VEKTORIKARTTOJEN HYÖDYNTÄMINEN PAIKKATIETOSOVELLUKSISSA

Seppo Nevalainen

10. 12. 2001

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

TIIVISTELMÄ

Vektoriformaattia käytetään yleisesti karttakuvien tallennuksessa. Tietyissä tapauksissa vektorikarttojen sisältämää informaatiota kannattaa muokata erilaisilla menetelmillä, jotta informaatiota voidaan hyödyntää tehokkaasti etsittäessä ratkaisuja käsiteltäviin ongelmiin. Tässä tutkielmassa tarkastelen vektori- ja rasteriformaatteja karttakuvien tallennuksessa, vektorimuodossa olevien karttakuvien rasterointia, vektorikarttojen muuntamista verkoiksi ja lähestymistapoja verkko-ongelmien ratkaisemiseksi. Rasteroinnin yhteydessä tuon esille niitä ratkaisuja, joihin päädyimme Dynamap-projektissa toteuttaessamme rasterointia vektorikartoille.

Avainsanat: Karttakuvat, vektorigrafiikka, rasterointi, verkkoalgoritmit, lokaalihaku, geneettinen algoritmi, kauppamatkustajan ongelma.

1. JOHDANTO 1

2. DIGITAALISTEN KARTTAKUVIEN ESITYSMUODOT 4
2.1 RASTERIKARTAT 4
2.2 VEKTORIKARTAT 6
2.3 FORMAATTIEN VERTAILUA 12

3. VEKTORIKARTTOJEN MUUNTAMINEN RASTERIKSI 16
3.1 KARTTOJEN RASTEROINNIN TAVOITTEET 16
3.2. VIIVOJEN PIIRTÄMINEN 17
3.3 YMPYRÖIDEN PIIRTÄMINEN 21
3.4 PRIMITIIVIEN TÄYTTÄMINEN 23
3.5 MERKKIEN GENEROINTI 27
3.6 PRIMITIIVIEN LEIKKAAMINEN 32
3.7 ANTIALIASOINTI 33

4. VERKKO-ONGELMAT JA NIIHIN SOVELLETUT RATKAISUMENETELMÄT 36
4.1 VEKTOREIDEN MUUNTAMINEN VERKOIKSI 36
4.2 POLYNOMISESSA AJASSA RATKEAVAT ONGELMAT 40
4.3 NP-VAIKEAT ONGELMAT 45
4.4 KOMBINATORISET OPTIMOINTIONGELMAT 46
4.5 LÄHESTYMISTAPOJA KOMBINATORISIIN OPTIMOINTIONGELMIIN 50
4.6 ALGORITMEJA KOMBINATORISTEN OPTIMOINTIONGELMIEN RATKAISEMISEEN 51
4.7 GENEETTISEN ALGORITMIN SOVELTAMINEN KAUPPAMATKUSTAJAN ONGELMAAN 57

5. YHTEENVETO 59

VIITTEET 61

1. JOHDANTO

Tietoliikenneverkkojen ja tietoliikenneverkkoja hyödyntävien laitteiden nopea kehittyminen viimeisten vuosien aikana on luonut edellytykset monille uusille palveluille, jotka ovat asiakkaiden käytettävissä tietoliikenneverkkojen välityksellä. Nämä palvelut toteutetaan tavallisesti siten, että palvelun tarjoajan tietoaines sijaitsee palvelintietokoneella, jolta asiakkaat käyvät hakemassa tarvitsemaansa tietoa tietoliikenneverkon välityksellä omaan päätelaitteeseensa. Päätelaite voi olla esimerkiksi tavallinen pöytätietokone, kämmenmikro tai GSM-puhelin.

Tutkielmassa tarkastelen vektorikarttojen hyödyntämistä *paikkatietosovelluksissa*. *Paikkatieto* kuvaa kohdetta tai ilmiötä, joka on joko suoraan tai epäsuoraan paikannettu maanpinnan suhteen [8]. *Paikkatietojärjestelmät* (*GIS = Geographical Information System*) ovat yksi esimerkki paikkatietosovelluksista. Kirjallisuudessa paikkatietojärjestelmälle esitetään useita vaihtoehtoisia määritelmiä [11, 7, 23]. Tutkielmassa viittaan termillä paikkatietojärjestelmä järjestelmään, joka kykenee tallettamaan maantieteellistä tietoa tietokantaan, hakemaan kyseistä tietoa tietokannasta sekä käsittelemään sitä erilaisilla maantieteellisen tiedon käsittelyyn tarkoitetuilla operaatioilla. *Henkilökohtainen navigointi* [21] on eräs paikkatiedon soveltamisalue.

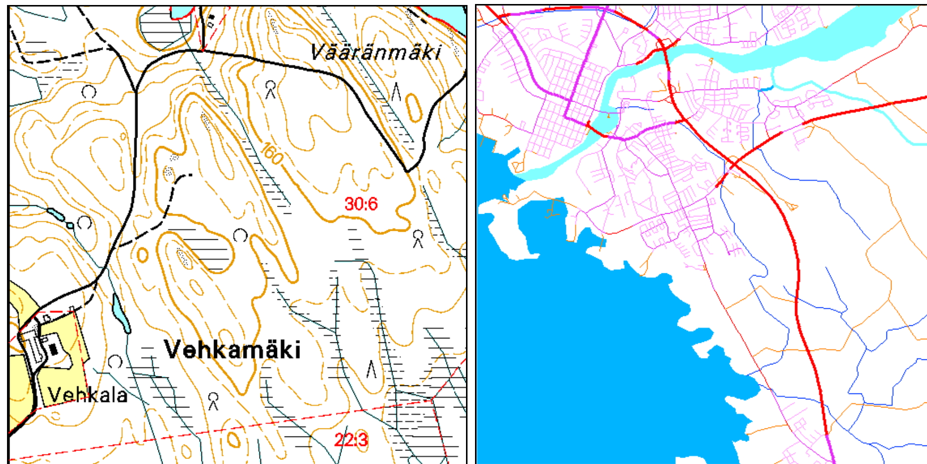
Karttatietokannoissa säilytettävät digitaaliset kartat ovat yleensä talletettu vektoriformaatissa, jossa maantieteelliset objektit esitetään 2-ulotteisten *vektoreiden* avulla. Teoksessa [27] esitetään vektorille seuraava määritelmä: n-ulotteinen vektori x on reaaliluvuista $\xi_1, \xi_2, \ldots, \xi_n$ muodostettu kokoelma, joka on järjestetty muotoon

$$x = \begin{pmatrix} \xi_1 \\ \xi_2 \\ . \\ . \\ . \\ \xi_n \end{pmatrix}.$$

Lukuja $\xi_1, \xi_2, \ldots, \xi_n$ kutsutaan x:n komponenteiksi.

Tässä tutkielmassa tarkastelen sitä, kuinka vektoreiden sisältämää informaatiota voidaan hyödyntää erilaisissa paikkasovelluksissa. Perehdyn tutkielmassa kolmeen osaongelmaan: Karttojen tallennukseen rasteri- ja vektoriformaateilla, vektorikuvan muuntamiseen rasterikuvaksi ja vektorikarttoihin liittyvien kysymysten ratkaisemiseen verkkoalgoritmien avulla. Luvussa kaksi käyn läpi vektori- ja rasteriformaattien ominaispiirteitä sekä tarkastelen sitä, kuinka kyseisiä formaatteja hyödynnetään maantieteellisen tiedon säilyttämisessä. Lisäksi vertailen vektori- ja rasteriformaatteja keskenään. Kolmannessa luvussa käsittelen vektorikarttojen rasterointia ja esittelen yksinkertaisia menetelmiä perusprimitiivien rasteroimiseksi. Tuon luvussa esille Dynamap-projektissa toteuttamiamme käytännön ratkaisuja primitiivien rasteroimiseksi [30]. Neljännessä luvussa tarkastelen vektorikarttoihin liittyvien kysymysten ratkaisemista verkkoalgoritmien avulla. Käsittelen niitä kysymyksiä ja ongelmia, joita liittyy vektoreiden muuntamiseen verkoiksi sekä tarkastelen erilaisia lähestymistapoja verkko-ongelmien ratkaisemiseksi. Mikäli vektorikarttojen mahdollisesti sisältämät erilaiset verkostot, kuten tie- ja rautatieverkostot, kyetään muuntamaan tyydyttävästi verkoiksi, voidaan paikkatietosovelluksiin liittyvien ongelmien ratkaisemisessa tehokkaasti hyödyntää verkkoja ja verkko-ongelmia käsittelevää laajaa kirjallisuutta [2, 5, 9, 13, 25, 28].

Tämän tutkielman aikana käytän esimerkkikarttoina pääasiassa kahta karttaa, jotka olivat testiaineistona toteuttaessamme rasterointia Dynamap-projektissa. Karttakuva 1 on talletettu *Shapefile*-formaatissa [12] ja esittää otetta suuremmasta kartasta, joka on talletettu maanmittauslaitoksen maastotietokantaan [20]. Karttakuva 2 on talletettu *SVF*-formaatissa [29] ja esittää otetta Joensuun kaupunkikartasta. Nämä kaksi esimerkkikarttaa on esitetty kuvassa 1.1.



Kuva 1.1: Karttakuva 1 (vasemmalla) ja Karttakuva 2 (oikealla).

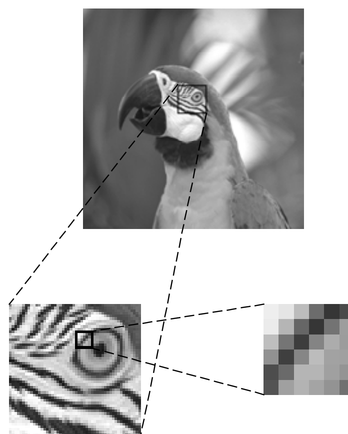
2. DIGITAALISTEN KARTTAKUVIEN ESITYSMUODOT

Teoksessa [7] *kartta* on määritelty seuraavasti: kartta on kokoelma digitaalista informaatiota, joka on kerätty tietyltä alueelta maapallon pinnalta. *Karttakuva* on kartasta tuotettu visuaalinen esitys. Viittaan jatkossa termillä karttakuva digitaaliseen karttakuvaan. *Digitaalisen* ja *analogisen* karttakuvan välisiä eroja on tarkasteltu teoksessa [23].

Digitaalisessa muodossa olevan kuvan tallennusformaatit jaetaan perinteisesti kahteen ryhmään: vektoriformaatteihin sekä rasteriformaatteihin. Tässä luvussa esitän yhteenvedon rasteri- ja vektoriformaattiin liittyvistä yleisistä ominaisuuksista ja tarkastelen kyseisiä formaatteja karttakuvien tallennuksessa. Mainitsen luvussa lyhyesti yleisimpiä rasteri- ja vektoriformaatteja sekä esittelen yksityiskohtaisemmin *Shapefile*- ja SVF-vektoriformaatteja [12, 29].

2.1 RASTERIKARTAT

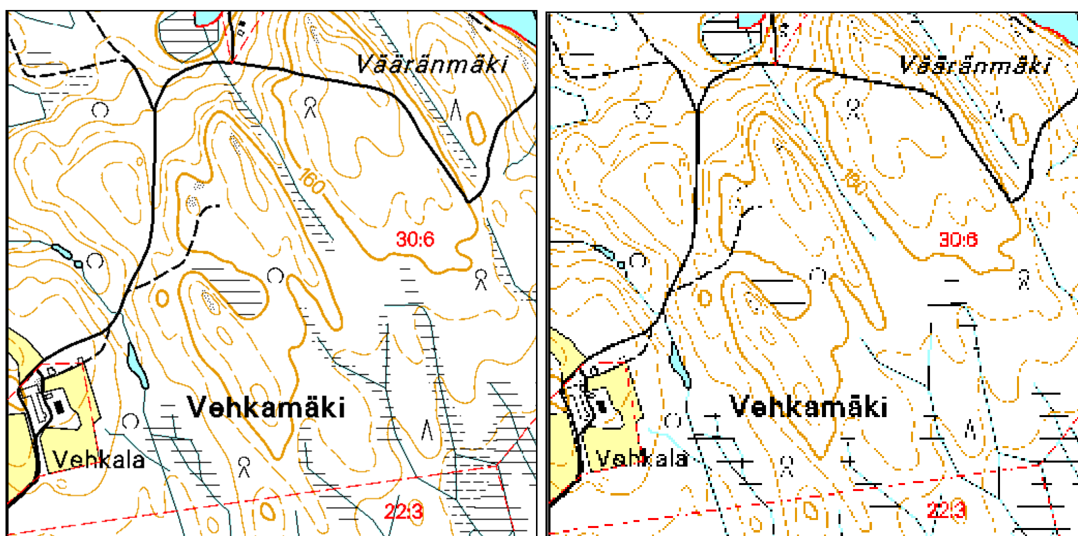
Rasteriformaatit tallentavat kuvan yksittäisistä kuvapisteistä eli pikseleistä muodostettuna matriisina (kuva 2.1). Rasterikuvan *resoluutio* määritellään tämän matriisin rivien ja sarakkeiden määrän avulla. Rasterikuvan tarkkuus riippuu pitkälti siitä, kuinka suuri kuvan resoluutio on. Rasterikuvan tarkkuuteen vaikuttaa myös kuvan sisältämien värisävyjen määrä (kuva 2.2). Valtaosa nykyisistä näyttö- ja tulostuslaitteista esittää käsittelemänsä kuvan rasterimuodossa.



Kuva 2.1: Rasterikuva ja yksittäiset pikselit.

Kuva 2.2: Kolmella eri harmaasävyjen määrällä (64, 16, 4)
talletettu rasterikuva.

Rasterikuva on sidottu siihen resoluutioon, jolla se talletetaan. Kuvaa voi tarkastella eri resoluutioilla, mutta kuvan laatu kärsii mikäli tarkastelussa käytetty resoluutio eroaa kuvan tallettamiseen käytetystä resoluutiosta. Mikäli kuvan resoluutiota kasvatetaan, joudutaan kuvaan lisäämään uusia pikseleitä. Mikäli resoluutiota pienennetään, joudutaan osa kuvan sisältämistä pikseleistä poistamaan kuvasta. Laadun huononeminen on nähtävissä kuvassa 2.3 esitetyssä rasterikuvassa, jota on tarkasteltu kuvan tallennukseen käytetyllä resoluutiolla (vasemmalla) sekä resoluutiolla, joka on kaksi kertaa tallennusresoluutiota pienempi (oikealla).



Kuva 2.3: Karttakuva 1 tarkasteltuna kahdella eri resoluutiolla.

Talletettaessa karttoja rasteriformaatissa yksittäisiä maantieteellisiä kohteita ei talleteta erikseen, vaan ne esitetään toisiinsa liittyvien kuvapisteiden avulla. Mikäli karttojen sisältämiin kohteisiin halutaan liittää ominaisuustietoa, joudutaan se tallettamaan erilliseen tiedostoon. Ominaisuustieto voidaan liittää rasterikartan kohteeseen muodostamalla linkki kyseistä kohdetta esittävien pikseleiden ja ominaisuustiedon välille. Edellä kuvattu menettelytapa on kuitenkin epäkäytännöllinen, sillä muutettaessa rasterikartan resoluutiota myös kartan kohteita esittävien pikseleiden sijainti muuttuu ja ominaisuustieto joudutaan linkittämään tällöin uudelleen.

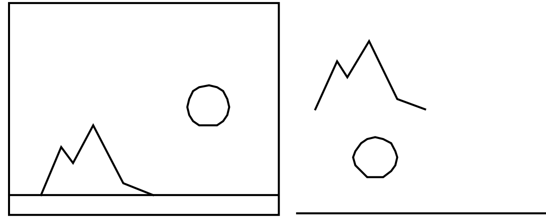
Karttojen sisältämät kohteet voivat sijaita toistensa kanssa päällekkäin. Esimerkiksi tie kulkee sillan kohdalla vesialueen yläpuolella. Mikäli kartta on rasterimuodossa, ainoastaan yksi päällekkäisistä kohteista voidaan esittää kokonaisuudessaan yhdessä karttakuvassa. Tämä johtuu siitä, että yhteen kuvapisteeseen eli kartan koordinaattiin voidaan sijoittaa ainoastaan yksi arvo.

Kuvien tallennukseen käytetään mm. seuraavia rasteriformaatteja:

- GIF (*graphics interchange format*) [6]
- JPEG (*Joint Photographic Express Group*) [6]
- TIFF (*tagged image file format*) [6]
- PBM (*portable bitmap*) [6]

2.2 VEKTORIKARTAT

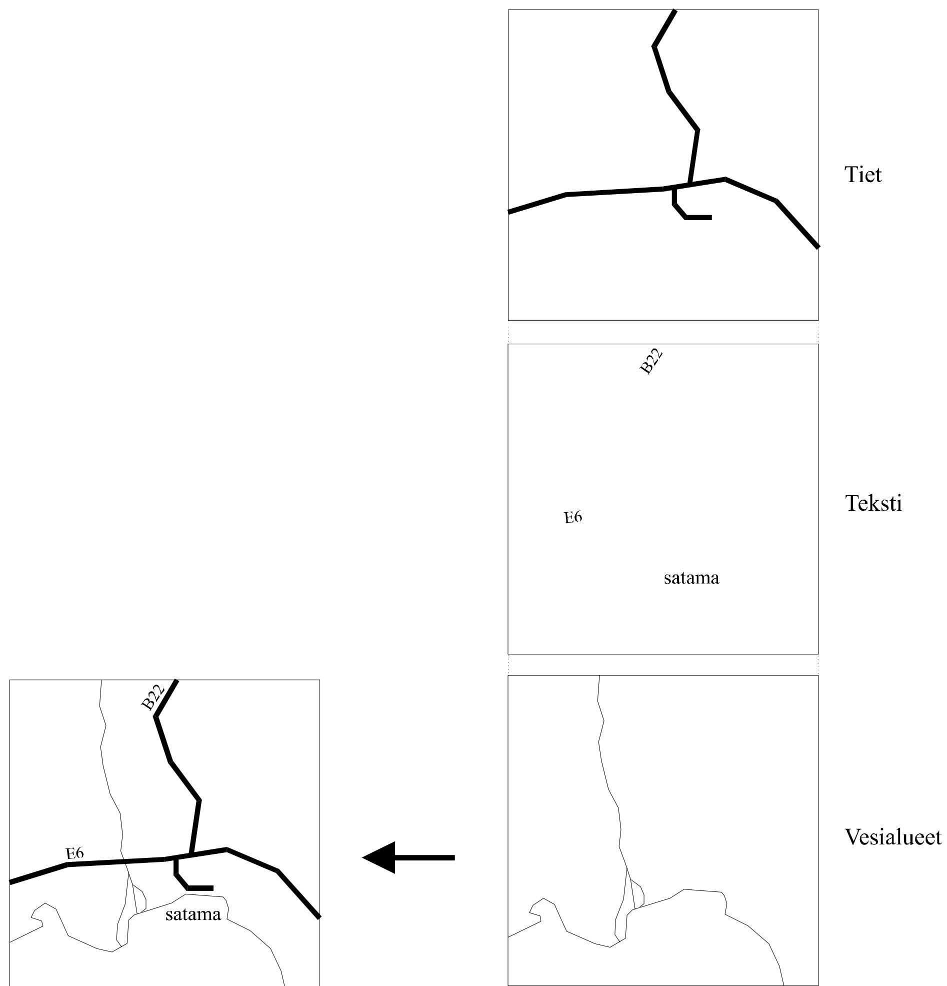
Vektorimuodossa oleva kuva koostuu *primitiiveistä* [14]. Perusprimitiivejä ovat *piste*, *viiva* sekä *monikulmio*. Tämän lisäksi erilaisissa määritelmissä primitiivien joukkoon lasketaan mm. *moniviiva (polyline)*, *suorakulmio*, *kaari* ja *ympyrä* (kuva 2.4). Edellä esitellyt primitiivit ovat kaksiulotteisia primitiivejä, joiden lisäksi kolmiulotteisille esityksille on määritelty omia erillisiä primitiivejä. Kaikki primitiivit voidaan kuvata yksittäisten vektoreiden avulla.



Kuva 2.4: Vektorikuva ja sen sisältämät primitiivit.
Primitiivit ovat (ylhäältä alas) moniviiva, ympyrä ja viiva.

Vektorikuvan sisältämät kohteet esitetään primitiivien avulla joko suoraan yhdellä tarkoitukseen sopivalla primitiivillä kuten horisontti viivan avulla tai muodostamalla kyseinen kohde joukosta sopivasti ryhmiteltyjä primitiivejä. Esimerkiksi auto voidaan esittää kahden ympyrän sekä erilaisten viivojen ja monikulmioiden avulla.

Määriteltäessä karttakuvia vektoriformaattien avulla pyritään yksittäiset maantieteelliset kohteet esittämään yksittäisten primitiivien avulla. Tällöin jokainen maantieteellinen kohde on erikseen määritelty ja kyseiseen kohteeseen liittyvä ominaisuustieto, jota säilytetään yleensä erillisessä tietokantatiedostossa, voidaan yhdistää yksiselitteisesti vastaavaan primitiiviin. Useat vektoriformaatit mahdollistavat kuvan jaottelun useisiin eri kerroksiin, jolloin maantieteelliset kohteet voidaan ryhmitellä omiksi *semanttisiksi kerroksiksi* kuvan sisällä (kuva 2.5). Tämä tarkoittaa käytännössä sitä, että esimerkiksi vesistöjä, peltoja ja korkeuskäyriä kuvaavat primitiivit voidaan tallettaa omiksi ryhmikseen vektorikuvan sisällä ja sopivan välineen avulla poimia kuvasta erilleen. Toteutuksessamme suoritimme vektorikuvan semanttisten kerrosten purkamisen erilleen Shapefile- ja SVF-formaatissa oleville kuville.



Kuva 2.5: Karttakuva ja sen sisältämät semanttiset kerrokset.

Kuvien tallennukseen käytetään mm. seuraavia vektoriformaatteja:

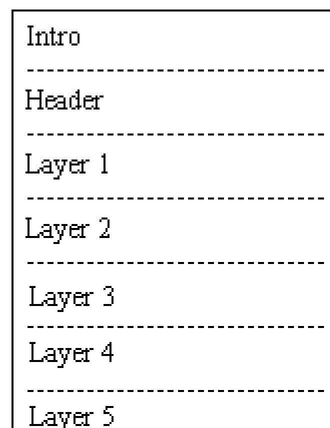
- SVG (*Scalable Vector Graphics*) [32]
- MIF (*MapInfo Interchange Format*) [22]
- SVF (*Simple Vector Format*) [29]
- SHP (*ESRI Shapefile*) [12]

2.2.1 SIMPLE VECTOR FORMAT (SVF)

SVF on *SoftSource*:n ja NCSA:n yhdessä kehittämä vektoriformaatti, joka on suunniteltu ensisijaisesti internet-käyttöä varten. SVF tukee hyperlinkkejä sekä kuvan esittämistä erillisinä

kerroksina. Mahdollisuus purkaa kuva erillisiksi kerroksiksi on yksi syy siihen, miksi SVF sopii hyvin myös karttakuvien tallennusformaatiksi. Toinen karttakuvien esittämisen kannalta tärkeä ominaisuus SVF-formaatissa on mahdollisuus määrittää kuvalle mittakaava ja *kiintopiste* (*base point*). Tämän ominaisuuden avulla SVF-formaatin avulla voidaan esittää todellisiin koordinaatteihin sidottua informaatiota.

SVF-formaatti tallettaa kaiken kuvan sisältämän tiedon yhteen SVF-päätteiseen tiedostoon. Tiedosto koostuu kolmesta osasta, jotka ovat *tunnus* (*intro)*, *otsake* (*header*) ja *runko* (*body*). Tunnus sisältää merkkijonon ”SVF” sekä versionumeron. Otsakkeessa määritellään koko kuvaa koskevat ominaisuudet, kuten *kerrostaulukko* (*LayerTable*), *väritaulukko* (*ColorTable*) ja *taustaväri* (*BackgroundColor*). Runkoon talletetaan kuvan sisältämät primitiivit sekä niihin liittyvä ominaisuustieto erillisiin kerroksiin jaoteltuna. SVF-tiedostossa ei ole kiinteää rakennetta, vaan tiedon tyyppi ja merkitys ilmaistaan *lippujen* (*tag*) avulla. Kuvassa 2.6 on havainnollistettu SVF-tiedoston rakennetta.



Kuva 2.6: SVF-tiedoston mahdollinen rakenne.

SVF-formaatti mahdollistaa tiedon tallettamisen seitsemän perusprimitiivin avulla. Nämä perusprimitiivit ovat piste, viiva, moniviiva, suorakulmio, ympyrä, kaari ja *Bezier-kaari*. Monikulmio talletetaan moniviivana, jonka päätepiste on sama kuin aloituspiste. Viivasta ja moniviivasta on olemassa myös suhteellinen muoto, jolloin primitiivin koordinaatit esitetään siirtyminä suhteellisten koordinaattien mukaan. SVF-formaatti käyttää tiedon tallentamiseen

yksi- tai kaksitavuisia kokonaislukuja. Liukulukuja käytetään ainoastaan kuvan mittakaavan tallentamisessa. Kuvassa 2.7 on esitetty ote SVF-syntaksista.

```
<SVF file> = <intro> + <header> + <body>
<intro> = "SVFv" + <version number> + NullTerminator
…
<header> = {<header item>}
<body> = {<command>}
…
<header item> = <name> | <extents> | <layer table> | <color table> |
 <background> | <transparent> | <notify table> | <imagewidth> | <transform> | <flags>
<name> = Name + <null terminated string>
<extents> = Extents + <xmin> + <ymin> + <xmax> + <ymax>
<layer table> = LayerTable + <number of entries> + {<state> + <null terminated string>}
…
<command> = <point> | <moveto> | <lineto> | <polyline> | <relmoveto> | <rellineto> |
            <relpolyline> | <rectangle> | <circle> | <arc> | <bezier> | <text> | <text height> |
            <setcolor> | <setlayer> | <penwidth> | <fillmode> | <data> | <set1dlinkinfo> |
            <set2dlinkinfo> | <invisible>
<point> = Point + <x> + <y>
<moveto> = MoveTo + <x> + <y>
<lineto> = LineTo + <x> + <y>
<polyline> = Polyline + <number of points> + {<x> + <y>}
<relmoveto> = RelMoveTo + <xoffset> + <yoffset>
…
<setcolor> = SetColor + <color>
<setlayer> = SetLayer + <layer>
<penwidth> = SetPenWidth + <width>
```

Kuva 2.7: Ote SVF-formaatin syntaksista.

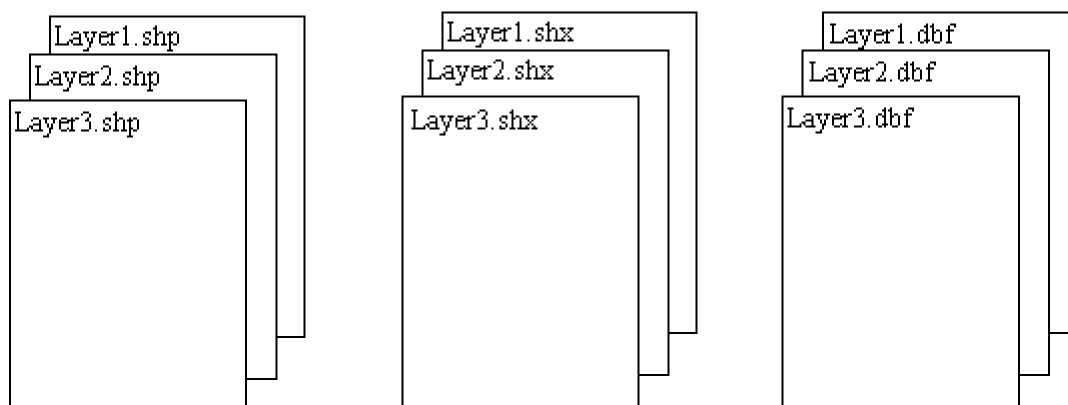
2.2.2 SHAPEFILE (SHP)

Shapefile on ESRI:n (*Environmental Systems Research Institute, Inc*) vektoriformaatti, joka on kehitetty erityisesti maantieteellisen tiedon ja kartta-aineistojen tallentamista varten.

Yksittäinen Shapefile-muodossa talletettu kuva koostuu kolmesta erillisestä tiedostosta, jotka ovat seuraavat:

- Päätiedosto (main file, *.shp)
- Indeksitiedosto (index file, *.shx)
- Tietokantatiedosto (dBASE table, *.dbf)

Päätiedostoon talletetaan graafiset objektit eli primitiivit. Yksi tiedosto voi sisältää ainoastaan yhdenlaisia primitiivejä. Indeksitiedostoon talletetaan tieto siitä, missä kohtaa päätiedostoa mikäkin primitiivejä vastaava tietue sijaitsee, sekä tieto sen pituudesta. Tietokantatiedosto sisältää graafisiin objekteihin mahdollisesti liittyvän ominaisuustiedon ja esitysasuun liittyvän tiedon. Tiedoston nimen ja tietueiden järjestyksen tulee vastata vastaavan indeksitiedoston nimeä ja tietueiden järjestystä. Käytettäessä Shapefile -formaattia karttakuvaan liittyvät eri kerrokset talletetaan erillisiin tiedostoihin, toisin kuin SVF-formaatissa. Kuvassa 2.8 on havainnollistettu Shapefile-formaatin tapaa tallettaa kuvan sisältämää informaatiota.



Kuva 2.8: Shapefile-tiedostot kuvalle, joka koostuu kolmesta eri kerroksesta.

Shapefile tukee kolmea perusprimitiiviä: pistettä, moniviivaa ja ympyrää, sekä näiden eräitä variaatioita. Kaikista primitiiveistä talletetaan primitiivin tyyppi sekä primitiivin koordinaatit. Piste-tietue sisältää ainoastaan tyypin tunnisteen sekä yhden *x*- ja *y*-koordinaatin. Shapefile mahdollistaa moniviivat, jotka koostuvat erillisistä osista. Tästä syystä moniviiva-tietueen

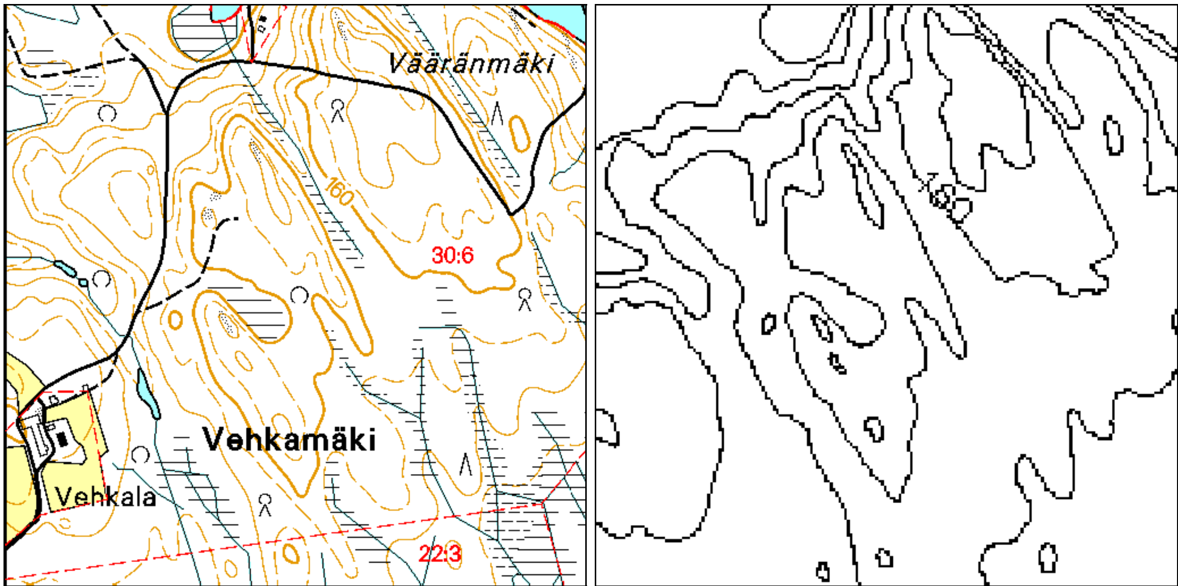
rakenne on seuraava: tyypin tunniste, primitiiviä rajoittava suorakulmio (Bounding box), osien lukumäärä, pisteiden lukumäärä, Osat-taulukko ja Pisteet-taulukko. Osat-taulukko sisältää indeksit, jotka osoittavat moniviivan eri osien ensimmäisten koordinaattien sijainnin Pisteet-taulukossa. Pisteet-taulukko sisältää moniviivan kaikki koordinaatit. Kuvassa 2.9 on esitetty moniviiva-tietueen rakenne. Monikulmio-tietueen rakenne on identtinen moniviiva-tietueen rakenteen kanssa. Shapefile käyttää tiedon tallentamiseen 32-bittisiä etumerkillisiä kokonaislukuja ja 64-bittisiä etumerkillisiä liukulukuja.

```
PolyLine
{
    Double[4]            Box              // Bounding Box
    Integer              NumParts         // Number of Parts
    Integer              NumPoints        // Total Number of Points
    Integer[NumParts]    Parts            // Index to First Point in Part
    Point[NumPoints]     Points           // Points for All Parts
}
```

Kuva 2.9: Moniviiva-tietueen rakenne.

2.3 FORMAATTIEN VERTAILUA

Tässä luvussa vertailen vektori- ja rasteriformaattien etuja ja haittoja keskittyen karttakuvien tallennukseen. Kuvassa 2.10 on esitetty esimerkkikartta 1 sekä siitä karkealla resoluutiolla rasteroitu korkeuskäyrät sisältävä kerros. Vektoriformaattina on Shapefile ja rasteriformaattina GIF.



Kuva 2.10: Esimerkkikuva 1 sekä siitä rasteroitu yksittäinen kerros.

Vektorimuodossa olevan kuvan tärkeimpiä etuja on riippumattomuus mittakaavasta. Koska kuvan sisältämät kohteet on määritelty koordinaattipisteiden avulla, voidaan kuvaa tarkastella eri mittakaavoissa ilman, että kuvan laatuun tulisi muutoksia. Rasterikuvien laatu sen sijaan heikkenee, mikäli kuvaa tarkastellaan eri mittakaavassa kuin missä se on talletettu.

Vektorikuvan piirtäminen näytölle ja tulostaminen paperille ovat aikaa vieviä operaatioita, koska vektorikuvan sisältämät tiedot joudutaan muuntamaan rasterimuotoon. Muuntamisen tarve johtuu siitä, että valtaosa tällä hetkellä käytössä olevista näyttö- ja tulostusvälineistä hyödyntää rasteritekniikkaa.

Vektoriformaatissa yksittäiset kuvan kohteet voidaan määritellä erillisten primitiivien avulla, minkä vuoksi ominaisuustiedon liittäminen kohteisiin on suoraviivaista ja helppoa. Erityisesti karttakuvien kohdalla tämä on tärkeä etu, sillä suuri osa karttakuvien sisältämän informaation käsittelystä perustuu maantieteellisiin kohteisiin ja niihin liittyvään ominaisuustietoon. Rasteriformaatissa yksittäisiä kuvan sisältämiä kohteita ei määritellä erikseen, vaan ne esitetään kuvapisteiden avulla.

Vektoriformaatti sopii rasteriformaattia huonommin epäsäännöllisten kohteiden esittämiseen. Kohteet voivat olla epäsäännöllisiä esimerkiksi muodon tai värityksen osalta. Vektoriformaatissa jokainen kohde määritellään erikseen kohteen muodostavien koordinaattien avulla. Mitä epäsäännöllisempi kohde on muodoltaan, sitä useampia koordinaattipareja joudutaan käyttämään sen esittämiseen. Rasteriformaatissa kohteen rajoja ei erikseen määritellä ja tästä syystä kohteen mahdollinen epäsäännöllisyys ei lisää siihen käytettyjen pikseleiden määrää. Väritykseltään epäsäännöllisten kohteiden esittäminen vektoriformaatin avulla on ongelmallista, koska kyseisessä formaatissa kohdetta esittävän primitiivin täyttöväri on yhteinen koko primitiivin alueella. Tämä tarkoittaa sitä, että jokaista eri värisävyä varten pitää määritellä oma primitiivi, mikä kasvattaa nopeasti kuvan esittämiseen vaadittavien primitiivien kokonaismäärää ja täten myös kuvan vaatimaa tilaa. Rasterikuvassa erilaiset värisävyt voidaan tallettaa yksittäisiin kuvapisteisiin, jotka esittävät väritettävää primitiiviä.

Rasteriformaatit ovat yleisempiä kuin vektoriformaatit ja tästä syystä rasterimuodossa olevat kuvatiedostot ovat helpommin siirrettävissä. Rasterimuodossa olevat kuvat vievät keskimäärin vähemmän tilaa kuin vektorimuodossa olevat kuvat, etenkin jos kuvan tallennukseen käytetään sellaista rasteriformaattia, joka hyödyntää kuvan tiivistämismenetelmiä [31].

Dynamap-projektissa pyritään toteuttamaan menetelmiä, joiden avulla karttoja voidaan käyttää mobiileissa ympäristöissä pienillä laskennallisilla ja tilallisilla resursseilla. Toteuttamamme ohjelman tehtävänä on purkaa vektorikartta erillisiksi binäärisiksi kerroksiksi, jotka konvertoidaan rasterikuvaksi ja ominaisuustiedon sisältäväksi projektitiedostoksi. Tällöin kartta voidaan pakata alkuperäistä kuvaa pienempään tilaan. Dynamap-projektissa pakkaus suoritetaan *MISS-formaatin* [16, 31] avulla. MISS-formaatti on Joensuun Yliopistossa kehitetty rasterikuvien tallennusformaatti, joka käyttää tiivistysmenetelmänään kontekstipohjaista tilastollista mallinnusta ja aritmeettista koodausta. MISS-formaatti tukee seuraavia karttoihin liittyviä ominaisuuksia:

- Tiivis tallennuskoko
- Kartan esittäminen eri resoluutioilla (*zoomaus*)
- Kyky nopeaan kartan *vieritykseen* (*pannaus*)

Taulukossa 2.1 on esitetty yhden projektissa käytetyn karttakuvan tilantarve ennen rasterointia ja rasteroinnin jälkeen. Shapefile- muodossa kartta koostuu seuraavista kolmentyyppisistä tiedostoista:

1. Päätiedostot (shp) — *8,838,968* tavua. Nämä tiedostot sisältävät karttakuvan graafiset primitiivit.
2. Tietokantatiedostot (dbf) — *8,525,716*. Nämä tiedostot sisältävät primitiiveihin liittyvän ominaisuustiedon.
3. Indeksitiedostot (shx) — *242,840*. Indeksitiedot nopeampaa käsittelyä varten. Toteutuksemme ei huomioi tätä informaatiota.

Tiivistämättömän kartan kokonaistilantarve Shapefile- formaatissa on **17,607,524** tavua. ZIP-menetelmällä tiivistetyn kartan kokonaistilantarve on **4,542,128** tavua.

Taulukko 2.1: Toteutuksessamme rasteroidun kuvan viemä tila vektori- ja rasteriformaateissa.

Mittakaava	**Resoluutio (metrejä/pikseli**	**Bittikartan koko**	**Tiivistämätön koko**	**MISS**	**Tiivistämis-suhde**
1:8000	2	5000x5000	12,500,575	784,592	16.7

3. VEKTORIKARTTOJEN MUUNTAMINEN RASTERIKSI

Rasteroinnilla tarkoitetaan vektorimuodossa olevan grafiikan muuntamista rasterimuotoon. Rasteroinnin tavoitteena on muuntaa vektorikuvan sisältämät primitiivit rasterikuvassa käytetyiksi kuvapisteiksi siten, että rasterimuodossa olevan kuvan visuaalinen ulkoasu säilyy mahdollisimman identtisenä vektorimuodossa olevan kuvan ulkoasun kanssa.

Vektorimuotoisen grafiikan muuntaminen rasterimuotoon on kohtalaisen suoraviivainen tehtävä ja siihen on kehitelty useita tehokkaita algoritmeja [14]. Tässä luvussa tarkastelen aluksi karttakuvien rasteroinnin tavoitteita, minkä jälkeen käsittelen primitiivien rasterointiin kehiteltyjä perusalgoritmeja. Samalla esittelen niitä käytännön ratkaisuja, joihin päädyimme toteuttaessamme rasterointia.

3.1 KARTTOJEN RASTEROINNIN TAVOITTEET

Digitaalisessa muodossa olevia karttoja voidaan säilyttää paikkatietojärjestelmissä. Useat paikkatietojärjestelmät mahdollistavat karttojen tallentamisen sekä rasteri- että vektorimuodossa, ja sisältävät menetelmät, joilla karttoja voidaan halutessa muuntaa vektorista rasteriin [7]. Karttojen sisältämän informaation erilaiset käsittelytavat soveltuvat eri tavalla rasteri- ja vektorimuodossa oleville kartoille. Mikäli tiettyä vektorimuodossa olevaa kartta-aineistoa halutaan toistuvasti käsitellä menetelmillä, jotka ovat helppoja suorittaa rasterikartoille, on tällöin järkevää ensin rasteroida kartta-aineisto.

Henkilökohtaiseen navigointiin liittyen on karttakuvien rasteroinnille olemassa erityinen tarve, jonka aiheuttavat päätelaitteiden rajalliset tilaresurssit sekä pyrkimys mahdollisimman nopeaan tiedonsiirtoon palvelimen ja päätelaitteen välillä. Mikäli palvelimelta päätelaitteille liikkuvat karttakuvat rasteroidaan ja pakataan, tiedonsiirto on nopeampaa ja päätelaitteella voidaan säilyttää yhtä aikaa useampia karttakuvia, kuin jos kuvat olisivat vektorimuodossa ja pakkaamattomia.

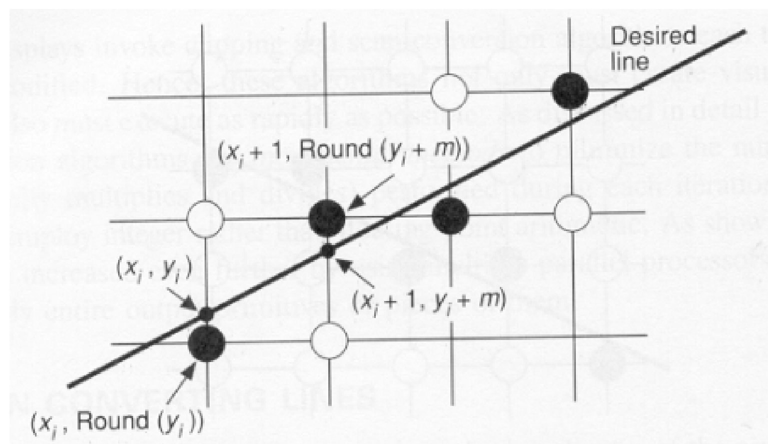
3.2. VIIVOJEN PIIRTÄMINEN

Tässä luvussa esiteltävät viivanpiirtoalgoritmit olettavat piirrettävän viivan kulmakertoimen suuruuden olevan välillä -1 ja 1. Viivat, joiden kulman suuruus ei ole kyseisellä välillä, voidaan piirtää luvussa esiteltävillä algoritmeilla symmetriaa hyödyntäen.

Piirrettäessä viivoja rasterimuotoon, tulisi pyrkiä seuraaviin tavoitteisiin [14]:

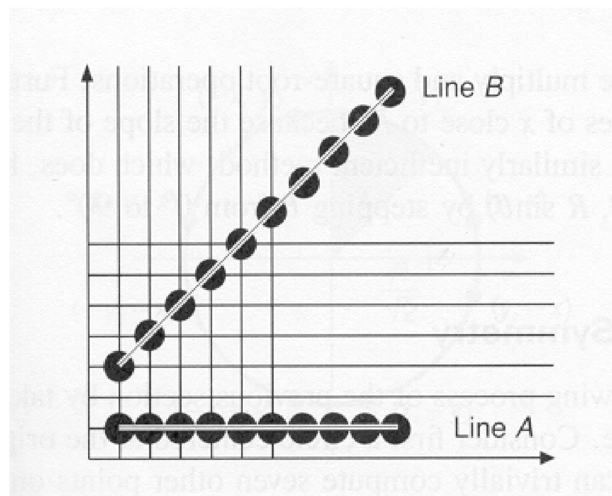
1. Suorien viivojen tulisi näyttää suorilta myös rasteriesityksessä.
2. Viivojen tulisi, mikäli mahdollista, kulkea päätepisteiden kautta.
3. Kaikki viivat tulisi piirtää yhtäläisellä intensiteetillä pituudesta ja suunnasta riippumatta.
4. Viivat tulisi piirtää niin nopeasti kuin mahdollista.

Viivojen piirtämisalgoritmit paikallistavat ideaalisen linjan lähellä sijaitsevat pikselit, joista rasterimuodossa oleva viiva muodostetaan (kuva 3.1). Yksinkertaisin strategia viivojen piirtämiseen on käyttää suoraviivaista *kasvatusalgoritmia* (*basic incremental algorithm*) [14], jonka perusidea on seuraava: kasvatetaan *x*-koordinaattia yksi askel kerrallaan viivan alkupisteestä loppupisteeseen. Lasketaan *y*-koordinaatti jokaista *x*-koordinaatin askelta kohden suoran yhtälön avulla. "Käännetään päälle" pikselit, jotka ovat lähimpänä saatuja y-koordinaatteja.



Kuva 3.1: ideaalinen viiva ja sitä lähimpänä olevat pikselit [14].

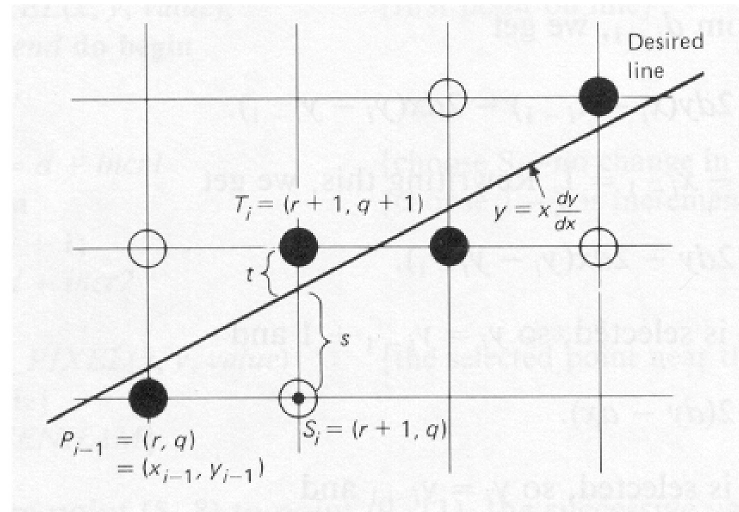
Eräs tähän lähestymistapaan liittyvä ongelma on viivan vaihteleva intensiteetti, mitä on havainnollistettu kuvassa 3.2. Ongelma syntyy siitä, että edellä esitelty suoraviivainen kasvatusalgoritmi huomioi suoran pituuden ainoastaan x-akselin suhteen. Tällöin eripituisten viivojen piirtämiseen käytetään joissain tapauksissa yhtä paljon pikseleitä.



Kuva 3.2: viivojen piirtämiseen liittyvä vaihtelevan intensiteetin ongelma [14].

Suoraviivainen kasvatusalgoritmi ei kykene täyttämään luvun alussa mainittua yhtäläisen intensiteetin vaatimusta. Tämän lisäksi suoraviivainen kasvatusalgoritmi on tehoton ja hidas, koska se perustuu murtolukulaskentaan ja kutsuu pyöristysfunktiota jokaista y-koordinaatin laskemista varten.

Bresenham esitteli vuonna 1965 viivanpiirtoalgoritmin [15], joka hyödyntää pelkästään kokonaislukulaskentaa ja välttää täten myös pyöristysfunktion käytön. Bresenhamin algoritmi mahdollistaa seuraavan y-koordinaatin laskemisen yksinkertaisen kasvatusmenetelmän avulla, hyödyntämällä edellisen y-koordinaatin kohdalla suoritettua laskentaa. Bresenhamin algoritmin toimintaa on havainnollistettu kuvassa 3.3. Toteuttamamme viivojen piirtämisen suorittava funktio hyödyntää Bresenhamin algoritmia.



Kuva 3.3: Esimerkki Bresenhamin algoritmin toiminnasta [15].

Bresenhamin algoritmin toimintaperiaate on seuraava: piirrettävän pisteen päätepisteet ovat ($x1$, $y1$) ja ($x2$, $y2$). Oletetaan, että ensimmäinen piste on lähempänä origoa ja muunnetaan molemmat pisteet arvoilla ($-x1$, $-y1$). Tällöin viivan pisteet ovat (0, 0) ja (dx, dy), joista $dx = x2-x1$ ja $dy = y2-y1$. Viivan yhtälö on nyt $y = (dx/dy)x$. Muunnoksen jälkeen pisteen P_{i-1} koordinaatit esitetään parina (r, q). Piste P_{i-1} on tarkasteltavaa pistettä edellinen piste. Tällöin $S_i = (r+1, q)$ ja $T_i = (r+1, q+1)$.

Kuten kuvasta 3.3 voidaan havaita,

$$s = dy/dx(r+1)-q,\ t = q+1-dy/dx(r+1). \tag{3.1}$$

Koska dx on positiivinen, voidaan epäyhtälöä $dx(s-t) < 0$ käyttää testinä pisteen S_i valitsemiselle. Määritellään tämä epäyhtälö symbolilla d_i; tällöin

$$d_i = 2(r*dy-q*dx)+2dy-dx. \tag{3.2}$$

Mikäli $d_i \geq 0$, piste T_i valitaan. Tällöin $y_i = y_{i-1}+1$ ja

$$d_{i+1} = d_i+2(dy-dx). \tag{3.3}$$

Mikäli $d_i < 0$, piste S_i valitaan. Tällöin $y_i = y_{i-1}$ ja

$$d_{i+1} = d_i + 2dy. \tag{3.4}$$

Yllä esiteltyjen kaavojen avulla voidaan d_{i+1} laskea iteratiivisesti edellisen arvon d_i avulla. Valinta pisteiden S_i ja T_i välillä voidaan suorittaa yksinkertaisen vertailun avulla. Aloitusarvo d_i löydetään ratkaisemalla yhtälö (3.2) arvolla i = 1, kun tiedämme, että (x_0, y_0) = (0,0). Tällöin

$$d_1 = 2dy - dx. \tag{3.5}$$

Bresenhamin algoritmi on esitetty kuvassa 3.4.

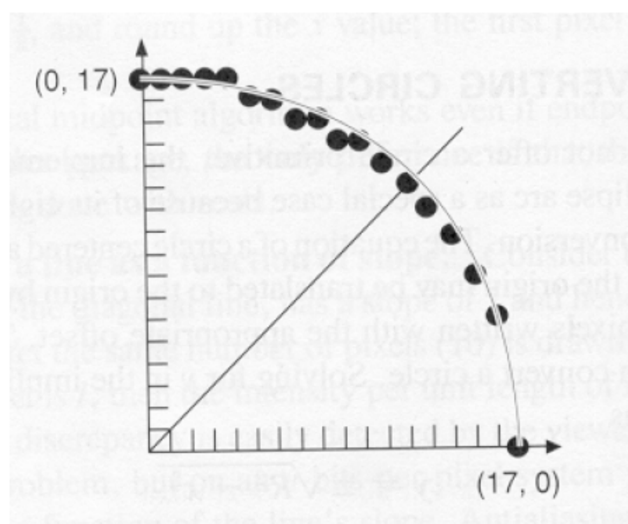
```
Procedure BRESENHAM(x1, y1, x2, y2, value:integer);
var dx, dy, incr1, incr2, d, x, y, xend:integer;
begin
    dx := ABS(x2-x1);
    dy := ABS(y2-y1);
    d := 2 * dy - dx;       {Aloitusarvo d kaavasta 3.5}
    incr1 := 2 * dy;        {vakio, jota käytetään arvon d kasvattamisessa, mikäli d < 0}
    incr2 := 2 * (dy - dx);           {vakio, jota käytetään arvon d kasvattamisessa,
mikäli d ≥ 0}
    if x1 > x2
      then begin            {aloitetaan pisteestä, jonka x-koordinaatti on pienempi}
        x := x2;
        y := y2;
        xend := x1;
        end;
      else begin
        x := x1;
        y := y1;
        xend := x2;
        end;
      WRITE_PIXEL(x, y, value) {viivan ensimmäinen piste)
      while x < xend do begin
        x := x + 1;
        if d < 0;
          then d := d + incr1               {valitaan S1 - y-koordinaatti ei muutu}
          else begin            {valitaan T1 - y-koordinaattia kasvatetaan}
              y := y +1;
              d := d + incr2
              end
          WRITE_PIXEL(x, y, value)   {väritetään valittu piste}
      end {while}
end {BRESENHAM}
```

Kuva 3.4: Bresenhamin viivanpiirtoalgoritmi [15].

Moniviivat, suorakulmiot ja monikulmiot voidaan piirtää viivan piirtämiseen suunniteltujen algoritmien avulla soveltamalla näitä peräkkäin. Piirrettävät suorat määritellään suora- ja monikulmioiden kulmien avulla.

3.3 YMPYRÖIDEN PIIRTÄMINEN

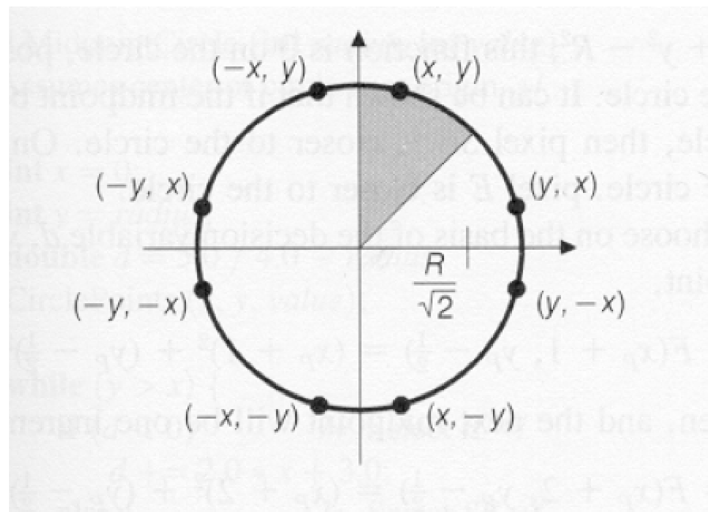
Ympyröiden piirtämiseen suunnitellut algoritmit noudattavat pitkälti luvussa 3.2 esiteltyjen algoritmien ideaa. Piirrettäessä ympyrää, hyödynnetään yhtälöä $x^2 + y^2 = R^2$, joka on sellaisen ympyrän yhtälö, jonka keskipiste sijaitsee origossa. Piirrettävä ympyrä saadaan yllä kuvattua yhtälöä vastaavaksi ympyräksi siirtämällä sitä tiettyjen kokonaislukujen verran origon suuntaan. Tämä siirtymä täytyy luonnollisesti huomioida piirrettäessä algoritmin avulla laskettuja kuvapisteitä. Edellä kuvattu yhtälö voidaan muuttaa muotoon $y = \pm (R^2 - x^2)^{1/2}$, jolloin ympyrän neljännes voidaan piirtää kasvattamalla x:ää välillä (0, … , R) ja ratkaisemalla $+y$ jokaisessa vaiheessa. Menetelmä toimii, mutta se jättää suuria aukkoja piirrettyyn ympyrään. Edellä kuvatun menetelmän avulla piirretty ympyrän neljännes on esitetty kuvassa 3.5.



Kuva 3.5: Yksinkertaisen kasvatusmenetelmän avulla piirretty ympyrän neljännes [14].

Toteuttamamme ympyröiden piirtämisestä huolehtiva funktio hyödyntää Bresenhamin algoritmia [14], joka laskee ympyrän kaikki pisteet yksinkertaista kasvatusmenetelmää hyödyntäen. Alkuperäinen Bresenhamin algoritmi laskee pisteet koko ympyrän alueelta mutta toteutuksessamme päädyimme käyttämään ratkaisua, joka laskee pisteet yhdelle kahdeksasosalle ympyrästä, x:n arvoille $x = 0$ ja $x = R/\sqrt{2}$. Tässä ratkaisussa ympyrän muut pisteet muodostetaan kuvassa 3.6 esitettyä 8-symmetrisuutta hyödyntäen.

Ympyröitä piirrettäessä pyritään hyödyntämään ympyröiden symmetrisuutta (kuva 3.6): Tämä tarkoittaa käytännössä sitä, että piirtoalgoritmin avulla piirretään ympyrän neljännes tai kahdeksasosa ja ympyrä täydennetään kopioimalla symmetriaa hyödyntäen pikselit ympyrän muihin osiin.



Kuva 3.6: ympyrän symmetrisuus [14].

Mikäli hyödynnetään 8-symmetrisuutta, voidaan kuvan 3.5 esittämältä ongelmalta välttyä. Tällöin kuvan 3.5 oikealla alhaalla sijaitseva ympyrän kahdeksasosa piirretään symmetriaa hyödyntäen ja ainoastaan vasemmalla ylhäällä sijaitseva kahdeksasosa piirretään algoritmia hyödyntäen. Alemmasta kahdeksasosasta tulee ylemmän kahdeksasosan peilikuvio ja pikselit tulevat väritettyä tasaisesti.

Kuvassa 3.7 on esitetty ympyrän pikseleiden värittämiseen tarkoitettu funktio, joka laskee ja piirtää automaattisesti kaikki kahdeksan symmetrisuuden avulla pääteltävissä olevaa ympyrän pikseliä, kun yhden tällaisen pikselin koordinaatit välitetään funktiolle parametreina.

```
Void CirclePoints (int x, int y, int value)
{
        WritePixel (x, y, value);
        WritePixel (y, x, value);
        WritePixel (y, -x, value);
        WritePixel (x, -y, value);
        WritePixel (-x, -y, value);
        WritePixel (-y, -x, value);
        WritePixel (-x, y, value);
        WritePixel (-y, x, value);
} /* CirclePoints */
```

Kuva 3.7: symmetrisuutta hyödyntävä pikselien värittämisestä huolehtiva funktio [14].

3.4 PRIMITIIVIEN TÄYTTÄMINEN

Monikulmiot ja ympyrät piirretään usein erilaisten viivanpiirtoalgoritmien avulla. Viivanpiirtoalgoritmit piirtävät kuitenkin ainoastaan näiden primitiivien ääriviivat, jättäen primitiivien täyttämisen toisten algoritmien tehtäväksi. Seuraavaksi käsittelen joitakin primitiivien täyttämiseen liittyviä kysymyksiä ja esittelen menetelmiä täyttämisen suorittamiseksi.

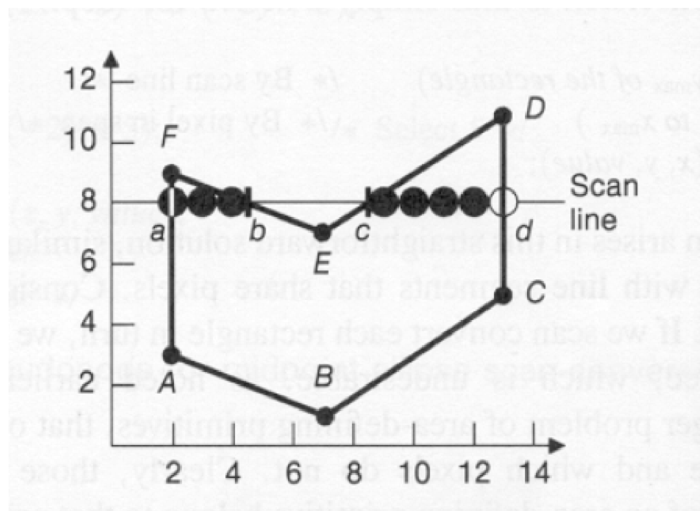
Primitiivien täyttäminen voidaan jakaa kahteen osaan: Määritetään ne pikselit, jotka täytetään sekä määritetään arvo, jolla kyseiset pikselit täytetään. Jälkimmäinen osatehtävä on suoraviivainen ja helppo toteuttaa, erityisesti mikäli monikulmio täytetään yhdellä värillä. Ongelmia esiintyy lähinnä määriteltäessä aluetta, johon täyttämisoperaatio kohdistuu [14]. Suorakulmioiden täyttäminen yhdellä värillä on helppo ja nopea toteuttaa, koska jokainen suorakulmion leikkaava rivi on identtinen muiden suorakulmion leikkaavien rivien kanssa.

Tällöin suorakulmio voidaan täyttää kahden sisäkkäisen silmukan aikana. Kuvassa 3.8 on esitetty lyhyt algoritmi, jonka avulla suorakulmion täyttäminen voidaan suorittaa.

```
for (y := ymin to ymax)
    for (x := xmin to xmax)
        VäritäPikseli(x, y, arvo)
```

Kuva 3.8: suorakulmion täyttämisalgoritmi.

Monikulmioita täytettäessä ei voida edetä näin suoraviivaisesti, koska monikulmioiden rajat eivät ole yhtä yksiselitteisesti määritettävissä. Monikulmiolla ei ole yhtä tiettyä muotoa, toisin kuin suorakulmiolla. Lisäksi monikulmiot voivat sisältää "reikiä" ja leikata yksittäisen *x*-akselin useammasta kohdasta, mikä vaikeuttaa monikulmioiden täyttämistä. Eräs tapa suorittaa monikulmion täyttäminen on käyttää algoritmia, joka käsittelee kuvaa rivi kerrallaan [14]. Algoritmi ratkaisee jokaisen rivin kohdalla, mitkä pikselit kyseiseltä riviltä väritetään ja mitkä jätetään värittämättä (kuva 3.9).

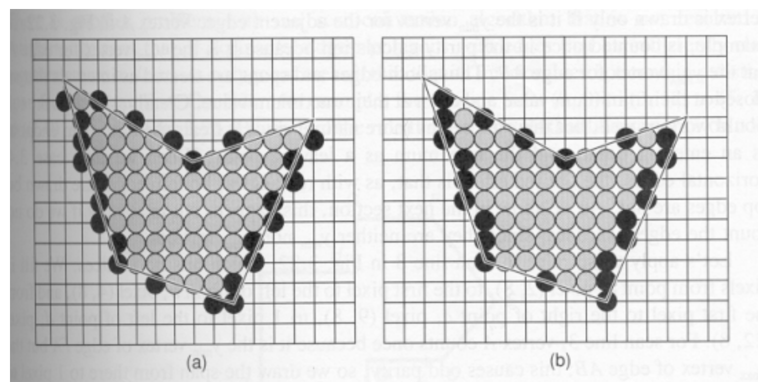


Kuva 3.9: monikulmio ja käsiteltävä rivi 8 [14].

Algoritmin perusperiaate on seuraava: etsitään kaikki kohdat, joissa käsiteltävä rivi leikkaa jonkin monikulmion reunoista. Lajitellaan nämä leikkauskohdat x-koordinaatin mukaisesti nousevaan järjestykseen. Väritetään kaikki pikselit, jotka jäävät leikkauskohtien koordinaateista muodostettujen koordinaattiparien väliin siten, että ne ovat monikulmion sisäpuolella.

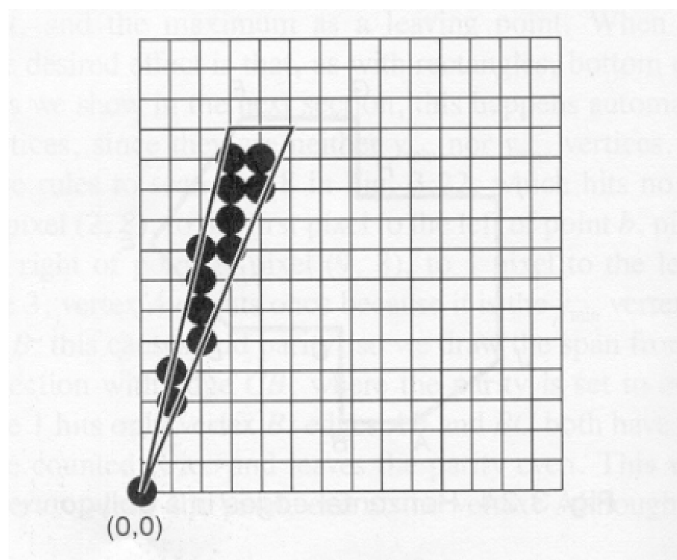
Pariteettibittiä käytetään ratkaistaessa onko käsiteltävä pikseli monikulmion sisä- vai ulkopuolella. Aluksi bitti on parillinen ja aina kun kohdataan käsiteltävän rivin ja monikulmion reunan muodostama leikkauspiste, vaihdetaan bitti parittomaksi/parilliseksi. Algoritmi värittää ne pikselit, joiden kohdalla bitti on pariton.

Mikäli leikkauskohtien koordinaatit pyöristetään aina lähimpään pikseliin, syntyy tilanne, jossa myös monikulmioiden ulkopuolella sijaitsevia pikseleitä tulee väritetyksi (kuva 3.10a). Tällöin syntyy ongelmia, mikäli kaksi monikulmiota sijaitsevat vierekkäin siten, että ne jakavat yhden sivun. Monikulmioiden jakamasta reunasta voi tulla epätasainen, mikä synnyttää vaikutelman siitä, että monikulmiot "uppoaisivat" toistensa sisään. Mikäli leikkauskohtien koordinaatteja ei pyöristetä aina lähimpään pikseliin, vaan siihen pikseliin, joka sijaitsee reunan sisäpuolella, voidaan tältä ongelmalta välttyä. Kuvassa 3.10 on esitetty kaksi monikulmiota, joista kohdan (a) monikulmiossa on väritetty myös monikulmion ulkopuolisia pikseleitä ja kohdan (b) monikulmiossa on väritetty ainoastaan monikulmion sisään jäävät pikselit. Kuvassa 3.10 monikulmiota käsitellään rivi kerrallaan ja mustalla merkityt pikselit ovat monikulmiolle löydettyjä reunapikseleitä kullakin rivillä.



Kuva 3.10: Monikulmion täyttäminen [14].

Piirtämällä ainoastaan monikulmioiden sisälle jäävät pikselit, voidaan välttää joitakin ongelmia, mutta tämä menetelmä myös synnyttää uusia ongelmia. Eräs menetelmän käytön synnyttämä ongelma on *suikaleet* (*slivers*) eli alueet, jotka ovat niin ohuita, ettei niiden sisäpuolelle jää väritettäviä pikseleitä, mutta jotka ovat kuitenkin osa piirrettävää monikulmiota. Mikäli edellä kuvattua ongelmaa ei ratkaista, voi täytettävä monikulmio sisältää aukkokohtia, kuten kuvasta 3.11 voidaan nähdä.



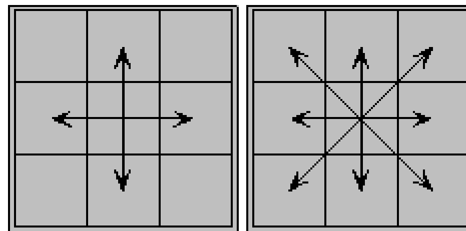
Kuva 3.11: suikale [14].

Toinen suoraviivainen perusmenetelmä monikulmioiden täyttämiseen on rekursiivinen *flood-fill-algoritmi* [15] (kuva 3.12). Kuvan 3.12 algoritmi ottaa parametrina aloituspisteen, täyttövärin sekä korvattavan värin. Mikäli aloituspiste on väritetty korvattavalla värillä, algoritmi värittää aloituspisteen täyttövärillä sekä yrittää kutsua itseään rekursiivisesti välittäen parametreina käsiteltävän pisteen neljä naapuripistettä. Algoritmi on *4-konnektiivinen*, koska se välittää parametreina pääilmansuunnissa sijaitsevat neljä naapuripistettä. Toinen mahdollisuus on käyttää *8-konnektiivista* flood-fill-algoritmia, joka välittää parametreina myös väli-ilmansuunnissa sijaitsevat naapuripisteet. Näiden kahden

konnektiivisuuden eroa on havainnollistettu kuvassa 3.13. Toteutuksessamme käytimme 4-konnektiivista flood-fill -algoritmia.

```
Procedure FLOOD_FILL_4 (x, y, old_value, new_value)

begin
    if READ_PIXEL(x, y) = old_value
        then begin
                WRITE_PIXEL(x, y, new_value);
                FOOD_FILL_4(x, y-1, old_value, new_value);
                FOOD_FILL_4(x, y+1, old_value, new_value);
                FOOD_FILL_4(x-1, y, old_value, new_value);
                FOOD_FILL_4(x+1, y, old_value, new_value);
        end
end
```

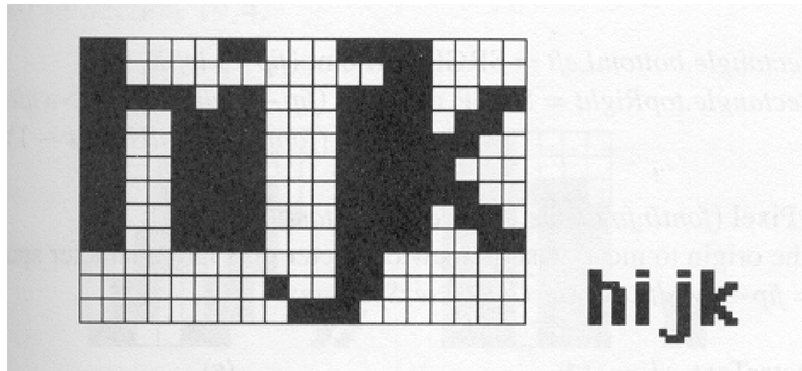
Kuva 3.12: Rekursiivinen flood-fill_4 algoritmi [15].



Kuva 3.13: 4- ja 8-konnektiivisuus.

3.5 MERKKIEN GENEROINTI

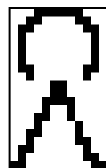
Merkkien generoimiseksi on olemassa kaksi perustekniikkaa [14]. Yleisin ja laskennallisesti vaativin tapa on määritellä merkit (pyöristetyiksi) monikulmioiksi ja muuntaa ne rasterimuotoon tarvittaessa käyttämällä monikulmioiden piirtämiseen suunniteltuja rasterigrafiikka-algoritmeja. Toinen menetelmä on määritellä jokaisen fontin jokainen merkki valmiiksi *bittikartaksi*, joka kopioidaan haluttuun kohtaan piirrosaluetta. Kuvassa 3.14 on esitetty osa yhdelle fontille määriteltyä bittikartastoa.



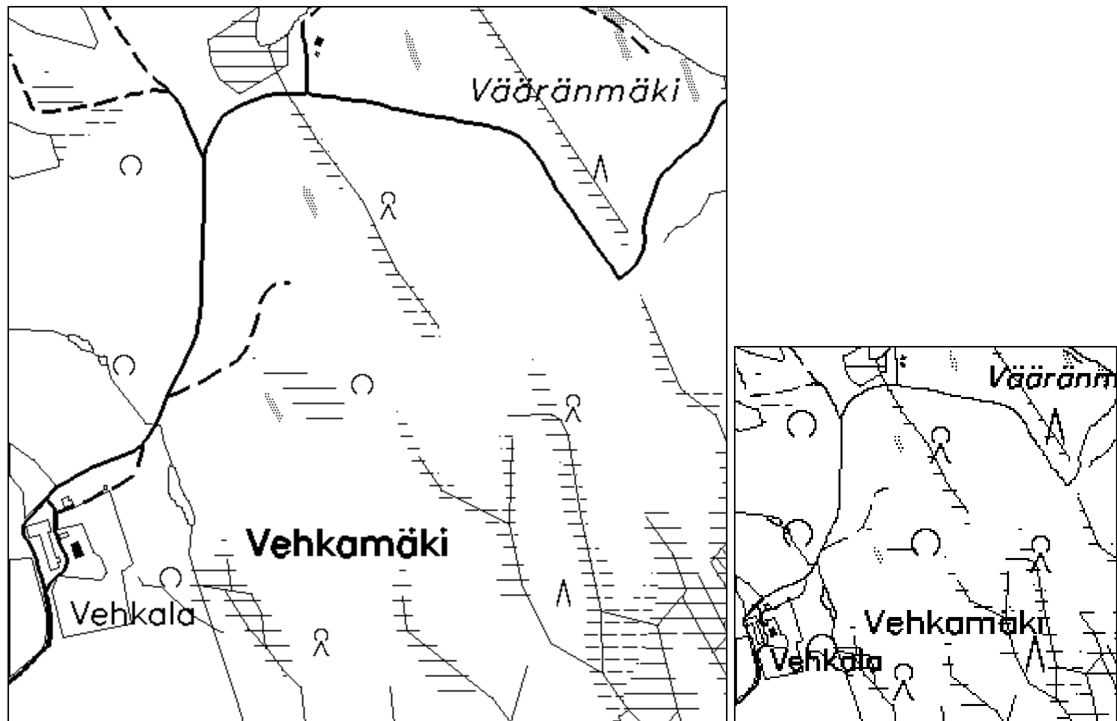
Kuva 3.14: esimerkki bittikarttojen avulla määritellyistä merkeistä [14].

Bittikarttamenetelmän eräs haitta on se, että se vaatii erilliset bittikartat jokaiselle fontin, koon ja tyylin yhdistelmälle, mikäli halutaan säilyttää tekstin visuaalinen ulkoasu hyvällä tasolla. Esimerkiksi yksi fontti kahdeksassa eri koossa ja neljällä eri tyylillä (normaali, lihavoitu, kursivoitu ja lihavoitu kursivoitu) vaatii 32 bittikartastoa.

Toteutuksessamme käytimme bittikarttoja topologisten merkkien rasteroinnissa (kuva 3.15). Toteuttamamme ohjelma mahdollisti vektorikarttakuvan rasteroinnin useaa eri resoluutiota käyttäen. Topologisia merkkejä varten käyttämämme bittikartasto sisältää kuitenkin ainoastaan yhtä resoluutiota varten laaditut bittikartat. Tämä aiheutti ongelmia tietyillä resoluutiolla rasteroituihin karttakuviin. Kuva 3.16 esittää kahdella eri resoluutiolla rasteroitua karttaa, joista oikeanpuoleisen karttakuvan sisältämät topologiset merkit ovat suhteettoman suuria muihin karttasymboleihin verrattuna.



Kuva 3.15: Bittikartta, joka esittää topologista merkkiä.



Kuva 3.16: kahta eri resoluutiota käyttäen rasteroitu karttakuva.

Generoitaessa merkkejä monikulmiomenetelmällä yksittäisen merkin vaatima tilantarve on suurempi kuin bittikarttamenetelmässä, mutta se voidaan piirtää useassa eri koossa käyttämällä *skaalausmenetelmiä*. Lisäksi eri tyylien toteuttamiseksi ei tarvita erillisiä merkkejä vaan ne voidaan johtaa perusmerkistä erilaisilla muokkausoperaatioilla ja esimerkiksi tekstin pyörittäminen ja kallistaminen on helpompi suorittaa.

Toteutuksessamme generoimme tekstin käyttämällä *Hershey-vektorifontteja* [3], jotka koostuvat noin 2000 merkistä. Päädyimme käyttämään Hershey-vektorifonttien osajoukkoa, jota *kutsutaan simplex- merkeiksi* (kuva 3.17).

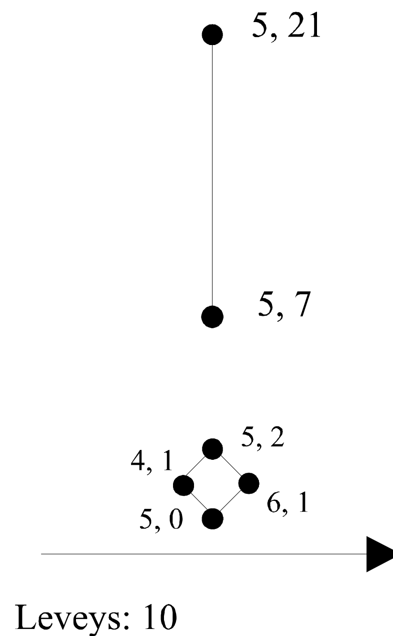
Kuva 3.17: Simplex-vektorifontit [3].

Hershey-vektorifonteissa yksittäinen merkki määritellään kokonaisluvuista koostuvan listan avulla. Kokonaisluvut ilmaisevat merkin leveyden sekä niiden viivojen koordinaatit, joiden avulla merkki piirretään. Kuvassa 3.18 on esitetty osa simplex-merkkijoukolle määritellystä 2-ulotteisesta kokonaislukutaulukosta. Taulukko on toteutettu c-kielellä ja yksittäinen taulukon rivi vastaa yhtä merkkiä.

```
int simplex[95][112] = {
0,16, /* Ascii 32 */
 -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,
8,10, /* Ascii 33 */
 5,21, 5, 7,-1,-1, 5, 2, 4, 1, 5, 0, 6, 1, 5, 2,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,
5,16, /* Ascii 34 */
4,21, 4,14,-1,-1,12,21,12,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,
```

Kuva 3.18: Osa Simplex-merkkijoukolle määritellystä kokonaislukutaulukosta [3].

Kuvassa 3.18 esitetyn taulukon lukujen merkitys on seuraava: kunkin taulukon rivin kohdalla ensimmäinen luku kertoo merkin piirtämiseen vaadittavien koordinaattien määrän. Toinen luku kertoo merkin leveyden. Seuraavat luvut ilmaisevat ne koordinaattiparit, joiden välille viivat piirretään. Koordinaattipari (-1, -1) tarkoittaa, ettei viivaa edellisestä koordinaattiparista seuraavaan koordinaattipariin piirretä. Kuvassa 3.19 on piirretty kuvan 3.18 sisältämän kokonaislukutaulukon toisen rivin esittämä merkki (Ascii 33).



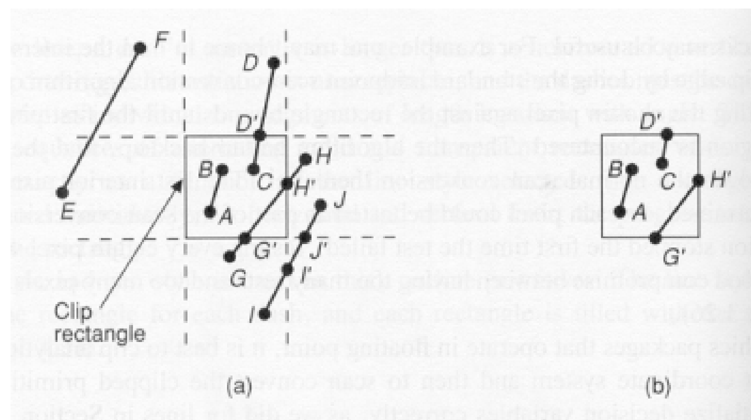
Kuva 3.19: Simplex-merkkijoukon sisältämä huutomerkki.

Rasteroitaessa tekstiä Hersheyn fonttien avulla, voidaan tekstiä kirjoittaa kohtalaisen helposti eri suuntiin koordinaattimuunnosten avulla. Toteutuksessamme päädyimme Hershey-fonttiin ja Simplex-merkkeihin lähinnä kahdesta syystä: Tarvitsimme tekstin rasteroimiseen menetelmän, jolla tekstiä voidaan kirjoittaa vapaasti kaikkiin suuntiin. Tämän lisäksi yksi vakiofontti muutamassa eri kirjainkoossa ja perustyylivaihtoehdoilla oli riittävä.

3.6 PRIMITIIVIEN LEIKKAAMINEN

Primitiivien *leikkaaminen* (*clipping*) on tärkeä osa-alue primitiivien piirtämisessä. Lähes poikkeuksetta primitiivien piirtäminen kohdistuu tietylle kuva-alueelle, etenkin piirrettäessä primitiivejä näytölle. Tällöin täytyy primitiivien piirtämisestä huolehtivan ohjelman kyetä piirtämään ainoastaan se osa primitiivistä, joka on käsiteltävän kuva-alueen rajojen sisäpuolella. Leikkaaminen voidaan suorittaa analyyttisesti ennen primitiivin piirtämistä, samanaikaisesti primitiivin piirtämisen kanssa tai osana pikselien värittämistä [14]. Tässä luvussa esittelen lyhyesti primitiivien analyyttista leikkaamista. Keskityn viivojen leikkaamiseen suorakulmion muotoisen leikkausalueen avulla.

Viivat, jotka leikkaavat suorakulmion muotoisen *leikkausalueen* (*clip region*) katkaistaan aina yhteen viivasegmenttiin. Leikkausalueen reunalla olevat viivat tulkitaan leikkausalueen sisällä oleviksi ja piirretään muodostettavaan kuvaan. Kuvassa 3.20 on esimerkki suorakulmion muotoisesta leikkausalueesta ja erilaisista tilanteista, joissa viivojen leikkaamista sovelletaan.



Kuva 3.20: erilaisia tapauksia, joissa viivoja joudutaan leikkaamaan [14].

Perusidea viivojen leikkaamisessa on seuraava:

1. Tutkitaan, mitkä viivoista voidaan hyväksyä triviaalisti. Tällaisia ovat viivat, joiden molemmat päätepisteet ovat leikkausalueen sisäpuolella.

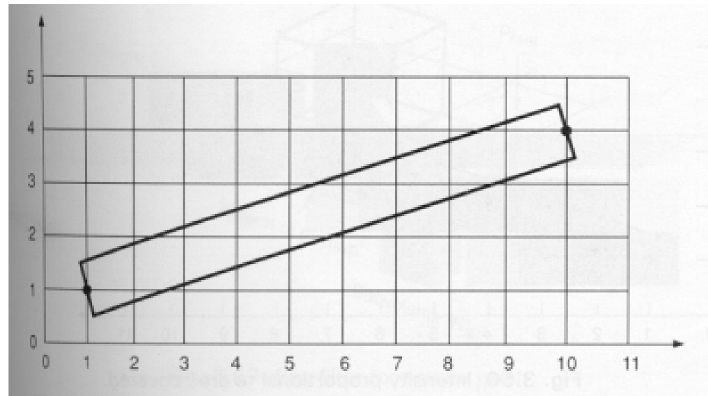
2. Mikäli viivan toinen päätepiste sijaitsee leikkausalueen ulkopuolella ja toinen sisäpuolella, lasketaan se piste, jossa viiva ja leikkausalue leikkaavat toisensa. Tämän leikkauspisteen kohdalta käsiteltävä viiva jaetaan kahtia ja näin saadaan kaksi viivaa, joista toinen voidaan hylätä ja toinen hyväksyä.

3. Mikäli viivan molemmat päätepisteet ovat leikkausalueen ulkopuolella, joudutaan suorittamaan lisää laskentaa, että saadaan selville, onko viiva kokonaisuudessaan leikkausalueen ulkopuolella vai kulkeeko se leikkausalueen kautta. Ensimmäisessä tapauksessa viiva voidaan hylätä kokonaisuudessaan. Jälkimmäisessä tapauksessa etsitään viivan ja leikkausalueen väliset leikkauspisteet, joiden kohdalta viiva jaetaan osiin. Syntyneistä uusista viivoista yksi voidaan hyväksyä kokonaisuudessaan ja loput viivat voidaan hylätä.

3.7 ANTIALIASOINTI

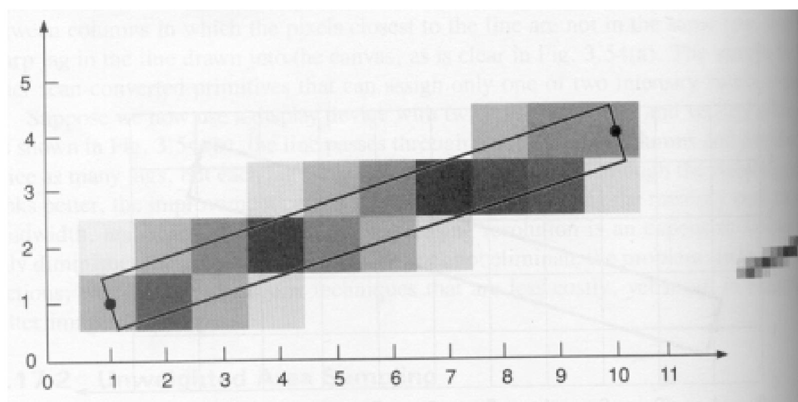
Yleinen rasterigrafiikka-algoritmeilla piirrettyihin primitiiveihin liittyvä ongelma on pykäläkuviota sisältävät reunat (jagged edges), jotka ilmaantuvat rasterimuotoon piirrettyihin primitiiveihin. Tämä ilmiö on seurausta siitä, että piirtovaiheessa jokaisen pikselin väri joko korvataan piirrettävän primitiivin värillä tai jätetään ennalleen. Tätä kutsutaan *aliasoinniksi*. Aliasointi-ilmiön ehkäisemiseksi on kehitetty erilaisia tekniikoita, joita nimitetään yhteisesti *antialiasoinniksi* [14]. Aliasointi-ilmiön vaikutusta voidaan pienentää kasvattamalla resoluutiota. Tämä on kuitenkin sekä laskennallisesti että muistin kannalta kallis ratkaisu, joka ainoastaan vähentää aliasoinnin haitallista vaikutusta, ei poista sitä.

Eräs menetelmä, joka pyrkii poistamaan aliasoinnin vaikutuksen, *on painottamaton alueen täyttömenetelmä* (*unweighted area sampling*) [14]. Tämä menetelmä perustuu siihen havaintoon, että vaikka ideaalin viivan leveys olisi 0, piirrettävän viivan leveys on aina oltava suurempi kuin 0. Piirretty primitiivi vie aina jonkin äärellisen tilan rasteriruudukolla. Kuvassa 3.21 on esitetty erään piirretyn viivan viemä alue.



Kuva 3.21: pisteestä (1,1) pisteeseen (10,4) piirretty viiva [14].

Painottamattomassa alueen täyttömenetelmässä luovutaan lähestymistavasta, jossa jokainen pikseli joko väritetään tai jätetään värittämättä ja käytetään lähestymistapaa, jossa pikselille määritetään kirkkausarvo sen mukaan, miten suuri osa siitä on primitiivin peitossa. Saadun kirkkausarvon perusteella pikselille määritetään väri, jolla se väritetään. Kuvassa 3.22 on havainnollistettu tätä menettelytapaa.



Kuva 3.22: primitiivin kirkkauden suhteellisuus primitiivin peittämään pinta-alaan [14].

Painottamattomassa alueen täyttömenetelmässä ainoastaan se, kuinka suuri osa pikselin alueesta on primitiivin peitossa, vaikuttaa siihen, minkä kirkkausarvon ja täten myös värin

kyseinen pikseli saa. Menetelmässä ei ole merkitystä sillä, minkä kohdan pikselistä primitiivi peittää.

Painotetussa alueen täyttömenetelmässä (*weighted area sampling*) [14] peruslogiikka on vastaava edellisen menetelmän kanssa. Menetelmä eroaa edellisestä menetelmästä siinä, että tässä menetelmässä myös sillä, minkä osan pikseliä primitiivi peittää, on vaikutusta siihen, minkä kirkkausarvon ja värin käsiteltävä pikseli saa. Pikselin eri alueilla on erilainen painoarvo määriteltäessä pikselin kirkkautta. Edellä kuvatut menetelmät ovat sovellettavissa kuville, joiden värisävyjen määrä on suurempi kuin kaksi. Toteutuksessamme kuvat rasteroitiin mustavalkoisiksi binäärikuviksi, eikä antialiasointia tästä syystä toteutettu.

4. VERKKO-ONGELMAT JA NIIHIN SOVELLETUT RATKAISUMENETELMÄT

Karttakuvat sisältävät usein osia erilaisista verkoista, joita olisi hyvä pystyä hyödyntämään. Tällaisia verkkoja ovat lähinnä tie- ja rautatieverkot. Tämän lisäksi esimerkiksi voimalinjat, tietoliikennelinjat tai vesistöreitit voivat muodostaa verkkoja, joita halutaan tarkastella. Mikäli näitä vektorikarttojen sisältämiä verkkoja halutaan tarkastella tietokoneessa suoritettavien algoritmien avulla, tulee vektoreina määritelty verkko poimia karttakuvasta erilleen ja tallettaa erilliseen tietorakenteeseen. Luvussa 4.1 tarkastelen vektoreiden muuntamista verkoiksi ja ongelmia, joita tähän osatehtävään liittyy.

Mikäli vektoreiden muuntaminen verkoksi onnistuu, voidaan karttoihin liittyvien verkko-ongelmien ratkaisemisessa hyödyntää laajaa aiheesta laadittua kirjallisuutta [2, 5, 9, 13, 25, 28]. Luvuissa 4.2 - 4.3 tarkastelen verkko-ongelmia aikavaativuus-käsitteen avulla. Luvussa 4.4 määrittelen kombinatorisen optimointiongelman käsitteen ja esittelen *kauppamatkustajan ongelman*. Luvussa 4.5 esittelen lähestymistapoja *kombinatorisiin optimointiongelmiin*. Luvussa 4.6 tarkastelen joitakin heuristisia perusalgoritmeja, joiden avulla kombinatorisia optimointiongelmia pyritään ratkaisemaan. Luvussa 4.7 tarkastelen kauppamatkustajan ongelman ratkaisemista *geneettisellä algoritmilla*, joka on eräs heuristinen lähestymistapa kombinatorisiin optimointiongelmiin.

4.1 VEKTOREIDEN MUUNTAMINEN VERKOIKSI

Verkoksi muuntamisesta on olemassa niukasti julkaistua tutkimusmateriaalia ja useasti muuntaminen on suoritettu tietyille tiedossa oleville kartta-aineistoille luotaessa aineistoa verkkoalgoritmien tutkimukselle. Tällöin muuntamisessa on voitu suorittaa vektorien valinta ja syntyneen verkon oikeellisuuden tarkistaminen manuaalisesti. Käytännön sovelluksissa nämä osatehtävät tulisi pystyä automatisoimaan mahdollisimman pitkälle.

Vektorikarttojen verkoksi muuntamisen perustavoitteena on poimia vektorikarttojen sisältämä topologia omaksi erilliseksi tietorakenteeksi. Haluttu ratkaisu voidaan saavuttaa yleensä kohtuullisessa ajassa muuntamalla vektorikartta ensin verkoksi ja suorittamalla käsiteltävänä olevan ongelman ratkaisemiseen suunniteltu algoritmi kyseisessä verkossa.

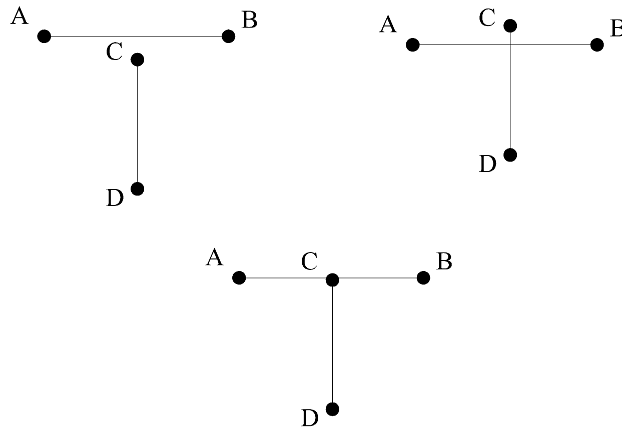
Vektorikartan muuntaminen verkoksi koostuu näkemykseni mukaan seuraavista osatehtävistä:

1. Halutun informaation erottaminen muusta informaatiosta
2. Vektori-informaatiossa olevien epätarkkuuksien ja epäolennaisuuksien poistaminen
3. Solmujen ja kaarten muodostaminen poimitusta informaatiosta

Ensimmäisen osatehtävän onnistumisen kannalta on tärkeää, että vektorikartasta erotettavalla informaatiolla on jokin yhteinen, kyseisen informaation muusta vektorikartan informaatiosta omaksi ryhmäkseen erottava piirre. Looginen valinta, mikäli käytettävä vektoriformaatti tämän mahdollistaa, on ryhmitellä kartan sisältämä informaatio omiksi semanttisiksi kerroksiksi (esimerkiksi kerros 1 = vesialueet, kerros 2 = tieverkosto). Tällöin haluttu informaatio voidaan poimia kohtuullisen nopeasti ja yksinkertaisen päättelylogiikan avulla. Toinen mahdollisuus ryhmitellä vektorikartan sisältämä informaatio on käyttää tietynlaisille objekteille tiettyä väriä (esimerkiksi sininen väri = vesialue, musta väri = tieverkosto) tai ryhmitellä objektit ominaisuustiedon avulla, esimerkiksi tallettamalla tietä esittävien moniviivojen ominaisuustietoihin tunnus joka ilmaisee kyseisen moniviivan semanttisen merkityksen. Mikäli haluttua informaatiota ei voida tunnistaa muusta informaatiosta tällaisen yksiselitteisen kriteerin avulla, on osatehtävän onnistunut toteuttaminen käytännössä mahdotonta.

Vektori-informaatiossa olevien mahdollisten epätarkkuuksien poistaminen on kriittinen osatehtävä koko muuntamisprosessin onnistumisen kannalta. Syntynyttä verkkoa käytetään esimerkiksi lyhimmän reitin etsimisessä todellisessa tilanteessa ja jo yhden kriittisen solmun tai kaaren puuttuminen voi kasvattaa algoritmin löytämän lyhimmän reitin pituutta moninkertaiseksi todelliseen lyhimpään reittiin nähden. Käytännön sovelluksissa tällaisen virheen mahdollisuutta ei voida hyväksyä.

Eräs esimerkki digitaalisen kartan sisältämistä mahdollisista epätarkkuuksista on tilanne, jossa t-risteyksen kohdalla kartassa tiet eivät kosketa toisiaan (kuva 4.1). Mikäli verkkoa muodostaessa ei tarkasteta epätarkkuuksia, voi verkko sisältää kuvan tapauksessa kaaret *A-B* ja *C-D* vaikka verkon tulisi sisältää kaaret *A-C*, *C-B* ja *C-D*.

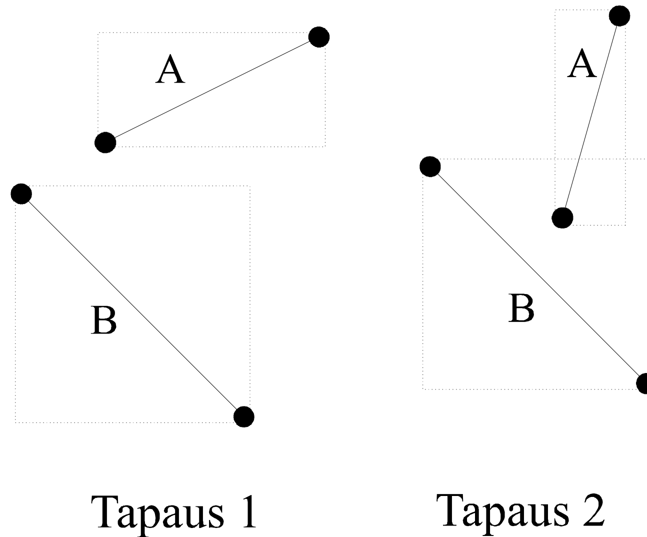


Kuva 4.1: T-risteykseen liittyviä mahdollisia epätarkkuuksia (ylärivissä)
ja tavoiteltu tilanne (alarivissä).

kuvan 4.1 esittämien epätarkkuuksien poistamiseen voidaan käyttää menetelmää, jossa määritellään tietty virhetoleranssi. Mikäli poikkiviiva ja pystyviivan päätepiste ovat tarpeeksi lähellä toisiaan, tulkitaan kyseinen kartan kohta t-risteykseksi ja muodostetaan verkkoon kyseiseen kohtaan uusi solmu. Virhetoleranssin käyttö sisältää kuitenkin tiettyjä ongelmia. Virhetoleranssimenetelmä on riskialtis, sillä se saattaa tunnistaa risteyksiksi sellaisia kartan kohtia, jotka eivät todellisuudessa ole risteyksiä. Tällöin muodostettu verkko sisältää kaaria ja solmuja, jotka eivät ole todellisuudessa olemassa. Tästä voi puolestaan seurata, että verkkoalgoritmit tuottavat ratkaisuja, jotka eivät ole kelvollisia.

Epätarkkuuksien tarkistaminen ja poistaminen verkkoa muodostettaessa vie erittäin paljon aikaa, koska viivojen läheisyyden tutkiminen vaati paljon laskentaa. Tämä johtuu siitä että jokaista viivaa pitää verrata jokaiseen muuhun karttakuvan sisältämään viivaan. Eräs menetelmä laskennan vähentämiseksi on käyttää ensimmäisessä tarkasteluvaiheessa objektit rajaavia suorakulmioita (bounding box) varsinaisten viivojen sijaan. Menetelmän avulla suuri osa mahdollisista vertailuista voidaan suorittaa koordinaattien vertailun avulla, sen sijaan, että jouduttaisiin etsimään viivojen risteyspisteitä (kuva 4.2). Kuvan 4.2 kohdalla tapauksessa 1 huomataan etteivät viivoja rajaavat suorakulmiot kosketa toisiaan, eikä viivojen yksityiskohtaisempi tarkastelu ole tarpeen. Tapauksen 2 kohdalla viivoille joudutaan

suorittamaan yksityiskohtaisempaa tarkastelua. Suorakulmioita hyödyntävä menetelmä on tehokas, mutta se vaatii paljon muistia.

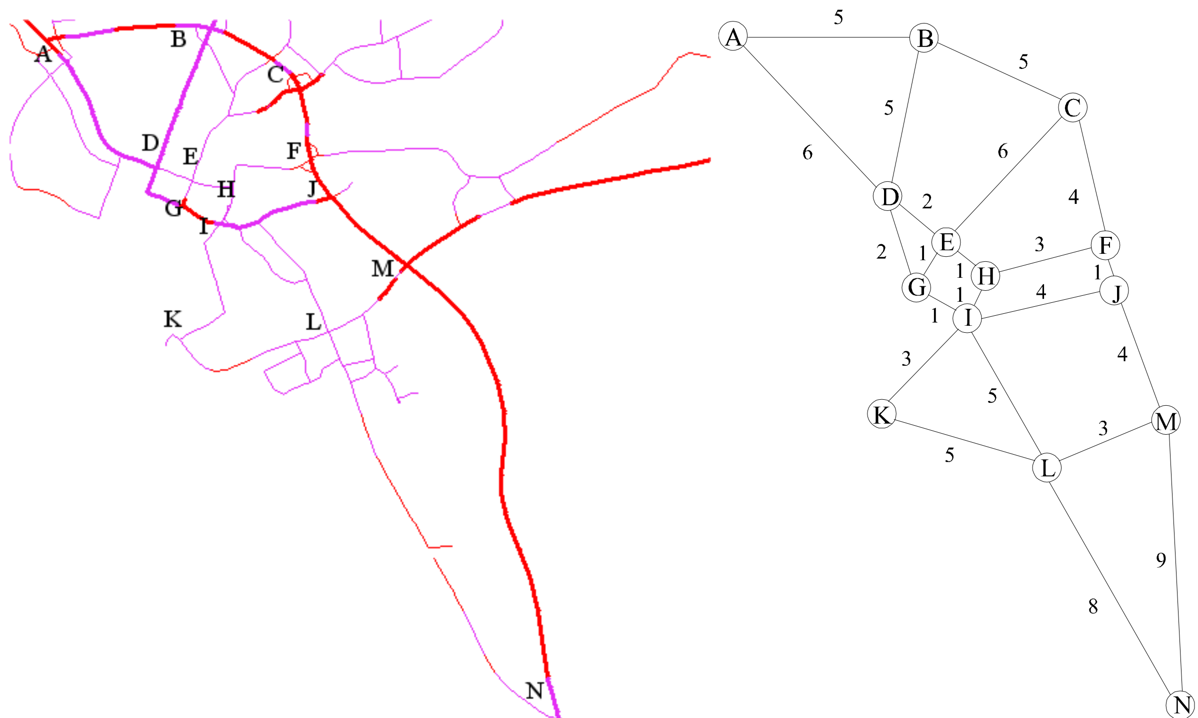


Kuva 4.2: Suorakulmioiden käyttö epätarkkuuksien poistamisessa.

Verkosta tulee poistaa epätarkkuuksien lisäksi myös mahdolliset epäolennaisuudet. Vektorikarttojen sisältämät verkot talletetaan lähinnä moniviivojen avulla. Nämä moniviivat sisältävät sellaista informaatiota, jota ei tarvita verkko-ongelmia ratkaistaessa. Tarkasteltaessa verkko-ongelmia on tarpeellista tietää solmujen sijainnit, niiden välisten kaarien olemassaolo ja kaarten pituudet. Yksittäistä tieosuutta esittävän kaaren muoto sen sijaan on epäoleellista informaatiota. Kuvassa 4.3 esimerkiksi tieosuutta *M-N* esittävä moniviiva sisältää useita koordinaattipareja jotka voidaan poistaa muodostettavasta verkosta. Kartasta muodostetussa verkossa kyseinen tieosuus on määritelty kaarella *M-N* ja kaaren sijainti voidaan määrittää kaaren päätesolmujen koordinaattien avulla. Toinen esimerkki epäoleellisesta informaatiosta on verkon solmun *F* ympärillä olevat monitasoristeyksen ramppeja esittävät moniviivat. Tieverkostoa esittävän verkon kannalta riittää, että se sisältää solmun F sekä kaaret *C-F*, *F-H* ja *F-J*. Ramppeja esittävät moniviivat voidaan jättää pois muodostettavasta verkosta.

Kolmas osatehtävä eli verkon solmujen ja kaarien muodostaminen on suoraviivainen ja triviaali tehtävä, mikäli kaksi ensimmäistä osatehtävää on kyetty ratkaisemaan. Kuvassa 4.3

on esitetty kartan 2 kerrokset 2,3,4, 8, 9 ja 10 sekä niiden avulla muodostamani verkko. Kerrokset esittävät osaa kartan sisältämästä tieverkostosta. Olen jättänyt verkosta pois joitakin tieosuuksia, jotta kartan koko säilyy sopivana verkon käyttämiseksi havaintoesimerkkinä. Lisäksi olen muodostanut kaaren solmujen *L* ja *M* välille, koska tarkasteltaessa tieverkostoa kokonaisuudessaan kyseisten solmujen välillä on suora yhteys. Esitän verkon kaarien etäisyydet kokonaislukuina, jotka ilmaisevat kaarien pituuksia suhteessa toisiinsa.



Kuva 4.3: Osa kartasta 2 (vasemmalla) sekä siitä muodostettu verkko (oikealla).

4.2 POLYNOMISESSA AJASSA RATKEAVAT ONGELMAT

Tarkasteltaessa erilaisia tietojenkäsittelytieteen ongelmia tehdään usein erottelu polynomisessa ajassa ratkeavien ongelmien ja eksponentiaalisessa ajassa ratkeavien ongelmien välillä [5, 9]. Polynomisessa ajassa ratkeavat ongelmat ovat ongelmia, joiden ratkaisualgoritmit ovat aikavaativuudeltaan lineaarisia tai jonkin polynomin rajoittamia suhteessa syötteen kokoon. Tällaisia ongelmia pidetään käytännössä ratkeavina ongelmina [9].

Mikäli aikavaativuutta ilmaiseva polynomi on asteeltaan korkea, esimerkiksi N^{100} (N = syötteen koko), ei ongelmaa ole perusteltua pitää käytännössä ratkeavana. Ristiriita kahden edellisen lauseen välillä ei ole kuitenkaan kovin merkityksellinen, koska hyvin harva käytännön ongelma on sellainen, jonka ratkaisualgoritmi on aikavaativuudeltaan korkea-asteisen polynomin rajoittama [9]. Yksi esimerkki polynomisessa ajassa ratkeavista verkko-ongelmista on lyhimpien polkujen etsiminen verkosta, joka sisältää vain positiivisia kaarien painoja.

Klassinen algoritmi lyhimpien polkujen etsimiseen yhdestä verkon solmusta kaikkiin verkon muihin solmuihin on *Dijkstran algoritmi* [5]. Algoritmin perusperiaate on seuraava: etäisyydet lasketaan taulukkoon D, jonka indeksit ovat välillä 2 … n, missä n ilmaisee verkon solmujen määrän. Aloitussolmua merkitään kirjaimella s. Alustetaan taulukko D siten, että jokaiseen taulukon indeksiin i sijoitetaan kaareen (s, i) pituus, mikäli kaari on olemassa ja ∞, mikäli kyseisten solmujen välillä ei ole kaarta. Jokaisella algoritmin suoritushetkellä pidetään yllä niiden verkon solmujen joukkoa S, joille minimaalinen polku aloitussolmusta kyseiseen solmuun on jo tiedossa. Tämän lisäksi pidetään yllä niiden solmujen joukkoa C, joka sisältää verkon solmut, jotka eivät kuulu joukkoon S. Jokaisessa algoritmin iteraatiossa valitaan joukosta C solmu v, jonka etäisyys aloitussolmuun on pienin. Solmu v lisätään joukkoon S ja poistetaan joukosta C. Lisäksi iteraatiossa tarkastetaan jokaisen joukkoon C kuuluvan solmun w kohdalla, onko solmun etäisyys aloitussolmuun suurempi kuin solmun v etäisyys aloitussolmuun + kaari solmusta v solmuun w. Mikäli edellisessä lauseessa kuvattu ehto täyttyy, korvataan taulukossa D etäisyys $D[w]$ etäisyydellä ($D[v]$ + kaaren (v, w) pituus). Algoritmin suorituksen loppuessa verkon kaikki solmut ovat joukossa S ja joukko C on tyhjä joukko. Taulukko D pitää sisällään lyhimmät etäisyydet aloitussolmusta s muihin verkon solmuihin. Kuvassa 4.4 esitetyn Dijkstran algoritmin toteutuksen aikavaativuus on $O(n^2)$, missä n on syötteen koko [5]. L on solmujen väliset etäisyydet sisältävä taulukko ja N verkon solmujen joukko.

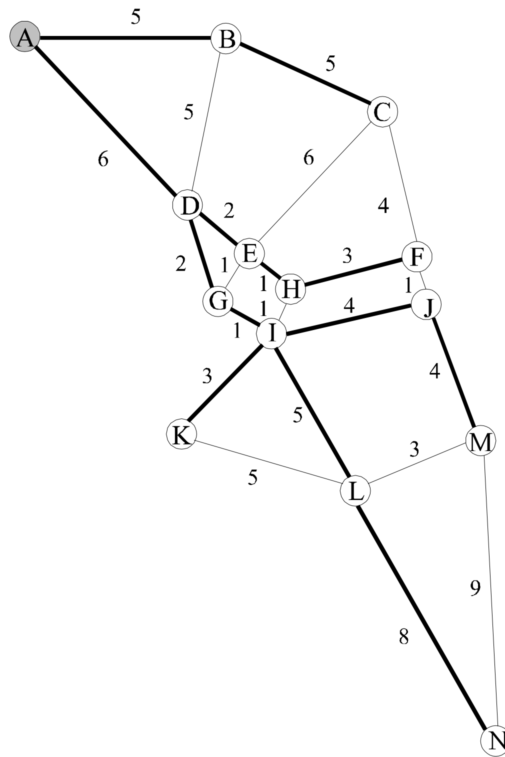
```
Function Dijkstra (L[1…n, 1…n]):array[2…n]

C ← {2,3, … , n}   {S = N \ C}
for i ← 2 to n do D[i] ← L[1,i]
toista n kertaa
    v ← joukossa C oleva solmu, jonka etäisyys aloitussolmuun on pienin
    C ← C \ {v}
    jokaiselle alkiolle w ∈ C suorita
        D[w] ← min(D[w], D[v] + kaaren (v, w) pituus)
return D
```

Kuva 4.4: Dijkstran algoritmi [5].

Dijkstran algoritmin soveltaminen esimerkkiverkolle palauttaa seuraavan taulukon: $D = \{5, 10, 6, 8, 12, 8, 9, 9, 13, 12, 14, 17, 22\}$. Yllä esitetty Dijkstran algoritmin toteutus palauttaa ainoastaan lyhimpien polkujen pituuden. Mikäli halutaan saada selville myös ne solmut, joista kukin lyhin polku koostuu, seuraavat muutokset kuvassa 4.4 kuvattuun algoritmiin riittävät: muodostetaan algoritmin suorituksen alussa taulukko $P[i]$, jossa $i = 2, \ldots, n$ ja alustetaan taulukon alkiot aloitussolmun tunnuksella. Muutetaan jokaiselle alkiolle w suoritettava lause muotoon:

```
if D[w] >  D[v] + kaaren (v, w) pituus then
        D[w] ←  D[v] + kaaren (v, w) pituus
        P[w] ← v.
```

Taulukon P indeksi $P[v]$ sisältää sen solmun tunnuksen, joka edeltää solmua v tarkasteltavalla lyhimmällä polulla. Esimerkkiverkon tapauksessa algoritmin palauttaman taulukon P sisältö oli seuraava: $P = \{A, B, A, D, H, D, E, G, I, I, I, J, L\}$. Kuvassa 4.5 on graafisesti esitetty lyhimmät polut solmusta A muihin verkon solmuihin.



Kuva 4.5: Dijkstran algoritmin avulla löydetyt lyhimmät polut esimerkkiverkolle.

Muodostettaessa verkkoja kartoista voidaan verkon solmuihin liittää sijaintietoa. Tarkasteltaessa erilaisia ongelmia tällaisissa verkoissa voidaan sijaintitietoa hyödyntää ratkaisualgoritmin suorituksessa. Teoksessa [26] on esitetty algoritmi *A**, joka etsii polun kahden solmun välillä. Algoritmi palauttaa polun, joka on lyhyt mutta ei välttämättä lyhin mahdollinen. Edellä mainitusta syystä algoritmi A* on *heuristinen hakualgoritmi*.

Algoritmin toiminnan ymmärtämiseksi tarkastelemme painotettua ratkaisuverkkoa, jonka jokaiseen tilaan s liitämme funktiot g, h ja f. Funktioilla on seuraava merkitys:

$g(s)$ = lyhimmän alkutilasta tilaan s johtavan polun pituus,
$h(s)$ = lyhimmän tilasta s lopputilaan johtavan polun pituus ja
$f(s) = g(s) + h(s)$ = lyhimmän alkutilasta tilan s kautta lopputilaan johtavan polun pituus.

Ongelmana on, ettei minkään yllä kuvatun funktion arvoja tiedetä heti, vaan joudutaan käyttämään epäluotettavaa tietoa. Merkitsemme:

$g'(s)$ = lyhimmän jo löydetyn alkutilasta tilaan s johtavan polun pituus,

$h'(s)$ = arvio lyhimmän tilasta s lopputilaan johtavan polun pituudelle ja

$f'(s) = g'(s) + h'(s)$.

Funktiota h' kutsutaan heuristiseksi funktioksi ja funktiota f' ohjausfunktioksi. Algoritmi A* on esitetty kuvassa 4.6.

```
Pane alkutila s listaan AVOIN;
g'(s) := 0; f'(s) := h'(s);
while AVOIN ei-tyhjä ja ei lopetuskäskyä do
    Poista listasta AVOIN se tila s, jonka f' on pienin;
    Lisää s listaan SULJETTU;
    if s on lopputila then
        Tulosta ratkaisu paluuosoittimia pitkin ja lopeta
    else
        Generoi s:n seuraajat s_i
        for s_i do
            g''(s_i) := g'(s_i) + w(s, s_i)
            if s_i on uusi tila then
                Lisää s_i listaan AVOIN
                Aseta paluuosoitin s_i:stä s:ään
                g'(s_i) := g''(s_i); f'(s_i) := g'(s_i) + h'(s_i)
            else {s_i listassa AVOIN tai SULJETTU}
                if g''(s_i) < g'(s_i) then
                    Siirrä s_i listaan AVOIN
                Aseta s_i:n paluuosoitin s:ään
                g'(s_i) := g''(s_i); f'(s_i) := g'(s_i) + h'(s_i)
```

Kuva 4.6: A*-algoritmi [26].

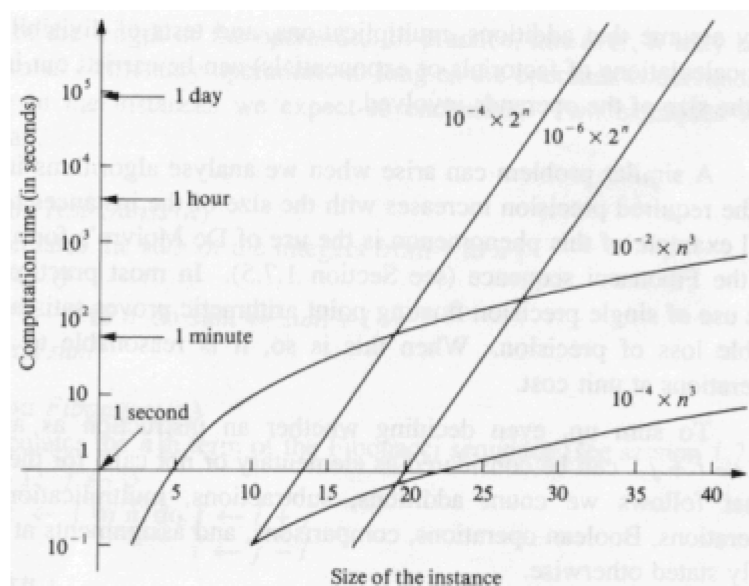
Etsitään polkua solmujen A ja M välillä kuvan 4.3 esimerkkiverkossa. Käytetään heuristisena funktiona solmujen linnuntietä mitattuja etäisyyksiä solmuun M. Etäisyydet on esitetty taulukossa 4.1. Algoritmin suoritus palauttaa ratkaisuna polun, joka kulkee seuraavien solmujen kautta: A, D, G, I, J, M.

Taulukko 4.1: Linnuntietä mitatut etäisyydet kaikista muista solmuista solmuun M kuvan 4.3 verkossa.

A	B	C	D	E	F	G	H	I	J	K	L	N
15	12	9	8	7	5	7	6	6	4	7	3	9

4.3 NP-VAIKEAT ONGELMAT

NP-vaikeat ongelmat ovat ongelmia, joiden ratkaisemiseksi ei tunneta tehokasta algoritmia [28, 9]. Optimaalisen ratkaisun löytävät tunnetut algoritmit NP-vaikeisiin ongelmiin ovat aikavaativuudeltaan eksponentiaalisia. Kuvassa 4.7 on tarkasteltu algoritmien suoritusaikoja suhteessa syötteen kokoon. Algoritmien aikavaativuudet on merkitty algoritmeja esittävien viivojen viereen. Kuvasta voi havaita, kuinka aikavaativuudeltaan eksponentiaaliset algoritmit vievät jo pienillä syötteillä kohtuuttoman paljon aikaa, mikäli niitä halutaan hyödyntää käytännön sovelluksissa.



Kuva 4.7: Erilaisten algoritmien suoritusaika suhteessa syötteen kokoon [5].

Kauppamatkustajan ongelma on eräs NP-vaikea verkko-ongelma. Esittelen kauppamatkustajan ongelman kombinatoristen optimointiongelmien yhteydessä. NP-vaikeiden ongelmien luokka on laaja ja se sisältää useita tärkeitä ongelmia, joihin törmätään myös käytännön sovellutuksissa [28].

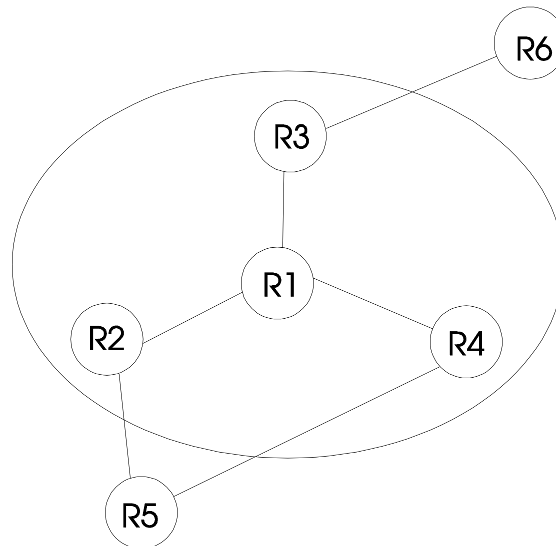
4.4 KOMBINATORISET OPTIMOINTIONGELMAT

Kombinatorinen optimointiongelma on joko *minimointiongelma* tai *maksimointiongelma* ja tavoitteena on löytää toimiva ratkaisu siten, että kyseinen ratkaisu minimoi (tai maksimoi) *optimointifunktion*. Magyar [19] esittää optimointiongelmalle seuraavan kaavan:

$$\min f(x_1, \dots, x_n) \quad \therefore \; x \in F \qquad (4.1)$$

Kaavassa 4.1 x on päätösmuuttuja ja f on optimointifunktio. F on kelvollisten ratkaisujen joukko. Kombinatorisilla optimointiongelmilla tarkoitetaan ongelmia, joissa täytyy löytää optimaalinen järjestely tai järjestys ongelman elementeille.

Mikäli tehtävä on luonteeltaan maksimointitehtävä, voidaan se myös esittää minimointitehtävänä min $-f(x)$. Mikäli muuttujista ja optimointifunktiosta ei oleteta mitään etukäteen, on kyseessä *globaali optimointiongelma*. *Lokaalisti optimaalisia* ratkaisuja voidaan määritellä naapuristofunktion avulla. Ratkaisu on lokaalisti optimaalinen silloin, kun se on parempi kuin kaikki muut ratkaisut kyseisessä naapuristossa. Lokaalioptimoinnilla tarkoitetaan yksittäisten ratkaisujen korvaamista lokaalisti optimaalisilla ratkaisuilla. Kuvassa 4.8 ratkaisun *R1* naapuristo on erotettu ympyrällä. Ympyrän sisälle jäävät viivat ilmaisevat *R1*:lle naapuristofunktion avulla löydetyt naapuriratkaisut.

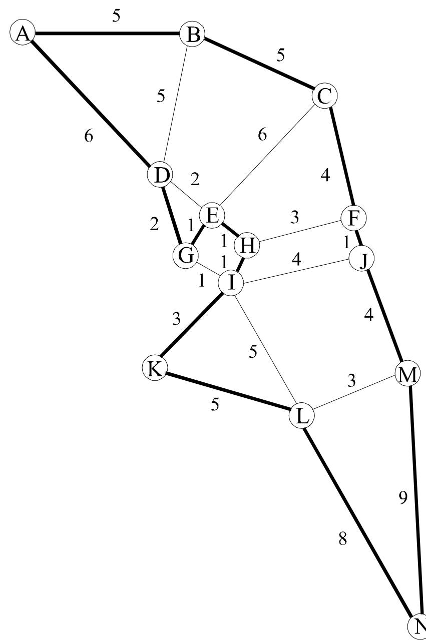


Kuva 4.8: Naapuristo.

Kauppamatkustajan ongelma [10] on kombinatorinen optimointiongelma, jossa syötteenä on joukko kaupunkeja, joista on olemassa polkuja toisiin kaupunkeihin. Jokaiselle polulle on määritelty sen pituus. Kauppamatkustajan tehtävänä on käydä jokaisessa kaupungissa kerran ja palata tämän jälkeen siihen kaupunkiin, josta hän aloitti matkansa. Tavoitteena on löytää kauppamatkustajalle sellainen reitti, jota kulkemalla hänen kokonaismatkansa on mahdollisimman pieni.

Kauppamatkustajan ongelman kuvaamisessa käytetään yleensä verkkoa ja siihen liittyviä käsitteitä kuten polku ja *osapolku*. Verkon solmujen jono $(v_1, v_2, \ldots, v_n)$ on polku, jos kaikkien jonossa peräkkäin sijaitsevien solmujen välillä on olemassa kaari. Kauppamatkustajan ongelmassa polun tulee sisältää lisäksi kaari jonon ensimmäisen ja viimeisen solmun välillä ollakseen kelvollinen. Jonkin tietyn polun osapolku on solmujen jono, joka löytyy kokonaisuudessaan samassa järjestyksessä kyseisestä polusta. Esimerkiksi polun $(v_1, v_2, \ldots, v_6)$ osapolkuja ovat mm. polut (v_1, v_2) ja (v_2, v_3, v_4, v_5).

Kauppamatkustaja-ongelma on kombinatorinen optimointiongelma, jossa tehtävänä on löytää sellainen solmujen järjestys (permutaatio), joka on laillinen kauppamatkustajapolku, ja joka minimoi polun pituuden. Kuvassa 4.9 on esitetty kauppamatkustajan ongelman esimerkki-instanssi.



Kuva 4.9: Kauppamatkustajan ongelma esimerkkiverkossa
(yksi laillinen kauppamatkustajapolku on esitetty vahvennettujen kaarien avulla).

Yksittäinen ratkaisu voidaan esittää seuraavasti: *A-B-C-D-E-F-G-H-I-J-K-L-M-N*. Tässä esityksessä luvut esittävät solmuja ja väliviivat esittävät kaaria. Ratkaisu on kelvollinen, jos jokaisen perättäisen solmun välissä on kaari ja lisäksi on olemassa kaari viimeisestä solmusta ensimmäiseen solmuun. Täten kuvan 4.9 tapauksessa kelvollinen ratkaisu on esimerkiksi permutaatio *A-B-C-F-J-M-N-L-K-I-H-E-G-D* ja kelvoton ratkaisu esimerkiksi *A-C-F-J-M-L-N-K-I-H-E-G-D-B*.

Teoksessa [24] mainitaan kolme vektoriesitystä, joita voidaan käyttää kauppamatkustajan ongelman ratkaisujen koodaamisessa. Nämä ovat *naapuruuteen perustuva esitys* (*adjacency representation*), *järjestykseen perustuva esitys* (*ordinal representation*) ja *polkuesitys* (*path representation*). Esittelen seuraavaksi lyhyesti nämä kolme vektoriesitystä. Esitystapojen analysointia ja vertailua on käsitelty teoksessa [24].

Naapuruuteen perustuvassa esityksessä ratkaisu esitetään kaupunkien listana siten, että kaupunki j sijaitsee listan kohdassa i jos ja vain jos kierros kulkee kaupungista i kaupunkiin j. esimerkiksi lista (2 4 8 3 9 7 1 5 6) esittää ratkaisua 1-2-4-3-8-5-9-6-7. Naapuruuteen perustuvassa esityksessä listat voivat esittää myös epäkelpoja ratkaisuja. Esimerkiksi lista (2 4 8 1 9 3 5 7 6) esittää ratkaisua, joka sisältää ennenaikaisen kehän 1-2-4-1.

Järjestykseen perustuvassa esityksessä ratkaisu esitetään kaupunkien listana. Esityksessä käytetään järjestettyä listaa C, joka toimii *viitattavana kohteena* (*reference point*). Viitattava kohde voi olla esimerkiksi lista (1 2 3 4 5 6 7 8 9). Kierros 1-2-4-3-5-6-7-8-9 esitetään *viittausten listana* l = (1 1 2 1 4 1 3 1 1), jota tulkitaan seuraavasti: ensimmäinen luku listassa l on 1, joten otetaan ensimmäinen luku listasta C ratkaisun ensimmäiseksi kaupungiksi ja poistetaan luku listasta C. Osittainen ratkaisu on nyt 1. Seuraava luku listassa on 1, joten otetaan ensimmäinen luku listasta C ratkaisun toiseksi kaupungiksi ja poistetaan luku listasta C. Osittainen ratkaisu on nyt 1-2. Seuraava luku listassa on 2, joten otetaan toinen luku listasta C ratkaisun kolmanneksi kaupungiksi ja poistetaan luku listasta C. Osittainen ratkaisu on nyt 1-2-4. tätä menetelmää toistetaan, kunnes lista l on käyty loppuun. Lopullinen ratkaisu esimerkkitapauksessa on 1-2-4-3-8-5-9-6-7.

Polkuesityksessä ratkaisu 5-1-7-8-9-4-6-2-3 esitetään yksinkertaisesti kaupunkien listana (5 1 7 8 9 4 6 2 3).

Edellä kuvattujen vektoriesitysten lisäksi kauppamatkustajan ongelman ratkaisu voidaan esittää myös matriisina [24]. Esimerkiksi ratkaisu 3-1-2-8-7-4-6-9-5 voidaan esittää kuvan 4.10 matriisina. Kuvan 4.10 matriisilla M on seuraavat ominaisuudet (n on kaupunkien lukumäärä):

1. Ykkösten määrä on tasan $\frac{n(n-1)}{2}$,
2. $m_{ii} = 0$ kaikille $1 \leq i \leq n$ ja
3. jos $m_{ij} = 1$ ja $m_{jk} = 1$ silloin $m_{ik} = 1$.

	1	2	3	4	5	6	7	8	9
1	0	1	0	1	1	1	1	1	1
2	0	0	0	1	1	1	1	1	1
3	1	1	0	1	1	1	1	1	1
4	0	0	0	0	1	1	0	0	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1
7	0	0	0	1	1	1	0	0	1
8	0	0	0	1	1	1	1	0	1
9	0	0	0	0	1	0	0	0	0

Kuva 4.10: Ratkaisun 3-1-2-8-7-4-6-9-5 matriisiesitys.

4.5 LÄHESTYMISTAPOJA KOMBINATORISIIN OPTIMOINTIONGELMIIN

Lähestymistavat kombinatorisiin optimointiongelmiin voidaan jakaa kolmeen luokkaan [1]: *Tarkat algoritmit, likimääräisalgoritmit* ja *heuristiset algoritmit.*

Tarkat algoritmit takaavat globaalisti optimaalisen ratkaisun löytymisen ja ovat aikavaativuudeltaan pahimmassa tapauksessa eksponentiaalisia. Tunnetuimpia tällaisia algoritmeja on *täydellinen haku*, jossa käydään systemaattisesti läpi kaikki mahdolliset kelvolliset ratkaisut. Useita kehittyneempiä ratkaisuja on laadittu, mutta ne ovat edelleen pahimmassa tapauksessa eksponentiaalisia.

Likimääräisalgoritmit eivät takaa optimaalisuutta, vaan ne takaavat, että ratkaisun hyvyys on tietyn raja-arvon päässä optimaalisesta ratkaisusta. Tämä saavutetaan yleensä polynomisessa ajassa. Tällaisia algoritmeja ovat ε-likimääräisalgoritmit.

Algoritmi on heuristinen, mikäli se tuottaa yleensä hyvän (lähes optimaalisen) tuloksen vaikeaan optimointiongelmaan kohtuullisessa ajassa. Heuristiset algoritmit eivät kuitenkaan anna mitään takuita saavuttamansa ratkaisun hyvyydestä, vaan ne voivat tuottaa myös huonoja

ratkaisuja. Heuristisilla algoritmeilla pyritään ratkomaan vaikeita kombinatorisia optimointiongelmia tehokkaasti. Yleisesti käytettyihin heuristisiin algoritmeihin lukeutuvat mm. *lokaalihaku* ja *evolutionaariset algoritmit*.

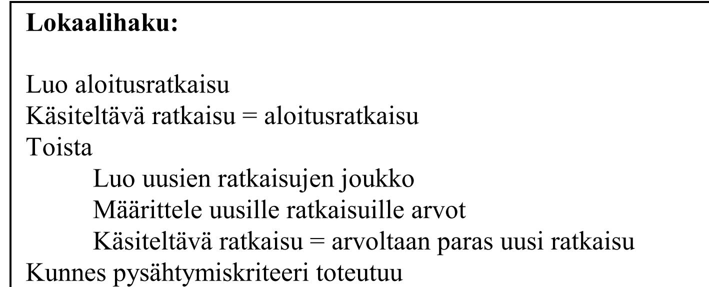
4.6 ALGORITMEJA KOMBINATORISTEN OPTIMOINTIONGELMIEN RATKAISEMISEEN

Tässä luvussa esittelen *lokaalihaun* periaatteen. Tämän lisäksi tarkastelen kolmea lokaalihaun variaatiota: *simuloitua jäähdytystä*, *tabuhakua* ja geneettistä algoritmia.

4.6.1 LOKAALIHAKU

Luultavasti yksinkertaisin ja yllättävän tehokas heuristinen algoritmi kombinatoristen optimointiongelmien ratkaisemiseen on lokaalihaku. Lokaalihaulla on pitkä historia kombinatorisessa optimoinnissa jossa sitä on käytetty 1950-luvulta lähtien [1]. Lokaalihakua käytettiin ensimmäiseksi kauppamatkustajan ongelmaan ja myöhemmin sitä alettiin käyttää myös *verkon partitiointiin* ja *erilaisiin permutointitehtäviin*. Tietokoneiden suorituskyvyn kasvu sekä kehittyneet tietorakenteet ovat tehneet lokaalihakualgoritmeista vakavasti otettavan vaihtoehdon valittaessa algoritmia, jonka avulla pyritään ratkomaan suuria ongelmainstansseja. Joustavuus ja toteutuksen helppous ovat mahdollistaneet monien monimutkaisten ongelmien käsittelemisen lokaalihakualgoritmien avulla. Käytännön kokemusten kautta on havaittu, että lokaalihakualgoritmit löytävät hyviä ratkaisuja useisiin ongelmiin kohtuullisessa ajassa [1].

Lokaalihaku perustuu yritykseen ja erehdykseen. Perusidea on yksinkertainen ja helppo toteuttaa erilaisiin ongelmiin. Lokaalihaun rakenne on esitetty kuvassa 4.11.


Lokaalihaku:

Luo aloitusratkaisu
Käsiteltävä ratkaisu = aloitusratkaisu
Toista
    Luo uusien ratkaisujen joukko
    Määrittele uusille ratkaisuille arvot
    Käsiteltävä ratkaisu = arvoltaan paras uusi ratkaisu
Kunnes pysähtymiskriteeri toteutuu

Kuva 4.11: Lokaalihaku.

Lokaalihaussa luodaan ensin aloitusratkaisu. Tästä ratkaisusta luodaan pienten muutosten avulla mahdollisten uusien ratkaisujen joukko. Luodut uudet ratkaisut ovat käsiteltävän ratkaisun naapuriratkaisuja. Nämä uudet ratkaisut arvioidaan ja niistä paras valitaan uudeksi käsiteltäväksi ratkaisuksi. Tätä toistetaan joko ennalta päätetty määrä tai kunnes ennalta määrätty pysähtymiskriteeri toteutuu.

Lokaalihaun eräs erikoistapaus on *Hill-climbing*, jossa uudet ratkaisut luodaan siten, että ne ovat aina joko parempia tai yhtä hyviä kuin nykyinen ratkaisu [17]. Täten algoritmi etenee jatkuvasti kohti parempia ratkaisuja eli lähenee lokaalia optimia. Algoritmin heikkous on siinä, että löydettyään lokaalin optimin se ei enää pyri etenemään kohti muualla hakuavaruudessa sijaitsevia, mahdollisesti parempia ratkaisuja.

Lokaalihausta on kehitelty useita erilaisia variaatioita, joista suuri osa perustuu analogioihin luonnon prosesseista. Esimerkiksi simuloitu jäähdytys ja geneettiset algoritmit kuuluvat tähän kategoriaan. Myös tabuhaku on eräs lokaalihaun variaatio.

4.6.2 SIMULOITU JÄÄHDYTYS

Simuloitu jäähdytys [1] perustuu fysikaaliseen jäähdytysprosessiin, jossa kiinteän aineen rakennetta parannetaan ensin sulattamalla aine ja sen jälkeen jäähdyttämällä se hitaasti takaisin kiinteään olomuotoon. Simuloitu jäähdytys -algoritmissa aloitusratkaisua pyritään parantamaan asteittain. Algoritmi on aluksi tilassa, jossa se jollain todennäköisyydellä hyväksyy uudeksi ratkaisuksi myös huonoja ratkaisuja, eli ratkaisuja, jotka ovat kustannukseltaan edellistä ratkaisua kalliimpia. Tätä todennäköisyyttä kuvaa

jäähdytysprosessin lämpötila. Hyvät ratkaisut hyväksytään uudeksi ratkaisuksi aina kun ne valitaan. Todennäköisyys, jolla huonoja ratkaisuja valitaan, laskee asteittain ja sitä kontrolloidaan parametreilla, jotka määritellään *jäähdytysaikataulun* (*cooling schedule*) avulla. Lopulta tämä todennäköisyys laskee nollaan, jonka jälkeen yleensä etsitään vielä lokaalihaun avulla lokaalisti optimaalinen ratkaisu. Simuloidun jäähdytyksen toimintaperiaate on esitetty kuvassa 4.12.

```
Simuloitu jäähdytys:
Määrittele jäähdytysaikataulu
Valitse aloituslämpötila
Luo aloitusratkaisu
Toista
    Toista
        Valitse käsiteltävän ratkaisun satunnainen naapuriratkaisu
        Mikäli valittu ratkaisu on parempi kuin käsiteltävä ratkaisu
            Käsiteltävä ratkaisu = valittu ratkaisu
        Muuten
            Lämpötilan määrittämällä todennäköisyydellä
            käsiteltävä ratkaisu = valittu ratkaisu
        Kunnes jäähdytysaikataulun mukainen määrä iteraatioita tulee täyteen
        Laske lämpötilaa jäähdytysaikataulun mukaisesti
Kunnes lämpötila == 0
```

Kuva 4.12: Simuloitu jäähdytys.

Simuloitu jäähdytys löytää sitä parempia ratkaisuja, mitä korkeampi aloituslämpötila on ja mitä hitaammin lämpötilaa lasketaan. Nämä kuitenkin myös kasvattavat voimakkaasti algoritmin suoritusaikaa. Simuloitua jäähdytystä käytetään normaalisti monien muiden lokaalihakualgoritmien tavoin heuristiikkana, jolloin jäähdytysaikataulua voidaan niin halutessa nopeuttaa huomattavasti.

4.6.3 TABUHAKU

Tabuhaku on iteratiivinen parannusmetodi, joka hyväksyy myös kustannukseltaan huonompia ratkaisuja [1]. Tabuhaku perustuu *tabulistan* hyödyntämiseen. Tabulista on algoritmin suorituksen aikana päivitettävä kiinteän kokoinen lista, joka on yleensä kohtalaisen lyhyt. Tabulista sisältää yleensä noin 10-20 ratkaisua, jotka eivät ole käytettävissä muutaman

seuraavan iteraation aikana. Hyväksymällä kustannukseltaan huonompia ratkaisuja etsinnälle annetaan mahdollisuus ajautua pois lokaalista minimistä, mikä puolestaan mahdollistaa etsintäavaruuden laajemman läpikäymisen. Seuraavaksi ratkaisuksi valitaan aina nykyisen ratkaisun paras naapuri, myös silloin kun tämä naapuriratkaisu on huonompi kuin nykyinen ratkaisu. Valittava ratkaisu ei saa kuitenkaan sisältyä ylläpidettävään tabulistaan. Tabulistaan lisätään valittu uusi ratkaisu ja siitä poistetaan ratkaisu, joka on ollut tabulistalla pisimmän aikaa. Tabulistan avulla pyritään estämään se, että haku etsiytyisi aina takaisin samaan lokaaliin optimiin. Tabuhaun toimintaperiaate on esitetty kuvassa 4.13.

```
Tabuhaku:
Luo aloitusratkaisu
Luo tabulista = tyhjä lista
Käsiteltävä ratkaisu = aloitusratkaisu
Toista
        Valitse paras tabulistaan kuulumaton naapuriratkaisu
        Käsiteltävä ratkaisu = valittu naapuriratkaisu
        Päivitä tabulistaa
Kunnes pysähtymiskriteeri toteutuu
```

kuva 4.13: Tabuhaku.

4.6.4 GENEETTINEN ALGORITMI

Geneettiset algoritmit [1, 18] ovat eräs lokaalihaun etsintästrategia edellä mainittujen simuloidun jäähdytyksen ja tabuhaun ohella. Perusluonteeltaan geneettiset algoritmit eroavat näistä etsintästrategioista siinä, että niillä käsitellään kandidaattiratkaisujen *populaatiota* yksittäisen ratkaisun parantamisen sijaan. Geneettiset algoritmit ovat populaatioon perustuvia heuristiikkoja ja ne ovat saaneet inspiraationsa luonnon evoluutiosta. Geneettiset algoritmit imitoivat luonnonvalintaa ja pyrkivät muokkaamaan ratkaisujen populaatiota siten, että vahvat (mahdollisimman optimaaliset) ratkaisut säilyvät seuraavaan populaatioon. Tällä tavoin populaation geeniaines vahvistuu ja hakua suoritetaan jatkuvasti optimaalisempien ratkaisujen keskuudessa.

Geneettisen algoritmin alussa on joukko erilaisia ratkaisuja. Nämä ratkaisut muodostavat populaation ja niitä kutsutaan usein *kromosomeiksi.* Jokaisen iteraation aikana muodostetaan

uusia ratkaisuja geneettisten operaatioiden avulla. Nämä operaatiot ovat *risteytys* (*crossover*) ja *mutaatio* (*mutation*). Risteytystä käytettäessä uusia ratkaisuja muodostetaan yhdistämällä kaksi olemassa olevaa ratkaisua. Mutaation avulla puolestaan tehdään pieniä muutoksia olemassa oleviin ratkaisuihin. Tämän jälkeen ratkaisuille lasketaan *soveltuvuusarvo* (*fitness value*) ja valitaan käsiteltävän populaation ja uusien ratkaisujen joukosta parhaat ratkaisut. Nämä ratkaisut siirtyvät seuraavaan populaatioon ja ovat siten mukana seuraavalla iteraatiokierroksella. Geneettisen algoritmin toimintaperiaate on esitetty kuvassa 4.14.

Geneettinen algoritmi:
Luo aloitusratkaisujen populaatio
käsiteltävä populaatio = aloitusratkaisujen populaatio
Toista
    Luo uusia ratkaisuja risteyttämällä
    Sovella mutaatiota uusiin ratkaisuihin (valinnainen)
    Laske uusille ratkaisuille soveltuvuusarvot
    Valitse soveltuvuusarvoltaan parhaat ratkaisut
    Käsiteltävä populaatio = valitut parhaat ratkaisut
Kunnes pysähtymiskriteeri toteutuu

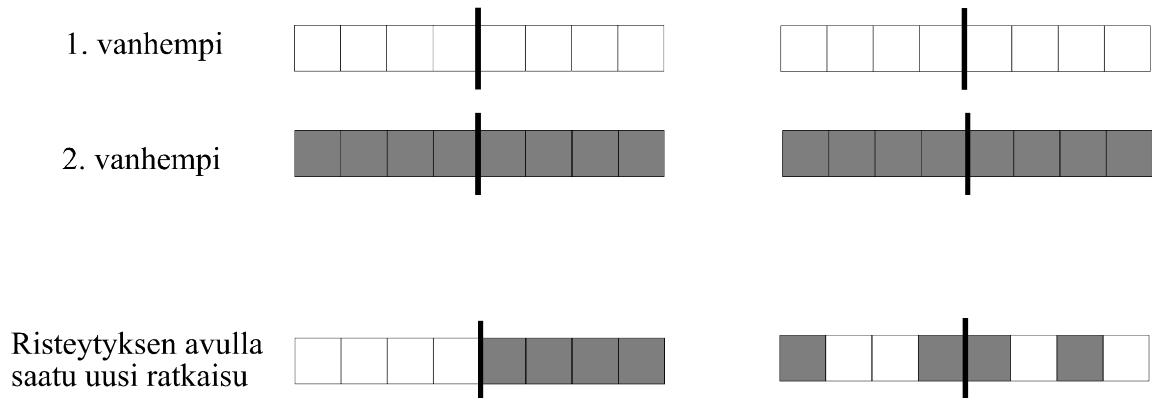
Kuva 4.14: Geneettinen algoritmi.

Yksilöitä populaatiossa arvioidaan soveltuvuusarvon avulla. Soveltuvuusarvo lasketaan soveltuvuusfunktion (fitness function) avulla ja se muodostaa perustan valintamekanismille. Soveltuvuusfunktio on käytännössä optimointifunktio f, jonka avulla pyritään ohjaamaan algoritmin etenemistä haluttuun suuntaan. Soveltuvuusarvo sinänsä ei ole oleellista, vaan lopputuloksena saatava ratkaisu kyseiselle ongelmainstanssille. Soveltuvuusarvoa tarvitaan ainoastaan algoritmin suorituksen aikana ohjaamaan hakua kohti parempia ratkaisuja.

Risteytys on geneettisen algoritmin keskeinen vaihe. Siinä uusia ratkaisuja luodaan risteyttämällä olemassaolevia ratkaisuja keskenään. Yleensä risteytettäviä ratkaisuja on kaksi kappaletta mutta myös useampia ratkaisuja on mahdollista risteyttää keskenään.

Kahden ratkaisun risteyttämisen voi suorittaa usealla eri tavalla. Erilaisia risteyttämisoperaatioita ovat esimerkiksi kuvassa 4.15 esitetyt *satunnainen risteytys* (*random*

crossover) ja *järjestykseen perustuva risteytys* (*order based crossover*), sijaintiin perustuva risteytys (*position based crossover*) ja *yhdenmukainen risteytys* (*uniform crossover*).

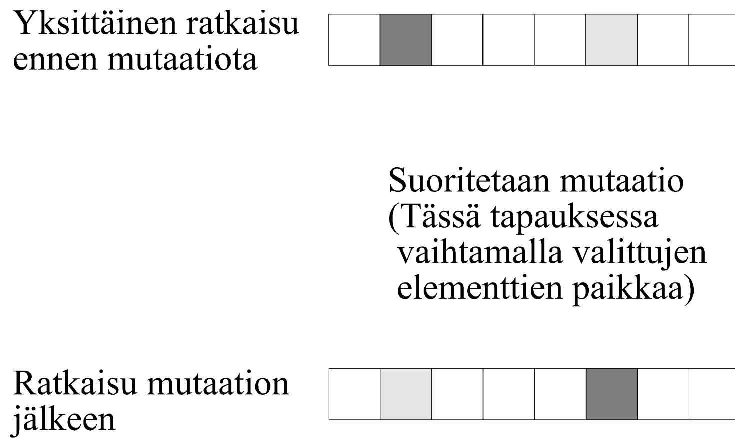


Kuva 4.15: Järjestykseen perustuva risteytys (vasemmalla) ja satunnainen risteytys (oikealla).

Satunnaisessa risteyttämisessä ratkaisu luodaan siten, että valitaan ensin toiselta vanhemmalta sattumanvaraisesti puolet kyseisen ratkaisun elementeistä uuteen ratkaisuun ja tämän jälkeen suoritetaan sama toimenpide toiselle vanhemmalle. Tämän menetelmän ongelma on siinä, että saatu uusi ratkaisu voi sisältää joitakin elementtejä useaan kertaan, ja jotkut elementit voivat puolestaan puuttua uudesta ratkaisusta kokonaan.

Yhdenmukainen risteyttäminen suoritetaan siten, että määritellään jokin todennäköisyys, jonka perusteella päätetään jokaisen elementin kohdalla, kummalta vanhemmalta kyseinen elementti peritään. Mikäli todennäköisyys on 0.5, on kummankin vanhemman elementeillä yhtä suuri todennäköisyys tulla valituksi uuden ratkaisun elementiksi.

Mutaatiossa (kuva 4.16) jonkin ratkaisun yksittäisiä osia korvataan joillakin erilaisilla mutta valideilla osilla. Mutaatio, jota voi tapahtua yleensä pienellä todennäköisyydellä, mahdollistaa haun siirtymisen uusille tutkimattomille alueille hakuavaruudessa. Kaikki geneettiset algoritmit eivät käytä mutaatiota. Yleensä parhaimpia ratkaisuja saadaan käyttämällä sekä risteytystä että mutaatiota algoritmin suorituksen aikana [1]. *Mutaatiotaajuus* (*mutation rate*) ilmaisee, kuinka voimakasta mutaatio on silloin, kun mutaatiota käytetään.



Kuva 4.16: Mutaatio.

4.7 GENEETTISEN ALGORITMIN SOVELTAMINEN KAUPPAMATKUSTAJAN ONGELMAAN

Geneettisen algoritmin kohdalla risteytysmenetelmän valinta on erittäin tärkeää. Yksinkertainen lähestymistapa on käyttää satunnaista risteyttämistä, jossa kaksi vanhempaa ratkaisua yhdistetään ottamalla satunnaisesti valittu puoli yhdeltä vanhemmalta ja loput toiselta vanhemmalta. Tämä ei kuitenkaan välttämättä tuota kelvollista uutta ratkaisua käsiteltävään ongelmaan. Kauppamatkustajan ongelmassa voi yllä kuvattu menetelmä tuottaa uusia ratkaisuja, joissa sama solmu (kaupunki) esiintyy useammassa eri kohdassa permutaatiota, kun taas jokin toinen solmu ei esiinny lainkaan. Kelvollisia ratkaisuja voidaan kuitenkin generoida seuraavasti: valitaan ensimmäiseltä vanhemmalta osapolku, jonka pituus on $N/2$ (N = ratkaisun sisältämien elementtien määrä). Tämän jälkeen etsitään osapolusta puuttuvat solmut toiselta vanhemmalta ja yhdistetään ne ensimmäiseltä vanhemmalta saatuun osapolkuun. Pahimmassa tapauksessa kuitenkin toiselta vanhemmalta saadut solmut eivät muodosta osapolkuja vaan koko polku joudutaan muodostamaan useiden yhdistämisoperaatioiden avulla [17].

Kuvan 4.9 ongelmainstanssin kohdalla satunnaista risteyttämistä voidaan soveltaa seuraavasti: ensimmäinen vanhempi on polku *A-B-C-F-M-N-L-K-I-H-E-G-D* ja siitä valitaan *osapolku A-B-C-F-M-N*. Toinen vanhempi on *A-B-C-E-H-F-M-N-L-K-I-G-D*. Tässä tapauksessa toiselta

vanhemmalta saatavat solmut *L*, *K*, *I*, *H*, *E*, *G* ja *D* muodostavat kaksi osapolkua, joten kokonainen uusi polku voidaan muodostaa kahden yhdistämisoperaation avulla. Toinen geneettiseen algoritmiin liittyvä ongelma on, miten huomioidaan ja käsitellään populaatiossa mahdollisesti sijaitsevat saman ratkaisun useammat esiintymät. Esimerkiksi populaation *yksilöt A-B-C-D-E-F-G-H-I-J-K-M-N*, *B-C-D-E-F-G-H-I-J-K-M-N-A* ja *C-D-E-F-G-H-I-J-K-M-N-A-B* kuvaavat samaa ratkaisua ja voisi kuvitella, että kaikkien kolmen yksilön pitäminen populaatiossa heikentäisi populaation vaihtelua ja täten hidastaisi siinä tapahtuvan evoluution nopeutta.

4.7.2 BRADYN ALGORITMI

Eräs tunnettu kauppamatkustajan ongelmaan suunniteltu algoritmi on Bradyn algoritmi [4]. Brady käytti risteytysmenetelmää, jossa tutkitaan risteytettäviä ratkaisuja kunnes löydetään molemmissa esiintyvä osapolku, joka sisältää täsmälleen samat kaupungit mutta eri järjestyksessä. Uusi ratkaisu luodaan poistamalla osapoluista pidempi ja korvaamalla se vastaavalla lyhyemmällä osapolulla.

Lokaalioptimoinnissa Brady käytti seuraavaa menetelmää. Hän valitsi satunnaisesti jonkin osapolun tutkittavasta ratkaisusta. Osapolussa olevien solmujen läpikäyntijärjestys käännettiin, mikäli tämä lyhensi ratkaisun pituutta. Mahdollisesti käännettäviä polkuja etsittiin ennalta päätetty määrä.

Brady kokeili useita eri tapoja, joilla hän yhdisti risteytystä ja lokaalioptimointia. Parhaimmat tulokset hän saavutti seuraavalla menetelmällä. Lokaalioptimoinnissa Brady ei valinnut osapolkuja satunnaisesti vaan etsi systemaattisesti osapolkuja, joiden kääntäminen lyhensi ratkaisun pituutta. Lokaalioptimointia suoritettiin niin kauan, kuin tällaisia osapolkuja löytyi. Lokaalioptimoinnin jälkeen Brady sovelsi risteytystä, jonka jälkeen suoritettiin jälleen lokaalioptimointi ennen seuraava risteytystä. Populaatiossa, jossa oli 12 ratkaisua, kaikki ratkaisut otettiin mukaan risteytykseen ja parit valittiin satunnaisesti. Algoritmin testauksesta saadut tulokset osoittivat, että Bradyn algoritmi oli lupaava ja löysi hyviä ratkaisuja kohtuullisessa ajassa.

5. YHTEENVETO

Tässä tutkielmassa olen tarkastellut vektorikarttojen hyödyntämistä paikkatietosovelluksissa. Tarkastelussa olen keskittynyt rasteri- ja vektoriformaattien vertailuun karttojen tallentamisessa, vektorikarttojen rasterointiin sekä vektorikartoista muunnettujen verkkojen hyödyntämiseen.

Rasteri- ja vektoriformaatit mahdollistavat kaksi erilaista tapaa karttojen tallentamiseen. Tutkielmassa suorittamani tarkastelun mukaan vektoriformaatti sopii parhaiten karttojen tallentamiseen tietokantaan. Vektoriformaatissa kartta on talletettu siten, että sitä voidaan laadun heikkenemättä tarkastella eri mittakaavoilla. Lisäksi vektorimuodossa oleva kartan muuntaminen rasteriksi on suoraviivainen tehtävä. Henkilökohtaiseen navigointiin liittyvissä sovelluksissa, joissa aika- ja tilaresurssit ovat rajalliset, on käytettävän tallennusformaatin tilantarpeella suuri merkitys. Rasteriformaattien vahvuus on niiden pieni tilantarve, etenkin mikäli käytetään kuvantiivistämismenetelmiä.

Vektorikuvien muuntaminen rasterikuviksi on suoraviivainen ja kohtalaisen yksinkertainen tehtävä, jota varten on kehitelty useita tehokkaita ja helposti ymmärrettäviä algoritmeja. Mikäli muuntaminen tapahtuu rasterikuvaksi, joka ei ole binäärikuva, voidaan rasterikuvan laatua parantaa antialiasointimenetelmillä. Rasteroinnin vaativimpia osuuksia on tekstin ja mahdollisten erikoismerkkien kuten topologisten merkkien muuntaminen. Mikäli käytetään bittikarttamenetelmää, ovat bittikartastot sidottuja siihen resoluutioon, jolle ne on määritelty. Mikäli vektorikuvia rasteroidaan usealla eri resoluutiolla, joudutaan tekemään valinta bittikartastojen suuren määrän ja syntyvien rasterikuvien laadun heikkenemisen välillä. Toteuttamassamme ohjelmassa käytimme bittikartastoja topologisten merkkien rasterointiin. Määrittelimme ainoastaan yhden bittikartaston, jota käytimme kaikilla resoluutioilla. Tekstin rasteroinnin toteutimme Hersheyn vektorifonttien avulla, jonka havaitsimme käyttökelpoiseksi menetelmäksi silloin kun erilaisia fonttityylejä ei tarvitse rasteroida alkuperäisessä ulkoasussaan.

Useat vektorikartat sisältävät erilaisia verkkoja. Mikäli näitä verkkoja esittävät vektorit muunnetaan verkkorakenteeksi, voidaan paikkatietosovelluksiin liittyvien verkko-ongelmien ratkaisemisessa hyödyntää laajaa olemassa olevaa kirjallisuutta. Vektoreiden muuntaminen

verkoksi on kuitenkin ongelmallista. Suurin ongelma on verkkoja esittävän vektoriaineiston erottaminen muusta vektoriaineistosta. Tämän ongelman ratkaisemiseksi tarvitaan tietoa vektorikartan sisäisestä talletusrakenteesta. Toinen ongelma vektoreiden muuntamisessa verkoiksi on mahdollisten epätarkkuuksien poistaminen, joka on erittäin paljon laskentaa vaativa tehtävä. Epätarkkuuksien poistamiseksi on olemassa erilaisia menetelmiä, joiden avulla tehtävään liittyvää laskentaa voidaan vähentää.

Verkko-ongelmia voidaan tarkastella ongelmien ratkaisemiseen suunniteltujen algoritmien aikavaativuuksien kautta. Myös kombinatorisen optimointiongelman käsite tarjoaa lähestymistavan verkko-ongelmiin. Paikkatietosovelluksiin liittyvät ongelmat vaihtelevat helposti ratkaistavista ongelmista NP-vaikeisiin ongelmiin. Kombinatoristen optimointiongelmien ratkaisemiseksi on kehitelty heuristisia algoritmeja, jotka löytävät lähes optimaalisen ratkaisun kohtuullisessa ajassa. Heurististen algoritmien avulla myös NP-vaikeille kombinatorisille optimointiongelmille voidaan tuottaa ratkaisuja siten, että algoritmeja voidaan käyttää käytännön sovelluksissa.

VIITTEET

[1] E. Aarts & K. Lenstra (toim.), *Local Search in Combinatorial Optimization*. John Wiley & sons., Chichester, 1997.

[2] R. K. Ahuja, T. L. Magnanti & J.B Orlin, *Network Flows - Theory, Algorithms and Applications*. Prentice-Hall Inc., New Jersey, 1993.

[3] P. Bourke, *Hershey Vector Font*. Internet WWW-sivu: `http://astronomy.swin.edu.au/pbourke/other/hershey/index.html` (19.11. 2001).

[4] R. M. Brady, Optimization strategies gleaned from biological evolution. *Nature* **317**, 804-806, 1995.

[5] G. Brassard & P. Bratley, *Algorithmics - Theory and Practise*. Prentice-Hall International Inc., Englewood Cliffs, 1988.

[6] C. W. Brown & B. J. Shepherd, *Graphics File Formats - Reference and Guide*. Manning Publications Co., Greenwich, 1995.

[7] P. A. Burroughs & R. A. McDonnell, *Principles of Geographical Information Systems*. Oxford University Press Inc., New York, 1998.

[8] CEN/TC, *Geographic information – vocabulary*. Internet WWW-sivu:

`http://comelec.afnor.fr/servlet/ServletForum?form_name=cForumPage&file_name=Z13C%2FPUBLIC%2FWEB%2FENGLISH%2Fpren.htm&login=invite&password=invite`
(4. 12. 2001).

[9] T. H. Cormen, C. E. Leiserson & R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[10] A. Croes, A method for solving traveling salesman problems. *Operations Research* **6**, 791-812, 1958.

[11] L. Dumaine, *Vector Map Tools Dedicated to Mobile Devices Context*, lisensiaatintutkielma, University Joseph Fourier, Grenoble, Ranska, 2001.

[12] ESRI *Shapefile Technical Description*. Environmental Systems Research Institute Inc., 1998.

[13] S. Even, *Graph Algorithms*. Computer Science Press Inc., Rockville, 1979.

[14] J. D. Foley, A. Van Dam, S. K. Feiner & J. F. Hughes, *Computer Graphics - Principles and Practise*. Addison-Wesley Publishing Company Inc., 1996.

[15] J. D. Foley & A. Van Dam, *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Publishing Company Inc., 1983.

[16] P. Fränti, E. Ageenko, P. Kopylov, S. Gröhn & F. Berger, Compression of map images for real-time applications. *Technical Report* A-2001-1, Department of Computer Science, University of Joensuu, Finland, 2001.

[17] P. Fränti, *Heuristic Algorithms for Combinatorial Optimization Problems*, Lecture Notes, Department of Computer Science, University of Joensuu, Finland, 2000.

[18] D. Goldberg, *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, 1989.

[19] H. Magyar, *On Solutions for Some Industrially Motivated Combinatorial Optimization Problems*. Ph.D. Thesis, Turku Centre of Computer Science, Finland, 2000.

[20] Maanmittauslaitos, *Maastotietokanta*. Internet WWW-sivu: `http://www.nls.fi/kartta/selosteet/ts/maastotietokanta.html` (4. 12. 2001).

[21] Maanmittauslaitos, *Paikkatietotekniikan perusteet*. Internet WWW-sivu: `http://www.nls.fi/ptk/perusteet/index.html` (4. 12. 2001).

[22] MapInfo Corporation, MapInfo Professional käsikirja, 1992.

[23] D. Martin, *Geographic Information Systems - Sosioeconomic Applications*. Routledge, Lontoo, 1996.

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

[25] B. M. E. Moret & H. D. Shapiro, *Algorithms from P to NP*. Benjamin/Cummings Publishing Company Inc., Redwood City, 1991.

[26] M. Penttonen, *Tekoäly*, Luentomoniste, Tietojenkäsittelytieteen laitos, Joensuun yliopisto, 1994.

[27] G. W. Stewart, *Introduction to Matrix Computation*. Academic Press Inc, New York, 1973.

[28] R. Sedgewick, *Algorithms in C - Part 5, Graph Algorithms*. Addison-Wesley Publishing Company Inc., 2001.

[29] SoftSource Corp. *Information on SVF (Simple Vector Format)*. Internet WWW-sivu, URL:
`http://www.softsource.com/svf/index.html` (19. 11. 2001).

[30] V. Veis & S. Nevalainen, *Convertor - The software for converting a vector map to the raster format*, Erikoistyö, TKT:n laitos, Joensuun yliopisto, 2001.

[31] V. Veis, *Representation of Digital Maps*, Pro Gradu-tutkielma, Joensuun yliopisto, 2001.

[32] World Wide Web Consortium (W3C), Scalable Vector Graphics (SVG) 1.0 Specification. Internet WWW-sivu:
`http://www.w3.org/TR/SVG/index.html` (19.11.2001).