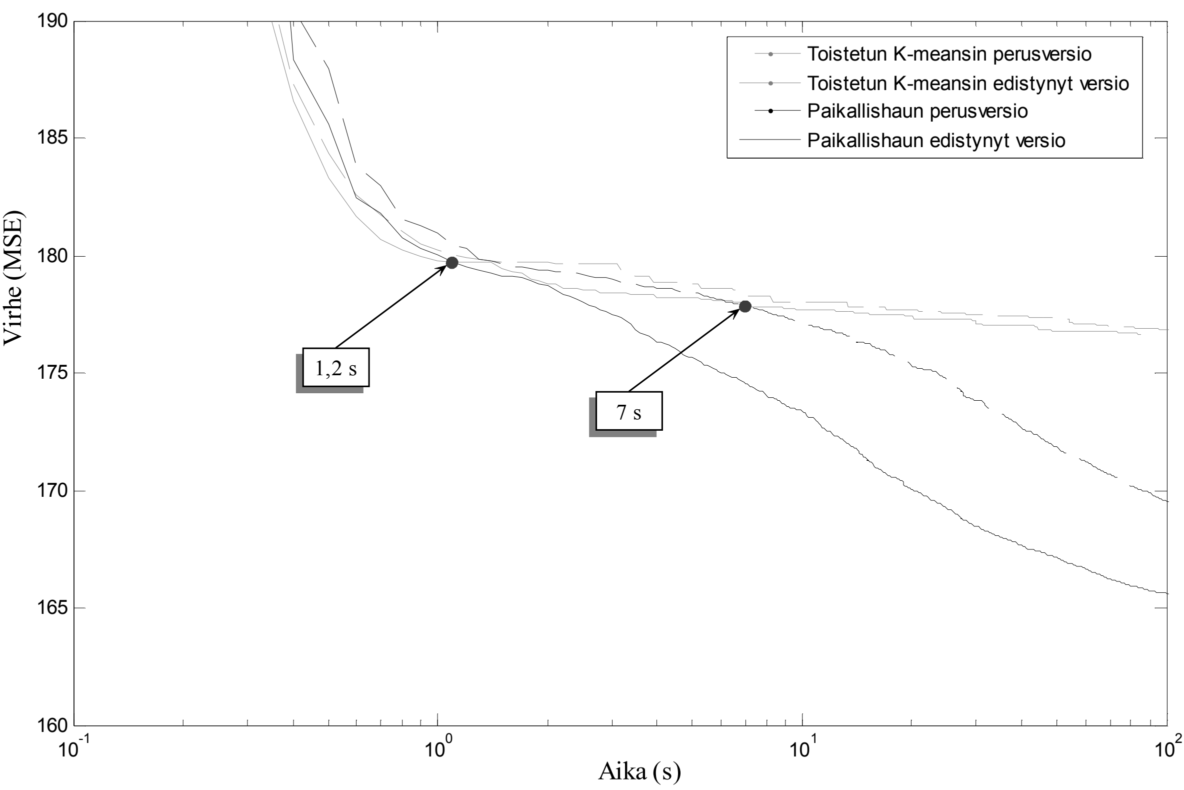
Kuva 5.3: K-means –iteraatioiden määrän vaikutus paikallishaun edistyneen version tehokkuuteen, kun klusteroitavana aineistona on *Bridge*. Kuvaajat ovat 10 ajon keskiarvoja.

Taulukko 5.2: K-means –iteraatioiden määrän vaikutus ratkaisun laatuun, kun aikaa käytetty 100 sekuntia. Luvut ovat 10 suorituksen keskiarvoja ja parhaat tulokset ovat lihavoituna.

	Bridge	*House*	*Miss America*	*Birch1*
1	166,35	6,02	**5,39**	4,86
2	165,62	6,01	5,40	**4,83**
3	**165,54**	6,00	5,42	4,86
4	165,59	5,99	5,40	4,85
5	165,56	**5,98**	5,41	4,85

5.3.2. Edistyneen paikallishaun vertailua

Kuvassa 5.4 on verrattu sekä paikallishaun että toistetun K-meansin perusversioita ja edistyneitä versioita toisiinsa. Kuvasta havaitaan, että toistetun K-meansin edistynyt versio on aluksi paikallishaun molempia versioita tehokkaampi. Tilanne kuitenkin kääntyy paikallishaun eduksi edistyneen version tapauksessa reilun yhden sekunnin jälkeen ja perusversionkin tapauksessa jo noin seitsemän sekunnin jälkeen. Lisäksi mitä enemmän aikaa kuluu, sitä ylivoimaisempi paikallishaku on toistettuun K-meansiin verrattuna. Paikallishaun molemmat versiot ovat tässä tapauksessa toistettua K-meansia tehokkaampi, kun aikaa on käytettävissä riittävästi. Paikallishaun edistynyt versio on aina perusversiota tehokkaampi, kuten myös toistetun K-meansin edistynyt versio on aina perusversiota tehokkaampi. Esimerkiksi kun aikaa on kulunut 50 sekuntia, toistetun K-meansin perusversion tuottaman ratkaisun virhe on 177,29 ja edistyneen version virhe on 176,79, mutta paikallishaun perusversion virhe 171,85 ja edistyneen version 167,15.



Kuva 5.4: Paikallishaun perusversion sekä toistetun K-meansin että paikallishaun edistyneiden versioiden vertailua virhe-aika –asteikolla, kun klusteroitavana aineistona on ollut *Bridge*. Kuvaajat ovat 10 toiston keskiarvoja.

Taulukossa 5.3 on esitetty toistetun K-meansin ja paikallishaun edistyneiden versioiden tuloksia *Bridge*, *House*, *Miss America* sekä *Birch1* aineistoille. Taulukosta nähdään, että paikallishaku voittaa toistetun K-meansin aina, jos käytettävissä on riittävästi aikaa. Aineistojen *Bridge*, *House* ja *Miss America* tapauksessa aikaa tarvitaan alla kaksi sekuntia. Aineiston *Birch1* tapauksessa aikaa tarvitaan sen sijaan jo noin 100 sekunnin luokkaa. Erot tämän tarvittavan ajan suhteen selittynevät aineiston *Birch1* muita suuremmalla alkiomäärällä, sillä isoilla aineistoilla kestää pidempään ennen kuin paikallishaun lähestymistavasta saadaan täysi hyöty irti.

Taulukko 5.3: Toistetun K-meansin ja paikallishaun edistyneiden versioiden tulokset.

	Bridge		*House*		*Miss America*		*Birch1*	
	K-means	RLS	K-means	RLS	K-means	RLS	K-means	RLS
2 s	179,32	170,10	7,23	7,08	6,07	5,98	10,08	8,09
5 s	178,17	167,15	6,61	6,55	5,97	5,85	5,91	6,02
20 s	177,57	170,10	6,55	6,18	5,89	5,66	5,32	5,49
100 s	176,80	165,62	6,48	6,01	5,87	5,40	5,10	4,83

Fränti ja Virmajoki (2006) ovat vertailleet paikallishaun edistynyttä versiota muihin klusterointimenetelmiin. Heidän mukaansa paikallishaku ei pärjää vertailussa kaikkein parhaimmille menetelmille, kuten geneettiselle paikallishaulle (Fränti & al, 1997b; Kivijärvi & al., 2003) tai *iterative shrinking* –menetelmälle (Fränti & Virmajoki, 2006). Paikallishaku on kuitenkin varteenotettava vaihtoehto yksinkertaisen toteutuksen ja helposti säädettävissä olevan ajankäytön vuoksi.

5.3.3. Saavutettu nopeutus

Taulukossa 5.4 on esitetty toistetun K-meansin ja paikallishaun suoritusajat sekä niiden perusversioille että edistyneille versioille. Taulukosta nähdään toistetun K-meansin ja paikallishaun nopeutuneen kaikilla tutkituista aineistoista. Paikallishaku on hyötynyt toistettua K-meansia enemmän käytetyistä nopeutusmenetelmistä. Esimerkiksi aineiston *Bridge* tapauksessa paikallishaun edistyneen version suoritusajaksi jäi 17 prosenttia alkuperäisestä ja toistetun K-meansin edistyneen version ajaksi jäi 56 prosenttia alkuperäisestä. Syy menetelmien nopeuseroon lienee toistetun K-meansin paikallishakua suurempi sentroidien aktiivisuus, sillä toistettu K-means muodostaa jokaisella iteraatiolla uuden satunnaisen lähtöratkaisun mutta paikallishaku muuttaa nykyisesta ratkaisusta vain yhden sentroidin sijaintia.

Havaitaan, että toistetun K-meansin ja paikallishaun nopeutumiset ovat kuitenkin kytköksissä toisiinsa. Nimittäin toistetun K-meansin edistyneelle versiolle jäi suoritusajaksi 48 prosenttia aineistolla *Miss America* ja paikallishaulla vastaavasti 9 prosenttia alkuperäisestä. Nyt verrattaessa esimerkiksi aineiston *Bridge* vastaaviin lukuihin havaitaan, että kummallakin menetelmällä nopeutuminen on ollut vähäisempää.

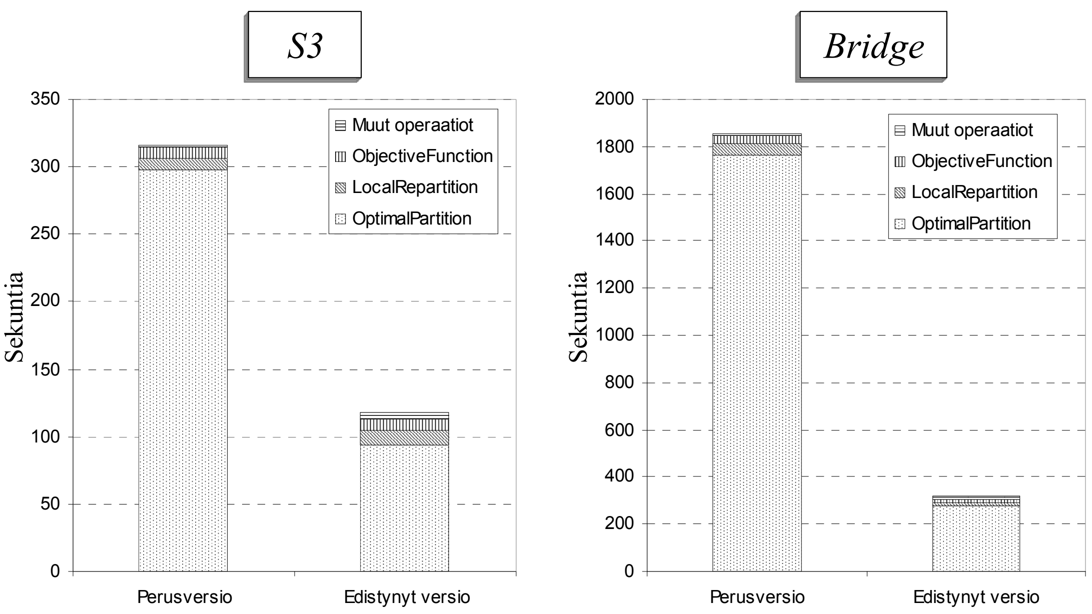
Taulukko 5.4: Toistetun K-meansin ja paikallishaun suoritusajat sekä ilman nopeutusta että nopeutusmenelmiä käyttäen. Toistettua K-meansia on ajettu 1000 iteraatiota ja paikallishakua 10000 iteraatiota. Suoritusajat ovat 10 toiston keskiarvoja.

	Toistettu K-means			**Paikallishaku**		
	Perus	Edistynyt	Jäljelle jäi	Perus	Edistynyt	Jäljelle jäi
Bridge	1707	951	56 %	1593	267	17 %
House	9944	3712	37 %	6552	537	8 %
Miss America	4240	2039	48 %	3500	320	9 %
Birch1	46339	28052	61 %	7323	3364	46 %

Taulukosta 5.1 havaittiin eroavaisuuksia karsittujen vertailujen haun suorittamassa alkioiden luokittelussa. Esimerkiksi aineiston *Birch1* tapauksessa lähes 11 prosentissa tapauksista jouduttiin tekemään täyshaku, kun taas aineiston *House* tapauksessa täyshakua tarvittiin vain yhdessä prosentissa tapauksista. Nyt vastaavasti aineiston *Birch1* tapauksessa paikallishaun edistyneen version ajaksi jäi 46 prosenttia, kun aineiston *House* tapauksessa aikaa jäi vain 8 prosenttia alkuperäisestä. Nämä eroavaisuudet alkioiden luokittelussa selittävät taulukon 5.3 erot nopeutumisprosenteissa.

5.3.4. Nopeutus operaatiotasolla

Kuvassa 5.5 on vertailtu paikallishaun perusversion ja edistyneen version operaatioiden käyttämiä aikoja. Nopeutusmenetelmät ovat erityisesti leikanneet *OptimalPartition*-operaation suoritusaikaa reippaasti, mikä oli tarkoituskin. *OptimalPartition*-operaation suhteellinen nopeutus on suurempi aineiston *Bridge* kuin aineiston *S3* tapauksessa. Tämä selittynee ainakin osittain sillä, että PDS-menetelmällä saatava hyöty on suurempi aineistolla *Bridge* sen alkioiden suuremman ulotteisuuden vuoksi.



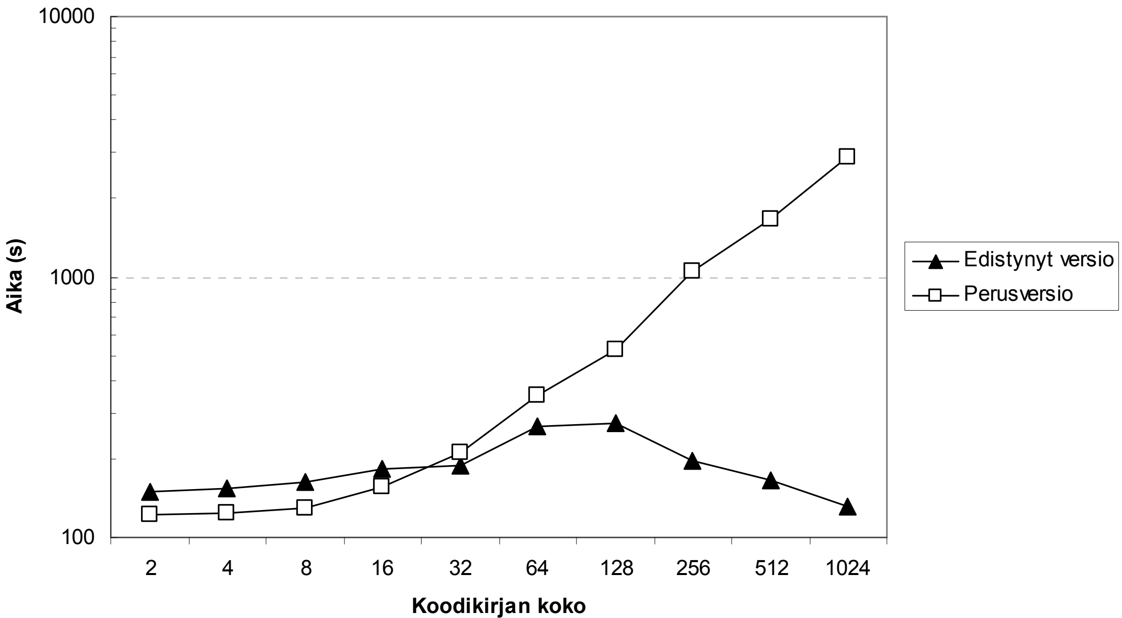
Kuva 5.5: Paikallishaun perusversion ja edistyneen version operaatioiden käyttämät ajat *S3*- ja *Bridge*-aineistolle.

Kuvasta havaitaan *ObjectiveFunction*- ja *LocalRepartition*-operaatioiden ajantarpeen lisääntyneen aineiston *S3* tapauksessa. Operaatioiden hidastumisen syyksi paljastuu etäisyyslaskujen hidastuminen, sillä keskimääräiseksi etäisyyslaskun suorittamiseen kuluva aika on kasvanut 64 nanosekunnista 78 nanosekuntiin. Etäisyyslaskujen lisääntynyt ajantarve selittynee sillä, että etäisyyslaskennassa käytettävä PDS-menetelmä aiheuttaa enimmäkseen lisätyötä kaksiulotteisen aineiston tapauksessa. Aineiston *Bridge* tapauksessa ovat kaikki kolme operaatiota nopeutuneet ja etäisyyslaskennassa tarvittavat aika on vähentynyt 803 nanosekunnista 271 nanosekuntiin. Tämä osoittaa PDS-menetelmällä saavutettavan hyödyn olevan riippuvainen aineiston ulotteisuudesta siten, että suurempiulotteisella aineistolla saadaan suurempi nopeutus. Nopeutuksen jälkeenkin *OptimalPartition*-operaatio yhä dominoi ajankäyttöä molemmilla aineistoilla, joten *LocalRepartition*- ja *ObjectiveFunction*-operaatioiden ajankäytöllä ei vieläkään ole suuremmin vaikutusta suoritusaikaan.

5.3.5. Koodikirjan koon vaikutus

Koodikirjan koon kaksinkertaistamisen pitäisi kaksinkertaistaa myös suoritusaika, sillä paikallishaun aikavaativuus, O(*NMK*), on lineaarisesti riippuva koodikirjan koosta. Karsittujen vertailujen hakua käytettäessä voi kuitenkin käydä niin, että suoritusaika ei välttämättä kasvakaan koodikirjan koon kasvaessa. Näin ollen paikallishaun vaatiman ajan arviointi voi olla vaikeaa.

Kuvassa 5.6 on esitetty paikallishaun perusversion ja edistyneen version keskimääräiset suoritusajat koodikirjan koon funktiona. Havaitaan, että koodikirjan koilla 2–32 perusversion ja edistyneen version suoritusajat ovat lähes samat kuitenkin siten, että perusversio on hieman nopeampi koilla 2–16. Koodikirjan koon 32 jälkeen alkaa tapahtua, sillä perusversion aikavaativuus nousee jyrkästi koilla 64–1024 noudattaen suurinpiirtein kaksinkertaistumissääntöä, mutta edistyneen version suoritusaika saavuttaa huippunsa koodikirjan koolla 64 ja alkaa sen jälkeen laskemaan. Havainnon selittänee sentroidien vähentynyt suhteellinen aktiivivuus koodikirjan koon kasvaessa. Klustereiden määrän ollessa korkea satunnainen korvaus muuttaa nimittäin pienempää osaa ratkaisusta, sillä esimerkiksi poistettavassa klusterissa on todennäköisesti vähemmän alkioita ja lisättävä sentroidi vaikuttanee pienempään määrään alkioita. Johtopäätöksenä tästä kaikesta on, että käytetystä nopeutusmenetelmästä vaikuttaisi olevan sitä enemmän hyötyä, mitä suurempi on klustereiden lukumäärä, ja että pienillä koodikirjan koilla voi perusversio olla edistynyttä versiota nopeampi.



Kuva 5.6: Koodikirjan koon vaikutus paikallishaun suoritusaikaan aineiston *Bridge* tapauksessa. Kuvaajat ovat 5 toiston keskiarvoja.

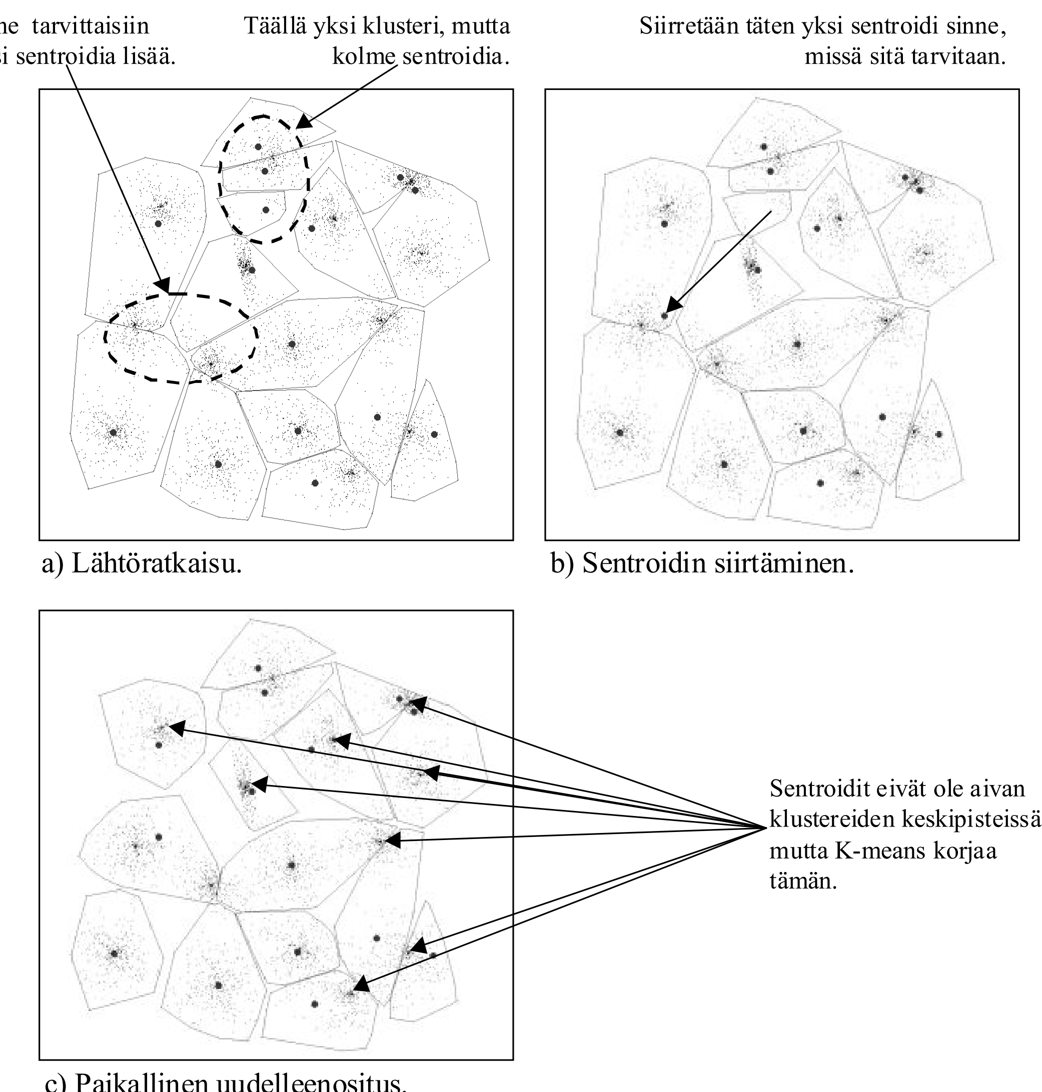
6. Deterministinen sentroidin korvaus

Satunnainen sentroidin korvaus tekee runsaasti turhia korvauksia, jotka eivät johda ratkaisun paranemiseen. Kohdan 4.1 havainnollistuksesta nähtiin paikallishaun testaavan 162 ratkaisuehdokasta oikeiden klustereiden löytämiseksi, mutta näistä vain 7 johti ratkaisun paranemiseen. Samasta satunnaisesta lähtöratkaisusta on kuitenkin mahdollista löytää oikea klusterointi vain kahdella sentroidin vaihdolla, jos sentroidin korvaus tehdään aina sopivasti ja tulos hienosäädetään K-meansilla. Tällöin sentroidia korvatessa pyritään siirtämään ylimääräinen sentroidi sinne, missä sitä tarvitaan. Tätä nimitetään seuraavassa *deterministiseksi korvaukseksi*.

Kohdassa 6.1 esitettävä sentroidin korvaus on keinotekoinen, koska poistettava sentroidi ja lisättävän sentroidin sijainti on päätelty kuvista katsomalla. Kuitenkin esimerkki on siinä mielessä todellinen, että se käyttää uusien ratkaisujen luomiseen aiempien lukujen askeleita, joita ovat sentroidin korvaus, paikallinen uudelleenositus ja K-means. Kohdassa 6.2 tutkitaan, kuinka tämä poistettavan ja lisättävän sentroidin valinta voitaisiin automatisoida.

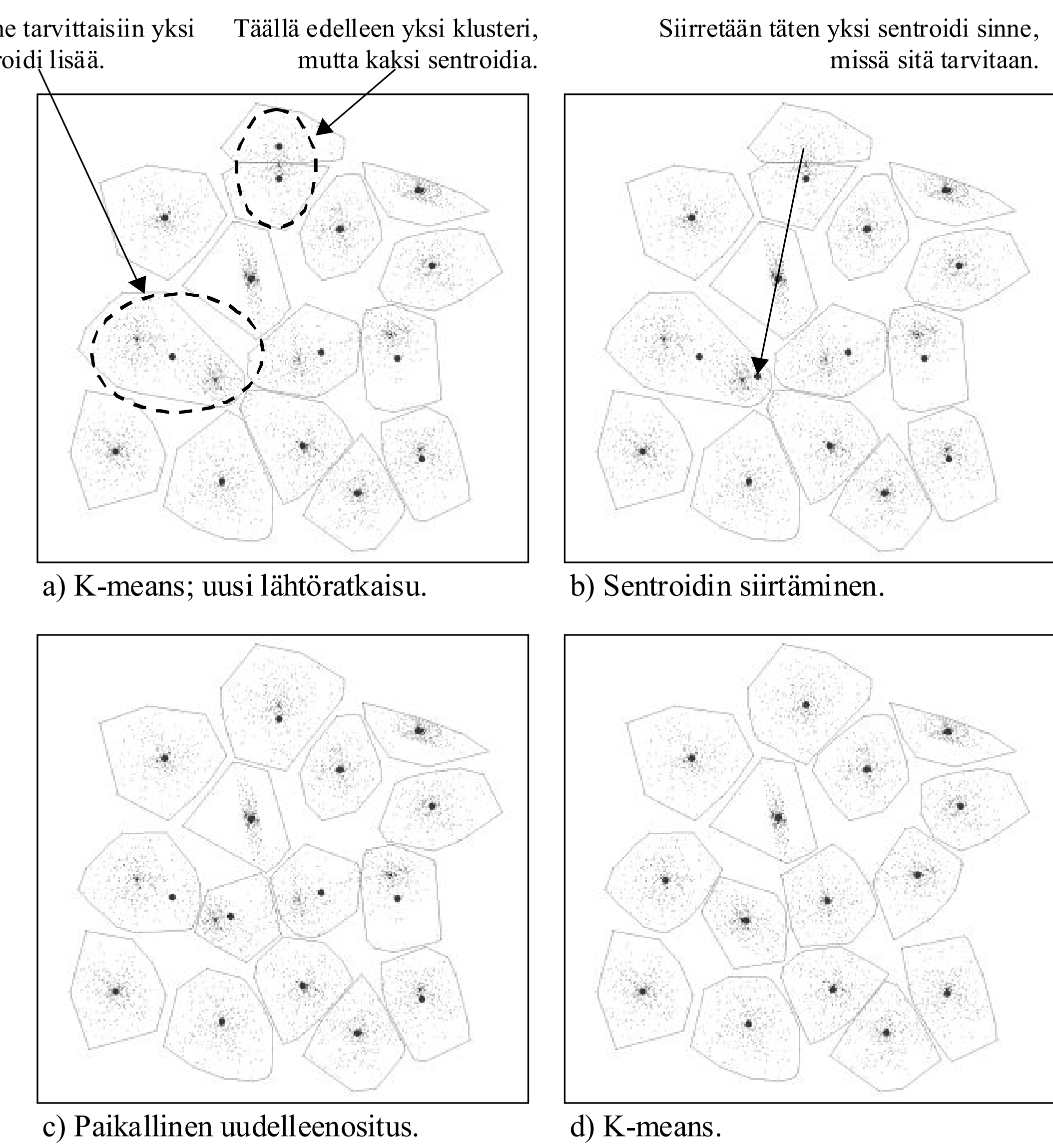
6.1. Determinististen sentroidin korvauksen idea

Kuvan 6.1 lähtöratkaisu on sama kuin satunnaistetun paikallishaun demonstraation lähtöratkaisu kuvassa 4.2. Havaitaan, että lähtöratkaisussa on ylhäällä keskellä kolmen sentroidin keskittymä, mutta tällä alueella on vain yksi klusteri. Toisaalta keskellä vasemmalla on kaksi klusteria, joiden alkiot ovat jakautuneet muihin klustereihin. Näin ollen sentroidin korvaus kannattaa tehdä siten, että siirretään jokin ylhäältä keskeltä olevista sentroideista keskelle vasemmalle, minne on tarvetta perustaa uusi klusteri. Edellä kuvattu sentroidin siirtäminen on esitetty kuvassa 6.1b. Ratkaisu on vielä kuvassa 6.1c esitetyn paikallisen uudelleenosituksen jälkeen hieman keskeneräinen, sillä useat sentroidit ovat kaukana klustereiden keskipisteistä. Kaksi K-means –iteraatiota parantaa ratkaisua huomattavasti, kuten voidaan nähdä kuvasta 6.2a.



Kuva 6.1: Esimerkki sentroidin deterministisestä korvausmenetelmästä.

Havaitaan, että kuvan 6.2a ratkaisussa on ylhäällä keskellä kahden sentroidin keskittymä, mutta tällä alueella on vain yksi klusteri. Toisaalta keskellä vasemmalla on yksi sentroidi, mutta kaksi klusteria. Näin ollen kuvassa 6.2b on esitetty sentroidin siirtäminen ylhäältä keskelle, jossa sitä tarvitaan. Kuvassa 6.2c on paikallisen uudelleenosituksen jälkeen tilanne se, että kaikki klusterit ovat löytyneet, mutta sentroidit eivät ole vielä aivan klustereidensa keskipisteissä. Kaksi K-means –iteraatiota muokkaa ratkaisua siten, että sentroidit löytävät klustereidensa keskipisteet, kuten kuvasta 6.2d nähdään. Saatu ratkaisu on keskineliövirheellä mitattuna yhtä hyvä kuin satunnaisen paikallishaun saavuttama ratkaisu kuvassa 4.3c.



Kuva 6.2: Esimerkki sentroidin korvausmenetelmästä jatkuu.

Taulukossa 6.1 on esitetty, kuinka moni satunnaisista vaihdoista keskimäärin johtaa ratkaisun paranemiseen. Taulukossa esitetyt luvut ovat 10 toiston keskiarvoja. Taulukosta nähdään, että sentroidien korvauksista suurin osa on aivan turhia, sillä ne eivät johda ratkaisun paranemiseen. Huonoimmassa tapauksessa 10000 korvauksesta vain noin 0,079 % johti ratkaisun paranemiseen (*Birch1*) ja parhaimmillaankin 10000 korvauksesta alle 4 % (*Miss America*).

Kuvien 6.1 ja 6.2 esimerkin nojalla voidaan päätellä, että suorittamalla sentroidin korvaus aina sopivasti, olisi mahdollista saavuttaa parannusta jokaisella iteraatiolla. Tarvittavien onnistuneiden korvausten lukumäärä tulisi tällöin ilmeisesti olemaan korkeintaan samaa luokkaa kuin klustereiden lukumäärä, sillä kukin korvaus siirtäisi yhden sentroidin jonkin klusterin läheisyyteen. Taulukosta 6.1 nähtiin, että satunnaistetun paikallishaun käyttämä sentroidin korvaus ei ole erityisen tehokas tapa naapuriratkaisujen muodostamiseen. Näin ollen seuraavaksi keskitymme tutkimaan, miten deterministisen korvauksen voisi toteuttaa algoritmisesti.

Taulukko 6.1: Onnistuneiden sentroidin korvausten lukumäärä iteraatioiden suhteen täysin satunnaisen korvauksen tapauksessa. Luvut ovat 10 toiston keskiarvoja.

	20	50	100	200	500	1000	2000	5000	10000
Bridge	14.1	28.5	47.2	74.7	122.5	162.7	200.7	249.9	284.8
House	16.7	34.5	55.4	81.6	135.2	186.6	248.8	332.8	389.1
Miss America	14.8	31.5	50	78.4	111.3	135.6	158.4	183.5	203.6
Birch1	11.1	17.2	23.8	30.4	36.7	40.2	43	44.7	45.1
Birch2	7.2	12.5	17.3	26	33.8	40.6	43.7	43.7	43.7
Birch3	11.8	20.4	31.1	43.4	53.9	62.1	72.3	85.4	93.9
Data_S1	5.7	7.4	7.8	7.9	7.9	7.9	7.9	7.9	7.9
Data_S2	7.1	9.9	12.2	12.4	13	13.7	14.2	14.3	14.3
Data_S3	5.1	7.6	9.2	10.7	12.2	13.5	14.5	16.1	16.3
Data_S4	6.1	8.3	10.9	12.7	15.3	17.4	19.5	21.5	22.4

6.2. Deterministisen sentroidin korvauksen toteutus

Deterministisessä korvauksessa naapuriratkaisu voitaisiin muodostaa osituksen tasolla käyttämällä hierarkkisten menetelmien tapaan klustereiden jakamista ja yhdistelemistä (Fränti & Kivijärvi, 2000; Kaukoranta & al, 1998). Suorittamalla peräkkäin kahden klusterin yhdistäminen ja yhden klusterin jakaminen kahtia saadaan klustereiden määrä pysymään samana ja siirrettyä yksi klusteri eri paikkaan. Hierarkkisten operaatioiden ollessa hankalahkoja toteuttaa ja hierarkkisten menetelmien vaatiessa klustereiden käsittelemistä kokonaisena, mikä rajoittaa jonkin verran niiden sijoittelua, emme nyt käsittele tarkemmin tätä vaihtoehtoa. Mainittakoon kuitenkin, että Kärkkäinen ja Fränti (2002) ovat käyttämällä klustereiden yhdistelyä ja jakamista naapuriston generointiin rakentaneet version paikallishausta, joka muuttaa haun edetessä klustereiden määrää ja hakee näin klustereiden oikean määrän.

Deterministinen korvaus tarkoittaa toisaalta sitä, että ensin poistamme yhden klusterin ja sen jälkeen luomme uuden klusterin. Klusterin poisto ja luonti tehdään tällöin siirtämällä yhtä sentroidia. Sentroidin siirron jälkeen ositus saadaan satunnaisen version tapaan palautettua optimaaliseksi sentroidien suhteen käyttämällä paikallista uudelleenositusta. Deterministinen korvaus käyttääkin paikallishaun edistyneen version runkoa K-means-iteraatioiden määrän ollessa kaksi. Deterministisen korvauksen toteutus pseudokoodin tasolla on esitetty liitteessä 2. Käsittelemme seuraavaksi vaihtoehtoja poistettavan ja luotavan klusterin valitsemiseksi.

6.2.1. Poistettavan klusterin valinta

Fritzken (1997) esittämässä sentroidin korvauksessa poistettavaksi klusteriksi valitaan se, jonka poisto kasvattaa optimointifunktion arvoa vähiten. Kunkin klusterin poistamisesta aiheutuvan virheen kasvun selvittämiseksi pidämme yllä ensi- ja toissijaista ositusta. *Ensisijaisessa osituksessa* kukin alkio liitetään siihen sentroidiin, joka on kyseistä alkiota lähinnä. Ensisijainen ositus on aiemmin käyttämämme normaali ositus *P*. *Toissijaisessa osituksessa* kukin alkio liitetään vastaavasti toiseksi lähimpään sentroidiin. Toissijaista ositusta merkitään kuvauksena $Q = \{q_1,\ldots,q_N\}$. Kaavassa 6.1 on esitetty toissijaisen osituksen laskeminen. Kaava 6.1 on analoginen kaavan 2.6 kanssa.

$$q_i = \min_{1\le j\le M \wedge j\ne p_i} d(x_i, c_j) \quad \forall\ i \in [1, N] \qquad \textbf{(6.1)}$$

Fritzken (1997) mukaan klusterin poistamisesta aiheutuvan virheen saamme selville tutkimalla, kuinka paljon kyseisen klusterin alkioiden siirtäminen toissijaisiin klustereihin kasvattaisi virhettä.

Kaavassa 6.2 on laskettu klusterin j poistamisesta aiheutuva virhe D_j. Poistettavaksi valitaan klusteri, joka kasvattaa virhettä vähiten. Kutsumme jatkossa tätä poistokriteeriä *Fritzke-poistoksi*.

$$D_j = \sum_{p_i = j} \left(d(x_i, c_{q_i}) - d(x_i, c_j) \right) \quad \forall\ j \in [1, M] \qquad \textbf{(6.2)}$$

Fränti ja Virmajoki (2006) ovat esittäneet sofistikoituneempia menetelmiä klusterin poistamisesta aiheutuvan virheen laskemiseksi. Näissä menetelmissä otetaan huomioon klustereiden väliset kokoerot sekä alkioiden siirtymisen vaikutus sentroidien liikkumiseen. Heidän esittämänsä laskenta perustuu oivallukseen, että alkion siirtäminen toiseen klusteriin tulee aiheuttamaan kohdeklusterin sentroidin liikkumista lisättyä alkiota kohden. Kaavassa 6.3 on esitetty toissijaisen osituksen laskeminen ottaen huomioon sentroidin liikkuminen.

$$q_i = \min_{1 \le j \le M \,\wedge\, j \ne p_i} \frac{|T_j|}{|T_j + 1|} d(x_i, c_j) \quad \forall\ \ i \in [1, N] \qquad \textbf{(6.3)}$$

Toissijaisen osituksen löytämiseksi lasketaan alkion siirtokustannus kuhunkin klusteriin. Kustannus riippuu kohdeklusterin koosta sekä alkion etäisyydestä kohdeklusterin sentroidiin. Etäisyyttä sentroidiin painotetaan sitä enemmän, mitä isompi klusteri on kyseessä. Alkion siirto isoon klusteriin aiheuttaa nimittäin pienemmän muutoksen sentroidin sijaintiin kuin siirto pienempää klusteriin.

Vastaavasti klusterin j poistokustannusta D_j laskettaessa voidaan ottaa huomioon sentroidien liikkuminen. Kaavassa 6.4 on esitetty Fräntin ja Virmajoen (2006) *yksinkertaiseksi laskennaksi* (*simple calculation*) nimeämä poistokustannuksen laskeminen. Laskenta antaa oikean tuloksen, mikäli jokainen alkio siirtyy eri klusteriin. Käytännössä kuitenkin useampia alkioita tulee siirtymään samaan klusteriin, jolloin virhe tulee hieman yliarvioiduksi (Fränti & Virmajoki, 2006). Kutsumme tätä *IS-poistoksi* erona aiemman esitettyyn Fritzken poistokriteeriin.

$$D_j = \sum_{p_i = j} \left(\frac{|T_{q_i}|}{|T_{q_i} + 1|} d(x_i, c_{q_i}) - d(x_i, c_j) \right) \qquad \textbf{(6.4)}$$

6.2.2. Luotavan klusterin valinta

Luotavan klusterin valinta jakautuu kahteen erilliseen tehtävään. Ensin on valittava olemassa oleva klusteri, jonka alueelle lisäys tehdään. Tämän jälkeen täytyy valita sijainti tämän klusterin sisältä. Sijainnin valinta klusterin sisällä ei todennäköisesti tule olemaan kovinkaan kriittistä, koska sen vaikutus on melko paikallinen ja koska sentroidin korvauksen jälkeen suoritettavat paikallinen uudelleenositus ja K-means muokkaavat joka tapauksessa ratkaisua paikallisesti.

Fritzken (1997) mukaisesti uusi klusteri luodaan sen klusterin läheisyyteen, joka aiheuttaa eniten virhettä. Kaavassa 6.5 on esitetty klusterin j aiheuttaman virheen E_j laskeminen. Fritzke (1997) lisää uuden sentroidin valitun klusterin sentroidin välittömään läheisyyteen, mikä aiheuttaa valitun klusterin jakautumisen kahteen osaan. Toinen vaihtoehto on lisätä uusi sentroidi klusterin kauimman alkion paikalle, jolloin uusi klusteri todennäköisesti kerää alkioita myös viereisistä klustereista.

$$E_j = \sum_{p_i = j} d(x_i, c_j) \qquad \textbf{(6.5)}$$

6.2.3. LBG-U

LBG-U on Fritzken (1997) esittämä lähtöratkaisun luontia lukuunottamatta täysin deterministinen paikallishaku. Menetelmä lähtee liikkeelle satunnaisesta lähtöratkaisusta, jolle suoritetaan K-meansia, kunnes ratkaisu ei enää parane. Ratkaisusta poistetaan se klusteri, jonka poisto kaavan 6.2 mukaan kasvattaa optimointifunktion arvoa vähiten, ja poistetun klusterin sentroidi siirretään sen klusterin sentroidin läheisyyteen, joka kaavan 6.5 mukaan aiheuttaa eniten virhettä. Muokatulle ratkaisulle suoritetaan K-meansia, kunnes ratkaisu ei enää parane. Näin saatu ratkaisu hyväksytään, mikäli se oli aiempaa parempi. Tätä toistetaan niin kauan kuin ratkaisu paranee.

LBG-U eroaa esittämästämme täysin deterministisestä paikallishausta K-means –iteraatioiden määrässä. Paikallishaussamme ei nimittäin lähtöratkaisulle suoriteta K-meansia eikä sentroidin korvauksella muodostetuille ratkaisuille K-meansia suoriteta loppuun asti.

6.2.4. Sentroidin korvauksen aikavaativuus

Taulukossa 6.2 on esitetty yhteenvetona eri paikallishakuversioiden poiston ja lisäyksen aikavaativuudet. Taulukon aikavaativuudet on poimittu liitteiden 1 ja 2 aikavaativuusanalyyseista. Havaitaan deterministisen poiston olevan aikavaativuudeltaan samaa luokkaa *OptimalPartition*-operaation kanssa ja deterministisen lisäyksen samaa luokkaa *LocalRepartition*- ja *ObjectiveFunction*-operaatioiden kanssa. Aiempien profilointien nojalla odotettavissa on siis, että deterministinen poisto tulee viemään reilusti aikaa ja deterministinen lisäys tulee viemään vain vähän aikaa.

Taulukko 6.2: Paikallishaun eri versioiden poiston ja lisäyksen aikavaativuudet.

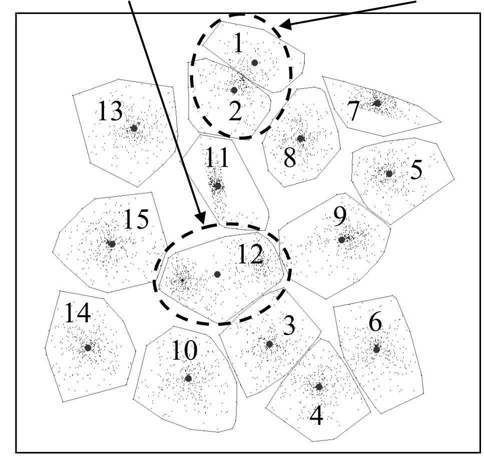
	Poisto	Lisäys
Satunnainen poisto ja lisäys	$O(1)$	$O(MK)$
Deterministinen poisto ja satunnainen lisäys	$O(NMK)$	$O(MK)$
Satunnainen poistoja deterministinen lisäys	$O(1)$	$O(NK)$
Deterministinen poisto ja lisäys	$O(NMK)$	$O(NK)$
LBG-U	$O(NMK)$	$O(NK)$

6.3. Deterministisen sentroidin korvauksen havainnollistaminen

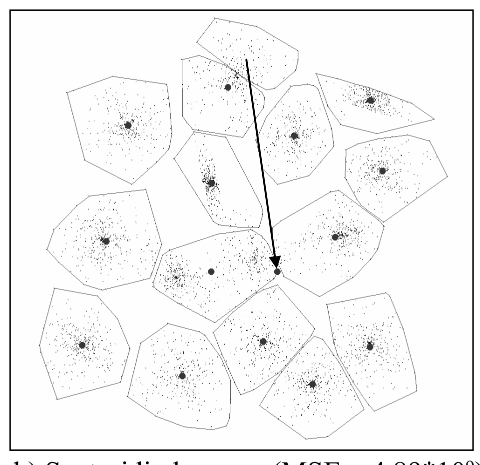
Kuvassa 6.3 on annettu esimerkki edellä esitetystä deterministisestä sentroidin korvauksesta, jossa käytetään IS-poistoa ja klusterin kauimpaan alkioon lisäystä. Klusteroitavana aineistona on esimerkissä käytetty *S2* aineistoa. Havaitaan, että kuvan 6.3 lähtöratkaisussa on ylhäällä keskellä kahden sentroidin keskittymä, mutta tällä alueella vain yksi klusteri, ja että toisaalta keskellä olisi tarvetta yhdelle sentroidille. Yksi sentroidi tulisi täten siirtää klusterista 1 tai 2 klusterin 12 lähistölle.

Taulukossa 6.3 on laskettu kullekin klusterille j kaavan 6.4 mukainen virhe ja kaavan 6.5 mukainen poistokustannus. Kuvan 6.3a klustereiden numerot vastaavat taulukon 6.3 numerointia. Deterministisessä korvauksessa poistettavaksi klusteriksi valitaan se, jonka poistokustannus on pienin, ja uusi sentroidi lisätään eniten virhettä aiheuttavan klusterin kauimpaan alkioon. Näin ollen klusterin 1 sentroidi siirretään klusterin 12 kauimman alkion paikalle, kuten nähdään kuvasta 6.3b.

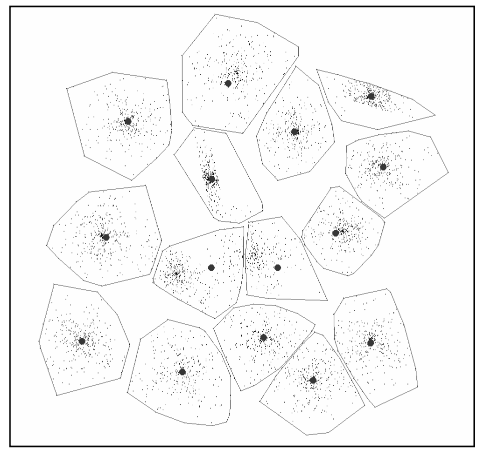
Paikallisen uudelleenosituksen jälkeen (kuva 6.3c) lisätty sentroidi on halkaissut eniten virhettä aiheuttaneen klusterin ja poistetun klusterin alkiot ovat tulleet lisätyksi naapuriklustereihin. Ratkaisu näyttää hyvältä paitsi, että halkaistun klusterin sentroidit eivät ole kohdallaan. Muutama K-means –iteraatio (kuva 6.3d) kuitenkin siirtää sentroidit kohdalleen korjaten hieman muitakin klustereita.



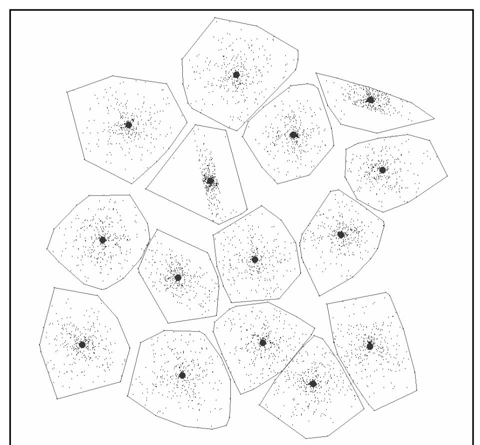
a) Lähtöratkaisu (MSE ≈ $1{,}73 \cdot 10^9$).



b) Sentroidin korvaus (MSE ≈ $4{,}89 \cdot 10^9$).



c) Paikallinen uudelleenositus (MSE ≈ $1{,}59 \cdot 10^9$).



d) K-means (MSE ≈ $1{,}33 \cdot 10^9$).

Kuva 6.3: Deterministinen korvaus.

Taulukko 6.3: Kuvan 6.3a klustereiden aiheuttamat virheet sekä poistokustannukset.

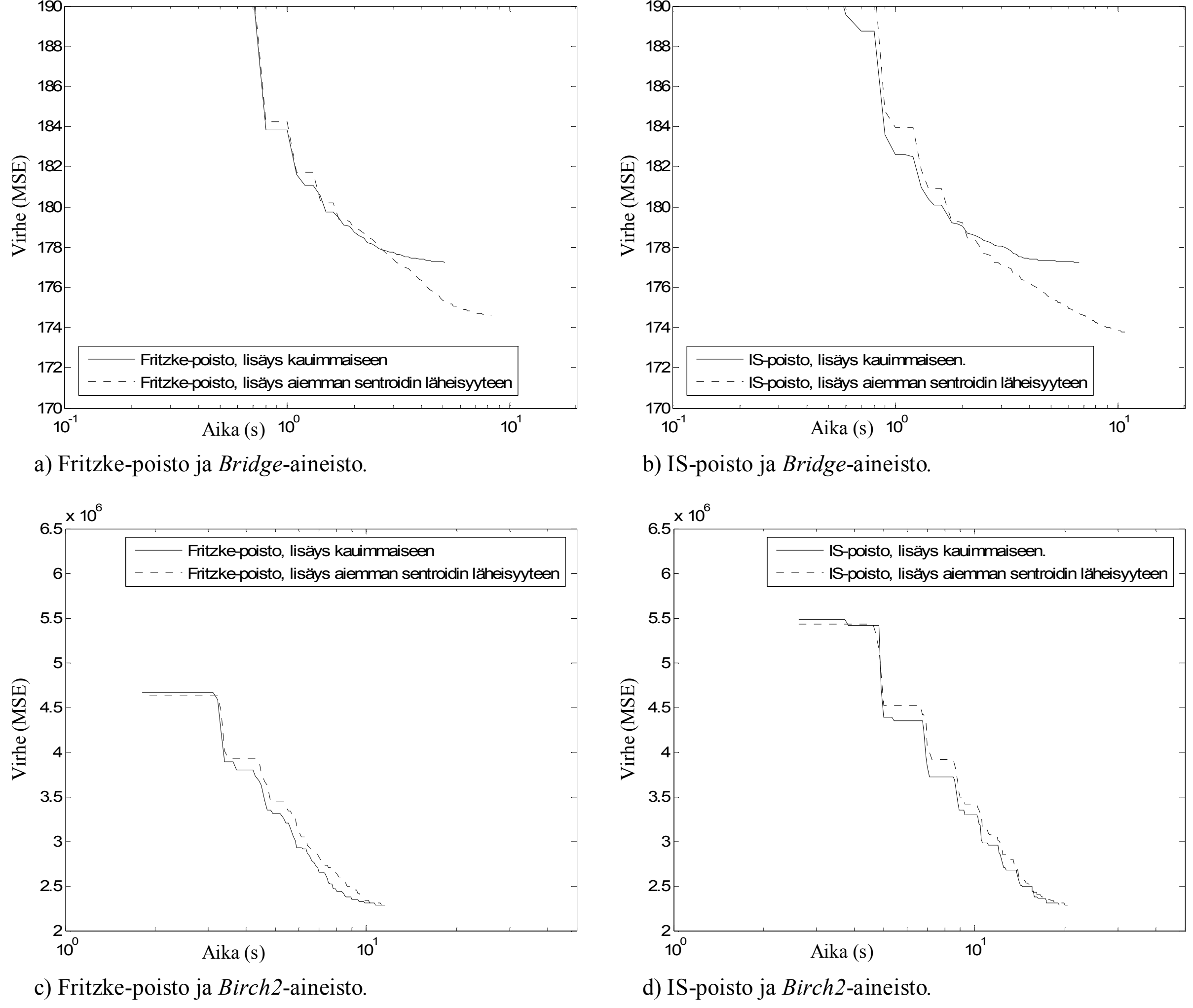
	Poistokustannus (D_j)	**Klusterin aiheuttama virhe (E_j)**
1	**0.80**	0.39
2	1.04	0.64
3	5.48	1.09
4	5.66	0.92
5	6.50	0.76
6	7.67	1.01
7	8.47	0.45
8	9.10	0.75
9	9.90	1.42
10	11.09	1.26
11	11.47	0.61
12	12.17	**4.70**
13	14.61	0.94
14	16.41	0.93
15	16.68	1.41

6.4. Deterministisen sentroidin korvauksen analysointia

Sentroidin korvauksessa on kaksi erillistä osaa: poistettavan klusterin valinta ja luotavan klusterin sijainnin valinta. Deterministisen sentroidin korvauksen toteuttamiseksi on täten kolme eri versiota, sillä kumpikin näistä osista voi olla joko deterministinen tai satunnainen. Lisäksi sekä poistolle että lisäykselle on kohdassa 6.2 esitetty eri toteutusvaihtoehtoja. Tutkimme seuraavaksi eri toteutusvaihtoehtojen tehokkuutta täysin deterministiselle versiolle, jossa siis sekä poisto että lisäys ovat deterministisiä.

6.4.1. Lisäys- ja poistomenetelmän vaikutus

Kuvasta 6.4 nähdään, että erot lisäysmenetelmien välillä ovat melko pieniä. Esimerkiksi Fritzke-poiston tapauksessa *Bridge* aineistolla (kuva 6.4a) on yhden sekunnin kohdalla virhe 183,93 käytettäessä lisäystä kauimmaiseen ja 184,14 käytettäessä lisäystä aiemman sentroidin läheisyyteen. Kuitenkin jo muutaman sekunnin kuluttua paremmuusjärjestys kääntyy päinvastoin. Aiemman sentroidin läheisyyteen lisäys tuottaa paremman ratkaisun *Bridge* aineistolla (kuvat 6.4a ja 6.4b), joten käytämme jatkossa lisäystä aiemman sentroidin läheisyyteen. Ennakko-odotusten vastaisesti luotavan klusterin sijainnin valinnalla näyttäisi olevan hieman merkitystä tehokkuuteen ja saatavan ratkaisun laatuun.



a) Fritzke-poisto ja *Bridge*-aineisto.

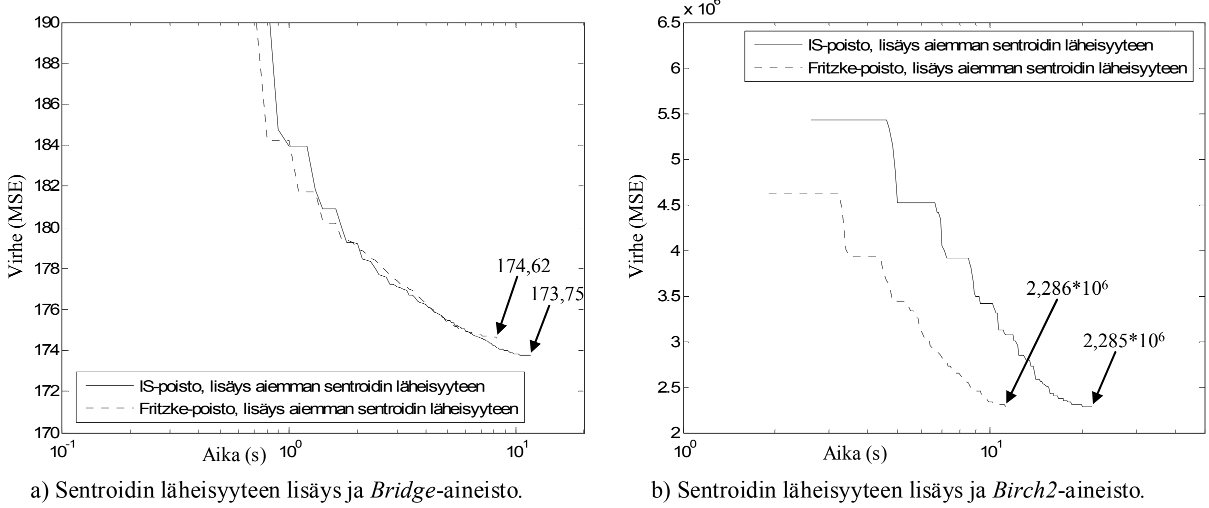
b) IS-poisto ja *Bridge*-aineisto.

c) Fritzke-poisto ja *Birch2*-aineisto.

d) IS-poisto ja *Birch2*-aineisto.

Kuva 6.4: Lisäysmenetelmän vaikutus paikallishaun tehokkuuteen aineistojen *Bridge* ja *Birch2* tapauksessa. Kuvaajat ovat 10 ajon keskiarvoja.

Poistomenetelmät ovat aineiston *Bridge* tapauksessa (kuva 6.5a) erittäin tasavertaisia tehokkuudessa, mutta IS-poisto kuitenkin tuottaa hieman paremman ratkaisun. Sen sijaan aineistolla *Birch2* (kuva 6.5b) Fritzke-poisto on IS-poistoa tehokkaampi. Esimerkiksi kolmen sekunnin suorituksen jälkeen virhe on $5{,}44 \cdot 10^6$ IS-poiston tapauksessa ja vastaavasti $3{,}86 \cdot 10^6$ Fritzke-poiston tapauksessa. Molemmat menetelmät tosin tuottavat loppujen lopuksi yhtä hyvät ratkaisut. Käytämme täten jatkossa IS-poistoa, sillä se vaikuttaisi tuottavan paremman ratkaisun ei-klusteroituneelle *Bridge* aineistolle (kuva 6.5a). Ratkaisun laatua on tarkoituksenmukaista pitää valintakriteerinä hieman paremman tehokkuuden sijaan, sillä täysin deterministinen paikallishaku etenee vain niin kauan kuin löytää parannusta. Tässä kohdassa valittuja menetelmiä tullaan jatkossa käyttämään kaikissa paikallishaun deterministisissä versioissa.



a) Sentroidin läheisyyteen lisäys ja *Bridge*-aineisto.

b) Sentroidin läheisyyteen lisäys ja *Birch2*-aineisto.

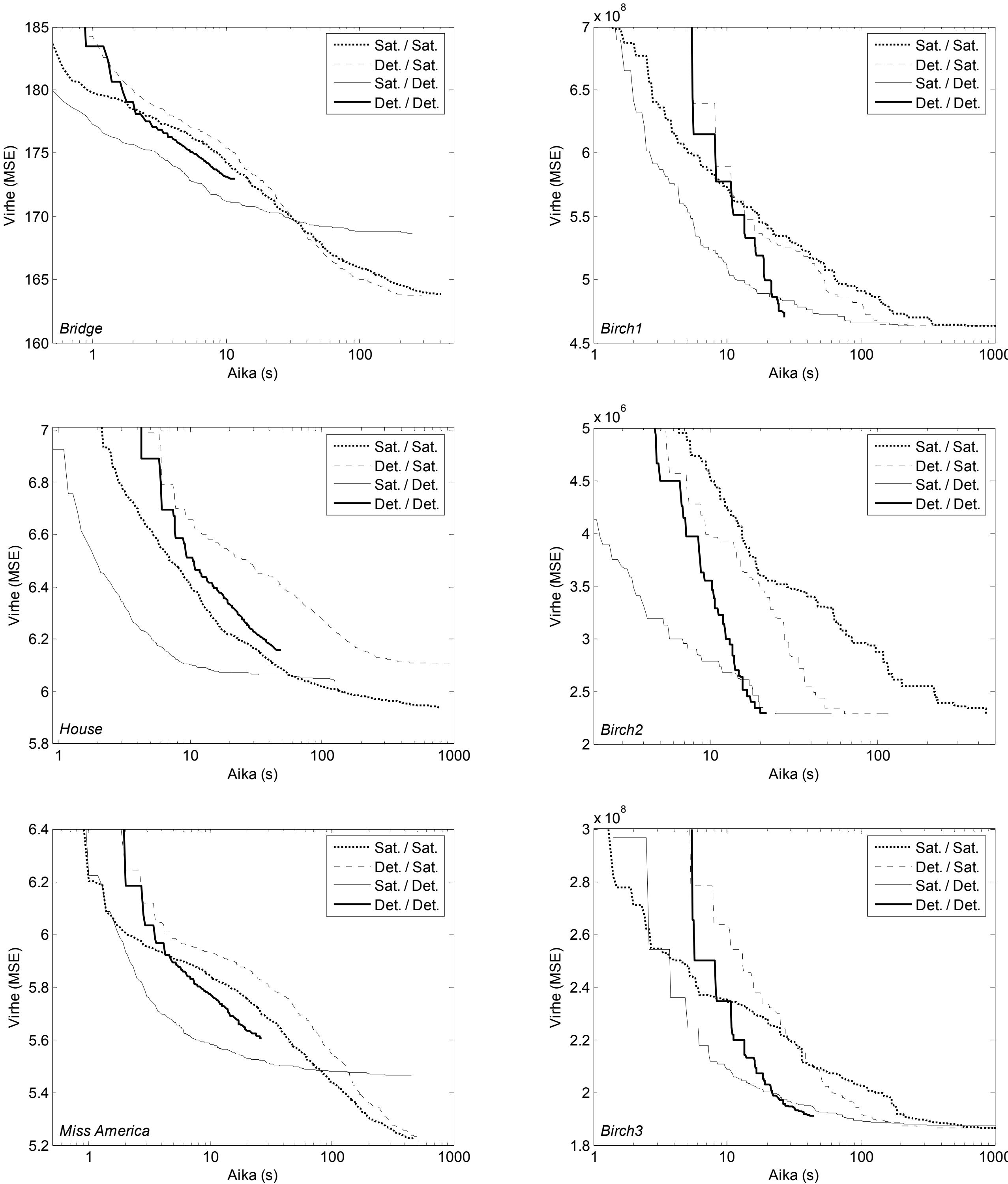
Kuva 6.5: Poistomenetelmän vaikutus paikallishaun tehokkuuteen aineistojen *Bridge* ja *Birch2* tapauksessa. Kuvaajat ovat 10 ajon keskiarvoja.

6.4.2. Determinististen versioiden vertailua kuva- ja Birch-aineistoilla

Kuvasta 6.6 havaitaan paikallishaun satunnaisen ja determinististen versioiden keskinäisen paremmuuden riippuvan klusteroitavasta aineistosta. Vaikuttaa siltä, että paikallishaun deterministiset versiot toimivat parhaiten *Birch1*, *Birch2* ja *Birch3* tyyppisillä aineistoilla, joissa on selkeitä klustereita. Ei-klusteroituneilla aineistoilla, joita *Bridge*, *House* ja *Miss America* edustavat, deterministiset versiot eivät pärjää paikallishaun satunnaiselle versiolle. Tämä siitäkin huolimatta, vaikka valitsimme aiemmin poistomenetelmäksi IS-poiston, joka toimi Fritzke-poistoa paremmin ei-klusteroituneella aineistolla. Huomattavaa on myös, että *Birch1*, *Birch2* ja *Birch3* aineistoilla kaikki versiot pääsevät samaan ilmeisesti optimaaliseen lopputulokseen. Muilla kuvan 6.6 aineistoilla sen sijaan saatavan ratkaisun laadussa on eroa versioiden välillä.

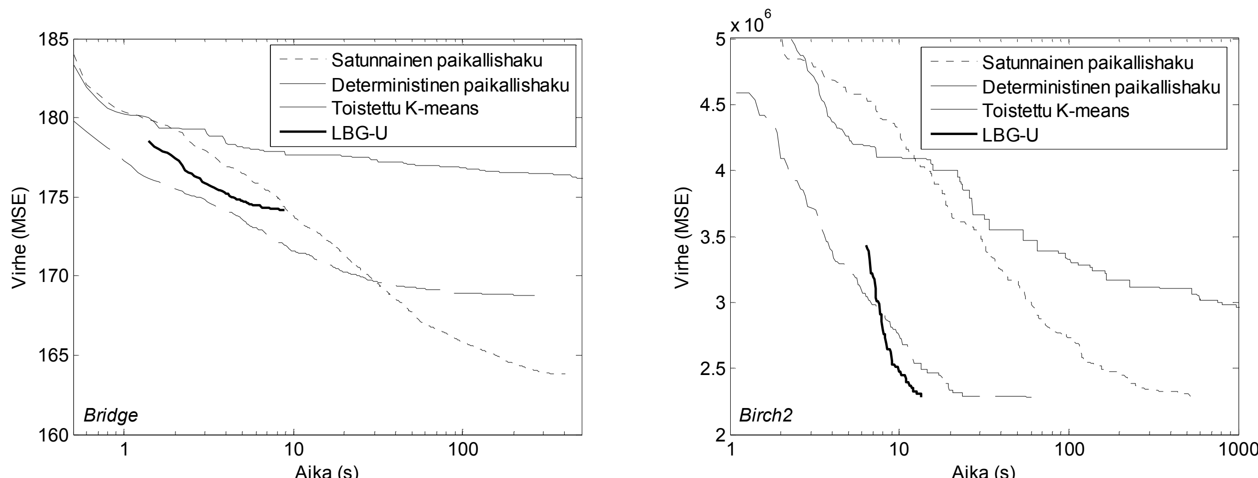
Parhaan paikallishakuversion valinta riippuu siis klusteroitavasta aineistosta. Täysin deterministinen versio löytää nopeimmin optimaalisen ratkaisun *Birch*-aineistoilla, mutta häviää muilla kuvan 6.6 aineistoilla muille menetelmille ratkaisun laadussa. Täysin satunnainen sekä deterministinen poistoa käyttävä versiot ovat suorituskyvyltään melkoisen samankaltaisia, varsinkin aineistojen *Bridge* ja *Miss America* tapauksessa. Satunnaista poistoa ja determinististä lisäystä käyttävä versio sen sijaan tarjoaa kompromissin täysin satunnaisen ja täysin deterministisen versioiden väliltä. Kyseisen version tehokkuus *Birch*-aineistoilla on nimittäin lähes täysin deterministisen version luokkaa. Lisäksi muilla kuvan 6.6 aineistoilla se on huomattavasti täysin determinististä versiota tehokkaampi ollen

aluksi tehokkain kaikista versioista. Satunnaista poistoa ja determinististä lisäystä käyttävä versio on tämän vuoksi valittu edustamaan deterministisiä versioita.



Kuva 6.6: Paikallishaun versioiden tehokkuuden vertailua eri aineistoilla. Paikallishaun versio on ilmoitettu muodossa poisto/lisäys. Kuvaajat ovat 10 ajon keskiarvoja.

Kuvassa 6.7 on vertailtu paikallishaun satunnaista ja determinististä versiota sekä LBG-U että toistettu K-means –menetelmiin. Voidaan havaita, että toistettu K-means voittaa tehokkuudessa paikallishaun satunnaisen version vain aluksi. LBG-U-menetelmälle ja paikallishaun deterministiselle versiolle toistettu K-means ei pärjää tehokkuudessa. Lisäksi havaitaan, että aineistolla *Bridge* on paikallishaun deterministinen versio aluksi tehokkain, mutta noin 30 sekunnin jälkeen satunnainen versio voittaa sen tehokkuudessa. Täysin determinististen paikallishakumme tapaan LBG-U tuottaa aineistolla *Bridge* laadultaan huonomman ratkaisun kuin nyt testatut paikallishaun versiot. Aineiston *Birch2* tapauksessa deterministinen paikallishaku ja LBG-U päihittävät deterministisinä menetelminä tehokkuudessa satunnaisen paikallishaun. Täysin determinististen paikallishakumme tapaan LBG-U voittaa *Birch2* aineistolla tehokkuudessa nyt testatut paikallishaun versiot.

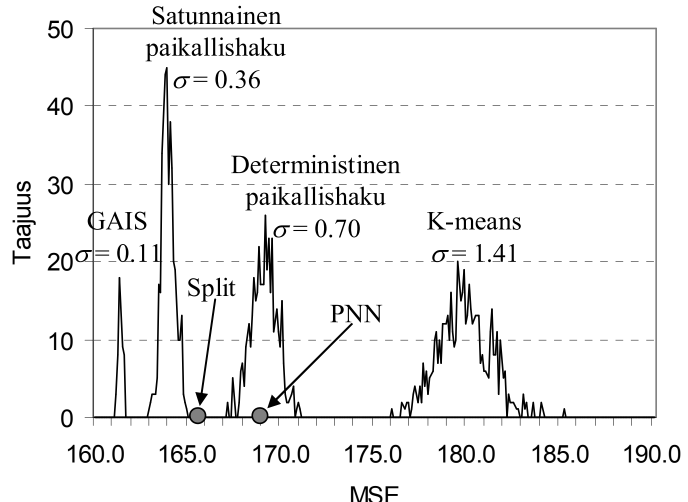


Kuva 6.7: Paikallishaun vertailua toistetun K-meansin edistyneeseen versioon sekä LBG-U-menetelmään. Kuvaajat ovat 10 ajon keskiarvoja.

Kuvassa 6.8 on havainnollistettu eri klusterointimenetelmien tuottamien ratkaisujen virheiden jakautumista. Virheiden jakaumia on tarkasteltu, sillä mahdollisimman pienen keskimääräisen virheen lisäksi on toivottavaa, että virheen hajonta olisi pientä. Satunnaista ja determinististä paikallishakua on verrattu GAIS-, K-means-, Split- ja PNN-menetelmiin. Geneettinen paikallishaku (GAIS; Fränti & Virmajoki, 2006) on otettu mukaan vertailuun, sillä se on, ainakin Virmajoen (2004) mukaan, paras klusterointimenetelmä, kun kriteerinä käytetään ratkaisun laatua. K-means on laajalti käytetty menetelmä ja ollut mukana jo aiemmissakin testeissämme. Hierarkkisista menetelmistä vertailussa on mukana yksi jakava (Split; Fränti & al., 1997a) ja yksi yhdistävä (PNN; Virmajoki, 2004) menetelmä.

Kuvasta 6.8 nähdään, että geneettinen paikallishaku on testatuista menetelmistä paras tuottaen parhaan ratkaisun pienimmällä hajonnalla. Lisäksi havaitaan, ettei täysin satunnainen versio ainoastaan saavuta parempaa ratkaisua kuin deterministinen versio, mutta sen antamien ratkaisujen hajonta on myös pienempää. K-means tuottaa muihin menetelmiin nähden verrattaen kehnon ratkaisun ja vieläpä isoimmalla hajonnalla. Tässä tutkielmassa käytetyllä K-meansin toistetulla versiolla on toki yksittäiseen K-means –ajoon verrattuna pienempi hajonta, mutta ratkaisun laatu jää silti silläkin heikoksi. Hierarkkiset menetelmät sijoittuvat ratkaisun laadussa paikallishakujen välimaastoon. Testattujen hierarkkisten menetelmien ollessa täysin deterministisiä ei niiden ratkaisujen virheissä ole hajontaa.

Satunnainen paikallishaku näyttäisi kaiken kaikkiaan tuottavan kohtuullisen hyviä ratkaisuja kohtuullisen pienellä hajonnalla. Tässä vertailussa ainoastaan geneettinen paikallishaku oli satunnaista paikallishakua parempi. On kuitenkin otettava huomioon, että geneettinen paikallishaku on huomattavasti paikallishaun satunnaista versiota hankalampi toteuttaa.



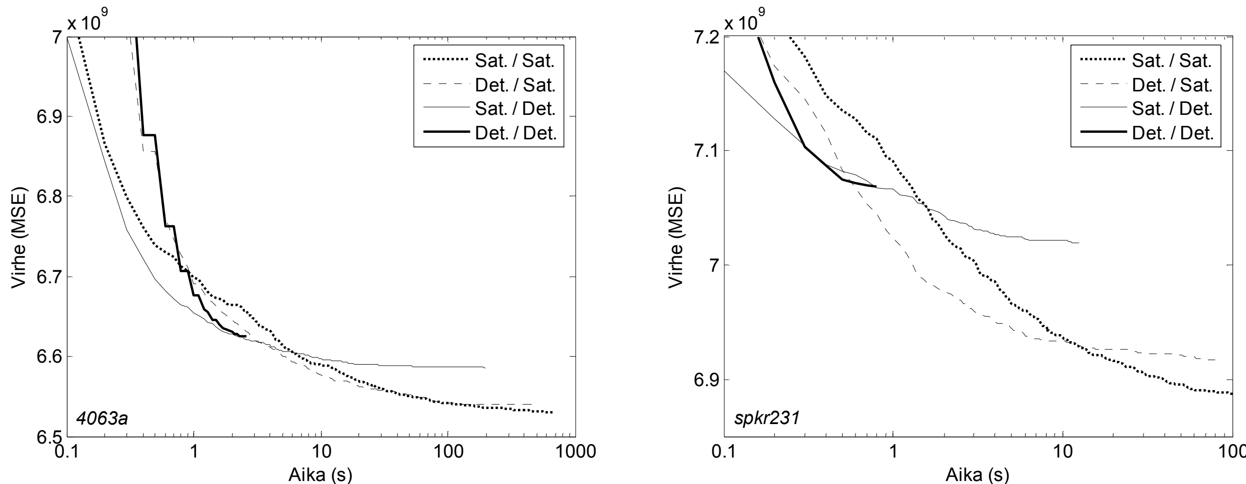
Kuva 6.8: MSE-arvojen jakautuminen *Bridge* aineistolla. GAIS menetelmää on ajettu 50 kertaa, paikallishakuja 300 kertaa ja k-meansia 500 kertaa. Menetelmiä Split ja PNN ei ole toistettu, sillä ne ovat täysin deterministisiä.

6.4.3. Determinististen versioiden vertailua puheaineistoilla

Paikallishaun satunnaisen version on havaittu toimivan deterministisiä versioita paremmin kuva-aineistoilla, joissa ei ole havaittavissa selkeitä klustereita. Kinnusen (2005) sekä Kinnusen ja muiden (2001) mukaan puheaineistoissa ei ole myöskään havaittavissa klustereita, joten seuraavassa on testattu, yleistyvätkö kuva-aineistoille tehdyt havainnot myös puheaineistoille. Testit on ajettu yhdelle puhujalle *NIST-1999*-tietokannasta (puhuja *4063a*) ja vastaavasti yhdelle puhujalle *TIMIT*-tietokannasta (puhuja *spkr231*).

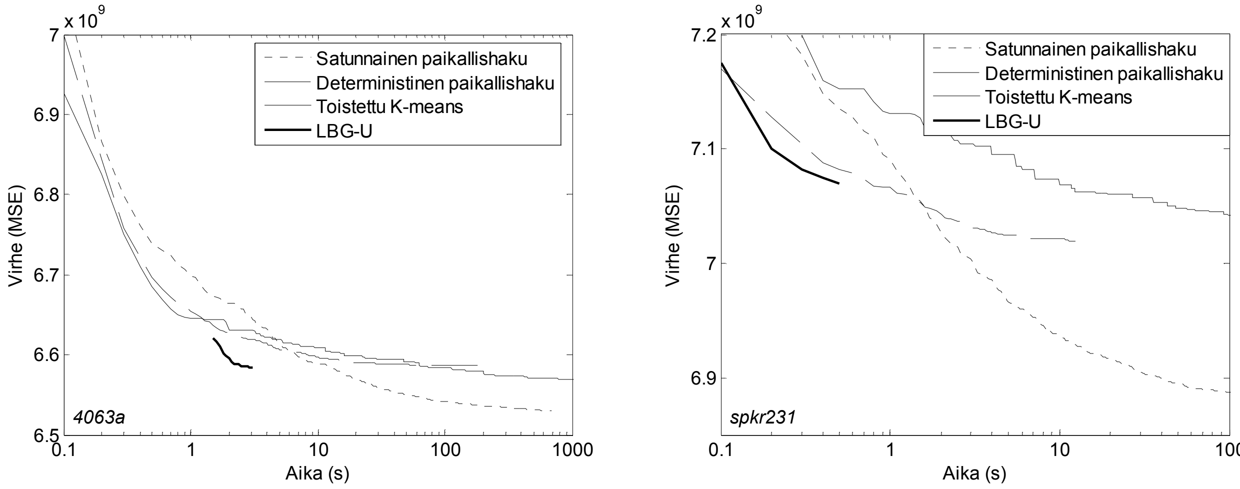
Kuvasta 6.9 nähdään, että tehdyt havainnot paikallishakuversioiden keskinäisestä paremmuudesta yleistyvät myös puheaineistoille. Nimittäin aluksi täysin deterministinen sekä determinististä lisäystä käyttävät versiot ovat tehokkaimmat. Nämä versiot eivät kuitenkaan pärjää muille paikallishaun versioille, sille ne pystyvät parantamaan ratkaisua vain tiettyyn pisteeseen asti. Täysin satunnainen sekä deterministinen poistoa käyttävät versiot ovat suorituskyvyltään melkoisen samankaltaisia, varsinkin aineiston *4063a* tapauksessa. Täysin satunnainen versio saavuttaa laadultaan parhaan ratkaisun, kuten myös kuvan 6.6 kuva-aineistoilla suoritetusta testauksesta havaittiin.

Havaitaan, että erot paikallishakuversioiden välillä ovat suuremmat *TIMIT*-tietokannasta olevan aineiston *spkr231* kuin *NIST-1999*-tietokannasta olevan aineiston *4063a* tapauksessa. Kinnusen ja muiden (2004) tulosten mukaan *TIMIT*-tietokannan materiaali on *NIST-1999*-tietokannan materiaalia helpompaa puhujantunnistuksessa. Tämä voi selittää eroavaisuudet aineistojen välillä.



Kuva 6.9: Paikallishaun versioiden tehokkuuden vertailua puheaineistoilla. Paikallishaun versio on ilmoitettu muodossa poisto/lisäys. Kuvaajat ovat 10 ajon keskiarvoja.

Kuvassa 6.10 on vertailtu puheaineistoilla paikallishaun satunnaista ja determinististä versiota sekä LBG-U että toistettu K-means –menetelmiin. Joitakin eroavaisuuksia verrattuna kuvan 6.7 *Bridge* kuva-aineistolla tehtyyn vastaavaan vertailuun on havaittavissa. Nimittäin aiemmissa vertailuissa paikallishaku on voittanut tehokkuudessa toistetun K-meansin, mutta nyt aineistolla *4063a* toistettu K-means on tasoissa deterministisen paikallishaun kanssa. Lisäksi LBG-U pärjää determinististä paikallishakua paremmin. LBG-U-menetelmä on olennaisesti sama kuin täysin deterministinen paikallishakumme paitsi, että siinä K-meansia iteroidaan niin kauan kuin saadaan parannusta. Näin ollen voisi olla ehkä aiheellista tutkia suoritettavien K-means-iteraatioiden määrän vaikutusta paikallishakumme determinististen versioiden tehokkuuteen.



Kuva 6.10: Paikallishaun vertailua puheaineistojen tapauksessa toistetun K-meansin edistyneeseen versioon sekä LBG-U-menetelmään. Kuvaajat ovat 10 ajon keskiarvoja.

Satunnainen paikallishaku vaikuttaisi olevan hyvä valinta klusterointimenetelmäksi myös puheaineistoilla, mikäli halutaan minimoida puhujamallin virhe suhteessa *opetusaineistoon*. Puhujamallien luontiin käytettävän klusterointimenetelmän valintaa ovat tutkineet tarkemmin Kinnunen ja muut (2000) sekä Kinnunen ja muut (2006). Kinnusen ja muiden (2006) mukaan klusterointimenetelmän valinta ei ole kovin kriittistä, sillä mikä tahansa järkevä klusterointimenetelmä riittää.

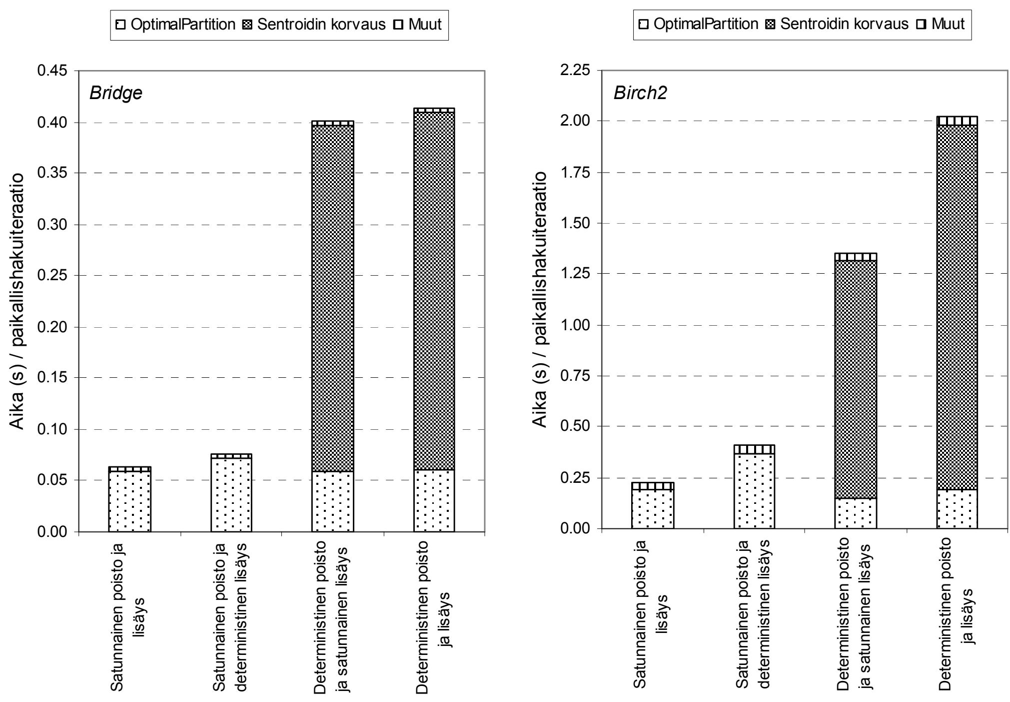
He kuitenkin suosittelevat käyttämään satunnaista paikallishakua sen yksinkertaisen rakenteen ja sen saavuttamien hyvien tulosten vuoksi.

6.4.4. Determinististen versioiden suorituskyky

Paikallishaun tehokkuus riippuu sekä tarvittavien iteraatioiden määrästä että yhteen iteraatioon kuluvasta ajasta. Toivottavaa siis olisi, että yhden iteraation suorittamiseen kuluisi mahdollisimman vähän aikaa ja että kullakin iteraatiolla saavutettaisiin mahdollisimman paljon parannusta, jolloin tarvittavien iteraatioiden määrä olisi pieni. Seuraavaksi tutkitaan näitä kahta tehokkuustekijää erikseen.

Kuvassa 6.11 on esitetty paikallishaun kunkin version keskimääräinen ajankäyttö yhdellä paikallishakuiteraatiolla. Kuvasta nähdään täysin satunnaisen version vievän vähiten aikaa iteraatiota kohden ja täysin deterministisen version eniten. Havaitaan lisäksi deterministisen poiston vievän runsaasti aikaa ja deterministisen lisäyksen vievän vain hyvin vähän. Täysin deterministisen sekä determinististä poistoa ja satunnaista lisäystä käyttävien versioiden sentroidin korvaukseen käyttämät ajat ovat nimittäin aineiston *Bridge* tapauksessa suunnilleen samat. Kohdassa 6.2.4. esitettyjen aikavaativuuksien nojalla nämä havainnot ovat ymmärrettäviä, sillä deterministisen poiston aikavaativuus on $O(NMK)$ mutta lisäyksen aikavaativuus on vain $O(NK)$.

Aineiston *Birch2* tapauksessa havaitaan huomattava ero täysin deterministisen sekä determinististä poistoa ja satunnaista lisäystä käyttävien versioiden välillä sentroidin korvaukseen kuluneessa ajassa. Ero *Bridge* aineistolla tehtyihin havaintoihin selittyy sillä, että determinististä poistoa ja satunnaista lisäystä käyttävä versio ei saavuta parannusta aivan jokaisella iteraatiolla toisin kuin täysin deterministinen versio. Mikäli tarkasteltaisiin ainoastaan niitä iteraatioita, joilla kumpikin versio löysi parannusta, olisivat tulokset *Bridge* aineistolla havaitun kaltaisia.



Kuva 6.11: Paikallishaun eri versioiden ajankäyttö yhtä iteraatiota kohden. Luvut ovat 10 paikallishakuiteraation 10 toiston keskimääräisiä aikoja.

Kuvasta 6.11 havaitaan deterministisen lisäyksen lisäävän optimaalisen osituksen muodostavan *OptimalPartition* operaation vaatimaa aikaa eli toisin sanoen klustereiden aktiivisuutta. Esimerkiksi *Birch2* aineistolla täysin satunnainen versio käyttää osituksen muodostamiseen 0,19 sekuntia ja determinististä lisäystä käyttävä versio vastaavasti 0,37 sekuntia. Deterministinen poisto vaikuttaisi puolestaan säilyttävän samana tai hieman nopeuttavan optimaalisen osituksen muodostamista. Havainnot voitaneen selittää siten, että poistettavaksi valitaan klusteri, jonka poistokustannus on pienin. Tällöin poistettavaksi tyypillisesti valittaneen pieni ja väärässä paikassa oleva klusteri, jonka poisto aiheuttaa vain harvojen klustereiden aktiivisuuden. Klusterin luonti tehdään sen klusterin alueelle, joka aiheuttaa suurimman virheen. Tilanne tällöin lienee kuten kuvan 6.3 tapauksessa, että luotavaan klusteriin tulee alkioita useista eri klustereista, jolloin luonti aiheuttaa useiden klustereiden aktiivisuutta.

Paikallishaun onnistuneiden korvausten lukumäärät on listattu taulukossa 6.4. Aineiston *Bridge* tapauksessa havaitaan determinististen versioiden aluksi onnistuvan parantamaan ratkaisua täysin satunnaista versiota useammin. Tämä sopii yhteen aiempien havaintojen kanssa, sillä determinististen versioiden havaittiin aineistolla *Bridge* käyttävän enemmän aikaa iteraatiota kohden ja silti pääsevän aluksi parempiin tuloksiin. Näin ollen determinististen versioiden on alussa löydettävä täysin satunnaista versiota useammin tai enemmän parannusta. Täysin satunnainen versio alkaa löytää deterministisiä versioita useammin parannusta suoritettaessa enemmän iteraatioita.

Taulukko 6.4: Onnistuneiden sentroidin korvausten lukumäärät iteraatioiden suhteen, kun klusteroitavana aineistona on *Bridge*. Paikallishaun versio on ilmoitettu muodossa poisto/lisäys. Onnistumiset ovat 10 toiston keskiarvoja.

Bridge

	20	50	100	200	500	1000	2000	5000	10000
Sat. / Sat.	14.1	28.5	47.2	74.7	122.5	162.7	200.7	249.9	284.8
Det. / Sat.	17,7	39.0	64,4	92,2	134,5	167,4	203,6	246,4	258,7
Sat. / Det.	15,9	31,8	49,7	69,2	104,3	126,2	145,5	148,3	151,5
Det. / Det.	19,3	26,9	26,9	26,9	26,9	26,9	26,9	26,9	26,9

Birch2

	20	50	100	200	500	1000	2000	5000	10000
Sat. / Sat.	7.2	12.5	17.3	26	33.8	40.6	43.7	43.7	43.7
Det. / Sat.	13.0	18,7	22,2	22,3	22,3	22,3	22,3	22,3	22,3
Sat. / Det.	9,7	14,2	15,5	15,5	15,5	15,5	15,5	15,5	15,5
Det. / Det.	5,3	5,3	5,3	5,3	5,3	5,3	5,3	5,3	5,3

Aineistolla *Birch2* havaitaan täysin deterministisen version suorituksen päättyvän jo alle 20 iteraatiossa ja muidenkin determinististen versioiden suorituksen päättyvän alle sadassa iteraatiossa. Täysin satunnainen versio sen sijaan jatkaa suoritustaan parin tuhannen iteraation paikkeille. Myös *Birch2* aineistolla deterministiset versiot pystyvät aluksi parantamaan ratkaisua satunnaista versiota useammin. Suoritettaessa enemmän iteraatioita täysin satunnainen versio löytää deterministisiä versioita useammin parannusta. Tästäkin huolimatta jää se kuitenkin tehokkuudessa deterministisille versioille. Näin ollen voidaan päätellä, että deterministiset versiot tuottavat täysin satunnaista versiota enemmän parannusta kullakin onnistuneella sentroidin korvauksella. Esimerkiksi täysin deterministinen versio tarvitsee *Birch2* aineistolla noin 8 kertaa vähemmän onnistuneita sentroidin korvauksia kuin täysin satunnainen versio päästäkseen samaan lopputulokseen. Ero tarvittavien paikallishakuiteraatioiden määrässä on vastaavasti useampi satakertainen.

Suoritetuista testeistä havaittiin, että onnistuneiden sentroidin korvausten määrää iteraatiota kohden on deterministisillä versioilla saatu parannettua. Tämä on kuitenkin tapahtunut yhteen iteraatioon kuluvan ajan kustannuksella. Esimerkiksi täysin deterministinen versio vie *Birch2* aineistolla iteraatiota kohden noin 9 kertaa enemmän aikaa kuin täysin satunnainen versio. Täysin deterministi-

nen versio kuitenkin löytää jokaisella iteraatiolla parannusta ja näin ollen se saavuttaa muita paikallishaun versioita paremman tehokkuuden.

6.5. Paikallishaun jatkokehitysideoita

Esitettävät jatkokehitysideat on jaettu kahteen ryhmään, jotka ovat nopeuttaminen ja laadun parantaminen. Nopeuttamisessa paikallishaun tehokkuutta pyritään parantamaan vähentämällä yhteen iteraatioon kuluvaa aikaa. Laadun parantamisessa kehitetään paikallishaun deterministisiä versioita siten, että ne pystyvät saavuttamaan entistä parempia ratkaisuja ei-klusteroituneilla aineistoilla.

6.5.1. Nopeuttaminen

Luvussa 5 valitsimme PDS-menetelmän osaksi lähimmän naapurin haun nopeuttamista sen yksinkertaisuuden ja yleiskäyttöisyyden vuoksi. Kuitenkin Kaukoranta ja muut (2000) ovat nopeuttaneet K-meansia enemmän käyttämällä PDS-menetelmän lisäksi muita parempia menetelmiä kuten MPS-menetelmää. Näin ollen *OptimalPartition*-operaatiota voidaan nopeuttaa vielä hieman käyttämällä jotain parempaa lähimmän naapurin haun nopeutusmenetelmää.

Kohdassa 4.2.1 selvitettiin K-means –iteraatioiden määrän vaikutus satunnaisen paikallishaun tehokkuuteen. Iteraatioiden määrän vaikutusta tulisi tarkastella myös determinististen versioiden tapauksessa. LBG-U-menetelmän saavuttamat deterministä paikallishakuamme paremmat tulokset kuva- ja puheaineistoilla viittaavat siihen, että lisäiteraatiot voisivat parantaa menetelmän tehokkuutta. LBG-U vastaa deterministä paikallishakua Fritzke-poistolla sillä erolla, että LBG-U-menetelmä iteroi K-meansia niin kauan, kun se konvergoi.

K-meansin nopeuttamisen lisäksi olisi syytä myös miettiä, miten K-meansin suorittaminen voitaisiin välttää. Nopeutusta voidaan saavuttaa, mikäli K-meansia ei suoriteta huonoille ratkaisuille ollenkaan, vaan hylätään ne suoraan. Hylkäyskriteerinä voisi olla esimerkiksi, että sentroidin korvauksen ja paikallisen uudelleenosituksen jälkeen ratkaisun pitää olla nykyistä ratkaisua parempi, tai ainakin riittävän lähellä sitä, jotta ratkaisulle suoritettaisiin K-meansia. Tämä idea toimisi parhaiten satunnaisen version kanssa, sillä se muodostaa runsaasti ratkaisuja, jotka eivät tuota parannusta.

Poistettavan sentroidin deterministisen valinta on kallis operaatio sen vaatiman toissijaisen osituksen laskemisen vuoksi. Toissijaista ositusta ei kuitenkaan tarvitse laskea aina uudestaan alusta asti, vaan sitä voidaan ylläpitää samaan tapaan kuin ylläpidetään nyt ensisijaista ositusta. Tämä on mahdollista yleistämällä karsittujen vertailujen haun idea tapaukseen, jossa kaksi sentroidia liikkuu suhteessa alkioon. Koska kunkin alkion ensi- ja toissijainen sentroidi voi olla aktiivinen tai staattinen, on tässä yleistetyssä karsittujen vertailujen haussa neljä vaihtoehtoa kahden (aktiivinen ja staattinen) sijaan.

6.5.2. Laadun parantaminen

Paikallishaun determinististen versioiden ongelmana ei-klusteroituneilla aineistoilla on, että ne pystyvät parantamaan ratkaisua vain tiettyyn pisteeseen asti. Yksinkertainen ratkaisuyritelmä täysin deterministisen version tapauksessa on, että mikäli korvaus ei tuota tulosta, poistetaan seuraavalla iteraatiolla se klusteri, jonka poistokustannus on seuraavaksi alin. Tämä johtaa kuitenkin parhaimmillaan vain satunnaista poistoa ja deterministä lisäystä käyttävän version tasolle ratkaisun laadussa. Nimittäin nyt molemmat versiot yrittävät loppujen lopuksi kunkin klusterin poistamista klusterin lisäyskriteerin ollessa sama.

Eräs varteenotettava vaihtoehto on paikallishakujen parhaat puolet yhdistävä *hybridiversio*, jossa haku muuttuu asteittain deterministisestä satunnaiseksi. Esimerkiksi aluksi suoritetaan täysin determinististä paikallishakua, kunnes ratkaisu ei enää parane. Sen jälkeen vaihdetaan determinististä lisäystä tai poistoa käyttävään versioon ja lopuksi siirrytään täysin satunnaiseen paikallishakuun. Siirtymähetken päättelemiseksi ei kuitenkaan liene mitään yksinkertaista keinoa. Eräs mahdollisuus on mitata ratkaisun paranemisnopeutta tai –kiihtyvyyttä ja vaihtaa versiota nopeuden tai kiihtyvyyden laskiessa liian alas. Kuva-aineistoilla tehtyjen testien perusteella täysin deterministinen versio kannattaa jättää väliin ja hybridissä kannattaa aloittaa determinististä lisäystä käyttävällä versiolla, josta vaihdetaan täysin satunnaiseen versioon.

Hieman edellistä muistuttava idea on tehdä täysin deterministisestä paikallishausta stokastinen versio, jossa satunnaisuuden vaikutusta sentroidin korvauksessa lisättäisiin asteittain. Idea muistuttaa simuloitua jäähdytystä, mutta toiminta on päinvastainen, sillä nyt satunnaisuuden määrä lisääntyy. Deterministisen heuristiikan vaikutus on aluksi suuri ja ne korvaukset, jotka deterministisyyden mukaan ovat hyviä, tulevat erittäin todennäköisesti valituksi. Haun edetessä ja satunnaisuuden lisääntyessä valitaan heuristiikan mukaan entistä huonompia korvauksia. Lopulta kaikki korvaukset ovat yhtä todennäköisiä, jolloin haku on muuttunut täysin satunnaiseksi paikallishauksi. Ongelmana idean toteuttamisessa on, kuinka valitaan satunnaisuuden lisääntymisnopeus. Todennäköisesti satunnaisuuden lisääntymisnopeuden tulee riippua jollain tavalla aineistosta.

Tämän kohdan alussa ehdotettua ei-optimaalisen ratkaisun sallivaa täysin deterministisestä versiosta muokattua paikallishakuversiota voidaan jalostaa edelleen lisäämällä siihen tabulista. Tässä paikallishaun tabuhakuversiossa ylläpidetään tabulistaa sekä poistettavalle että lisättävälle klusterille. Tavallisesta tabuhausta poiketen huonompia ratkaisuja ei edelleenkään hyväksyttäisi, vaan tabulistan tarkoituksena olisi pakottaa algoritmi kokeilemaan muitakin ratkaisuja kuin deterministisen valinnan mukaista. Tabuhakuversio säilyttää menetelmän deterministisyyden.

Deterministisen naapuristofunktion kehittämisellä voi todennäköisesti parantaa ratkaisun laatua käytettävän ajan kustannuksella. Hansen ja Mladenović (2001) ovat esittäneet J-means –menetelmän. Menetelmä on deterministinen paikallishaku, jossa kullakin iteraatiolla kokeillaan kaikki mahdolliset sentroidin korvaukset ja valitaan näistä paras. Lisäystä kuitenkin rajoitetaan siten, ettei se voi tapahtua olemassa olevan sentroidin läheisyyteen. Tätä menetelmää soveltaen voidaan lisätä entisestään deterministisyyttä nykyiseen naapuristofunktioon ainakin kolmella eri tavalla. Voidaan valita optimaalinen kohdeklusteri lisäykselle, kokonaan optimaalinen sijainti lisättävälle sentroidille tai äärimmäisenä vaihtoehtona J-meansin tapaan käydä läpi kaikki mahdolliset sentroidin korvaukset ja valita näistä optimaalinen. Optimaalinen eli paras korvaus haetaan kussakin tapauksessa suorittamalla ratkaisulle paikallinen uudelleenositus ja mittaamalla näin saadun ratkaisun virhe. Optimaalisen kohdeklusterin valinta aiheuttaa täten $O(NMK)$ työn, optimaalisen sijainnin valinta $O(N^2K)$ työn ja optimaalisen korvauksen valinta $O(N^2MK)$ työn.

7. Yhteenveto

Klusterointi on määritelty tässä tutkielmassa kombinatorisena optimointiongelmana, ja sen ratkaisemiseen on käytetty paikallishakua. Paikallishaku perustuu naapuruuden käsitteeseen. Naapuriratkaisun muodostamiseksi on tutkittu sekä satunnaisia että determinististiä mahdollisuuksia.

Paikallishaun satunnainen versio on rakenteeltaan yksinkertainen, mutta voittaa silti tehokkuudessa esimerkiksi laajalti käytetyn K-means –menetelmän. Vertaaminen K-meansiin on relevanttia, sillä se on osana paikallishaun naapuriratkaisun hienosäätöä. Mikäli naapuriratkaisujen muodostaminen valittaisiin huonosti, muodostaisi paikallishaku pahimmassa tapauksessa vain uusia satunnaisia ratkaisuja K-meansille.

Paikallishaussa on kaksi säädettävää parametria. Paikallishakuiteraatioiden määrällä voidaan säätää ratkaisun laadun ja käytettävän ajan välistä suhdetta. Tarvittavalle paikallishakuiteraatioiden määrälle on johdettu arvio kohdassa 4.2.2. K-means –iteraatioiden määrästä havaittiin, että K-meansin tekeminen paikallishaun osana kannattaa. Sopiva K-means –iteraatioiden määrä satunnaista versiota käytettäessä on kaksi. Satunnaisen version pullonkaulaksi havaittiin lähimmän naapurin haku *OptimalPartition*-operaatiossa. Operaatiota nopeutettin käyttämällä PDS- ja karsittujen vertailujen haku –menetelmiä. Karsittujen vertailujen haku on alun perin esitetty käytettäväksi K-meansin nopeuttamiseen. Havaittiin kuitenkin, että nopeutusmenetelmistä on jopa enemmän hyötyä, kun K-meansia tehdään paikallishaun osana. Nopeutetun paikallishaun suoritusaika on kuva-aineistoilla vain 8–17% alkuperäisestä suoritusajasta.

Satunnaisen version havaittiin muodostavan runsaasti naapuriratkaisuja, jotka eivät kuitenkaan paranna ratkaisua. Esimerkiksi kuva-aineiston *Miss America* tapauksessa 10000 muodostetusta ratkaisusta ainoastaan 4 % johti parannukseen. Tämä on potentiaalinen tehostuskohde, joten paikallishaun tehokkuutta on pyritty parantamaan tekemällä paikallishausta deterministisempi. Tällä pyritään siihen, että paikallishaku löytää useammin parannusta ja toivon mukaan pääsee näin nopeammin hyvään ratkaisuun. Paikallishaun determinististen versioiden havaittiin löytävän, ainakin aluksi, satunnaista versiota useammin parannusta. Tämä kuitenkin tapahtuu suoritusajan kustannuksella.

Kuva- ja puheaineistoilla suoritetun testauksen nojalla paikallishaun deterministiset versiot vaikuttavat olevan satunnaista versiota tehokkaampia aineistoilla, joissa on havaittavissa selkeitä klustereita. Aineistoilla, joilla ei ole havaittavissa selkeää rakennetta, deterministiset versiot pärjäävät tehokkuudessa satunnaiselle versiolle vain aluksi. Lisäksi determinististen versioiden ongelma näillä aineistoilla on, että ne pystyvät parantamaan ratkaisua vain tiettyyn pisteeseen asti. Parhaan paikallishakuversion valinta vaikuttaisi siis riippuvan klusteroitavasta aineistosta. Täysin deterministinen versio on tehokkain aineistoilla, joissa on selkeitä klustereita. Täysin satunnainen versio on paras valinta ei-klusteroituneille aineistoille. Klusterin satunnaisesti poistava ja luotavan klusterin sijainnin deterministisesti valitseva versio on hyvä kompromissi täysin satunnaisen ja täysin deterministisen version välillä.

Lisätutkimusta tarvitaan sekä determinististen versioiden ei-klusteroituneilla aineistoilla saavutettavan laadun parantamiseksi että deterministisen korvauksen tehokkuuden parantamiseksi. Kohdassa 6.5. on esitetty muutamia jatkokehitysideoita näihin liittyen. Ratkaisun laatua voidaan kehittää muuttamalla käytettävää naapuriratkaisujen muodostamista. Tehokkuutta saavutetaan pitämällä yllä poistettavan sentroidin deterministisen valinnan vaatimaa toissijaista ositusta samaan tapaan kuin ensisijaista ositusta pidetään yllä.

Viitteet

Aarts, E.H.L., Lenstra, J.K. (1997) *Local search in combinatorial optimization*. John Wiley & Sons, Chichester.

Aarts, E.H.L., Van Laarhoven, P.J.M. (1985) Simulated annealing: an introduction. *Statistica Neerlandica* **43**, 31–52.

Aarts, E.H.L., Van Laarhoven, P.J.M., Lenstra, J.K., Ulder, N.L.J. (1994) A computational study of local search algorithms for job stop scheduling. *ORSA Journal on Computing* **6**, 118–125.

Aghbari, Z.A. (2005) Array-index: a plug&search k nearest neighbors method for high-dimensional data. *Data & Knowledge Engineering* **52**(3), 333–352.

Al-Sultan, K.S. (1995) A tabu search approach to the clustering problem. *Pattern Recognition* **28**(9), 1443–1451.

Al-Sultan, K.S., Khan, M.M. (1996) Computational experience on four algorithms for the hard clustering problem. *Pattern Recognition Letters* **17**(3), 295–308.

Anderson, E.J. (1996) Mechanisms for local search. *European Journal of Operational Research* **88**(1), 139–151.

Andritsos, P., Tzerpos, V. (2005) Information-theoretic software clustering. *IEEE Transactions on Software Engineering* **31**(2), 150–165.

Aurenhammer, F. (1991) Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys* **23**(3), 345-405.

Bandyopadhyay, S., Maulik, U. (2002) Efficent prototype reordering in nearest neighbor classification. *Pattern Recognition* **35**(12), 2791–2799.

Bei, C.-D., Gray, R.M. (1985) An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Transactions on Communications* **33**(10), 1132–1133.

Bentley, J.L., Preparata, F.P., Faust, M.G. (1982) Approximation algorithms for convex hulls. *Communication of the ACM* **25**(1), 64-68.

Carson, T. (2001) *Empirical and analytic approaches to understanding local search heuristics*. Väitöskirja, Kalifornian yliopisto, San Diego.

Chen, J., Bouman, C.A., Dalton, J.C. (2000) Hierarchical browsing and search of large image databases. *IEEE Transactions on Image Processing* **9**(3), 442-455.

Chen, S.-H., Hsieh, W.M. (1991) Fast algorithm for VQ codebook design. *IEE Proceedings in Communications, Speech and Vision* **138**(5), 357–362.

Cios, K., Pedrycz, W., Swiniarski, R. (1998) Data mining methods for knowledge discovery. Kluwer, Boston.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2001) *Introduction to algorithms*, Second edition. MIT Press, Cambridge.

Delport, V., Koschorreck, M. (1995) Genetic algorithm for codebook design in vector quantization. *Electronic Letters* **31**(2), 84–85.

Dimitriou, A., Impagliazzo, R. (1996) Towards a rigorous analysis of local optimization algorithms. *28th ACM Symposium on the Theory of Computing*, ACM Press, New York, 304–313.

Everitt, B.S. (1993) *Cluster analysis*, Third edition. Arnold, London.

Fasulo, D. (1999) *An analysis of recent work on clustering algorithms*. Technical Report # 01-03-02, Washingtonin yliopisto, Seattle. Saatavana myös `http://citeseer.ist.psu.edu/fasulo99analysi.html` (13.04.2006).

Frawley, W.J., Pietetsky-Shapiro, G., Zimmerman, H.G. (1992) Knowledge discovery in databases: an overview. *AI Magazine* **13**(3), 57–70.

Free Software Foundation (FSF) (2006) *GNU binutils*. WWW-sivusto, `http://www.gnu.org/software/binutils/` (13.04.2006).

Fritzke, B. (1997) The LBG-U method for vector quantization – an improvement over LBG inspired from neural networks. *Neural Processing Letters* **5**(1), 35–45.

Fränti, P. (2000) Genetic algorithm with deterministic crossover for vector quantization. *Pattern Recognition Letters* **21**(1), 61–68.

Fränti, P., Gyllenberg, H.G. Gyllenberg, M. Kivijärvi, J., Koski, T., Lund, T., Nevalainen, O. (2000) Minimizing stochastic complexity using local search and GLA and with applications to classification of bacteria. *Biosystems* **57**(1), 37–48.

Fränti, P., Kivijärvi, J. (2000) Randomized local search algorithm for the clustering problem. *Pattern Analysis and Applications* **3**(4), 358-369.

Fränti, P., Kaukoranta, T., Nevalainen, O. (1997a) On the splitting method for VQ codebook generation. *Optical Engineering* **36**(11), 3043–3051.

Fränti, P., Kivijärvi, J., Kaukoranta, T., Nevalainen, O. (1997b) Genetic algorithms for large-scale clustering problems. *The Computer Journal* **40**(9), 547–554.

Fränti, P., Kivijärvi, J., Nevalainen, O. (1998) Tabu search algorithm for codebook generation in VQ. *Pattern Recognition* **31**(8), 1139–1148.

Fränti, P., Virmajoki, O. (2006) Iterative shrinking method for the clustering problems. *Pattern Recognition* **39**(5), 761–775.

Fränti, P., Virmajoki, O., Kaukoranta T. (2002) Branch-and-bound technique for solving optimal clustering. *16th International Conference on Pattern Recognition*, Québec, August 2002, 2: 232–235.

Fränti, P., Xu, M., Kärkkäinen, I. (2003) Classification of binary vectors by using ΔSC distance to minimize stochastic complexity. *Pattern Recognition Letters* **24**(1–3), 65–73.

Garey, M.R., Johnson, D.S. (1979) *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, San Francisco.

Garey, M.R., Johnson, D.S., Witsenhausen, H.S. (1982) The complexity of the generalized Lloyd-Max problem. *IEEE Transactions on Information Theory* **28**(2), 255-256.

Gersho, A., Gray, R.M. (1992) *Vector quantization and signal compression*. Kluwer, Dordrecht.

Graham, R.L., Knuth, D.E., Patashnik, O. (1994) *Concrete mathematics – a foundation for computer science*, Second edition. Addison-Wesley, Reading.

Gupta, J.N.D., Hennig, K., Werner, F. (2000) Local search heuristics for two-stage flow shop problems with secondary criterion. *Computers & Operations Research* **29**(2), 123–149.

Glover, F. (1989) Tabu search: part I. *ORSA Journal on Computing* **1**, 190–206.

Han, J., Kamber, M. (2001) *Data mining: concepts and techniques*. Morgan Kaufmann Publishers, San Francisco.

Hansen, P., Mladenović, N. (2001) J-means: a new local search heuristic for minimum sum of squares clustering. *Pattern Recognition* **34**(2), 405-413.

Hjaltason, G.R., Samet, H. (2003) Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems* **28**(4), 517–580.

Holland, J.H. (1975) *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor.

Jain, A.K., Dubes, R.C. (1988) *Algorithms for clustering data*. Prentice-Hall, Englewood Cliffs, New Jersey. Saatavana myös `http://www.cse.msu.edu/~jain/Clustering_Jain_Dubes.pdf` (13.04.2006).

Jain, A.K., Murty, M.N., Flynn, P.J. (1999) Data clustering: a review. *ACM Computing Surveys* **31**(3), 264-323.

Johnson, D.S. (1990) Local optimization and the traveling salesman problem. *17th International Colloquium on Automata, Languages, and Programming*, Warwick, July 1990, 446–461.

Kaufman, L., Rousseeuw, P.J. (1990) *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, New York.

Kaukoranta T., Fränti, P., Nevalainen, O. (1998) Iterative split-and-merge algorithm for VQ codebook generation. *Optical Engineering* **37**(10), 2726–2732.

Kaukoranta, T., Fränti, P., Nevalainen, O. (2000) A fast exact GLA based code vector activity detection. *IEEE Transactions on Image Processing* **9**(8), 1337–1342.

Kiers, H.A.L., Rasson, J.-P., Groenen, P.J.F., Schader, M. (2000) *Data analysis, classification, and related methods*. Springer, Berlin.

King, B. (1967) Step-wise clustering procedures. *Journal of the American Statistical Association* **62**(317), 86–101.

Kinnunen, T. (2005) *Optimizing spectral feature based text-independent speaker recognition*. Väitöskirja, Tietojenkäsittelytiede, Joensuun yliopisto, Joensuu.

Kinnunen, T., Karpov, E., Fränti, P. (2004) Real-time speaker identification. *8th International Conference on Spoken Language Processing*, Jeju Island, October 2004, 3: 1805-1808.

Kinnunen, T., Kilpeläinen, T., Fränti, P. (2000) Comparison of clustering algorithms in speaker identification. *IASTED International Conference on Signal Processing and Communications*, Marbella, September 2000, 222-227.

Kinnunen, T., Kärkkäinen I., Fränti, P. (2001) Is speech data clustered? – statistical analysis of cepstral features. *7th European Conference on Speech Communication and Technology*, Aalborg, September 2001, 4: 2627-2630.

Kinnunen T., Tuononen, M., Fränti, P. (2006) Which clustering algorithm to select for text-independent speaker recognition. *IEEE Transactions on Audio, Speech and Language Processing* (submitted 19.02.2006).

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P. (1983) Optimization by simulated annealing. *Science* **220**(4598), 671–680.

Kivijärvi, J., Fränti, P., Nevalainen, O. (2003) Self-adaptive genetic algorithm for clustering. *Journal of Heuristics* **9**(2), 113–129.

Kopylov P., Fränti, P. (2004) Color quantization of map images. *4th IASTED International Conference on Visualization, Imaging, and Image Processing*, Marbella, September 2004, 837–842.

Kärkkäinen, I., Fränti, P. (2002) Dynamic local search for clustering with unknown number of clusters. *16th International Conference on Pattern Recognition*, Québec, August 2002, 2: 240–243.

Lázaro, J., Arias, J., Martín, J.L., Cuadrado, C., Astarloa, A. (2005) Implementation of a modified fuzzy c-means clustering algorithm for real-time applications. *Microprocessors and Microsystems* **29**(8–9), 375–380.

Linguistic Data Consortium (LDC) (2006) *LDC - linguistic data consortium*. WWW-sivusto, `http://www.ldc.upenn.edu/` (13.04.2006).

Lehtinen, J., Kivijärvi, J. (2001) Clustering context properties of wavelet coefficients in automatic modelling and image coding. *11th International Conference on Image Analysis and Processing*, Palermo, September 2001, 151–155.

Leon, E., Clarke, G., Callaghan, V., Sepulveda, F. (2004) Real-time detection of emotional changes for inhabited environments. *Computers & Graphics* **28**(5), 635–642.

Lyman, P., Hal, R.V. (2003) *How Much Information*. http://www.sims.berkeley.edu/how-much-info-2003 (13.04.2006).

Ma, E.W.M., Chow, T.W.S. (2004) A new shifting grid clustering algorithm. *Pattern Recognition* **37**(3), 503-514.

Martin, O., Otto, S.W., Felten, E.W. (1992) Large-step markov chains for the TSP incorporating local search heuristics. *Operations Research Letters* **11**(1) 219–224.

McQueen, J.B. (1967) Some methods for classification and analysis of multivariate observations. *5th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, 1: 281–297.

Metropolis, N., Rosenbluth, A.W., Teller, A.H., Rosenbluth, M.N., Teller E. (1953) Equation of state calculations by fast computing machines. *Journal of Chemical Physics* **21**(6), 1087–1092.

Murthya, C.A., Chowdhuryb, N. (1996) In search of optimal clusters using genetic algorithms. *Pattern Recognition Letters* **17**(8), 803–927.

Mühlenbein, H., Gorges-Schleuter, M., Krämer, O. (1988) Evolution algorithms in combinatorial optimization. *Parallel Computing* **7**, 65–85.

National Institute of Standards and Technology (NIST) (2006) *Speech group home*. WWW-sivusto, http://www.nist.gov/speech/ (13.04.2006).

Papadimitriou, C.H., Steiglitz, K. (1982) *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Englewood Cliffs.

Petrou, M. (1999) *Image processing: the fundamentals*. John Wiley & Sons, Chichester.

Ra, S.-W., Kim, J.-K. (1993) A fast mean-distance-ordered partial codebook search algorithm for image vector quantization. *IEEE Transactions on Circuits and Systems* **40**(9), 576–579.

Rabiner, L., Juang, B-H. (1993) *Fundamentals of speech recognition*. Prentice-Hall, New Jersey.

Rus, C., Astola, J. (2005) Legend based elevation layers extraction from a color-coded relief scanned map. *IEEE International Conference on Image Processing*, Genova, September 2005, 2: 237–240.

Seal, S., Komarina, S., Aluru, S. (2005) An optimal hierarchical clustering algorithm for gene expression data. *Information Processing Letters* **93**(3), 143-147.

Selim, S.Z., Ismail, M.A. (1984) K-means-type algorithms: a generalized convergence theorem and characterization of local optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6**(1), 81–87.

Selman, B., Levesque, H.J., Mitchell, D. (1992) GSAT: A new method for solving hard satisfiability problems. *10th National Conference on Artificial Intelligence*, San Jose, July 1992, 440–446.

Shokoufandeh, A., Mancoridis, S., Denton, T., Maycock, M. (2005) Spectral and meta-heuristic algorithms for software clustering. *Journal of Systems and Software* **77**(3), 213–223.

Sneath, P.H.A., Sokal, R.R. (1973) *Numerical taxonomy*. Freeman, London.

Soong, F., Juang, B-H., Rabiner, L. (1987) A vector quantization approach to speaker recognition. *AT & T Technical Journal* **66**(2), 14–26.

Sung, C.S., Jin, H.W. (2000) A tabu-search-based heuristics for clustering. *Pattern Recognition* **33**(5), 849–858.

Späth, H. (1985) *Cluster dissection and analysis: theory, FORTRAN programs, examples*. Ellis Horwood Limited, Chichester.

Theodoridis, S., Konstantinos, K. (2003) *Pattern recognition*, Second edition. Academic Press, London.

Tseng, L.Y., Yang, S.B. (2000) A genetic clustering algorithm for data with non-spherical-shape clusters. *Pattern Recognition* **33**(7), 1251–1259.

Tuononen, M. (2004) *Ohjelmaselostus – randomized local search, repeated K-means ja global k-means*. Erikoistyö, Tietojenkäsittelytiede, Joensuun yliopisto, Joensuu.

Tyson, N.G. (1994) *University down to earth*. Columbia University Press, New York.

Virmajoki, O. (2004) *Pairwise nearest neighbour method revisited*. Väitöskirja, Tietojenkäsittely-tiede, Joensuun yliopisto, Joensuu.

Ward, J.H. (1963) Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association* **58**(301), 236–244.

Wegener, I., Witt, C. (2005) On the optimization of monotone polynomials by simple randomized search heuristics. *Combinatorics, Probability and Computing* **14**(1), 225–247.

Wu, X., Witten, I. (1985) *A fast k-means type clustering algorithm*. Research Report 85/197/10, Calgaryn yliopisto, Alberta.

Wu, X., Zhang, K. (1991) A better tree-structured vector quantizer. *IEEE Data Compression Conference*, Snowbird, April 1991, 392–401.

Zhang, T., Ramakrishnan, R., Livny, M. (1997) BIRCH: a new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery* **1**(2), 141–182.

Liite 1: Paikallishaun edistyneen version toteutus

Luvussa 4 todettiin esitetyn paikallishaun perusversion pullonkaulana olevan optimaalisen osituksen muodostavan *OptimalPartition*-operaation. Kaukoranta ja muut (2000) ovat esittäneet karsittujen vertailujen haun keinona muodostaa optimaalinen ositus tehokkaammin osana K-meansia. Fränti ja Kivijärvi (2000) ovat lisäksi erityisesti kehottaneet käyttämään Kaukorannan ja muiden (2000) esittämää karsittujen vertailujen hakua paikallishaun tehostamiseksi. Tässä liitteessä on esitetty edistyneen version käyttämän PDS-menetelmän sekä karsittujen vertailujen haun toteutus.

PDS-menetelmä

Kuvassa L1.1 on esitetty pseudokoodina alkioiden välisen etäisyyden laskeminen PDS-menetelmää käyttäen. Kuvassa L1.1 esitetty *Dist*-operaatio saa parametreinaan alkiot x_i ja x_j, joiden välinen etäisyys halutaan saada selville, sekä rajan *maxDist*, jota pidemmälle etäisyyslaskua ei jatketa. Operaatio *Dist* palauttaa alkioiden x_i ja x_j välisen etäisyyden, mikäli se on pienempi kuin *maxDist*. Mikäli alkioiden x_i ja x_j etäisyys on suurempi kuin *maxDist*, palauttaa *Dist*-operaatio jonkin etäisyyden, joka on suurempaa kuin *maxDist*.

```
Dist (xi,xj,maxDist) {
    sum := 0;
    k   := 1;

    WHILE (sum < maxDist) AND (k <= K) DO {
        sum := sum + (xi^k - xj^k)^2;
        k   := k + 1;
    }

    RETURN sum;
}
```

Kuva L1.1: Etäisyyden laskeminen PDS-menetelmää käyttäen.

Huomattavaa on, että nyt *Dist*-operaation ja koko tässä kohdassa esiteltävän paikallishaun käyttämä etäisyys on euklidisen etäisyyden neliö. Syynä euklidisen etäisyyden neliön käyttämiseen on yksinkertaisesti tehokkuussyyt. Nimittäin muutoin PDS-menetelmää käytettäessä jouduttaisiin laskemaan toistuvasti siihen mennessä saadun etäisyyden neliöjuurta verrattaessa etäisyyttä löydettyyn minimietäisyyteen. *Dist*-operaation tehokkuus on erityisen tärkeää, sillä esimerkiksi kuvasta 4.7 nähdään, että etäisyyslaskuihin kulutetaan kaikista menetelmän operaatioista eniten aikaa.

Kuvassa L1.1 esitetty *Dist*-operaatio korvaa luvussa 3 esitetyn *Dist*-operaation. Myös jatkossa esitettävät nimeltään identtiset operaatiot tulevat korvaamaan vastaavat luvussa 3 esitetyt operaatiot. Koska paikallishaun operaatioiden tarvitsee laskea myös täsmällisiä alkioiden välisiä etäisyyksiä, määritellään vakio *MAXINT*, joka on suurin mahdollinen kokonaisluku. Nyt kutsumalla *Dist*-operaatiota $Dist(x_i,x_j,MAXINT)$ saadaan laskettua alkioiden x_i ja x_j välinen tarkka etäisyys.

Lähimmän sentroidin haun suorittava *FindNearestRepresentative*-operaatio hyötyy etäisyyslaskussa käytettävästä PDS-menetelmästä. Nimittäin lähimmän sentroidin haku tahtoo alkion ja sentroidin välistä etäisyyttä laskiessaan vain tietää, onko kyseinen sentroidi lähempänä alkiota kuin siihen mennessä löydetty lähin sentroidi. Kuvassa L1.2 esitetty lähimmän sentroidin haun suorittava *FindNearestRepresentative*-operaatio hakee koodikirjasta *C* alkiota *x* lähimpänä olevan sentroidin.

Parametri *guess* on alkuarvaus alkion *x* lähimmäksi sentroidiksi. Koska PDS-menetelmän tehokkuus riippuu suuresti nykyisen lähimmän alkion laadusta (Kaukoranta & al., 2000), valitaan *FindNearestRepresentative*-operaation lähtöarvaukseksi *guess* ositusta muodostettaessa edellisen osituksen mukainen lähin sentroidi.

FindNearestRepresentative-operaatio pitää yllä suorituksensa aikana siihen asti löydettyä lähintä sentroidia c_{nrst} sekä etäisyyttä *dst* kyseiseen sentroidiin. Etäisyyttä alkion *x* ja tutkittavana olevan sentroidin c_i välillä laskettaessa annetaan *Dist*-operaatiolle etäisyyslaskun maksimietäisyydeksi etäisyys *dst* siihen mennessä löydettyyn lähimpään sentroidiin.

```
FindNearestRepresentative(C,x,guess) {
    nrst := guess;
    dst  := Dist(x,c_guess);

    FOR i := 1 TO M DO {
        d := Dist(x,c_i,dst);

        IF d < dst THEN {
            nrst := i;
            dst  := d;
        }
    }

    RETURN (nrst,dst);
}
```

Kuva L1.2: Lähimmän sentroidin haku PDS-menetelmää hyödyntäen.

Huomattavaa on, että myös *LocalRepartition*- ja *ObjectiveFunction*-operaatiot täytyy muuttaa käyttämään *Dist*- ja *FindNearestRepresentative*-operaatioiden muuttuneita versioita. Muutokset näihin operaatioihin ovat kuitenkin suoraviivaisia, joten tarvittavia muutoksia ei kuvata tässä yksityiskohtaisemmin.

Karsittujen vertailujen haku

Karsittujen vertailujen haun toteuttamiseksi joudutaan tekemään muutoksia sekä koodikirjan muodostavaan *OptimalRepresentatives*-operaatioon että osituksen muodostavaan *OptimalPartition*-operaatioon. Osituksen pohjalta optimaalisia sentroideja laskettaessa *OptimalRepresentatives*-operaatiolla täytyy kirjata ylös muuttuneet sentroidit. Sentroidien pohjalta optimaalista ositusta laskettaessa *OptimalPartition*-operaatiolla päätellään sentroidien liikkumisen perusteella kullekin alkiolle, haetaanko lähintä sentroidia ainoastaan muuttuneiden vai kaikkien sentroidien joukosta.

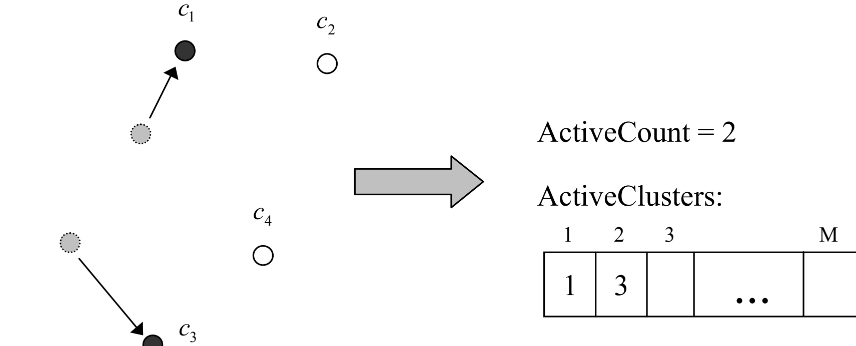
Kuvassa L1.3 on esitetty *OptimalRepresentatives*-operaation pseudokoodi. Operaatio laskee kuvassa 3.10 esitetyn perusversion tapaan osituksen pohjalta kunkin klusterin optimaalisen sentroidin sekä lisäksi kirjaa ylös muuttuneet sentroidit. Sentroidien muuttumisen tutkiminen aiheuttaa $O(MK)$ lisätyön, sillä kutakin uutta sentroidia täytyy verrata aiempaan sentroidiin kunkin komponentin suhteen. Vertailussa toki tiedetään sentroidien erilaisuus heti, kun löytyy ensimmäinen eroava komponentti. Muuttuneiden sentroidien kirjaaminen ei kuitenkaan muuta *OptimalRepresentatives*-operaation aikavaativuutta, joka on yhä $O(NK)$.

```
OptimalRepresentatives(P,C,X) {
    /* Taulukon Sum[1..M] alkioina on vektoreita ja
       taulukon Count[1..M] alkioina on kokonaislukuja.
       Molemmat taulukot on alustettu nollaksi. */

    FOR i := 1 TO N DO {
        j      := p_i;
        Sum_j   := Sum_j + x_i;  /* vektoreiden alkioittainen summa */
        Count_j := Count_j + 1;
    }

    j := 0;

    FOR i := 1 TO M DO {
        IF Count_i <> 0 THEN {
            old  := c_i;
            c_i   := Sum_i / Count_i;

            /* sentroidi muuttunut, joten klusteri aktiivinen */
            IF c_i <> old THEN {
                ActiveCount             := ActiveCount + 1;
                ActiveClusters_ActiveCount := i;
            }
        }
    }

    RETURN (C,ActiveClusters,ActiveCount);
}
```

Kuva L1.3: Sentroidien laskeminen ja aktiivisten klustereiden kirjaaminen.

OptimalRepresentatives-operaatio palauttaa muuttuneiden sentroidien määrän *ActiveCount* ja aktiiviset klusterit taulukossa *ActiveClusters*. Viimeksi mainittu tulee automaattisesti nousevaan järjestykseen, sillä uudet sentroidit lasketaan nousevassa järjestyksessä aloittaen ensimmäisestä klusterista. Näin ollen voidaan *OptimalPartition*-operaatiossa selvittää klusterin aktiivisuus hakemalla *ActiveClusters*-taulukosta *binäärihaulla* (*binary search*; Cormen & al., 2001) kyseistä klusteria.

Kuvassa L1.4 on havainnollistettu *OptimalRepresentatives*-operaation suorittamaa aktiivisten sentroidien kirjaamista. Kuvan L1.4 tilanteessa sentroidit c_1 ja c_3 ovat muuttuneet. Näin ollen aktiivisten sentroidien ja samalla myös aktiivisten klustereiden lukumäärä *ActiveCount* on kaksi. Edelleen aktiiviset klusterit 1 ja 3 on kirjattu ylös *ActiveClusters*-taulukkoon.



Sentroidit:

- Aktiivinen, uusi sijainti
- Aktiivinen, vanha sijainti
- Staattinen

Kuva L1.4: Aktiivisten sentroidien kirjaaminen.

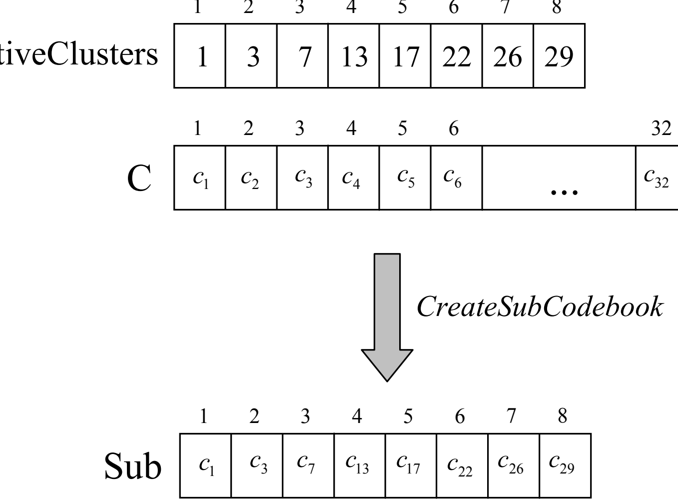
Karsittujen vertailujen hakua käyttävä *OptimalPartition*-operaatio tulee suorittamaan lähimmän sentroidin hakuja sekä koko koodikirjasta *C* että sen osajoukosta *Sub*, jossa on vain aktiiviset sentroidit. Kuvassa L1.5 on esitetty alikoodikirjan *Sub* muodostavan *CreateSubCodebook*-operaation pseudokoodi. Operaation muodostama koodikirja on samassa järjestyksessä *ActiveClusters*-taulukon kanssa, joten alikoodikirjasta Sub suoritetun lähimmän naapurin haun tuloksena saatu sentroidi on mahdollista palauttaa alkuperäisen koodikirjan sentroidiksi.

```
CreateSubCodebook(C,ActiveClusters,ActiveCount) {
    FOR i := 1 TO ActiveCount DO {
        j    := ActiveClusters_i;
        Sub_i := c_j;
    }

    RETURN Sub;
}
```

Kuva L1.5: Alikoodikirjan luonti.

Alikoodikirjan muodostamista on havainnollistettu esimerkillä kuvassa L1.6, jossa klustereiden lukumäärä *M* on 32 ja aktiivisten klustereiden lukumäärä *ActiveCount* on 8. *CreateSubCodebook*-operaatio luo alikoodikirjan *Sub*, jossa on vain aktiiviset sentroidit. Toisin sanoen jos $ActiveClusters_i$ on j, niin tällöin Sub_i on c_j.

Olkoon käsiteltävänä alkio x_i, joka siis kuuluu klusteriin $j = p_i$. Klusteri j on aktiivinen, mikäli luku j löytyy *ActiveClusters*-taulukosta. Koska *ActiveClusters*-taulukko on *OptimalRepresentatives*-operaation jäljiltä nousevassa järjestyksessä, voidaan luvun j haku toteuttaa tehokkaasti binäärihakuna. Kuvan L1.7 pseudokoodissa esiintyvä *BinarySearch*-operaatio etsii siis binäärihakua käyttäen taulukosta *ActiveVectors* alkion j ja palauttaa taulukon indeksin, josta kyseinen alkio löytyi. Mikäli alkiota j ei löytynyt, palauttaa *BinarySearch*-operaatio arvon -1. Binäärihaun aikavaativuus pahimmassa tapauksessa on $O(\log n)$, jossa n on taulukon koko (Cormen & al., 2001).



Kuva L1.6: Alikoodikirjan luominen.

Kuvassa L1.7 on esitetty pseudokoodina osituksen muodostaminen karsittujen vertailujen hakua käyttäen. *OptimalPartition*-operaatio saa parametrinaan tavanomaisten aiemman osituksen P, koodikirjan C ja aineiston X lisäksi tiedon aktiivista klustereista *ActiveClusters*-taulukossa ja niiden määrästä *ActiveCount*-muuttujassa. Lisäksi operaatio saa *Distances*-taulukossa kunkin alkion etäisyydet sentroidiinsa ennen *OptimalRepresentatives*-operaation suoritusta. Alkion x_i, joka kuuluu siis klusteriin $j = p_i$, aiempi etäisyys sentroidiinsa c_j on siis $Distances_i$. Operaatio laskee ja palauttaa saamansa koodikirjan pohjalta optimaalisen osituksen P sekä nykyisiä etäisyyksiä vastaamaan korjatun *Distances*-taulukon.

OptimalPartition-operaatio luo aluksi *CreateSubCodebook*-operaatiota käyttäen tarvitsemansa alikoodikirjan *Sub*, jossa on ainoastaan aktiiviset sentroidit. Tämän jälkeen operaatio etsii kullekin alkiolle lähimmän sentroidin suorittaen tapauskohtaisesti joko vain osittaisen haun tai täyden haun. Alkiolle x_i, joka kuuluu siis klusteriin $j = p_i$, voidaan havaita kolme erilaista tapausta. Nimittäin käsiteltävä alkio x_i voi kuulua staattiseen klusteriin, käsiteltävä alkio voi kuulua aktiiviseen klusteriin ja sentroidi c_j on liikkunut lähemmäksi alkiota tai käsiteltävä alkio voi kuulua aktiiviseen klusteriin ja sentroidi c_j on liikkunut kauemmaksi alkiosta. Näistä ensimmäinen ja toinen tapaus ovat periaatteessa samoja, sillä tällöin lähintä sentroidia tarvitsee hakea vain alikoodikirjasta *Sub*. Erona kuitenkin on, että toisessa tapauksessa aiempi sentroidi kuuluu alikoodikirjaan, jolloin aiempaa sentroidia voidaan käyttää alkuarvauksena. Lisäksi ensimmäisessä tapauksessa on muistettava, että myös nykyinen, staattinen sentroidi voi olla lähin. Kolmannessa tapauksessa on haku suoritettava täydestä koodikirjasta C.

Esimerkiksi kuvan L1.6 tilanteessa, jossa alkio x_i kuuluu aktiiviseen klusteriin $p_i = 13$ ja sentroidi c_{13} on liikkunut lähemmäksi alkiota x_i selvittää *OptimalPartition*-operaatio ensin alkion x_i nykyisen klusterin $j = p_i = 13$ sekä klusterin 13 indeksin alikoodikirjassa *Sub* binäärihaulla $k = BinarySearch(ActiveClusters,\ 8,\ 13) = 4$. Koska klusteri löytyi alikoodikirjasta, on se aktiivinen. Lisäksi sentroidi c_{13} on liikkunut lähemmäksi alkiota x_i eli $dstCurr < Distances_i$, joten tässä tapauksessa lähimmän sentroidin etsintä alkiolle x_i voidaan rajoittaa alikoodikirjaan *Sub*. Alikoodikirjasta löydetty lähin sentroidi $nrst = FindNearestRepresentative(Sub,\ x_i,\ 13) = 7$ täytyy vielä saada viittaamaan oikeaan täyden koodikirjan C klusteriin, joka on $nrst = ActiveClusters_{nrst} = ActiveClusters_7 = 26$. Koska nyt löydetty alkion x_i lähin sentroidi c_{26} on eri kuin aiempi lähin c_{13}, muuttaa *OptimalPartion*-operaatio vielä ositusta vastaavasti ja päivittää *Distances*-taulukkoon uuden etäisyyden.

```
OptimalPartition(P,C,X,ActiveClusters,ActiveCount,Distances) {

    /* luodaan koodikirja Sub, jossa vain aktiiviset sentroidit */
    Sub := CreateSubCodebook(C,ActiveClusters,ActiveCount);

    FOR i := 1 TO N DO {
        j       := pi;
        k       := BinarySearch(ActiveClusters,ActiveCount,j);
        dstCurr := Dist(xi,cj,MAXINT);

        /* alkio kuuluu staattiseen klusteriin */
        IF k < 0 THEN {
            /* etsintä voidaan siis rajoittaa alikoodirjaan Sub */
            (nrst,dstNew) := FindNearestRepresentative(Sub,xi,0);

            IF dstNew < dstCurr THEN {
                nrst := active[nrst];
            } ELSE {
                nrst := j;
            }

        /* alkio kuuluu aktiiviseen klusteriin ja
           sentroidi on liikkunut lähemmäksi alkiota */
        } ELSE IF (dstCurr < Distancesi) (
            /* etsintä voidaan siis rajoittaa alikoodirjaan Sub */
            (nrst,dstNew) := FindNearestRepresentative(Sub,xi,k);
            nrst          := ActiveClusters[nrst];

        /* alkio kuuluu aktiiviseen klusteriin ja
           sentroidi on liikkunut kauemmaksi alkiosta */
        } ELSE {
            /* etsintä täytyy suorittaa koko koodikirjasta C */
            (nrst,dstNew) := FindNearestRepresentative(C,xi,j);
        }

        IF nrst <> j THEN {
            pi        := nrst;
            Distancei := dstNew;
        } ELSE {
            Distancei := dstCurr;
        }
    }

    RETURN (P,Distances);
}
```

Kuva L1.7: Osituksen muodostaminen karsittujen vertailujen hakua käyttäen.

OptimalPartition-operaatio saa parametrina *Distances*-taulukon, jossa on kunkin alkion etäisyys sentroidiinsa ennen *OptimalRepresentatives*-operaation suoritusta. *OptimalPartition*-operaatio korjaa suorituksensa aikana *Distances*-taulukkoon muuttuneet etäisyydet, mutta kuitenkin K-meansissa tarvitsee luoda taulukon ensimmäinen versio. Kuvassa L1.8 on esitetty *CalculateDistances*-operaation pseudokoodi, joka laske saamalleen ratkaisulle kunkin alkion ja sentroidinsa välisen etäisyyden ja palauttaa nämä etäisyydet *Distances*-taulukossa.

```
CalculateDistances(P,C,X) {
    totalError := 0;

    FOR i := 1 TO N DO {
        j          := pi;
        Distancei  := Dist(xi,cj,MAXINT);
        totalError := totalError + Distancei;
    }

    RETURN (Distance,totalError);
}
```

Kuva L1.8: Etäisyystaulukon muodostaminen.

Kuvassa L1.9 on esitetty pseudokoodina uudistettu versio *KMeans*-operaatiosta, joka käyttää tässä aiemmin esitettyjä karsittujen vertailujen haun mukaisia *OptimalRepresentatives*- sekä *OptimalPartition*-operaatioita. Kuvan L1.9 *KMeans*-operaatio laskee aluksi nykyiselle ratkaisulle etäisyystaulukon *Distances* käyttäen edellä esitettyä *CalculateDistances*-operaatiota. Tämän jälkeen operaatio kutsuu kullakin iteraatiolla *OptimalRepresentatives*- ja *OptimalPartition*-operaatioita peräjälkeen, kunnes saavutetaan haluttu iteraatioiden määrä *I* tai kunnes parannusta ei enää tapahdu. Parannusta ei enää tapahdu, kun kaikki sentroidit ovat staattisia eli kun aktiivisten sentroidien lukumäärä *count* on nolla.

```
KMeans(P,C,X) {
    i         := 1;
    count     := MAXINT;
    Distances := CalculateDistances(P,C,X);

    WHILE (i <= I) AND (count > 0) DO {
        (C,Active,count) := OptimalRepresentatives(P,C,X);
        (P,Distances)    := OptimalPartition(P,C,X,Active,count,
                                             Distances);
        i := i + 1;
    }

    RETURN (P,C);
}
```

Kuva L1.9: Karsittujen vertailujen hakua käyttävä K-means.

Liite 2: Deterministisen korvauksen toteutus

Luvussa 5 havaittiin sentroidin satunnaisen korvauksen tuottavan paljon ratkaisuehdokkaita, jotka eivät johda ratkaisun paranemiseen. Ratkaisuna tähän esitettiin luvussa 6 deterministisyyden lisäämistä sentroidin korvaukseen. Tässä liitteessä on esitetty käytettyjen klusterien poisto- ja luontikriteerien toteutukset (Fritzke, 1997; Fränti & Virmajoki, 2006).

Deterministinen poistettavan klusterin valinta (Fritzke)

Kuvassa L2.1 on esitetty alkiota toiseksi lähimpänä olevan sentroidin haun pseudokoodi. Haun suorittava *FindSecondNearestRepresentative*-operaatio saa parametreinaan koodikirjan C, alkion x ja alkiota lähinnä olevan sentroidin *nrst*. Toiseksi lähimmän sentroidin haku on esitetty kaavassa 6.1. Koska operaatio tarvitsee parametrinaan alkiota lähinnä olevan sentroidin, on operaation käyttö taloudellista vain silloin kuin ensisijainen ositus on tiedossa, muutoin samalla vaivalla voitaisiin laskea sekä alkiota lähinnä että toiseksi lähinnä olevat sentroidit. Operaatio hyödyntää opittuun tapaan PDS-menetelmää toiseksi lähimmän sentroidin haussa ja sen aikavaativuus on $O(MK)$, sillä suorituksen aikana tulee lasketuksi alkion x etäisyydet kaikkiin sentroideihin.

```
FindSecondNearestRepresentative(C,x,nrst) {
    secNrst := -1;
    secDst  := MAXINT;

    FOR i := 1 TO M DO {
        d := Dist(x,ci,secDst);

        IF (d < secDst) AND (i <> nrst) THEN {
            secNrst := i;
            secDst  := d;
        }
    }

    RETURN (secNrst,secDst);
}
```

Kuva L2.1: Toiseksi lähimmän sentroidin haku.

Kuvassa L2.2 on esitetty poistettavan klusterin valinnan tekevän *SelectClusterToBeSwapped*-operaation pseudokoodi. Operaatio laskee kunkin klusterin poistamisesta aiheutuvan virheen kasvun ja valitsee poistettavaksi klusteriksi sen, joka kasvattaa virhettä vähiten. Virheen kasvun laskeminen on esitetty kaavassa 6.2. Virheen kasvua laskettaessa kaikki alkiot oletetaan yksinkertaisti siirrettävän toiseksi lähimpään klusteriinsa, eikä sentroidien liikkumista oteta huomioon. Fränti ja Virmajoki (2006) ovat esittäneet edistyneempiä kaavoja klusterin poiston aiheuttaman virheen kasvun arvioimiseksi. Turhaa työtä operaatiossa vältettäisiin välittämällä *distances*-taulukko *Kmeans*-operaatiosta. Operaation aikavaativuus on $O(NMK)$, sillä kullekin aineiston alkiolle selvitetään toiseksi lähin sentroidi, jonka aikavaativuus on $O(MK)$.

```
SelectClusterToBeSwapped(P,C,X) {
    FOR i := 1 TO M DO {
        priError_i := secError_i := 0;
    }

    FOR i := 1 TO N DO {
        priClus  := p_i;
        priDst   := Dist(x,c_priClus,MAXINT);
        (secClus, secDst) := FindSecondNearestRepresentative(C,x_i,p_i);

        priError_priClus := priError_priClus + priDst;
        secError_priClus := secError_priClus + secDst;
    }

    /* valitaan klusteri, jonka poisto kasvattaa optimointifunktion
       (MSE) arvoa vähiten */
    min = -1;
    err = MAXINT;
    FOR i := 1 TO M DO {
        e := secError_i - priError_i;

        IF e < err THEN {
            min = i;
            err = e;
        }
    }

    RETURN min;
}
```

Kuva L2.2: Korvattavan sentroidin deterministinen valinta.

Kuvassa L2.3 on esitetty deterministisen sentroidin poiston suorittava *DeterministicSwap*-operaatio. Operaatio saa parametrinaan koodikirjan C, aineiston X sekä korvattavan sentroidin j. Operaatio korvaa sentroidin j satunnaisella aineiston alkiolla. Aikavaativuus on O(MK) satunnaisen aineiston alkion valitsevan *SelectRandomDataObject*-operaation vuoksi.

```
DeterministicSwap(C,X,j) {
    c_j   := SelectRandomDataObject(C,X);

    RETURN C;
}
```

Kuva L2.3: Sentroidin deterministinen poisto.

Deterministinen poisto ulottuu aivan pääoperaatiotasolle. Kuvassa L2.4 on esitetty determististä poistoa käyttävän paikallishaun pseudokoodi. Paikallishaun suorittavan operaation nimi on *DeterministicRandomLocalSearch*, sillä sen sentroidin korvaus, ja näin ollen koko naapuristofunktio, sisältää sekä deterministisiä että satunnaisia piirteitä. Erona satunnaista korvausta käyttävään paikallishakuun tässä kutsutaan aina tarvittaessa *SelectClusterToBeSwapped*-operaatiota korvattavan klusterin selvittämiseksi. Operaatiota ei kutsuta joka iteraatiolla, sillä edellisellä iteraatiolla

poistettavaksi valittua sentroidia voidaan käyttää, mikäli ratkaisu ei ole muuttunut. Determinististä poistoa käyttävän paikallishaun aikavaativuus on O(*TNMK*), kuten on myös satunnaista korvausta käyttävän version.

```
RandomDeterministicLocalSearch() {
    C := SelectRandomRepresentatives(X);
    P := OptimalPartition(C,X);
    j := SelectClusterToBeSwapped(P,C,X);

    FOR i := 1 TO T DO {
        C'      := DeterministicSwap(C,X,j);
        P'      := LocalRepartition(P,C',X,j);
        (P',C') := KMeans(P',C',X);

        IF ObjectiveFunction(P',C',X) < ObjectiveFunction(P,C,X) THEN {
            (P, C) := (P',C');
            j      := SelectClusterToBeSwapped(P,C,X);
        }
    }

    RETURN (P,C);
}
```

Kuva L2.4: Determinististä poistoa käyttävä paikallishaku.

Deterministinen poistettavan klusterin valinta (IS)

Kuvassa L2.5 on esitetty alkiota toiseksi lähimpänä olevan sentroidin haku käyttäen IS-menetelmää. *DistIS*-operaatio suorittaa saamansa alkion *x* ja klusterin *i* välisen etäisyyden ottaen huomioon, että alkion siirtyessä kyseiseen klusteriin liikkuu klusterin sentroidi alkiota kohden. Tätä laskentaa varten operaation tarvitsee tietää käsiteltävän klusterin koko. Klusterien kokoja voidaan pitää yllä jatkuvasti päivittämällä kunkin alkion liikkuessa klusterin alkiomäärä tai laskee kunkin iteraation jälkeen uusiksi.

Varsinaisen haun suorittava *FindSecondNearestIS*-operaatio saa parametreinaan koodikirjan *C*, alkion *x* ja alkiota lähinnä olevan sentroidin *nrst*. Toiseksi lähimmän sentroidin haku käyttäen IS-menetelmää on esitetty kaavassa 6.3. Operaation aikavaativuus on O(*MK*), sillä suorituksen aikana tulee lasketuksi alkion *x* etäisyydet kaikkiin sentroideihin.

Poistettavaksi klusteriksi valitaan IS-menetelmää käytettäessä klusteri, jonka poisto kasvattaa virhettä vähiten. Poistettavan klusterin valinnan periaate on esitetty kuvan L2.2 *SelectClusterToBeRemoved*-operaatiossa, jonka aikavaativuudeksi tulee O(*NMK*).

```
DistIS(x,C,i,maxDist) {
    fac := ClusterSize(i)/(ClusterSize(i)+1);
    sum := 0;
    k   := 1;

    WHILE (sum < maxDist) AND (k <= K) DO {
        sum := sum + fac*(xk-ci^k)^2;
        k   := k + 1;
    }

    RETURN sum;
}


FindSecondNearestIS(C,x,nrst) {
    secNrst := -1;
    secDst  := MAXINT;

    FOR i := 1 TO M DO {
        d := DistIS(x,C,i,secDst);

        IF (d < secDst) AND (i <> nrst) THEN {
            secNrst := i;
            secDst  := d;
        }
    }

    RETURN (secNrst,secDst);
}
```

Kuva L2.5: Toiseksi lähimmän sentroidin haku IS-menetelmää käyttäen.

Deterministinen luotavan klusterin valinta

Suurimman eli eniten virhettä tuottavan klusterin haku on esitetty pseudokoodina kuvassa L2.6. Haun suorittava *FindLargestCluster*-operaatio palauttaa eniten virhettä tuottavan klusterin lisäksi kyseisen klusterin kauimmaisen alkion. Operaation aikavaativuus on O(*NK*), sillä suorituksen aikana lasketaan alkioiden ja sentroidien väliset etäisyydet. Turhaa työtä operaatiossa vältettäisiin välittämällä *distances*-taulukko *Kmeans*-operaatiosta.

Luotavan klusterin sentroidin paikan valitsemiseen on kaksi eri kriteeriä. Luotavan klusterin sentroidi sijoitetaan joko eniten virhettä aiheuttavan klusterin kauimmaisen alkion paikalle tai klusterin sentroidin lähelle. Luotavan klusterin sentroidin laskeminen on esitetty pseudokoodina kuvassa L2.7. Valinnan aikavaativuus on O(*NK*) *FindLargestCluster*-operaation vuoksi.

Determinististen klusterin luonnin toteuttamiseksi täytyy sentroidin korvauksen tekevä operaatio muuttaa siten, että korvaava sentroidi voidaan välittää sille parametrina. Itse korvauksen suorittava operaatio keventyy ja sen aikavaativuudeksi jää vain O(*K*), joka alkion kopiointityö.

```
FindLargestCluster(P,C,X) {
    FOR i := 1 TO M DO {
        error_i   := 0;
        xtremed_i := -1;
    }

    FOR i := 1 TO N DO {
        j       := p_i;
        d       := Dist(x_i,c_j,MAXINT);
        error_j := error_j + d;

        IF xtremed_j < d THEN {
            xtremed_j := d;
            xtremei_j := i;
        }
    }

    min := -1;
    err := MAXINT;
    FOR i := 1 TO M DO {
        IF error_i < err THEN {
            min := i;
            err := error_i;
        }
    }

    RETURN (min,xtremei_min);
}
```

Kuva L2.6: Eniten virhettä tuottavan klusterin haku.

```
SelectRepresentativeToBeAddedFurthest(P,C,X) {
    (clus,xtreme) := FindLargestCluster(P,C,X);

    RETURN x_xtreme;
}


SelectRepresentativeToBeAddedNearPrev(P,C,X) {
    (clus,xtreme) := FindLargestCluster(P,C,X);

    a   := c_clus;
    k   := RANDOM(1,K);
    a_k := a_k + 1;

    RETURN a;
}
```

Kuva L2.7: Luotavan klusterin sentroidin valinta.