Rasterikarttojen tiivistäminen

Sami Gröhn

5.11.2002

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Langattomissa päätelaitteissa käytettävien karttojen suurin ongelma on niiden valtava tilantarve. Karttoja mahtuu laitteeseen vain muutama, eikä uusien kokonaisten karttalehtien siirtäminen palvelimelta ole mahdollista reaaliajassa.

Ratkaisuna on karttojen tiivistäminen. Kartat voidaan jakaa semanttisiin komponentteihin, joista kukin komponentti voidaan edelleen jakaa erillisiin lohkoihin. Tällöin koko karttaa ei tarvitse pitää muistissa, vaan tarvittavat kartan osat voidaan tarvittaessa siirtää reaaliaikaisesti päätelaitteeseen.

Tässä tutkielmassa tutustutaan joihinkin kuvantiivistyksen perusmenetelmiin sekä niitä käyttäviin binäärikuvien tiivistysstandardeihin: G4, GIF, JBIG ja JBIG2. Lisäksi esitellään Joensuun yliopistossa erityisesti karttakuville suunniteltu MISS-menetelmä.

Menetelmiä vertaillaan kokeellisesti joukolle karttakuvia erilaisilla päätelaitteilla sekä tiedonsiirtomenetelmillä.

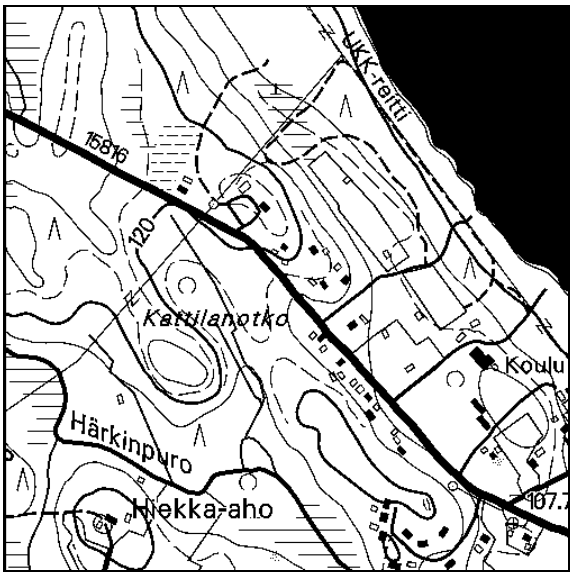
Avainsanat: karttakuvat, häviötön kuvantiivistys, tilastollinen mallintaminen, Huffman-koodaus, aritmeettinen koodaus, jaksonpituuskoodaus, sanakirjamenetelmät, G4, GIF, JBIG, JBIG2.

Sisällys

1 Johdanto

Langattomien päätelaitteiden, kuten kämmenmikrojen ja matkapuhelimien nopea kehitys on mahdollistanut käyttäjille monia uusia palveluja. Eräs näistä on paikannuspalvelut ja niihin liittyvä karttakuvien reaaliaikainen käyttö. Paikannus voidaan suorittaa joko GPS-laitteella [Kaplan 1996] tai matkapuhelinverkossa [Dye ja Buckingham 1999]. Karttojen on oltava käytettävissä reaaliajassa käyttäjän sijainnista riippumatta. Usein käyttäjän päätelaitteessa on lisäksi huomattavia rajoituksia, kuten vähäinen muistin määrä, heikko laskentateho ja hidas tietoliikenneyhteys. Ratkaisuna rajoituksiin on karttojen tiivistäminen, jolloin niiden tilavaatimus ja siten myös siirtoon kuluva aika pienenevät merkittävästi.

Tässä tutkielmassa tarkastellaan rasteripohjaisten karttakuvien tiivistämistä ja käyttöä. Karttakuvien erityisominaisuuksista johtuen ne on tehokkainta jakaa semanttisiin binäärisiin kerroksiin ja tiivistää kukin kerroksista erikseen [Arps ja Truong 1994]. Tällä tavoin kartat muodostuvat joukosta kaksivärisiä *binäärikuvia* (binary image) (kuva 1.1). Esittelen neljä eri periaatteisiin pohjautuvaa binärikuvien tiivistysstandardia [Welch 1984, ITU-T 1988, JBIG 1993, JBIG 1999] sekä eräästä näistä menetelmistä Joensuun yliopistossa edelleen erityisesti karttakuville kehitetyn menetelmän [RealMap 2000]. Tarkastelussa kiinnitetään erityisesti huomiota reaaliaikaisuuteen ja käytettävyyteen päätelaitteissa, joiden muisti- ja suorituskykyresurssit ovat hyvin rajalliset.



Kuva 1.1. Esimerkki binäärisestä karttakuvasta.

Luvussa 2 esitellään joitain kuvantiivistyksen perusmenetelmiä. Tiivistyksen voidaan ajatella muodostuvan kahdesta osasta: *mallinnuksesta* (modeling) ja *koodauksesta* (coding). Luvussa esitellään myös kaksi tiivistysmenetelmää: LZW-algoritmiin [Welch 1984] perustuva GIF [Miano 1999] ja MMR-algoritmia [Yasuda 1980] käyttävä G4 [ITU-T 1988]. Kolmas luku käsittelee digitaalisia karttakuvia. Karttakuvat voivat olla joko *rasteri-* tai *vektorimuotoisia*. Vektorikuvat soveltuvat hyvin skaalaukseen ja niihin voidaan liittää helposti ominaisuustietoa, jonka avulla voidaan esimerkiksi laskea lyhin reitti kahden paikan välillä. Rasterikuvien vahvuuksia puolestaan ovat tehokkaalla tiivistyksellä saatava pieni tilantarve ja eri rasteriformaattien yhteensopivuus. Luvussa 4 esitellään kontekstimalliin ja aritmeettiseen koodaajaan perustuva JBIG [JBIG 1993]. Luvussa 5 esiteltävä JBIG2 [JBIG 1999] on JBIG:n laajennus, jossa kuva jaetaan erityyppisiin osiin, jotka tiivistetään kukin omalla menetelmällään [Howard et al. 1998]. Kuudes luku esittelee JBIG-menetelmistä edelleen kehityn MISS-menetelmän [RealMap 2000], johon on lisätty myös tuki dynaamiselle kartanhallinnalle [Fränti, Kopylov ja Veis 2002]. Seitsemäs luku sisältää kokeellisen osuuden, jossa vertaillaan esiteltävien menetelmien tiivistystehoa keskenään joukolle Maanmittauslaitoksen karttakuvia [Maanmittauslaitos 2002].

2 Häviöttömän tiivistämisen menetelmiä

Häviötön kuvantiivistys voidaan käsittää koostuvan kahdesta osasta: mallintamisesta ja koodauksesta. Mallintamisen tarkoituksena on muodostaa koodaajalle kuvasta malli, jota käyttäen kuva voidaan koodata mahdollisimman tiiviisti. Koska koodaus voidaan suorittaa lähes optimaalisesti aritmeettisella koodaajalla [Rissanen ja Langdon 1979], tiivistämisen suurin ongelma onkin löytää mahdollisimman hyvä mallinnusmenetelmä [Fränti 1999].

Tässä tutkielmassa esiteltävät menetelmät ovat kaikki häviöttömiä, eli purettu kuva on täsmälleen samanlainen kuin ennen tiivistystä.

2.1 Tilastollinen mallintaminen

Tilastollisessa mallinnuksessa koodattavia symboleita pyritään ennustamaan niiden todennäköisyysjakauman perusteella.

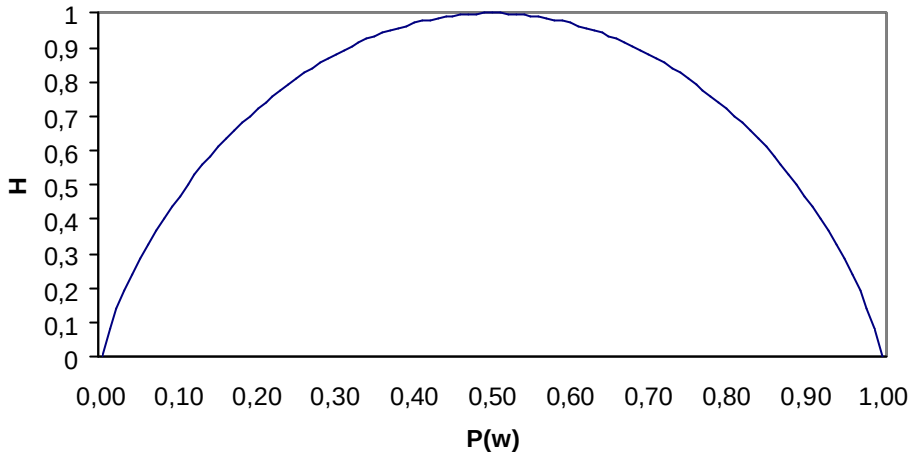
Tiivistyksen tarkoituksena on poistaa kuvan tietosisällön ylimääräinen redundanssi eli toisto. Symbolin *informaatioarvoa* mitataan sen *entropialla* (entropy):

$$H(x) = -\log_2 p(x) \tag{1}$$

missä x on symboli ja $p(x)$ sen esiintymistodennäköisyys. Entropia $H(x)$ kertoo symbolin koodaamiseen käytettävän bittimäärän optimaaliseen tulokseen pääsemiseksi [Shannon 1948]. Todennäköisyyden kasvaessa entropia pienenee, joten todennäköisillä symboleilla tulisi olla lyhemmät koodisanat kuin epätodennäköisillä. Mallin kokonaisentropia saadaan laskemalla entropioiden odotusarvo

$$H = -\sum_{x=1}^{n} p(x) \cdot \log_2 p(x) \tag{2}$$

missä n on kaikkien symbolien joukon muodostaman aakkoston koko. Mitä epätasaisempi mallin todennäköisyysjakauma on, sitä pienempi on sen entropia (kuva 2.1).



Kuva 2.1. Binäärisen todennäköisyysmallin entropia valkoisen pikselin todennäköisyyden funktiona.

Mallin muodostamiseksi on useita vaihtoehtoja [Fränti 1999]. Yksinkertaisin on *staattinen malli (static model)*, jossa samaa oletettua mallia käytetään kaikille kuville. Staattisen mallin ongelmana on todellisen syötteen ja mallin eroavaisuudet, jotka heikentävät tiivistystehoa. Esimerkiksi kuvalle 1.1 voitaisiin muodostaa pikselin värin todennäköisyyksille staattinen malli taulukon 2.1 mukaisesti. Kuvan todellinen jakauma on $p(w) = 0.749$ ja $p(b) = 0.251$, joten mallin kokonaisbittimääräksi saadaan $0.749 \times 0.152 + 0.251 \times 3.322 \approx 0.95$ bittiä per pikseli (bpp). Binäärikuvan jokainen pikseli voi saada joko arvon 0 tai 1, joten tiivistämättömän binäärikuvan entropia on 1.00 bpp. *Tiivistyssuhde* on siten 1 / 0.95 ≈ 1.05. Tiivistyssuhde siis kertoo kuinka paljon tiivistämätön kuva vie tilaa verrattuna tiivistettyyn kuvaan.

symboli	p(x)	H(x)
w	0.900	0.152
b	0.100	3.322

Taulukko 2.1. Staattinen todennäköisyysmalli kuvalle 1.1. Mallin kokonaisentropia on 0.95 bpp.

Astetta kehittyneemmässä *semi-adaptiivisessa mallintamisessa (semi-adaptive model)* koodattava kuva käydään läpi kahdesti: Ensimmäisessä vaiheessa kuvasta muodostetaan malli, joka talletetaan koodattavan kuvan lisäksi. Toisessa vaiheessa kuva koodataan muodostetun mallin perusteella. Kuvalle 1.1 muodostettu semi-adaptiivinen malli pikselin värin todennäköisyydelle on taulukon 2.2 mukainen. Tässä tapauksessa kokonaisentropiaksi saadaan 0.81 bpp.

symboli	p(x)	H(x)
w	0.749	0.417
b	0.251	1.994

Taulukko 2.2. Semi-adaptiivinen todennäköisyysmalli kuvalle 1.1. Mallin kokonaisentropia on 0.81 bpp.

Adaptiivisessa mallintamisessa (adaptive model) kuvasta ei kerätä etukäteen mitään tietoa, vaan koodaaja päivittää mallia koodauksen edetessä. Symbolin x todennäköisyys *P(x)* lasketaan kaavalla

$$P(x) = \frac{f(x)}{\sum f} \tag{3}$$

missä *f(x)* on symbolin x esiintymien lukumäärä ja $\sum f$ kaikkien symboleiden esiintymien lukumäärien summa.

Koodauksen alkuvaiheessa käytetty malli voi olla tehoton, mutta se sopeutuu kuvaan, kun jo koodattuja pikseleitä päästään hyödyntämään. Koodattava kuva tarvitsee käydä läpi vain kerran, eikä itse mallia tarvitse tallettaa, koska myös dekoodaaja päivittää mallia koodauksen edetessä. Kuvalle 1.1. adaptiivista mallia käytettäessä kokonaisentropiaksi saadaan sama kuin semi-adaptiivisella mallilla, 0.81 bpp.

2.2 Jaksonpituuskoodaus

Jaksonpituuskoodauksessa käytetään hyväksi tietoa, että pikselin väri riippuu suuresti aiemmin koodatuista pikselin vasemmalla puolella sijaitsevista pikseleistä. Tyypillisesti samanvärisiä pikseleitä on useita peräkkäin, esimerkiksi kuvassa 1.1 oikeassa yläkulmassa sijaitseva vesistö sisältää vaakasuorasti pitkiä mustia pikselijonoja toistuen usealla eri rivillä. Oletetaan, että meillä on binäärikuvassa ensin 170 valkoista (w) pikseliä, sen jälkeen 20 mustaa (b) pikseliä ja lopuksi 110 valkoista pikseliä. Sen sijaan, että koodaisimme nämä 300 pikseliä yksittäin, voimme koodata jokaisen jakson pituuden, eli jonon (170w, 20b, 110w). Menetelmää, jossa koodataan jaksojen pituudet yksittäisten arvojen sijaan kutsutaan *jaksonpituuskoodaukseksi* (run length encoding) [Salomon 2000].

Jaksonpituuskoodauksessa kuva käydään läpi riveittäin vasemmalta oikealle ja ylhäältä alas. Ideana on siis etsiä samanväristen pikseleiden muodostamia peräkkäisiä jonoja, jaksoja. Jokaisesta jaksosta koodataan väri-informaatio sekä jakson pituus jolloin muodostuu seuraavanlainen koodivirta

$C_1, n_1, C_2, n_2, C_3, n_3, \ldots$

missä C_i on i:nnen jakson väri ja n_i on koodi jakson i pituudelle. Binäärikuvissa on vain kaksi väriä, joten jokaista valkoista jaksoa seuraa musta jakso ja päinvastoin. Siksi on riittävää koodata pelkästään jaksojen pituudet, väri-informaatiota ei tarvita. Jokaisen rivin

ensimmäisen jakson oletetaan olevan valkoinen. Jos ensimmäinen pikseli on musta, koodataan rivin alkuun nollan mittainen valkoinen jakso.

Jakson pituus vaikuttaa sen esiintymisen todennäköisyyteen. Keskimäärin esimerkiksi sadan pikselin mittainen musta jakso on epätodennäköisempi kuin kahden pikselin mittainen valkoinen jakso. Tämän vuoksi jaksojen pituudet koodataan vaihtuvamittaisella koodilla. Tiivistyksen saavuttamiseksi yleisimpien jaksonpituuksien koodien tulisi olla mahdollisimman lyhyitä. Koodauksessa voidaan käyttää esimerkiksi luvussa 2.5 esiteltävää Huffman-koodausta [Huffman 1952], jolloin koodit saadaan taulukosta 2.4. Koska erilaisten jaksonpituuksien määrä on äärettömän suuri, ei ole kannattavaa tehdä hyvin suurta koodikirjaa. Esimerkiksi G4-menetelmän kooditaulukko sisältää kahdenlaisia koodisanoja, *päättäviä* (terminating) sekä *kokoavia* (make-up), taulukot 2.3 ja 2.4 [ITU-T 1988]. Jaksonpituudet välillä 0..63 koodataan yhdellä päättävällä koodisanalla ja pituudet välillä 64..1728 koodataan yhdellä tai useammalla kokoavalla koodisanalla, jota seuraa päättävä koodisana. Kokoava koodisana edustaa jaksonpituutta 64 x M (missä M on välillä 1..27), jpka on pienempi tai yhtä suuri kuin koodattavan jakson pituus. Seuraava päättävä koodisana määrittelee eron kokoavan koodisanan ja koodattavan jakson pituuden välillä:

$$r_i = 64 \times m + t \tag{4}$$

missä t=0..63 on päättävä koodisana ja m=1..27 on kokoava koodisana. Jos r_i < 63, on riittävää käyttää pelkästään päättävää koodisanaa.

Koodataan aiemmin mainittu esimerkki 170 valkoista, 20 mustaa ja 110 valkoista pikseliä taulukoiden 2.3 ja 2.4 Huffman-koodauksella. Ensin koodataan 128:aa valkoista vastaava kokoava koodisana 10010 ja sen jälkeen päättävä koodisana 00101011, joka vastaa erotusta 170-128=42. 20:n mustan jakson koodi on 00001101000 ja 110:n valkoisen koodi saadaan ottamalla ensin 64:n valkoisen koodi 11011 ja lopuksi 00000101 vastaamaan valkoisen jakson pituutta 110-64=46. Lopullinen koodi on siis 10010 00101011

00001101000 11011 00000101 ja koodin pituus täten 5+8+11+5+8=37 bittiä, mikä on huomattavasti vähemmän kuin alkuperäisen esimerkin viemä 170+20+110=300 bittiä.

n	valkoinen	musta jakso	n	valkoinen	musta jakso
0	00110101	0000110111	32	00011011	000001101010
1	000111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010110	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	10011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	00001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	0000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011
28	0011000	000011001100	60	01001011	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111

Taulukko 2.3. Huffman-kooditaulukko jaksonpituuksille 0-63 (päättävät koodisanat) [ITU-T 1988].

n	valkoinen	musta jakso	n	valkoinen	musta jakso
64	11011	0000001111	960	011010100	0000001110011
128	10010	000011001000	1024	011010101	0000001110100
192	010111	000011001001	1088	011010110	0000001110101
256	0110111	000001011011	1152	011010111	0000001110110
320	00110110	000000110011	1216	011011000	0000001110111
384	00110111	000000110100	1280	011011001	0000001010010
448	01100100	000000110101	1344	011011010	0000001010011
512	01100101	0000001101100	1408	011011011	0000001010100
576	01101000	0000001101101	1472	010011000	0000001010101
640	01100111	0000001001010	1536	010011001	0000001011010
704	011001100	0000001001011	1600	010011010	0000001011011
768	011001101	0000001001100	1664	011000	0000001100100
832	011001101	0000001001101	1728	010011011	0000001100101
896	011010011	0000001110010	EOL		000000000001

Taulukko 2.4. Huffman-kooditaulukko jaksonpituuksille 64-1728 (kokoavat koodisanat). EOL-symboli tarkoittaa rivinvaihtoa [ITU-T 1988].

2.3 G4

ITU-T:n alun perin fakseihin kehittämä Group 4 (G4) [ITU-T 1988], käyttää tiivistykseen *READ-koodia* (Relative Element Address Designate) [Yasuda 1980]. READ-koodi on yhdistelmä ennustemallista sekä kaksiulotteista jaksonpituuskoodausta. Jaksojen pituuksien sijasta READ koodaa jaksojen vaihtumiskohdat suhteessa yläpuolella olevan rivin jaksojen vaihtumiskohtiin. READ-koodissa on kolme erilaista koodaamismoodia: *Pystysuora*, *vaakasuora* sekä *ohitus*.

Pystysuorassa moodissa jakson vaihtumiskohta koodataan suhteessa läheiseen vaihtumiskohtaan ylempänä olevassa rivissä. Vaihtumiskohta on läheinen, mikäli se se sijaitsee kolmen pikselin etäisyydellä. Siis pystysuorassa moodissa on seitsemän arvoa: -3, -2, -1, 0, +1, +2, +3. Jos läheistä vaihtumiskohtaa ei löydy, siirrytään vaakasuoraan moodiin. Siinä jakson pituus koodataan yksiulotteisella jaksonpituuskoodauksella. Kolmas moodi on ohitus, jota käytetään ilmaisemaan, ettei yläpuolella olevalta riviltä löydy

vastaavaa jakson vaihtumiskohtaa. Ohituskoodi siis ilmoittaa vastaanottajalle, että seuraava kokonainen jakso ylemmällä rivillä on ohitettava. Taulukossa 2.5 on esitelty READ-koodin kooditaulu.

Moodi:	**Koodisana:**
Ohitus	0001
Vaakasuora	001+H_w(wl)+H_b(bl)
Pystysuora +3	0000011
Pystysuora +2	000011
Pystysuora +1	011
Pystysuora 0	1
Pystysuora –1	010
Pystysuora –2	000010
Pystysuora -3	0000010

Taulukko 2.5. Kooditaulu READ-koodille. H_w ja H_b viittaavat Huffman-koodeihin taulukoissa 2.3 ja 2.4.

READ-koodin ymmärtämiseen tarvitaan seuraavanlaisia määritelmiä liittyen pikselijonojen värien vaihtumiskohtiin (kuva 5.1):

a_0: viimeisin pikseli jonka arvon sekä koodaaja että dekoodaaja tietävät. Rivin alussa tämä asetetaan kuvitteelliseen valkoiseen pikseliin rivin ensimmäisen pikselin vasemmalla puolella.

a_1: ensimmäinen jakson vaihtumiskohta a_0:sta oikealle. Tämä pikselin on erivärinen kuin pikseli a_0.

a_2: toinen jakson vaihtumiskohta a_0:sta oikealle. Tämän pikselin väri on sama kuin a_0 pikselin väri ja erisuuri kuin a_1.

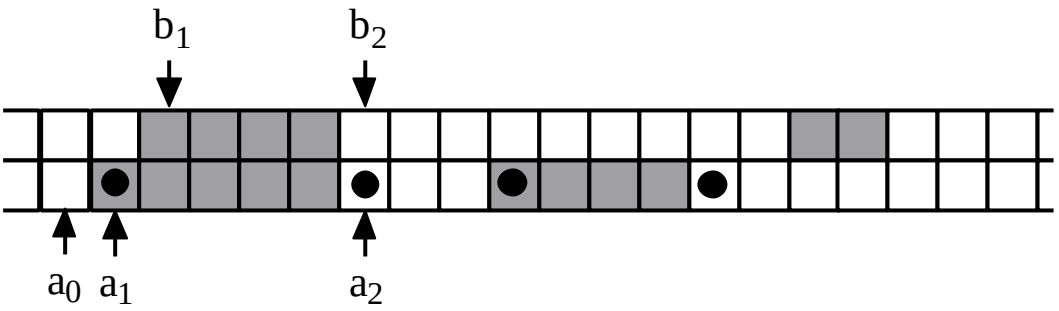
b_1: ensimmäinen jakson vaihtumiskohta koodattavan rivin yläpuoleisella rivillä, joka on eri värinen kuin a_0.

b_2: ensimmäinen jakson vaihtumiskohta b_1:stä oikealle. Tämän pikselin väri on sama kuin a_0:n väri.

Itse koodaus tapahtuu seuraavasti: Jos b_2 on a1:n vasemmalla puolella, käytetään ohitus-moodia. Tällöin koodataan bittijono 0001. Ohitus-koodin saatuaan dekoodaaja tietää, että kaikki pikselit a_0:sta b_1:n alapuolella olevaan pikseliin saakka ovat saman värisiä. Tämän jälkeen b_0:n alapuoleinen pikseli sijoitetaan a_0:aan ja $a_1=a_2$. Etsitään pikselit b_1 ja b_2.

Jos a_1 sijaitsee ennen b_1:ta x-suunnassa, lasketaan niiden välinen etäisyys. Mikäli etäisyys on ±3 tai vähemmän, koodataan a_1:n sijainti suhteessa b_1:een. Tämän jälkeen sijoitetaan $a_0=a_1$ ja jatketaan koodausta. Moodia kutsutaan nimellä pystysuora-moodi. Jos etäisyys on suurempi kuin kolme, käytetään vaakasuoraa moodia. Vaakasuorassa moodissa koodataan Huffman-koodit jaksoille a_0:sta a_1:een ja a_1:sta a_2:een.

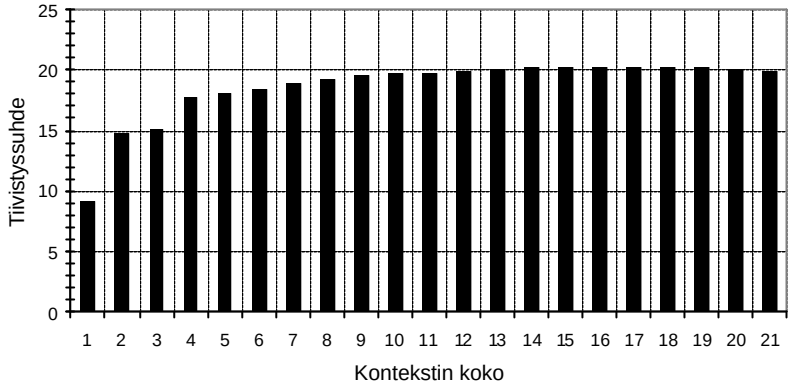
Koodataan esimerkiksi kuvan 2.2 alarivi READ-koodilla. Koska etäisyys $| a_1 - b_1| = -1$, käytetään pystysuoraa moodia ja koodiksi saadaan taulukosta 2.5 bittijono 010. Seuraavaksi mustat jaksot loppuvat samalla kohtaa, jolloin käytetään jälleen pystysuoraa moodia, jolle saadaan koodiksi 1. Koodattavalla rivillä on seuraavaksi viiden pikselin musta jakso, jolle ei löydy vastinetta yläpuoleiselta riviltä. Käytetään vaakasuoraa moodia ja koodataan kolme valkoista ja neljä mustaa pikseliä, jolloin saadaan koodeiksi 001 ja taulukosta 2.3 1000 sekä 011. Seuraavaksi käytetään ohitus-moodia, koska ylärivin kahden mustan jaksolle ei löydy vastinetta alariviltä. Koodiksi tulee siten 0001. Tällä tavoin koodattiin 19 pikseliä ja koodiksi saatiin 010100110000110001, koodinpituuden ollessa 18 bittiä. Käytännössä jaksojen pituudet ovat huomattavasti tätä esimerkkiä pidempiä, jolloin saadaan aikaan parempi tiivistysteho.



Kuva 2.2. Esimerkki READ-koodista.

2.4 Kontekstimalli

Kontekstimallissa (context model) hyödynnetään pikseleiden keskinäisiä riippuvuuksia [Langdon ja Rissanen 1981]. Naapuripikseleitä tutkitaan laajemmalta alueelta kuin jaksonpituuskoodauksessa. Esimerkiksi jos kuvan jonkin pikselin kaikki naapuripikselit ovat mustia, myös koodattava pikseli on hyvin suurella todennäköisyydellä musta. Todennäköisyysjakauma siis riippuu suuresti siitä, missä *kontekstissa* (context) se esiintyy. Siksi jokaiselle kontekstille käytetään erilaista mallia. Kuva, josta malli muodostetaan, käydään läpi rivi kerrallaan vasemmalta oikealle ja jokaisen pikselin naapuristosta kerätään tietoa. Kontekstien lukumääräksi tulee kaikki mahdolliset naapuripikseleiden kombinaatiot, jolloin esimerkiksi käytettäessä binäärikuvassa kymmenen pikselin naapuristoa saadaan 2^{10} = 1024 erilaista kontekstia. Kontekstimalli toimii erityisen hyvin binäärikuville kontekstien suhteellisen vähäisen lukumäärän vuoksi. Kontekstien lukumäärän kasvaessa myös mallin sopeutuminen kuvaan hidastuu, joten kontekstin koon pitäisi olla suhteessa kuvan kokoon [Moffat 1991]. Kuvassa 2.3 on esitetty tiivistyssuhde kontekstin koon funktiona joukolle testikuvia käytettäessä aritmeettista koodausta.



Kuva 2.3. Tiivistyssuhde kontekstin koon funktiona [Ageenko 2000].

Kuvassa 2.4 on muodostettu kahden naapuripikselin kontekstimalli kuvalle 1.1. Kyseistä mallia käytettäessä kuvan kokonaisentropiaksi saadaan kaavan 2 mukaisesti 0.26 bpp. Kuva saadaan siis tiivistettyä lähes neljäsosaan alkuperäisestä näinkin yksinkertaisella mallilla.

konteksti	pikseli	f(x)	p(x)	H(x)
		167681	0.985	0.021
		2483	0.015	6.099
		8222	0.472	1.083
		9191	0.528	0.922
		9127	0.532	0.911
		8034	0.468	1.095
		2236	0.049	4.339
		43026	0.951	0.073

Kuva 2.4. Kahden pikselin kontekstimalli kuvalle 1.1. Mallin kokonaisentropia on 0.26 bpp.

Periaatteessa epätasainen jakauma voidaan saavuttaa suurentamalla kontekstin kokoa. Tällöin malli vie kuitenkin entistä enemmän tilaa (semi-adaptiivinen mallinnus) tai mallin sopeutuminen kuvaan hidastuu (adaptiivinen mallinnus).

2.5 Sanakirjamenetelmät

Aiemmissa esitellyissä menetelmissä on käytetty tilastollista mallia yksittäisten symbolien koodaamiseen. Tiivistystä tapahtuu, kun symbolit koodataan alkuperäistä symbolia lyhemmiksi bittijonoiksi. Sanakirjaan perustuvat menetelmät sen sijaan koodaavat useita symboleita yhdeksi sanakirjasymboliksi [Salomon 2000].

Sanakirjamenetelmissä on ajatuksena rakentaa lista, eli sanakirja usein esiintyvistä symbolijonoista ja koodata näiden jonojen esiintymät viittauksilla listan indekseihin. Sanakirjamenetelmät toimivat parhaiten tilanteissa, joissa on suhteellisen pieni määrä erilaisia usein toistuvia symbolijonoja.

Staattinen sanakirja on samanlainen kaikille syötteille. Koska sekä koodaaja että dekoodaaja tietävät sen ennalta, sitä ei tarvitse tallentaa tiivistetyn tiedoston mukana. Staattinen sanakirja on käyttökelpoinen jos syötteestä on olemassa etukäteistietoa. Esimerkiksi tiivistettäessä Pascal-kielistä ohjelmakoodia merkkijonot ’if’, ’then’ ja ’else’ esiintyvät usein, joten ne on kannattavaa laittaa sanakirjaan.

Semi-adaptiivinen sanakirja käy syötteen läpi kahdesti: ensimmäisellä läpikäynnillä syöte analysoidaan ja sen perusteella muodostetaan alkusanakirja, jota käytetään toisessa läpikäynnissä tiivistyksen tehostamiseksi [Akimov 2001]. Tällöin myös alkusanakirja joudutaan tallettamaan tiivistettyyn tiedostoon.

Yleisimmin käytetty adaptiivinen sanakirja on koodauksen alussa täysin tyhjä. Koodauksen edetessä symboleja lisätään sanakirjaan. Tällä tavoin sanakirja sopeutuu syötteeseen, eikä

sitä tarvitse välittää dekoodaajalle, koska dekoodaaja osaa rakentaa sanakirjan dekoodauksen aikana.

2.6 GIF

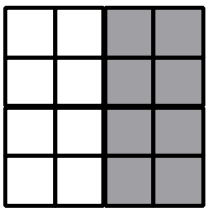
Compuserven kehittävä GIF-formaatti [Miano 1999] käyttää adaptiiviseen sanakirjaan pohajutuvaa LZW-algoritmia tiivistykseen [Welch 1984]. LZW-algoritmi on muunnos tunnetusta LZ78-algoritmista [Ziv ja Lempel 1978]. GIF-formaatti tukee väripalettikuvia, jotka sisältävät yleensä maksimissaan 256 väriä.

Koodauksen ja dekoodauksen aluksi tyhjään sanakirjaan lisätään kaikki mahdolliset yhden mittaiset symbolit, eli binäärikuvan tapauksessa yhden mittainen musta ja yhden mittainen valkoinen pikselijono. Alussa jokainen sanakirjaindeksi esitetään kolmella bitillä ja sanakirjan koon kasvaessa yli kahdeksan symbolin indeksin bittimäärää kasvatetaan yhdellä. Näin jatketaan kunnes sanakirjan maksimikoko 4096 saavutetaan, jonka jälkeen algoritmi toimii kuten staattinen sanakirja. Tällöin sanakirja ei enää sopeudu, vaikka kuvan tilastollinen malli muuttuisi täysin [Akimov 2001]. Toinen vaihtoehto olisi aloittaa uuden tyhjän sanakirjan täyttäminen. Kuvassa 2.5 on esitetty LZW-koodausalgoritmin toiminta.

```
p:=∅;
sanakirja:={w ∪ b};
for x:=1 to Y do
  for x:=1 to X do
    c:=Pikseli(x,y);
    if p+c ∈ sanakirja then
      p:=p+c;
    else
      Tulosta(p);
      sanakirja:=sanakirja ∪ p+c;
      p:=c;
    endif
  endfor
endfor
```

Kuva 2.5. LZW-koodausalgoritmi [Welch 1984].

Koodataan esimerkiksi kuvan 2.6 mukainen 4×4 binäärikuva. Ensimmäinen luettava symboli on muutujaan c sijoitettava valkoinen (w), joka on jo sanakirjassa. Seuraavaksi luetaan jälleen valkoinen symboli, jonka jälkeen luettujen symboleiden yhdistelmä (w,w) ei ole vielä sanakirjassa. Näin ollen koodataan w ja lisätään jono (w,w) sanakirjaan. Symboli w sijoitetaan muuttujaan p. Seuraavaksi luetaan b, ja jälleen yhdistelmä (w,b) puuttuu sanakirjasta. Koodataan w ja lisätään jono (w,b) sanakirjaan. Jatkettaessa koodausta samalla tavalla saadaan lopulliseksi sanakirjaksi taulukon 2.6 mukainen ja tulosteeksi jono (1,1,2,2,3,5,7,6,4,2)



Kuva 2.6. 4×4 binäärikuva.

indeksi	symboli
1	w
2	b
3	w w
4	w b
5	b b
6	b w
7	w w b
8	b b w
9	w w b b
10	b w w
11	w b b

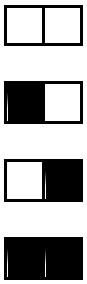
Taulukko 2.6. Lopullinen sanakirja koodattaessa kuva 4.2 LZW-algoritmilla.

Dekoodauksessa dekoodaaja rakentaa sanakirjaa samalla tavoin kuin koodauksessa. Näin itse sanakirjaa ei tarvitse välittää.

2.7 Koodaus

Huffman-koodaus [Huffman 1952] perustuu ajatukseen, että mitä useammin symboli esiintyy, sitä lyhempi on sitä kuvaava koodisana. Koodauksen aluksi kaikista symboleista yhdistetään kaksi pienimmän todennäköisyyden omaavaa uudeksi symboliksi, jonka todennäköisyydeksi tulee yhdistettävien symbolien todennäköisyyksien summa. Seuraavaksi yhdistetään jälleen kaksi pienimmän todennäköisyyden sisältävää symbolia. Näin jatketaan kunnes jäljellä on enää yksi symboli, jonka todennäköisyys on 1.

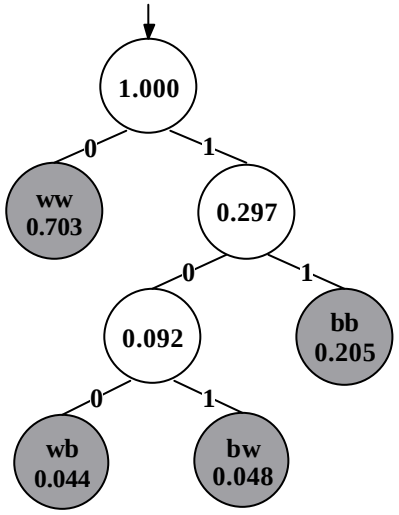
Esimerkiksi kuva 1.1 voitaisiin koodata kahden pikselin jaksoissa Huffman-koodauksella jolloin saataisiin kuvan 2.7 mukainen malli.



lukumäärä	p(x)	H(x)
87870	0.703	0.508
5955	0.048	4.392
5571	0.044	4.488
25604	0.205	2.287

Kuva 2.7. Todennäköisyysmalli kuvalle 1.1. koodattaessa se kahden pikselin jaksoissa. Mallin kokonaisentropia on 0.77 bpp

Saadut todennäköisyydet voidaan nyt koodata Huffman-puuksi kuvan 2.8 mukaisesti. Aluksi yhdistetään kaksi pienintä todennäköisyyttä 0.044 ja 0.048, joista muodostetaan puuhun uusi solmu sisältäen todennäköisyyden 0.044+0.048=0.092. Tämän jälkeen yhdistetään jälleen kaksi pienintä todennäköisyyttä, eli äsken lisätty 0.092 ja 0.205 yhdistetään uudeksi solmuksi. Solmujen yhdistämistä jatketaan samalla tavalla, kunnes puuhun saadaan muodostettua juurisolmu, jonka todennäköisyys on 1. Koodattujen symbolien Huffman-koodit saadaan kulkemalla puun juuresta halutun symbolin sisältämään lehtisolmuun.



Kuva 2.8. Kuvan 2.7 mallista muodostettu Huffman-puu.

symboli	p(x)	koodisana	koodin pituus
ww	0.703	0	1
bb	0.205	11	2
wb	0.044	100	3
bw	0.048	101	3

Taulukko 2.7. Kuvan 2.8 Huffman-puusta saadut koodit.

Koodattaessa kuva 1.1 kahden pikselin jaksoina taulukon 2.7 mukaisilla koodeilla saadaan tiivistetyn kuvan tilantarpeeksi 0.77 bittiä per pikseli.

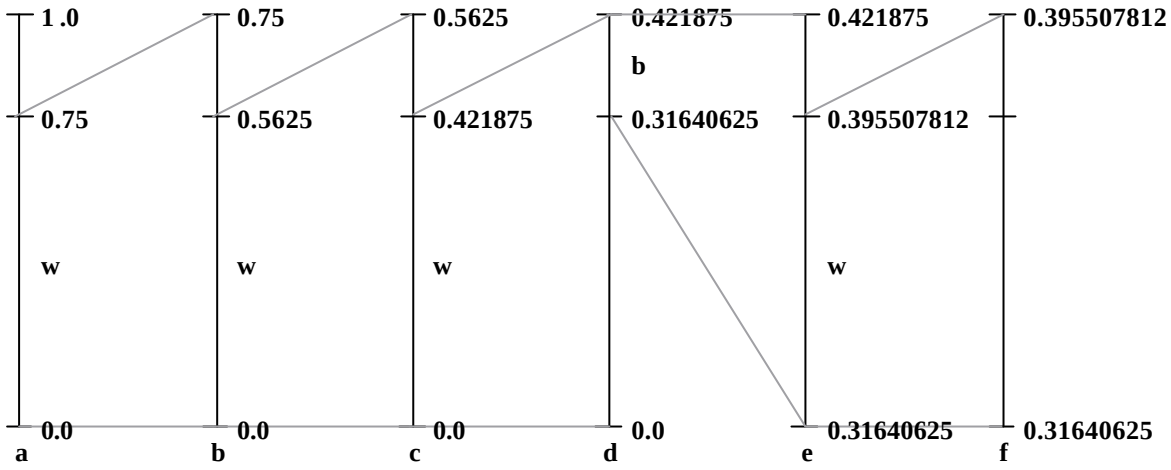
Huffman-koodauksen heikkoutena on, että jokaisen symbolin koodin pituus on vähintään yksi. Esimerkiksi muodostettaessa Huffman-puu kahdesta symbolista joiden todennäköisyydet ovat 0.99 ja 0.01 molempien symbolien koodin pituus on yksi. Tällaisissa tapauksissa Huffman-koodaus ei voi saavuttaa tiivistystä.

Koodaus voidaan kuitenkin suorittaa optimaalisesti *aritmeettisella koodaajalla (arithmetic coder)* [Rissanen ja Langdon 1979], jota sekä JBIG [JBIG 1993] että JBIG2 [JBIG 1999] käyttävät. Aritmeettisen koodauksen perusidea on esittää koko koodattava syöte yhtenä pienenä reaalilukuvälinä välillä [0,1[, kun Huffman-koodauksessa jokaiselle symbolille on oma koodinsa. Koodaus aloitetaan jakamalla väli [0,1[kahteen aliväliin mallin antamien mustan ja valkoisen pikselin todennäköisyyksien mukaan. Seuraavaksi valitaan koodattavaa pikseliä vastaava aliväli, joka jaetaan edelleen kahteen osaan mallin perusteella. Prosessia toistetaan jokaiselle koodattavalle pikselille, jolloin väli pienenee kunnes lopullinen väli määrittää koodattavan kuvan yksikäsitteisesti (kuva 2.9).

```
alaraja:=0;
yläraja:=0;
for x:=1 to Y do
  for x:=1 to X do
    c:=Pikseli(x,y);
    pituus:=yläraja-alaraja;
    yläraja:=alaraja×pituus×yläraja(c);
    alaraja:=alaraja×pituus×alaraja(c);
  endfor
endfor
tulosta(alaraja);
```

Kuva 2.9. Aritmeettinen koodaus [Rissanen ja Langdon 1979].

Koodataan esimerkiksi jono (w,w,w,b,w) mustia ja valkoisia pikseleitä aritmeettisella koodaajalla (kuva 2.10). Malliksi oletetaan että *p(w)*=0.75 ja *p(b)*=0.25. Mallia voitaisiin myös päivittää jokaisen koodatun pikselin jälkeen, jolloin kyseessä olisi adaptiivinen malli. Ensimmäinen pikseli on valkoinen, joten valitaan alempi aliväli ja päivitetään välin rajaa sen mukaisesti: 0.0+0.75×075=0.5625. Samalla tavoin koodataan myös toinen ja kolmas valkoinen pikseli. Neljäs pikseli on musta, joten valitaan silloin ylempi aliväli. Viimeisen pikselin kohdalla valitaan jälleen alempi aliväli. Näin kaikki pikselit on saatu koodattua ja lopullinen alaraja on 0.31640625. Muutettaessa desimaaliosa binääriluvuksi saamme arvon 11110 00101 10011 00001 10001. Syötteen pituus on siten koodattuna 25 bittiä.



Kuva 2.10. Aritmeettinen koodaus koodattaessa jono (w,w,w,b,w).

Dekoodauksessa käytetään lopullista low-arvoa, joka tässä esimerkissä siis on 0.31640625. Ensimmäinen pikseli saadaan selville sijoittamalla arvo 0.31640625 kuvan 2.10 pylvään a asteikkoon. Koska $0 \leq 0.31640625 < 0.75$, ensimmäinen symboli on w. Tämän jälkeen arvosta 0.31640625 vähennetään symbolin w alkuperäinen low-arvo, jolloin saadaan $0.31640625 - 0 = 0.31640625$. Jaettaessa näin saatu arvo vielä w:n alkuperäisellä todennäköisyydellä 0.75 saadaan tulokseksi 0.421875. Toinen pikseli saadaan selville sijoittamalla tämä arvo kuva 2.10 pylvään b asteikkoon, vähentämällä symbolin low-arvo ja jakamalla alkuperäisellä todennäköisyydellä. Samaa menetelmää toistetaan, kunnes koko syöte on saatu dekoodattua.

3 Digitaaliset karttakuvat

Nykyisin paperimuodossa olevia karttoja on saatavana ympäri maailman ja digitaalisessa muodossa olevien karttojen lukumäärä kasvaa koko ajan. Lisäksi paperikarttojen digitointi on suoraviivainen tehtävä. Tässä tutkielmassa testikuvina käytetyt Maanmittauslaitoksen [Maanmittauslaitos 2002] peruskartat mittakaavassa 1:20 000 ovat kooltaan 5000×5000 pikseliä, eli yksi karttakuva esittää 10 km $\times$ 10 km aluetta. Tyypillisesti karttakuva koostuu useasta erillisestä semanttisesta komponentista, kuten peruskartta, pellot, vesistöt ja korkeuskäyrät. Komponenttien riippumattomuus mahdollistaa käyttäjälle erilaisia näkymiä, jolloin vain halutut komponentit voidaan esittää.

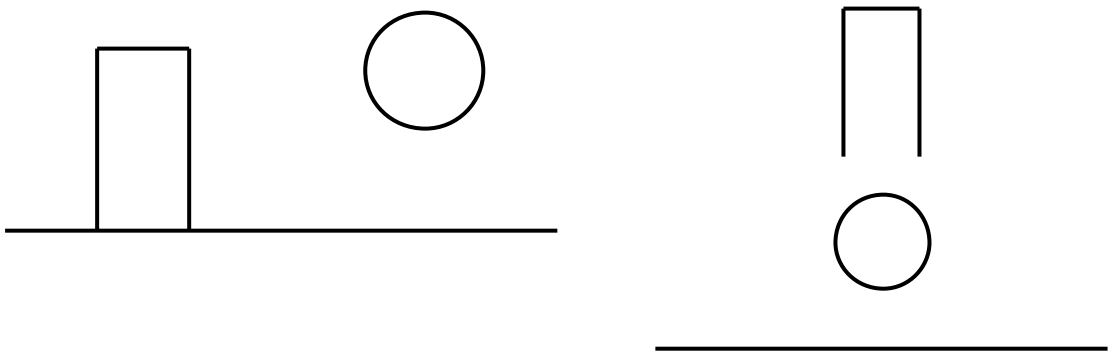
Digitaalisten karttakuvien käytössä suurin ongelma on valtava tilantarve. Yksi 10 km^2 binäärinen karttakuvan kerros vie rasterimuodossa tilaa 3 megatavua, ja neljästä binäärikomponentista muodostuva kokonainen kartta vaatii tilaa jo 12 megatavua. Tämä on yleensä liikaa etenkin mobiililaitteissa jotka tyypillisesti sisältävät 16–32 megatavua muistia [Gellersen ja Thomas 2000]. Ratkaisu muistitilavaatimuksiin on tiivistys. Tehokkaalla tiivistysmenetelmällä päästään tiivistyssuhteeseen 20:1 [Haskell et al. 1998], jolloin 12 megatavun kartta mahtuu 600 kilotavuun. Tiivistetyn kartan dekoodauksen on tapahduttava reaaliajassa päätelaitteessa, jonka laskentateho voi olla hyvin rajallinen [Gellersen ja Thomas 2000]. Karttojen tiivistäminen sen sijaan tapahtuu tehokkaalla palvelimella etukäteen, eikä siihen kuluvalla ajalla siten ole yhtä suurta merkitystä.

3.1 Digitaalisten karttojen esitysmuodot

Digitaalisten karttojen tallennusformaatit voidaan jakaa kahteen erilliseen luokkaan: Rasteriformaatteihin sekä vektoriformaatteihin. Rasterikartat tallentavat kuvan pikseleistä muodostettuna matriisina. Matriisin yksittäisessä solussa oleva arvo määrittää pikselin tyypin ja värikuvassa lisäksi värin, esimerkiksi vesistöä esittävässä binäärikomponentissa

arvo 1 solussa tarkoittaa, että kyseisessä kohdassa karttaa on vesistöä. Karttakuvien hyvin vähäisestä värimäärästä johtuen värikartat ovat yleensä palettikuvia: Matriisin solussa oleva arvo määrittää paletin indeksin, jossa on määritelty indeksin perusväreistä punainen, vihreä ja sininen koostuva RGB-arvo [Foley et al. 1996]. Tällöin myös paletti joudutaan tallettamaan kuvan mukana. Toinen vaihtoehto on tallentaa jokainen komponentti erikseen binäärikuvana.

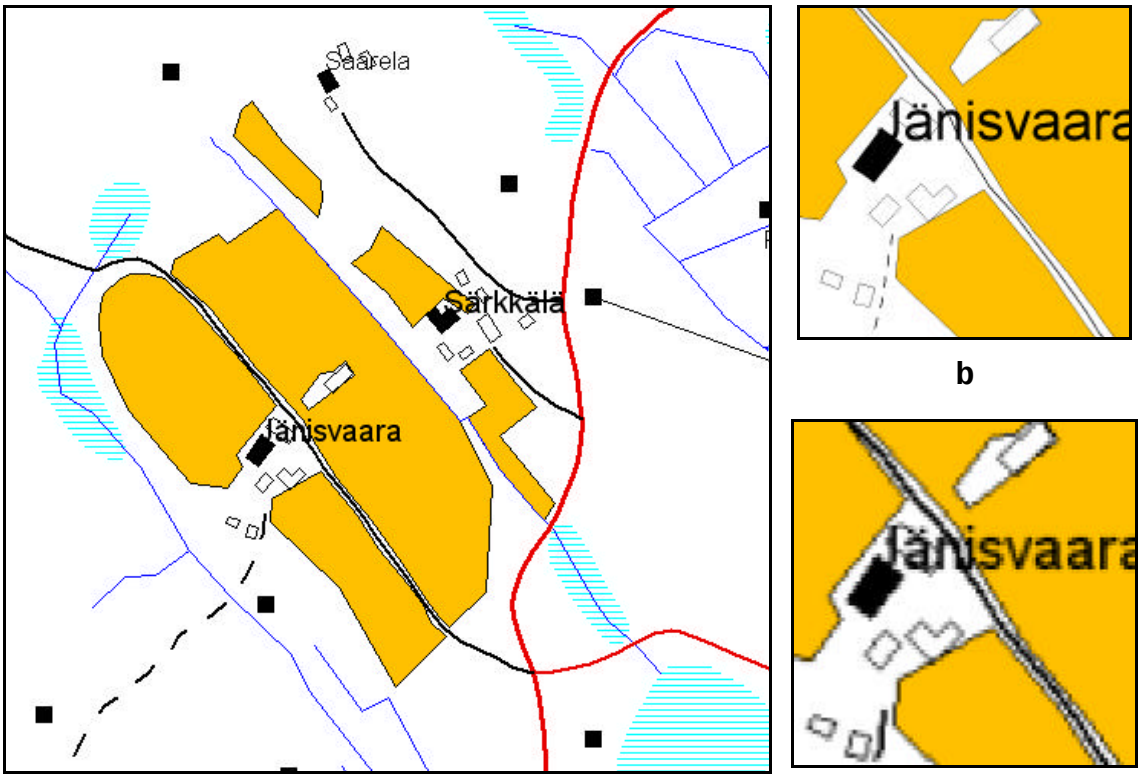
Vektoriformaateissa kartta muodostuu primitiiveistä [Foley et al. 1996], joita ovat mm. piste, viiva, moniviiva ja ympyrä (kuva 3.1). Kaikki primitiivit voidaan määritellä yksittäisten kaksiulotteisten vektoreiden avulla. Kartassa esimerkiksi tie voidaan esittää yhtenä moniviivana.



Kuva 3.1. Vektorikuva (vasemmalla) sekä sen muodostavat objektit (oikealla) ylhäältä alas: moniviiva, ympyrä ja viiva.

Karttakuvissa vektorin komponentit kertovat vektorin sijainnin pituus- ja leveysasteissa, eli koordinaatit. Koska jokainen maantieteellinen kohde muodostuu primitiiveistä, voidaan siihen yhdistää helposti ominaisuustietoa, kuten kohteen tyyppi. Täten eri semanttiset kerrokset (komponentit) voidaan helposti poimia erikseen ja voidaan esimerkiksi laskea lyhin reitti tietä pitkin kahden paikan välillä. Vektorikuvan esittäminen näytöllä vaatii laskentatehoa, koska kuva joudutaan ennen esittämistä muuntamaan näyttölaitteen ymmärtämään rasterimuotoon eli *rasteroimaan* (rasterize) [Nevalainen 2001].

Vektorimuoto sopii hyvin portaattomaan skaalaukseen, koska kuva voidaan rasteroida halutulla tarkkuudella. Rasterikuva puolestaan on sidottu resoluutioon, jolla se on talletettu. Muutettaessa rasterikuvan resoluutiota joudutaan joko lisäämän tai poistamaan pikseleitä, minkä seurauksena kuvan laatu väistämättä heikkenee. Käytännössä samasta alueesta onkin oltava useita rasterikarttoja eri mittakaavoilla [Fränti el al. 2001]. Kuva 3.2 esittelee skaalauksen eroja rasteri –ja vektorimuodossa.



Kuva 3.2. (a) Alkuperäinen kartta, (b) Vektorikartan suurennos, (c) Rasterikartan suurennos.

Rasterikuvissa objektit esitetään toisiinsa liittyvien kuvapisteiden avulla, joten ominaisuustieto joudutaan tallettamaan erilleen. Kuvapisteisiin sidottu ominaisuustieto joudutaan kuitenkin määrittämään uudelleen, jos resoluutiota myöhemmin muutetaan.

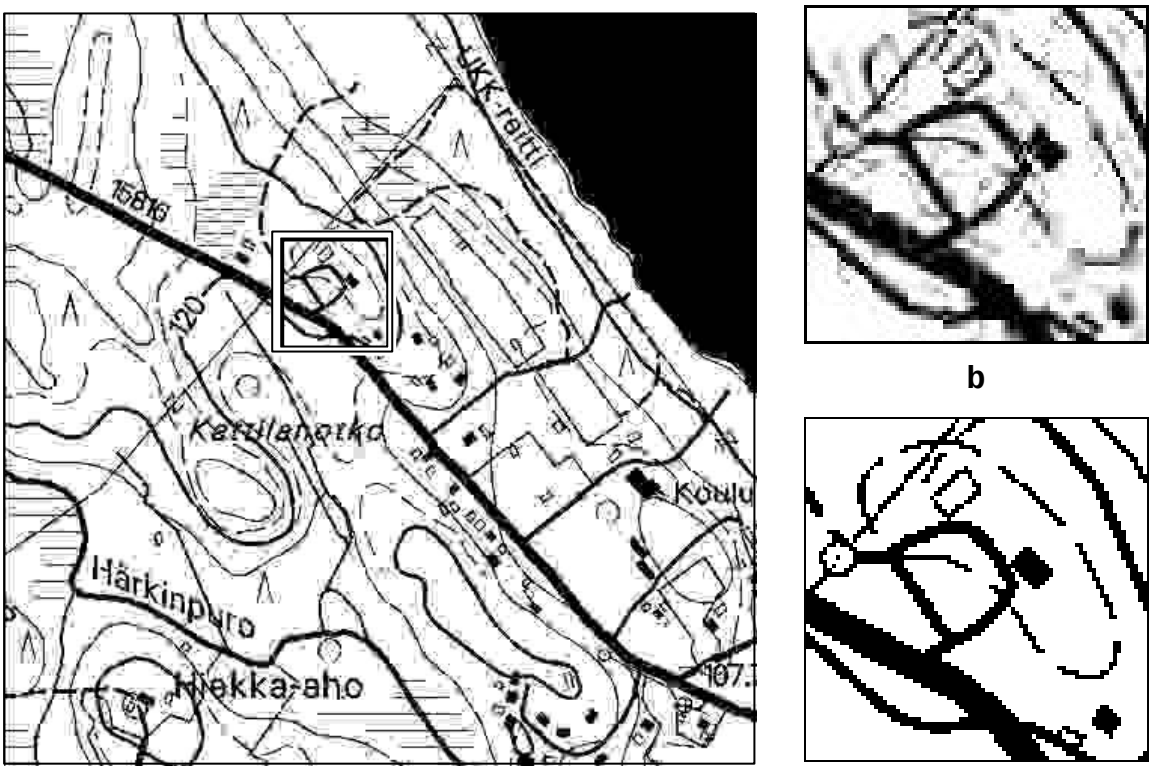
Käytettäessä tehokasta tiivistysmenetelmää rasterikuvien tilantarve voi olla huomattavasti vektorikuvia pienempi [Veis 2001]. Haittapuolena tiivistetty kuva joudutaan purkamaan ennen näytöllä esittämistä. Purkaminen on kuitenkin pystyttävä tekemään reaaliajassa, jotta näytön vieritys olisi sujuvaa. Rasteriformaatti sopii vektoriformaattia paremmin erityisesti epäsäännöllisten kohteiden esittämiseen. Mitä epäsäännöllisempi kohde on, sitä enemmän koordinaattipareja joudutaan käyttämään sen esittämiseen vektoriformaatissa. Rasteriformaatissa kohteen muotoa ei erikseen määritellä, joten esimerkiksi yksivärissä kuvassa on yhtä paljon pikseleitä kuin saman kokoisessa tiestöä esittävässä kuvassa. Sen sijaan kartan sisältämän tiedon määrä kasvaa yksityiskohtien lisääntyessä, jolloin tiivistysteho laskee.

Ongelmana vektorikartoissa on se, ettei niitä ole saatavana kaikkialta [Fränti et al. 2001]. Paperilla olevat karttalehdet on sen sijaan helppo digitoida rasterimuotoon tai muuntaa haluttuun formaattiin jostain toisesta rasteriformaatista. Rasteriformaatista vektoriformaattiin muunto on yleensä vaikea tehtävä. Lisäksi eri vektoriformaattien yhteensopimattomuus on rajoittava tekijä.

3.2 Häviöllinen ja häviötön tiivistys

Kuvantiivistys voidaan jakaa kahteen luokkaan: *häviölliseen* (lossy) ja *häviöttömään* (lossless) tiivistykseen. Häviöttömässä tiivistyksessä purettu kuva on täsmälleen samanlainen kuin alkuperäinen ennen tiivistystä. Sen sijaan häviöllisessä tiivistyksessä pikselin arvot saattavat muuttua alkuperäiseen verrattuna tiivistystehoa parannettaessa. Täysvärikuville suunnitellut häviölliset menetelmät pyrkivät hävittämään tietoa ihmissilmältä mahdollisimman huomaamattomasti, mutta valitettavasti karttakuvissa jokainen pikseli on merkityksellinen, toisin kuin valokuvissa. Kuvassa 3.3 on esimerkkinä tiivistetty karttakuva häviöllisellä JPEG-menetelmällä [Pennebaker ja Mitchell 1993]. Kyseinen JPEG-kuva vie saman verran tilaa kuin häviötön GIF-kuva, 0.52 bpp. JPEG-kuvaa tarkennettaessa näkyy selvästi kuinka monet yksityiskohdat ovat kadonneet ja kuvan

terävyys on heikentynyt oleellisesti. Näin ollen voidaan todeta, etteivät häviölliset tiivistysmenetelmät sovellu karttakuville.



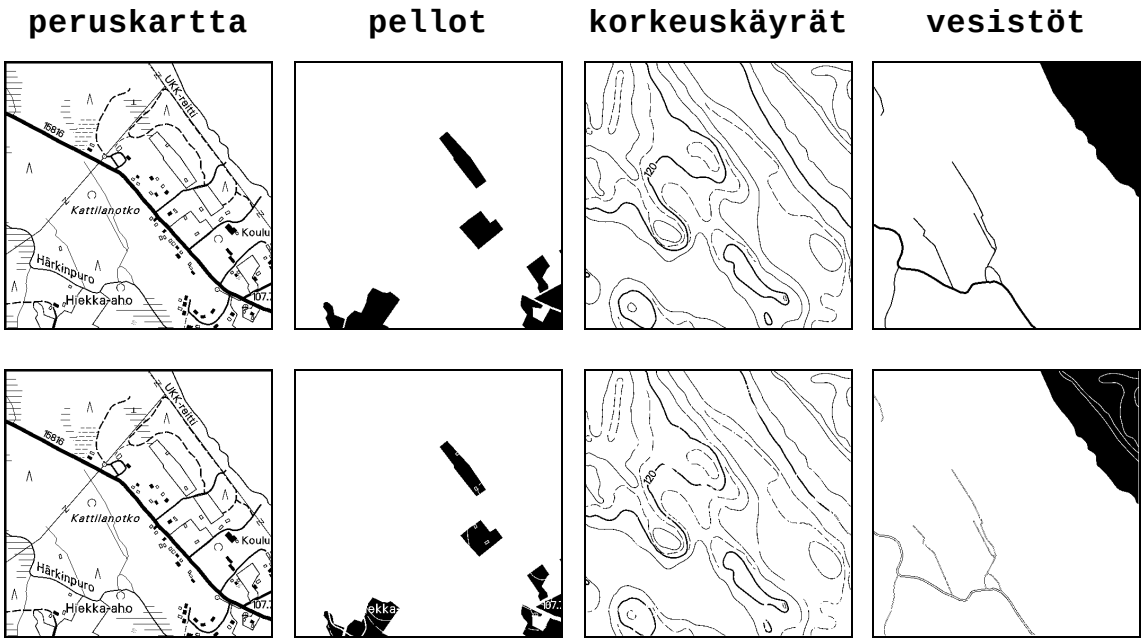
Kuva 3.3. (a) Kuva 1.1 tiivistettynä häviöllisellä JPEG-menetelmällä, (b) suurennus a:sta, (c) sama suurennus häviöttömästä GIF-kuvasta.

3.3 Jako binäärikomponentteihin

Karttakuvien semanttisten komponenttien pienestä lukumäärästä johtuen voidaan jokainen kerros tiivistää tehokkaasti erillisinä binäärikuvina. Erilliset komponentit mahdollistavat myös päällekkäisten kohteiden, kuten pellon poikki menevän tien tallentamisen. Mikäli kartta olisi ainoastaan yhdessä rasterikuvassa, päällekkäisistä kohteista voitaisiin esittää vain yksi, koska yhteen kuvapisteeseen voidaan tallettaa vain yksi arvo. Erillisten

binäärikuvien käyttö mahdollistaa käyttäjälle myös erilaisia näkymiä: vain käyttäjän valitsemat komponentit esitetään näytöllä.

Komponentteihin jako voidaan tehdä kolmella eri tavalla. Semanttinen jako tarkoittaa, että kukin kerros saadaan jo valmiiksi erillisenä. Tämä on mahdollista esim. Maanmittauslaitoksen [2002] kuvissa tai rasteroitaessa karttoja vektoriformaatista. Esimerkki semanttisesta jaosta on kuvassa 3.4 ylärivissä.



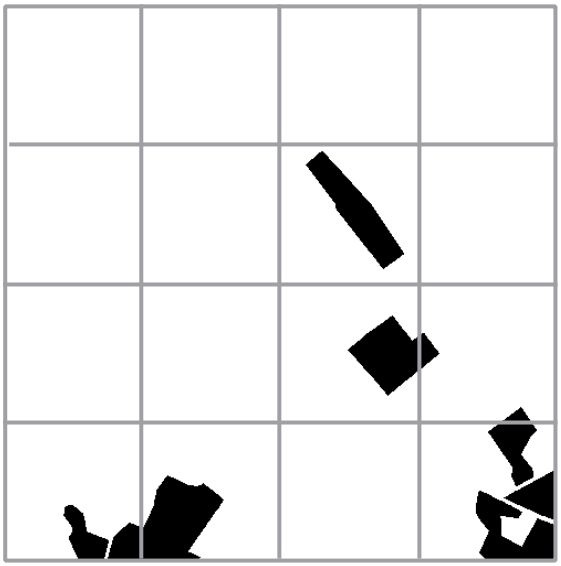
Kuva 3.4. Kuva 1.1 jaettuna komponentteihin semanttisella jaolla (ylärivi) ja värierottelulla (alarivi).

Värierottelussa komponentit erotetaan toisistaan värikuvasta pikselien värin perusteella. Tällöin ei kuitenkaan voida palauttaa kaikkea alkuperäisen semanttisen kerroksen tietoa. Lisäksi kerroksissa on paljon useassa kerroksessa toistuvaa näennäistä tietoa, joka heikentää tiivistystehoa. Esimerkiksi tekstielementit rikkovat järvien ja peltojen yhtenäisyyden lisäten täten tiedon määrää, kuten kuvan 3.4 alarivissä.

Bittitasoerottelua täytyy käyttää, kun alkuperäinen kartta on vain rasterimuodossa ja värien määrä on liian suuri tehokkaan värierottelun toteuttamiseksi. Kuva voi esimerkiksi olla 24-bittinen JPEG-muodossa talletettu digitoitu paperikartta. Tällöin värimäärää vähennetään kvantisoimalla 256:een, minkä jälkeen pikselien arvot jaetaan bittitasoihin *Gray-koodilla* [Weinberger, Rissanen ja Arps 1996].

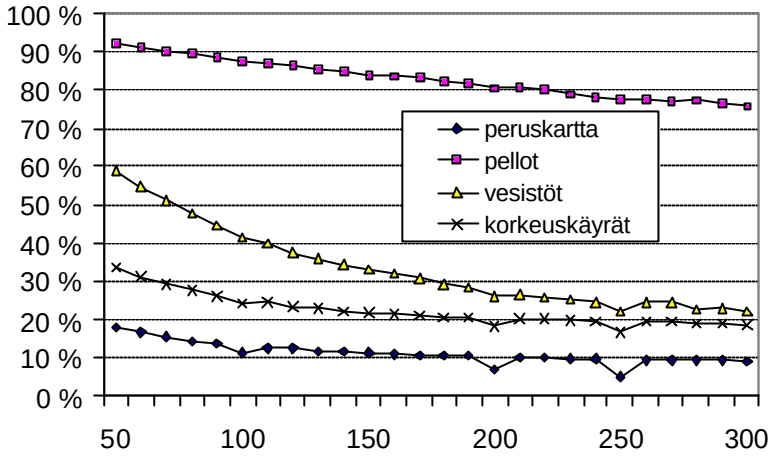
3.4 Jako lohkoihin

Tiivistetty kuva joudutaan yleensä purkamaan kokonaisuudessaan muistiin ennen sen esittämistä näytöllä, vaikka haluaisimme näyttää siitä vain murto-osan. Tämä on usein liian raskasta päätelaitteessa. Ratkaisuna ongelmaan on kuvan jakaminen useisiin pienempiin toisistaan riippumattomiin kuviin, lohkoihin [Fränti et al. 2001]. Lohkojaolla tarkoitetaan yhden kokonaisen binäärikomponentin jakamista neliönmuotoisiin tasakokoisiin lohkoihin. Esimerkiksi yksi 5000x5000 pikselin kokoinen karttalehti voidaan jakaa 2500:aan 100x100 pikselin lohkoon. Mikäli lohkot on tiivistetty toisistaan riippumatta, on mahdollista purkaa vain näytöllä kulloinkin näkyvät lohkot. Loput lohkot voidaan pitää muistissa tiivistettyinä muistivaatimuksien vähentämiseksi tai jopa siirtää päätelaitteeseen niitä tarvittaessa esimerkiksi GSM-yhteydellä suoraan karttapalvelimelta. Esimerkkki lohkojaosta on esitetty kuvassa 3.5.



Kuva 3.5. Esimerkki lohkojaosta kuvan 1.1. pellot-komponentille.

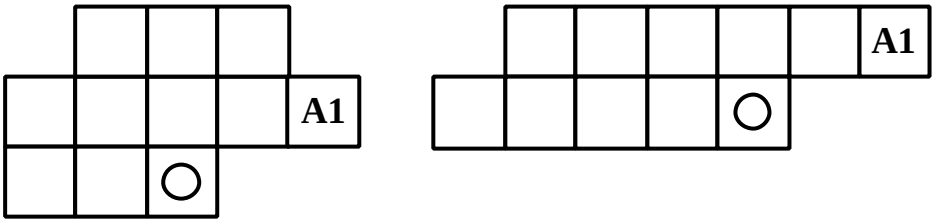
Lohkokoon pienetessä tiivistysteho heikkenee metatiedon, kuten otsakkeiden osuuden kasvaessa. Esimerkiksi JBIG2-menetelmässä lohkokoolla 128×128 tiivistettäessä lisääntyneiden otsakkeiden määrä kasvattaa tiedoston kokoa noin 6 % [Fränti, Ageenko ja Gröhn 2001]. Lisäksi koodaajan sopeutuminen moneen pieneen kuvaan kasvattaa oppimiskustannuksen osuutta ja siten heikentää tiivistystehoa, mutta toisaalta mahdollistaa paremmin sopeutumisen kuvan paikallisiin piirteisiin [Fränti et al. 2001]. Lohkokokoa pienennettäessä myös tyhjien yksiväristen lohkojen osuus kasvaa selvästi etenkin harvoin esiintyvien peltojen ja vesistöjen kohdalla (kuva 3.6). Tämän vuoksi tyhjät lohkot kannattaisikin tiivistää erityisellä koodisanalla tavallisen tiivistyksen sijaan.



Kuva 3.6. Yksiväristen lohkojen osuus binäärikomponenteille lohkokoilla 50-300.

4 JBIG

JBIG on adaptiiviseen kontekstimalliin ja aritmeettiseen koodaajaan perustuva tiivistysmenetelmä. Vaihtoehtoiset koodattavan pikselin välittömässä läheisyydessä sijaitsevien pikseleiden muodostamat pikselinaapuristot on esitelty kuvassa kuvassa 6.1. Pikseli A1 tarkoittaa adaptiivista pikseliä, jonka paikka pikselinaapuristossa voi muuttua kesken koodauksen. Tästä on hyötyä koodattaessa kuvia joissa on säännöllinen rakenne, kuten ditheroidut kuvat [Howard et al. 1998]. *Ditheroinnilla (dithering)* tarkoitetaan mustavalkokuvaa, jossa harmaasävyjä esitetään säännöllisten pikselikuvioiden avulla.



Kuva 4.1. JBIG:ssa käytettävät pikselinaapuristot [JBIG 1993].

Kuvassa 4.2 on esitelty JBIG-algoritmin toiminta. Tiivistettäessä kuva käydään läpi riveittäin vasemmalta oikealle. Jokaista pikseliä tiivistettäessä tutkitaan sen naapuripikseleitä ja lasketaan pikselin värille todennäköisyys naapuripikseleiden perusteella. Tämän jälkeen todennäköisyysmallia päivitetään.

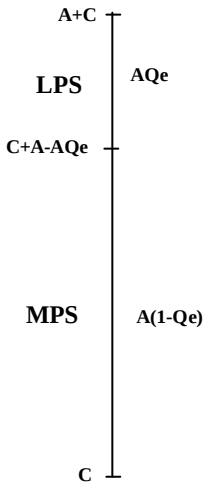
```
for x:=1 to Y do
  for x:=1 to X do
    c:=MuodostaKonteksti(x,y);
    KoodaaPikseli(pxy,c);
  endfor
endfor
```

Kuva 4.2. JBIG-algoritmin toiminta [JBIG 1993].

4.1 QM-koodaaja

JBIG:ssa käytettävä QM-koodaaja on aritmeettisen koodaajan binääriselle tiedolle suunniteltu muunnos [Pennebaker ja Mitchell 1988]. Se ei ole täysin optimaalinen, mutta sopeutuu tavallista aritmeettista koodaajaa nopeammin kuvaan ja toimii nopeammin, koska kaikki kertolaskut on eliminoitu.

QM-koodaajassa välin koodaamisessa käytetään kolmea muuttujaa: alarajaa *C*, välin pituutta *A* ja vähemmän todennäköisen symbolin todennäköisyyttä *Qe*. Jos koodattavan pikselin väri on sillä hetkellä todennäköisempi kahdesta väristä, kutsutaan sitä nimellä *MPS (more probable symbol)* ja päinvastaisessa tapauksessa sitä sanotaan *LPS*:ksi *(less probable symbol)* (kuva 4.3).



Kuva 4.3. Välin jakaminen QM-koodaajassa.

Välin pituus *A* on aina välillä 0.75 – 1.5, ollen keskimäärin 1.0. Kun välin pituus laskee alle 0.75:n, se kaksinkertaistetaan bitinsiirto-operaatioiden avulla. Koodattaessa pikseliä QM-koodaaja olettaa välin pituuden kertolaskujen eliminoimiseksi aina vakioksi 1.0. Näin ollen väli muuttuu seuraavasti:

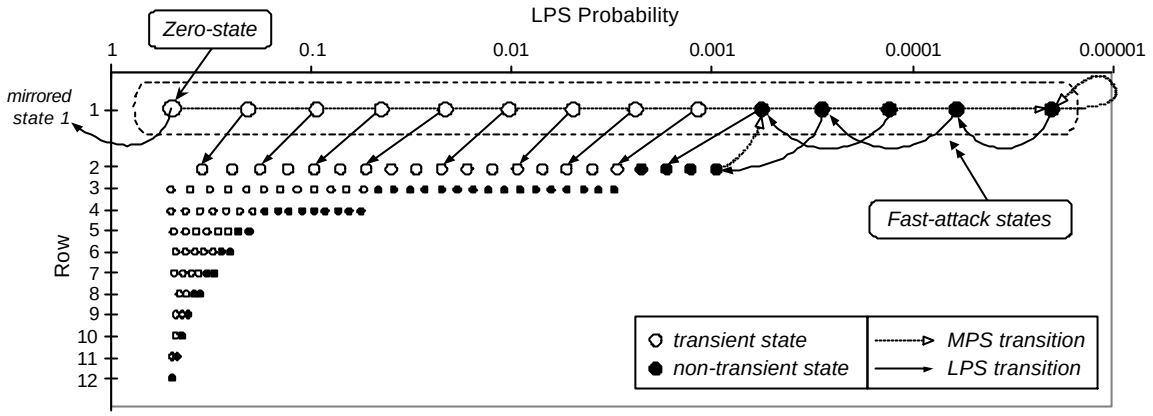
Koodattaessa MPS:

$$\begin{aligned} C &:= C \\ A &:= A \cdot (1 - Qe) = A - A \cdot Qe \approx A - Qe \end{aligned} \tag{5}$$

Koodattaessa LPS:

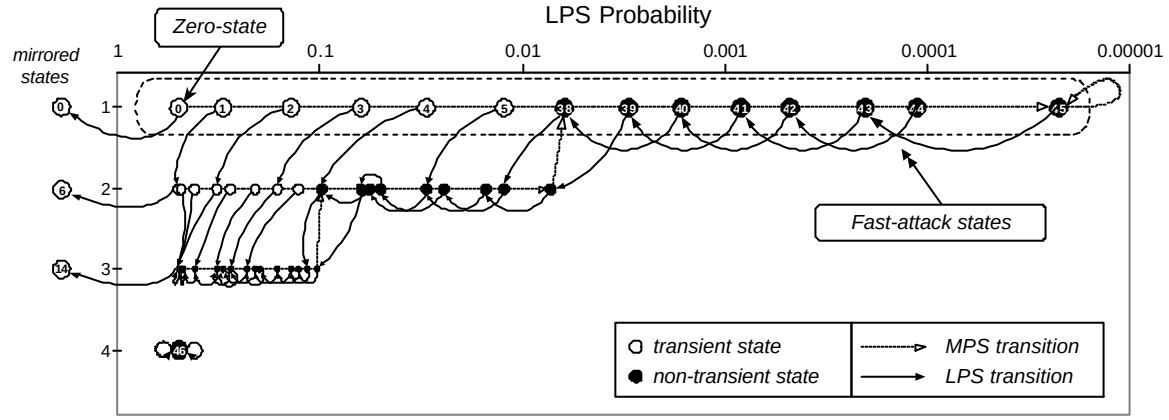
$$\begin{aligned} C &:= C + A \cdot (1 - Qe) = C + A - A \cdot Qe \approx C + A - Qe \\ A &:= A \cdot Qe \approx Qe \end{aligned} \tag{6}$$

QM-koodaaja sisältää myös mallinnuksen, joka ei perustu jo koodattujen pikseleiden frekvensseihin vaan tila-automaattiin (kuva 4.4), jonka avulla kuvaan sopeutuminen nopeutuu. Koodaus alkaa *nollatilasta (zero-state)*. Jokaisesta tilasta on siirtymät kahteen muuhun tilaan koodattavan pikselin värin perusteella. Koodauksen edetessä automaatti vakiintuu *stabiileihin (non-transient)* tiloihin. Aina koodattaessa LPS-symboli siirrytään tilakoneessa vasemmalle tai alas ja MPS:n jälkeen, jos välin pituus A laskee alle 0.75:n siirrytään oikealla tai yläpuolella olevaan tilaan. Välin kaksinkertaistaminen suoritetaan aina LPS:n jälkeen ja myös MPS:n jälkeen, jos tarpeellista. Mikäli välin kaksinkertaistus suoritetaan MPS:n jälkeen, automaatti siirtyy seuraavaan samalla rivillä oikealla puolella olevaan tilaan, jolla on pienempi LPS-todennäköisyys. LPS:n jälkeen kaksinkertaistettaessa väliä siirrytään aina tilaan, jossa on suurempi LPS-todennäköisyys: Vastaava tila alemmalla tasolla, mikäli ollaan *väliaikaisessa* (transient) tilassa, tai edeltävä tila samalla rivillä mikäli ollaan stabiilissa tilassa. Mikäli kuvan todennäköisyydet muuttuvat myöhemmin, voidaan stabiileihin tiloihin siirtyä takaisin muista stabiileista tiloista, mikä mahdollistaa sopeutumisen kuvan paikallisiin vaihteluihin.



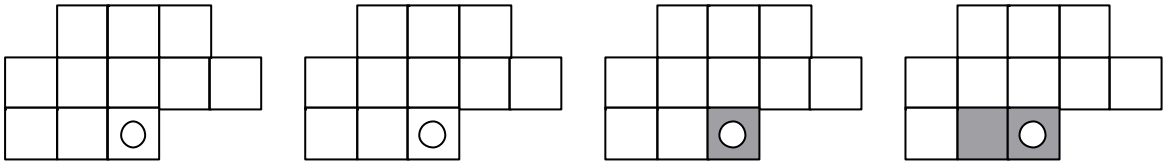
Kuva 4.4. QM-koodaajan käyttämä tila-automaatti todennäköisyyksien ennustamiseen [Pennebaker ja Mitchell 1988].

QM-koodaajasta edelleen kehitetyssä MQ-koodaajassa tilojen lukumäärää on vähennetty 226:sta 94:ään [Ageenko 2000]. MQ-koodaajan tila-automaatti on esitetty kuvassa 4.5.



Kuva 4.5. MQ-koodaajan tila-automaatti [Ageenko 2000].

Tarkastellaan kuvan 2.6 ylärivin koodaamista JBIG-algoritmilla käytettäessä kuvan 4.1 vasemmanpuoleista pikselinaapuristoa. Kuvan ulkopuolella oletetaan olevan valkoisia pikseleitä, jolloin kuvaa koodattaessa muodostuu kuvan 4.6 mukaiset pikselinaapuristot.

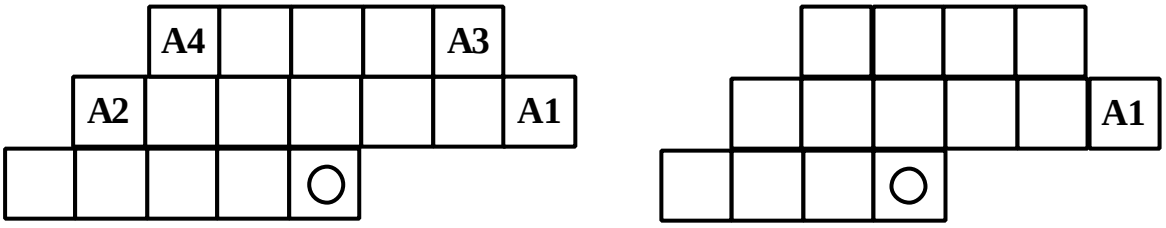


Kuva 4.6. Pikselinaapuristot koodattaessa kuvan 2.6 ylärivi.

Koodattaessa ensimmäinen vasemmassa ylänurkassa oleva valkoinen pikseli, muodostuu naapuristoksi kuvan 4.6 vasemmassa ylänurkassa oleva pikselinaapuristo. Koska kyseessä on ensimmäinen pikseli tässä naapuristossa, pikselin värin todennäköisyydet ovat vielä p(w)=1/2˜0.5 ja p(b)=1/2˜0.5. Kuvan valkoinen pikseli koodataan tätä mallia käyttäen aritmeettisella koodaajalla jonka jälkeen mallia päivitetään. Tällöin saadaan p(w)=2/3˜0.67 ja p(b)=1/3˜0.33. Seuraava koodattava pikseli on myös valkoinen pikseli samanlaisessa naapuristossa. Nyt aritmeettisessa koodauksessa käytetään äsken päivitettyä mallia, jonka jälkeen mallia päivitetään jälleen. Nyt p(w)=3/4˜0.75 ja p(b)=1/4˜0.25. Kolmas koodattava pikseli on musta, mutta pikselinaapuristo on jälleen samanlainen. Aritmeettisessa koodauksessa valitaan eri aliväli, koska koodattava pikseli on musta. Mallin päivityksen jälkeen todennäköisyydet ovat p(w)=3/5˜0.6 ja p(b)=2/5˜0.4. Viimeisen pikselin naapuristo on erilainen, jolloin käytetään eri todennäköisyyksiä. Koska koodattava pikseli on ensimmäinen tässä naapuristossa, p(w)=1/2˜0.5 ja p(b)=1/2˜0.5. Mustan pikselin aritmeettisen koodauksen jälkeen todennäköisyydet tässä pikselinaapuristossa ovat p(w)=1/3˜0.33 ja p(b)=2/3˜0.67.

5 JBIG2

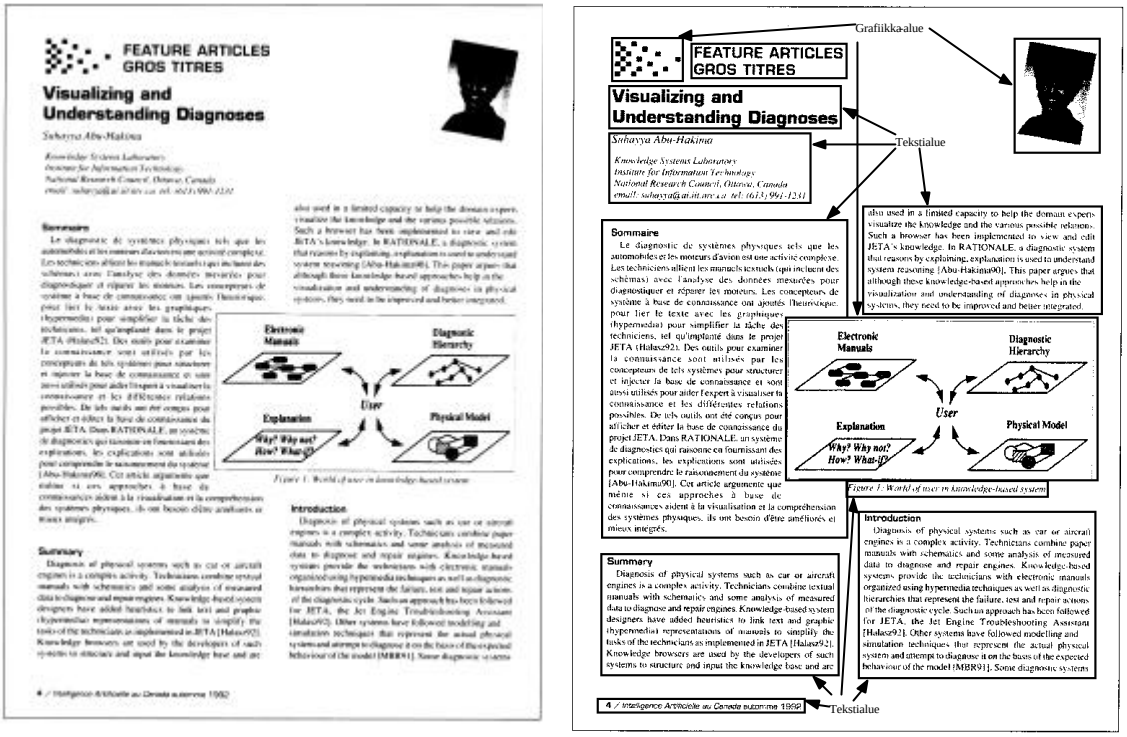
JBIG2:n pohjana on vanhempi JBIG-menetelmä [JBIG 1993], joka on sisällytetty uuteen menetelmään lähes sellaisenaan. JBIG2 pyrkii parantamaan edeltäjänsä tiivistystehoa koodaamalla kuvan eri tyyppiset alueet erilaisilla menetelmillä. *Segmentoinnin* (segmentation) tarkoituksena on jakaa kuva neliskulmaisiin alueisiin, segmentteihin, jotka sisältävät pelkästään tekstiä, harmaasävykuvia tai grafiikkaa. Grafiikan koodaus on lähes identtinen JBIG:n kanssa, JBIG:n pikselinaapuristojen lisäksi on valittavissa kaksi uutta pikselinaapuristoa (kuva 5.1). Harmaasävyalueiden koodaamista ei esitellä tarkemmin, koska siitä ei ole hyötyä karttakuvien tiivistyksessä.



Kuva 5.1. JBIG2:ssa käytettävät pikselinaapuristot kuvan 4.1 naapuristojen lisäksi [JBIG 1999].

5.1 Kuvan segmentointi

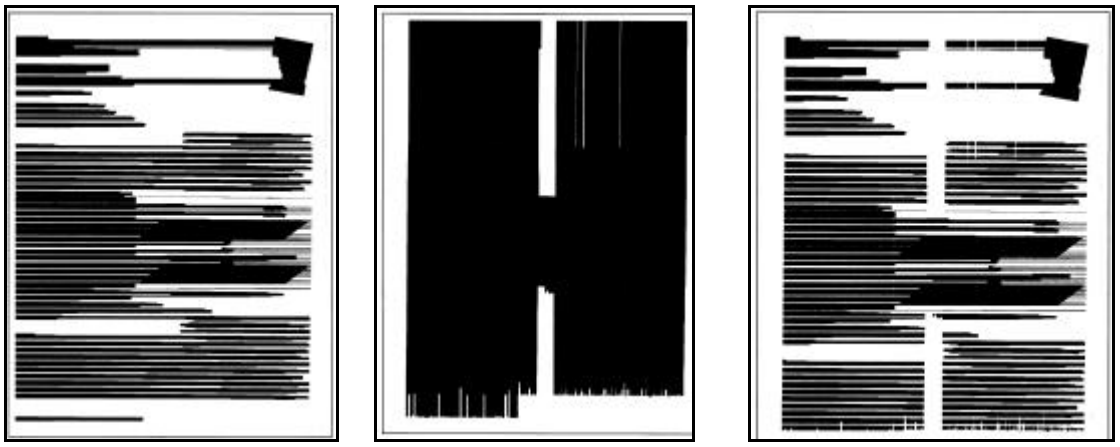
Jotta kuvan erityyppiset alueet voitaisiin koodata juuri niille suunnitelluilla menetelmillä, on ne voitava jotenkin erotella toisistaan. Tällöin puhutaan kuvan *segmentoinnista (segmentation)*. Segmentoinnin tarkoituksena on jakaa kuva suorakulmaisiin *alueisiin (region),* joista kukin sisältää pelkästään tekstiä tai grafiikkaa (kuva 5.2). Itse standardi ei kuitenkaan määrittele miten segmentointi on suoritettava vaan ainoastaan sen miten segmentoitu kuva talletetaan tiedostoon, jotta dekoodaaja osaisi sen purkaa.



Kuva 5.2. Vasemmalla alkuperäinen kuva joka sisältää erityyppistä tietoa. Oikealla esimerkki saman kuvan segmentoinnista alueisiin.

Segmentoinnin toteuttamiseen on olemassa kaksi vastakkaista lähestymistapaa: *ylhäältä alas (top-down)* ja *alhaalta ylös (bottom-up)*. Ylhäältä alas-menetelmät tarkastelevat kuvaa kokonaisuudessaan ja etsivät siitä yhteenkuuluvia kokonaisuuksia. Alhaalta ylös-menetelmissä merkit yhdistetään toistuvasti yhä suuremmiksi kokonaisuuksiksi.

Alhaalta ylös-menetelmät aloittavat yleensä pehmentämällä *(smearing) kuvan*. Pehmennyksessä väritetään mustaksi kaikki kahden mustan pikselin väliset pikselit, jos mustien pikseleiden keskinäinen etäisyys on alle ennalta määrätyn raja-arvon. Pehmennys voidaan tehdä erikseen vaaka- ja pystysuunnissa, jonka jälkeen kaksi saatua kuvaa yhdistetään loogisella AND-operaatiolla. Witten, Moffat ja Bellin [1999] mukaan menetelmä toimii kohtuullisesti kuville, joissa on yksinkertainen sijoittelu, kuten kuva 5.2. Esimerkki pehmennysoperaation toiminnasta kuvalle 5.2 on esitetty kuvassa 5.3.

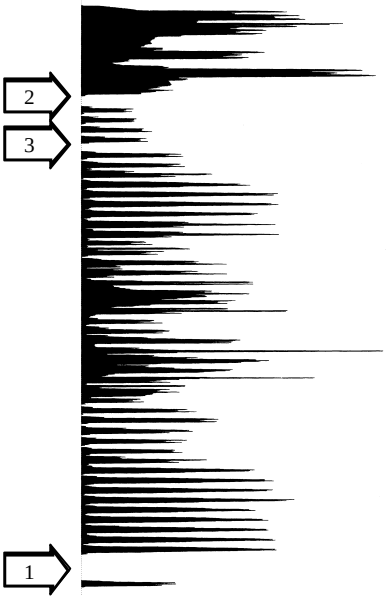


a **b** **c**

Kuva 5.3. Pehmennysoperaatio kuvalle 5.2. Pehmennys vaakasuunnassa (a), pystysuunnassa (b) sekä edelliset yhdistettynä (c) [Witten, Moffat, Bell 1999].

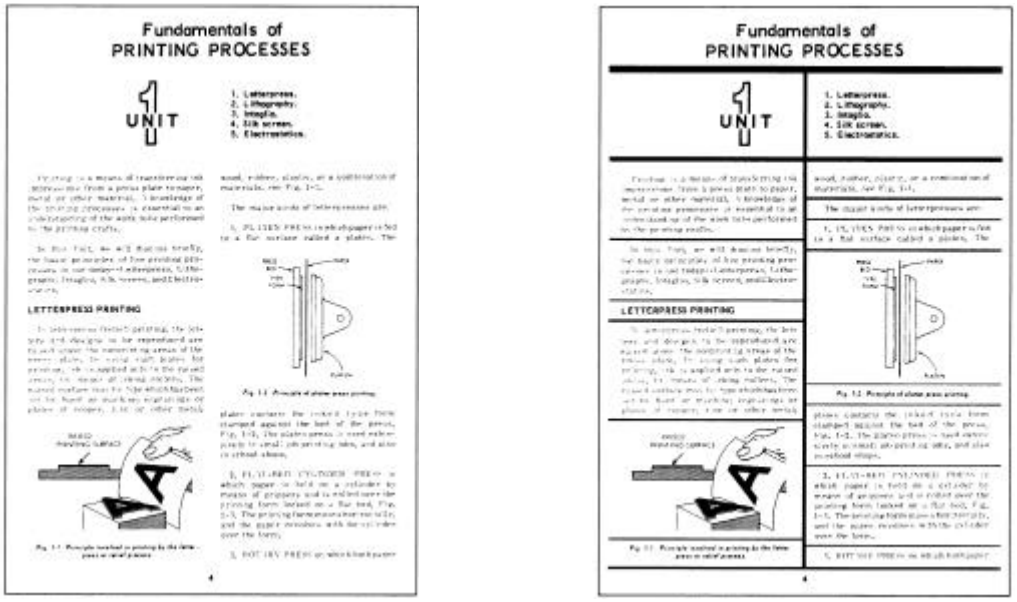
Toinen vaihtoehto suorittaa pehmennys on tutkia jokaisen pikselin naapuristoa ja värittää pikseli mustaksi jos naapuristossa on vähintään ennalta määrätty määrä mustia pikseleitä. Tämän jälkeen kuva käydään uudelleen läpi ja väritetään pikseli mustaksi, jos naapuristossa on yksikin musta pikseli. Tämä menetelmä on hitaampi mutta se antaa yleensä paremman lopputuloksen [Witten, Moffat, Bell 1999].

Ylhäältä alas-menetelmä jakaa kuvan lohkoihin rekursiivinen X-Y leikkaus algoritmilla [Witten, Moffat, Bell 1999]. Jokaisessa vaiheessa lasketaan kuvan jokaisella rivillä ja jokaisessa sarakkeessa olevien mustien pikseleiden lukumäärä. Tuloksena saadaan kaksi kuvan 5.4 tyylistä pylväsdiagrammia, joista toisessa on rivisummat ja toisessa sarakesummat.



Kuva 5.4. Kuvan 5.2 mustien pikseleiden lukumäärä riveittäin sekä kolme ensimmäistä leikkauskohtaa.

Diagrammeista etsitään levein valkoisten pikseleiden muodostama väli, josta kuva leikataan kahteen osaan. Sen jälkeen kuva leikataan seuraavaksi leveimmästä välistä ja tätä toistetaan kunnes vähintään ennalta määrätyn levyisiä välejä ei enää ole jäljellä (kuva 5.4). Kuvassa paksummat esittävät aikaisemmassa vaiheessa tehtyjä leikkauksia.



Kuva 5.4. Rekursiivinen X-Y leikkaus-algoritmi. Vasemmalla alkuperäinen kuva ja oikealla leikkausten paikat. [Witten, Moffat, Bell 1999].

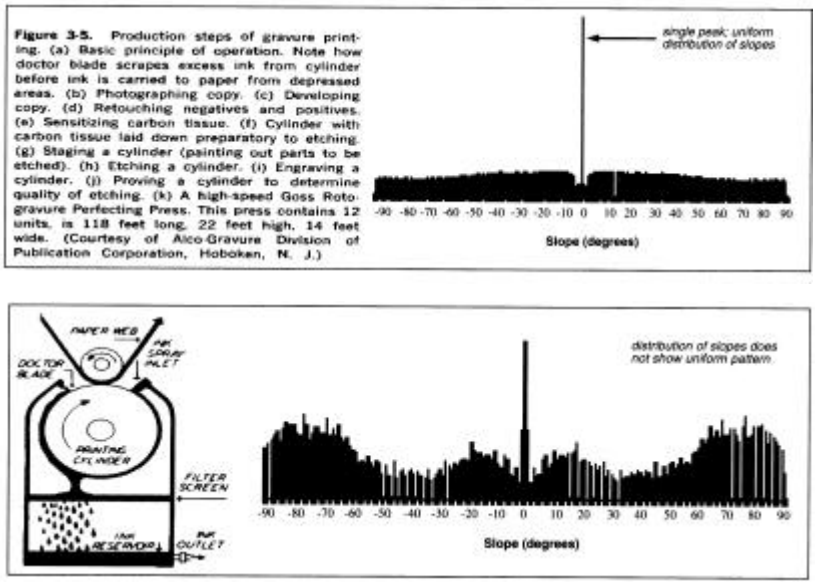
Algoritmi toimii hyvin yksinkertaisen rakenteen sisältäville kuville, kuten kuvan 5.4 esimerkki, mutta esimerkiksi kuvan 5.2 se segmentoi huonosti, koska siinä alueiden reunat eivät ole samalla tasalla pysty –ja vaakasuunnassa. Ylhäältä alas- ja alhaalta ylös-menetelmät voidaan yhdistää suorittamalla kuvan pehmennys ennen rekursiivista X-Y leikkausta, jolloin päästää yleensä parempaan lopputulokseen kuin suorittamalla menetelmät erikseen.

5.2 Segmenttien luokittelu

Kun kuva on segmentoitu alueisiin, on alueet vielä luokiteltava niiden sisältämän tiedon mukaan tekstialueisiin ja grafiikkaan. Luokittelu on tärkeä vaihe, koska epäonnistunut luokittelu johtaa selvästi heikompaan tiivistystulokseen. Jos segmentoinnin tuloksena on saatu eroteltua yksittäiset tekstikappaleet, kuten kuvassa 5.4, voidaan luokitteluperusteena

käyttää alueiden kokoa ja muotoa. Tekstialueet ovat yleensä leveitä ja matalia, kun taas grafiikka-alueet ovat epäsäännöllisempiä.

Luokittelu voidaan tehdä myös *merkkien (mark)* perusteella. Yhdellä merkillä tarkoitetaan alueella olevaa yhtenäistä mustien pikseleiden yhdistämää aluetta. Merkkien lukumäärä alueella on yleensä suuri tekstille ja pieni grafiikalle. Witten, Moffat ja Bell [1999] esittävät menetelmän, jossa jokaiselle kahden merkin kombinaatiolle lasketaan niiden välinen kulma. Näin saadusta jakaumasta muodostetaan *kallistumadiagrammi (slope diagram),* kuten kuvassa 5.5. Tekstille jakauma on tekstin säännöllisestä sijoittelusta johtuen hyvin tasainen ja grafiikalle epäsäännöllinen.

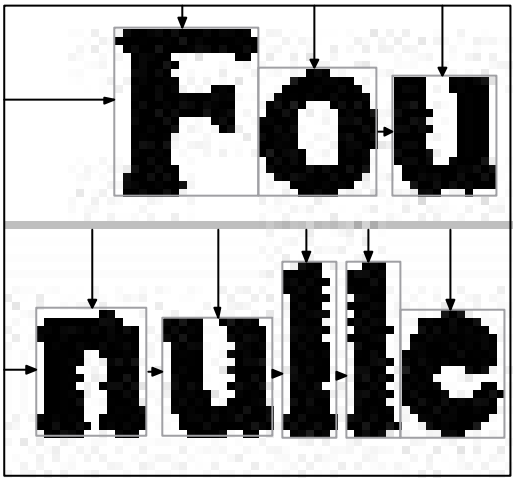


Kuva 5.5. Esimerkki kallistumadiagrammista tekstille (ylempi) ja grafiikalle (alempi).

5.3 Tekstin koodaus

Tekstialueet koodataan sanakirjan sekä siihen viittaavan tekstialueen avulla. Koodattava alue jaetaan ensin pystysuunnassa vakiokorkeusiin *raitoihin (stripe)*, ja jokaiseen raitaan kuuluvat kirjaimet koodataan erikseen (kuva 5.6).

Yhdeksi kirjaimeksi määritellään yleensä mustien pikseleiden muodostama yhtenäinen alue. Raidan sisältä etsitään kaikki kirjaimet jonka jälkeen ne koodataan järjestyksessä vasemmalta oikealle. Jokaista kirjainta koodattaessa tutkitaan löytyykö täsmäävä kirjain jo sanakirjasta. Jos kirjain löytyy, koodataan sanakirjan indeksi sekä kirjaimen sijainti. Sijaintitiedoissa koodataan y-koordinaatti raidan yläreunasta ja x-koordinaatti vasemmanpuoleisen kirjaimen oikeasta reunasta [JBIG 1999]. Mikäli täsmäävää kirjainta ei löydy sanakirjasta, lisätään sanakirjaan uusi kirjain sekä koodataan sen sijainti.



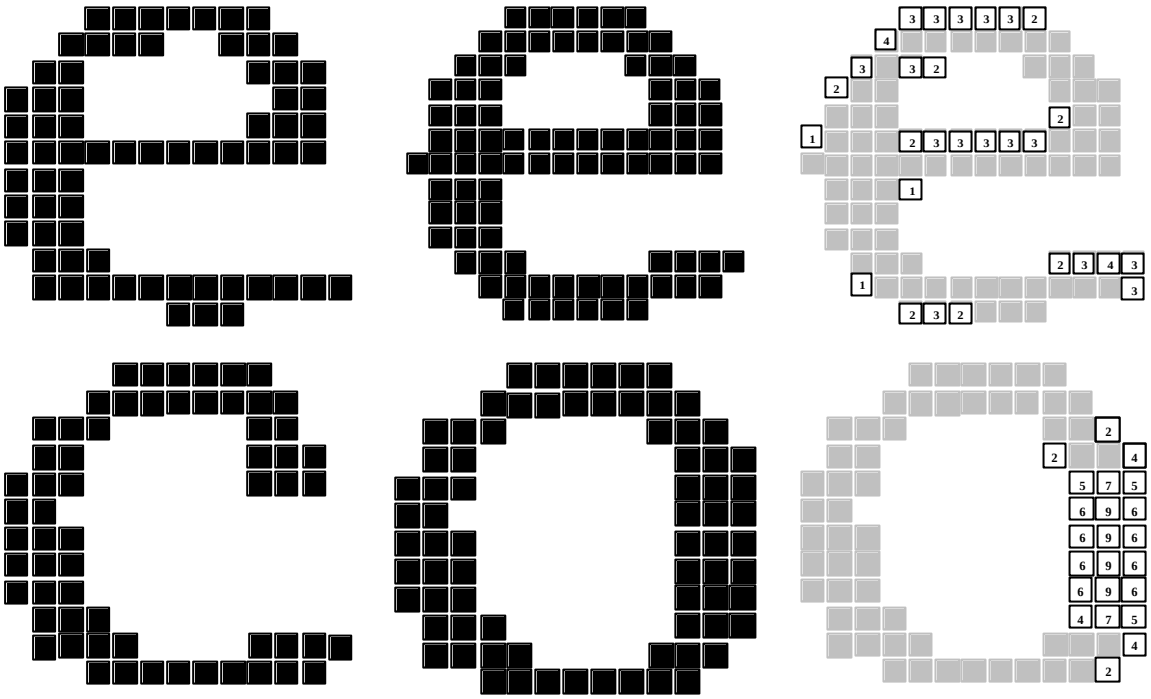
Kuva 5.6. Kirjainten sijainnin koodaus. X-koordinaatti koodataan suhteessa edelliseen kirjaimeen ja y-koordinaatti siirtymänä raidan yläreunasta.

Karttakuvissa tekstin koodausta voitaisiin hyödyntää peruskartta-komponentin teksteihin sekä muihin usein esiintyviin symboleihin, kuten puulaji ja kivi. Värierottelua käytettäessä myös mahdollisesti muissa komponenteissa olevat näkyvät kirjaimet voidaan lisätä sanakirjaan. Karttakuvissa ongelmana on tekstin sekoittuminen teiden ja muun tiedon

sekaan, jolloin kirjainten erottelu vaikeutuu. Lisäksi tekstit eivät monesti ole vaakasuorassa, vaan suunta vaihtelee esitettävän kohteen mukaan (kuva 1.1). Tämä kasvattaa sanakirjan kokoa ja heikentää tiivistystehoa.

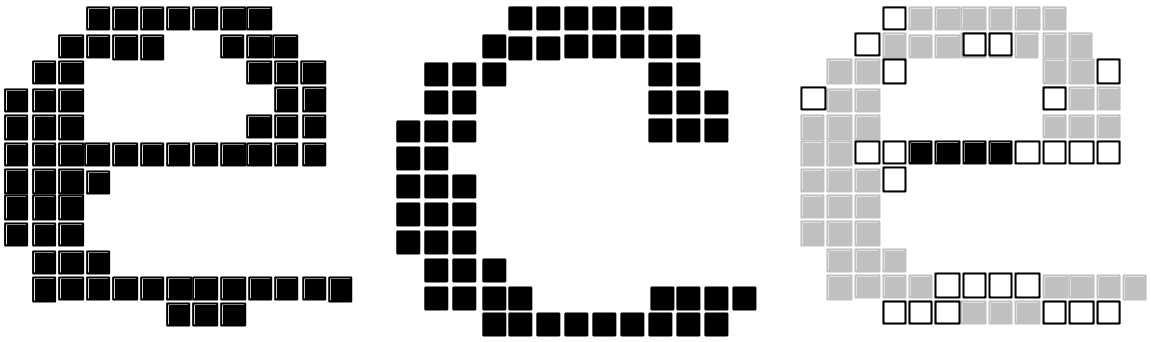
Tekstin koodauksessa oleellisessa osassa on *merkkien täsmäys (pattern matching)*, jonka avulla päätetään ovatko sanakirjassa oleva ja koodattava kirjain samat. Witten, Moffat ja Bell [1999] esittelevät kaksi täsmäysmenetelmää: *globaali täsmäys (global matching)* ja *paikallinen täsmäys (local matching)*.

Globaalissa täsmäyksessä tutkitaan kahden kirjaimen eroavien pikseleiden kokonaismäärää, jonka perusteella muodostetaan *virhekartta (error map)*. Virheitä painotetaan sen mukaan missä yhteydessä ne esiintyvät. Esimerkiksi kuvassa 5.7 ylärivillä verrataan kahta e-kirjainta esittävää bittikarttaa. Kuvien välillä eroavien pikseleiden lukumäärä (29) on suurempi kuin alarivillä verrattaessa c- ja o-kirjaimia toisiinsa (23). Alemmassa virhekartassa eroavat pikselit esiintyvät kuitenkin ryhmässä, jolloin niillä on suurempi painoarvo. Jokaiselle eroavalle pikselille virhekartassa lasketaan painotettu arvo sen 3×3 naapuristossa olevien eroavien pikseleiden summana. Näin laskettuna ylärivin e-kirjaimille saadaan virheeksi yhteensä 75 ja alarivin c-ja o-kirjainten virheeksi 131. Täsmäys hyväksytään jos virhe on pienempi kuin ennalta sovittu raja-arvo. Tässä tapauksessa raja-arvon tulisi olla välillä 75-131 jotta ylempi täsmäys hyväksyttäisiin ja alempi hylättäisiin.



Kuva 5.7. Kirjainten e ja e sekä c ja o globaali täsmäys käyttäen virhekarttaa [Witten, Moffat, Bell 1999].

Paikallisessa täsmäyksessä virhekarttaan merkitään eroavat pikselit (valkoiset pikselit kuvan 5.8 virhekartassa) sekä erityisesti sellaiset pikselit, jotka ovat jommassakummassa täsmättävistä bittikartoissa mustia ja toisessa kokonaan valkoisten pikseleiden ympäröimiä (mustat pikselit kuvan 5.8 virhekartassa). Täsmäys hylätään, jos mustalla pikselillä virhekartassa on naapureinaan enemmän mustia pikseleitä kuin ennalta määrätty raja-arvo tai samalla kohdalla sijaitseva pikseli on toisessa bittikartassa kokonaan valkoisten tai mustien pikseleiden ympäröimä. Kuva 5.8 havainnollistaa kuinka e- ja c-kirjainten paikallinen täsmäys johtaa täsmäyksen hylkäämiseen.



Kuva 5.8. Paikallinen merkkien täsmäys e- ja c-kirjaimille.

JBIG2:ssa on kaksi tapaa koodata tekstialueita: *PM&S (pattern matching and substitution)* sekä Howardin 1997 esittämä *SPM (soft pattern matching)*.

PM&S-menetelmässä (kuva 5.9) jokaista koodattavaa kirjainta verrataan sanakirjassa oleviin symboleihin. Jos koodattava kirjain muistuttaa tarpeeksi sanakirjassa olevaa symbolia, koodataan täsmäävän symbolin indeksi sanakirjassa. Muussa tapauksessa koodattava merkki tulkitaan uudeksi symboliksi ja sen bittikartta koodataan JBIG:llä. Koodauksen jälkeen symboli lisätään sanakirjaan. PM&S-menetelmä mahdollistaa suuren häviöllisen tiivistystehon jos kirjainten täsmäyksessä ollaan tarpeeksi ahneita. Tästä kuitenkin seuraa väistämättä virheitä, jos koodattava kirjain täsmää väärään symboliin.

PM&S-algoritmi

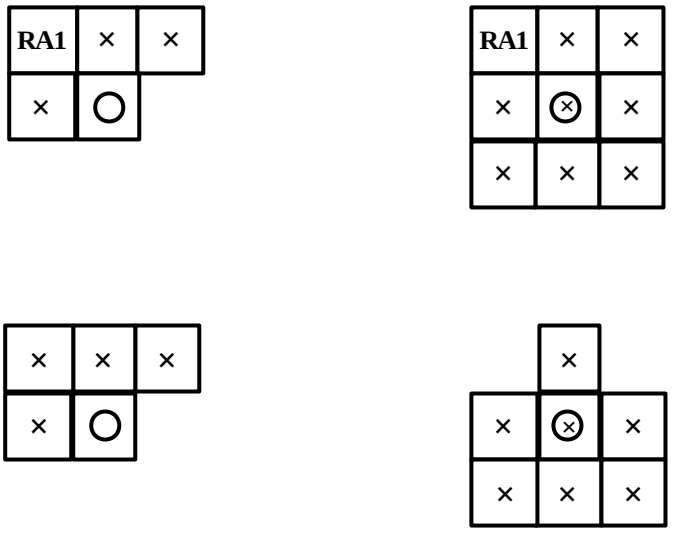
```
segmentoi alue symboleihin
for jokaiselle symbolille do
   etsi symbolille täsmäys
sanakirjasta
   if täsmäys löytyy
      koodaa täsmäävän symbolin
       indeksi
   else
      koodaa bittikartta
      lisää uusi symboli sanakirjaan
  endif
  koodaa symbolin sijainti
endfor
```

SPM-algoritmi

```
segmentoi alue symboleihin
for jokaiselle symbolille do
   etsi symbolille täsmäys sanakirjasta
   if täsmäys löytyy
      koodaa täsmäävän symbolin indeksi
      koodaa bittikartta täsmäävän symbolin
       avulla
      mahdollisesti lisää uusi symboli
       sanakirjaan
   else
      koodaa bittikartta
      lisää uusi symboli sanakirjaan
   endif
   koodaa symbolin sijainti
endfor
```

Kuva 5.9. JBIG2:n vaihtoehtoiset merkkienkoodausalgoritmit.

Täsmäysvirheiden välttämiseksi voidaan käyttää häviötöntä SPM-menetelmää (Howard et al. 1998]. Siinä koodataan kirjaimen sijainnin ja sanakirjaindeksin lisäksi *tarkennustietoa (refinement data)*, jonka avulla alkuperäinen merkki voidaan palauttaa. Käytännössä tämä tarkoittaa alkuperäisen bittikartan koodaamista indeksin lisäksi. Tarkennustieto koostuu koodattavan kirjaimen pikseleistä, jotka koodataan käyttäen konstekstina koodattavan merkin jo koodattuja pikseleitä sekä täsmäävän merkin pikseleitä (kuva 5.10). Tällä tavoin sanakirjan symbolien informaatiota voidaan tehokkaasti hyödyntää ja bittikartta saadaan yleensä tiivistettyä huomattavasti pienempään tilaan kuin jos täsmäystä ei olisi suoritettu ja olisi käytetty JBIG:iä. SPM ei tarvitse tarkkaa symbolien täsmäysmenetelmää kuten PM&S, koska täsmäysvirhe johtaa vain huonompaan tiivistyssuhteeseen.



Kuva 5.10. SPM:ssä käytettävät vaihtoehtoiset pikselinaapuristot. Vasemmalla koodattavan symbolin pikselit, oikealla täsmäävän symbolin pikselit [Howard et al. 1998].

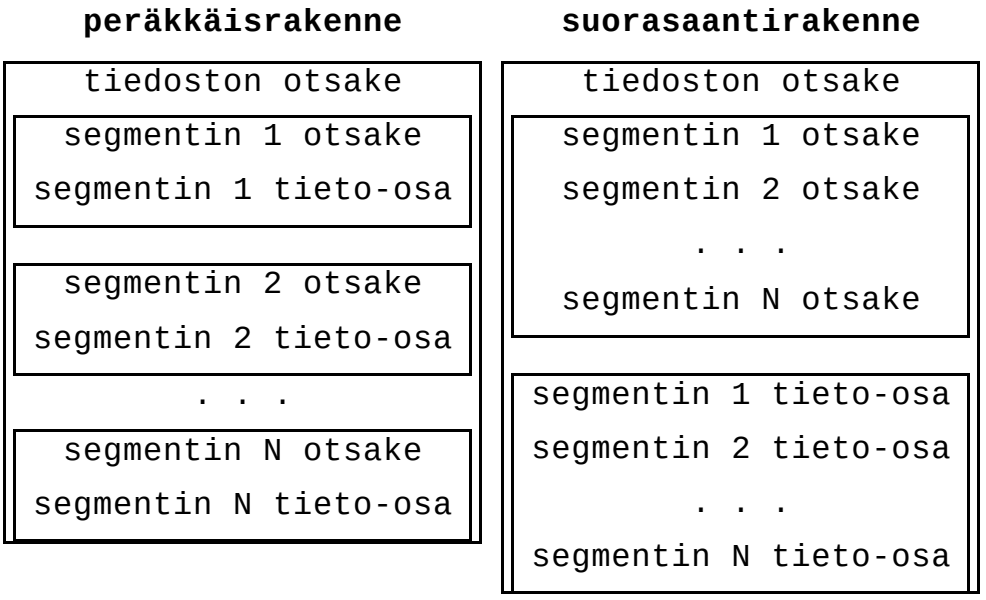
6 MISS

Joensuun yliopiston tietojenkäsittelytieteen laitoksen RealMap-projektissa [RealMap 2000] oli tavoitteena kehittää uusi karttakuville soveltuva formaatti, joka kumoaisi JBIG2:n rajoitukset tiivistystehon heikentymättä. Projektin aikana määrittelimme ja toteutimme MISS-formaatin, jota on edelleen kehitetty DynaMap-projektissa [DynaMap 2001]. Tärkeimpänä uudistuksena tiedostoformaatissa on tuki binäärikomponenttien jakamiselle tasakokoisiin lohkoihin ilman otsaketietojen toistamista sekä tehokas lohkojen suorasaanti. Jokainen lohko tiivistetään erikseen, jolloin koodaaja pystyy sopeutumaan paremmin kuvan paikallisiin ominaisuuksiin. Lisäksi lohkot on tiivistetty toisistaan riippumatta, joten vain näytöllä näkyvät osat voidaan purkaa, jolloin myös muistivaatimukset vähenevät [Fränti, Kopylov ja Ageenko 2002].

6.1 Lohkojaon toteutus JBIG2:ssa

JBIG2-standardi määrittelee ainoastaan dekoodaajan toiminnan sekä tiedostoformaatin muodon, koodaajan toteutus jätetään avoimeksi. Seuraavaksi tutkitaan kuinka luvussa 3.4 esitelty lohkojako voidaan tehokkaasti toteuttaa JBIG2:ssa.

JBIG2-tiedosto koostuu useista erillisistä osista, tiedostosegmenteistä. Jokainen tiedostosegmentti koostuu kahdesta osasta, otsakkeesta ja tieto-osasta. Otsake-osa sisältää tietoa segmentin tyypistä, koosta sekä mahdollisista segmenttien välisistä riippuvuuksista. Standardi määrittelee kaksi vaihtoehtoista tiedostorakennetta, *peräkkäisrakenteen (sequential)* sekä *suorasaannin (random-access)*, joissa tiedostosegmentin osat on järjestetty eri tavoin (kuva 6.1).



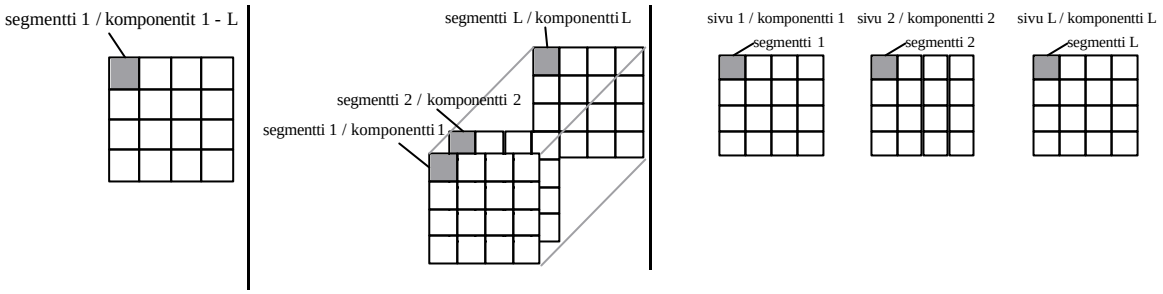
Kuva 6.1. JBIG2:n vaihtoehtoiset tiedostorakenteet.

Peräkkäisrakenteessa segmentin tieto-osa sijaitsee tiedostossa heti sitä vastaavan otsake-osan jälkeen. Suorasaantirakenteessa kaikki otsakkeet sijaitsevat peräkkäin tiedoston alussa ja tieto-osat seuraavat perässä. Seuraavassa tarkastelussa käytetään suorasaantirakennetta, koska siinä kaikki otsakeosat voidaan helposti lukea heti muistiin ja laskea niiden avulla vastaavien tieto-osien sijainnit. Lisäksi kuvasegmentit määritellään siten että yksi lohko kuvassa vastaa yhtä segmenttiä.

Tiedostoformaatti mahdollistaa kuvan jakamisen monelle sivulle, mahdollistaen loogisen tavan jakaa eri binäärikomponentit eri sivuille. Toinen hyödyllinen ominaisuus, segmentointi mahdollistaa lohkojen tallettamisen C×C kokoisiin neliönmuotoisiin paloihin.

Tarkastellaan kolmea eri tapaa tallettaa useita komponentteja samaan tiedostoon [Fränti, Ageenko ja Gröhn 2001]:

1) Lohkon komponentit samassa segmentissä peräkkäin sijoitettuna
2) Lohkon komponentit päällekkäisissä segmenteissä
3) Lohkon komponentit eri segmenteissä eri sivuilla



Kuva 6.2. Erilaiset tavat tallettaa lohkot JBIG2-tiedostoon. Tapa 1 vasemmalla, tapa 2 keskellä ja tapa 3 oikealla.

Yleensä otsaketietojen osuus Ω on kuvatiedostossa merkityksettömän pieni. Lohkojaon seurauksena tiedostosegmenttien lukumäärä kuitenkin kasvaa merkittävästi kasvattaen siten myös tiedoston kokoa. Myös kokonaan yksiväriset tyhjät lohkot on talletettava otsakkeineen. JBIG2-tiedostossa on kolmenlaisia otsakkeita. Tiedosto-otsakkeita (file header) on aina tasan yksi kooltaan 13 tavua. Sivu-otsakkeita (43 tavua) on yksi jokaista sivua kohden ja 31 tavun kokoisia segmentin otsakkeita yksi jokaista segmenttiä kohden. Näin ollen otsakkeiden vaatima tila kokonaisuudessaan lasketaan kaavalla

$$\Omega = \Omega_{tiedosto} + \Omega_{sivu} + \Omega_{segmentti} \cdot N_{sivu} \tag{7}$$

Vaihtoehdossa 1 kaikki komponentit on sijoitettu peräkkäin samaan segmenttiin. Tällä tavoin otsaketietojen osuus pienenee. Ratkaisu ei ole kuitenkaan täysin JBIG2-standardin mukainen, sillä standardi-dekoodaaja dekoodaa vain ensimmäisen komponentin. Tässä tapauksessa otsakkeiden osuus on

$$\Omega_1 = 13 + 43 + 31 \cdot \frac{X \cdot Y}{C^2} \tag{8}$$

Vaihtoehdossa 2 samaan lohkoon kuuluvat eri komponentit sijoitetaan samaan kohtaan päällekkäisiin segmentteihin. Ratkaisu on täysin standardin mukainen, mutta otsakkeiden osuus on vaihtoehtoon 1 verrattuna L-kertainen, missä L on komponenttien lukumäärä:

$$\Omega_2 = 13 + 43 + L \cdot 31 \cdot \frac{X \cdot Y}{C^2} \quad (9)$$

Vaihtoehdossa 3 jokainen komponentti sijoitetaan omalle sivulle. Katseluohjelman on osattava yhdistää eri sivut yhdeksi näkymäksi. Segmenttien otsakkeiden lukumäärä on sama kuin vaihtoehdossa 2, mutta sivujen otsakkeiden osuus kasvaa L-kertaiseksi:

$$\Omega_3 = 13 + L \cdot 43 + L \cdot 31 \cdot \frac{X \cdot Y}{C^2} \quad (10)$$

Vaihtoehto 1 on paras vaihtoehto mikäli tiedoston koko on ainoa kriteeri. Se ei kuitenkaan mahdollista suorasaantia eri komponentteihin. Toisaalta vaihtoehto 2 on ainoa, jonka standardi-dekoodaaja osaa näyttää oikein. Valitusta vaihtoehdosta riippumatta suurin ongelma on kuitenkin suorasaannin toteutus JBIG:ssa. Tiivistetyt lohkot vaativat eri määrän tilaa, joten lohkojen alkukohtia tiedostossa ei ennalta tiedetä. Tiedosto-segmenttien data-osien koot on kerrottu tiedoston alussa sijaitsevissa segmenttien otsakkeissa ja segmenttien kokojen perusteella voidaan laskea jokaisen segmentin sijainti tiedostossa. Tyhjiä lohkoja ei kuitenkaan voi jättää pois tiedostosta ja segmenttien järjestyksen on oltava sama kuin otsakkeiden järjestys. Nämä seikat voivat huonontaa oleellisesti tiivistystehoa [Fränti, Ageenko ja Gröhn 2001].

Taulukossa 6.1 on laskettu otsakkeiden osuudet tiedoston kokonaiskoosta liitteen testikuville. Jokaisesta kuvasta on tiivistetty neljä binäärikomponenttia: peruskartta, vesistöt, pellot ja korkeuskäyrät. Jokainen komponentti on edelleen jaettu kolmeen eri lohkokokoon:128 × 128, 256 × 256 sekä 5000 × 5000 (ei lohkojakoa). Lohkot on tiivistetty JBIG2-menetelmällä.

C = 5000	124101	201401	263112	431204	431306	keskiarvo
Ω_1	0.09 %	0.02 %	0.04 %	0.02 %	0.02 %	0.04 %
C = 256	124101	201401	263112	431204	431306	keskiarvo
Ω_2	9.36 %	3.01 %	6.52 %	3.54 %	3.83 %	5.25 %
C = 128	124101	201401	263112	431204	431306	keskiarvo
Ω_3	27.65 %	2.76 %	20.40 %	12.18 %	12.39 %	15.08 %

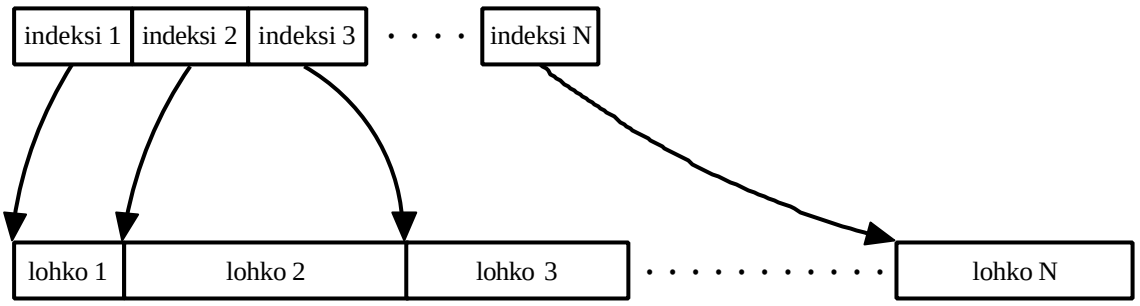
Taulukko 6.1. Otsaketietojen osuus suhteessa tiedoston kokoon eri lohkokoilla [Fränti, Ageenko ja Gröhn 2001].

Pienennettäessä lohkokokoa otsaketietojen osuus nousee varsin suureksi. Otsakkeet sisältävät paljon karttakuville hyödyttömiä kenttiä, kuten segmenttien välisiä viittauksia. Lisäksi segmentin koko on talletettu erikseen jokaiseen segmenttiin, vaikka ne ovat vakiokokoisia. Otsakkeiden osuuden vähentämiseksi standardiin on tehty ehdotus monitasoisille kuville, jolloin samalle sivulle voitaisiin tallentaa useita bittitasoja [Tompkins ja Kossentini 1999].

6.2 Lohkojaon toteutus MISS-menetelmässä

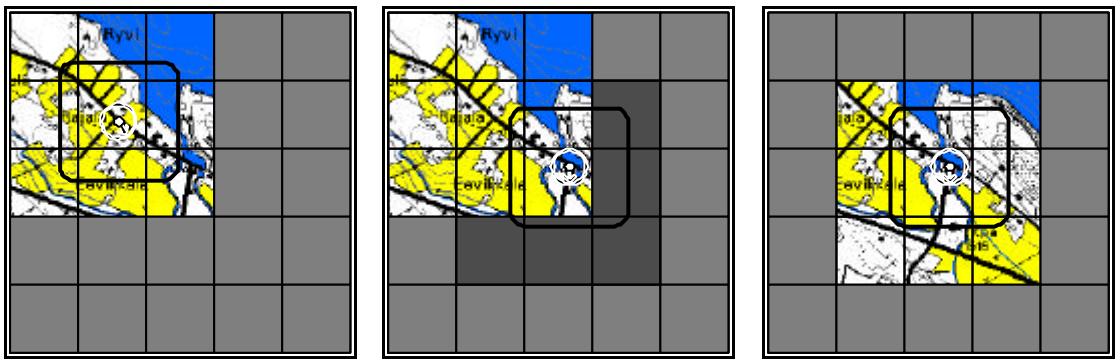
Seuraavaksi tarkastellaan millaisia muutoksia MISS-menetelmässä on tehty, jotta suorasaanti on saatu toteutettua tehokkaammin ja joustavammin kuin JBIG2:ssa.

Jotta lohkojakoa voitaisiin hyödyntää tehokkaasti, jokaiseen lohkoon täytyy olla tehokas suorasaanti. Vaikka aiemmin esitellyissä menetelmissä saataisiinkin tallennettua monta lohkoa samaan tiedostoon, niitä ei pystytä tehokkaasti käyttämään. Ratkaisuksi tähän ongelmaan MISS käyttää lohkoille indeksitaulua tiedoston alussa. Indeksitaulu sisältää tiedosto-osoittimen, jonka avulla halutun lohkon kuvadataan päästään suoraan käsiksi (Kuva 6.3). Lisäksi kokonaan yksivärisille lohkoille on määritelty indeksitaulussa oma koodinsa, jolloin itse kuvadataa ei tarvitse lainkaan tiivistää. Tästä on hyötyä erityisesti lohkokokoa pienennettäessä, jolloin yksiväristen lohkojen määrä kasvaa nopeasti.



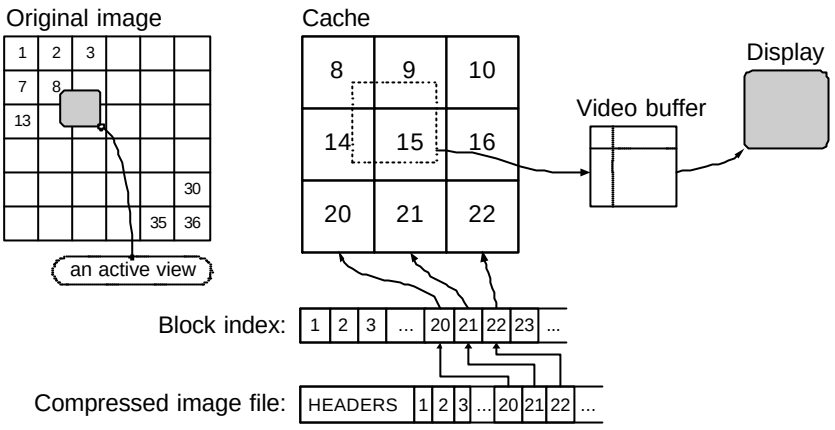
Kuva 6.3. Lohkoittaisen suorasaannin toteutus MISS-tiedostossa.

Näin toteutettu suorasaanti mahdollistaa myös dynaamisen kartan käytön: karttaan voidaan lisätä ja siitä voidaan poistaa lohkoja käyttäjän liikkumisen mukaan (kuva 6.4). Ainoastaan näytöllä näkyvät lohkot on purettu muistiin ja niitä päivitetään käyttäjän liikkeiden mukaan.



Kuva 6.4. Dynaaminen kartan käyttö reaaliaikaisesti. (a) alkutilanne, (b) käyttäjän liikkuminen, (c) päivitetty kartta. [Fränti et al. 2002].

Käytön nopeuttamiseksi voidaan käyttää erityistä välimuistia purettujen lohkojen väliaikaiseen varastointiin (kuva 6.5). Välimuisti sisältää purettuina nykyistä näkymää välittömästi ympäröivät lohkot. Välimuistin kokoa voidaan säätää saatavilla olevan vapaan muistitilan mukaan ja suorittimen ollessa toimettomana ympäröiviä lohkoja voidaan purkaa taustalla [Fränti et al. 2002].



Kuva 6.5. Välimuistin käyttö kartanhallinnassa [Fränti et al. 2002].

6.3 Eteenpäin-adaptiivinen mallinnus

Lohkojen lukumäärää kasvatettaessa yksittäisen lohkon koko pienenee, jolloin koodaajalla on vähemmän aikaa sopeutua. Tämä kasvattaa oppimiskustannusta, joka puolestaan heikentää tiivistystehoa. Lisäksi pienillä lohkoilla ongelmaksi muodostuu tiivistyksen tehottomuus lohkon reunojen lähellä, koska lohkon ulkopuolisia pikseleitä ei voida käyttää kontekstissa.

Lohkokokoa pienennettäessä myös oppimiskustannus kasvaa, koska lohkojen lukumäärä kasvaa ja yksittäisen lohkon koko pienenee. Tämän ongelman ratkaisemiseksi MISS mahdollistaa *eteenpäin-adaptiivisen* (forward-adaptive) mallinnuksen [Ageenko ja Fränti 1999]. Eteenpäin adaptiivinen mallinnus käy jokaisen lohkon läpi kaksi kertaa. Ensimmäisen kerran jälkeen kuvasta muodostetaan malli, jota käytetään aina alkumallina uutta lohkoa koodatessa (kuva 6.6). Alkumalli tehostaa myös koodausta, erityisesti pienillä lohkokoilla. Käytännössä kyseessä on siis semi-adaptiivinen mallinnus.

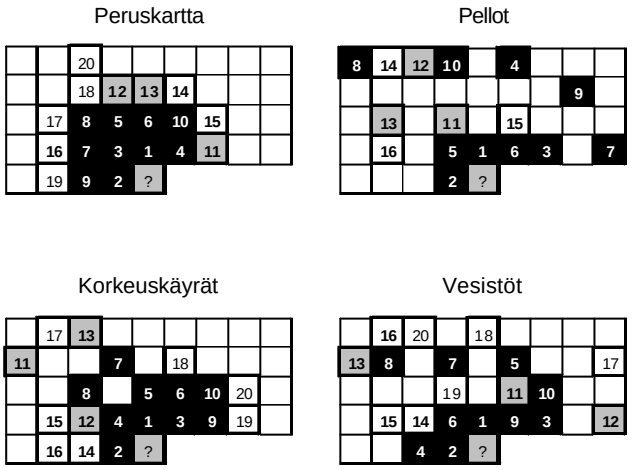
```
{ Läpikäynti 1: Analysoi kuva }
for each lohko b in kuva do
  for y:=1 to Y do
    for x:=1 to X do
      c:=Pikseli(x,y);
      k:=MuodostaKonteksti(c);
      yhteensä(k):=yhteensä(k)+1;
      if c=valkoinen then
        valkoiset(k):=valkoiset(k)+1;
      endif
    endfor
endfor

{ Muodosta malli }
for m:=1 to M do
  tila(m):=ValitseLähinTila(valkoiset(k)/yhteensä(k));
endfor

{ Läpikäynti 2: Tiivistä kuva }
for each lohko b in kuva do
  { Alusta malli}
  for m:=0 to M do
    AlustaMalli(m,tila(m,b));
  endfor
  { Tiivistä lohkot }
  for y:=1 to Y do
    for x:=1 to X do
      k:=MuodostaKonteksti(x,y);
      tila(k):=KoodaaPikseli(x,tila(k));
    endfor
  endfor
endfor
```

Kuva 6.6. Eteenpäin-adaptiivisen mallinnuksen periaate.

6.4 Optimoidut kontekstimallit

Binäärikomponenttien erilaisten semanttisten ominaisuuksien vuoksi on järkevää käyttää erilaista pikselinaapuristoa eri komponenteille [Ageenko, Kopylov ja Fränti 2001]. Pikselinaapuriston pikseleiden sijainnilla ei ole suoraa vaikutusta oppimiskustannukseen ja huolellinen pikseleiden valinta voi parantaa mallin tarkkuutta huomattavasti. Kuvassa 8.6

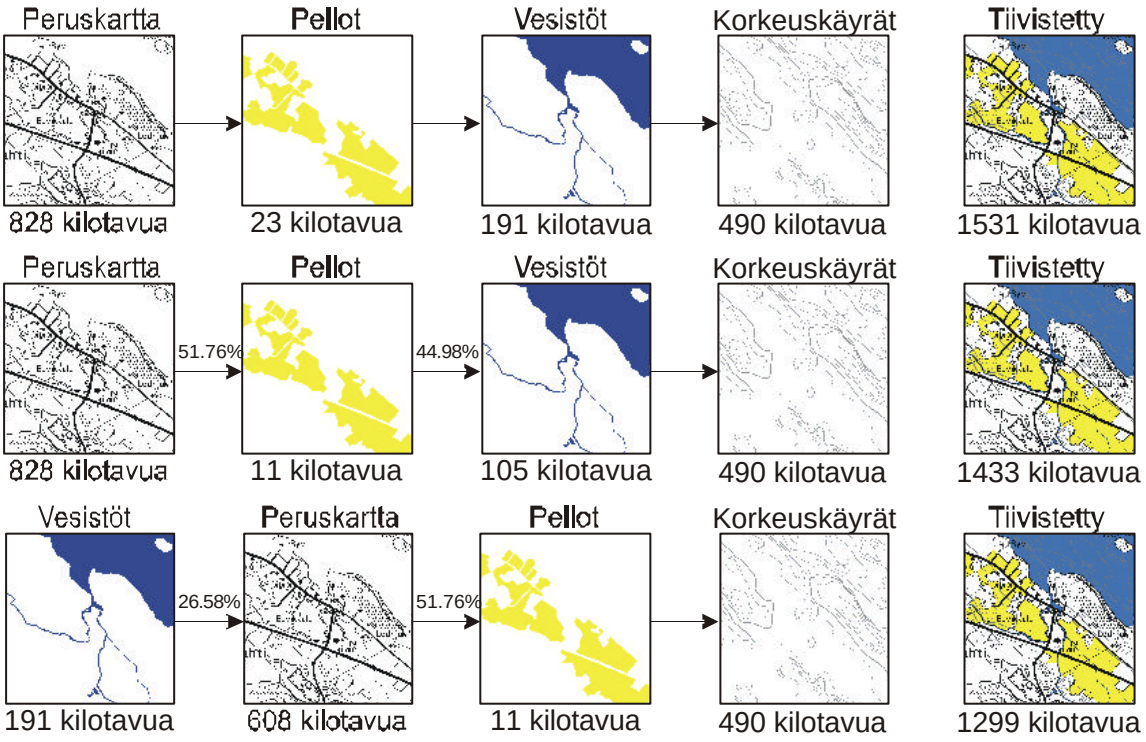
on esitelty optimoidut pikselinaapuristot eri binäärikomponenteille. Mallia laskettaessa pikselit on lisätty naapuristoon yksi kerrallaan siten, että tiivistysteho paranee jokaisen lisäyksen jälkeen mahdollisimman paljon. Kuvan 6.7 pikselinaapuristojen numerot kertovat lisäysjärjestyksen. Näitä pikselinaapuristoja käyttämällä saadaan pellot-komponentin tiivistystehoa parannettua keskimäärin 12 % [Ageenko, Kopylov ja Fränti 2001]. Muissa komponenteissa tiivistystehon parannus on vaatimattomampaa.



Kuva 6.7. Optimoidut kontekstimallit eri binäärikomponenteille [Ageenko, Kopylov ja Fränti 2001].

Karttakuvissa on myös huomattavissa eri komponenttien välisiä riippuvuuksia. Esimerkiksi vesistöjen kohdalla ei ole peltoja ja päinvastoin. Samoin peruskartassa on tyhjää vesistöjen kohdalla. Tällaisia säännönmukaisuuksia voidaan hyödyntää tiivistyksessä käytettäessä kaksitasoista kontekstimallia [Kopylov ja Fränti 2002]. Pikselinaapuristoon valitaan pikseleitä itse tiivistettävän komponentin lisäksi viitekomponentista (kuva 6.8). Karttakuvista voidaan huomata että peruskartta-, vesistöt– ja pellot-komponentit riippuvat toisistaan. Sen sijaan korkeuskäyrät on muista komponenteista riippumaton. Tiivistystehon kannalta paras tiivistysjärjestys saadaan selville kokeilemalla kaikki eri permutaatiot [Kopylov ja Fränti 2002]. Haittapuolena komponenttien riippuvuus toisistaan pakottaa purkamaan aina kaikki komponentit.

Käytettäessä tällaista kaksitasoista kontekstimallia tiivistysteho paranee noin 25% verrattuna tavalliseen 16-pikselin malliin [Kopylov ja Fränti 2002]. Koska malli optimoidaan aina tiivistysvaiheessa jokaiselle kuvalle erikseen, on tiivistäminen hyvin raskasta. Tämä ei kuitenkaan ole suuri ongelma karttakuville, koska tiivistysvaihe tapahtuu etukäteen tehokkaalla palvelimella.



Kuva 6.8. Kaksitasoinen kontekstimalli. Ylimmässä kaikki komponentit tiivistetään erikseen, kahdessa muussa käytetään kaksitasoista kontekstimallia [Kopylov ja Fränti 2002].

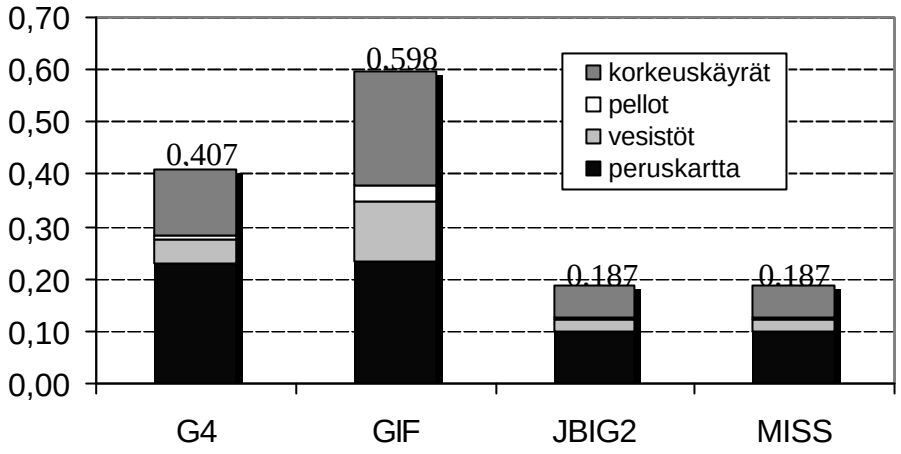
7 Tiivistysmenetelmien vertailua

Tässä luvussa vertaillaan kokeellisesti tutkielmassa aiemmin esitettyjä tiivistysmenetelmiä: GIF, G4, JBIG2 ja MISS. Näistä G4 ei sisällä omaa tiedostoformaattia, joten sillä tiivistetyt kuvat ovat TIFF-tiedoston [Salomon 2000] formaatissa. JBIG2- tuloksia laskettaessa on käytetty 10-pikselin kontekstimallia, MQ-koodaajaa ja koko kuva on tiivistetty grafiikka-alueena. MISS-tuloksille on käytetty 10-pikselin kontekstimallia, MQ-koodaajaa ja eteenpäin-adaptiivista mallia.

Testikuvina käytettiin viittä liitteessä 1 esiteltyä Maanmittauslaitoksen peruskarttalehteä numeroiltaan 124101, 201401, 263112, 431204 ja 431306. Jokaisesta kuvasta valittiin samat neljä komponenttia semanttisella jaolla: peruskartta, vesistöt, pellot ja korkeuskäyrät. Värierottelua käytettäessä tilantarve kasvaa hieman semanttisen jaon tuloksiin verrattuna [Fränti et al. 2001].

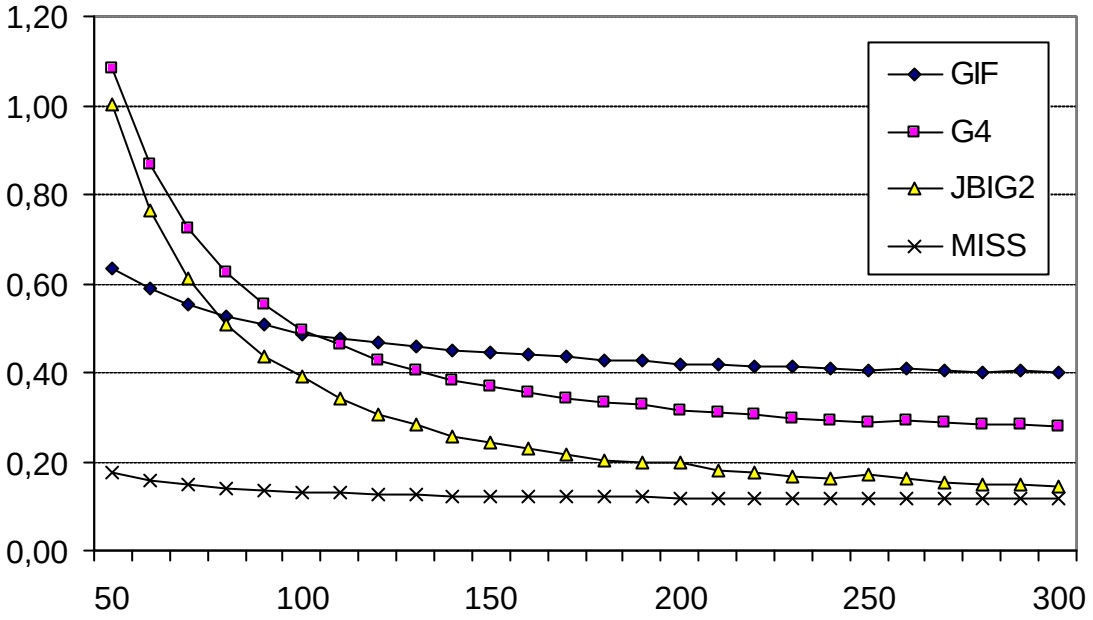
7.1 Tilantarve

Eri menetelmillä saadut tulokset tiivistettäessä koko karttakuva ilman lohkojakoa ovat kuvassa 7.1. Sitä tarkasteltaessa huomataan, että JBIG2 ja MISS ovat selkeästi tehokkaimmat menetelmät ollen yhtä tehokkaita. Tulosten samankaltaisuus selittyy erittäin samanlaisella tiivistysalgoritmilla ja ero muihin menetelmiin kontekstimallin ja aritmeettisen koodaajan yhdistelmän ylivertaisella tehokkuudella. G4 ja GIF ovat yhtä tehokkaita peruskartta- komponentille, mutta muut komponentit G4 tiivistää noin puoleen GIF:n tuloksista. Tämä selittyy peruskartta-komponentin sisältämän yksityiskohtien määrällä, G4:lle hyvin soveltuvia yksivärisiä vaakasuuntaisia pitkiä jaksoja ei juurikaan ole. Tiivistettäessä kartat GIF:n värikuvamoodissa tilantarve on hivenen suurempi kuin komponenttikohtaisten tulosten summa komponenttien vähäisen lukumäärän vuoksi. Mikäli komponentteja olisi enemmän, värikuva-GIF olisi tehokkaampi kuin erilliset binääri-GIF kuvat.



Kuva 7.1. Tiivistystulokset (bpp) eri menetelmillä ilman lohkojakoa.

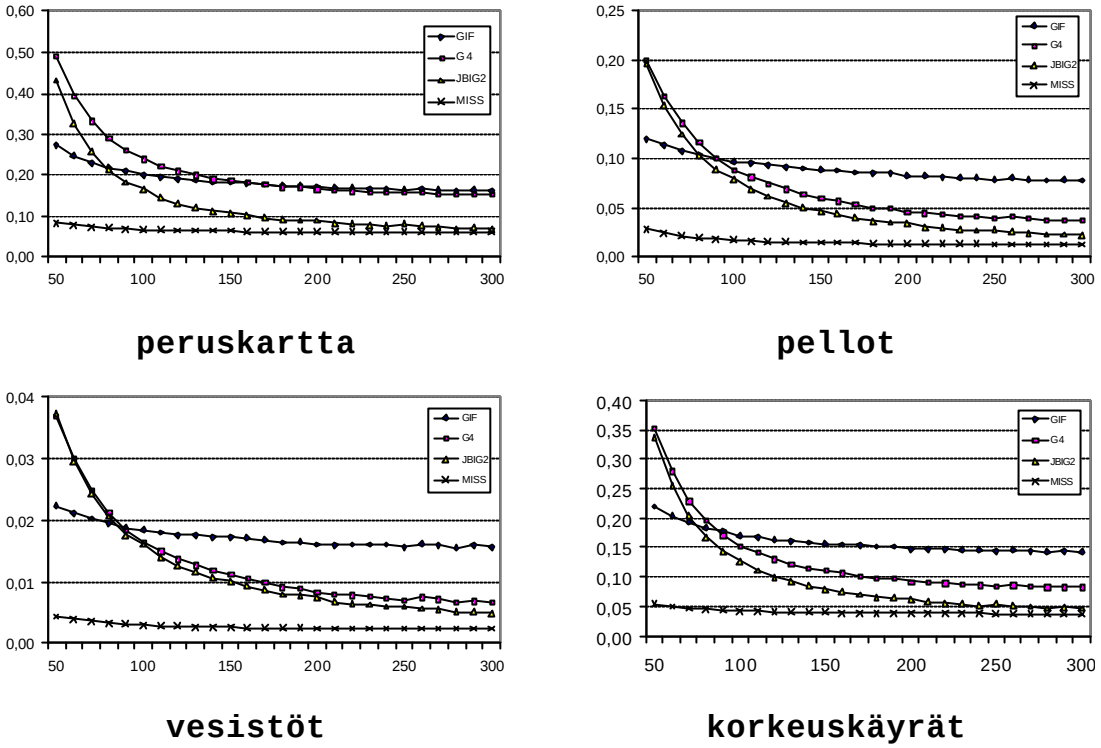
Kuten aiemmin on todettu, lohkojako sallii kartan joustavamman käytön vähemmillä resursseilla. Seuraavaksi tarkastellaan lohkojaon vaikututusta tiivistystehoon. Tämä on lähinnä arvioitu muille kuin MISS-formaatille, sillä lohkojaon todellinen käyttäminen vaatii tiedostoon samanlaisen indeksoinnin, jonka MISS sisältää. Indeksointi vaatii tilaa mutta toisaalta se vähentää muiden otsaketietojen osuutta, kuten luvussa 6.2 on todettu. Täten GIF, G4 ja JBIG2 –tulokset kärsivät otsaketietojen suuresta osuudesta verrattuna MISS-tuloksiin. Indeksoinnin välttämättömyyden vuoksi muiden menetelmien tuloksista on vähennetty tyhjät, yksiväriset lohkot. Ne voidaan koodata helposti suoraan otsaketietoihin MISS-formaatin tavoin. Kuvassa 7.2 on tarkasteltavana lohkokoot välillä 50-300. Tätä pienempien lohkojen tiivistyssuhde heikkenee huomattavasti ja suuremmat lohkot vaativat runsaasti tilaa päätelaitteessa ja ovat hitaita purkaa.



Kuva 7.2. Tiivistystulokset (bpp) eri menetelmillä lohkokoilla 50-300.

Lohkokokoa pienennettäessä huomataan, kuinka MISS:iä lukuunottamatta menetelmien tilantarve kasvaa rajusti. Pienillä lohkoilla MISS-formaatin eteenpäin-adaptiivinen mallinnus korostuu, samoin otsaketietojen osuus on huomattavasti muita menetelmiä pienempi.

Tarkasteltaessa komponenttikohtaisia tuloksia ne ovat hyvin samansuuntaisia (kuva 7.3). MISS on kaikissa tapauksissa tehokkain menetelmä ja JBIG2 toiseksi tehokkain, kun lohkokoko on suurempi kuin 100×100. Huomionarvoista on että G4 on jälleen selkeästi GIF:iä tehokkaampi peruskartta-komponenttia lukuun ottamatta. Lisäksi vesistöissä G4 on lähes yhtä tehokas kuin JBIG2, muissa komponenteissa se on selvästi heikompi. Vesistöt ovat tyypillisesti suuria yksivärisiä alueita, joissa G4:n jaksonpituuskoodaus on tehokasta.



Kuva 7.3. Tiivistystulokset (bpp) eri komponenteille lohkokoilla 50-300.

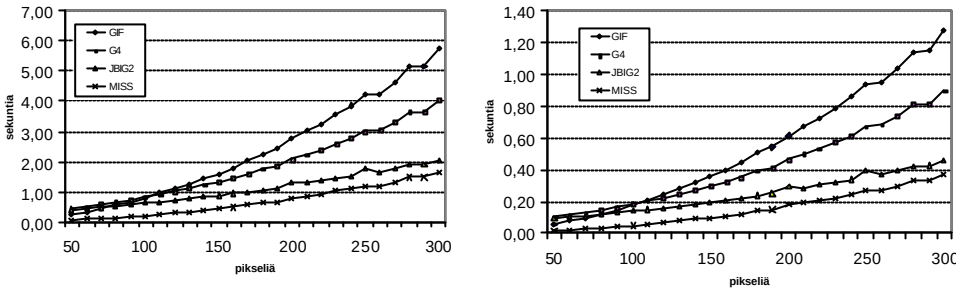
Kaikilla menetelmillä eniten yksityiskohtia sisältävä peruskartta-komponentti vie selvästi eniten tilaa ja valtaosin tyhjä vesistöt vie vähiten tilaa. Lisäksi paljon epäsäännöllisiä viivoja sisältävä korkeuskäyrät-komponentti näyttää olevan vaikea tiivistettävä kaikille menetelmille.

7.2 Hakuajat

Seuraavaksi tarkastellaan karttojen päätelaitteeseen siirtoon kuluvaa aikaa eri tiedonsiirtomenetelmillä sekä lohkojen purkuun kuluvaa aikaa eri tehoisilla suorittimilla. Nämä molemmat tehtävät tulisi pystyä suorittamaan reaaliaikaisesti, jolloin useiden sekuntien viiveet eivät ole hyväksyttävissä.

Kokonaishakuaika muodostuu kahdesta osasta: Siirtoajasta ja dekoodausajasta. Siirtoajalla tarkoitetaan lohkon palvelimelta päätelaitteeseen siirtämiseen kuluvaa aikaa. Dekoodausaika puolestaan kertoo kuinka kauan lohkon dekoodaus kestää. Lisäksi aikaa kuluu myös muihin tehtäviin, kuten kartan näytölle piirtoon ja kartan tiedostosta lukemiseen. Näitä ei oteta tarkastelussa huomioon, koska ne ovat vakioaikaisia eivätkä siten vaikuta tulosten järjestykseen.

Siirtoon kuluvaan aikaan vaikuttaa eniten käytettävä tiedonsiirtotapa. Nykyisin on tarjolla useita erilaisia langattomia menetelmiä, joista tässä kappaleessa tarkastellaan kahta: GSM-datan tiedonsiirtonopeus on 9600 bittiä sekunnissa ja General Packet Radio Systemin (GPRS) maksiminopeus on käytännössä 40200 bps. Ensin mainitussa menetelmässä itse siirtoajan lisäksi tulee myös yhteyden muodostamiseen kuluva aika, joka on useita sekunteja joka kerta yhteyttä muodostaessa. Yhteys voidaan pitää myös jatkuvasti päällä, mutta tällöin ongelmana on yhteyden pituuteen perustuva hinnoittelu. Karttoja käytettäessä tietoa siirretään vain harvoin, yleensä kun käyttäjä liikkuu puretun lohkon reunalle. GPRS-yhteyden sen sijaan voi pitää koko ajan päällä, koska hinnoittelu perustuu siirretyn tiedon määrään. Tällöin aikaa ei myöskään kulu itse yhteyden muodostamiseen. Kuvassa 7.4 on tarkasteltu yhden neljästä komponentista karttalohkon siirtoon kuluvaa keskimääräistä aikaa. Pientä lohkokokoa käytettäessä lohkoja joudutaan näkymää päivitettäessä tavallisesti siirtämään useampi kuin suuremmalla lohkokoolla.



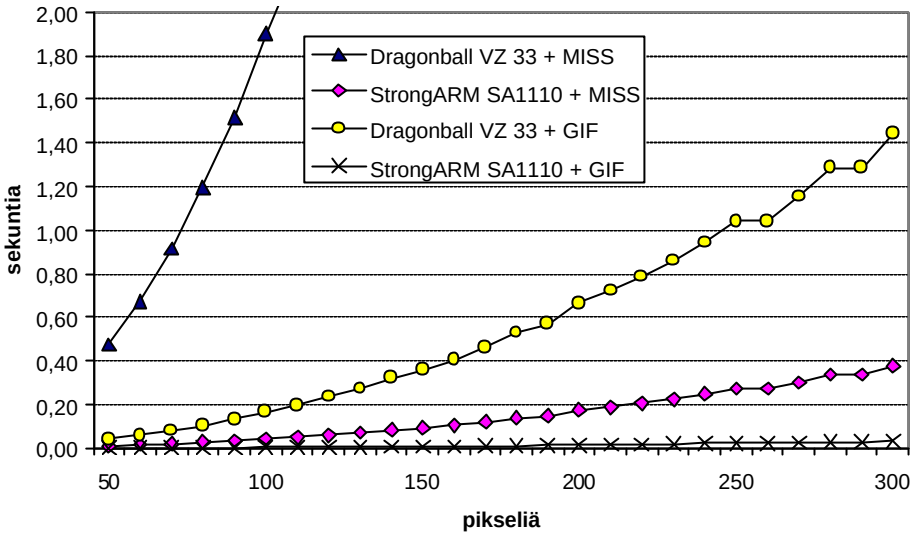
Kuva 7.4. Yhden lohkon hakuajat eri menetelmille lohkokoon funktiona eri tiedonsiirtovoilla: GSM-data (vasemmalla) sekä GPRS (oikealla).

Siirtoon kuluvia aikoja tarkasteltaessa huomataan, että hitaalla 9600 bps yhteydellä lohkokoon on oltava käytännössä 100×100 tai pienempi. Tätä suurempien lohkojen siirtoon kuluva aika yhdistettynä dekoodausaikaan on useita sekunteja, jolloin ei voida enää puhua reaaliaikaisuudesta. On kuitenkin huomioitava, että siirrettäessä suurempia lohkoja niitä tarvitsee siirtää keskimäärin vähemmän kuin pienempiä lohkoja. Esimerkiksi päivitettäessä 200 × 200 pikselin alue näytöllä, pitää siirtää vähintään 16 50 × 50 pikselin lohkoa. Lohkokoolla 100 × 100 siirrettävien lohkojen lukumäärä on tässä tapauksessa minimissään neljä.

Nopeammalla yhteydellä lohkoko voidaan tuplata noin 200×200 pikseliin. JBIG2 ja MISS mahdollistavat jopa yli 300×300 pikselin lohkot ilman siirtoajan merkittävää kasvua.

Dekoodausaikoja laskettaessa pohjana on käytetty Fränti et al.:n [2001] ilmoittamia lukuja MISS-ja GIF-menetelmille 1000 MIPS:n laskentateholla. Tässä tutkielmassa esimerkkilaitteina käytetään kahta erilaista kämmentietokonetta. Palmin tehokkain suoritin on Motorola Dragonball VZ 33, jossa on 33 megahertsiä ja laskentatehoa 5,4 MIPSiä [Motorola 2002]. Toinen yleinen kämmentietokonemalli on eri valmistajien PocketPC-laitteet, joiden Intel StrongARM SA1110-suoritin toimii 206 megahertsin nopeudella sisältäen 235 MIPSiä [Intel 2002]. Dekooausajat GIF-ja MISS-menetelmille on esitetty

kuvassa 7.5. JBIG2:n dekoodausaika on hyvin lähellä MISS-aikaa ja G4:n dekoodaus on raskaudeltaan lähellä GIF:n kertaluokkaa.



Kuva 7.5. Dekoodausajat GIF ja MISS-menetelmille eri suorittimilla lohkokoilla 50-300.

Kuvasta huomataan, että MISS:n käyttö on hyvin hidasta tehottomalla DragonBall-suorittimella. GIF:n dekoodaus sen sijaan onnistuu hitaallakin suorittimella lohkokoon ollessa alle 200×200 pikseliä. Nopeammalla StrongARM-suorittimella tiivistysmenetelmien väliset erot eivät ole yhtä merkittäviä ja kummallakin menetelmällä tiivistetyt lohkot pystytään purkamaan reaaliajassa.

Saatujen tulosten perusteella MISS-menetelmä on tiivistysteholtaan paras etenkin pienillä lohkokoilla. Lisäksi sillä tiivistettyjä lohkoja voidaan siirtää reaaliaikaisesti hitaallakin tiedonsiirtoyhteydellä, kun muilla menetelmillä tiivistettyjen lohkojen siirtoon kuluu moninkertaisesti aikaa. MISS ei kuitenkaan sovellu hyvin heikkotehoisen suorittimen sisältäviin päätelaitteisiin purkamisen ollessa monin verroin raskaampaa GIF-menetelmään verrattuna.

8 Yhteenveto

Tutkielmassa esiteltiin kuvantiivistyksen perusmenetelmiä: staattinen, semi-adaptiivinen ja adaptiivinen mallinnus, konteksimallinnus, sanakirjamenetelmät sekä aritmeettinen ja Huffman-koodaus. Lisäksi esiteltiin näihin menetelmiin pohjautuvia binäärikuvien tiivistysmenetelmiä: GIF, G4, JBIG, JBIG2 ja MISS.

Rasteri- ja vektoriformaatit mahdollistavat erilaiset tavat tallentaa digitaalisia karttoja. Vektorikartan etuja ovat mahdollisuus mittakaavan vaihtoon sekä ominaisuustiedon helppo liittäminen. Rasterikartan hyviä puolia ovat pieni tilantarve käytettäessä tiivistystä sekä helppo siirrettävyys ja yhteensopivuus. Rasterikarttojen tiivistystä voidaan edelleen parantaa käyttämällä häviöllistä tiivistystä, mutta ainakin JPEG:iä käytettäessä karttakuvista tulee tällöin usein käyttökelvottomia.

Digitaaliset kartat ovat usein saatavilla suoraan karttatietokannasta erillisinä binäärisinä komponentteina, joten ne voidaan tiivistää toisistaan riippumatta tiivistystehon kasvattamiseksi. Resurssivaatimusten vähentämiseksi jokainen komponentti voidaan lisäksi jakaa erillisiin lohkoihin. Lohkoihin on tällöin suorasaanti, jolloin vain näytöllä näkyvät lohkot voidaan purkaa muistiin. Tällä tavoin karttaa voidaan siirtää päätelaitteeseen lohkoittain esimerkiksi langattomassa puhelinverkossa.

Kokeellisessa osuudessa todettiin, kuinka MISS-menetelmä on ylivertainen muihin verrattuna karttakuvia tiivistettäessä pienellä lohkokoolla. Suurilla lohkokoilla JBIG2 on käytännössä yhtä tehokas. MISS-menetelmän tehokkuus selittyy nopean sopeutumisen mahdollistavalla eteenpäin-adaptiivisella mallinnuksella sekä otsaketietojen osuuden vähenemisenä. Purkamiseen tarvittavan laskentatehon vuoksi MISS ei kuitenkaan sovellu kaikkein heikkotehoisimpiin päätelaitteisiin. Tällöin on käytettävä jotain kevyempää menetelmää, kuten GIF:iä.

Viitteet

Ageenko E.I.: *Context-Based Compression of Binary Images*. Ph.D. Thesis, Joensuun yliopisto, Joensuu, 2000.

Ageenko E.I., Fränti P.: Forward-Adaptive Method for Compressing Large Binary Images, *Software Practice & Experience* 29(11): 943--952, 1999.

Ageenko E.I., Kopylov P., Fränti P.: On the Size and Shape of Multi-Level Context Templates for Compression of Map Images, *IEEE Int. Conf. on Image Processing* vol 3:458--461, Thessaloniki, Greece, 2001.

Akimov, A.: *Dictionary-based Compression of Map Images*, Pro Gradu-tutkielma, Joensuun yliopisto, 2001.

Arps R.B., Truong T.K.: Comparison of International Standards for Lossless Still Image Compression. *Proceedings of the IEEE* 82: 889--899, 1994.

Dye S., Buckingham S.: *Mobile Positioning*. Mobile Lifestreams, 1999.

Dynamap, Internet www-sivu: http://www.cs.joensuu.fi/pages/franti/dynamap (29.8.2002), 2001.

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

Fränti P.: *Image Compression*, Lecture notes, Joensuun yliopisto, Joensuu, 1999.

Fränti P., Ageenko E., Gröhn S.: *Storage of Multi-Component Digital Maps Using JBIG2 Image Compression Standard*. Research Report A-2001-2. Joensuun Yliopisto, Joensuu, 2001.

Fränti P., Ageenko E., Kopylov P., Gröhn S., Berger F.: *Compression of Map Images for Real-Time Applications*. Research Report A-2001-1. Joensuun yliopisto, Joensuu, 2001.

Fränti P., Ageenko E., Kopylov P., Gröhn S., Berger F.: Map Image Compression for Real-Time Applications, *Spatial Data Handling 2002 Symposium (SDH'02)*, Ottawa, Canada, July 2002.

Fränti P., Kopylov P., Ageenko E: Evaluation of Compression Methods for Digital Map Images, *IASTED Int. Conf. on Automation, Control and Information Technology (ACIT 2002)*, Novosibirsk, Russia, pp. 401--405, June 2002.

Fränti P., Kopylov P., Veis V.: Dynamic Use of Map Images in Mobile Environment, *IEEE Int. Conf. on Image Processing (ICIP'02)*, Rochester, New York,USA, September 2002.

Gellersen H.W., Thomas P.J.: *Proc. Second International Symposium on Handheld and Ubiquitous Computing (HUC'2000)*, Bristol UK, Springler-Verlag, September 25--27, 2000.

Haskell, B., Howard, P. G., LeCun, Y., Puri, A., Ostermann, J., Civanlar, M. R., Rabiner, L., Bottou, L. and Haffner, P.: Image and Video Coding – Emerging Standards and Beyond, *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8(7): 814--837, 1998.

Howard P.G., Kossentini F., Martins B., Forchammer S., Rucklidge W.J.: The Emerging JBIG2 Standard. *IEEE Trans. Circuits and Systems for Video Technology*, 8(7): 838--848, 1998.

Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. *Proc. Inst. Radio Engineers* 40(9): 1098--1101, 1952.

Intel: *Intel ® StrongARM SA-1110 Microprocessor Brief Datasheet*. Internet www-sivu, http://www.intel.com/design/strong/datashts/27824105.pdf (19.8.2002).

ITU-T: Recommendation T.6: *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus*, 1988.

JBIG. ISO/IEC International Standard 11544 *ISO/IEC/JTC1/SC29/WG9*, 1993.

JBIG Committee: JBIG2, Final Committee Draft 1999, Internet www-sivu, http://www.jpeg.org/jbigpt2.html (15.9.2002).

Kaplan E.D.: *Understanding GPS: Principles and Applications*. Artech House Telecommunications Library, March 1996.

Kopylov P., Fränti P.: Context Tree Compression of Multi-Component Map Images", *IEEE Data Compression Conference*, Snowbird, Utah, USA, 212--221, 2002.

Langdon G.G., Rissanen J.: Compression of Black-White Images With Arithmetic Coding. *IEEE Trans. Communications* 29 (6): 858--867, 1981.

Maanmittauslaitos, Internet www-sivu, http://www.nls.fi (22.9.2002).

Miano J.: *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*. Addison-Wesley, Boston, 1999.

Moffat A.: Two-Level Context Based Compression of Binary Images. *IEEE Proc. Data Compression Conference*, Snowbird, Utah, USA, 382--391, 1991.

Motorola Semiconductors: MC68VZ328 (DragonBall™ VZ) Integrated Portable System Processor Product. Internet www-sivu, http://e-www.motorola.com/brdata/PDFDB/docs/MC68VZ328P.pdf (19.8.2002).

Nevalainen S., *Vektorikarttojen Hyödyntäminen Paikkatietosovelluksissa*, Pro Gradu-tutkielma, Joensuun yliopisto, 2001.

Pennebaker W.B., Mitchell J.L.: Probability Estimation for the Q-coder. *IBM Journal of Research, Development* 32(6): 737--759, 1988.

Pennebaker W.B., Mitchell J.L.: *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

RealMap, Internet www-sivu, http://cs.joensuu.fi/pages/franti/kartta (29.8.2002), 2000.

Rissanen J., Langdon G.G.: Arithmetic Coding. *IBM Journal of Research and Development* 23(2): 149--162, 1979.

Salomon D.: *Data Compression the Complete Reference*. Springler-Verlag, New York 2000.

Shannon, C.E.: A Mathematical Theory of Communication. *Bell Systems Technical Journal*, 27, 379--423: 623--656, 1948.

Tompkins D., Kossentini F.: *Additional Extension Segments for JBIG2*. ISO/IEC JTC 1/SC 29/WG1 (ITU-T SG8), Document No 1318, July 1999.

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

Weinberger, M.J., Rissanen J., Arps R.: Application of Universal Context Modeling to Lossless Compression of Gray-Scale Images. *IEEE Trans. Image Processing*, 5(4): 575--586, 1996.

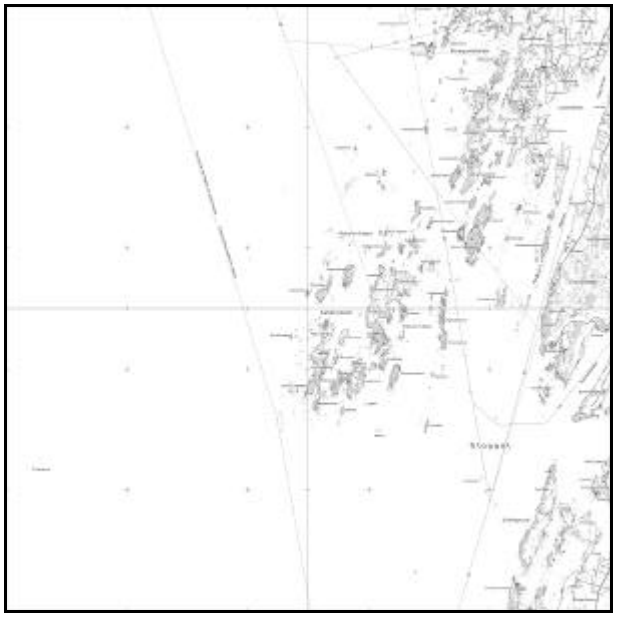
Welch T.: A Technique for High-Preformance Data Compression. *IEEE Computer* 17(6): 8--19, 1984.

Witten I.H., Moffat A., Bell T.C.: *Managing Gigabytes: Compressing and Indexing Documents and Images (2nd edition)*. Morgan Kaufmann, USA, 1999.

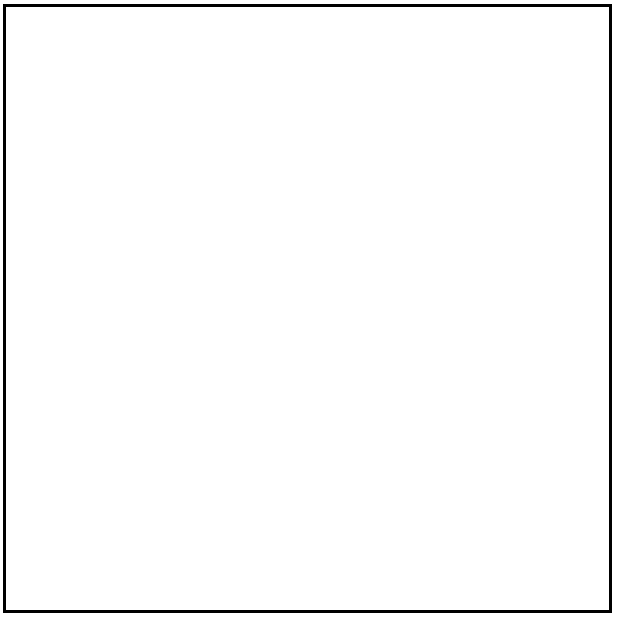
Yasuda Y.: Overview of Digital Facsimile Coding Techniques in Japan. *IEEE Proceedings* 68:830--845, July 1980.

Ziv J., Lempel A.: Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5): 530--536, 1978.

Liite: Testikuvat



124101 peruskartta

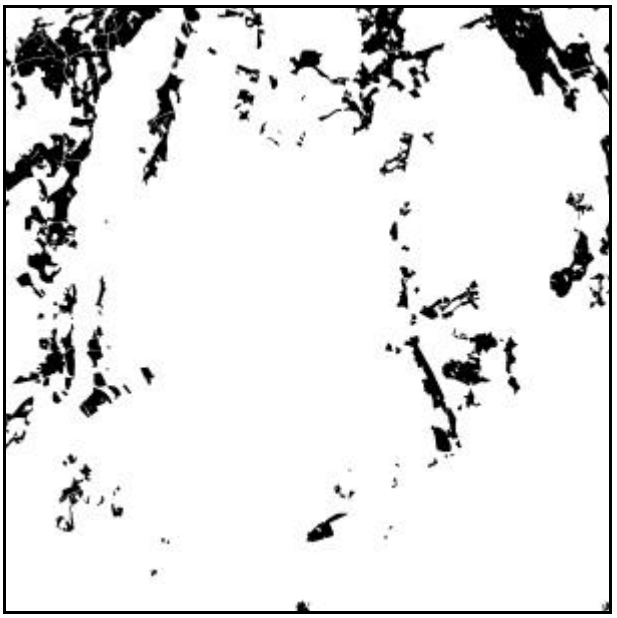


124101 pellot

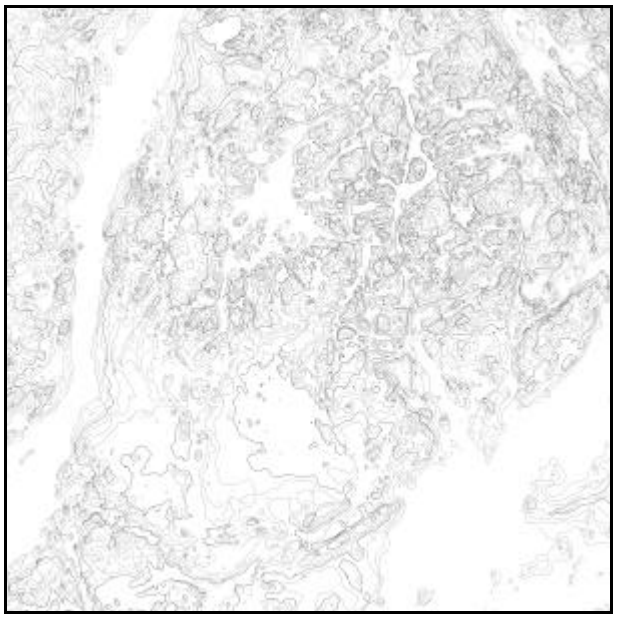
124101 korkeuskäyrät

124101 vesistöt

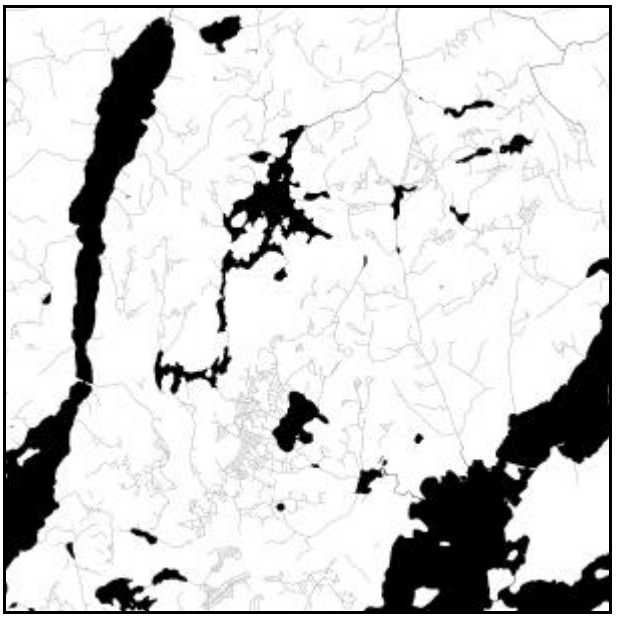
201401 peruskartta



201401 pellot



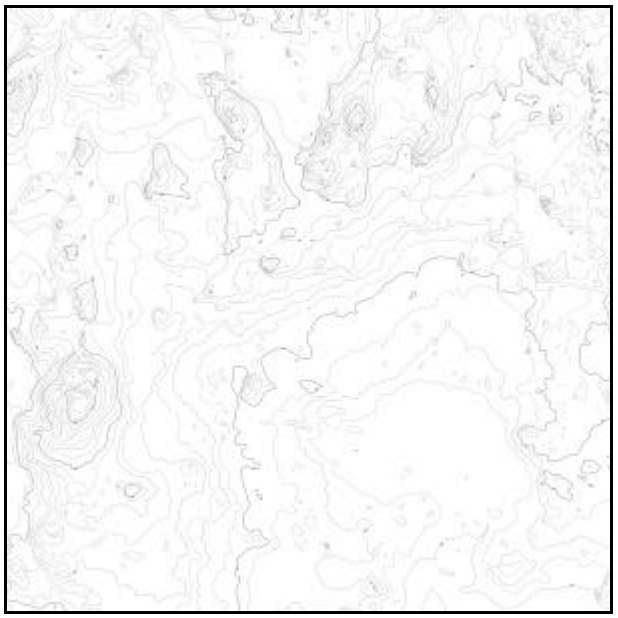
201401 korkeuskäyrät



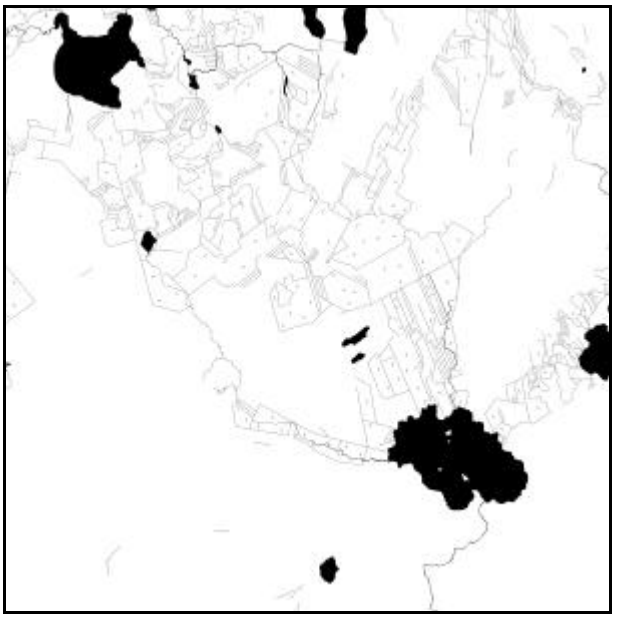
201401 vesistöt

263112 peruskartta

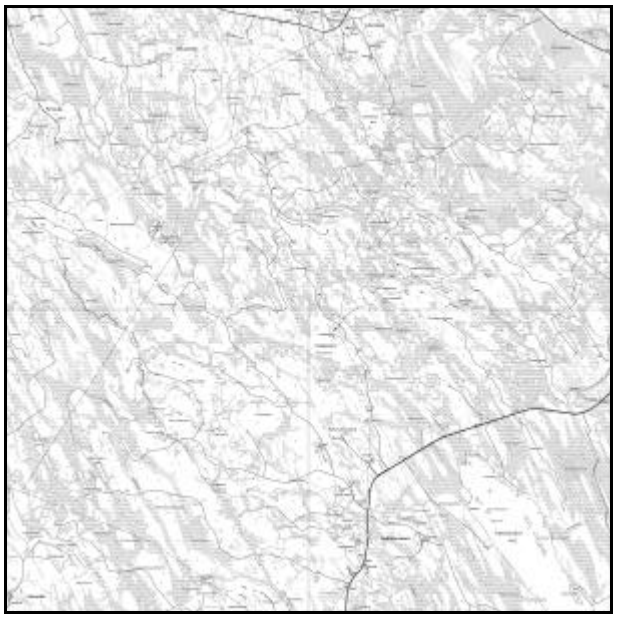
263112 pellot



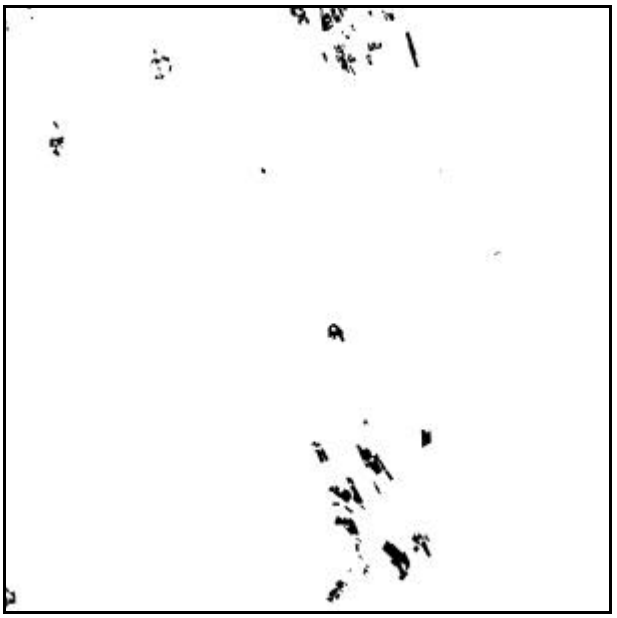
263112 korkeuskäyrät



263112 vesistöt

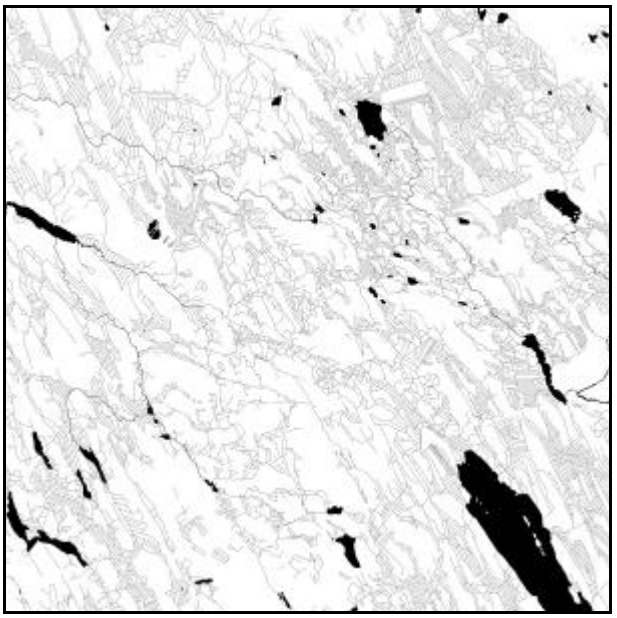


431204 peruskartta

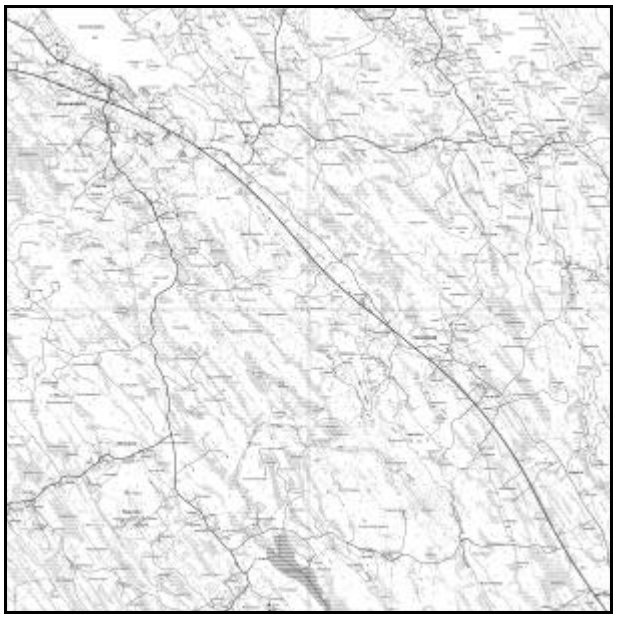


431204 pellot

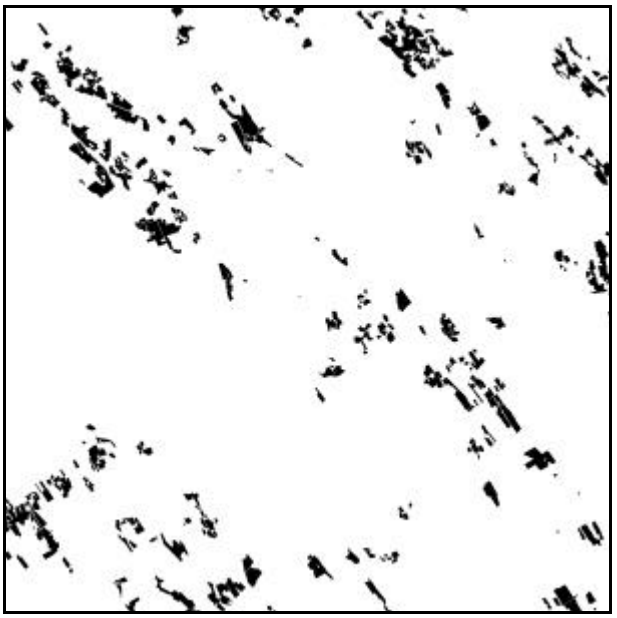
431204 korkeuskäyrät



431204 vesistöt

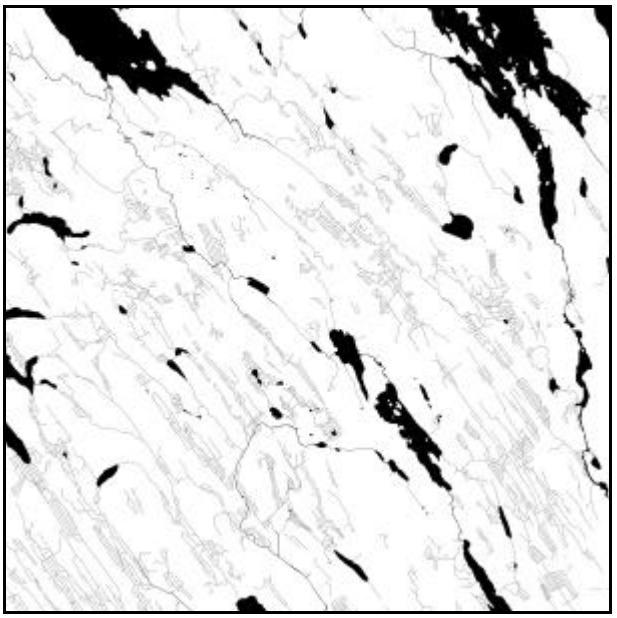


431306 peruskartta



431306 pellot

431306 korkeuskäyrät



431306 vesistöt