Päivi Toikkanen

Häviötön tiedon pakkaaminen

Tietotekniikan
pro gradu -tutkielma
26. marraskuuta 2014

Jyväskylän yliopisto

Tietotekniikan laitos

Kokkolan yliopistokeskus Chydenius

Tekijä: Päivi Toikkanen
Yhteystiedot: paivi.toikkanen@anvianet.fi
Puhelinnumero: 040-5841912
Ohjaaja: Risto T. Honkanen
Työn nimi: Häviötön tiedon pakkaaminen
Title in English: Lossless data compression
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 74
Tiivistelmä: Tämän työn aiheena on häviöttömät tiedonpakkausmenetelmät. Työssä tarkastellaan muutamia häviöttömiä tiedonpakkausmenetelmiä, joista selvitetään kunkin algoritmin toiminta ja joita vertaillaan toisiinsa. Työ on puhdas kirjallisuustutkimus, jossa on yritetty selvittää tutkittavien algoritmien paremmuus tiedon pakkaamiseen. Vertailun tekee ongelmalliseksi sopivan lähdeaineiston puute. Siten tässä työssä käsitellyn aineiston perusteella ei voida vetää suoraa johtopäätöstä menetelmien paremmuudesta, mutta työ antaa aiheen jatkotutkimuksille.
Avainsanat: algoritmi, häviötön tiedon pakkaaminen, informaatioteoria
Abstract: The subject of this thesis is lossless data compression. In this thesis a few lossless data compression methods are examined and compared with each other. The meaning of this work, which is purely a litterature study, is to find out, which of the algorithms would be most suitable for data compression. The difficulty of the comparison is due to lack of suitable litterature, whereas no conclusions about the superiority of the different methods can be drawn. However, this study gives reasons to study the subject further.
Keywords: algorithm, lossless data compression, information theory

Copyright © 2014 Päivi Toikkanen

All rights reserved.

Esipuhe

Tämä työ oli pitkissä kantimissa. Työ alkoi varsinaisesti keväällä 2008, jolloin päätin opinnäytetyöni aiheen yhdessä professori Ismo Hakalan kanssa. Työnimenä oli tuolloin *Tiedon pakkaaminen Huffmanin koodilla*. Kuten aina elämässä, asia mutkistuu kun se pitkittyy. Niin tässäkin tapauksessa.

Tutkintorakenteen muutos aiheutti sen, että alunperin yhdestä työstä tulikin kaksi erillistä työtä, kandityö ja progradu, mikä aiheutti lisää harmaita hiuksia. Pidin kandiseminaarin ennen kirjallisen työn luovuttamista. Vuodet vierivät ja kirjallinen työ jäi lojumaan mappiin Ö, kunnes sain uuden kimmokkeen jatkaa työtä syksyllä 2012, jolloin myös opinnäytetyöni lopullinen aihe muotoutui.

Haluan ensiksi kiittää työnantajaani Centria-ammattikorkeakoulu Oy:tä mahdollisuudesta opiskella tietotekniikkaa työn ohessa. Vaikka aktiivinen opiskelu loppuikin jo useita vuosia sitten, opiskelu työn ohessa erityisesti alkuvuosina oli merkittävä etu. Lämpöiset kiitokset Kokkolan yliopistokeskus Chydeniuksen informaatioteknologisen tiedekunnan henkilökunnalle kaikesta siitä avusta, jota olen pitkän matkani varrella saanut. Erityiskiitokset kuuluvat luonnollisesti työni ohjaajalle Risto Honkaselle, joka jaksoi kärsivällisesti korjailla ja kommentoida tuotostani. Viimeisenä mutta ei suinkaan vähäisimpänä haluan mainita puolisoni, jolta on löytynyt uskomatonta taitoa ja näkemystä asioihin, joihin hänellä ei ole ollut mitään aikaisempaa sidettä.

Kokkolassa, 26. päivänä marraskuuta 2014

Päivi Toikkanen

Sanasto

BWT	Häviötön tiedon pakkausmenetelmä, Burrows-Wheeler-Transformi, jonka kehittivät Michael Burrows ja David Wheeler vuonna 1994.
EVV	Nollasolmu muuttuvassa Huffman-puussa. Kirjainyhdistelmä tulee sanoista Ei Vielä Välitetty (Not Transmitted Yet).
FGK	Huffmanin koodista kehitetty mukautuva pakkausalgoritmi, jonka kehittivät Faller, Gallager ja Knuth 1970- ja 1980-luvuilla.
LZ	Yleisnimitys hakemistoihin perustuvista peruspakkausmenetelmistä, jotka kehittivät Abraham Lempel ja Jacob Ziv 1970-luvulla.
LZ77	Hakemistoihin perustuva pakkausmenetelmä, jonka kehittivät Abraham Lempel ja Jacob Ziv 1977.
LZ78	Hakemistoihin perustuva pakkausmenetelmä, jonka kehittivät Abraham Lempel ja Jacob Ziv 1978.
LZMA	Markovin LZ77-pakkausmenetelmästä kehittämä muunnos.
LZSS	Storer ja Syzmanskin LZ77-pakkausmenetelmästä kehittämä muunnos.
LZW	Terry Welchin LZ78-pakkausmenetelmästä kehittämä muunnos, johon mm. Compress-pakkausohjelma perustuu.
PCM	Pulse Code Modulation, suom. pulssikoodimodulaatio, on menetelmä, jolla äänitaajuussignaali koodataan digitaaliseen muotoon. Menetelmää käytetään digitaalisessa lankapuhelinverkossa.
PPMd	Dimirty Shkarinin PPM-pakkausalgoritmista (*Prediction of Partial Matching*) kehittämä muunnos.
PPMII	Dimirty Shkarinin PPM-pakkausalgoritmista (*Prediction of Partial Matching*) kehittämä muunnos.
SF	Häviöttömien pakkausmenetelmien klassikko, jonka kehittivät Claude Shannon ja Robert Fano 1940-luvulla.
V	FGK-algoritmin muunnos, jonka kehitti Vitter 1980-luvun lopulla.

Sisältö

Esipuhe **i**

Sanasto **ii**

1 Johdanto **1**

2 Johdatus informaatioteoriaan **2**

2.1 Informaatioteoriaan liittyvistä käsitteistä 4

2.1.1 Todennäköisyysavaruus 4

2.1.2 Tiedonsiirto 5

2.1.3 Informaatio 5

2.1.4 Kanavan kapasiteetti 11

2.2 Informaatioteoria ja tiedon pakkaaminen 13

2.2.1 Tiedonsiirtojärjestelmä 13

2.2.2 Lähdekooditeoria 14

2.2.3 Määrähäiriöteoria 15

3 Tiedon pakkaaminen **17**

3.1 Yleisimmät datatyypit 17

3.1.1 Teksti 18

3.1.2 Kuva 18

3.1.3 Video 19

3.1.4 Ääni 19

3.2 Mallinnus ja koodaus 21

3.2.1 Mitä mallinnuksella ja koodauksella tarkoitetaan? 21

3.2.2 Staattinen vs. adaptiivinen mallinnus 22

3.3 Tiedon pakkausmenetelmistä 22

3.3.1 Häviöttömät pakkausmenetelmät 24

3.3.2 Häviölliset pakkausmenetelmät 24

3.3.3 Suorituskyvyn arviointitapoja 25

4 Häviöttömiä tiedon pakkausmenetelmiä **27**
4.1 Shannon-Fano-koodaus 27
4.2 Huffman-koodaus 30
4.2.1 Muuttumaton Huffman-koodaus 31
4.2.2 Muuttuva Huffman-koodaus 36
4.3 Aritmeettinen koodaus 43
4.3.1 Muuttumaton aritmeettinen koodaus 43
4.3.2 Muuttuva aritmeettinen koodaus 48
4.4 Lempel-Ziv-koodaus 51
4.4.1 LZ77 -koodaus 52
4.4.2 LZ78-koodaus 58

5 Häviöttömien tiedon pakkausmenetelmien vertailua **60**
5.1 Staattisuus vs. adaptiivisuus 60
5.2 Tehokkuus 60
5.3 Suoritusnopeus 66
5.4 Virheiden sietokyky 67

6 Yhteenveto ja johtopäätökset **70**

Lähteet **72**

1 Johdanto

Tiedon pakkaamisen voidaan katsoa olevan informaatioteorian haara, jossa päätarkoituksena on vähentää lähetettävän tiedon määrää [14]. Ilmiöön törmää lähes päivittäin, vaikka sitä ei ehkä tietoisesti tule ajatelleeksi. Kukapa tulee miettineeksi esimerkiksi televisiota katsellessaan, että kaikki sieltä ulos tuleva tietovirta on pakattua tiedonsiirron aikana.

Miksi tietoa oikeastaan pakataan, kun pakkaamattomankin tiedon siirto ja tallentaminen onnistuvat? Tiedon pakkaamisella voidaan lisätä sekä tiedonsiirtokanavan että tallennusvälineen suorituskykyä ja vähentää siirrosta ja tallennuksesta aiheutuvia kustannuksia. Otetaan pari esimerkkiä. Jos jokin suuri kuvatiedosto pakataan puoleen sen alkuperäisestä koosta, se tarkoittaa käytännössä sitä, että tallennusvälineen suorituskyky on tuplaantunut. Ja mitä useampi tiedosto mahtuu tallennusvälineeseen, sitä vähemmän niitä tarvitaan, jolloin kustannukset pienenevät.

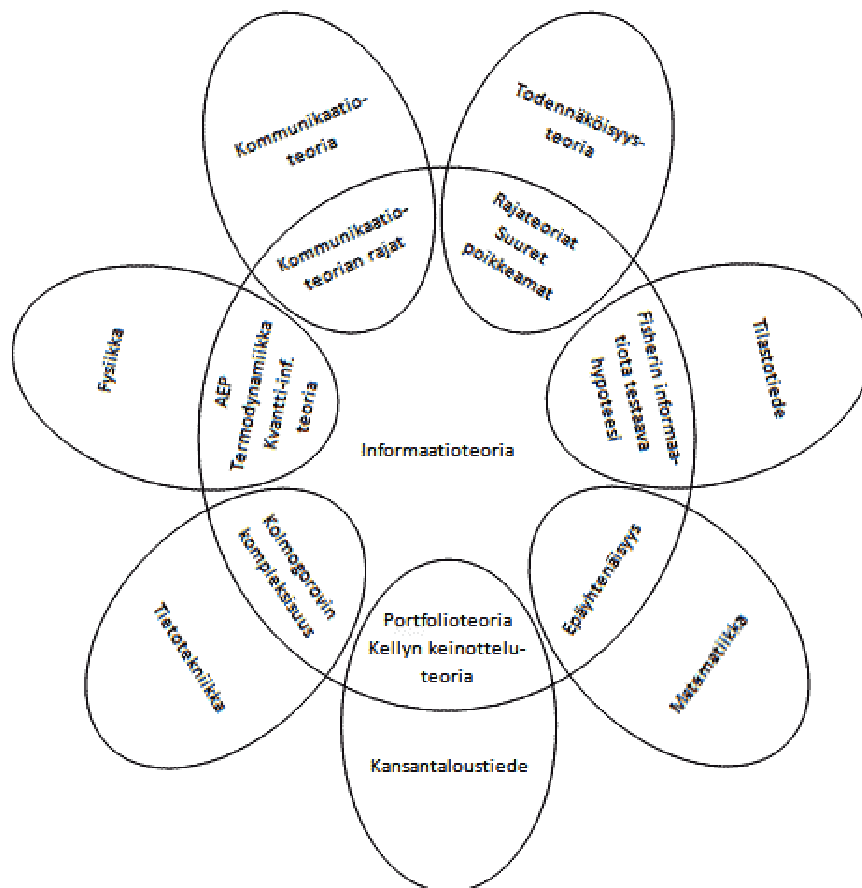
Morsen aakkoset mainitaan kirjallisuudessa esimerkkinä varhaisesta pakkaamisen muodosta 1800-luvulta [23, s. 2]. Morse havaitsi, että sähkösanomien tietyt kirjaimet esiintyivät useammin kuin toiset. Lyhentääkseen sanoman lähetysaikaa Morse merkitsi usein esiintyvät kirjaimet lyhyellä koodilla ja vastaavasti harvemmin esiintyvät kirjaimet pitkällä koodilla. Morsen ideoita pystyttiin hyödyntämään myöhemmin, kun tiedon pakkaamisen perusta luotiin 1940-luvulla.

Tämä työ on kirjallisuustutkimus häviöttömästä tiedon pakkaamisesta. Koska häviöttömiä tiedon pakkausmenetelmiä on paljon, olen valinnut tutkimukseni kohteeksi vain muutaman keskeisimmän menetelmän. Olen käyttänyt lähteinä pääasiassa eri julkaisusarjoissa ilmestyneitä tieteellisiä artikkeleita ja muutamia alan perusteoksia. Työn päätulokset liittyvät häviöttömiin tiedon pakkausmenetelmiin ja niiden keskinäiseen vertailuun.

Olen jakanut työn viiteen päälukuun. Luvussa 2 luon lyhyen katsauksen informaatioteoriaan. Luvussa 3 kerron tiedon pakkaamisesta yleensä. Luvussa 4 kuvaan muutamia keskeisiä häviöttömiä tiedon pakkausmenetelmiä. Luvussa 5 vertailen edellisessä luvussa käsiteltyjä häviöttömiä tiedon pakkausmenetelmiä toisiinsa. Luku 6 sisältää yhteenvedon ja johtopäätökset.

2 Johdatus informaatioteoriaan

Informaatioteoria (*Information theory*) on vahvasti matematiikkaan pohjautuva tieteen haara, joka tutkii tiedon mittaamista ja siirtämistä. Kirjallisuudessa esiintyy myös käsite kommunikaatioteoria (*Communication theory*), jota voitaisiin pitää informaatioteorian synonyyminä, mitä se ei kuitenkaan ole. Coverin ja Thomasin [4, s. 1] mukaan informaatioteoria on laajempi käsite, jonka vaikutus ulottuu sähkötekniikan alaan kuuluvaa kommunikaatioteoriaa laajemmalle fysiikkaan, kansantaloustieteeseen, matematiikkaan, tietotekniikkaan, tilastotieteeseen ja todennäköisyysteoriaan (kuva 2.1). Käytän tässä työssä selvyyden vuoksi pelkästään termiä informaatioteoria.

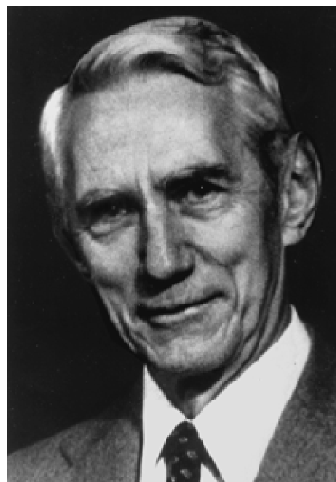


Kuva 2.1: Informaatioteorian suhde muihin tieteenaloihin [4, s. 2].

Ennen informaatioteorian kehittämistä oli jo olemassa laitteita, joilla viestejä saatiin välitettyä kahden pisteen välillä eri menetelmiä käyttäen. Verdú [32] esittelee joitakin näistä artikkelissaan *Fifty Years of Shannon Theory* seuraavaan tapaan: Lennätin (Morse, 1830-luku), puhelin (Bell, 1876), langaton lennätin (Marconi, 1887), AM Radio (1900-luvun alussa), yksitaajuusmodulaattori (Carson, 1922), televisio (1925-1927), kaukokirjoitin (1931), taajuusmodulaattori (Amstrong, 1936), monitaajuusmodulaattori (PCM) (Reeves, 1937-1939), vokooderi (Dudley, 1939) ja laajaspektriteknologia (1940-luku).

Verdún mukaan edellä esitetyissä järjestelmissä oli jo informaatioteoriaan viittaavia elementtejä, mutta niitä pidettiin toisistaan riippumattomina järjestelminä ilman mitään yhdistävää teoriaa. Joitakin aihetta sivuavia tutkimuksiakin tehtiin 20- ja 30-luvuilla. Näistä mainittakoot Nyquistin [18] ja Hartleyn [9] työt, joihin myös Shannonin käänteentekevä artikkeli pohjautuu.

Informaatioteorian kehitykseen vaikutti vahvasti amerikkalaisen monitieteilijän Claude Elwood Shannonin (kuva 2.2) 1940-luvun lopulla ilmestynyt luotettavaa tiedonsiirtoa käsitellyt kaksiosainen artikkeli *A Mathematical Theory of Communication*, jossa hän esitteli ensi kertaa tiedonsiirtojärjestelmän rakenteen ja osoitti, että virheetöntä signaalia välittävän kanavan suorituskyky on rajallinen. Lisäksi hän määritteli informaation käsitteen ja määrän sekä esitti, kuinka tieto voidaan esittää perinteisen analogisen esitystavan lisäksi myös digitaalisessa muodossa.



Kuva 2.2: Glaude Shannon, informaatioteorian isä [5].

Informaatioteoreettinen tutkimus keskittyi aluksi koodausteorioihin ja tiedon pakkaamiseen, mutta halu käytännön sovellusten kehittämiseen käänsi painopisteen luotettavan tiedonsiirron tutkimiseen [1]. Virheitä korjaavia koodeja alettiin hyödyntää 60-luvulla avaruuteen laukaistuissa luotaimissa. Muut käytännön sovellukset lisääntyivät pikku hiljaa ajan kuluessa.

2.1 Informaatioteoriaan liittyvistä käsitteistä

Tässä luvussa tarkastellaan lyhyesti joitakin informaatioteoriaan keskeisesti kuuluvia käsitteitä. Käsitteet kuvataan pääasiassa kahteen lähteeseen perustuen ellei toisin mainita [24, 25].

2.1.1 Todennäköisyysavaruus

Koska informaatioteoria hyödyntää tilastollisia käsitteitä, on aluksi syytä kerrata lyhyesti, mitä todennäköisyysavaruudella tarkoitetaan [30, s. 15-17].

Olkoon Ω alkeistapausten muodostama joukko, ns. perusjoukko. Olkoon F kokoelma Ω:n osajoukkoja, joita kutsutaan tapahtumiksi.

Määritelmä 3.1.1*. Kokoelma F perusjoukon Ω osajoukkoja on δ-algebra, jos*

1. $\Omega \in F$
2. $A \in F \Rightarrow A^c \in F$
3. $A_i \in F\ (i = 1, 2, ...) \Rightarrow \bigcup_{i=1}^{\infty} A_i \in F$

Tapahtuman A todennäköisyys, *P(A)* on reaaliluku, jonka on oltava yksikäsitteisesti määrätty, kun tapahtuma $A \in F$ on annettu.

Määritelmä 3.1.2*. Kuvaus $P : F \to R$ on todennäköisyys, jos*

1. $P(A) \geq 0\ \forall A \in F$,
2. $P(\Omega) = 1$,
3. $Jos A_i \in F(i = 1, 2, ...)\ ja$
   $A_i \bigcap A_j = \emptyset\ (i \neq j),\ niin$
   $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$

Kolmikko (Ω, F, P) on todennäköisyysavaruus, jos Ω on ei-tyhjä joukko, F on δ-algebra ja $P : F \to R$ on todennäköisyys.

2.1.2 Tiedonsiirto

Tiedonsiirto on laaja käsite, jolla tarkoitetaan yleistäen kahden eri osapuolen välistä viestintää eri menetelmiä käyttäen. Käsite kattaa kaiken inhimillisen toiminnan kirjoitetusta tekstistä ja puheesta eri taidelajeihin, mutta sitä voidaan laajentaa käsittämään myös kahden eri mekanismin välisen kommunikaation.

Shannon ja Wiever esittävät kolme eri tasoa, jotka liittyvät käsitteen laajuudesta syntyviin ongelmiin:

- Taso A (tekninen ongelma): Kuinka tarkkaan tiedonsiirrossa käytettävät symbolit voidaan lähettää?
- Taso B (semanttinen ongelma): Kuinka tarkkaan lähetetyt symbolit ilmaisevat sanoman sisältöä?
- Taso C (tehokkuusongelma): Kuinka tehokkaasti vastaanotettu viesti saa aikaan toimintaa halutulla tavalla?

Tekniset ongelmat tasolla A liittyvät symbolien siirtotarkkuuteen lähettäjältä vastaanottajalle (kirjoitettu puhe), tai jatkuvasti muuttuvaan signaaliin (puheen ja musiikin välitys television tai radion kautta), tai jatkuvasti muuttuvaan kaksiulotteiseen kuvioon (televisio) jne. Semanttiset ongelmat tasolla B liittyvät vastaanottajan kykyyn ymmärtää lähetetty viesti mahdollisimman tarkkaan lähettäjän tarkoittamalla tavalla. Kun taas tehokkuusongelmat tasolla C liittyvät siihen, kuinka onnistuneesti viestin vastaanottaja rupeaa käyttäytymään lähettäjän haluamalla tavalla.

Vaikka informaatioteoria käsittelee tiedonsiirtoa puhtaasti teknisenä ongelmana (taso A), eikä niinkään filosofisena ongelmana (tasot B–C), sillä on vaikutusta myös filosofisiin kysymyksiin. Se, mikä pätee tasolla A, pätee myös tasoihin B ja C.

2.1.3 Informaatio

Kansankielessä informaatio käsitetään viestin sisällöksi (*meaning of message*). Sähkötekniikassa sillä kuitenkin tarkoitetaan valinnanvapauden mittaa (*a measure of freedom of choice*) lähetettävää viestiä valittaessa.

Informaation mitta on bitti ja se saadaan tarjolla olevien viestien lukumäärän logaritmista yksinkertaistetulla kaavalla

$$y = \log_2 x, \tag{2.1}$$

jossa x tarkoittaa viestijoukkoa. Logaritmiksi on valittu 2-kantainen logaritmi yleisemmän 10-kantaisen Briggsin logaritmin sijaan, koska 2-kantaisella logaritmilla pystytään kuvaamaan yksinkertaisin tapaus paremmin kuin Briggsin logaritmilla. Yksinkertaisin tapaus on nimittäin sellainen, jossa viesti valitaan ainoastaan kahden mahdollisen viestin joukosta. Tällöin kaava antaa tulokseksi $log_2 2 = 1$ bittiä. Vastaavasti voidaan laskea sellaisen tapauksen sisältämä informaatio, jossa mahdollisia viestejä on enemmän, esimerkiksi 16. Tällöin $log_2 16 = 4$ bittiä.

Edellä kuvattu tilanne vaikeutuu, jos yksittäisen viestin sijaan valinta tehdään symboli kerrallaan usean mahdollisen symbolin joukosta siten, että valitut symbolit muodostavat lähetettävän viestin. Tässä tapauksessa symbolien valintaa säätelee todennäköisyys. Otetaan esimerkki englannin kielestä. Todennäköisyys sille, että symbolin *the* jälkeen valitaan jokin verbi, on lähes olematon. Sen sijaan todennäköisyys valita jokin substantiivi ko. symbolin jälkeen on hyvin suuri. Systeemiä, joka tuottaa symbolien sarjan todennäköisyyksiin perustuen, kutsutaan stokastiseksi prosessiksi (*stochastic process*). Stokastisen prosessin erikoistapausta, jossa aikaisemmin valittu symboli vaikuttaa seuraavan valinnan todennäköisyyteen, kutsutaan Markoffin prosessiksi (*Markoff process*).

Entropia

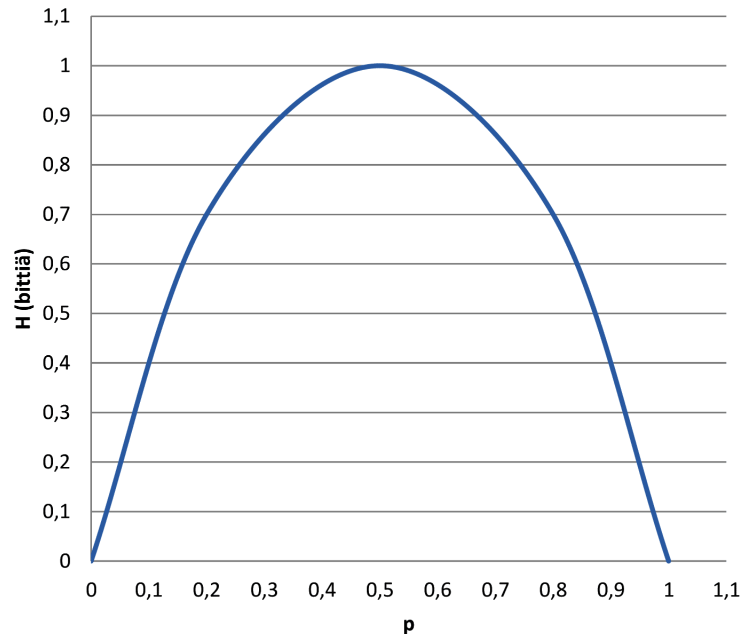
Entropia on tuttu käsite muun muassa termodynamiikasta, mutta informaatioteoriassa sillä tarkoitetaan tiedon määrää, jonka yksikkö on bitti, kuten aikaisemmin todettiin. On syytä mainita, että informaatioteorian näkökulmasta entropiassa on kyse todennäköisyysavaruudesta. Se voidaankin esittää matemaattisesti kaavalla [25]

$$H = -\sum p_i \log_2 p_i, \tag{2.2}$$

jossa p_i on symbolin i todennäköisyys.

Oletetaan, että meillä on valittavana vain kaksi mahdollista symbolia, joiden todennäköisyydet ovat p_1 ja $p_2 = 1 - p_1$ (kuva 2.3). Tässä tapauksessa H saa suurimman arvonsa eli 1, kun molemmat symbolit ovat yhtä todennäköisiä eli $p_1 = p_2 = 1/2$, mikä tarkoittaa sitä, että valinta näiden kahden symbolin välillä on täysin vapaa. Jos toinen symboleista muuttuu todennäköisemmäksi kuin toinen, esimerkiksi

$p_1 \geq p_2$ tai päinvastoin, H:n arvo pienenee. Jos toinen symboleista on hyvin todennäköinen toiseen verrattuna, esimerkiksi $p_1 \to 1$ ja $p_2 \to 0$ tai päinvastoin, $H \to 0$. Tapauksessa, jossa $p_1 = 1$ ja $p_2 = 0$ tai päinvastoin, $H = 0$ eli valinnanvapautta ei ole laisinkaan. Edellä kuvattu pätee myös tilanteeseen, jossa valittavana on useampia symboleja kerrallaan.



Kuva 2.3: Entropia kahden todennäköisyyden, p ja 1–p, tapauksessa.

Jotta entropian käsite tulisi ymmärrettävämmäksi, tarkastellaan seuraavaksi aakkostoa $\Omega = \{a_1, a_2, a_3, a_4, a_5\}$. Olkoot aakkosten todennäköisyydet vastaavasti $P = \{0.35, 0.25, 0.25, 0.1, 0.05\}$. Kaavan 2.2 mukaisesti merkkijonon entropiaksi saadaan $H = 2.079$ bittiä.

Tiedon pakkaamisen näkökulmasta entropia määrittää alarajan koodin keskimääräiselle pituudelle [21, s. 9]. Se saadaan kaavalla

$$L = \sum p_i L_i, \tag{2.3}$$

jossa L_i on i:nnen koodisanan pituus ja p_i i:nnen koodisanan esiintymistodennäköisyys.

Palataan nyt aakkoston pariin. Oletetaan, että annettu aakkosto koodataan seuraavasti: $a_1 \rightarrow 000, a_2 \rightarrow 001, a_3 \rightarrow 010, a_4 \rightarrow 100, a_5 \rightarrow 011$. Koodin keskimääräiseksi pituudeksi saadaan kaavan 2.3 mukaisesti $L = 3$ bittiä, mikä on varsin kaukana edellä lasketusta entropian arvosta. Jos aakkosto koodataankin toisin arvoilla $a_1 \rightarrow 0$, $a_2 \rightarrow 10$, $a_3 \rightarrow 110$, $a_4 \rightarrow 1110$ ja $a_5 \rightarrow 1111$, koodin keskimääräiseksi pituudeksi saadaankin nyt $L = 2.2$ bittiä, joka on lähempänä saatua entropian arvoa.

Edellä esitetystä voidaan päätellä, että koodin keskimääräisen pituuden määrittävää alarajaa kohti päästään, kun tekstissä todennäköisimmin esiintyvät aakkoset koodataan lyhyemmällä bittijonolla kuin epätodennäköisimmin esiintyvät aakkoset. Tähän perustuu häviötön tiedon pakkaaminen, johon palataan luvussa 3.

Relatiivinen entropia

Kun tiedon lähteen entropia tiedetään, saatua lukua voidaan verrata siihen entropian maksimiarvoon, joka voitaisiin saavuttaa samoissa olosuhteissa. Relatiivinen entropia H_{REL} (*relative entropy*) saadaan maksimientropian H_{MAX} ja lasketun entropian H_{CAL} välisenä erotuksesta kaavasta [25]

$$H_{REL} = H_{MAX} - H_{CAL}. \tag{2.4}$$

Yhdistetty entropia

Yhdistetty entropia (*joint entropy*) kuvaa diskreetin kanavan kokonaisentropiaa. Oletetaan, että on kaksi tapahtumaa, x ja y, joiden esiintymismahdollisuus on m x:n tapauksessa ja n y:n tapauksessa. Olkoon $p(i, j)$ todennäköisyys yhdistetylle tapahtumalle i x:n tapauksessa ja j y:n tapauksessa. Yhdistetty entropia määritellään kaavalla [25]

$$H(x, y) = -\sum_{i,j} p(i, j) \log p(i, j), \tag{2.5}$$

kun

$$H(x) = -\sum_{i,j} p(i,j) \log \sum_{j} p(i,j) \tag{2.6}$$

ja

$$H(y) = -\sum_{i,j} p(i,j) \log \sum_{i} p(i,j). \tag{2.7}$$

Voidaan osoittaa, että yhdistetty entropia on pienempi tai yhtäsuuri kuin erillisten entropioiden summa eli

$$H(x,y) \leq H(x) + H(y). \tag{2.8}$$

Yhtäsuuruus toteutuu vain siinä tapauksessa, että tapahtumat ovat itsenäisiä.

Ehdollinen entropia

Olkoon y jokin tapahtuma ja x sen arvo. Ehdollisella entropialla (*conditional entropy*) $H(y \mid x)$ tarkoitetaan y:n keskimääräistä entropiaa jokaisella x:n arvolla seuraavasti

$$H(y|x) = -\sum_{i,j} p(i,j) \log p_i(j). \tag{2.9}$$

Ehdollinen entropia kuvaa epävarmuutta y:stä tilanteessa, jossa x tunnetaan. Kun arvo $p_i(j)$ korvataan arvolla

$$p_i(j) = \frac{p(i,j)}{\sum_j p(i,j)}, \tag{2.10}$$

saadaan

$$H(y|x) = -\sum_{i,j} p(i,j) \log p(i,j) + \sum_{i,j} p(i,j) \log \sum_{j} p(i,j) = H(x,y) - H(x) \tag{2.11}$$

tai

$$H(x,y) = H(x) + H(y|x). \tag{2.12}$$

Yhdistetyn tapahtuman x,y epävarmuus (tai entropia) on siis x:n epävarmuus (entropia) plus y:n epävarmuus (entropia), kun x tunnetaan.

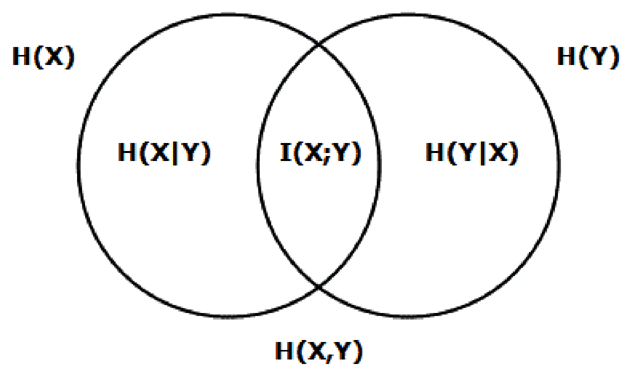
Keskinäisinformaatio

Keskinäisinformaatio $I(x;y)$ kuvaa lähteen x entropian $H(x)$ tai vastaanottajan y entropian $H(y)$ sekä ehdollisten entropioiden $H(x|y)$ tai $H(y|x)$ välistä erotusta (kuva 2.4) ja se voidaan esittää kaavoilla

$$I(x;y) = H(x) - H(x|y) \tag{2.13}$$

tai

$$I(x;y) = H(y) - H(y|x). \tag{2.14}$$



Kuva 2.4: Entropiaan liittyvien määritelmien välinen yhteys.

Redundanssi

Redundanssi tarkoittaa tiedon ylimäärää. Se saadaan ykkösen ja relatiivisen entropian erotuksena

$$R = 1 - H_{REL}. \tag{2.15}$$

Redundanssi on se osa viestiä, jota ei niinkään määrittele symbolien valinnan vapaus vaan niiden käyttöön vaikuttavat tilastolliset säännöt. Redundanssi on oikeastaan tarpeeton osa viestiä. Jos sitä ei olisi, viesti olisi silti eheä tai se voitaisiin ainakin saattaa eheäksi. Redundanssilla on merkitystä erityisesti tiedon pakkaamisessa, kun tieto pyritään saattamaan mahdollisimmän kompaktiin muotoon. Tähän palataan luvussa 3.

2.1.4 Kanavan kapasiteetti

Kanavan kapasiteettia voidaan yleisesti kuvata sen kyvyksi kuljettaa sitä, mitä tiedon lähde kulloinkin tuottaa. Jos kyseessä on yksinkertainen lähde, joka tuottaa ajaltaan samankestoisia symboleja, joiden sisältämä informaatio on s bittiä, ja jos kanava pystyy kuljettamaan esimerkiksi n symbolia sekunnissa, kanavan kapasiteetti C on ns bittiä sekunnissa.

Pitää kuitenkin ottaa huomioon seikka, että eri symbolit ovat eri pituisia. Tällöin kanavan kapasiteetti sisältää logaritmin ajaltaan tietyn kestoisista symboleista.

Kanavan kapasiteetin matemaattinen esitystapa vaihtelee riippuen siitä, minkälaisesta tiedonsiirtojärjestelmästä on kyse. Diskreetin kohinattoman kanavan kapasiteetiksi on määritelty

$$C = \lim_{T \to \infty} \frac{\log N(T)}{T}, \tag{2.16}$$

jossa $N(T)$ edustaa kestoltaan T olevia sallittuja signaaleja. Vastaavasti diskreetin kohinaisen kanavan kapasiteetiksi saadaan keskinäisinformaation $I(x;y)$ maksimiarvosta per symboli, joka voidaan siirtää kanavan yli eli

$$C = \max(H(x) - H(x|y)). \tag{2.17}$$

Jatkuvan kanavan kapasiteetti saadaan puolestaan kaavasta

$$C = W \log \frac{P + N}{N}, \tag{2.18}$$

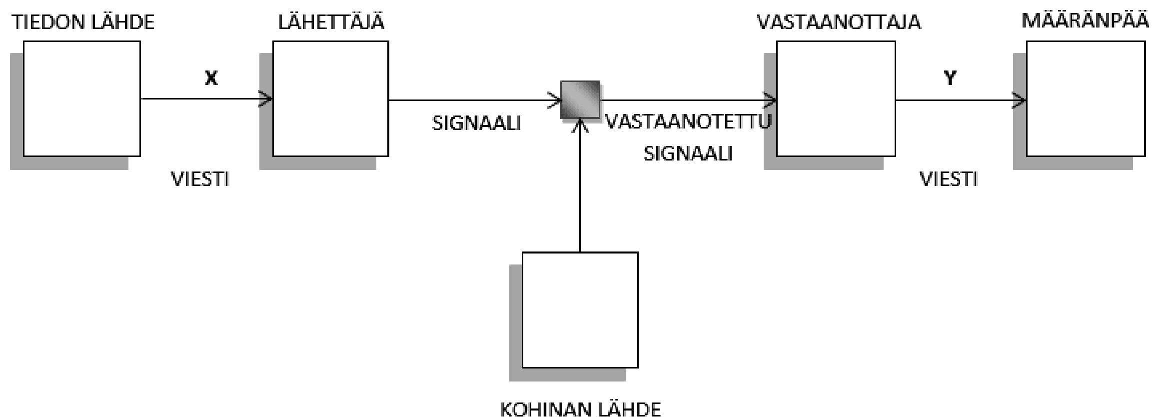
jossa W on kaistanleveys, P keskimääräinen lähettäjävoima ja N keskimääräinen häiriövoima.

2.2 Informaatioteoria ja tiedon pakkaaminen

Shannonin vuonna 1948 julkaisema kaksiosainen tutkimus *A Mathematical Theory of Communication* [25, 26] sisältää kolme merkittävää informaatioteoriaan liittyvää perustulosta: lähdekooditeorian (*Source Coding Theory*), kanavakooditeorian (*Channel Coding Theory*) ja nk. määrähäiriöteorian (*Rate Distortion Theory*), joista ensimmäiseksi ja viimeiseksi mainituilla on ollut erityisesti merkitystä tiedon pakkausmenetelmien kehittymiselle. Teoriat pohjautuvat artikkelissa esitettyyn malliin tiedonsiirtojärjestelmän rakenteesta.

2.2.1 Tiedonsiirtojärjestelmä

Tiedonsiirtojärjestelmän rakenne on esitetty kuvassa 2.5. Järjestelmässä tiedon lähde (*information source*) valitsee lähetettävän viestin useiden mahdollisten viestien joukosta. Viesti voi olla kirjoitettua tai puhuttua tekstiä, kuvia, musiikkia jne. (symboli X kuvassa 2.5). Lähettäjä (*transmitter*) muuntaa (koodaa) viestin kanavaan sopivaksi signaaliksi. Vastaanottaja (*receiver*) vastaanottaa signaalin, muuntaa (dekoodaa) sen takaisin alkuperäiseksi viestiksi (symboli Y kuvassa 2.5) ja välittää sen edelleen määränpäähän (*destination*). Kanava (*channel*) toimii signaalin välittäjänä lähettäjältä vastaanottajalle.



Kuva 2.5: Tiedonsiirtojärjestelmän rakenne Shannonin [25, s. 381] mukaan.

On hyvin epätavallista, että signaali pääsisi kulkemaan kanavassa täysin häiriöttä. Signaaliin liittyvää häiriötä kutsutaan kohinaksi (*noise*) ja kanavaa kohinaiseksi kanavaksi (*noisy channel*). Ideaalitilanteessa signaali pysyy kanavassa täysin muut-

tumattomana, jolloin puhutaan kohinattomasta kanavasta (*noiseless channel*).

Shannon luokitteli tiedonsiirtojärjestelmät vielä karkeasti kolmeen luokkaan: diskreettiin (*discrete*), jatkuvaan (*continuous*) ja sekajärjestelmään (*mixed*). Diskreetissä järjestelmässä sekä viesti että signaali esiintyvät diskreetteinä symboleina, esimerkkinä sähkösanoma. Jatkuvassa järjestelmässä sekä viesti että signaali ovat jatkuvia funktioita, esimerkkeinä radio- ja televisiolähetykset. Sekajärjestelmässä on aineksia molemmista edellä mainituista luokista, esimerkkinä puheen PCM-lähetys (*Pulse Code Modulation*). Tämä työ liittyy diskreettiin tiedonsiirtojärjestelmään.

2.2.2 Lähdekooditeoria

Kuten edellisessä alaluvussa todettiin, lähettäjä vastaanottaa viestin ja muuntaa sen kanavaan sopivaksi signaaliksi, joka kulkee kanavassa edelleen viestin vastaanottajalle. Yksinkertainen esimerkki tästä on puhelin. Se toimii ihmisen puheen välittäjänä, joka muuntaa äänen sähkövirraksi kulkemaan puhelinkaapelia pitkin toiseen puhelimeen. Lähetetty ja vastaanotettu signaali eivät oleellisesti eroa toisistaan.

Lähettäjä voi kuitenkin suorittaa viestille edellä kuvattua paljon monimutkaisempia operaatioita, kuten esimerkiksi käyttää jotakin koodia kirjoitetun tekstin salaamiseksi numerosarjaksi, joka sitten lähetetään kanavaan viestin vastaanottajalle. Lähettäjän roolina onkin koodata viesti ja vastaanottajan purkaa se takaisin alkuperäiseksi viestiksi.

Lähdekooditeoria tarkastelee tapaa esittää tieto mahdollisimman tehokkaalla tavalla siten, että kanavan kapasiteetti ei ylittyisi. Shannon havaitsi, että informaation määrän saa pienenemään, kun koodaa yhden merkin sijasta useamman merkin kerrallaan [25]. Menetelmän keksi ajallisesti Shannonin kanssa samaan aikaan tutkijakollega Fano [7]. Menetelmä nimettiinkin myöhemmin Shannon-Fano-koodaukseksi.

Uusia koodaustapoja keksittiin Shannonin artikkelia seuranneiden kolmen vuosikymmenen aikana Shannonin ja Fanon vanavedessä. David Huffman, Fanon oppilas informaatioteorian kurssilta, keksi lopputyönään tehokkaan koodaustavan numeroille ja kirjaimille vuonna 1952 [10]. Ziv ja Lempel kehittivät 1970-luvun lopulla nimeään kantavan LZ-koodin [36, 37] sekä Rissanen ja Langdon aritmeettisen koodin [20].

2.2.3 Määrähäiriöteoria

Yleensä pyritään mahdollisimman tehokkaaseen tiedonsiirtoon siten, että kanavaan lähetetään paljon viestejä samanaikaisesti ja mahdollisimman nopeasti. Tämä johtaa väistämättä siihen, että kanavaan lähetettyjen viestien määrä ylittää kanavan kapasiteetin, mikä aiheuttaa häiriötilanteita. Jotta häiriötilanteet voitaisiin minimoida, lähettäjän pitäisi priorisoida viestit tärkeytensä perusteella, ja joko tiivistää tai poistaa merkityksetön informaatio ennen lähetystä. Skeemoja, jotka on kehitetty tähdellisen materiaalin poimimiseksi ja ylimääräisen tai merkityksettömän materiaalin poistamiseksi lähetettävästä informaatiosta, kutsutaan tiedon pakkausalgoritmeiksi. Teoreettista koulukuntaa, joka tarkastelee tiedon pakkaamista informaatioteoreettisesta näkökulmasta, kutsutaan määrähäiriöteoriaksi. [3, s. 1]

Määrähäiriöteoriassa ollaan kiinnostuttu siitä, missä olosuhteissa lähteen tuottama informaatio saadaan palautettua vastaanottajapäässä siten, että palautettu informaatio sisältäisi häiriötä mahdollisimman vähän. Kysymykseen vastasi Shannon kaksiosaisen artikkelinsa toisessa osassa esittäen, että tiedonsiirtojärjestelmä sietää häiriötä D (*distortion*), mikäli informaation määrä R (*rate*) ei ylitä kanavan kapasitettia C (Shannonin 21. teoreema) [26].

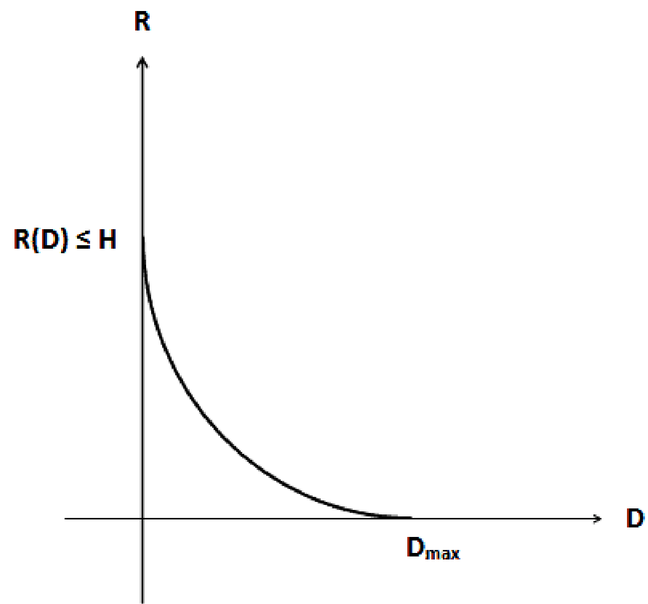
Häiriö D, jolle ehkä parempi suomennos olisi vääristymä, kuvaa sitä, miten paljon vastaanotettu informaatio poikkeaa lähetetystä informaatiosta. Määrä R on nk. määrähäiriöfunktio, joka on entropian H yleistys ja jonka yksikkö on entropian tapaan bittiä symbolia kohti. Se kirjoitetaan usein muodossa $R(D)$, mikä kuvaa lähteen R määrää tietyllä häiriön D arvolla (kuva 2.6). Shannon osoitti vuoden 1959 artikkelissaan [27], että määrän R minimi annetulla D:n arvolla saadaan kaavalla

$$R(D) = \min I(x;y), \tag{2.19}$$

jossa $I(x;y)$ tarkoittaa keskinäisinformaatiota. Kuten todettiin aikaisemmin luvussa 3, keskinäisinformaatio saadaan entropian ja ehdollisen entropian välisenä erotuksena.

Kuvasta 2.6 nähdään, että R:n arvo pienenee monotonisesti D:n kasvaessa saavuttaen arvon 0 jollakin D:n arvolla. Jos D saa arvon nolla, R vastaa lähteen entropiaa H, jolloin kanava on käytännössä täysin kohinaton ja ollaan tilanteessa, jota kuvataan edellisessä alaluvussa.

Tällä oli vaikutusta erityisesti häviölliseen tiedon pakkaamiseen liittyvien pak-



Kuva 2.6: Määrähäiriöfunktio *R(D)* häiriön *D* eri arvoilla [3, s. 7].

kausalgoritmien kehittymiseen. [23, s. 195–197].

3 Tiedon pakkaaminen

Tiedon pakkaamisen perusperiaatteena on pienentää numeerisen binäärimuodossa olevan datan kokoa. Tiedon pakkaamisessa on tärkeää osata erottaa käsitteet *data* ja *informaatio*. Data on informaation esitystapa, sen fyysinen ruumiillistuma. Sama informaatio on mahdollista esittää datan eri määrillä. Otetaan tästä esimerkkinä vaikkapa jokin kertomakirjallisuuteen kuuluva kirja. Jos sen sivumäärä on alunperin 300 sivua, kirja voidaan tiivistää lyhyempään muotoon ilman, että sen juoni häviää. Vastaavasti joku henkilö voi kertoa toiselle saman tarinan käyttäen enemmän sanoja kuin joku toinen, jolloin voidaan sanoa, että pidemmän tarinan kertonut käyttää turhia sanoja, kun taas toinen käyttää vain ne välttämättömät sanat. [22, s. 5]

Tietoa voidaan pakata, koska sen alkuperäinen esitystapa ei ole välttämättä lyhin mahdollinen, ts. se sisältää ylimääräistä ja näin ollen tarpeetonta informaatiota eli redundanssia. Tiedon pakkaaminen poistaa redundanssia, jolloin tiedon esitystapa pienenee. Pakattavissa olevalla tiedolla on jokin rakenne, joka aiheuttaa dataylimäärän. Dataa, joka ei sisällä ylimääräistä informaatiota, ei voida pakata. Täten tiedon pakkaaminen ei ole absoluuttista. Emme pysty sanomaan jonkin datatiedoston perusteella, onko se kooltaan riittävän pieni tai liian iso. Meidän täytyy etsiä siitä tarpeetonta informaatiota rakenteiden tai mallien kautta ja poistaa tarpeeton informaatio pakkaamalla se. Pakkaaminen pitäisi aina mitata vertaamalla pakatun datan kokoa alkuperäisen datan kokoon. [22, s. 5]

Tiedon pakkaamisen tulkinta tarpeettoman informaation poistamisena selittää myös sen, miksi jo kertaalleen pakattua dataa ei voi pakata uudelleen. Kun data pakataan, poistetaan siitä sen sisältämä tarpeeton informaatio rakenteen tai muotojen muodossa. Pakatulla datalla ei ole minkäänlaista rakennetta. Täten yritys pakata tällaista dataa epäonnistuu. [22, s. 6]

3.1 Yleisimmät datatyypit

Edellä puhuttiin lyhyesti tiedon ylimäärästä eli redundanssista, joka mahdollistaa tiedon pakkaamisen. Datan sisältämän redundanssin määrä vaihtelee datan tyypin mukaan, minkä takia on tärkeää valita kulloisellekin datalle sopivin pakkausmene-

telmä [22, s. 7]. Datan perustyypit ovat teksti, kuva, video ja audio eli ääni, joita tarkastelemme seuraavaksi.

3.1.1 Teksti

Teksti esiintyy tietokoneessa yksittäisinä merkkeinä binäärimuodossa. Kaikki merkit ovat kiinteämittaisia (*fixed-size*) syystä, että tällaiset koodit on helppo tallentaa muistiin, siirtää ja muokata. Aikaisemmin ASCII-koodi oli standardi merkkien koodaustapa tietokoneissa. ASCII-järjestelmässä jokainen merkki esitetään 7-bittisenä koodina, kaikenkaikkiaan niitä voi olla $2^7 = 128$. Uudemmassa Unicode-järjestelmässä merkit esitetään 16-bittisenä koodina, joita voi olla yhteensä $2^{16} = 65536$ tai jopa enemmän. [22, s. 7]

Tietyt kirjaimet esiintyvät tekstissä useammin kuin toiset, ja kiinteämittaisten koodien käyttö luo tekstiin rakennetta ja samalla informaatioylimäärää. Otetaan esimerkki englannin kielestä, jossa kirjaimet E, T ja A ovat yleisiä, kun taas kirjaimet J, Z ja Q ovat harvinaisempia. Täten tekstin voi koodata vaihtuvamittaisilla (*variable-length*) koodeilla siten, että usein esiintyvät kirjaimet koodataan pienemmällä bittimäärällä kuin harvemmin esiintyvät kirjaimet. [22, s. 7-8]

3.1.2 Kuva

Digitaalinen kuva koostuu suorakaiteen muotoon järjestäytyneistä kuvapisteistä eli pikseleistä, joilla on oma väriarvonsa ja jotka ovat toisistaan riippumattomia. Kun kuvapisteet ovat riittävän pieniä ja niitä on tiheästi vierekkäin, yksittäiset kuvapisteet eivät erotu, vaan ne muodostavat selväpiirteisen kuvan. Kuvan *koko* määritellään kuvapisteiden lukumääränä vaaka- ja pystysuunnassa. Mitä tiheämmin kuvapisteet ovat järjestäytyneet, sitä pienempi kuva on. Jos kuvaa suurennetaan, yksittäiset kuvapisteet suurenevat, jolloin kuvasta tulee epätarkka. Toinen kuvan tärkeä ominaisuus on sen *resoluutio*, jolla tarkoitetaan kuvapisteiden määrää tuumalla ja jonka yksikkö on ppi (*pixels per inch*) [11, s. 84-86].

Tiedon pakkaamisen näkökulmasta kuvapisteiden väri on tallentunut tietokoneelle kiinteämittaisena koodina. Kuvapisteen vieressä olevilla kuvapisteillä on taipumus samanlaisiin kiinteämittaisiin koodeihin, ts. kuvapisteet korreloivat keskenään. Keskenään korreloivat kuvapisteet luovat rakennetta, jolloin myös siihen sisältyy redundanssia [22, s. 8].

Digitaalisissa kuvissa esiintyvää redundanssia kutsutaan kuvapisteiden välisek-

si redundanssiksi (*interpixel redundancy*). Kuva voidaan tällöin pakata esimerkiksi vähentämällä viereisten kuvapisteiden määrää. Kuvapisteillä on samanlaisia värejä, joiden erot ovat pieniä, joten ne voidaan koodata vähemmällä bittimäärillä. Kuvilla on usein kuvapisteiden välisen redundanssin lisäksi koodausredundanssia ja psykovisuaalista redundanssia, joita molempia voidaan hyödyntää myös pakkaamisessa. Koodausredundanssi (*coding redundancy*) liittyy värien jakautumiseen. Kun tarkastellaan digitaalista kuvaa voidaan havaita, että tietyt värisävyt ovat yleisempiä kuin toiset. Värien jakaantuminen on tällöin epäyhdenmukainen. Jokainen kuvapiste koodataan vaihtuvamittaisella koodilla siten, että yleisimmin esiintyvät värit koodataan vähemmällä bittimäärällä kuin harvemmin esiintyvät värit. Psykovisuaalinen redundanssi (*psychovisual redundancy*) liittyy puolestaan ihmissilmän ominaisuuksiin. Silmä on herkkä valolle ja voi usein havaita vain muutamia valokvantteja. Silmän herkkyys valolle riippuu kuitenkin valon tyypistä. Silmä on erityisen herkkä valon kirkkaudelle, mutta vähemmän herkkä valon sisältämälle värille. Toisin sanoen, kuva voidaan pakata, jos jokainen kuvapiste ilmaistaan valon kirkkauden ja värivaihtelun avulla jälkimmäistä kovasti kvantisoiden [22, s. 8].

3.1.3 Video

Video koostuu digitoidusta liikkuvasta kuvasta ja/tai äänestä. Liikkuvan kuvan vaikutelma syntyy, kun yksittäisiä kuvia esitetään peräkkäin riittävällä nopeudella. Kun kuvien esitysnopeus on vähintään 17 kuvaa sekunnissa, silmän verkkokalvolle muodostuu vaikutelma liikkuvasta kuvasta. Kuvien esitysnopeus vaihtelee tekniikan mukaan, videoissa se on 25 kuvaa sekunnissa [11, s. 199].

Video sisältää kahdentyyppistä redundanssia: kuvan eli kehyksen sisällä olevaa redundanssia (*intraframe redundancy*), jossa jokaisen kehyksen sisältämät kuvapisteet korreloivat keskenään, ja kehysten välistä redundanssia (*interframe redundancy*), jossa vierekkäiset kehykset näyttävät olevan samanlaisia. Ensimmäistä redundanssityyppiä voidaan vähentää samoilla tavoilla, joista oli puhe kuvan pakkaamisen yhteydessä, kun taas jälkimmäiseen redundanssityyppiin pätee menetelmät, jotka vertaavat virekkäisiä kehyksiä keskenään ja koodaavat niiden välisen eron. [22, s. 8]

3.1.4 Ääni

Ääni on ilmassa etenevää pitkittäistä aaltoliikettä, jonka aiheuttaa ilmanpaineen vaihtelu jonkin ulkoisen ärsykkeen, esimerkiksi pilliin puhaltamisen tmv. seurauk-

sena. Kun aalto saavuttaa korvan tärykalvon, se värähtelee ääniaaltojen mukaan, jolloin ääniaistimus syntyy. Äänen värähtelytapa riippuu sen taajuudesta, jonka yksikkö on hertsi (Hz). Äänen taajuus ilmaisee, kuinka monta kertaa ääni värähtelee sekunnissa. Ihmiskorva pystyy erottamaan 20–20 000 Hz:n alueella olevat taajuudet [11, s. 248].

Tietokoneessa ääni on aina digitaalisessa muodossa eli ääniaallot muunnetaan numerosarjaksi, joka koostuu nollista ja ykkösistä. Muunnoksen tekee digitaalilaitteissa oleva A/D-muunnin (*Analog Digital Converter*), joka ottaa äänestä näytteitä ja antaa niille lukuarvon. Lukuarvo puolestaan määräytyy näytteenottohetkellä äänen voimakkuuden eli amplitudin perusteella, mitä kutsutaan äänen resoluutioksi. Näytteenottotaajuus määrittelee sen, kuinka monta ääninäytettä otetaan sekunnissa. Jos näytteenottotaajuus on 1 kHz, näytteitä otetaan 1000 kpl, jos näytteenottotaajuus on 22,05 kHz, näytteitä otetaan vastaavasti 22 050 kpl. [11, s. 261-262].

Mitä tiheämmin näytteitä otetaan, sitä paremmin ääni muuntuu digitaaliseen muotoon. Nyquist [18] totesi, että optimaalisen näytteenottotaajuuden pitäisi olla puolet äänen maksimitaajuudesta. Jos esimerkiksi korkein ihmiskorvan havaitsema taajuus on 20 kHz, näytteenottotajuuden tulisi olla vähintään 44 kHz, jotta kuuloalueen korkeimmat taajuudet saadaan tallennettua.

Tiedon pakkaamisen näkökulmasta äänitiedosto koostuu peräkkäisistä ääninäytteistä, jotka korreloivat keskenään täten synnyttäen redundanssia. Äänitiedoston voi pakata vähentämällä jokainen ääninäyte edeltäjästään ja korvaamalla ääninäytteiden välinen ero sopivalla vaihtuvamittaisella koodilla. [22, s. 9-10]

3.2 Mallinnus ja koodaus

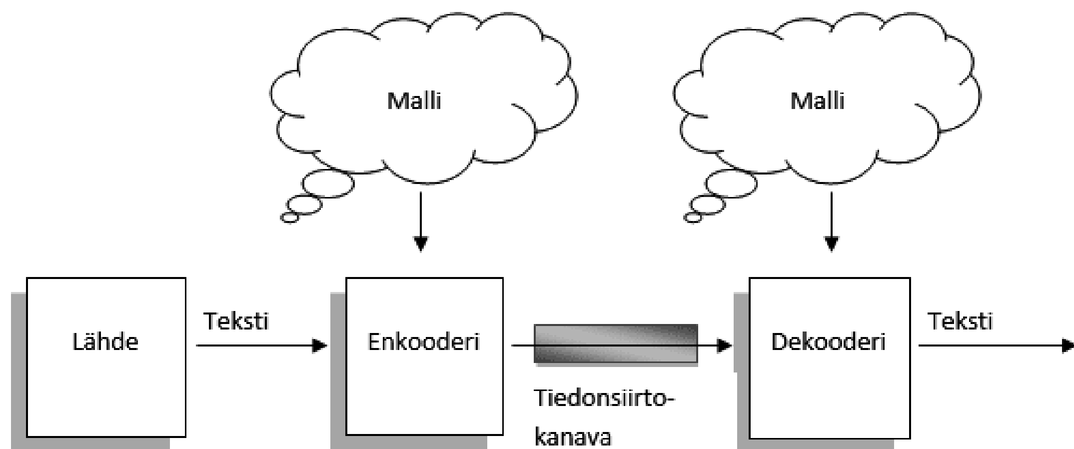
Seuraavaksi tarkastellaan tiedon pakkaamisen perusperiaatteita pääasiassa kolmeen lähteeseen perustuen [2, 16, 23].

3.2.1 Mitä mallinnuksella ja koodauksella tarkoitetaan?

Tiedon pakkaamisessa on tärkeää valita oikea menetelmä kulloistakin dataa varten. Jotta enkooderi osaisi koodata tietoa mahdollisimman tehokkaasti, sen pitäisi tuntea syöte ennakolta. Ennakoiva tieto syötteestä perustuu malliin.

Malli tarkoittaa tiedon ja sääntöjen kokoelmaa, joita käytetään syötteinä käytettävien symbolien tuottamiseksi ja tietyn koodaustavan valitsemiseksi pakkausta varten. Pakkausohjelma hyödyntää mallia laskiessaan symbolien todennäköisyyksiä ja enkooderi puolestaan hyödyntää mallia sopivan koodin tuottamiseksi symboleille niiden todennäköisyyksiin perustuen.

Mallinnus (*modeling*) ja koodaus (*coding*) ovat itse asiassa kaksi täysin eri asiaa. Termiä koodaus käytetään usein virheellisesti viittamaan koko tiedon pakkausprosessiin, vaikka koodaus on vain yksi osa sitä. Kun puhutaan esimerkiksi Huffman-koodauksesta, sillä tarkoitetaan pelkästään tiedon pakkausmenetelmää, vaikka se itse asiassa on koodausmenetelmä, jota käytetään yhdessä mallin kanssa tiedon pakkaamisessa.



Kuva 3.1: Mallin käyttö tiedon pakkaamisessa [2, s. 12].

Kuva 3.1 esittää mallin käytön perusidean, kun syötteenä käytetään tekstiä. Enkooderilla ja dekooderilla on käytössään samanlainen malli. Enkooderi pakkaa merkkijonon saamansa mallin mukaisesti ja dekooderi puolestaan purkaa vastaanottamansa koodin oikein saman mallin mukaisesti. Hyvän suorituskyvyn aikaansaamiseksi on tärkeää, että malli on juuri se oikea kyseessä olevalle syötteelle ja että enkooderilla ja dekooderilla on käytössä sama malli.

3.2.2 Staattinen vs. adaptiivinen mallinnus

On olemassa kolme eri mallinnustapaa, jolla sekä enkooderi että dekooderi voivat hyödyntää samaa mallia: *staattinen*, *semiadaptiivinen* ja *adaptiivinen* mallinnus. Staattisessa mallinnuksessa sekä enkooderi että dekooderi käyttävät koko prosessin ajan samaa sovittua mallia datan vaihteluista huolimatta. Adaptiivisessa mallinnuksessa malli mukautuu datan vaihteluihin. Semiadaptiivinen mallinnus on näiden kahden ääripään välimuoto.

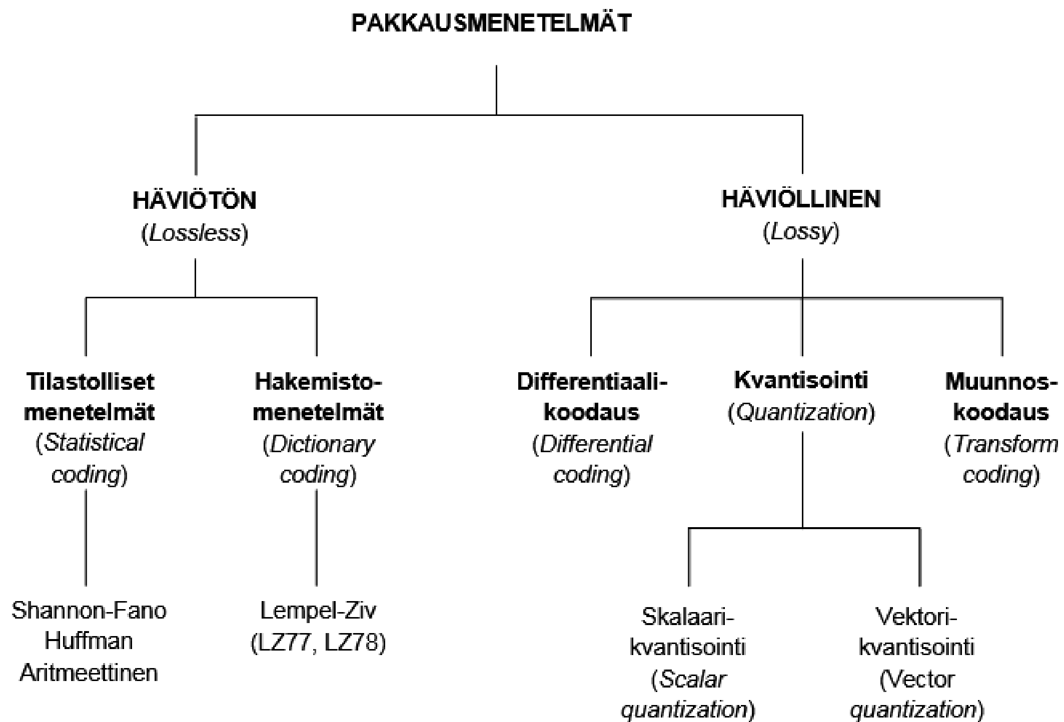
Otetaan yksinkertainen esimerkki asian havainnollistamiseksi. Oletetaan, että jokin merkkijono sisältää monta kappaletta samaa kirjainta, vaikkapa kirjaimen "a". Staattisessa mallinnuksessa kyseiselle kirjaimelle lasketaan jokin esiintymistodennäköisyys, joka on sama kaikille merkkijonon a-kirjaimille. Kyseinen merkkijono koodataan saman mallin mukaan ja tulokseksi saadaan tietty määrä bittejä. Adaptiivinen mallinnus eroaa staattisesta mallinnuksesta siinä, että se huomioi merkkijonossa aikaisemmin esiintyneet a-kirjaimet. Sitä mukaa, kun a-kirjaimia luetaan, niiden todennäköisyydet kasvavat. Kun nyt tällainen merkkijono koodataan, tulokseksi saadaan koodi pienemmällä bittimäärällä ja staattista mallia parempi pakkaustulos.

3.3 Tiedon pakkausmenetelmistä

Tiedon pakkaustekniikat voidaan jakaa kahteen pääryhmään: häviöttömiin (*lossless*) ja häviöllisiin (*lossy*) menetelmiin (kuva 3.2). Häviöllisellä menetelmällä saavutetaan suuri pakkaustehokkuus tietohävikin kustannuksella, koska pakattua tietoa ei pystytä palauttamaan täysin alkuperäiseen tilaansa. Tämä johtuu siitä, että tiedonsiirtokanavassa esiintyy jonkin verran häiriötä, jota joutuu pakattuun tietoon mukaan. Häviöllistä menetelmää voidaankin kutsua myös epätarkaksi (*inexact*) tai häiriölliseksi (*noisy*) pakkaustekniikaksi. Häviöllinen menetelmä osoittautuu tehokkaaksi,

jos sitä käytetään sellaisen tiedon pakkaamiseen, jossa tietohävikistä johtuva epätarkkuus ei ole ongelma. Näin on esimerkiksi graafisten kuvien tai analogisesta digitaaliseen muunnetun puheen laita.

Häviöttömillä menetelmillä pakattu tieto pystytään palauttamaan täysin alkuperäiseen tilaansa, mutta niillä ei voida saavuttaa yhtä hyvää pakkaustulosta kuin häviöllisillä menetelmillä. Häviöttömien menetelmien kohdalla oletetaan, että tiedonsiirtokanavassa ei esiinny häiriötä. Häviötöntä menetelmää onkin kutsuttu myös termeillä tarkka (*exact*) tai häiriötön (*noiseless*) pakkaustekniikka. Häviöttömiä menetelmiä käytetään sellaisen tiedon pakkaamiseen, joka ei siedä minkäänlaista tietohävikkiä. Tyypillinen esimerkki tästä on teksti. Tämä työ keskittyy näistä kahdesta mainitusta pakkaustekniikasta häviöttömiin menetelmiin.



Kuva 3.2: Pakkausmenetelmien luokittelu [14, 21, 23].

3.3.1 Häviöttömät pakkausmenetelmät

Häviöttömät pakkausmenetelmät voidaan jakaa kahteen pääryhmään: tilastollisiin ja hakemistomenetelmiin. Tilastollisissa menetelmissä pakkaus perustuu symbolien todennäköisyyksiin. Symbolit koodataan vaihtuvamittaisin koodein siten, että usein esiintyvät symbolit koodataan pienemmällä bittimäärällä kuin harvoin esiintyvät symbolit, jolloin redundanssi on minimissään. Tilastollisista menetelmistä on kirjallisuudessa käytetty myös termiä minimiredundanssikoodaus (*Minimum Redundancy Coding*), esim. [16]. Tässä työssä käsitellään tilastollisista menetelmistä Shannon-Fano-koodausta, Huffman-koodausta sekä aritmeettista koodausta.

Hakemistomenetelmien periaate poikkeaa jonkin verran tilastollisista menetelmistä. Hakemistomenetelmissä pakkaus perustuu niin kutsuttuihin hakemistoihin. Dataa luetaan ensin hakemistoon. Jos jokin symboli esiintyy jo hakemistossa, symboli saa arvokseen sitä hakemistossa kuvaavan osoittimen tai indeksin koodin sijaan. Mitä pidempi on vastaavuus, sitä parempi on pakkaustulos. Ehkä kuuluisin hakemistomenetelmistä lienee Abraham Lempelin ja Jacob Zivin koodausmenetelmä, LZ-koodaus, jota käsitellään myös tässä työssä.

3.3.2 Häviölliset pakkausmenetelmät

Häviölliset pakkausmenetelmät voidaan jakaa kolmeen pääryhmään: differentiaalikoodaukseen, kvantisointiin sekä muunnoskoodaukseen.

Differentiaalikoodaus perustuu arvojoukon arvojen välisen eron koodaukseen. Menetelmää käytetään erityisesti puheen ja kuvan pakkaamiseen.

Kvantisoinnissa arvojoukon arvot esitetään erillisinä arvoina. Reaaliluvut pyöristetään esimerkiksi lähimpään kokonaislukuun. Isot luvut muuntuvat puolestaan pieniksi luvuiksi. Analoginen signaali muuntuu kokonaisluvuiksi A/D- eli analogia-digitaali-muunnoksessa, jota käytetään puheen pakkausmenenetelmissä. Jos tarkasteltavat arvot ovat numeroita, kyseessä on skalaarikvantisointi, jos vektoreita, kyseessä on vektorikvantisointi.

Muunnoskoodaus perustuu käsiteltävälle datalle tehtävään matemaattiseen, yleensä lineaariseen, muunnokseen. Muunnoskoodausta käytetään tyypillisesti kuvien pakkaamiseen. Tunnettuja muunnoskoodausmenetelmiä ovat mm. diskreetti Karhunen-Loéve-muunnos (*Karhunen-Loéve transform, KLT*), diskreetit kosini- (*discrete cosine transform, DCT*) ja sinimuunnokset (*discrete sine transform, DST*) sekä diskreetti Walsh-Hadamark-muunnos (*discrete Walsh-Hadamark transform, DWHT*).

3.3.3 Suorituskyvyn arviointitapoja

Pakkausalgoritmia voidaan arvioida monin eri tavoin. Arvioinnin kohteena voi olla algoritmin suhteellinen kompleksisuus, algoritmin suorittamiseen vaadittava muistin määrä, algoritmin suoritusnopeus tietyssä laitteessa, tiivistymisen määrä ja kuinka hyvin pakattu tiedosto muistuttaa alkuperäistä pakkaamatonta tiedostoa [23, s. 5]. Seuraavassa keskitytään pelkästään kahteen viimeksi mainittuun.

Eräs tapa pakkauksen onnistumisen arvioimiseksi on laskea pakkaussuhde (*compression ratio*), joka määritellään seuraavasti [22, s. 16]:

$$CR = \frac{S_C}{S}, \tag{3.1}$$

jossa S_C on pakatun tiedoston koko ja S pakkaamattoman tiedoston koko.

Jos pakkaussuhde on esimerkiksi 1:2, tarkoittaa se käytännössä sitä, että tiedoston koko on pienentynyt 50 % alkuperäisestä. Jos suhde on 1 tai enemmän, pakatun tiedoston koko ylittää alkuperäisen tiedoston koon, jolloin puhutaan *negatiivisesta tiedon pakkaamisesta*. Pakkaussuhde ilmaisee myös sen, kuinka monta bittiä tarvitaan keskimäärin pakkaamaan yksi bitti alkuperäistä tiedostoa.

Pakkaussuhteen käänteinen suure on pakkaustekijä (*compression factor*), joka määritellään seuraavasti [22, s. 16]:

$$CF = \frac{S}{S_C}. \tag{3.2}$$

Lukua 1 vastaava tai suurempi arvo viittaavat tiedoston tiivistymiseen ja pienemmät arvot tiedoston laajenemiseen. Siis mitä suurempi luku, sitä parempi pakkaustulos.

Termi 100 x (1 – CR) on myös kelvollinen tapa mitata suorituskykyä [16, s. 7][22, s. 17]. Jos arvoksi saadaan esimerkiksi 60 %, tiedosto on tiivistynyt 40 % alkuperäisestä koostaan. Jos arvoksi saadaan jokin negatiivinen luku, tiedosto on kasvanut alkuperäistä kokoaan suuremmaksi.

Suorituskykyä voidaan ilmaista myös "määrällä"(*rate*), jolla tarkoitetaan yksittäistä näytettä kuvaavaa keskimääräistä bittien lukumäärää [23, s. 6]. Otetaan esimerkkinä jokin kuvatiedosto, joka on tiivistynyt 50 % alkuperäisestä koostaan. Jos alkuperäisen kuvatiedoston yksittäinen kuvapiste on 8 bittiä, pakatun kuvatiedoston vastaava kuvapiste on 4 bittiä. Voidaan siis sanoa, että määrä on 4 bittiä kuvapistettä kohti.

Häviöllisessä tiedon pakkaamisessa pakattua tietoa ei pystytä palauttamaan täysin alkuperäiseen muotoonsa, kuten aikaisemmin todettiin. Tästä syystä algoritmin suorituskykyä on arvioitava hieman eri tavalla kuin edellä on esitetty, ja yksi tällainen keino on arvioida pakatun tiedoston ja pakkaamattoman tiedoston välistä eroa, jota kutsutaan myös häiriöksi (*distortion*) ja johon viitattiin lyhyesti jo luvussa 2. Koska häviöllisiä tiedon pakkaamismenetelmiä käytetään tavallisesti video- ja audiotiedostojen pakkaamisessa ja koska pakkaamisen onnistumisen arvioinnin suorittaa viime kädessä ihmiskorva tai -silmä, joiden käyttäytymistä ei voida mallintaa matemaattisesti, pakatun tiedoston laadun arvioinnissa käytetään häiriön approksimoituja mittauksia. [23, s. 6]

Muita termejä, joita voidaan käyttää pakatun ja pakkaamattoman tiedon välisestä erosta, ovat tarkkuus (*fidelity*) ja laatu (*quality*). Sanomalla, että pakatun tiedon tarkkuus tai laatu ovat korkeita, tarkoitetaan pakatun ja pakkaamattoman tiedon välistä pientä eroa. [23, s. 6]

4 Häviöttömiä tiedon pakkausmenetelmiä

Kuten aikaisemmin on jo todettu, Claude Shannonin lähdekooditeoria loi pohjan häviöttömälle tiedon pakkaamiselle. Lähdekooditeorian taustalla vaikutti pyrkimys saada informaation määrä mahdollisimman pieneksi siten, että kanavan kapasiteetti ei ylittyisi.

Shannon ja tutkijakollega Robert Fano keksivät lähes samanaikaisesti yksinkertaisen menetelmän, jolla informaation määrän saa pienenemään. Menetelmä nimettiin myöhemmin Shannon-Fano-koodaukseksi. Tätä menetelmää tarkastellaan seuraavaksi [2, 16].

4.1 Shannon-Fano-koodaus

Shannon-Fano-koodaus etenee seuraavalla tavalla:

```
ALGORITMI 1. Shannon-Fano.

1. Listaa kaikki symbolit ja niiden todennäköisyydet.
2. Lajittele symbolit todennäköisyyksiensä perusteella laske-
   vaan järjestykseen.
3. Jaa symbolit kahteen osaan siten, että osien yhteenlasket-
   tu todennäköisyys on mahdollisimman lähellä toisiaan.
4. Ensimmäistä osaa kuvaa binääriluku 0 ja toista osaa binää-
   riluku 1. Tämä tarkoittaa sitä, että kaikkien ensimmäisen
   osan symbolien koodi alkaa binääriluvulla 0 ja toisen osan
   symbolien koodi binääriluvulla 1.
5. Sovella rekursiivisesti vaiheita 2-3 kumpaankin osaan, kun-
   nes kaikissa osissa on jäljellä vain yksi symboli.
```

Havainnollistetaan Shannon-Fano-algoritmia seuraavalla esimerkillä. Olkoon aakkosto $\Omega = \{a, e, i, o, u, !\}$ ja siitä muodostettu merkkijono *aeiou!*, jonka symbolien vastaavat todennäköisyydet ovat 0.2, 0.3, 0.1, 0.2, 0.1 ja 0.1. Symbolit järjestetään ensin todennäköisyyksiensä perusteella laskevaan järjestykseen kuvan 4.1 mukaisesti.

Symboli		Ryhmittely			Koodi
e	0.3	0.5	0.3		00
a	0.2		0.2		01
o	0.2	0.5	0.3	0.2	100
i	0.1			0.1	101
u	0.1		0.2	0.1	110
!	0.1			0.1	111

Kuva 4.1: Shannon-Fano-koodauksen periaate [2, s. 104].

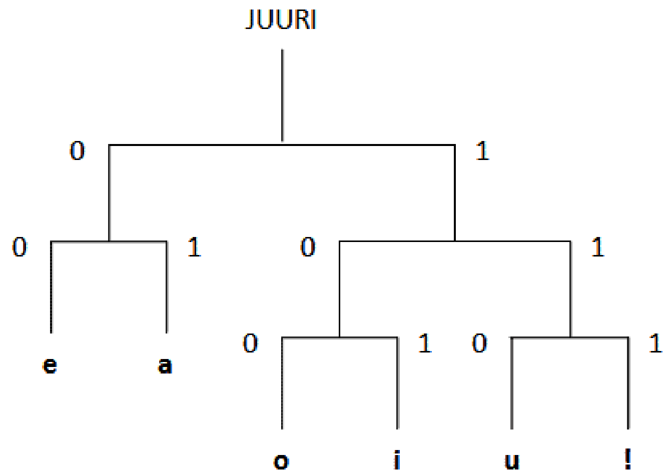
Symbolit jaetaan osiin viidessä eri vaiheessa. Aluksi symbolit *ea* muodostavat ensimmäisen osan ja symbolit *oiu!* toisen osan, joista kummankin yhteenlaskettu todennäköisyys on 0.5. Ensimmäisen osan symbolien koodi alkaa siis binääriluvulla 0 ja toisen osan symbolien koodi binääriluvulla 1.

Toisessa vaiheessa symbolit *ea* jakaantuvat osiksi *e* ja *a*, joista ensin mainitun todennäköisyys on 0.3 ja jälkimmäisen 0.2. Koska kummassakin osassa on jaon jälkeen vain yksi symboli, *e* saa lopulliseksi koodikseen 00 ja *a* koodikseen 01.

Kolmannessa vaiheessa symbolit *oiu!* jakaantuvat osiksi *oi* ja *u!*, joista ensin mainitun todennäköisyys on 0.3 ja jälkimmäisen 0.2. Symbolien *oi* koodi on tässä vaiheessa 10 ja symbolien *u!* koodi 11.

Neljännessä vaiheessa symbolit *oi* jakaantuvat osiksi *o* ja *i*, joista ensin mainitun todennäköisyys on 0.2 ja jälkimmäisen 0.1. Koska kummassakin osassa on jaon jälkeen vain yksi symboli, *o* saa lopulliseksi koodikseen 100 ja *i* koodikseen 101.

Lopuksi vielä jäljellä olevat symbolit *u!* jakaantuvat osiksi *u* ja *!*, joista kummankin todennäköisyys on 0.1. Koska kummassakin osassa on jaon jälkeen vain yksi symboli, *u* saa lopulliseksi koodikseen 110 ja *!* koodikseen 111. Edellä esitetty esimerkki voidaan nyt esittää puuna, jossa merkkijonon *aeiou!* symbolit muodostavat puun lehdet (kuva 4.2).



Kuva 4.2: Shannon-Fano-puu.

Taulukkoon 4.1 on koottu muutamia merkkijonon *aeiou!* symboleja kuvaavia suureita, joita ovat symbolien todennäköisyydet Tn, Shannon-Fano-koodit SF sekä niistä lasketut entropian H ja koodin keskimääräisen pituuden L arvot. Koodin keskimääräinen pituus ja entropia eli tiedon määrä, joka muodostaa alarajan koodin keskimääräiselle pituudelle, on laskettu jokaiselle merkkijonon symbolille erikseen käyttäen kaavoja 2.2 ja 2.3, ja arvot on lopuksi laskettu yhteen Summa-rivillä.

Taulukko 4.1: Shannon-Fano-koodin keskimääräisen pituuden vertailua entropiaan.

Symboli	Tn	SF	H	L
a	0.2	01	0.46	0.4
e	0.3	00	0.52	0.6
i	0.1	101	0.33	0.3
o	0.2	100	0.46	0.6
u	0.1	110	0.33	0.3
!	0.1	111	0.33	0.3
$\sum$			2.43	2.5

Taulukosta havaitaan, että suuren todennäköisyyden omaavat symbolit on koodattu vähemmällä bittimäärällä, kuin pienen todennäköisyyden symbolit, ja että merkkijonon koodin keskimääräinen pituus lähenee entropian arvoa. Shannon-Fano-koodi ei ole kuitenkaan optimaalinen, sillä koodin keskimääräinen pituus ei saavuta alinta mahdollista arvoa.

Koodin purkaminen

Kuvan 4.2 puun avulla voidaan havainnollistaa myös Shannon-Fano-koodin purkaminen, joka aloitetaan puun juuresta. Koodatusta merkkijononosta *aeiou!* luetaan ensimmäinen bitti, joka on 0. Juurisolmusta käännytään vasemmalle, minkä jälkeen luetaan toinen bitti. Koska toinen bitti on 1, käännytään seuraavasta juurisolmusta oikealle, jolloin saavutetaan puun ensimmäinen lehti ja sitä kuvaava symboli *a* voidaan lukea. Tapahtuma alkaa alusta puun juurisolmusta, kun luetaan merkkijonon seuraava vuorossa oleva bitti, ja se etenee edellä kuvatulla tavalla. Toimenpidettä jatketaan niin kauan kunnes kaikki merkkijonon *aeiou!* symbolit on saatu purettua.

4.2 Huffman-koodaus

Huffmanin koodi on pakkausalgoritmi, jonka David Huffman kehitti 1950-luvun alussa tekstin pakkaamiseen [10]. Se on suosittu menetelmä, johon monet tekstin ja kuvan pakkaamiseen tarkoitetut ohjelmistosovellukset perustuvat. Joissakin sovelluksissa käytetään pelkästään Huffmanin menetelmää, mutta sitä käytetään myös muiden menetelmien täydentäjänä. [21, s. 12]

Huffmanin menetelmä muistuttaa edellisessä luvussa kuvattua Shannon-Fano-menetelmää, mutta Huffmanin menetelmällä saavutetaan yleensä parempi pakkaustulos kuin Shannon-Fano-menetelmällä. Pääero näiden kahden menetelmän välillä on se, että Shannon-Fano-menetelmässä koodit muodostetaan ylhäältä alaspäin kun taas Huffmanin menetelmässä alhaalta ylöspäin. [22, s. 61]

Perusalgoritmi on ollut laajan tutkimuksen kohteena aina kehittämisvuodestaan 1952 lähtien. Perusmenetelmää on kirjallisuudessa nimitetty staattiseksi (*static*) / ei-adaptiiviseksi (*nonadaptive*) eli muuttumattomaksi menetelmäksi (esim. [14]). Koska kirjainten esiintymistodennäköisyyksiä ei kuitenkaan voida käytännössä tuntea ennakolta, oli tarpeen kehittää malli, joka korjaa nämä puutteet. Staattisen menetelmän rinnalle kehitettiinkin dynaaminen (*dynamic*) / adaptiivinen (*adaptive*) eli

muuttuva malli, jossa esiintymistodennäköisyyksien sallitaan vaihtuvan eli muuttuvan tapahtuman edetessä.

Muuttuvan mallin kehittivät alunperin Faller [6] ja Gallager [8], johon Knuth [12] teki myöhemmin parannuksia. Perusalgoritmia kutsutaankin kehittäjiensä mukaan FGK-algoritmiksi. Vitter [33] paranteli algoritmia edelleen, ja näin FGK-algoritmin sovelluksena syntyi niinikään kehittäjänsä mukaan nimetty V-algoritmi. Seuraavassa käsitellään muuttumattoman ja muuttuvan Huffmanin koodin muodostamista ja purkamista [21, 22, 23].

4.2.1 Muuttumaton Huffman-koodaus

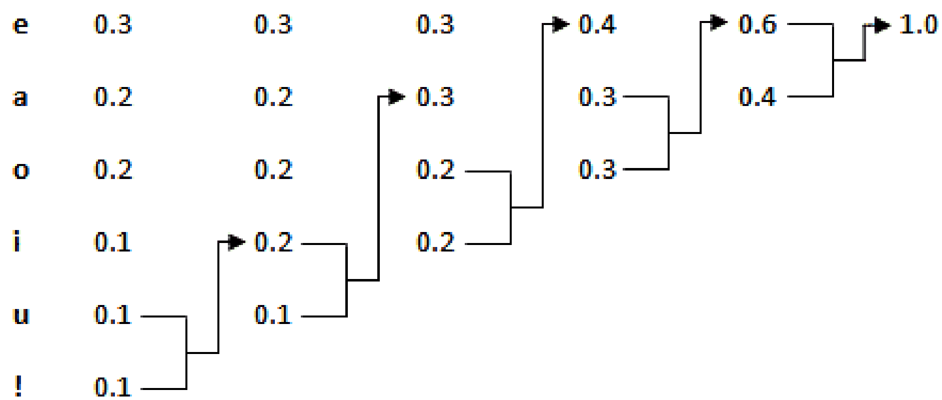
Muuttumattomassa Huffmanin menetelmässä aineisto luetaan kertaalleen läpi tilastotietojen keräämiseksi. Varsinainen koodaus tehdään vasta toisessa vaiheessa. Jos sama merkki esiintyy merkkijonossa useammin kuin yhden kerran, se saa aina saman koodin. Menetelmän perusperiaate on seuraava:

```
ALGORITMI 2. Muuttumaton Huffmanin algoritmi.

1. Listaa kaikki symbolit ja niiden todennäköisyydet.
2. Lajittele symbolit todennäköisyyksiensä perusteella laske-
   vaan järjestykseen. Symbolit muodostavat metsän, josta muo-
   dostetaan Huffman-puu.
3. Yhdistä kaksi todennäköisyydeltään epätodennäköisintä sym-
   bolia keskenään, ja korvaa symbolit uudella symbolilla, jo-
   ka saa todennäköisyydekseen kahden edellisen symbolin toden-
   näköisyyksien summan. Näin on muodostunut puu, joka lisä-
   tään metsään.
4. Jatka seuraavasta kahdesta todennäköisyydeltään epätodennä-
   köisimmästä symbolista, jotka yhdistetään ja korvataan uu-
   della symbolilla. Näin on muodostunut uusi puu, joka lisä-
   tään aikaisempaan metsään.
5. Jatka niin kauan kunnes symboleja on listalla enää yksi,
   joka on puun juurisolmu. Puu on nyt lopullisesti valmis.
```

Havainnollistetaan edellä esitettyä algoritmia samalla esimerkillä, jota käsiteltiin jo edellisessä luvussa eli merkkijonolla *aeiou!*, jonka symbolien vastaavat todennäköisyydet ovat siis 0.2, 0.3, 0.1, 0.2, 0.1 ja 0.1. Kuten Shannon-Fano-menetelmäs-

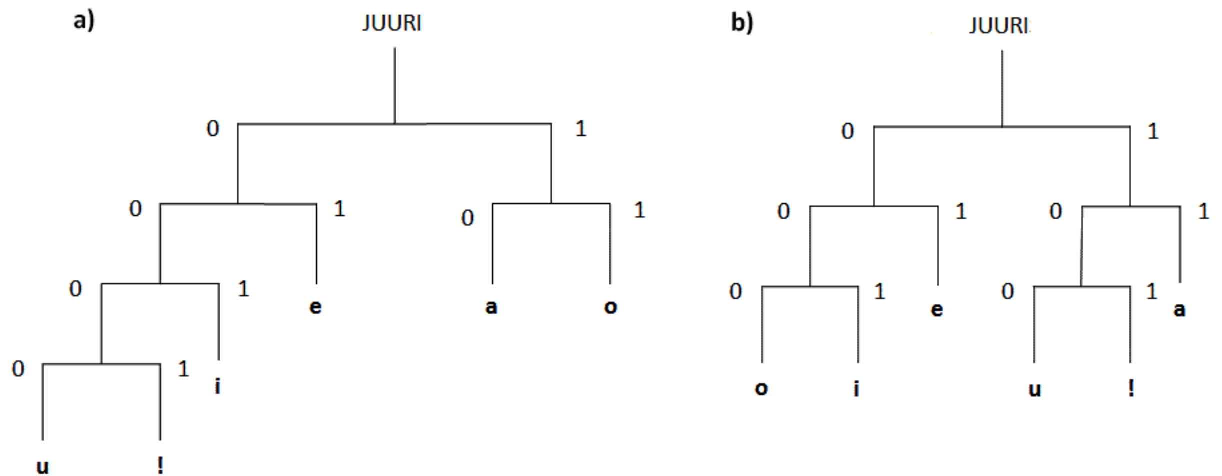
säkin, symbolit järjestetään ensin todennäköisyyksiensä perusteella laskevaan järjestykseen kuvan 4.3 mukaisesti.



Kuva 4.3: Muuttumattoman Huffmanin koodin muodostamisperiaate.

Aluksi kaksi pienimmän todennäköisyyden omaavaa symbolia *u* ja *!* yhdistetään ja ne korvataan uudella symbolilla *(u,!)*, joka saa arvokseen todennäköisyyksien summan eli 0.2. Jotta laskeva järjestys säilyisi, uusi symboli arvoineen siirtyy nyt listassa symbolien *o* ja *i* väliin. Seuraavaksi yhdistetään seuraavat kaksi pienimmän todennäköisyyden omaavaa symbolia *(u,!)* ja *i*, ja ne korvataan vastaavasti uudella symbolilla *((u,!),i)*, joka saa arvokseen 0.3. Jotta laskeva järjestys säilyisi, uusi symboli asetetaan symbolien *e* ja *a* väliin. Seuraavaksi yhdistetään symbolit *a* ja *o*, jotka korvataan uudella symbolilla *(a,o)* ja arvolla 0.4. Jotta laskeva järjestys säilyisi, uusi symboli asetetaan listan ylimmäksi ennen symbolia *e*. Jatketaan yhdistämistä symbolipareilla *e* ja *((u,!),i)*, jotka korvataan symbolilla *(((u,!),i),e)* ja arvolla 0.6. Jotta laskeva järjestys säilyisi, uusi symboli asetetaan listan ylimmäksi ennen symbolia *(a,o)*. Lopuksi yhdistetään kaksi jäljellä olevaa symbolia *(((u,!),i),e)* ja *(a,o)*, joiden yhteistodennäköisyys saa arvon 1.0.

Kuvan 4.3 esimerkistä muodostuu kuvan 4.4 a) kaltainen puu. Kuten Shannon-Fano-puussakin, Huffman-puun vasen haara saa arvokseen bitin 0 ja oikea haara bitin 1. Kun kuljetaan puuta pitkin puun juuresta aina puun lehteen, saadaan merkkijonon *aeiou!* symboleille koodit 10, 01, 001, 11, 0000 ja 0001.



Kuva 4.4: Muuttumaton Huffman-puu kahtena eri versiona a ja b, joiden välinen ero syntyy symbolien erilaisesta yhdistämistavasta. Katso tarkempi selitys tekstistä.

Taulukkoon 4.2 on koottu muutamia merkkijonon *aeiou!* symboleja kuvaavia suureita, joita ovat symbolien todennäköisyydet Tn, entropia H, Shannon-Fano-koodit SF, Huffman-koodit HF sekä niistä lasketut keskimääräisen pituuden arvot Shannon-Fano-koodille L_{SF} ja Huffman-koodille L_{HF}. Koodin keskimääräinen pituus ja entropia eli tiedon määrä, joka muodostaa alarajan koodin keskimääräiselle pituudelle, on laskettu jokaiselle merkkijonon symbolille erikseen käyttäen kaavoja 2.2 ja 2.3, ja arvot on lopuksi laskettu yhteen Summa-rivillä.

Taulukko 4.2: Shannon-Fano- ja Huffman-menetelmien keskinäistä vertailua.

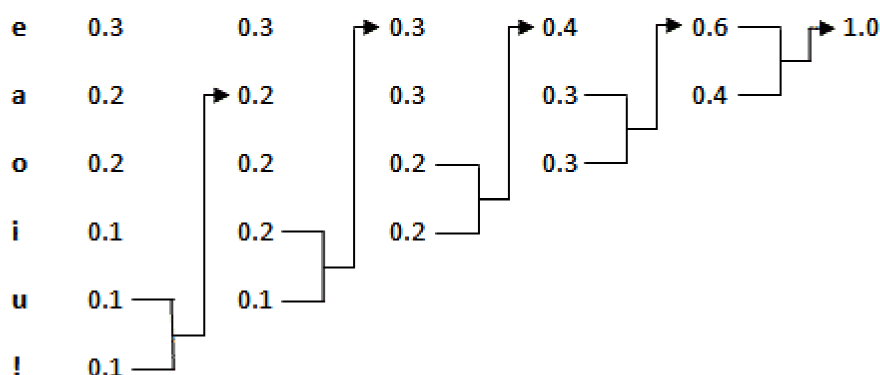
Symboli	Tn	H	SF	L_{SF}	HF	L_{HF}
a	0.2	0.46	01	0.4	10	0.4
e	0.3	0.52	00	0.6	01	0.6
i	0.1	0.33	101	0.3	001	0.3
o	0.2	0.46	100	0.6	11	0.4
u	0.1	0.33	110	0.3	0000	0.4
!	0.1	0.33	111	0.3	0001	0.4
$\sum$		2.43		2.5		2.5

Kuten Shannon-Fano-menetelmässä, myös Huffman-menetelmässä todennäköisimmän symbolin koodaamiseen tarvitaan vähemmän bittejä kuin epätodennäköisimmän symbolin koodaamiseen. Huffman-menetelmä tuottaa tässä esimerkissä kolme kahden bitin koodia, yhden kolmen bitin koodin ja kaksi neljän bitin koodia, kun taas Shannon-Fano-menetelmässä koodit ovat joko kaksi- tai kolmebittisiä. Menetelmien erilaisuudesta huolimatta koodin keskimääräinen pituus on molemmilla sama, joten ainakaan tämän esimerkin valossa Huffman-menetelmä ei pakkaisi paremmin kuin Shannon-Fano-menetelmä, mikä ei yleensä pidä paikkaansa [2, s. 105].

Kummankin menetelmän huono puoli on se, että koodin pituus ilmaistaan aina kokonaisluvulla, vaikka todellinen arvo olisi desimaaliluku. Taulukossa 4.2 laskettu koodin keskimäärinen pituus 2.5 on todellisuudessa 3, mikä on melko kaukana saadusta entropian arvosta 2.43. Tästä syystä Huffman-menetelmääkään ei voida pitää optimaalisena koodausmenetelmänä.

Huffmanin koodi ei ole yksiselitteinen, kuten voidaan todeta kuvasta 4.5, jossa symbolit on yhdistetty hieman eri tavalla kuin kuvassa 4.3 mutta kuitenkin niin, että syntyneen koodin keskimääräinen pituus pysyy samana. Tämä ilmenee taulukosta 4.3, jossa on esitetty merkkijonon *aeiou!* symbolien todennäköisyydet Tn ja entropia H, merkkijonosta muodostetut kahdet erilaiset Huffman-koodit HF_a ja HF_b sekä niistä lasketut keskimääräisen pituuden arvot L_{HF_a} ja L_{HF_b} sekä varianssin Var_{HF_a} ja Var_{HF_b} arvot. Koodin keskimääräinen pituus ja entropia eli tiedon määrä, joka muodostaa alarajan koodin keskimääräiselle pituudelle, on laskettu jokaiselle merkkijonon symbolille erikseen käyttäen kaavoja 2.2 ja 2.3, kun taas varianssi on laskettu niinikään jokaiselle symbolille erikseen käyttäen Salomonin [22, s. 64] esittämää laskutapaa, ja saadut arvot on laskettu yhteen Summa-rivillä.

Taulukosta huomataan, että koodilla HF_b on pienempi varianssi kuin koodilla HF_a, mikä itse asiassa tekee ensin mainitusta koodista jälkimmäistä paremman [22, s. 64]. Varianssi tässä esimerkissä ilmaisee sen, kuinka paljon yksittäisten koodisanojen koot poikkeavat keskiarvosta. Varianssilla on merkitystä etenkin silloin, kun koodaaja lähettää pakatun merkkijonon tiedonsiirtokanavaa pitkin eteenpäin. Jos varianssi on pieni, sitä pienemmän puskurin koodaaja tarvitsee koodattujen bittien lähettämiseksi eteenpäin ja päin vastoin. Mikäli koodaaja vain yksinkertaisesti kirjoittaa pakatun merkkijonon tiedostoksi, varianssilla ei ole juurikaan merkitystä.



Kuva 4.5: Muuttumattoman Huffmanin koodin muodostamisperiaate, kun symboliparit on yhdistetty hieman toisin. Esimerkki on esitetty puuna kuvassa 4.4 b).

Taulukko 4.3: Muuttumattoman Huffmanin menetelmän kaksi eri puuta.

Symboli	Tn	H	HF_a	L_{HF_a}	Var_{HF_a}	HF_b	L_{HF_b}	Var_{HF_b}
a	0.2	0.46	10	0.4	0.05	11	0.4	0.05
e	0.3	0.52	01	0.6	0.075	01	0.6	0.075
i	0.1	0.33	001	0.3	0.025	001	0.3	0.025
o	0.2	0.46	11	0.4	0.05	000	0.6	0.1125
u	0.1	0.33	0000	0.4	0.225	100	0.3	0.025
!	0.1	0.33	0001	0.4	0.225	101	0.3	0.025
$\sum$		2.43		2.5	0.65		2.5	0.31

Koodin purkaminen

Huffmanin koodin purkualgoritmi on yksinkertainen ja se etenee samaan tapaan kuin Shannon-Fano-menetelmässä. Koodatusta merkkijononosta *aeiou!* luetaan ensimmäinen bitti, joka on 1. Juurisolmusta käännytään oikealle, minkä jälkeen luetaan toinen bitti. Koska toinen bitti on 0, käännytään seuraavasta juurisolmusta vasemmalle, jolloin saavutetaan puun ensimmäinen lehti ja sitä kuvaava symboli *a* voidaan lukea. Tapahtuma alkaa alusta puun juurisolmusta, kun luetaan merkkijonon seuraava vuorossa oleva bitti, ja se etenee edellä kuvatulla tavalla. Toimenpidettä jatketaan niin kauan kunnes kaikki merkkijonon *aeiou!* symbolit on saatu purettua.

4.2.2 Muuttuva Huffman-koodaus

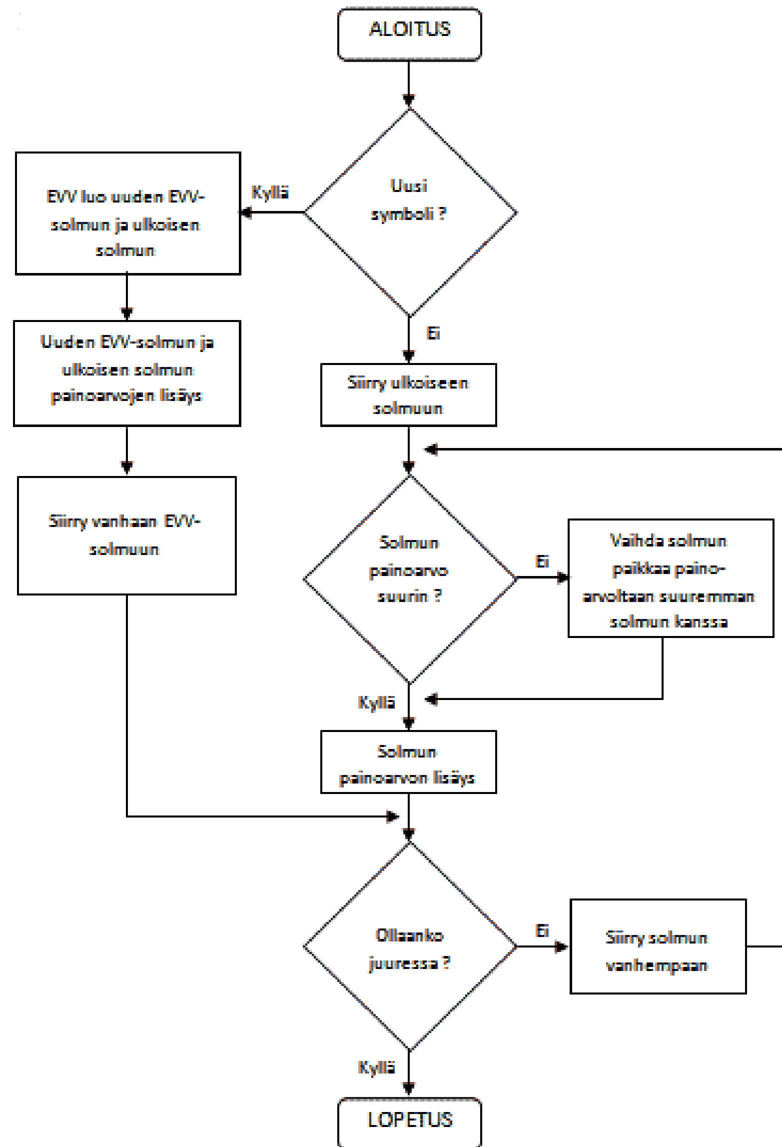
Muuttuva Huffmanin menetelmä poikkeaa edellä kuvatusta Huffmanin perusalgoritmista siinä, että enkooderilla ei ole ennestään tilastotietoja aineistosta. Muuttuvassa menetelmässä enkooderi sekä lukee että koodaa aineiston samanaikaisesti. Lisäksi saman symbolin koodisana voi vaihtua moneenkin otteeseen tapahtuman edetessä. Dekooderi toimii samaan tapaan kuin enkooderi koodia purettaessa.

Aluksi Huffman-puu on tyhjä ja sisältää ainoastaan yhden yksittäisen solmun EVV (*Ei Vielä Välitetty*), joka edustaa kaikkia vielä lukemattomia symboleja ja jonka painoarvo on nolla. Sitä mukaa kun symboleja luetaan, niitä kuvaavat solmut lisätään puuhun, joka järjestäytyy uudelleen kuvassa 4.6 esitetyn vuokaavion mukaisesti. Päivityksen tarkoituksena on säilyttää puun sisarominaisuus (*sibling property*), mikä tarkoittaa sitä, että puun jokaisella solmulla juurisolmua lukuun ottamatta on sisar ja että solmut esiintyvät painoarvojensa (= todennäköisyys) mukaisessa järjestyksessä [8]. Jokainen puu, jolla on tämä ominaisuus, on Huffman-puu.

Symbolien koodit sovitaan ennen merkkijonon koodaamista seuraavaan tapaan [23, s. 58]. Jos merkkijono kuuluu kooltaan m olevaan aakkostoon $\Omega = \{a_1, a_2, ...a_m\}$, valitaan e ja r siten, että $m = 2^e + r$ ja $0 \leq r < 2^e$. Symboli a_k koodataan numeron $(k-1)$ $(e+1)$-bittisenä binäärimuotona, jos $1 \leq k \leq 2r$, muutoin a_k koodataan numeron $(k-r-1)$ e-bittisenä binäärimuotona. Tarkasteltavassa esimerkissä aakkostona on $\Omega = \{a, e, i, o, u, !\}$, jolloin m=6, e=2 ja r=2. Näin saadaan taulukon 4.4 mukainen koodilista.

Taulukko 4.4: Symbolien ennalta sovitut koodit.

Symboli	Koodi
a	000
e	001
i	010
o	011
u	10
!	11



Kuva 4.6: Päivitystapahtuma muuttuvalle Huffmanin algoritmille [23, s. 59].

Muuttuvan Huffmanin koodin muodostamisen periaate on seuraava [23, s. 62]:

```
ALGORITMI 3. Muuttuva Huffmanin koodausalgoritmi.

1. Aloita tyhjällä puulla.
2. Lue merkkijonon symboli.
     JOS symboli esiintyy ensimmäisen kerran, lähetä koodi
     EVV-solmulle ja heti sen perään listassa näkyvä koodi
     symbolille.
     MUUTOIN lähetä symbolille koodi, joka saadaan kulkemal-
     la puun juuresta symbolia kuvaavaavaan lehteen.
3. Päivitä puu.
4. JOS kyseessä on viimeinen symboli, lopeta.
   MUUTOIN siirry kohtaan 2.
```

Kuten aikaisemmin todettiin, alussa puu sisältää vain yhden solmun, EVV-solmun. Siksi ensimmäisenä luettava symboli saa koodikseen ennalta sovitun koodin koodilistasta (taulukko 4.4). Tämän jälkeen jokainen uusi symboli saa koodikseen EVV-solmun koodin, joka saadaan kulkemalla puun juuresta ko. solmuun ja jota seuraa symbolin ennalta sovittu koodi koodilistasta. Tämän nk. nollasolmun koodin lisääminen ennen symbolin koodia on merkityksellinen siksi, että siitä vastaanottaja tietää symbolia kuvaavan solmun puuttuvan puusta. Jos luettavalla symbolilla on jo oma solmu puussa, symbolin koodi saadaan yksinkertaisesti kulkemalla puun juuresta ko. solmuun. Kun symbolin koodi on lähetetty ja uudet solmut lisätty puuhun, symboli otetaan pois koodilistasta.

Havainnollistetaan asiaa esimerkillä, jossa koodataan merkkijono *aaeiou!* (kuva 4.7). Solmuihin merkitään selvyyden vuoksi järjestysnumero, joka kuvaa kyseisen solmun paikkaa puussa. Koska puu tulee sisältämään yhteensä 13 solmua, aloitetaan solmujen numerointi luvusta 13, jolloin viimeinen solmu saa numeron 1.

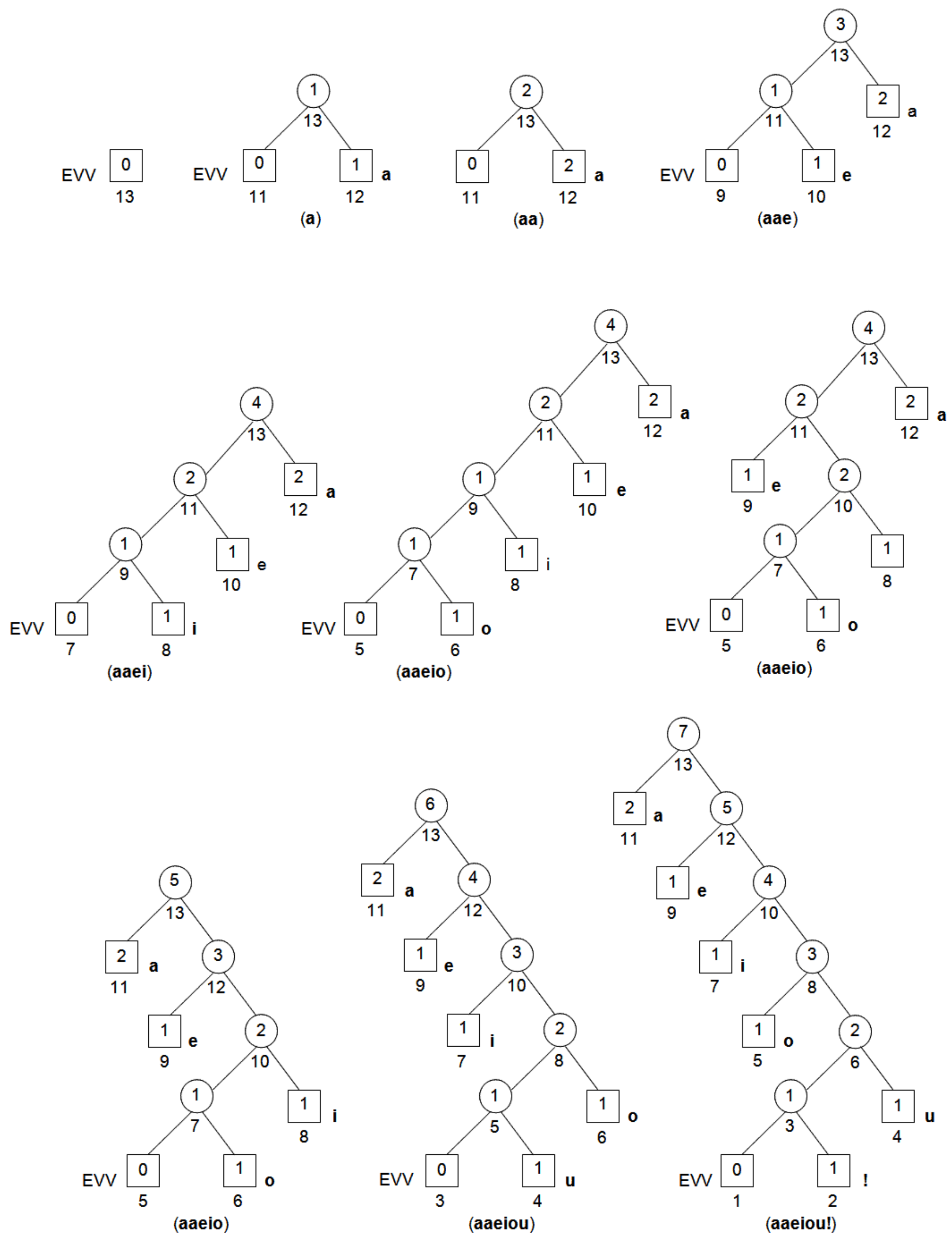
Alussa lähdetään liikkelle tyhjästä puusta eli EVV-solmusta, jonka järjestysnumero on siis 13 ja painoarvo 0 (ks. ensimmäinen kuva ylärivissä kuvassa 4.7). Aluksi luetaan merkkijonon ensimmäinen symboli *a* (ks. toinen kuva ylärivissä kuvassa 4.7). Koska ko. symboli esiintyy ensimmäisen kerran, se saa koodikseen binääriluvun 000 taulukon 4.4 mukaisesti. Sitten EVV-solmu luo uuden EVV-solmun (solmu 11) ja *a*-solmun (solmu 12), joka saa painoarvokseen 1. Tämän jälkeen juurisolmun (solmu 13) painoarvo lisääntyy yhdellä.

Seuraava luettava symboli on jälleen *a* (katso kolmas kuva ylärivissä kuvassa 4.7). Koska sillä on jo oma solmu puussa, symbolin *a* koodi saadaan kulkemalla puun juuresta ko. solmuun eli koodiksi saadaan 1. Solmun *a* painoarvo lisääntyy yhdellä, minkä jälkeen myös juurisolmun (solmu 13) painoarvo lisääntyy yhdellä.

Seuraavaksi luetaan symboli *e* (ks. viimeinen kuva ylärivissä kuvassa 4.7). Koska symboli esiintyy ensimmäisen kerran, se saa koodikseen EVV-solmun koodin 0 ja symbolin *e* koodin 001 taulukon 4.4 mukaisesti. Sitten EVV-solmu (solmu 11) luo uuden EVV-solmun (solmu 9) ja *e*-solmun (solmu 10), joka saa painoarvokseen 1. Tämän jälkeen solmun 11 painoarvo lisääntyy yhdellä. Huomataan, että edellä lisätty *a*-solmu (solmu 12) on siirtynyt puussa ylemmäs. Puuta ei kuitenkaan tarvitse päivittää. Lopuksi juurisolmun (solmu 13) painoarvo lisääntyy yhdellä.

Seuraavana on vuorossa symboli *i* (ks. ensimmäinen kuva keskellä kuvassa 4.7), joka luetaan niinikään ensimmäisen kerran. Symboli saa koodikseen EVV-solmun koodin 00 ja symbolin *i* koodin 010 taulukon 4.4 mukaisesti. Kuten edellä, EVV-solmu (solmu 9) luo uuden EVV-solmun (solmu 7) ja *i*-solmun (solmu 8), joka saa painoarvokseen 1. Sen jälkeen solmun 9 painoarvo lisääntyy yhdellä. Jälleen huomataan, että kaikki aikaisemmin lisätyt solmut ovat siirtyneet puussa ylöspäin. Koska solmu 9 ja *e*-solmu (solmu 10) ovat painoarvojensa mukaisessa järjestyksessä, puuta ei tarvitse päivittää. Siirrytään tarkastelemaan seuraavan tason solmua 11. Myös solmut 11 ja 12 ovat painoarvojensa mukaisessa järjestyksessä, joten puuta ei edelleenkään tarvitse päivittää. Tämän jälkeen solmun 11 painoarvo lisääntyy yhdellä. Lopuksi juurisolmun (solmu 13) painoarvo lisääntyy yhdellä.

Seuraavaksi luetaan symboli *o* (ks. toinen ja kolmas kuva keskeltä sekä ensimmäinen kuva alarivistä kuvassa 4.7). Symboli saa koodikseen EVV-solmun koodin 000 ja *o*-solmun koodin 011 taulukon 4.4 mukaisesti. Jälleen EVV-solmu (solmu 7) luo uuden EVV-solmun (solmu 5) ja *i*-solmun (solmu 6), joka saa painoarvokseen 1. Sen jälkeen solmun 7 painoarvo lisääntyy yhdellä. Ja jälleen huomataan, että kaikki aikaisemmin lisätyt solmut ovat siirtyneet puussa ylöspäin. Koska solmu 7 ja *i*-solmu (solmu 8) ovat painoarvojensa mukaisessa järjestyksessä, puuta ei tarvitse päivittää. Siirrytään tarkastelemaan seuraavan tason solmua 9. Se ja *e*-solmu (solmu 10) vaihtavat paikkaa keskenään, koska ne ovat painoarvoiltaan väärässä järjestyksessä. Vaihdossa myös niiden järjestysnumerot vaihtuvat. Vasta tämän jälkeen solmun 10 painoarvo lisääntyy yhdellä. Siirrytään tarkastelemaan seuraavan tason solmua 11. Se ja solmu 12 ovat painoarvoltaan väärässä järjestyksessä, joten niiden keskinäinen paikka ja järjestysnumero vaihtuvat. Tämän jälkeen solmun 12 painoar-



Kuva 4.7: Muuttuvan Huffmanin koodin muodostaminen merkkijonolle *aaeiou!* Katso kuvan selitys tekstistä.

vo lisääntyy yhdellä. Lopuksi myös juurisolmun (solmu 13) painoarvo lisääntyy yhdellä.

Toiseksi viimeisenä symbolina luetaan symboli *u* (ks. keskimmäinen kuva alarivissä kuvassa 4.7). Symboli saa nyt koodikseen EVV-solmun koodin 1100 ja *u*-solmun koodin 10 taulukon 4.4 mukaisesti. Ja kuten edellä, uusi EVV-solmu (solmu 3) ja *u*-solmu (solmu 4) lisätään puuhun, jolloin aikaisemmin lisätyt solmut siirtyvät puussa ylemmäs. Puun solmut käydään läpi taso kerrallaan ja puu päivitetään tarvittaessa. Viimeiseksi luetaan symboli *!* (ks. viimeinen kuva alarivissä kuvassa 4.7). Symboli koodataan EVV-solmun koodilla 11100 ja *!*-solmun koodilla 11. Jälleen uusi EVV-solmu (solmu 1) ja *!*-solmu (solmu 2) lisätään puuhun, jolloin aikaisemmin lisätyt solmut siirtyvät puussa ylemmäs. Puun solmut käydään läpi taso kerrallaan ja puu päivitetään tarvittaessa. Merkkijonon *aeiou!* koodiksi saadaan lopulta 00010001000100000111100101110011.

Koodin purkaminen

Muuttuva Huffmanin koodi puretaan seuraavan purkutapahtuman mukaisesti [23, s. 64]:

```
ALGORITMI 4. Muuttuva Huffmanin purkualgoritmi.

1. Siirry puun juureen.
2. Tutkitaan, onko solmu puun lehti.
    JOS ei, luetaan bitti ja siirrytään kyseiseen solmuun.
    MUUTOIN tutkitaan, onko solmu EVV-solmu.
3. Tutkitaan, onko solmu EVV-solmu.
    JOS on, luetaan e:n osoittama määrä bittejä.
      Onko binäärilukua vastaava desimaaliluku p < r ?
        JOS ei, tee lisäys (p+r).
        MUUTOIN lue vielä yksi bitti.
      Pura (p+r):ä vastaava symboli EVV-listalta.
    MUUTOIN pura solmua kuvaava symboli.
4. Päivitä puu.
5. Lue seuraava bitti.
    JOS bittejä on vielä jäljellä, siirry kohtaan 1
    MUUTOIN lopeta.
```

Kun saatua binääriesitystä luetaan, puussa kuljetaan edes takaisin samaan tapaan kuin merkkijonoa koodattaessa. Kun puun lehti saavutetaan, sitä vastaava symboli puretaan. Jos lehti on EVV-solmu, binääriesityksestä luetaan e:n ilmaisema määrä bittejä. Jos sitä vastaava luku on vähemmän kuin r:n ilmaisema arvo, luetaan vielä yksi bitti. Symbolia vastaava indeksi saadaan kun (e)- tai $(e+1)$-bittistä binäärilukua vastaavaan desimaalilukuun lisätään luku 1. Kun symboli on saatu purettua, puu päivitetään ja jatketaan seuraavien bittien lukemista.

Seuraavaksi katsotaan, miten edellä saadun merkkijonon *aaeiou!* koodi puretaan. Alussa lähdetään liikkeelle tyhjästä puusta. Dekooderi lukee aluksi kaksi ensimmäistä bittiä, koska e:n arvo on 2. Kyseiset kaksi bittiä 00 vastaavat lukua 0. Koska se on vähemmän kuin r:n arvo 2, dekooderi lukee vielä yhden bitin, jolloin saadaan ensimmäisen symbolin koodi. Kun koodia vastaavaan desimaalilukuun 0 lisätään 1, saadaan symbolin indeksiksi luku 1, joka on symbolin a indeksi. Täten ensimmäiseksi puretaan symboli a, ja puu päivitetään kuvan 4.6 mukaisesti.

Seuraavaksi dekooderi lukee bitin 1, joka saadaan kulkemalla puun juurta pitkin solmuun *a*. Symboli *a* puretaan ja puu päivitetään.

Seuraava bitti on 0, joka vastaa EVV-solmua. Dekooderi lukee tämän jälkeen seuraavat kaksi bittiä 00, jotka vastaavat lukua 0. Dekooderi lukee vielä yhden bitin, jolloin saadaan desimaalilukua 1 vastaavat bitit 001. Kun lukuun 1 lisätään 1, saadaan symbolia *e* vastaavaksi indeksiksi 2. Täten *a*:n jälkeen puretaan siis symboli *e* ja puu päivitetään.

Seuraavat kaksi bittiä 00 vastaavat EVV-solmua. Dekooderi jatkaa lukemista kahdella seuraavalla bitillä 01, joka vastaa desimaalilukua 1. Dekooderi lukee vielä yhden bitin, jolloin saadaan desimaalilukua 2 vastaavat bitit 010. Kun lukuun 2 lisätään 1, saadaan symbolia *i* vastaavaksi indeksiksi 3. Näin puretaan seuraavana symbolina *i* ja puu päivitetään kuten edellä. Toimenpidettä jatketaan niin kauan, kunnes merkkijonon kaikki symbolit on saatu purettua.

4.3 Aritmeettinen koodaus

Aritmeettinen koodaus on suhteellisen uusi pakkausmenetelmä. Menetelmän perusperiaatteen esitti ensimmäisenä Peter Elias 1960-luvun alkupuolella ilmestyneessä julkaisemattomassa tutkimuksessaan, jota referoi ensimmäisenä Abramson vuonna 1963 kirjassaan *Information Theory and Coding*. Ensimmäiset aritmeettisen koodauksen käytännön menetelmät toivat julki Richard Pasco ja Jorma Rissanen 1970-luvun puolivälissä, joiden tutkimuksia seurasi lukuisa joukko muita aiheeseen liittyviä artikkeleita 1970- ja 1980-luvuilla. [35]

Lukuisista julkaisuista huolimatta aritmeettinen koodaus pysyi lähinnä kuriositeettina, kunnes kiinnostus ja ymmärrys menetelmää kohtaan lisääntyivät erityisesti Moffat ym. [15] ja Witten ym. [35] julkaisemien tutkimusten johdosta.

Aritmeettisesta menetelmästä on Huffmanin menetelmän tapaan sekä muuttumaton että muuttuva versio. Näitä kumpaakin tarkastellaan seuraavassa.

4.3.1 Muuttumaton aritmeettinen koodaus

Aritmeettinen koodaus on menetelmä, jossa koodattavaa merkkijonoa kuvaa puoliavoin väli [0,1). Algoritmi on rekursiivinen siten, että se koodaa ja/tai purkaa merkkijonon symboli kerrallaan jokaisella toistokerralla. Algoritmi jakaa ensin edel-

lä mainitun välin osiin ja asettaa sitten jonkun kyseisistä osista uudeksi väliksi, jota koodattava merkkijono edustaa. [13]

Kun merkkijono pitenee, sitä kuvaava väli pienenee ja sitä määrittävien bittien lukumäärä kasvaa. Todennäköisimmät symbolit pienentävät väliä vähemmän kuin epätodennäköisemmät symbolit luoden vähemmän bittejä merkkijonoon. [35]

Muuttumattoman aritmeettisen koodausmenetelmän algoritmi on esitetty seuraavassa [22, s. 124]:

```
ALGORITMI 5. Muuttumaton aritmeettinen koodausalgoritmi.

1. Aloita asettamalla aloitusväliksi [0,1).
2. Toista seuraavia kahta vaihetta syötteen jokaiselle symbo-
   lille s.
     Jaa aloitusväli alaväleihin, joiden koko on verrannolli-
     nen symbolien todennäköisyyksiin.
     Valitse symbolia s vastaava alaväli ja määritä se uudek-
     si aloitusväliksi.
3. Kun kaikki symbolit on käyty läpi edellä kuvatulla tavalla,
   tulos on mikä tahansa luku, joka kuvaa aloitusväliä ainut-
   kertaisesti.
```

Tarkastellaan seuraavaksi, kuinka aritmeettinen koodaus tapahtuu käytännössä [35]. Oletetaan, että tarkasteltava merkkijono kuuluu aikaisemmista luvuista tuttuun aakkostoon $\Omega = \{a, e, i, o, u, !\}$. Ensimmäisessä vaiheessa enkooderi rakentaa taulukossa 4.5 esitetyn mallin, joka perustuu aakkoston symbolien todennäköisyyksiin.

Taulukko 4.5: Malli muuttumattomalle aritmeettiselle koodausesimerkille.

Symboli	Tn	Väli
a	0.2	[0, 0.2)
e	0.3	[0.2, 0.5)
i	0.1	[0.5, 0.6)
o	0.2	[0.6, 0.8)
u	0.1	[0.8, 0.9)
!	0.1	[0.9, 1.0)

Koodataan nyt esimerkkinä vaikkapa merkkijono *eaii!*. Enkooderi tietää, että kyseinen merkkijono sijoittuu välille [0,1). Kun enkooderi lukee ensimmäisen symbolin *e*, enkooderi pienentää alkuperäistä aloitusväliä määrittämällä ko. symbolin välin [0.2, 0.5) uudeksi aloitusväliksi ja jakamalla sen pienempiin alaväleihin, joiden koko on verrannollinen kunkin symbolin todennäköisyyteen (taulukko 4.6, kuva 4.8). Enkooderi lukee seuraavaksi symbolin *a* ja määrittää uudeksi aloitusväliksi [0.2,0.26) sekä jakaa sen edelleen pienempiin alaväleihin, joiden koko on verrannollinen kunkin symbolin todennäköisyyteen. Sitten enkooderi lukee vuorostaan symbolin *i* ja määrittää jälleen uudeksi aloitusväliksi [0.23,0.236) ja jakaa sen pienempiin alaväleihin edellä esitettyyn tapaan. Enkooderi toimii samoin loppujen symbolien kanssa, ja lopputuloksena merkkijonon *eaii!* koodiksi valitaan jokin luku väliltä [0.23354,0.2336) binäärimuodossa.

Taulukko 4.6: Merkkijonon *eaii!* aloitusvälin muuttuminen kutakin symbolia luettaessa.

	Symboli	Väli
aloitusväli		[0,1)
luettaessa	e	[0.2,0.5)
	a	[0.2,0.26)
	i	[0.23,0.236)
	i	[0.233,0.2336)
	!	[0.23354,0.2336)

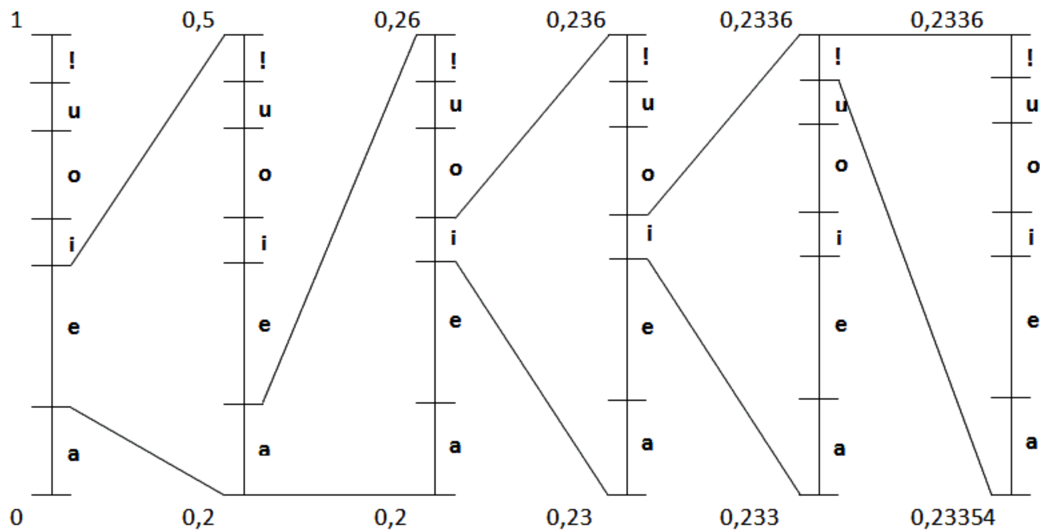
Algoritmi päivittää taulukossa 4.6 esitetyt välit aina uutta symbolia lukiessaan seuraavasti [22, s. 125]:

$$UusiAla = VanhaAla + Väli \cdot AlaVäli(X) \quad (4.1)$$

ja

$$UusiYlä = VanhaAla + Väli \cdot YläVäli(X), \quad (4.2)$$

joissa *UusiAla* (*NewLow*) tarkoittaa uutta alarajaa, *UusiYlä* (*NewHigh*) uutta ylärajaa, *VanhaAla* (*OldLow*) vanhaa alarajaa, *Väli* (*Range*) väliä, *AlaVäli(X)* (*LowRange*) tarkasteltavan symbolin X aloitusvälin alarajaa ja *YläVäli(X)* (*HighRange*) vastaavaa ylärajaa. Täten taulukossa 4.6 esimerkiksi symbolin a välin alarajaksi saadaan $UusiAla = 0,2 + (0,5 - 0,2) \cdot 0 = 0,2$ ja ylärajaksi $UusiYlä = 0,2 + (0,5 - 0,2) \cdot 0,2 = 0,26$.



Kuva 4.8: Muuttumaton aritmeettinen koodaus merkkijonolle *eaii!* [35, s. 522].

Koodin purkaminen

Aritmeettisen koodin purkaminen tapahtuu koodaukseen nähden käänteisessä järjestyksessä [22, s. 126-127]. Otetaan esimerkiksi edellä koodattu merkkijono *eaii!*. Ensimmäisessä vaiheessa dekooderi luo taulukossa 4.5 esitetyn mallin uudelleen. Tämän jälkeen alkaa varsinainen merkkijonon purkaminen lukemalla koodista sen ensimmäinen luku, joka on 2. Tästä dekooderi tietää, että koko luvun täytyy olla muotoa 0.2. Kyseinen luku on symbolia *e* kuvaavalla välillä [0.2, 0.5), joten ensimmäisen symbolin täytyy olla *e*. Tämän jälkeen dekooderi eliminoi symbolin *e* vaikutuksen koodista vähentämällä siitä symbolia *e* kuvaavan välin alarajan 0.2 ja jakamalla ko. välillä 0.3. Tulokseksi saadaan 0.1118, joka kertoo dekooderille, että ollaan välillä [0, 0.2), jolloin seuraavan symbolin pitää olla *a*.

Tietyn symbolin X eliminointi koodista tapahtuu seuraavalla tavalla [22, s. 127]:

$$Koodi = (Koodi - Alaväli(X))/Väli \tag{4.3}$$

jossa *Koodi* (*Code*) tarkoittaa merkkijonon koodia, *AlaVäli(X)* (*LowRange*) tarkasteltavan symbolin *X* aloitusvälin alarajaa ja *Väli* (*Range*) väliä. Taulukossa 4.7 on esitetty dekoodauksen vaiheet, kun merkkijonon koodiksi on valittu koodin alaraja 0.23354.

Taulukko 4.7: Muuttumaton aritmeettinen dekoodaus.

Symboli	(*Koodi*	-	*AlaVäli(X)*)	/	*Väli*	=	*Koodi*
e	0.23354		0.2		0.3		0.1118
a	0.1118		0		0.2		0.559
i	0.559		0.5		0.1		0.59
i	0.59		0.5		0.1		0.9
!	0.9		0.9		0.1		0

4.3.2 Muuttuva aritmeettinen koodaus

Muuttuva aritmeettinen menetelmä poikkeaa edeltäjästään siinä, että enkooderi/ dekooderi ei tiedä luettavan merkkijonon symbolien todennäköisyyksiä ennakolta. Edellä esitetystä johtuen jokainen symboli saa aluksi puoliavoimelta väliltä [0,1) saman todennäköisyyden 1/n, jossa n on merkkijonon symbolien kokonaismäärä. Sitä mukaa kun koodaus etenee, symbolien todennäköisyydet ja symboleja kuvaavat välit päivittyvät.

Todennäköisyydet päivittyvät siten, että aina luettavan symbolin lukumäärä kasvaa yhdellä, kuin myös merkkijonon symbolien kokonaismäärä. Muiden symbolien lukumäärä pysyy sitävastoin samana. Symboleja kuvaavat välit päivittyvät aina suhteessa niiden muuttuneisiin todennäköisyyksiin. Menetelmän algoritmi on esitetty seuraavassa:

```
ALGORITMI 6. Muuttuva aritmeettinen koodausalgoritmi.

1. Aloita asettamalla aloitusväliksi [0,1).
2. Jaa aloitusväli alaväleihin, joiden koko on 1/n aloitusvä-
   lin koosta.
3. Lue merkkijonon ensimmäinen symboli s.
     Aseta symbolia vastaava alaväli uudeksi aloitusväliksi.
     Päivitä symbolien todennäköisyydet siten, että symbolia
     s vastaava todennäköisyys kasvaa.
     Päivitä symbolien alavälit.
4. Jatka kohdasta 3, kunnes kaikki symbolit on käyty läpi.
5. Luvun valinta.
```

Tarkastellaan seuraavaksi, kuinka koodaus tapahtuu käytännössä. Valitaan koodattavaksi merkkijonoksi *eaii!*, joka kuuluu aakkostoon $\Omega = \{a, e, i, o, u, !\}$. Ennen varsinaista koodausta enkooderi rakentaa taulukossa 4.8 esitetyn mallin, jossa aakkoston symboleilla on sama todennäköisyys.

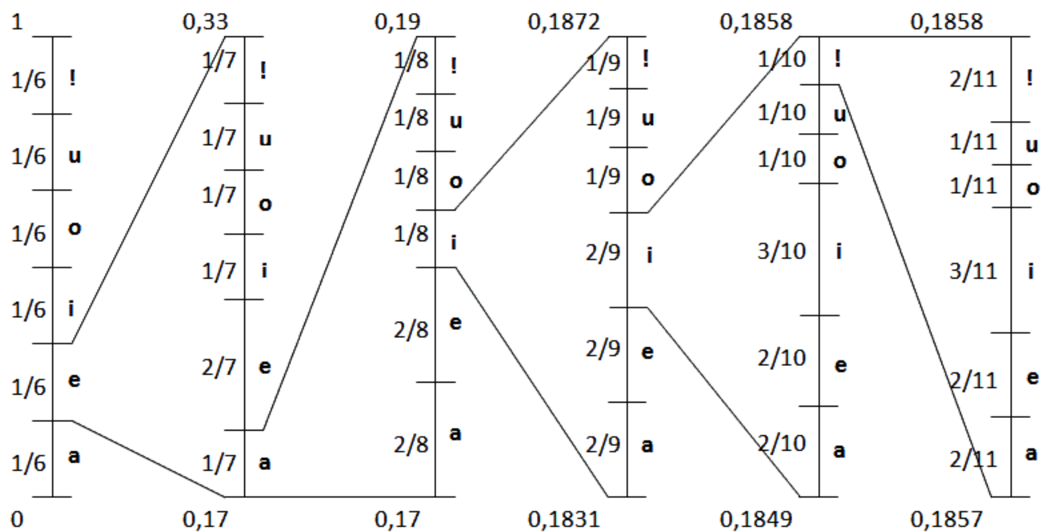
Taulukko 4.8: Malli muuttuvalle aritmeettiselle koodausesimerkille. Koodauksen edetessä sekä todennäköisyydet että välit muuttuvat.

Symboli	Tn	Väli
a	1/6	[0, 0.17)
e	1/6	[0.17, 0.33)
i	1/6	[0.33, 0.50)
o	1/6	[0.50, 0.67)
u	1/6	[0.67, 0.83)
!	1/6	[0.83, 1.0)

Enkooderi tietää, että kyseinen merkkijono sijoittuu välille [0,1). Kun enkooderi lukee ensimmäisen symbolin *e*, enkooderi pienentää alkuperäistä aloitusväliä määrittämällä ko. symbolin välin [0.17, 0.33) uudeksi aloitusväliksi ja jakamalla sen pienempiin alaväleihin, joiden koko on verrannollinen symbolien päivittyneisiin todennäköisyyksiin (taulukko 4.9, kuva 4.9). Enkooderi lukee seuraavaksi symbolin *a* ja määrittää uudeksi aloitusväliksi [0.17,0.19) sekä jakaa sen edelleen pienempiin alaväleihin, joiden koko on verrannollinen symbolien päivittyneisiin todennäköisyyksiin. Enkooderi toimii samoin loppujen symbolien kanssa, ja lopputuloksena merkkijonon *eaii!* koodiksi valitaan jokin luku väliltä [0.1857,0.1858) binäärimuodossa.

Taulukko 4.9: Merkkijonon *eaii!* symbolien todennäköisyyksien ja aloitusvälin muuttuminen kutakin symbolia luettaessa.

	Symboli	Tn	Väli
aloitusväli			[0,1)
luettaessa	e	2/7	[0.17,0.33)
	a	2/8	[0.17,0.19)
	i	2/9	[0.1831,0.1872)
	i	3/10	[0.1849,0.1858)
	!	2/11	[0.1857,0.1858)



Kuva 4.9: Muuttuva aritmeettinen koodaus merkkijonolle *eaii!*.

Koodin purkaminen

Muuttuvan aritmeettisen koodauksen purkaminen tapahtuu muuttumattoman menetelmän tapaan käänteisessä järjestyksessä koodaukseen nähden. Toisin kuin muuttumattoman menetelmän tapauksessa, binääriesitys ei muutu purkamisen edetessä, vaan dekooderi vertaa aina samaa lukua päivittyneisiin aloitusväleihin.

Otetaan jälleen esimerkiksi edellä koodattu merkkijono *eaii!*, jota kuvatkoon luku 0.1857 binäärimuodossa. Ensimmäisessä vaiheessa dekooderi lukee merkkijonon symbolit ja rakentaa taulukon 4.8 uudelleen. Tämän jälkeen alkaa varsinainen merkkijonon purkaminen lukemalla koodista sen kaikki luvut. Tästä dekooderi tietää, että koko luvun täytyy olla muotoa 0.1857. Koska kyseinen luku asettuu symbolia e kuvaavalle välille, ensimmäisen symbolin täytyy olla e. Tämän jälkeen symbolien todennäköisyydet ja aloitusvälit päivittyvät kuvan 4.9 tapaan.

Seuraavaksi dekooderi purkaa symbolin a, koska luku 0.1857 asettuu symbolia a kuvaavalle aloitusvälille. Jälleen symbolien todennäköisyydet ja aloitusvälit päivittyvät.

Seuraavaksi dekooderi purkaa symbolin i, koska luku 0.1857 asettuu symbolia i kuvaavalle aloitusvälille. Tämän jälkeen symbolien todennäköisyydet ja aloitusvälit päivittyvät jälleen. Näin jatketaan, kunnes viimeinen symboli on saatu purettua. Edellä esitetty dekoodausvaihe voidaan todeta oikeaksi vastaväitteen avulla.

Vastaväite: Dekoodaus on edennyt oikein toimivien vaiheiden jälkeen laskennan tilaan $\tilde{f}^k$*, jossa dekooderi purkaa oikein symbolin* α*, mutta siirtyy tämän jälkeen virheelliseen laskennan tilaan* $\tilde{f}^{k+1}$*.*

Näin ei voi olla, sillä dekoodaus etenee aina täsmälleen vastaavaan tilaan $\tilde{f}^{k+1}$ dekooderin purkaessa symbolia α tilassa $\tilde{f}^k$ kuin enkooderi koodatessaan symbolia α tilassa $\tilde{f}^k$. Näin ollen vastaväite on epätotta ja väite totta.

4.4 Lempel-Ziv-koodaus

Ajatus hakemistojen käytöstä tiedon pakkaamisessa liittyy oikeastaan jo aikaan, jolloin lennätin keksittiin [4, s. 441]. Tuohon aikaan yrityksiä veloitettiin kirjainmäärän mukaan, mistä syystä isot yritykset kehittivät koodikirjoja, jotka sisälsivät koodisanat yleisimmin käytetyille fraaseille. Kyseisiä koodisanoja voitiin sitten käyttää korvaamaan merkkijonoja lennättimien välisessä viestinnässä.

Informaatioteoreettinen tutkimus oli 1970-luvun lopulle tultaessa keskittynyt tilastollisiin pakkausmenetelmiin ja erityisesti Huffman-koodaukseen [16, s. 207]. Abraham Lempel ja Jacob Ziv mullistivat alan vuoden 1977 artikkelillaan *A Universal Algorithm for Sequential Data Compression* [36] sekä vuoden 1978 jatkoartikkelillaan *Compression of Individual Sequences via Variable-Rate Coding* [37], jotka varsinaisesti käynnistivät hakemistomenetelmiin perustuvan tutkimuksen. Aluksi tutkimukset herättivät mielenkiintoa lähinnä teoreettisessa mielessä, mutta menetelmää alettiin oikeastaan soveltaa käytäntöön vasta Welchin [34] kehitettyä algoritmista yksinkertaisen ja tehokkaan version.

Lempelin ja Zivin alkuperäisiin artikkeleihin pohjautuvia perusmenetelmiä kutsutaan yleisesti LZ77- ja LZ78-menetelmiksi, joista voi myös käyttää nimityksiä LZ1 ja LZ2 [23, s. 121]. Menetelmät ovat poikineet keksimistään seuranneena vuosikymmenenä lukuisan joukon eri variaatioita, joista tärkeimmät on listattu taulukkoon 4.10. Muita Lempel-Ziv-pohjaisia menetelmiä on esitellyt muun muassa Bell ym. [2].

Taulukko 4.10: Lempel-Ziv-pohjaisia koodausmenetelmiä.

Menetelmä	Kehittäjä(t)
LZ77	Lempel ja Ziv [36]
LZ78	Lempel ja Ziv [37]
LZSS	Storer ja Syzmanski [28]
LZW	Welch [34]

Lempel-Ziv-koodausmenetelmä on muuttuva hakemistomenetelmä, kuten suurin osa tunnetuista hakemistomenetelmistä [16, s. 203]. Muuttuvassa menetelmässä hakemisto päivittyy samaan aikaan kun dataa luetaan ja koodataan. Hakemistomenetelmiin kuuluu myös muuttumattomia versioita, jotka eroavat muuttuvista menetelmistä siinä, että hakemisto tunnetaan ennakolta, mikä tekee menetelmästä joustamattoman. Muuttumattomilla menetelmillä ei ole ollutkaan käytännön kannalta merkitystä.

4.4.1 LZ77 -koodaus

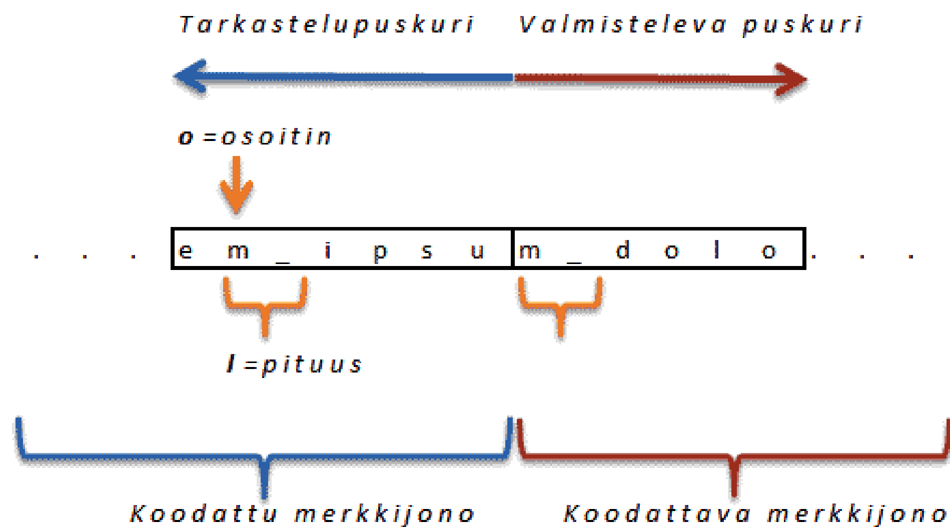
LZ77-menetelmässä enkooderi tutkii syöttötietoa niin kutsutun liukuvan ikkunan läpi, mistä syystä menetelmää kutsutaan myös Liukuva ikkuna (*Sliding Window*) -menetelmäksi (esim. [16, s. 215]). Ikkuna koostuu kahdesta osasta: tarkastelupuskurista (*search buffer*) ja valmistelevasta puskurista (*look-ahead buffer*), joista ensin mainittu sisältää koodatun syöttötiedon ja jälkimmäinen seuraavana koodausvuorossa olevan syöttötiedon [23, s. 121]. Menetelmän algoritmi on esitetty seuraavassa:

```
ALGORTMI 7. LZ77-koodausalgoritmi.

1. Lue merkkijonon seuraava symboli i valmistelevasta pusku-
   rista.
2. Etsi symbolille pisin vastaavuus tarkastelupuskurista.
3. Jos vastaavuus löytyy, koodaa symboli(t) kolmoismerkintänä
   (o,p,k), jossa o on pisintä vastaavuutta kuvaava osoitin,
   p merkkijonon pituus ja k vastaavuutta seuraava symboli,
   ja vie se hakemistoon.
4. Siirrä merkkijonoa  vastaavuuden verran vasemmalle.
5. Jatka kohdasta 1 niin kauan, kunnes valmistelevan puskurin
   kaikki symbolit on käyty läpi.
```

Koodauksen perusperiaate on esitetty kuvassa 4.10. Enkooderi etsii tarkastelupuskurista vastaavuutta valmistelupuskurin ensimmäiselle koodattavalle symbolille. Jos vastaavuus löytyy, enkooderi laittaa kyseisen paikan kohdalle osoittimen ja rupeaa etsimään lisää vastaavuuksia valmistelevasta puskurista. Enkooderi pyrkii löytämään aina pisimmän vastaavuuden. Kun pisin vastaavuus löytyy, enkooderi koodaa merkkijonon symbolit kolmoismerkintänä (*o,p,k*), jossa *o* tarkoittaa osoitinta eli vastaavan merkkijonon ensimmäisen symbolin paikkaa valmistelevasta puskurista katsottuna, *p* vastaavan merkkijonon pituutta ja *k* löydettyä vastaavuutta seuraavaa koodisanaa valmistelevassa puskurissa. Kuvan 4.10 esimerkissä valmistelevan puskurin ensimmäiselle symbolille *m* on löytynyt vastaavuus tarkastelupuskurista, jolloin enkooderi laittaa osoittimen symbolin *m* kohdalle. Kyseisen symbolin jälkeen löytyy vielä toinen vastaavuus eli symboli _. Tällöin merkkijonon *m_* koodiksi tulee (*6,2,d*), joka lisätään hakemistoon. Koodattava merkkijono siirtyy tämän jälkeen kolmella merkillä eteenpäin tarkastelupurkurissa, jolloin seuraava valmistelevassa puskurissa käsiteltävä oleva symboli on *o*.

Tarkastellaan seuraavaksi, kuinka Liukuva ikkuna -koodaus tapahtuu käytännössä (kuva 4.11). Oletetaan, että tarkasteltava merkkijono kuuluu jälleen aikaisemmista luvuista tuttuun aakkostoon $\Omega = \{a, e, i, o, u, !\}$ ja että koodattavana merkkijonona on *iaeuaiaoaeuauuauuao*. Oletetaan lisäksi, että tässä esimerkissä liukuvan ikkunan pituus on yhteensä 13 merkkiä (tarkastelupuskuri 7 merkkiä, valmisteleva puskuri 6 merkkiä), vaikka käytännössä puskurit ovat kooltaan suurempia [23, s. 121].

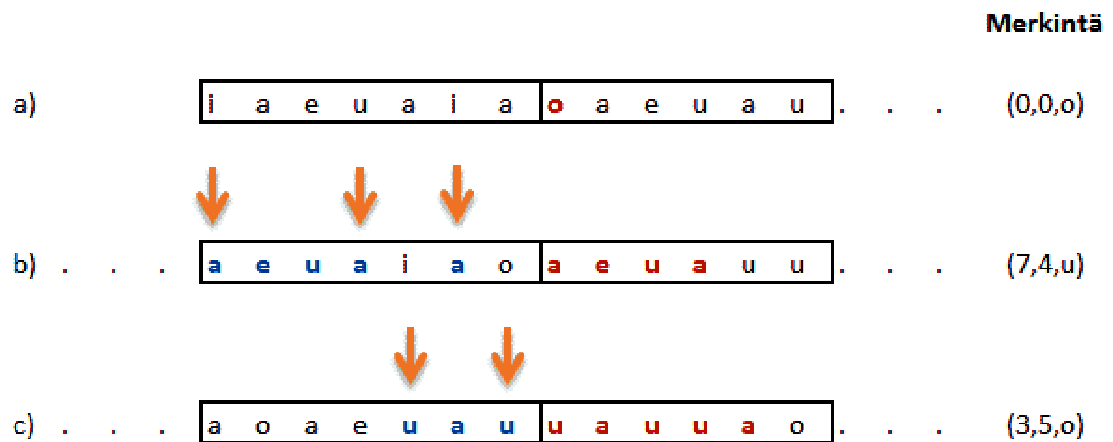


Kuva 4.10: LZ77-koodauksen perusperiaate.

Oletetaan vielä, että merkkijonon alku *iaeuaia* on jo koodattu ja että valmistelevassa puskurissa on merkkijono *oaeuau* (kuva 4.11 a)). Enkooderi etsii merkkijonon ensimmäiselle symbolille *o* vastaavuutta tarkastelupuskurin jo koodatusta osasta. Kuten huomataan, vastaavuutta ei löydy, joten *o* saa hakemistossa kolmoismerkinnän (*0,0,o*), ja ikkuna liukuu yhden merkin oikealle.

Nyt valmistelevassa puskurissa on merkkijono *aeuauu* (kuva 4.11 b)). Jälleen enkooderi etsii merkkijonon ensimmäiselle symbolille *a* vastaavuutta tarkastelupuskurista. Enkooderi löytää ensimmäisen vastaavuuden kahden merkin päästä valmistelevasta puskurista, ja vastaavuuden pituus on yksi. Enkooderi etsii muita vastaavuuksia ja löytääkin seuraavan neljän merkin päästä valmistelevasta puskurista, ja vastaavuuden pituus on tälläkin kertaa yksi. Enkooderi löytää vielä yhden vastaavuuden seitsemän merkin päästä. Nyt vastaavuuden pituus onkin neljä, jolloin enkooderi valitsee tämän viimeksi löydetyn vastaavuuden eli merkkijonon *aeua*, joka saa hakemistossa kolmoismerkinnän (*7,4,u*), ja ikkuna liukuu viisi merkkiä oikealle.

Viimeisessä vaiheessa valmistelevassa puskurissa on merkkijono *uauuao* (kuva 4.11 c)). Enkooderi löytää merkkijonon ensimmäiselle kirjaimelle *u* kaksi vastaavuutta tarkastelupuskurista: ensimmäisen vastaavuuden, joka on yhden merkin päässä valmistelevasta puskurista ja jonka pituus on yksi, sekä toisen vastaavuuden, joka on kolmen merkin päässä valmistelevasta puskurista ja jonka pituus näyttäisi aluksi olevan kolme. Tässä tapauksessa pituus onkin viisi kolmen sijaan, ja merkkijono



Kuva 4.11: LZ77-koodausesimerkki. Tarkastelun kohteena oleva symboli/merkkijono on merkitty vahvennettuna punaisena ja sitä vastaavat osumat vahvennettuna sinisenä. Oranssi osoitin näyttää löydetyn osuman aloituskohdan.

uauua saa hakemistossa kolmoismerkinnän (*3,5,o*). Ikkuna liukuu nyt kuusi merkkiä oikealle seuraavaa vaihetta varten. Näin jatketaan niin kauan, kunnes merkkijonon kaikki symbolit on käsitelty.

Hakemistossa olevat kolmoismerkinnät koodataan vielä vaihtuvamittaisin koodein seuraavan periaatteen mukaan [23, s. 122]:

$$Koodi = \lceil log_2 S \rceil + \lceil log_2 W \rceil + \lceil log_2 A \rceil, \tag{4.4}$$

jossa S tarkoittaa tarkastelupuskuria, W liukuvaa ikkunaa kokonaisuudessaan ja A syöttötiedoston kokoa.

LZ77-koodauksen yksi ongelma ilmenee tilanteessa, jossa merkkijono sisältää paljon epäsäännöllisesti esiintyviä symboleja, joille ei löydy vastaavuutta tarkastelupuskurista, mikä tekee menetelmästä tehottoman [23, s. 125]. Tämä voidaan välttää käyttämällä nk. lippubittiä (*flag*) erotusmerkkinä koodatun ja koodaamattoman symbolin välissä, jolloin koodaamisessa voidaan käyttää kaksoismerkintää kolmoismerkinnän sijaan. Tämä LZ77-muunnos tunnetaan nimellä LZSS kehittäjiensä Storer ja Syzmanski mukaan [28].

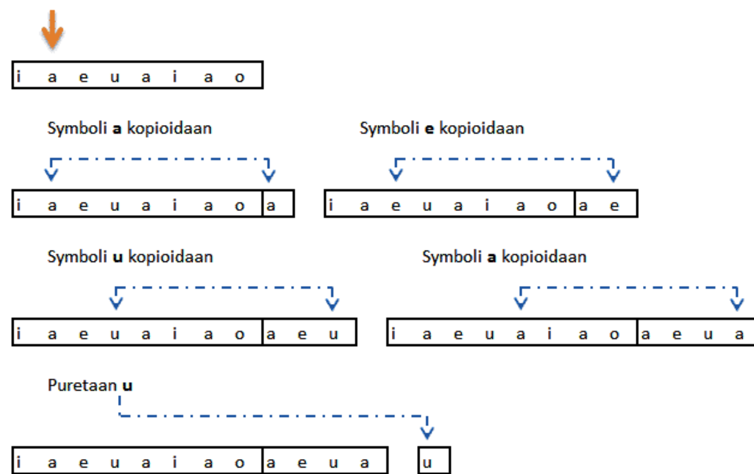
Toinen ongelmallinen tilanne syntyy silloin, jos tarkastelupuskuri on lyhyempi kuin toistuvan merkkijonon pituus [23, s. 126]. Tällöin menetelmä laajentaa tietoa pakkaamisen sijaan.

Koodin purkaminen

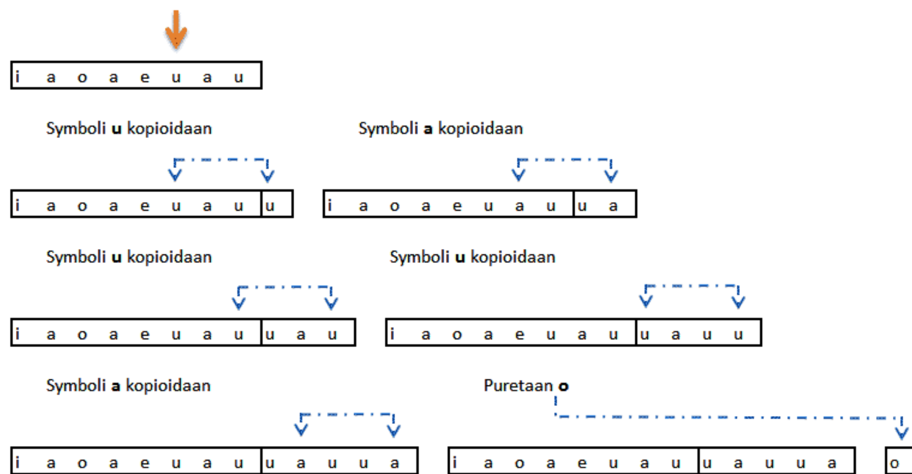
Tarkastellaan seuraavaksi edellä koodatun merkkijonon *iaeuaiaoaeuauuauuao* purkamista. Oletetaan aluksi, että merkkijonon alku *iaeuaia* on jo purettu ja dekooderi vastaanottaa kolmoismerkinnät (*0,0,o*), (*7,4,u*) ja (*3,5,o*). Ensimmäinen kolmoismerkintä on helppo purkaa, koska aikaisemmin puretusta merkkijonosta ei löytynyt yhtään vastaavuutta, jolloin seuraava symboli on *o*. Purettu merkkijono on tällöin *iaeuaiao*. Seuraavan kolmoismerkinnän ensimmäinen merkintä kertoo dekooderille, että osoitinta on siirrettävä takaisinpäin seitsemän merkin verran ja kopioitava siitä neljä merkkiä kuvan 4.12 mukaisesti.

Tarkastellaan seuraavaksi kolmoismerkinnän (*3,5,o*) purkamista (kuva 4.13). Kolmoismerkinnän ensimmäinen merkintä kertoo dekooderille, että osoitinta on siirrettävä taaksepäin kolmen merkin verran. Kolmoismerkinnän toinen merkintä ilmaisee kopioitavien merkkien määrän, joka on viisi. Dekooderi kopioi ensin kolme ensimmäistä merkkiä *uau*. Osoitin siirtyy jälleen kopioidakseen viimeksi kopioidun symbolin *u*. Samaan tapaan kopioituu symboli *a*. Vaikka alunperin kopioitiin vain kolme merkkiä, purettujen merkkien lukumääräksi tulikin viisi.

Edellinen esimerkki osoittaa, että merkkijono voi jatkua valmistelevan puskurin puolelle, vaikka purkaminen alkaakin tarkastelupuskurista.



Kuva 4.12: Kolmoismerkinnän (*7,4,u*) purkaminen (vasemmalta oikealle).



Kuva 4.13: Kolmoismerkinnän (*3,5,o*) purkaminen (vasemmalta oikealle).

4.4.2 LZ78-koodaus

Toisin kuin LZ77-menetelmässä, LZ78-menetelmässä ei käytetä liukuvaa ikkunaa [22, s. 95] [23, s. 126]. Merkkijonosta pyritään kuitenkin löytämään pisin vastaavuus, joka viedään hakemistoon kaksoismerkintänä (*i,k*) ja jossa *i* tarkoittaa pisintä vastaavuutta osoittavaa indeksiä ja *k* pisimmän vastaavuuden viimeisen merkin koodia. Menetelmän algoritmi on esitetty seuraavassa:

```
ALGORITMI 8. LZ78-koodausalgoritmi.

1. Aloita tyhjästä hakemistosta paikasta 0.
2. Lue merkkijonon seuraava symboli i.
3. Etsi symbolille pisin vastaavuus hakemistosta.
4. Vie pisin vastaavuus hakemistoon seuraavalle vapaalle
   paikalle kaksoismerkintänä (i,k), jossa i on pisintä
   vastaavuutta osoittava indeksi ja k pisimmän vastaavuuden
   viimeisen merkin koodi.
5. Jatketaan kohdasta 2, kunnes merkkijonon kaikki symbolit
   on käyty läpi.
```

Tarkastellaan seuraavaksi, miten LZ78-koodaus tapahtuu käytännössä (taulukko 4.11). Olkoon esimerkkinä edellisen luvun tapaan aakkoston $\Omega = \{a, e, i, o, u, !\}$ merkkijono *iaeuaiaoaeuauuauuao*.

Alussa hakemisto on tyhjä. Sen täyttäminen alkaa tyhjällä merkkijonolla paikasta nolla. Merkkijonosta luetaan aluksi sen ensimmäinen symboli *i*. Koska se ei voi löytyä tyhjästä hakemistosta, symboli *i* viedään hakemiston ensimmäiselle paikalle 1 merkinnällä (*0,i*). Samoin käy kolmen seuraavan symbolin *a*, *e* ja *u* kanssa, joita ei löydy hakemistosta ennestään ja jotka viedään hakemiston seuraaville paikoille 2-4 vastaavalla kaksoismerkinnällä kuin symbolilla *i*.

Seuraavaksi luetaan symboli *a*, joka löytyy hakemistosta paikasta 2. Symbolin *a* jälkeen luetaan symboli *i*. Koska yhdistelmää *ai* ei löydy hakemistosta ennestään, yhdistelmä viedään hakemiston seuraavalle vapaalle paikalle kaksoismerkinnällä (*2,i*). Ja näin jatketaan eteenpäin merkkijonon loppuun asti.

Taulukko 4.11: LZ78-koodauksen periaate.

Indeksi	Vastaavuus	Merkintä
0	0	
1	i	(0,i)
2	a	(0,a)
3	e	(0,e)
4	u	(0,u)
5	ai	(2,i)
6	ao	(2,o)
7	ae	(2,e)
8	ua	(4,a)
9	uu	(4,u)
10	au	(2,u)
11	uao	(4,o)
12	"tyhjä"	(0,eof)

Myös LZ78-menetelmää on yritetty parantaa. Suosituin LZ78-muunnos lienee Welchin [34] kehittämä algoritmi LZW, jossa enkooderi lähettää hakemistoon ainoastaan indeksin *i* kaksoismerkinnän (*i,k*) sijaan. Kyseistä menetelmää on kuvannut myös Phillips [19].

Koodin purkaminen

Koodia purettaessa LZ78-dekooderi täyttää ja ylläpitää hakemistoa samaan tapaan kuin enkooderi, mikä tekee prosessista monimutkaisemman kuin LZ77-koodin tapauksessa [22, s. 97].

Dekooderi vastaanottaa ensin kaksoismerkinnän (*0,i*), jonka ensimmäinen merkki *0* kertoo dekooderille, että hakemistosta ei löydy ennestään kyseistä symbolia, joten merkkijonon ensimmäisen symbolin täytyy olla *i*. Seuraavaksi dekooderi vastaanottaa kaksoismerkinnän (*0,a*), jossa ensimmäinen merkki on jälleen *0*. Tästä dekooderi päättelee, että hakemistosta ei löydy ennestään kyseistä symbolia, jolloin merkkijonon toisen symbolin täytyy olla *a*. Ja näin jatketaan merkkijonon loppuun.

5 Häviöttömien tiedon pakkausmenetelmien vertailua

Luvussa 3.3 käsiteltiin tiedon pakkausmenetelmiä, niiden luokittelua ja suorituskyvyn eri arviointitapoja. Tässä luvussa tarkastellaan suorituskyvyn arviointitapoja lähemmin ja selvitetään, pystyykö niiden avulla päättelemään, mikä tässä työssä käsitellyistä menetelmistä olisi paras tiedon pakkaamiseen.

5.1 Staattisuus vs. adaptiivisuus

Kuten aikaisemmin on todettu, pakkausmenetelmä voi olla staattinen, adaptiviinen tai niiden välimuoto. Tässä työssä käsitellyt pakkausmenetelmät kuuluvat kahteen ensin mainittuun. Staattisessa menetelmässä pakkaus tapahtuu kaksivaiheisesti: ensin data luetaan kertaalleen tilastotietojen keräämiseksi, minkä jälkeen se pakataan. Adaptiivinen menetelmä on yksivaiheinen, koska data päivittyy prosessin aikana.

Kumpi näistä tavoista on sitten parempi? Voidaan olettaa, että staattiset menetelmät ovat hitaampia kuin adaptiiviset menetelmät, koska pakkausprosessi on kaksivaiheinen ja vie tästä syystä enemmän aikaa. Toisaalta, adaptiiviset menetelmät vaativat todennäköisesti tietokoneelta enemmän tehoja, koska data päivittyy prosessin aikana. Jos koneessa ei ole riittävästi tehoja, se vaikuttaa myös osaltaan suoritusnopeuteen, jolloin adaptiiviset menetelmät voivat olla jopa staattisia menetelmiä hitaampia.

5.2 Tehokkuus

Tiedon pakkaamisen tarkoituksena on poistaa datasta tiedon ylimäärää eli redundanssia. Tällöin tieto tiivistyy alkuperäistä kokoaan pienemmäksi. Koodin keskimääräisen pituuden alarajaksi määritettiin luvussa 2.1 entropia. Mitä enemmän tieto tiivistyy, ts. mitä enemmän tiedon ylimäärää saadaan poistettua, sen lyhyempi on koodin keskimääräinen pituus, ja sen lähemmäksi entropiaa päästään.

Lelewer ja Hirsberg [14] vertasivat artikkelissaan mm. eri pakkausmenetelmillä saatuja koodin keskimääräisiä pituuksia L keskenään. Tähän työhön liittyvät menetelmät sisälsivät Shannon-Fano-menetelmän, staattisen ja adaptiivisen Huffmanin

menetelmän, staattisen aritmeettisen menetelmän sekä LZ78-menetelmän.

He tutkivat esimerkkiä *EXAMPLE*, johon kuuluivat symbolit $\alpha = \{a, b, c, d, e, f, g, \sqcup\}$. Symboli $\sqcup$ tarkoittaa tässä välilyöntiä. Symbolien todennäköisyydet olivat vastaavasti $\{0.05, 0.075, 0.1, 0.125, 0.15, 0.175, 0.2, 0.125\}$. Paras pakkaustulos saatiin aikaiseksi staattisella aritmeettisella menetelmällä, jonka avulla koodin keskimääräiseksi pituudeksi L saatiin 116 bittiä ja jolla siis päästiin lähimmäksi koodin keskimääräisen pituuden määrittävää alarajaa eli entropiaa (taulukko 5.1).

Taulukko 5.1: Eri pakkausmenetelmillä saatuja koodin keskimääräisen pituuden arvoja, esimerkkinä *EXAMPLE* [14].

Menetelmä	L
Shannon-Fano	117
Staattinen Huffman	117
Adaptiivinen Huffman	129
Staattinen aritmeettinen	116
LZ78	173

Edellä esitetystä yksittäisestä esimerkistä ei tietenkään voida vetää johtopäätöksiä algoritmin hyvyydestä. Koodin keskimääräisistä pituuksista huomataan, että sekä adaptiivinen Huffmanin menetelmä että LZ78-menetelmä eivät pääse oikeuksiinsa, kun esimerkkinä on lyhyt merkkijono, joka sisältää vain erillisiä symboleja. Kuten aiemmin todettiin, adaptiivinen menetelmä on tehokkain silloin, kun merkkijono sisältää useita samoja symboleja, jotka kasvattavat kukin vuorollansa symbolin todennäköisyyttä. Tällöin merkkijono koodataan pienemmällä bittimäärällä, jolloin saadaan aikaiseksi parempi pakkaustulos.

Algoritmien tehokkuuksia ovat tutkineet mm. Vitter [33], Welch [34] ja Witten ym.[35]. Vitter vertasi adaptiivisia Huffmanin algoritmeja FGK (H_{FGK}) ja V (H_V) staattiseen Huffmanin algoritmiin (H_{STAT}). Kutakin menetelmää testattiin erilaisiin tekstitiedostoihin, Pascalin koodiin sekä TeX-ohjelmalla luotuihin tiedostoihin. Algoritmien suorituskykyä arvioitiin saadun koodin sisältämien bittien määrällä. Taulukkoon 5.2 on koottu tiedot pienehkön tekstitiedoston osalta ja taulukkoon 5.3 Pascalin koodin osalta.

Taulukosta 5.2 voidaan havaita, että mitä pienempi tiedosto on, staattinen menetelmä näyttäisi päihittävän adaptiivisen menetelmän suorituskyvyssä. Tiedoston koon kasvaessa tilanne näyttäisi kääntyvän päinvastaiseksi. Sama näyttäisi pätevän myös taulukkoon 5.3. Saaduista tuloksista on kuitenkin todettava, että erot eri menetelmien välillä eivät loppujen lopuksi ole kovinkaan suuret.

Taulukko 5.2: Huffmanin algoritmien vertailua pienelle tekstitiedostolle, kun n on tiedoston koko 8-bittisellä koodisanalla ja k erilaisten merkkijonojen lukumäärä [33]. Luvut kussakin algoritmisarakkeessa tarkoittavat kyseisen algoritmin prosenttiosuutta alkuperäisestä merkkijonon pituudesta.

n	k	H_{STAT}	H_V	H_{FGK}
100	96	83.0	71.1	82.4
500	96	83.0	80.8	83.5
961	97	83.5	82.3	83.7

Taulukko 5.3: Huffmanin algoritmien vertailua Pascalin koodille, kun n on tiedoston koko bitteinä ja k erilaisten merkkijonojen lukumäärä [33]. Lukujen selitys kuten taulukossa 5.2.

n	k	H_{STAT}	H_V	H_{FGK}
100	32	57.4	56.2	58.9
500	49	61.5	62.2	63.0
1000	57	61.3	61.8	62.4
10000	73	59.8	59.9	60.0
12067	78	59.6	59.8	59.9

Welch [34] arvioi Lempel-Ziv-koodien suorituskykyä erilaisille tietotyypeille laskemalla niille pakkaussuhteet. Welch laski pakkaussuhteen tässä työssä esitetyn pakkaustekijän kaavalla 3.2, joten alkuperäisestä taulukosta poiketen pakkaussuhteen tilalla käytetään tässä työssä termiä pakkaustekijä (CF). Tarkasteltavina tietotyyppeinä olivat teksti, Cobol-tiedostot, valikoima liukulukuja, formatoitu tieteellinen aineisto, systeemin lokitiedosto, ohjelman lähdekoodi sekä objektikoodi. Welch

ei katsonut tarpeelliseksi erotella eri menetelmiä toisistaan, sillä ne antoivat toistensa suhteen hyvin samankaltaisia tuloksia. Tulokset näkyvät taulukossa 5.4. Tulokset tekstin osalta osoittivat, että pitkät tekstidokumentit eivät tiivistyneet paremmin kuin lyhyemmät tekstidokumentit viitaten siihen, että tiedon ylimäärä ei riippuisikaan sisällön korrelaatiosta.

Taulukko 5.4: Lempel-Ziv-koodien suorituskyky eri tietotyyppien osalta [34].

Tietotyyppi	*CF*
Englanninkielinen teksti	1.8
Cobol-tiedostot	2-6
Liukulukuvalikoima	1.0
Formatoitu tieteellinen aineisto	2.1
Systeemin lokitiedosto	2.6
Ohjelman lähdekoodi	2.3
Objektikoodi	1.5

Witten ym. [35] vertasi omassa tutkimuksessaan adaptiivista aritmeettista menetelmää adaptiiviseen Huffmanin menetelmään, jossa aritmeettisen menetelmän testaukseen käytettiin optimoitua C-kielistä toteutusta ja adaptiivisen Huffmanin menetelmän testaukseen Compress-pakkausohjelmaa, joka koostuu FGK-algoritmista. Testattavina tietotyyppeinä olivat teksti, C-ohjelma, VAX-objektiohjelma, aakkosto sekä vääristymätilasto. Taulukossa 5.5 on esitetty koodauksen tulokset bitteinä aritmeettiselle menetelmälle (A_T) ja Huffmanin menetelmälle (H_T). Taulukosta voidaan havaita, että aritmeettisen menetelmän pakkaustehokkuus on kutakuinkin parempi kuin adaptiivisen Huffmanin menetelmän kaikkien testattavien tietotyyppien osalta.

Taulukko 5.5: Adaptiivisten aritmeettisen ja adaptiivisen Huffmanin algoritmien pakkaustehokkuuksien vertailua [35]. Koodauksen tulos on ilmaistu bitteinä (b).

	A_T	H_T
Teksti	57,718	57,781
C-ohjelma	62,991	63,731
VAX-objektiohjelma	73,546	76,950
Aakkosto	59,292	60,127
Vääristymätilasto	12,092	16,257

Kun vertaillaan eri menetelmiä keskenään, on luontevaa vertailla niitä sisältäviä pakkausohjelmia keskenään. Taulukkoon 5.6 on listattu muutama esimerkki käytössä olevista pakkausohjelmista ja niissä käytetyistä algoritmeista. Taulukosta havaitaan, että useissa pakkausohjelmissa käytetään useampaa kuin yhtä algoritmia. Lisäksi monet ohjelmat ovat LZ-pohjaisia, mikä osaltaan viittaa menetelmän saavuttamaan suosioon.

PC-maailmassa käytetyin pakkausohjelma on WinZip, Unix-puolella GNUzip [29]. WinZip pakkaa datan Internetissä yleisimmin käytössä olevaan .zip-tiedostomuotoon. WinZipin kaltaisia pakkausohjelmia ovat PowerArchiver ja UltimateZip. GNUzip tuottaa .gz-muotoisia yksittäisiä tiedostoja, mutta ei arkistoi niitä kuten WinZip. Siksi sen kanssa käytetään .tar-muotoa, joka ei pakkaa, vaan yhdistää tiedostot pakkausta varten. Yhdessä niiden pääte on .tar.gz tai .tgz.

MacIntosh-koneet ovat sitten oma lukunsa. Niissä pakkaukseen käytetään esimerkiksi StuffIt-ohjelmaa, joka tuottaa .sit-muodossa olevia tiedostoja [29]. Toinen yleinen MacIntosh-koneissa käytetty pakkausohjelma on Macbinary, joka tuottaa .bin-päätteisiä tiedostoja.

Taulukko 5.6: Muutama esimerkki eri pakkausohjelmien sisältämistä algoritmeista, jotka on koottu eri Internet-sivustoilta. Katso lyhenteiden selitys sanastosta.

Ohjelma	Kehittäjä	Algoritmi
ARC	System Enhancement Associates	Huffman
ARJ	ARJ Software	LZSS+Huffman
Compress	Unisys	LZW
GZip	Gailly, J-L.	LZ77
JAR	ARJ Software	LZSS+Huffman
PKARC	PKWARE Inc.	Huffman
PowerArchiver	ConexWare Inc	LZMA
RK	Taylor, M.	LZ+PPMZ
7-Zip	Pavlov, I.	LZMA+PPMII+LZ77+BWT
WinRAR	Roshal, E.	LZ77+PPMII+Huffman
WinZip	WinZip Computing	LZH+LZW+SF+Huffman+PPMd

Tiina Vekkilä [31] käsitteli opinnäytetyössään digitaalisen tiedon pakkaamista ja sen etuja yritykselle. Lisäksi hän tutki eri multimediaelementtejä ja testasi niiden pakkaamista.

Taulukossa 5.7 on vertailtu eri pakkausohjelmien tehokkuuksia laskemalla pakkaussuhde kullekin pakkausohjelmalle [31]. Pakkaussuhde on itse asiassa laskettu pakkaustekijän kaavalla 3.2, joten alkuperäisestä taulukosta poiketen pakkaussuhteen tilalla käytetään tässä työssä termiä pakkaustekijä (*CF*). Pakkausohjelmiksi valikoitui maksullisten ohjelmien kokeiluversioita, ilmaisohjelmia sekä Windows XP- ja Vista-käyttöjärjestelmien sisältämä pakkaustoiminto. Pakattavina tiedostoina olivat 100 kappaletta JPEG-kuvia, 10 kappaletta TIFF-kuvia, Adobe Photoshop 7.0 -ohjelmisto sekä 13 kappaletta PDF-tiedostoja. Taulukossa tiedostojen nimet on lyhennetty.

Taulukosta voidaan havaita, että JPEG-kuvien pakkaus ei onnistunut kovinkaan hyvin käytetyillä menetelmillä, TIFF-kuvien pakkaus onnistui taas paremmin. Käytetyistä pakkausohjelmista 7-Zip pakkasi tiedostot kaikista parhaiten.

Taulukko 5.7: Pakkausohjelmien tehokkuus, kun koon yksikkönä 1 Mt [31, s. 6].

Tiedostot	Koko	WinZip	WinRAR	PowerArchiver	7-Zip	IZArc	Win
100 JPEG	76.6	76.4	76.6	76.5	75.9	76.5	76.5
CF		**1.00**	**1.00**	**1.00**	**1.01**	**1.00**	**1.00**
10 TIFF	83.9	66.5	38.4	66.6	51.9	66.6	66.6
CF		**1.26**	**2.18**	**1.26**	**1.62**	**1.26**	**1.26**
PS 7.0	132	72.5	62.4	72.3	52.7	72.5	72.6
CF		**1.82**	**2.12**	**1.83**	**2.50**	**1.82**	**1.82**
13 PDF	68.7	42.9	41.0	43.0	40.0	43.0	43.0
CF		**1.60**	**1.68**	**1.60**	**1.72**	**1.60**	**1.60**

5.3 Suoritusnopeus

Kun tiedon pakkaamista käytetään tiedon lähetykseen liittyvissä sovelluksissa, päämääränä on menetelmän nopeus [14]. Lähetyksen nopeus riippuu lähetettävien bittien määrästä, enkooderin koodaukseen käyttämästä ajasta lähettäjäpäässä ja dekooderin purkamiseen käyttämästä ajasta vastaanottajapäässä. Vaikka menetelmän vaatimuksena tiedon säilytyssovelluksissa on ennen kaikkea pakkauksen määrä, on menetelmän nopeudellakin merkitystä, kun arvioidaan menetelmän käyttökelpoisuutta.

Witten ym. [35] vertasi adaptiivisen aritmeettisen ja adaptiivisen Huffmanin menetelmien keskinäisen tehokkuuden lisäksi myös niiden enkoodaukseen ja dekoodaukseen käyttämää aikaa, joista tulokset on esitetty taulukossa 5.8. Taulukossa aritmeettisen menetelmän enkoodaukseen käyttämä aika löytyy sarakkeesta A_{t_e} ja dekoodaukseen käyttämä aika löytyy sarakkeesta A_{t_d}. Vastaavat ajat Huffmanin menetelmälle löytyvät sarakkeista H_{t_e} ja H_{t_d}. Taulukosta voidaan todeta, että aritmeettinen menetelmä oli Huffmanin menetelmää parempi myös suoritusaikojen osalta. Aritmeettisen menetelmän enkoodaukseen ja dekoodaukseen käyttämä aika oli nimittäin noin puolet Huffmanin menetelmän käyttämästä ajasta.

Taulukko 5.8: Adaptiivisten aritmeettisen ja Huffmanin algoritmien vertailua suoritusnopeuden osalta [35]. Suoritusnopeudet on ilmaistu mikrosekunteina (μs).

	A_{t_e}	A_{t_d}	H_{t_e}	H_{t_d}
Teksti	214	262	550	414
C-ohjelma	230	288	596	441
VAX-objektiohjelma	313	406	822	606
Aakkosto	223	277	598	411
Vääristymätilasto	143	170	215	132

Vekkilä [31] vertasi opinnäytetyössään pakkausohjelmien keskinäisen tehokkuuden lisäksi niiden keskinäistä suoritusaikaa. Taulukkoon 5.9 on kerätty käytettyjen pakkausohjelmien pakkaukseen käyttämä aika. Taulukosta voi todeta, että nopeimmin pakkasivat WinZip, PowerArchiver ja IZArc, kun taas 7-Zip, joka tehokkuusvertailussa selvisi ykköseksi, suoriutui pakkauksesta muita hitaammin.

Taulukko 5.9: Pakkausohjelmien nopeus, kun ajan yksikkönä on minuutti [31, s. 6].

Tiedostot	WinZip	WinRAR	PowerArchiver	7-Zip	IZArc	Win
100 JPEG	0:16	1:14	0:14	1:47	0:28	0:11
10 TIFF	0:17	0:52	0:19	2:24	0:30	0:14
PS 7.0	0:38	1:45	0:28	2:52	0:44	0:19
13 PDF	0:13	1:22	0:11	1:15	0:21	0:09

5.4 Virheiden sietokyky

Kuten aikaisemmin on todettu, häviöttömät pakkausmenetelmät liittyvät diskreettiin häiriöttömään kommunikaatiojärjestelmään, jolloin oletetaan, että tiedonsiirtokanavassa ei esiinny minkäänlaista häiriötä ja että itse tiedonsiirrossa ei aiheudu tiedon menetystä. Tällainen malli ei ole kovinkaan realistinen, sillä todelliset tiedonsiirtojärjestelmät ovat alttiita kahdenlaisille virheille: vaihevirheille (*phase error*) ja taajuusvirheille (*amplitude error*)[17]. Vaihevirheet ilmenevät siten, että koodisymboli joko häviää tai monistuu. Taajuusvirheissä puolestaan koodisymbolit voivat kor-

ruptoitua. Seikka, missä määrin kanavassa esiintyvät virheet huonontavat tiedon lähetystä, on tärkeä piirre tiedon pakkausmenetelmän valinnassa. Pakkausalgoritmin alttius virheille riippuu mitä suurimmassa määrin siitä, onko menetelmä staattinen vai adaptiivinen.

Huffmanin koodit ovat itsestään korjautuvia, mikä tarkoittaa sitä, että lähetysvirheet eivät etene liian pitkälle [14]. Toisessa päässä vastaanotetaan jonkin aikaa virheellisiä koodeja, mutta ennen pitkää vastaanottajapää synkronoi lähettäjäpään kanssa. Synkronointi staattisissa menetelmissä tarkoittaa sitä, että sekä lähettäjä että vastaanottaja tunnistaa koodisanan alun samaan tapaan.

Taulukossa 5.10 on esimerkki Huffmanin koodin kyvystä toipua vaihevirheistä. Esimerkkinä on merkkijono *BCDAEB*, joka on koodattu Huffmanin staattisella menetelmällä. Taulukon vasen sarake näyttää koodin, jossa kunkin symbolin osuus on eroteltu pisteellä. Oikeassa sarakkeessa näkyy koodia vastaava merkkijono ennen virhettä sekä 1, 2 ja 4 bitin häviämisen jälkeen. Ensimmäisen bitin häviäminen johtaa uudelleensynkronointiin kolmannen bitin jälkeen siten, että merkkijonosta häviää vain sen ensimmäinen symboli *B* korvautuen symboliparilla *AA*. Kun toinen bitti häviää, koodin 8 ensimmäistä bittiä tulkitaan väärin, jolloin synkronisointiin päästään 9:stä bitistä alkaen. Neljännen bitin häviäminen aiheuttaa saman kuin toisen bitin häviäminen.

Taulukko 5.10: Huffmanin koodin toipuminen vaihevirheestä [14].

Koodi	Merkkijono
011.010.001.1.000.011.	*BCDAEB*
1.1.010.001.1.000.011.	1 bitti hävinnyt, *AACDAEB*
010.1.000.1.1.000.011.	2 bitti hävinnyt, *CAEAAEB*
011.1.000.1.1.000.011.	4 bitti hävinnyt, *BAEAAEB*

Taulukossa 5.11 on esimerkki Huffmanin koodin kyvystä toipua taajuusvirheistä. Esimerkkinä on nytkin merkkijono *BCDAEB*, joka on koodattu Huffmanin staattisella menetelmällä. Taulukon selitys on muuten sama kuten edellä, mutta tässä tapauksessa 1, 2 ja 4 bitit ovat vaihtuneet vastabiteikseen. Kun ensimmäinen tai toinen bitti vaihtuvat, vain kolme ensimmäistä bittiä, ts. merkkijonon ensimmäinen symboli, häiriytyvät. Neljännen bitin vaihtuminen aiheuttaa synkronisoinnin 9:stä bitistä alkaen.

Taulukko 5.11: Huffmanin koodin toipuminen taajuusvirheestä [14].

Koodi	Merkkijono
011.010.001.1.000.011.	*BCDAEB*
1.1.1.010.001.1.000.011.	1 bitti vaihtunutt, *DCDAEB*
001.010.001.1.000.011.	2 bitti vaihtunut, *AAACDAEB*
011.1.1.000.1.1.000.011.	4 bitti vaihtunut, *BAAEAAEB*

Edellä käsitellyt esimerkit ovat esimerkkejä Huffmanin koodin taipumuksesta itsestään synkronointiin (*self-synchronizing*), mutta näin ei aina tarvitse olla [14]. Vaikka itsestään synkronointia tapahtuu, se ei silti estä virheen syntymistä. Ongelmallista on myös se, että itsestään synkronointi ei takaa, että virhettä ei olisi tapahtunut.

Adaptiiviset menetelmät ovat alttiimpia lähetysvirheille kuin staattiset menetelmät [14]. Synkronisointiin adaptiivisten menetelmien tapauksessa ei riitä pelkästään merkkijonon koodin synkronointi, vaan siinä tarvitaan myös koodauksen takana vaikuttavan mallin synkronointia. Adaptiivisten menetelmien itsestään synkronointitaipumuksesta ei ole todisteita. Joka tapauksessa ei staattiset eikä liioin adaptiivisetkaan menetelmät siedä virheitä.

6 Yhteenveto ja johtopäätökset

Informaatioteorian juuret ulottuvat moniin eri tieteenaloihin. Sen katsotaan syntyneen 1940-luvun lopulla, jolloin amerikkalaisen Claude Shannonin kaksiosainen artikkeli *A Mathematical Theory of Communication* julkaistiin Bell System Technical Journal -julkaisusarjassa. Shannon esitteli artikkelissaan ensi kertaa tiedonsiirtojärjestelmän rakenteen ja osoitti, että virheetöntä signaalia välittävän kanavan suorituskyky on rajallinen. Lisäksi hän määritteli informaation käsitteen ja määrän sekä esitti, kuinka tieto voidaan esittää perinteisen analogisen esitystavan lisäksi myös digitaalisessa muodossa

Shannonin kaksiosainen artikkeli sisältää tiedon pakkaamisen kannalta kaksi merkittävää informaatioteoreettista perustulosta: lähdekooditeorian (*Source Coding Theory*) ja nk. määrähäiriöteorian (*Rate Distortion Theory*). Lähdekooditeoria tarkastelee tapaa esittää tieto mahdollisimman tehokkaalla tavalla siten, että tiedonsiirtokanavan kapasiteetti eli suorituskyky ei ylittyisi. Määrähäiriöteoriassa puolestaan ollaan kiinnostuttu siitä, missä olosuhteissa lähteen tuottama informaatio saadaan palautettua vastaanottajapäässä siten, että palautettu informaatio sisältäisi häiriötä mahdollisimman vähän. Lähdekooditeoria synnytti häviöttömät tiedonpakkausmenetelmät ja määrähäiriöteoria häviölliset tiedonpakkausmenetelmät.

Häviöttömien tiedonpakkausmenetelmien kohdalla oletetaan, että tiedonsiirtokanavassa ei esiinny häiriötä, jonka seurauksena pakattu tieto pystytään palauttamaan täysin alkuperäiseen tilaansa. Häviöttömillä pakkausmenetelmillä voidaankin pakata sellaista tietoa, joka ei siedä tiedon häviämistä, esimerkkinä teksti. Häviöllisissä menetelmissä pakattuun tietoon siirtyy tiedonsiirtokanavassa esiintyvää häiriötä, jolloin tietoa ei pystytä palauttamaan täysin alkuperäiseen tilaansa. Häviöllisillä menetelmillä voidaankin pakata sellaista tietoa, jossa tiedon häviäminen ei ole ongelma, kuten kuvia.

Tässä työssä on tutkittu neljää häviötöntä tiedonpakkausmenetelmää: Shannon-Fano-menetelmää, Huffmanin menetelmää, aritmeettista menetelmää ja Lempel-Ziv-menetelmää, joista kolme ensin mainittua kuuluvat nk. tilastollisiin menetelmiin ja jälkimmäinen hakemistomenetelmiin. Menetelmistä on ensin selvitetty perusalgoritmin toiminta, minkä jälkeen menetelmiä on verrattu keskenään. Vertailun lähtö-

kohdaksi on valittu algoritmin tehokkuus, lähinnä pakkauksen määrä ja suoritusnopeus. Lisäksi on tarkasteltu lyhyesti myös virheensietokykyä.

Menetelmien keskinäistä vertailua hankaloittaa sellaisen aineiston vähyys, jossa olisi huomioitu kaikki tässä työssä käsitellyt menetelmät. Tutkituissa lähteissä on vertailtu joko saman menetelmän eri versioita toisiinsa tai kahta eri menetelmää toisiinsa. Tässä kirjallisuustutkimuksessa käsitellyn aineiston pohjalta ei voi vetää suoranaisia johtopäätöksiä tutkittujen menetelmien paremmuudesta, vaikka tutkimukset viime vuosikymmeninä ovatkin painottuneet Lempelin ja Zivin menetelmään. Algoritmien tehokkuus kun on viime kädessä riippuvainen käsiteltävästä aineistosta [14]. Jatkotutkimuksen aiheena voisikin olla menetelmien tarkempi analysointi. Menetelmät voisi ohjelmoida C-kielisenä tai muulla sopivalla koodilla, jonka avulla testattaisiin toisistaan poikkeavien tietotyyppien enkoodausta ja dekoodausta. Tietotyypeistä voisi huomioida myös häviöllisillä menetelmillä pakattavat tietotyypit kuten kuvatiedostot. Huomiota voisi kiinnittää pakkauksen määrän lisäksi erityisesti myös suoritusnopeuteen.

Lähteet

[1] AFTAB, O., CHEUNG, P., KIM, A., THAKKAR, S., JA YEDDANAPUDI, P. Information theory and the digital age, 2001. URL: http://mit.edu/6.933/www/Fall2001/Shannon2.pdf, luettu 25.6.2011.

[2] BELL, T. C., CLEARY, J. G., JA WITTEN, I. H. *Text Compression*. Englewood Cliffs (N.J.): Prentice-Hall, 1990.

[3] BERGER, T. *Rate Distortion Theory: A Mathematical Basis for Data Compression*. Englewood Cliffs (N.J.): Prentice-Hall, 1971.

[4] COVER, T. M., JA THOMAS, J. A. *Elements of Information Theory*. New York: John Wiley and Sons, 2nd ed., 2006.

[5] DATA-COMPRESSION.COM. Theory of data compression, 2000-2015. URL: http://www.data-compression.com/theory.shtml, viitattu 13.11.2014.

[6] FALLER, N. An adaptive system for data compression. *Record of the 7th Asilomar Conference on Circuits, Systems and Computers* (1973), 593–597.

[7] FANO, R. M. Transmission of information. Tekninen raportti 65, The research Laboratory of electronics, M.I.T., 1949.

[8] GALLAGER, R. G. Variations on a theme by Huffman. *IEEE Transactions on Information Theory 24*, 6 (1978), 668–674.

[9] HARTLEY, R. V. L. Error detecting and error correcting codes. *Bell System Technical Journal 7*, 3 (1928), 535–563.

[10] HUFFMAN, D. A. A method for construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[11] KERÄNEN, V., LAMBERG, N., JA PENTTINEN, J. *Digitaalinen media*. Jyväskylä: Docendo, 2005.

[12] KNUTH, D. E. Dynamic Huffman coding. *Journal of Algorithms 6*, 2 (1985), 163–180.

[13] LANGDON, G. G. An introduction to arithmetic coding. *IBM Journal of Research and Development 28*, 2 (1984), 135–149.

[14] LELEWER, D. A., JA HIRSCHBERG, D. S. Data compression. *ACM Computing Surveys 19*, 3 (1987), 261–296.

[15] MOFFAT, A., NEAL, R. M., JA WITTEN, I. H. Arithmetic coding revisited. *ACM Transactions on Information Systems 16*, 3 (1998), 256–294.

[16] NELSON, M., JA GAILLY, J.-L. *The Data Compression Book*. New York: M&T Books, 2nd ed., 1996.

[17] NEUMANN, P. G. Efficient error-limiting variable-length codes. *IRE Transactions on Information Theory 8*, 4 (1962), 292–304.

[18] NYQUIST, H. Certain factors affecting telegraph speed. *Bell System Technical Journal 3*, 2 (1924), 324–346.

[19] PHILLIPS, D. Generalized kraft inequality and arithmetic coding. *The Computer Applications Journal Circuit Cellar Ink. 27* (1992), 36–48.

[20] RISSANEN, J. J., JA LANGDON, G. G. Arithmetic coding. *IBM Journal of Research and Development 23*, 2 (1979), 149–162.

[21] SALOMON, D. *A Guide to Data Compression Methods*. New York: Springer-Verlag, 2002.

[22] SALOMON, D. *A Concise Introduction to Data Compression*. London: Springer-Verlag, 2008.

[23] SAYOOD, K. *Introduction to Data Compression*. California: Morgan Kaufmann Publisher, 3rd ed., 2006.

[24] SHANNON, C., JA WIEVER, W. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[25] SHANNON, C. E. A mathematical theory of communication (part 1). *Bell System Technical Journal 27*, 3 (1948), 379–423.

[26] SHANNON, C. E. A mathematical theory of communication (part 2). *Bell System Technical Journal 27*, 4 (1948), 623–656.

[27] SHANNON, C. E. Coding theorems for a discrete source with a fidelity criterion. *IRE National Convention Record*, Pt. 4 (1959), 142–163.

[28] STORER, J. A., JA SYZMANSKI, T. G. Data compression via textual substitution. *Journal of the ACM 29*, 4 (1982), 928–951.

[29] SUORANTA, L. Pakkauksen salat. *MikroPC*, 7 (2002), 60–63.

[30] TUOMINEN, P. *Todennäköisyyslaskenta 1*. Helsinki: Limes ry, 5. painos, 2000.

[31] VEKKILÄ, T. Digitaalisen tiedon pakkaaminen, 2007. URL: http://www.theseus.fi/bitstream/handle/10024/11929/2007-12-03-19.pdf?sequence=1, viitattu 19.09.2014.

[32] VERDÚ, S. Fifty years of Shannon theory. *IEEE Transactions on Information Theory 44*, 6 (1998), 2057–2078.

[33] VITTER, J. S. Design and analysis of dynamic Huffman codes. *Journal of the ACM 34*, 4 (1987), 148–152.

[34] WELCH, T. A. A technique of high-performance data compression. *Computer 17*, 1 (1984), 8–19.

[35] WITTEN, I. H., NEAL, R. M., JA CLEARY, J. G. Arithmetic coding fod data compression. *Communications of the ACM 30*, 6 (1987), 148–152.

[36] ZIV, J., JA LEMPEL, A. A universal algorithm for data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.

[37] ZIV, J., JA LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5 (1978), 530–536.