GIT-VERSIOHALLINNAN MATEMAATTISET PERUSTEET

Tuomas Suutari

Pro gradu -tutkielma
Marraskuu 2012

MATEMATIIKAN JA TILASTOTIETEEN LAITOS
TURUN YLIOPISTO

TURUN YLIOPISTO
Matematiikan ja tilastotieteen laitos

SUUTARI, TUOMAS: Git-versiohallinnan matemaattiset perusteet
Pro gradu -tutkielma, 79 s.
Matematiikka
Marraskuu 2012

Työssä esitetään Git-versiohallintajärjestelmään liittyviä tietorakenteita ja toimintoja matemaattisesta näkökulmasta. Kuvaillaan Gitin käyttämä tietojen tallennustapa ja annetaan yleiskuva Gitin tärkeimmistä toiminnoista. Erityisen tarkasti Gitin toiminnoista esitetään pakkausmenetelmä, tiedostojen erojen vertailu ja pakettitiedostoissa käytettävä deltapakkaus.

Deflate-pakkausmenetelmästä tutustutaan sen käyttämään Huffman-koodaukseen, LZ77-koodaukseen ja koodauskaavioiden pakkaukseen. Lisäksi määritellään deflate-pakatun tietovirran rakenne.

Esitetään tiedostojen erojen vertailun matemaattinen määritelmä sekä näytetään miten tähän liittyvä pisimmän yhteisen alijonon hakeva algoritmi voidaan toteuttaa erilaisilla menetelmillä, joiden aikakompleksisuudet poikkeavat merkittävästi toisistaan.

Kuvaillaan Gitin pakettitiedoston rakenne ja sen muodostamisen algoritmeja. Lisäksi annetaan matemaattinen määritelmä siinä käytetylle deltapakkaukselle ja esitetään deltapakkauksen algoritmi ja siinä käytetty Rabinin sormenjälki.

Esitettävissä algoritmeissa esiintyy muutamia perusmenetelmiä kuten dynaaminen ohjelmointi, ahnas algoritmi sekä hajota ja hallitse -menetelmä.

Asiasanat: Git, versiohallinta, Huffman-koodaus, LZ77-koodaus, deflate-pakkaus, tiedostojen erojen vertailu, muokkausgraafi, pakettitiedosto, deltapakkaus, Rabinin sormenjälki.

Sisältö

Algoritmit

Kuvat

Johdanto

Lähdin tutkimaan Git-versiohallinnan taustalla olevaa matematiikkaa, koska tiesin Gitin toimintaperiaatteen perustuvan hyvin yksinkertaiseen matemaattisesti kiinnostavaan malliin. Yksinkertaisuudessaan se tallentaa tiedot objekteiksi, jotka muodostavat suunnatun syklittömän graafin (directed acyclic graph, DAG) ja viittaukset objekteihin tapahtuvat tunnisteilla, jotka muodostetaan kryptografisella hash-funktiolla objektin sisällöstä. Lisäksi minua kiehtoi Gitin kyky pakata suurenkin projektin koko versiohistoria hyvin pieneen tilaan.

Työssä selvitetään millaista matematiikkaa tarvitaan versiohallintajärjestelmän rakentamiseen käyttäen esimerkkinä Gitin toteutusta. Tässä tutkielmassa on rajoituttu tutkimaan vain pientä osaa versiohallintaan liittyvästä matemaattisesta perustasta, koska aihe on kokonaisuudessaan hyvin laaja. Pyritään vastaamaan kysymykseen, että mihin tuo Gitin tehokas projektin versiohistorian pakkaus perustuu. Tätä lähestytään käsittelemällä pakkausmenetelmiä yksittäisen objektin pakkaamisen kannalta ja toisaalta versiohallinnalle ominaisen objektin useamman eri version pakkaamisen kannalta. Lisäksi tutkitaan objektien erojen vertailuun liittyviä algoritmeja, jotka ovat versiohallinnassa erittäin keskeisessä asemassa.

Kappaleessa 1 esitetään työssä käytetyt merkinnät ja kappaleessa 2 yleiskuva Gitistä: Kerrotaan Gitin ominaisuuksista ja esitetään sen käyttämän tietovaraston rakenne sekä millaisia toimintoja Gitillä tavallisesti tehdään.

Kappaleessa 3 esitetään Gitin käyttämän deflate-pakkausmenetelmän toiminta. Osoitetaan, että siinä käytetty Huffman-koodaus on optimaalinen sekä kuvaillaan LZ77-koodaus. Myös deflate-pakatun tietovirran rakenne ja koodauskaavioiden pakkaus esitellään. Gitin käyttämä tiedostojen erojen vertailun algoritmi esitetään kappaleessa 4 lähtien ilmeisemmistä, mutta hitaammista perusalgoritmeista. Kappaleessa 5 esitetään Gitin käyttämän pakettitiedoston rakenne ja muodostamistavat sekä näytetään miten siinä käytetty Rabinin sormenjälki voidaan laskea nopeasti.

Työn lopussa johtopäätösten jälkeen on sanasto, johon on kerätty suurin osa työssä käytetyistä termeistä suomennetussa muodossa sekä alkuperäisessä englanninkielisessä muodossaan. Sanastossa on termien esiintymisten sivunumerot, joten sitä voi käyttää myös hakemistona.

1 Merkinnät

Esitetään seuraaville kappaleille yhteiset merkinnät.

1.0.1 Merkintä. Kun A on joukko ja $a_1, a_2, \ldots, a_n \in A$, niin jono $a_1 a_2 \cdots a_n$ on *sana*, jonka *aakkosto* on A. Kaikkien aakkoston A sanojen joukkoa merkitään

$$A^* = \{\, a_1 a_2 \cdots a_n \mid a_1, a_2, \ldots, a_n \in A, n \geq 0 \,\},$$

epätyhjien sanojen joukkoa

$$A^+ = \{\, a_1 a_2 \cdots a_n \mid a_1, a_2, \ldots, a_n \in A, n \geq 1 \,\}$$

ja n-pituisten sanojen joukkoa

$$A^n = \{\, a_1 a_2 \cdots a_n \mid a_1, a_2, \ldots, a_n \in A \,\}.$$

1.0.2 Merkintä. Sanan $a \in A$ pituutta merkitään $|a|$. Tyhjää sanaa merkitään symbolilla ε. Siis $|\varepsilon| = 0$. Jos halutaan korostaa, että kyseessä on nimenomaan aakkoston A tyhjä sana, niin käytetään merkintää ε_A.

1.0.3 Merkintä. Sanoista x ja y yhdistettyä sanaa merkitään xy.

1.0.4 Merkintä. Olkoon sana $w = w_1 w_2 \cdots w_n \in A^*$, missä $w_i \in A$ kaikilla $i = 1, \ldots, n$. Sanan w osasanalle $w_a w_{a+1} \cdots w_b$ käytetään merkintää $w[a \ldots b]$ ja sen yksittäiselle merkille w_k käytetään merkintää $w[k]$. Toisinaan käytetään myös lyhennysmerkintöjä $w[a \ldots] = w[a \ldots |w|]$ ja $w[\ldots b] = w[1 \ldots b]$. Sovitaan lisäksi, että jos $b > |w|$, niin $w[a \ldots b] = w[a \ldots]$, tai jos $b < a$, niin $w[a \ldots b] = \varepsilon$.

1.0.5 Merkintä. Joukon $\mathcal{B} = \{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{255}\}$ alkioita sanotaan *tavuiksi*. *Tavujono* on tavujen muodostama sana eli joukon $\mathcal{B}^*$ alkio.

Kun tavujonoa halutaan tulkita tekstinä eli merkkien jonona, niin tarvitaan koodaus, joka kertoo miten tavujono muunnetaan merkkijonoksi. Tämän työn kannalta käytetty koodaus ei ole olennainen muuta kuin muutamien lyhyiden merkkijonojen esittämisessä. Otetaan niitä varten käyttöön seuraava merkintä.

1.0.6 Merkintä. Tekstin koodausta tavuiksi merkitään laittamalla koodattava teksti `"tasalevyiseksi ja lainausmerkkeihin"`. Käytetty koodaus on ASCII [Net+69].

1.0.7 Merkintä. Joukon $\beta = \{\bar{0}, \bar{1}\}$ alkioita sanotaan biteiksi.

1.0.8 Merkintä. Olkoon $x \in \mathbb{R}$ reaaliluku. Luvun x itseisarvo on $|x|$. Pienin kokonaisluku, joka on suurempi tai yhtä suuri kuin x, on $\lceil x \rceil$. Suurin kokonaisluku, joka on pienempi tai yhtä suuri kuin x, on $\lfloor x \rfloor$.

1.0.9 Merkintä. Äärellisen joukon A alkioiden lukumäärää merkitään $|A|$.

2 Git

Git on hajautettu versiohallintajärjestelmä (distributed version control system, DVCS), joka on suunniteltu pienien ja suurien projektien tarpeisiin. Git on avoimen lähdekoodin (open source) ohjelma, joten se on helposti kaikkien saatavilla ilmaiseksi ja sen toimintaa on helppo tutkia. [Git12a]

Git on helppo oppia ja sen toiminnot ovat erittäin nopeita. Se on useimmissa toiminnoissa jopa kymmeniä kertoja nopeampi kuin suosittu Subversion-versiohallintajärjestelmä [Git12d]. Koska Git on hajautettu, se antaa enemmän valinnanvaraa versiohallinnan käyttötapojen suhteen kuin perinteinen keskitetty versiohallinta [Git12a]. Git erottuu edukseen erityisesti uusien kehityshaarojen luomisen ja yhdistämisen helppoudessa [Git12b]. Lisäksi Git tallentaa tiedot tavalla, joka takaa tietojen eheyden. Tietojen muuttaminen muuttamatta tietoon osoittavia tunnisteita on erittäin vaikeaa, sillä tiedot on suojattu kryptografisella hash-funktiolla [Git12c].

Linus Torvalds suunnitteli ja toteutti Gitin ensimmäiset versiot alun perin Linux-ytimen lähdekoodin hallintatarpeisiin huhtikuussa 2005. Kun hän oli heinäkuussa 2005 saanut järjestelmän tarpeeksi valmiiksi teknisten käyttäjien tarpeisiin, hän luovutti sen ylläpidon Junio Hamanolle. Hamano oli vastuussa Gitin 1.0-version julkaisusta joulukuussa 2005 ja toimii Gitin ylläpitäjänä edelleen vuonna 2012. [Wik12c]

2.1 Versiohallinnan tarkoitus

Versiohallintajärjestelmän tarkoitus on mahdollistaa tiedostojen muutosten hallinta. Sen tärkein tehtävä on tallentaa tiedostoista kaikki sille osoitetut versiot sekä mahdollistaa pääsy näihin tarvittaessa. Muita tyypillisiä versiohallinnan perustoimintoja ovat uusien ja vanhojen tiedostoversioiden vertailu sekä kaikkien tehtyjen muutosten listaaminen. [Cha11a, Kohta ”About Version Control”]

2.2 Tietovarasto

Git tallentaa tiedostojen eri versiot tietovarastoonsa (repository) objekteina. Jokaisella objektilla on oma tunnisteensa, jota käytetään objektiin viitatessa.

Eri versioita ei kuitenkaan tallenneta objekteja muuttamalla vaan objektit ovat sisällöltään muuttumattomia (immutable). [Wik12c]

Objekteja tallennetaan Gitin tietovarastoon kahdella eri tavalla: irrallisina objekteina (loose object) ja pakettitiedostoihin (packfile). Irralliset objektit pakataan ja tallennetaan objektin tunnisteen perusteella määräytyvään sijaintiin, josta Git löytää ne tarvittaessa nopeasti. [Cha11b]

Pakkausmenetelmästä on kerrottu tarkemmin kappaleessa "3 Pakkausmenetelmä" ja pakettitiedostosta kappaleessa "5 Pakettitiedosto".

Gitin tietovarastossa on objektien lisäksi ns. valmistelualue (staging area), viittauksia (reference, ref), viittauksien muutoslokeja sekä asetustiedostoja. Lisäksi tietovarastoon on mahdollista lisätä eri toimintojen yhteyteen liitettäviä komentojonoja (hook script). [Cha11b]

Gitin tietovarasto sijaitsee yleensä "`.git`"-nimisessä hakemistossa työskentelyhakemistossa (working tree) [Wik12c]. Työskentelyhakemiston merkityksestä on kerrottu tarkemmin kohdassa 2.5.

2.3 Objektit

Git tallentaa historian objekteilla, joita on neljää eri tyyppiä. Ne ovat nimeltään *blob*, *tree*, *commit* ja *tag*. Kaikilla objekteilla on tavujonomuoto ja tunniste, joka muodostetaan objektin tyypin ja tavujonomuodon perusteella SHA1-hash-funktiolla.

Esitetään seuraavaksi objektin tunnisteen tarkka määritelmä ja objektityyppien kuvaukset. Esitys perustuu lähteisiin [Cha11b, Kohta "Git Objects"] ja [Duy12]. Objektityyppien mahdollistavien rakenteiden ymmärtämisen helpottamiseksi on Kuvassa 2.1 (sivulla 8) esitetty esimerkki yhdestä mahdollisesta Git-tietovaraston objektien muodostamasta graafista.

2.3.1 Objektin tunniste

Merkitään objektin o tavujonomuotoa symbolilla $o_S \in \mathcal{B}^*$ ja sen tyypin nimestä ASCII-koodauksella muodostettua tavujonoa symbolilla o_T. Siis

$$o_T \in \{\texttt{"blob"}, \texttt{"tree"}, \texttt{"commit"}, \texttt{"tag"}\} \subset \mathcal{B}^*.$$

Olkoon sha1: $\mathcal{B}^* \to \beta^{160}$ *SHA1-hash-funktio*, joka siis kuvaa jokaisen tavujonon 160-bittiseksi bittijonoksi. SHA1-funktion tarkka määritelmä löytyy esimerkiksi lähteestä [NIST95].

Objektin o tunniste on

$$\text{sha1}(o_T \mathbf{b}_{32} \text{ascnum}(|o_S|) \mathbf{b}_0 o_S),$$

missä $\mathbf{b}_{32}$ on välilyöntimerkki, $\mathbf{b}_0$ on nollatavu ja $\text{ascnum}(|o_S|) \in \mathcal{B}^*$ on tavujonomuodon pituuden kymmenkantainen esitys ASCII-koodattuna tavujonoksi. Objektin tunniste esitetään usein 40-merkkisessä heksadesimaalimuodossa. [Cha11b]

2.3.1 Esimerkki. Muodostetaan tunniste blob-objektille, jonka tavujonomuoto on `"versiohallinta"`. Tunniste on siis

$$\text{sha1}(\texttt{"blob 14"}\mathbf{b}_0\texttt{"versiohallinta"}) = \bar{0}\bar{1}\bar{1}\bar{1}\bar{0}\bar{0}\bar{0}\bar{1} \cdots \bar{0}\bar{1}\bar{1}\bar{0}\bar{0}\bar{0}\bar{0} \in \beta^{160}.$$

Kun tämä 160 bittiä pitkä bittijono tulkitaan binäärilukuna ja esitetään kyseinen luku vielä heksadesimaalimuodossa, saadaan

7862f963427073616ae34ad216d9f82521e2db60.

2.3.2 Blob-objekti

Objektityypeistä yksinkertaisin on blob. Se on Gitin kannalta vain tavujono ilman rakennetta. Tiedostojen sisällöt tallennetaan blob-objekteina.

2.3.3 Tree-objekti

Tree-objektilla tallennetaan hakemistopuun yksi taso. Se sisältää joukon tietueita, jotka viittaavat blob-, tree- tai commit-objekteihin. Jokaisessa tietueessa on kolme tietoa: moodi, objektin tunniste ja nimi.

Moodin arvo kertoo viitataanko tiedostoon, symboliseen linkkiin, hakemistoon vai alimoduuliin (submodule). Lisäksi ajettavilla tiedostoilla on eri moodi kuin tavallisilla tiedostoilla. Tiedostot ja symboliset linkit tallennetaan blob-viittauksina, hakemistot tree-viittauksina ja alimoduulit commit-viittauksina.

Tree- ja blob-objektien avulla on siis mahdollista tallentaa hakemiston sisältö alihakemistoineen ja tiedostoineen muodostamalla blob-objektit jokaiselle tiedostolle ja tree-objektit jokaiselle hakemistopuun tasolle lehdistä juureen.

2.3.4 Commit-objekti

Kun Git-tietovarastoon tallennetun projektin tiedostojen ja hakemistojen tilasta on tallennettu vedos (snapshot) tree- ja blob-objekteilla, niin viittaus juuren tree-objektiin voidaan tallentaa commit-objektilla. Commit-objektiin tallennetaan lisäksi viesti ja tiedot muutosten tekijästä (author) ja tallentajasta (committer) sähköpostiosoitteiden ja aikaleimojen kera sekä viittaukset commit-objektin vanhempiin (parent).

Commit-objektin vanhemmat ovat commit-objekteja, jotka edeltävät kyseistä commit-objektia. Tyypillisesti commit-objektilla on yksi vanhempi, mutta ns. juuri-commitilla (root commit) ei ole yhtään vanhempaa ja kehityshaarat yhdistävällä ns. merge-committilla on kaksi vanhempaa tai enemmän.

2.3.5 Tag-objekti

Tag-objekti sisältää tiedot tagin luojasta, aikaleiman, viestin ja objektiviittauksen. Sen sisältämä objektiviittaus osoittaa yleensä commit-objektiin. Tag-objektilla on kuitenkin mahdollista viitata minkä tahansa tyyppiseen Git-objektiin – myös toiseen tag-objektiin.

Tag-objektin tärkein käyttötarkoitus on nimetä jokin tietty commit-objekti nimellä, joka on helpompi muistaa kuin tunnisteen 40-merkkinen heksadesimaaliesitys. Commit-objektin luotuaan Git tallentaa kyllä commit-objektin tunnisteen aktiivisen kehityshaaran mukaan nimettyyn viittaukseen. Tämä kehityshaaran viittaus kuitenkin muuttuu joka commitilla, joten näiden avulla pystytään viittaamaan vain kunkin kehityshaaran viimeiseen commit-objektiin. Tag-objekti viittaa aina siihen objektiin, johon se alun perin luotiin viittaamaan, joten se on luonteeltaan pysyvämpi.

Tyypillinen käyttökohde tag-objekteille on luoda julkaisuversion mukaan nimetty tag-objekti, joka viittaa kyseisen julkaisuversion commit-objektiin. Tällöin tag-objektin nimi voisi olla esimerkiksi ”v1.0”.

2.3.6 Esimerkki objektirakenteesta

Kuvassa 2.1 on esitetty Git-objektien muodostama graafi esimerkkitilanteesta.

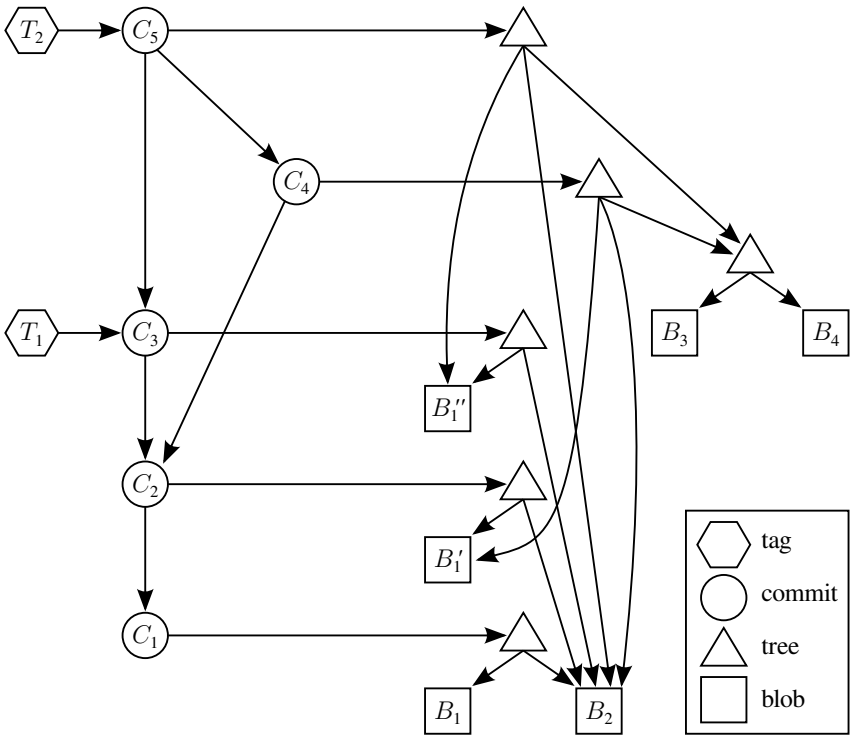
Kuvassa C_1 on ensimmäisenä tehty commit eli juuricommit. Sen osoittamassa puussa on kaksi tiedostoa, joiden sisällöt ovat tallennettu blob-objekteihin B_1 ja B_2. Minkään tiedostojen nimiä ei ole merkitty kuvaan. (Jos olisi, niin ne pitäisi merkitä tree-objekteista lähteviin nuoliin.)

Commit C_2 on tehty commitin C_1 jälkeen. Siinä on muutettu tiedostoa, jonka sisältö oli edellisessä commitissa blob-objektissa B_1. Uusi sisältö on blob-objektissa B_1'. Commitissa C_3 on tehty muutos samaan tiedostoon ja uusi blob-objekti on B_1''.

Commit C_4 on tehty pohjautuen committiin C_2 ja siinä on lisätty uusi alihakemisto, jonka sisällä on kaksi tiedostoa blob-objekteilla B_3 ja B_4.

Commitissa C_5 on yhdistetty commitit C_3 ja C_4 ja lopputuloksen tree-objekti sisältää molempien committien muutokset.

Kuvaan on merkitty myös tag-objektit T_1 ja T_2.



Kuva 2.1: Esimerkki Git-objektien muodostamasta graafista

2.4 Hash-funktiosta

2.4.1 Määritelmä. Hash-funktion h sanotaan olevan *törmäyksiä vastustava* (collision resistant), jos on käytännössä mahdotonta löytää kaksi toisistaan poikkeavaa syötettä x ja y, joille olisi $h(x) = h(y)$. [ECR11]

Gitin käyttämän hash-funktion olisi hyvä olla törmäyksiä vastustava, koska sillä muodostetaan objektiviittauksissa käytetyt tunnisteet.

SHA1-hash-funktion törmäyksiä on olemassa, sillä $\mathcal{B}^*$ on ääretön ja β^{160} on äärellinen. SHA1 on kryptografinen hash-funktio, jonka uskottiin olevan törmäyksiä vastustava, kun se julkaistiin vuonna 1995 [NIST95]. Myöhemmin siitä on kuitenkin löytynyt heikkouksia, eikä sitä enää luokitella törmäyksiä vastustavaksi [ECR11]. Yhtään törmäystä ei ole kuitenkaan vielä löydetty [ECR11]. Gitin tarpeisiin SHA1 on kuitenkin toistaiseksi riittävä, sillä Git ei käytä SHA1-funktiota sen tietoturvaominaisuuksien vuoksi vaan ainoastaan tiedon eheyden varmistamiseen satunnaisia virheitä vastaan [Tor07].

2.4.2 Määritelmä. Hash-funktio h on *yksisuuntainen*, jos annetusta kuvasta $y = h(x)$ on käytännössä mahdotonta selvittää alkukuvaa x. Tästä ominaisuudesta käytetään myös nimitystä alkukuvan vastustavuus (pre-image resistance). [ECR11]

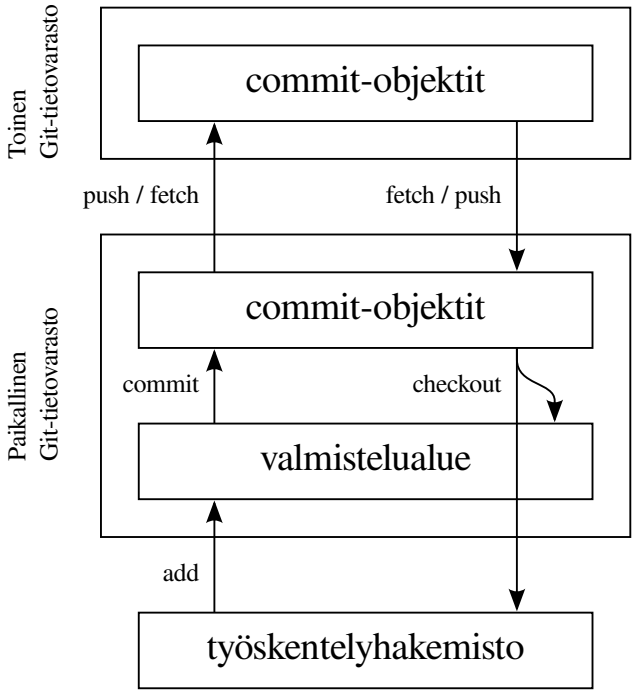
Käytetyn SHA1-hash-funktion yksisuuntaisuus aiheuttaa sen, että objektien muodostama graafi on käytännössä aina suunnattu syklitön graafi. Syklin muodostamiseksi pitäisi pystyä muodostamaan objektien ketju $o_1, o_2, \ldots, o_n$, missä kaikilla $i = 1, 2, \ldots, n-1$ objekti o_{i+1} sisältäisi objektin o_i tunnisteen ja lisäksi objekti o_1 sisältäisi objektin o_n tunnisteen. Tällaista rakennetta ei pysty kovin helposti tarkoituksellisesti muodostamaan, sillä ensimmäistä objektia muodostaessa pitäisi tietää mikä viimeisen objektin tunniste tulee olemaan vaikka se riippuu ensimmäisestä objektin sisällöstä.

2.5 Tallennustasot

Muutosten tallentaminen Git-versiohallintaan tapahtuu muokkaamalla tiedostoja ns. työskentelyhakemistossa (taso 1) ja lisäämällä näitä muutoksia sitten valmistelualueelle (taso 2). Valmistelualueelle on tallennettu muun muassa

kaikkien tiedostojen polkunimet ja viittaukset tiedostojen sisällöt tallentaviin blob-objekteihin. Muutosten lisääminen valmistelualueelle luo blob-objektin jokaisen muuttuneen tiedoston uudesta sisällöstä ja päivittää valmistelualueen viittaukset osoittamaan luotuihin blob-objekteihin. Kun valmistelualueella on halutut muutokset, niistä voidaan muodostaa commit-objekti (taso 3). Tässä yhteydessä aiemmin luodut blob-objektit kerätään hakemistorakenteeksi luomalla uusia tree-objekteja, joista juuriobjektin tunniste tallennetaan muodostettavaan commit-objektiin. Usein muodostetut commit-objektit viedään lopulta vielä toiseen Git-tietovarastoon (taso 4), joka voi sijaita samalla tai toisella tietokoneella. Muutoksia voidaan tuoda myös toiseen suuntaan, mikä on yleistä varsinkin silloin, kun Gittiä käytetään projektissa, jossa työskentelee useita ihmisiä. [Wik12c]

Kuva 2.2 havainnollistaa Gitin tallennustasoja. Siinä tasojen välillä olevien nuolien tekstit viittaavat Gitin eri komentoihin, joita on kuvailtu lyhyesti kohdassa 2.6.



Kuva 2.2: Gitin tallennustasot

2.6 Tärkeimmät toiminnot

Tämän kohdan tarkoitus on antaa käsitys tavallisimmista Gitin toiminnoista, jotta työn seuraavissa kohdissa tehdyt matemaattiset tarkastelut olisi helpompi ymmärtää kokonaiskuvan kannalta. Kohdan ei ole tarkoitus olla käyttöohje, joten toimintojen käyttötapoja ei esitetä. Gitin opetteluun löytyy monia lähteitä, joista esimerkiksi Scott Chaconin kirja ”Pro Git” [Cha09] on erinomainen. Kirja on luettavissa Gitin kotisivujen kautta osoitteesta `http://git-scm.com/book/`, josta siitä voi ladata myös PDF-version.

2.6.1 Tietovaraston luonti

Tavallisin tapa aloittaa Gitin käyttö on joko tehdä oma tietovarasto (komennolla ”`git init`”) tai kloonata toinen tietovarasto (”`git clone`”).

2.6.2 Muutosten tekeminen

Muutokset tiedostoihin tehdään työskentelyhakemistossa ja lisätään sitten valmistelualueelle (”`git add`”). Valmistelualueen sisällöstä luodaan commit-objekti (”`git commit`”), minkä yhteydessä annetaan myös commit-viesti. Tekijätiedot Git täyttää automaattisesti commit-objektiin, kunhan se on aseteltu oikein (”`git config`”).

2.6.3 Muutosten tarkastelu

Gitillä on mahdollista tarkastella kahden commit-objektin tallentaman tilan eroa toisiinsa (”`git diff`” tai ”`git show`”). Tällöin näytetään – paitsi lisätyt ja poistetut tiedostot – myös muuttuneen tiedoston kahden eri version välinen ero. Tiedostojen erojen vertailua on käsitelty kappaleessa ”4 Tiedostojen erojen vertailu”.

Git voi näyttää myös listan tehdyistä muutoksista (”`git log`”), jolloin commit-objektien muodostamaa graafia kuljetaan lehdestä juureen ja näytetään jokaisesta viesti ja tekijätiedot.

2.6.4 Kehityshaarojen kanssa työskentely

Kehityshaarojen käyttökohteita ovat esimerkiksi jonkin idean kokeileminen toisessa kehityshaarassa tai vanhojen versioiden ylläpito erillään uudesta versiosta.

Kun idealle on luotu oma kehityshaaransa ("`git branch`") ja idea osoittautui toimivaksi, niin kehityshaaran voi yhdistää takaisin pääkehityshaaraan ("`git merge`"). Toisaalta, jos idea osoittautui huonoksi, kehityshaaran voi vain poistaa ("`git branch -D`"), jolloin pääkehityshaaraan ei tule turhia muutoksia.

Ylläpitäessä vanhempaa versiota omassa kehityshaarassaan on mahdollista tehdä toisessa kehityshaarassa uusia ominaisuuksia ja tehdä vanhaan versioon pelkästään korjauksia. Tällöin voidaan esimerkiksi tehdä korjaukset ensin vanhaan versioon ja tuoda ne sitten uudempaan versioon yhdistämällä vanhan version kehityshaara uuden version kehityshaaraan. Toinen vaihtoehto on tehdä korjaukset uudempaan versioon ja tuoda ne sitten vanhempaan versioon poimimalla yksittäisen commit-objektin tallentamat muutokset ("`git cherry-pick`").

Kun kehityshaaraa vaihtaa ("`git checkout`"), niin Git vaihtaa työskentelyhakemiston sisällön vastaamaan kyseisen kehityshaaran viimeisimmän commit-objektin tallentamaa tilaa ja merkitsee muistiin kyseisen kehityshaaran nimen.

2.6.5 Toisien tietovarastojen kanssa työskentely

Muutoksia voi työntää toiseen Git-tietovarastoon ("`git push`") tai hakea toisesta tietovarastosta ("`git fetch`"). Tässä yhteydessä annetaan yleensä kehityshaara, jonka tuorein commit-objekti ja kaikki sen viittaamat objektit kopioidaan toiseen tietovarastoon. Objekteja, jotka ovat jo toisessa tietovarastossa, ei kopioida. Kopioiminen tehdään käytännössä muodostamalla kopioitavista objekteista pakettitiedosto, joka lähetetään toiseen tietovarastoon.

Toinen vaihtoehto muutosten lähettämiseen on tehdä muutoksista korjaustiedosto (patch) ("`git format-patch`"), jonka toisen Git-tietovaraston omistaja ottaa vastaan ja tuo sen muutokset omaan tietovarastoonsa ("`git am`").

3 Pakkausmenetelmä

Git käyttää irrallisten objektien tallennuksen yhteydessä Zlib deflate -pakkausmenetelmää [Cha11b, Kohta "Object Storage"]. Myös pakettitiedostoon kerätyt objektit on pakattu samalla menetelmällä [Cha11b, Kohta "Packfiles"].

Erityisesti pakettitiedoston tallennuksessa pakkaus on tärkeää, sillä pakettitiedostojen tulisi olla pieniä tietojen siirtämisen helpottamiseksi ja levytilan säästämiseksi. Kuitenkin pakkausmenetelmän tulisi olla riittävän nopea, jotta toiminnot eivät hidastuisi liikaa. Zlib deflate -algoritmi on hyvä kompromissi nopeuden ja pakkaussuhteen välillä [Ter09]. Lisäksi deflate-algoritmin etu on, että se on yleisesti käytetty [Sal06, Kohta 3.23] ja zlib-toteutus on saatavilla monelle eri alustalle [RA12].

3.1 Zlib

Zlib-pakatun tietovirran muoto on määritelty RFC1950-standardissa. Zlib-pakatun tietovirran alussa on kahden tavun kokoinen otsikko. Otsikkotiedoissa on periaatteessa mahdollista valita muitakin pakkaustapoja kuin deflate, mutta RFC1950 ei kuitenkaan määrittele muita vaihtoehtoja. Muita otsikkoon sisältyviä tietoja ovat pakkauksessa käytetty ikkunan koko (window size) ja pakkaustaso (compression level). Otsikkotavujen jälkeen seuraa deflate-menetelmällä pakattu data ja lopussa on vielä neljän tavun kokoinen Adler-32-tarkistussumma alkuperäisestä pakkaamattomasta datasta. [Net+96a]

Deflate-pakkausmenetelmässä käytetään hieman muunneltua LZ77-pakkausmenetelmää ja Huffman-koodausta [Net+96b]. Tutkitaan seuraavaksi tarkemmin Huffman-koodausta ja sen jälkeen LZ77-pakkausta. Lopuksi vielä näytetään tarkemmin, miten deflate-pakkaus toimii.

3.2 Huffman-koodaus

Huffman-koodaus on yhden tyyppinen entropiakoodaus [Wik12b], jonka esitteli David Huffman vuonna 1952 [Huf52]. Deflate-pakkausmenetelmässä sitä käytetään kahden eri aakkoston symbolien esittämiseen mahdollisimman lyhyesti [Net+96b].

Huffman-koodauksella jokainen lähdeaakkoston symboli koodataan koodisanaksi siten, että useammin esiintyville symboleille valitaan lyhyemmät tai yhtä pitkät koodisanat kuin harvemmin esiintyville. Tämä vähentää tiedon esityksessä tarvittavien bittien määrää ja itse asiassa Huffman-koodaus on optimaalinen tapa muodostaa symboleihin liitettävät koodisanat.

Seuraavaksi esitetään Huffmanin koodausalgoritmi ja osoitetaan, että sillä muodostettu koodi on optimaalinen. Esitys perustuu lähteeseen [Yli06].

Koodausalgoritmin esittämiseen sekä optimaalisuuden määrittelemiseen ja todistamiseen tarvitaan kuitenkin hieman pohjakäsitteitä, joten esitetään ensin muutama määritelmä ja merkintä.

3.2.1 Määritelmä. *Lähdeaakkosto* on äärellinen joukko, jonka alkioita sanotaan *symboleiksi*. Symbolien esiintymistodennäköisyydet voidaan antaa *todennäköisyysvektorilla* tai ilmoittamalla ne symbolien kanssa yhdessä antamalla *(todennäköisyys)jakauma*.

3.2.2 Merkintä. Lähdeaakkostoa merkitään tässä esityksessä yleensä symbolein $X = \{x_1, \ldots, x_n\}$, todennäköisyysvektoria $(p_1, \ldots, p_n) \in [0,1]^n$ ja jakaumaa $\{(x_1, p_1), \ldots, (x_n, p_n)\}$.

3.2.3 Määritelmä. *Koodi* on mikä tahansa epätyhjä sanojen joukko $C \subset A^+$, missä A on äärellinen *(koodi)aakkosto*. Koodin alkioita sanotaan *koodisanoiksi*.

3.2.4 Määritelmä. *Koodauskaavio* on kuvaus $\phi\colon X \to C$ ja *koodausfunktio* on kuvaus $\Phi\colon X^+ \to C^+$, missä X on lähdeaakkosto ja C on koodi. Koodausfunktio on *yksiselitteinen*, jos se on injektio. Koodauskaavio ϕ *määrittelee* koodausfunktion Φ_ϕ, jonka arvot määräytyvät lausekkeesta

$$\Phi_\phi(z_1 \cdots z_k) = \phi(z_1) \cdots \phi(z_k),$$

missä $k \geq 0$ ja $z_1, \ldots, z_k \in X$.

3.2.5 Määritelmä. Sana $u \in A^*$ on sanan $v \in A^*$ *prefiksi* tai *alkuosa*, jos $v = uw$ jollain sanalla $w \in A^*$.

3.2.6 Määritelmä. Koodi C on *prefiksikoodi*, jos mikään sen koodisanoista ei ole toisen prefiksi, eli $v \neq uw$ kaikilla $u, v \in C$, $w \in A^*$, $u \neq v$. Koodi C

on *yksikäsitteisesti dekoodattavissa*, jos koodisanoille $u_1, \ldots, u_j, v_1, \ldots, v_k \in C$ ehdosta $u_1 \cdots u_j = v_1 \cdots v_k$ seuraa, että $j = k$ ja $u_i = v_i$ kaikilla $i = 1, \ldots, k$.

Selvästi prefiksikoodi on aina yksikäsitteisesti dekoodattavissa, mutta ei välttämättä toisin päin.

3.2.7 Esimerkki. Koodi $\{\bar{0}, \bar{1}\bar{0}, \bar{1}\bar{1}\}$ on prefiksikoodi ja koodi $\{\bar{0}, \bar{0}\bar{1}\}$ on yksikäsitteisesti dekoodattavissa, mutta ei prefiksikoodi. Koodi $\{\bar{0}, \bar{1}, \bar{0}\bar{1}\}$ ei ole yksikäsitteisesti dekoodattavissa.

Seuraavaksi keskitytään kuitenkin vain prefiksikoodeihin, sillä seuraavan lemman perusteella prefiksikoodeilla on mahdollista päästä ihan yhtä lyhyisiin koodisanoihin kuin millä tahansa yksikäsitteisesti dekoodattavissa olevilla koodeilla.

3.2.8 Lemma. *Jos koodi $C = \{w_1, \ldots, w_k\} \subset A^+$ on yksikäsitteisesti dekoodattavissa, niin on olemassa prefiksikoodi $C' = \{w'_1, \ldots, w'_k\} \subset A^+$, jolle $|w'_i| = |w_i|$ kaikilla $i = 1, \ldots, k$.*

Todistus. Sivuutetaan. Ks. esimerkiksi [Yli06, Seuraus 3.3.3]. □

3.2.9 Määritelmä. Koodauskaavion $\phi\colon \{x_1, \ldots, x_n\} \to C$ *koodisanojen keskipituus* todennäköisyysvektorin $(p_1, \ldots, p_n)$ suhteen on

$$\Lambda_\phi = \sum_{i=1}^{n} p_i \, |\phi(x_i)|.$$

3.2.10 Määritelmä. Koodauskaavio $\phi\colon X \to C$ on *optimaalinen* (jakauman $\{(x_1, p_1), \ldots, (x_n, p_n)\}$ suhteen), jos C on prefiksikoodi, ϕ injektio ja jokaiselle muulle vastaavat ehdot täyttävälle koodauskaaviolle $\phi'\colon X \to C'$ koodisanojen keskipituus on suurempi tai yhtäsuuri kuin koodauskaavion ϕ koodisanojen keskipituus eli $\Lambda_{\phi'} \geq \Lambda_\phi$.

3.2.11 Lemma. *Optimaalinen koodauskaavio on aina olemassa.*

Todistus. Olkoon $X = \{x_1, \ldots, x_n\}$ koodattavien symbolien joukko, jonka todennäköisyysvektori on $(p_1, \ldots, p_n)$, ja A käytettävä koodiaakkosto. Eräs koodi C saadaan esimerkiksi valitsemalla $C = A^m$, missä m on riittävän suuri

kokonaisluku, jolle $|A|^m \geq n$. Tämä on selvästi prefiksikoodi ja injektiivinen koodauskaavio $\phi\colon X \to C$ on mahdollista muodostaa, sillä $|C| = |A|^m \geq |X|$. Optimaalisen koodauskaavion löytämiseksi riittää tutkia koodauskaavioita ϕ', joille $\Lambda_{\phi'} \leq \Lambda_\phi$. Koodisanojen keskipituuden määritelmästä seuraa

$$p_i^{-1}\Lambda_{\phi'} = |\phi'(x_i)| + p_i^{-1}\sum_{j\neq i} p_j\,|\phi'(x_j)| \geq |\phi'(x_i)|\,,$$

joten sanan $\phi'(x_i)$ pituus tulisi olla korkeintaan $p_i^{-1}\Lambda_{\phi'} \leq p_i^{-1}\Lambda_\phi$. Mahdollisia koodeja ja koodauskaavioita on siis äärellinen määrä. □

3.2.12 Lemma. *Olkoon $\{(x_1, p_1), \ldots, (x_n, p_n)\}$ todennäköisyysjakauma, missä $p_1 \geq \cdots \geq p_n$ ja $n \geq 2$ sekä $C \subset A^+$ prefiksikoodi, jolle $|C| = n$, ja $\phi\colon \{x_1, \ldots, x_n\} \to C$ injektiivinen koodauskaavio, jolle on voimassa*

$$|\phi(x_i)| \leq \cdots \leq |\phi(x_{i+k})|\,,\ \textit{jos}\ p_i = \cdots = p_{i+k}. \tag{3.1}$$

Jos koodauskaavio ϕ on optimaalinen, niin seuraavat ehdot ovat voimassa:

(i) Suurempiin todennäköisyyksiin liittyvät koodisanat eivät voi olla pitempiä kuin pienempiin todennäköisyyksiin liittyvät, eli $|\phi(x_i)| \leq |\phi(x_j)|$ aina, kun $p_i > p_j$.

(ii) Kahteen vähiten todennäköiseen symboliin liittyvät koodisanat ovat yhtä pitkät eli $|\phi(x_{n-1})| = |\phi(x_n)|$.

(iii) Pituudeltaan $|\phi(x_n)|$ olevien koodisanojen joukossa on kaksi sanaa, joissa muut merkit kuin viimeinen ovat samat.

Todistus. (i) Jos joillain i ja j olisi $p_i > p_j$ ja $|\phi(x_i)| > |\phi(x_j)|$, niin vaihtamalla sanat $\phi(x_i)$ ja $\phi(x_j)$ keskenään saataisiin injektiivinen koodauskaavio, joka antaisi lyhyemmän koodisanojen keskipituuden kuin ϕ.

(ii) Kohdan (i) ja ehdon (3.1) perusteella $|\phi(x_{n-1})| \leq |\phi(x_n)|$. Jos olisi $|\phi(x_{n-1})| < |\phi(x_n)|$, niin voitaisiin jättää sanan $\phi(x_n)$ viimeinen merkki pois, jolloin saataisiin edelleen prefiksikoodi ja koodauskaavio, joka antaisi lyhyemmän koodisanojen keskipituuden kuin ϕ.

(iii) Kohdan (ii) perusteella pituudeltaan $|\phi(x_n)|$ olevia koodisanoja on vähintään kaksi. Jos näiden joukossa ei olisi mainitunlaista sanaparia, niin voitaisiin niistä kaikista jättää viimeinen merkki pois, jolloin saataisiin edelleen prefiksikoodi ja koodauskaavio, joka antaisi lyhyemmän koodisanojen keskipituuden kuin ϕ.

□

Rajoitutaan nyt yksinkertaisuuden vuoksi binäärikoodauksiin, siis tästä eteenpäin koodiaakkosto $A = \beta = \{\bar{0}, \bar{1}\}$.

3.2.13 Lemma. *Olkoon $p_1 \geq p_2 \geq \cdots \geq p_n$ ja ϕ_0 koodauskaavio, joka on optimaalinen jakauman*

$$\{(x_1, p_1), \ldots, (x_{n-2}, p_{n-2}), (z, p_{n-1} + p_n)\}$$

suhteen, $X = \{x_1, \ldots, x_n\}$ sekä $\phi_1 \colon X \to C_1$ koodauskaavio, joka määritellään seuraavasti:

$$\phi_1(x) = \begin{cases} \phi_0(x), & \text{jos } x \in \{x_1, \ldots, x_{n-2}\} \\ \phi_0(z)\bar{0}, & \text{jos } x = x_{n-1} \\ \phi_0(z)\bar{1}, & \text{jos } x = x_n. \end{cases}$$

Tällöin koodauskaavio ϕ_1 on optimaalinen jakauman

$$\{(x_1, p_1), \ldots, (x_n, p_n)\}$$

suhteen.

Todistus. Selvästi C_1 on prefiksikoodi ja ϕ_1 injektio. Tehdään vastaoletus, että koodauskaavio ϕ_1 ei ole optimaalinen. Lemman 3.2.11 perusteella optimaalinen koodauskaavio on kuitenkin olemassa. Olkoon $\phi_1' \colon X \to C_1'$ optimaalinen koodauskaavio jakauman $\{(x_1, p_1), \ldots, (x_n, p_n)\}$ suhteen. Voidaan olettaa, että ϕ_1' toteuttaa myös Lemman 3.2.12 vaatimuksen (3.1). Lemman 3.2.12 kohdan (iii) nojalla koodisanojen C_1' joukossa on kaksi sanaa, jotka ovat pituudeltaan $|\phi_1'(x_n)|$ ja eroavat vain viimeisen merkin osalta sekä kohdan (ii) nojalla myös $\phi_1'(x_{n-1})$ on yhtä pitkä. Yhtä pitkien sanojen vaihtaminen keskenään ei pidennä koodisanojen keskipituutta, joten voidaan olettaa, että juuri $\phi_1'(x_n)$

ja $\phi_1'(x_{n-1})$ eroavat vain viimeisen merkin osalta. Merkitään näiden yhteistä alkuosaa $\phi_1'(x_n)A^{-1}$ ja määritellään koodauskaavio $\phi_0' \colon \{x_1, \ldots, x_{n-2}, z\} \to C_0'$ seuraavasti:

$$\phi_0'(x) = \begin{cases} \phi_1'(x), & \text{jos } x \in \{x_1, \ldots, x_{n-2}\} \\ \phi_1'(x_n)A^{-1}, & \text{jos } x = z. \end{cases}$$

Nyt ϕ_0' on injektio ja C_0' on prefiksikoodi. Verrataan seuraavaksi koodisanojen keskipituuksia. Huomataan, että

$$\begin{aligned} \Lambda_{\phi_0'} &= \sum_{i=1}^{n-2} p_i \left|\phi_0'(x_i)\right| + (p_{n-1} + p_n) \left|\phi_0'(z)\right| \\ &= \sum_{i=1}^{n-2} p_i \left|\phi_1'(x_i)\right| + (p_{n-1} + p_n)(\left|\phi_1'(x_n)\right| - 1) \\ &= \sum_{i=1}^{n-2} p_i \left|\phi_1'(x_i)\right| + p_{n-1} \left|\phi_1'(x_n)\right| + p_n \left|\phi_1'(x_n)\right| - (p_{n-1} + p_n) \end{aligned}$$

ja koska $|\phi_1'(x_n)| = |\phi_1'(x_{n-1})|$, niin saadaan edellisestä

$$\Lambda_{\phi_0'} = \sum_{i=1}^{n} p_i \left|\phi_1'(x_i)\right| - (p_{n-1} + p_n) = \Lambda_{\phi_1'} - (p_{n-1} + p_n).$$

Koodauskaavion ϕ_1 määritelmän perusteella $|\phi_1(x_n)| = |\phi_1(x_{n-1})| = |\phi_0(z)| + 1$ ja $|\phi_1(x_i)| = |\phi_0(x_i)|$ kaikilla $i \in \{1, \ldots, n-2\}$, joten

$$\begin{aligned} \Lambda_{\phi_0} &= \sum_{i=1}^{n-2} p_i \left|\phi_0(x_i)\right| + (p_{n-1} + p_n) \left|\phi_0(z)\right| \\ &= \sum_{i=1}^{n-2} p_i \left|\phi_1(x_i)\right| + p_{n-1} \left|\phi_1(x_{n-1})\right| + p_n \left|\phi_1(x_n)\right| - (p_{n-1} + p_n) \\ &= \Lambda_{\phi_1} - (p_{n-1} + p_n). \end{aligned}$$

Vastaoletuksen mukaan $\Lambda_{\phi_1'} < \Lambda_{\phi_1}$, joten edeltävän nojalla saadaan

$$\Lambda_{\phi_0'} = \Lambda_{\phi_1'} - (p_{n-1} + p_n) < \Lambda_{\phi_1} - (p_{n-1} + p_n) = \Lambda_{\phi_0}.$$

Tämä on ristiriidassa koodauskaavion ϕ_0 optimaalisuuden kanssa. □

Edellisen lemman perusteella on mahdollista muodostaa algoritmi, joka tuottaa optimaalisen koodin mille tahansa todennäköisyysjakaumalle, sillä koodauskaavion ϕ_0 jakauma saadaan koodauskaavion ϕ_1 jakaumasta yhdistämällä kaksi pienimmän todennäköisyyden symbolia yhdeksi symboliksi. Tätä jatkamalla päädytään lopuksi kahden symbolin jakaumaan, jolle triviaalisti saadaan koodi $\{\bar{0}, \bar{1}\}$. Tähän perustuu seuraavaksi esitettävä Huffmanin koodausalgoritmi.

3.2.14 Algoritmi. *Huffmanin koodausalgoritmi* toimii seuraavasti: Muodostetaan binääripuu, jonka lehtinä ovat koodattavat symbolit. Puu muodostetaan laittamalla aluksi kaikki symbolit listaan solmuiksi. Sitten listasta poistetaan kaksi pienimmän todennäköisyyden solmua ja lisätään tilalle uusi solmu, jonka lapset ovat nuo poistetut solmut ja todennäköisyys niiden yhteenlaskettu todennäköisyys. Tätä jatketaan kunnes listassa on enää yksi solmu – puun juuri. Lopuksi puun kaaret merkitään niin, että solmun toisen lapsen kaari saa merkin $\bar{0}$ ja toisen lapsen kaari merkin $\bar{1}$. Symbolien koodisanat muodostetaan katenoimalla merkit niistä kaarista, jotka ovat puun juuren ja symbolin lehtisolmun välisellä polulla. Algoritmi on esitetty pseudokoodina Koodilistauksessa 3.1.

Huffmanin koodausalgoritmi on hyvä esimerkki ns. ahnaasta algoritmista. Ahnaassa algoritmissa ratkaisu rakennetaan vaiheittain valiten joka vaiheessa aina osaratkaisu, jolla saavutetaan välitön ja siinä tilanteessa itsestään selvä hyöty. [DPV06, s. 139]

Koodilistaus 3.1 Huffmanin koodausalgoritmi

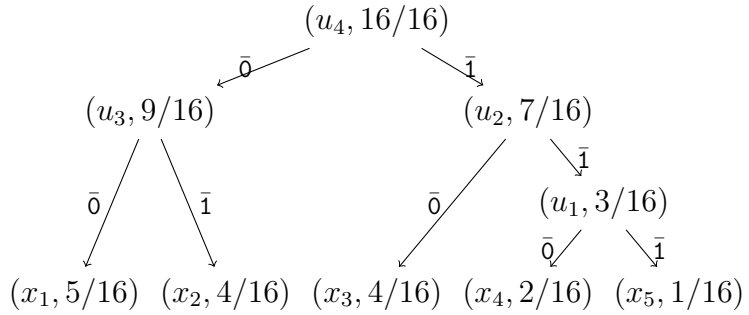
Vaatimus: LUKUMÄÄRÄ($symbolit$) ≥ 2 ▷ Vähintään kaksi symbolia

1: **funktio** TEEHUFFMANKOODI($symbolit, tn$) ▷ tn=todennäköisyydet
2: $juurisolmu \leftarrow$ TEEHUFFMANPUU($symbolit, tn$)
3: **palauta** TEEKOODAUSKAAVIOPUUSTA($juurisolmu, \varepsilon$) ▷ ε=tyhjä sana
4: **funktio** TEEHUFFMANPUU($symbolit, tn$)
5: $solmut \leftarrow symbolit$ ▷ Solmujen lista alustetaan symboleilla
6: **niin kauan kun** LUKUMÄÄRÄ($solmut$) ≥ 2 **tee**
7: $solmut \leftarrow$ JÄRJESTÄ($solmut, tn$) ▷ Järjestä todennäköisyyksien mukaan
8: $s_1 \leftarrow solmut[1]$ ▷ Pienimmän todennäköisyyden solmu
9: $s_2 \leftarrow solmut[2]$ ▷ Toiseksi pienimmän todennäköisyyden solmu
10: $u \leftarrow$ UUSISOLMU()
11: $u.oikeaLapsi \leftarrow s_1$
12: $u.vasenLapsi \leftarrow s_2$
13: $tn[u] \leftarrow tn[s_1] + tn[s_2]$ ▷ u:n todennäköisyydeksi s_1:n ja s_2:n todennäköisyyksien summa
14: $solmut \leftarrow [u] + solmut[3\ldots]$ ▷ u listaan, s_1 ja s_2 pois
15: **palauta** $solmut[1]$
16: **funktio** TEEKOODAUSKAAVIOPUUSTA($solmu, w$) ▷ w=prefiksi
17: **jos** ONLEHTI($solmu$) **niin**
18: $symboli \leftarrow solmu$ ▷ Lehtisolmuina ovat symbolit
19: **palauta** $\{(symboli \rightarrow w)\}$ ▷ Palauta yhden alkion joukko
20: **muuten**
21: $V \leftarrow$ TEEKOODAUSKAAVIOPUUSTA($solmu.vasenLapsi, w\bar{0}$)
22: $O \leftarrow$ TEEKOODAUSKAAVIOPUUSTA($solmu.oikeaLapsi, w\bar{1}$)
23: **palauta** $V \cup O$ ▷ Palauta kuvausjoukkojen unioni

3.2.15 Esimerkki. Muodostetaan Huffmanin koodausalgoritmilla koodauskaavio jakaumalle

$$\{(x_1, 5/16), (x_2, 4/16), (x_3, 4/16), (x_4, 2/16), (x_5, 1/16)\}.$$

Ensin symboleista muodostetaan binääripuu yhdistämällä aina kaksi pienimmän todennäköisyyden solmua ja lopuksi kaaret merkitään merkeillä $\bar{0}$ ja $\bar{1}$. Binääripuusta tulee seuraavanlainen:



Sitten koodisanat muodostetaan katenoimalla juuren u_4 ja symbolin x_i välisen polun merkit. Koodauskaavioksi saadaan

$$\{x_1 \to \bar{0}\bar{0}, x_2 \to \bar{0}\bar{1}, x_3 \to \bar{1}\bar{0}, x_4 \to \bar{1}\bar{1}\bar{0}, x_5 \to \bar{1}\bar{1}\bar{1}\}.$$

3.2.16 Lause. *Huffmanin koodausalgoritmi tuottaa optimaalisen koodin mille tahansa jakaumalle* $X_n = \{(x_1, p_1), \ldots, (x_n, p_n)\}$*, missä* $n \geq 2$*.*

Todistus. Oletetaan, että $p_1 \geq p_2 \geq \ldots \geq p_n$. Todistetaan väite induktiolla.

Kun $n = 2$, niin ensimmäisessä vaiheessa algoritmi tekee binääripuun, jossa on juurisolmu ja sen lapsina symbolit x_1 ja x_2. Tästä algoritmi tuottaa koodauskaavion $\{x_1 \to \bar{1}, x_2 \to \bar{0}\}$, joka on optimaalinen.

Oletetaan seuraavaksi, että algoritmi tuottaa optimaalisen koodauskaavion aina kun $n = k$ eli myös jakaumalle

$$Z = \{(x_1, p_1), \ldots, (x_{k-1}, p_{k-1}), (z, p_k + p_{k+1})\}.$$

Algoritmi tuottaa jakaumalle X_{k+1} muuten samanlaisen binääripuun kuin jakaumalle Z, mutta siinä missä jakauman Z puussa alkio z on lehtisolmu,

jonka todennäköisyysarvo on $p_k + p_{k+1}$, on jakaumalla X_{k+1} puussa sen alla vielä lehtisolmut x_k ja x_{k+1}. Näin ollen algoritmi tuottaa jakaumalle Z ja X_{k+1} muuten samanlaiset koodauskaaviot, mutta jakauman X_{k+1} koodauskaavioon tulee alkioille x_k ja x_{k+1} koodisanoiksi $w\bar{1}$ ja $w\bar{0}$, missä w on alkion z koodisana jakauman Z koodauskaaviossa. Lemman 3.2.13 perusteella näin muodostettu koodauskaavio on optimaalinen. □

3.3 LZ77-koodaus

LZ77-koodausalgoritmin kehittivät Jacob Ziv ja Abraham Lempel. He julkaisivat sen vuonna 1977 artikkelissa "A Universal Algorithm for Sequential Data Compression" [ZL77], minkä jälkeen algoritmia on käytetty pohjana usealle eri pakkausalgoritmille, joista suosituin on kohdassa 3.4 käsiteltävä deflate-pakkaus. [Wik12d]

Algoritmin perusidea on korvata syötteessä toistuvia merkkijonoja viittauksilla aiempiin kohtiin. Enkooderi tuottaa jonon kolmikoita (p, l, x), missä luku l kertoo kolmikon koodaaman sanan pituuden, p osoittaa toistuvan sanan kohdan aiemmassa syötteessä ja x on seuraava merkki toistuvan sanan jälkeen. Toistuvan sanan pituus on siis $l - 1$. Toistojen löytämiseksi enkooderilla on puskuri, jossa se pitää osaa syötteestä. Puskuri on jaettu kahteen osaan. Sen vasemmalla puolella on jo koodattua syötettä ja oikealla puolella seuraavaksi koodattava syöte. Tavoitteena on löytää puskurista seuraavaksi koodattavan syötteen mahdollisimman pitkä prefiksi, joka alkaa puskurin vasemmalta puolelta. Aina koodattuaan yhden kolmikon, enkooderi liikuttaa syötettä oikealta vasemmalle puskurissaan. LZ77-koodausta kutsutaankin toisinaan liukuvan ikkunan (sliding window) pakkausmenetelmäksi [Sal06, Kohta 3.3]. [ZL77]

Merkitään puskurin pituutta symbolilla n, sen oikean puolen pituutta symbolilla L_S ja syötteen aakkostoa symbolilla A. Tuotettujen koodisanojen aakkosto voisi olla mikä tahansa, mutta käytetään myös sille tässä samaa aakkostoa A. Oletetaan, että $|A| \geq 2$.

Alussa, ennen kuin yhtään merkkiä on vielä koodattu, puskurin vasen puoli pitää alustaa joillain merkeillä. Olkoon $a_0 \in A$ merkki, jolla puskurin vasen puoli alustetaan.

Koodatun kolmikon esitystavoissa on eri lähteiden välillä poikkeavuuksia.

Alkuperäisessä Lempelin ja Zivin artikkelissa parametrin p arvo 1 viittaa puskurin alkuun, kun toisissa lähteissä se viittaa vasemman osan oikeanpuoleisimpaan merkkiin. Samoin l on toisissa lähteissä toiston pituus, mutta alkuperäisessä artikkelissa se on kolmikon koodaaman sanan pituus eli yhtä suurempi. Tässä esityksessä käytetään parametreille p ja l samaa esitystapaa kuin Lempelin ja Zivin artikkelissa. [ZL77; Sal06; Wik12d]

3.3.1 Esimerkki. Olkoon $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $n = 15$ ja $L_S = 8$. Oletetaan, että enkooderilla on puskurissa

$$\mathtt{abaabcb} \cdot \mathtt{abcababa}.$$

Enkooderi on siis koodaamassa oikean puolen sanaa $w = \mathtt{abcababa}$ ja yrittää etsiä sanan w prefiksejä, jotka alkaisivat vasemmalta puolelta. Seuraavaksi kolmikoksi (p, l, x) voisi tulla esimerkiksi $(1, 3, \mathtt{c})$, $(2, 1, \mathtt{a})$ tai $(4, 4, \mathtt{a})$. Enkooderi valitsee näistä viimeisen, sillä se on paras vaihtoehto, koska se koodaa eniten syötteen merkkejä. Tämän jälkeen syötteestä luetaan lisää merkkejä ja puskurin sisältöä liikutetaan vasemmalle. Tämän jälkeen puskurissa on

$$\mathtt{bcbabca} \cdot \mathtt{babacccc},$$

missä lopun `cccc` on syötteestä luetut uudet merkit.

Kolmikon (p, l, x) osoittama toiston pituus $l - 1$ voi olla suurempi kuin puskurin vasemmalla puolella on merkkejä kohdasta p eteenpäin, mikä mahdollistaa yksinkertaisen jakson pituuden koodauksen (run-length encoding, RLE) [Wik12d]. Havainnollistetaan tätä esimerkillä.

3.3.2 Esimerkki. Olkoon $n = 15$, $L_S = 11$ ja enkooderilla puskurissa

$$\mathtt{babc} \cdot \mathtt{abcabcabcac}.$$

Tällöin enkooderi tuottaa kolmikon $(2, 11, \mathtt{c})$ eli toiston pituus on $l - 1 = 10$ vaikka puskurin vasemmalla puolella on vain kolme merkkiä kohdasta $p = 2$ eteenpäin.

Esitetään vielä, miten kolmikot (p, l, x) koodataan aakkoston A koodisanoiksi. Olkoon $\alpha = |A|$. Parametri p voi saada arvokseen luvun $1, \ldots, n - L_S$ ja

l puolestaan $1, \ldots, L_S$. Täten siis arvon p esittämiseen riittää $\lceil \log_\alpha (n - L_S) \rceil$ aakkoston A merkkiä ja arvon l esittämiseen $\lceil \log_\alpha L_S \rceil$ merkkiä. Koko kolmikko voidaan siis koodata yhdeksi sanaksi, jonka pituus on

$$L_C = \lceil \log_\alpha (n - L_S) \rceil + \lceil \log_\alpha L_S \rceil + 1.$$

Koska arvot n ja L_S määräävät koodisanan ja sen osien pituuden, on olennaista, että enkooderilla ja dekooderilla on käytössään samat arvot.

Koodilistauksissa 3.2 ja 3.3 on esitetty LZ77-enkooderi ja -dekooderi pseudokoodina.

Koodilistaus 3.2 LZ77-enkoodausalgoritmi

Vaatimus: $S \in A^*$, $n > L_S > 0$ ja $a_0 \in A$.

funktio ENKOODAALZ77(S, n, L_S) ▷ Enkoodaa sana S LZ77-algoritmilla
    $C \leftarrow \varepsilon$ ▷ Muuttuja johon koodattu sana kerätään
    $B_0 \leftarrow a_0^n$ ▷ n kpl merkkiä a_0
    $l_0 \leftarrow L_S$
    $m \leftarrow 0$ ▷ S:stä luettujen merkkien määrä
    $i \leftarrow 1$
    niin kauan kun $m < |S|$ **tee**
        $B_i \leftarrow B_{i-1}[l_{i-1} + 1 \ldots]S[m + 1 \ldots m + l_{i-1}]$ ▷ Lue l_{i-1} uutta merkkiä
        $m \leftarrow m + l_{i-1}$
        $(p_i, l_i) \leftarrow$ ETSIPISINVASTAAVUUS($B_i, n - L_S$)
        $C_{i1} \leftarrow$ ENKOODAALUKU($p_i - 1, \lceil \log_\alpha (n - L_S) \rceil$) ▷ Aloituskohta
        $C_{i2} \leftarrow$ ENKOODAALUKU($l_i - 1, \lceil \log_\alpha L_S \rceil$) ▷ Pituus
        $C_{i3} \leftarrow B_i[n - L_S + l_i]$ ▷ Seuraava merkki
        $C \leftarrow CC_{i1}C_{i2}C_{i3}$ ▷ Lisää koodatut sanat aiemman perään
        $i \leftarrow i + 1$
    palauta C
funktio ETSIPISINVASTAAVUUS(B, K)
    ... ▷ Etsi suurin mahdollinen luku l ja luku p, joille $1 \leq p \leq K$, $1 \leq l \leq |B| - K$ ja $B[p \ldots p + l - 2] = B[K + 1 \ldots K + l - 1]$.
    palauta (p, l)
funktio ENKOODAALUKU(N, L)
    ... ▷ Enkoodaa luku N sanaksi $w \in A^*$, jonka pituus on L
    palauta w

Koodilistaus 3.3 LZ77-dekoodausalgoritmi

Vaatimus: $C \in A^*$, $n > L_S > 0$ ja $a_0 \in A$.

funktio DEKOODAALZ77(C, n, L_S) ▷ Dekoodaa LZ77-enkoodattu sana C
  $B_1 \leftarrow a_0^{L_S}$ ▷ L_S kpl merkkiä a_0
  $L_p \leftarrow \lceil \log_\alpha L_S \rceil$
  $L_l \leftarrow \lceil \log_\alpha (n - L_S) \rceil$
  $L_C \leftarrow L_p + L_l + 1$
  $N \leftarrow |C| \,/ L_C$ ▷ kolmikoiden (p, l, x) lukumäärä
  kaikilla $i \leftarrow 1, 2, \ldots, N$ **tee**
    $C_i \leftarrow C[(i-1)L_C + 1 \ldots iL_C]$
    $p_i \leftarrow$ DEKOODAALUKU$(C_i[1 \ldots L_p]) + 1$ ▷ Aloituskohta
    $l_i \leftarrow$ DEKOODAALUKU$(C_i[L_p + 1 \ldots L_p + L_l]) + 1$ ▷ Pituus
    $x_i \leftarrow C_i[L_p + L_l + 1]$ ▷ Seuraava merkki
    kaikilla $k \leftarrow 1, 2, \ldots, l_i - 1$ **tee**
      $B_i[L_S + k] \leftarrow B_i[p_i + k - 1]$
    $B_i[L_S + l_i] \leftarrow x_i$
    $S_i \leftarrow B_i[L_S + 1 \ldots L_S + l_i]$ ▷ viimeiset l_i merkkiä
    $B_{i+1} \leftarrow B_i[1 + l_i \ldots L_S + l_i]$ ▷ viimeiset L_S merkkiä
  palauta $S_1 S_2 \cdots S_N$

Havainnollistetaan Koodilistauksien 3.2 ja 3.3 enkooderin ja dekooderin toimintaa seuraavilla esimerkeillä.

3.3.3 Esimerkki. Olkoon käytettävä aakkosto $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, alustusmerkki $a_0 = \mathtt{a}$, puskurin koko $n = 20$ ja hakusanan pituus $L_S = 4$. Tutkitaan miten enkooderi koodaa sanan

$$S = \mathtt{dadadadadadadadbbbbbbbbc}.$$

Seuraavassa taulukossa on esitetty puskurin B_i sisältö ja muodostetun kolmikon (p_i, l_i, x_i) arvot eri vaiheissa i.

i	B_i	(p_i, l_i, x_i)
1	$\mathtt{aaaa} \cdot \mathtt{dadadadadadadadb}$	$(1, 1, \mathtt{d})$
2	$\mathtt{aaad} \cdot \mathtt{adadadadadadadbb}$	$(3, 15, \mathtt{b})$
3	$\mathtt{dadb} \cdot \mathtt{bbbbbbbc}$	$(4, 8, \mathtt{c})$

Kohdan p koodaamiseen tarvitaan $L_p = \lceil \log_4 L_S \rceil = \log_4 4 = 1$ merkki ja pituuden l koodaamiseen $L_l = \lceil \log_4 (n - L_S) \rceil = \log_4 16 = 2$ merkkiä.

Lopputuloksena on siis sana

$$C = (\texttt{a} \cdot \texttt{aa} \cdot \texttt{d})(\texttt{c} \cdot \texttt{dc} \cdot \texttt{b})(\texttt{d} \cdot \texttt{bd} \cdot \texttt{c}) = \texttt{aaadcdcbdbdc}.$$

Sana S saatiin siis koodattua puolet lyhyemmäksi sanaksi C, sillä $|S| = 24$ ja $|C| = 12$.

3.3.4 Esimerkki. Näytetään miten edellisen esimerkin sana saadaan dekoodattua. Syötteenä on siis $C = \texttt{aaadcdcbdbdc}$ sekä parametrit $n = 20$ ja $L_S = 4$.

Dekoodausalgoritmin 3.3 alussa, rivillä 2, puskurin B_1 sisältö alustetaan $L_S = 4$ kappaleella merkkiä $a_0 = \texttt{a}$ ja loput puskurit $B_2, B_3, \ldots$ alustetaan aina jokaisen vaiheen lopussa, rivillä 16, viimeisenä dekoodattuun L_S merkkiin. Rivin 12 silmukassa puskurin B_i sisältöä jatketaan merkki kerrallaan lukemalla saman puskurin aiempia merkkejä.

Alla olevassa taulukossa on havainnollistettu dekoodausalgoritmin vaiheita listaamalla eri muuttujien arvot jokaisen vaiheen i lopussa.

i	C_i	(p_i, l_i, x_i)	B_i	S_i
1	aaad	$(1, 1, \texttt{d})$	$\texttt{aaaa} \cdot \texttt{d}$	d
2	cdcb	$(3, 15, \texttt{b})$	$\texttt{aaad} \cdot \texttt{adadadadadadadb}$	adadadadadadadb
3	dbdc	$(4, 8, \texttt{c})$	$\texttt{dadb} \cdot \texttt{bbbbbbbc}$	bbbbbbbc

Saatiin siis dekoodattua sana $S = S_1S_2S_3 = \texttt{dadadadadadadadbbbbbbbbc}$.

3.4 Deflate-pakkaus

Deflate on suosittu pakkausmenetelmä, jota käytettiin alun perin hyvin tunnetussa Zip-ohjelmassa. Sen lisäksi, että Git käyttää sitä toteutuksessaan [Cha11b], se on käytössä monissa muissa sovelluksissa kuten PNG-kuvissa, PDF-tiedostoissa ja Internetin HTTP-protokollassa [Sal06, Kohta 3.23].

Deflate-menetelmän on kehittänyt Philip Katz osana Zip-tiedostomuotoa, jota hänen kehittämänsä PKZIP-ohjelma käyttää [Sal06, Kohta 3.23]. Deflate-muodon kuvaus on julkisesti saatavilla RFC1951-standarissa [Net+96b].

Esitetään seuraavaksi deflate-pakkausmenetelmän pääkohdat. Esitys perustuu pääosin kirjaan [Sal06, Kohta 3.23], mutta merkintätapoja on muutettu

matemaattisemmiksi ja kohdassa 3.4.4 käytetyt termit ja Lemma 3.4.4 eivät perustu lähteisiin.

Tavallisessa LZ77-koodauksessa tuotetaan jono kiinteän pituisia kolmikoita kohta-pituus-symboli, jolloin menee tilaa hukkaan, kun koodataan useita symboleita, joita ei aiemmasta syötteestä löydy. Deflate-pakkauksessa tuotetaankin tietovirtaan joko pelkästään symboli tai sitten pari kohta-pituus. Dekooderi erottaa kummasta on kyse, koska symbolit ja kohdat ovat laitettu samaan aakkostoon. Symbolit, kohdat ja pituudet ovat vielä koodattu Huffman-koodauksella, jotta useammin toistuvat symbolit, kohdat ja pituudet vievät vähemmän tilaa kuin harvemmin toistuvat. Monimutkaisin osa deflate-pakkausta onkin noiden Huffman-koodauskaavioiden tiivis esitystapa. Tutustutaan näihin tarkemmin alla, mutta aloitetaan perusteista.

3.4.1 Pakatun tietovirran yleinen rakenne

Deflate-koodauksen syöte on tavuja ja koodattu tietovirta koostuu biteistä eli lähdeaakkosto on $\mathcal{B}$ ja kohdeaakkosto on β.

Deflate-muotoinen tietovirta on jaettu paloihin (block), joita on tietovirrassa yksi tai useampia. Paloja on kolmea eri tyyppiä: pakkaamaton, kiinteällä sanastolla pakattu ja dynaamisella sanastolla pakattu. Enkooderi voi valita sopivan tyyppisen palan tilanteen mukaan. Molemmat pakatut palatyypit hyödyntävät yllä mainittua muunneltua LZ77-pakkausmenetelmää ja Huffman-koodausta. Kiinteän ja dynaamisen sanaston palojen ero on se, että kiinteän sanaston palassa symbolien koodauskaaviot on sovittu etukäteen ja dynaamisen sanaston paloissa koodauskaaviot on pakattu palan alkuun.

Deflate-pakatun tietovirran $S \in \beta^+$ muoto on siis

$$S = P_1 P_2 \cdots P_N,$$

missä $N \geq 1$ ja symbolit $P_i \in \beta^+$ $(i = 1, \ldots, N)$ ovat siis paloja. Kaikki palat alkavat bitillä, joka kertoo onko kyseessä viimeinen pala. Palan tyyppi on koodattu seuraavaan kahteen bittiin, joiden jälkeen palan sisällön muoto riippuu palan tyypistä. Olkoon $P_i = f_i t_i P_i'$ kaikilla $i = 1, \ldots, N$, missä $f_i \in \beta$ on ”viimeinen pala”-bitti ja $t_i \in \beta^2$ koodaa palan tyypin. Tarkemmin sanottuna

$f_i = \bar{0}$ kaikilla $i = 1, \ldots, N-1$ ja $f_N = \bar{1}$. Pakkaamattomalla palalla $t_i = \bar{0}\bar{0}$, kiinteän sanaston pakatulla palalla $t_i = \bar{0}\bar{1}$ ja dynaamisen sanaston pakatulla palalla $t_i = \bar{1}\bar{0}$.

3.4.2 Pakkaamaton palatyyppi

Pakkaamattoman palan muoto on

$$ftae_l\overline{e_l}u,$$

missä $f \in \beta$ ja $t = \bar{0}\bar{0}$ ovat "viimeinen pala"-bitti ja palan tyyppi kuten yllä. Bittijono $a \in \bigcup_{i=0}^{7} \{\bar{0}^i\}$ on kohdistus, $e_l \in \beta^{16}$ koodaa pakkaamattoman datan pituuden $l \geq 0$ 16-bittisessä muodossa ja $\overline{e_l} \in \beta^{16}$ on sen yhden komplementti. Viimeisenä on itse pakkaamaton data $u = u_1 u_2 \ldots u_l \in \mathcal{B}^*$.

Bittijono a kohdistaa seuraavat bitit kahdeksalla jaolliseen kohtaan tietovirran S alusta, joten jos S jaetaan tavuihin $S = s_1 \cdots s_k$, missä $s_1, \ldots, s_k \in \mathcal{B}$ ($k = |S|/8$), niin pakkaamattomat tavut asettuvat kohdakkain eli $s_{j+i} = u_i$ jollain $j > 0$ ja kaikilla $i = 1, \ldots, l$.

Pakkaamatonta palaa voidaan käyttää, kun enkooderi huomaa, että syöte ei pakkaudu tai toisinaan myös pakottamalla enkooderi tekemään pakkaamattomia paloja joka tapauksessa. Koska suurin mahdollinen arvo luvulle l on $2^{16} - 1 = 65535$ ja $|fte_l\overline{e_l}| = 35 < 5 \cdot 8$, niin K tavuisen syötteen deflate-koodaamiseen pakkaamattomilla paloilla tarvitaan enintään $K + 5\lceil K/65535 \rceil$ tavua.

3.4.3 Pakatut palatyypit

Pakatuissa paloissa käytetään LZ77-koodauksen muunnelmaa. Alkuperäisessä LZ77-koodauksessa käytetään jonoa kolmikkoja (p, l, x), mutta muunnelmassa käytetäänkin jonoa, joka koostuu symboleista x ja pareista (p, l). Näitä kutsutaan tässä *LZ77'-parametreiksi*. Parametri x koodaa symbolin x, jota kutsutaan myös *literaaliksi*, ja pari (p, l) pituudeltaan l olevan toiston alkaen kohdasta p. Kohta p ilmoitetaan muunnelmassa etäisyytenä, joten esimerkiksi $(p, l) = (1, 1)$ tarkoittaa, että toistetaan edellinen merkki.

Pakatun palan muoto on

$$ft\Phi z_1 \cdots z_k z,$$

missä $f \in \beta$ ja $t \in \{\bar{0}\bar{1}, \bar{1}\bar{0}\}$ ovat ”viimeinen pala”-bitti ja palan tyyppi kuten yllä. Dynaamisen sanaston palalla sanaan $\Phi \in \beta^*$ on pakattu jäljempänä käytettävät koodauskaaviot ja kiinteän sanaston palalla $\Phi = \varepsilon$. Sanat $z_1, \ldots, z_k \in \beta^+$ ovat koodattuja LZ77’-parametreja ja sana $z \in \beta^+$ on palan päätössymbolin koodisana.

LZ77’-parametrien ja päätössymbolin koodauksessa käytetään kahta aakkostoa:

$$E = \{e_0, e_1 \ldots, e_{285}\}$$

ja

$$D = \{d_0, d_1, \ldots, d_{29}\}.$$

Aakkostossa E on literaalit ja pituuskoodit ja aakkostossa D on etäisyyskoodit.

Oletetaan, että käytössä on kaksi injektiivistä koodauskaaviota:

$$\phi_E \colon E \to C_E$$

ja

$$\phi_D \colon D \to C_D,$$

missä koodit $C_E, C_D \subset \beta^+$ ovat prefiksikoodeja. Kiinteän sanaston paloille nämä on määrätty etukäteen ja dynaamisen sanaston paloille ne määräytyvät sanan Φ perusteella. Kuvaus sanan Φ muodosta on kohdassa 3.4.4.

Kunkin sanan z_i muoto on joko

$$z_i = \phi_E(e_n), \tag{3.2}$$

missä $n \in \{0, \ldots, 255\}$, tai

$$z_i = \phi_E(e_n) x_E \phi_D(d_m) x_D, \tag{3.3}$$

missä $n \in \{257, \ldots, 285\}$, $m \in \{0, \ldots, 29\}$, $x_E \in \beta^{X_E(n)}$, $x_D \in \beta^{X_D(m)}$ ja

funktioiden

$$X_E \colon \{257, \ldots, 285\} \to \mathbb{N}$$

ja

$$X_D \colon \{0, \ldots, 29\} \to \mathbb{N}$$

arvot määräytyvät sivun 31 Taulukoiden 1 ja 2 perusteella. Päätössymbolin koodisana on

$$z = \phi_E(e_{256}). \tag{3.4}$$

Bittijonojen x_E ja x_D pituus määräytyy siis edeltävän koodatun symbolin perusteella. Ajateltuna kaksikantaisina lukuesityksinä ne voidaan dekoodata luvuiksi, joita merkitään tässä $(x_E)_2 \in \mathbb{N}$ ja $(x_D)_2 \in \mathbb{N}$. Tyhjälle bittijonolle ε_β on $(\varepsilon_\beta)_2 = 0$.

Koska koodit C_E ja C_D ovat prefiksikoodeja ja koodauskaaviot ϕ_E ja ϕ_D injektiivisiä, niin koodatut muodot (3.2), (3.3) ja (3.4) voidaan erottaa toisistaan. Lisäksi voidaan myös dekoodata näistä luvut n ja m sekä bittijonojen x_E ja x_D arvot.

Määritellään funktiot $\lambda \colon \{257, \ldots, 285\} \times \mathbb{N} \to \mathbb{N}$ ja $\delta \colon \{0, \ldots, 29\} \times \mathbb{N} \to \mathbb{N}$ seuraavasti:

$$\lambda(n, x) = \lambda_0(n) + x$$

ja

$$\delta(m, x) = \delta_0(m) + x,$$

missä

$$\lambda_0(n) = 3 + \sum_{i=257}^{n-1} 2^{X_E(i)}$$

ja

$$\delta_0(m) = 1 + \sum_{i=0}^{m-1} 2^{X_D(i)}.$$

Funktioiden λ_0 ja δ_0 arvot ovat myös Taulukoissa 1 ja 2.

Muotoa (3.2) oleva z_i-sana koodaa (literaalin) tavun $\mathbf{b}_n \in \mathcal{B}$, ja muotoa (3.3) oleva z_i-sana koodaa LZ77'-parametrin (p, l), missä etäisyys $p = \delta(m, (x_D)_2)$ ja pituus $l = \lambda(n, (x_E)_2)$.

Näin koodattujen (p, l) parien etäisyyden p pienin arvo on $\delta(0, 0) = 1$ ja

Taulukko 1: Funktiot X_E ja λ_0

n	$X_E(n)$	$\lambda_0(n)$	n	$X_E(n)$	$\lambda_0(n)$	n	$X_E(n)$	$\lambda_0(n)$
257	0	3	267	1	15	277	4	67
258	0	4	268	1	17	278	4	83
259	0	5	269	2	19	279	4	99
260	0	6	270	2	23	280	4	115
261	0	7	271	2	27	281	5	131
262	0	8	272	2	31	282	5	163
263	0	9	273	3	35	283	5	195
264	0	10	274	3	43	284	5	227
265	1	11	275	3	51	285	0	258
266	1	13	276	3	59			

Taulukko 2: Funktiot X_D ja δ_0

m	$X_D(m)$	$\delta_0(m)$	m	$X_D(m)$	$\delta_0(m)$	m	$X_D(m)$	$\delta_0(m)$
0	0	1	10	4	33	20	9	1025
1	0	2	11	4	49	21	9	1537
2	0	3	12	5	65	22	10	2049
3	0	4	13	5	97	23	10	3073
4	1	5	14	6	129	24	11	4097
5	1	7	15	6	193	25	11	6145
6	2	9	16	7	257	26	12	8193
7	2	13	17	7	385	27	12	12289
8	3	17	18	8	513	28	13	16385
9	3	25	19	8	769	29	13	24577

suurin arvo $\delta(29, 2^{13} - 1) = 32768$ sekä pituuden l pienin arvo on $\lambda(257, 0) = 3$ ja suurin arvo on $\lambda(285, 0) = 258$. Myös kaikki arvot näiden välillä pystytään koodaamaan.

Deflate-dekooderi tarvitsee siis vähintään 32768 tavun puskurin, jotta se voi kopioida merkkejä niin kaukaa. Puskuria ei tyhjennetä palojen välissä vaan pari (p, l) voi viitata myös edellisiin paloihin.

3.4.4 Koodauskaavioiden pakkaus

Tutkimalla Huffmanin koodausalgoritmia 3.2.14 huomataan, että sillä voi muodostaa useamman erilaisen koodauskaavion. Esimerkiksi jos algoritmin binääripuun muodostamisen vaiheessa solmujen listassa on useampi kuin kaksi solmua, joilla on pienin todennäköisyys, niin joudutaan valitsemaan, että mitkä kaksi näistä otetaan uuden solmun lapsiksi. Puun kaarien merkitsemisessä joudutaan valitsemaan, että miten päin merkit $\bar{0}$ ja $\bar{1}$ merkitään solmun ja sen lapsien välisiin kaariin.

Kaikki Huffmanin koodausalgoritmilla muodostetut koodauskaaviot ovat kyllä optimaalisia, mutta koodauskaavioiden pakkaamiseksi määritellään menetelmä, jolla koodauskaaviot määräytyvät täysin pelkästään symbolien koodisanojen pituuksista.

Oletetaan seuraavaksi, että symboleille $X = (x_1, \ldots, x_n)$ ja aakkostolle A on määritelty järjestys relaatiolla $<$, jolloin myös joukolle A^* voidaan määritellä leksikografinen järjestys, jota merkitään tässä myös symbolilla $<$.

3.4.1 Määritelmä. Injektiivinen koodauskaavio $\phi\colon X \to C$, missä $C \subset A^+$, on *normalisoitu*, jos se toteuttaa seuraavat kolme ehtoa:

(i) C on prefiksikoodi.

(ii) Koodin C lyhyemmät koodisanat ovat (joukon A^* leksikografisessa järjestyksessä) ennen pidempiä, eli kaikilla $u, v \in C$ on $u < v$, jos $|u| < |v|$.

(iii) Jos kahdella symbolilla on samanmittainen koodisana, niin (aakkoston X järjestyksessä) pienemmän symbolin koodisana on (joukon A^* leksikografisessa järjestyksessä) ennen suuremman symbolin koodisanaa, eli $\phi(x) < \phi(y)$, jos $|\phi(x)| = |\phi(y)|$ ja $x < y$.

Seuraavaksi esitettävä algoritmi on keskeisessä osassa koodauskaavioiden pakkauksessa. Sen esityksessä käytetään merkintää $\mathbb{N}_+ = \{1, 2, 3, \ldots\}$.

3.4.2 Algoritmi. Määritellään *pituuskoodausalgoritmi*, jonka syötteenä on vektori koodisanojen pituuksia $(l_1, \ldots, l_n) \in {\mathbb{N}_+}^n$, missä $n \geq 1$, ja tuloksena koodauskaavio $\phi\colon \{1, \ldots, n\} \to C$, jolle $C \subset A^+$ ja $|\phi(i)| = l_i$ kaikilla $i = 1, \ldots, n$. Algoritmin määrittelemälle *pituuskoodausfunktiolle* käytetään merkintää

$$\mathcal{L}_n\colon {\mathbb{N}_+}^n \to (\{1, \ldots, n\} \to C).$$

Algoritmi koostuu seuraavista vaiheista:

1. Kaikilla $k \in \mathbb{N}$, merkitään k-pituisten koodisanojen määrää symbolilla

$$c_k = |\{\, i \in \{1, \ldots, n\} \mid l_i = k \,\}|\,.$$

2. Määritellään funktio $f\colon \{l_1, \ldots, l_n\} \to \mathbb{N}$ asettamalla

$$f(k) = \sum_{i=1}^{k-1} \alpha^{k-i} c_i,$$

   missä $\alpha = |A|$, ja funktio $s\colon \{1, \ldots, n\} \to \mathbb{N}$ asettamalla

$$s(i) = |\{\, j \in \{1, \ldots, i-1\} \mid l_j = l_i \,\}|\,.$$

3. Olkoon $t_i = f(l_i) + s(i)$ kaikilla $i \in \{1, \ldots, n\}$.

4. Koodauskaavio $\phi\colon \{1, \ldots, n\} \to C$ voidaan nyt määritellä asettamalla

$$\phi(i) = \mathrm{radix}_A(t_i, l_i) \in A^*,$$

   missä $\mathrm{radix}_A(t_i, l_i)$ tarkoittaa luvun t_i esitystä α-kantaisena l_i-pituisena lukuna käyttäen aakkoston A merkkejä.

Toinen mahdollinen toteutus pituuskoodausfunktiolle on esitetty Koodilistauksessa 3.4.

Koodilistaus 3.4 Pituuskoodausalgoritmi

```
1: funktio L_n(l_1, ..., l_n)
2:     M ← max{l_1, ..., l_n}
3:     kaikilla k ← 0, ..., M tee
4:         c_k ← |{ i ∈ {1, ..., n} | l_i = k }|
5:     q ← 0
6:     kaikilla k ← 1, ..., M tee
7:         q ← (q + c_{k−1})α
8:         s_k ← q
9:     kaikilla i ← 1, ..., n tee
10:        k ← l_i
11:        φ(i) ← radix_A(s_k, k)      ▷ Määrittele φ(i)
12:        s_k ← s_k + 1
13:    palauta φ
```

Esitetään seuraavaksi ns. McMillanin lause, jota tarvitaan seuraavan lemman todistuksessa. Lauseessa esiintyvä epäyhtälö (3.5) on nimeltään Kraftin epäyhtälö.

3.4.3 Lause (McMillan)**.** *Olkoon A aakkosto ja $\alpha = |A|$. Jos $C \subset A^+$ on yksikäsitteisesti dekoodattavissa oleva koodi, josta voidaan valita sellaiset m eri koodisanaa, joiden pituudet ovat $l_1, \ldots, l_m$, niin*

$$\sum_{i=1}^{m} \alpha^{-l_i} \leq 1. \tag{3.5}$$

Todistus. Sivuutetaan. Ks. esimerkiksi [Yli06, Lause 3.3.1]. □

3.4.4 Lemma. *Olkoon $\{u_1, u_2, \ldots, u_n\} \subset A^+$ yksikäsitteisesti dekoodattavissa oleva koodi. Tällöin pituuskoodausfunktiolla saatu koodauskaavio*

$$\phi = \mathcal{L}_n(|u_1|, \ldots, |u_n|)$$

on normalisoitu.

Todistus. Kutakin pituutta k kohden pituuskoodausalgoritmin 3.4.2 luvut t_i, missä $l_i = k$, käyvät järjestyksessä kaikki luvut

$$f(k),\ f(k)+1,\ \ldots,\ f(k)+c_k-1, \tag{3.6}$$

joten jos nämä ovat pienempiä kuin α^k, jolloin ne kaikki voidaan esittää α-kantaisina lukuina k merkillä, niin normalisoinnin määritelmän ehto (iii) täyttyy. Kraftin epäyhtälöstä (3.5) saadaan

$$\sum_{i=1}^{\infty} \alpha^{-i} c_i \leq 1,$$

minkä perusteella nähdään, että

$$f(k) = \alpha^k \left(\sum_{i=1}^{\infty} \alpha^{-i} c_i - \sum_{i=k}^{\infty} \alpha^{-i} c_i \right) \leq \alpha^k - \sum_{i=k}^{\infty} \alpha^{k-i} c_i \leq \alpha^k - c_k.$$

Jonon (3.6) suurimmalle alkiolle saadaan siis yläraja

$$f(k) + c_k - 1 \leq \alpha^k - 1 < \alpha^k,$$

mikä takaa, että kaikki luvut t_i voidaan esittää $\mathrm{radix}_A(t_i, l_i)$-muodossa.

Olkoon nyt koodi C koodauskaavion ϕ arvojoukko. Merkitään kaikilla $k \in \mathbb{N}$ ja $i = 1, \ldots, c_k$

$$w(k, i) = \mathrm{radix}_A(f(k) + i - 1, k),$$

jolloin siis edellisen perusteella sanat $w(k, 1), \ldots, w(k, c_k)$ ovat leksikografisessa järjestyksessä pienimmästä suurimpaan ja

$$C = \bigcup_{k \in \mathbb{N}} \bigcup_{i=1}^{c_k} \{w(k, i)\}.$$

Funktiolle f saadaan helposti myös rekursiokaava

$$f(k) = (f(k-1) + c_{k-1})\alpha. \tag{3.7}$$

Jos koodissa C pisimmät sanat, jotka ovat pituutta k lyhyempiä, ovat pituudeltaan $k - m$, niin $c_{k-m+1} = c_{k-m+2} = \ldots = c_{k-1} = 0$ ja rekursiomuodosta (3.7) saadaan

$$f(k) = (f(k-m) + c_{k-m})\alpha^m.$$

Tällöin sanan $w(k,1)$ pituutta $k-m$ oleva prefiksi on

$$\mathrm{radix}_A(f(k-m)+c_{k-m}, k-m),$$

joka on suurempi kuin leksikografisessa järjestyksessä suurin pituutta $k-m$ oleva sana $w(k-m, c_{k-m})$. Täten siis normalisoinnin määritelmän ehto (ii) täyttyy. Toisaalta, koska jokaisen sanan j-pituinen aito prefiksi on aina suurempi kuin mikään j-pituinen sana koodissa C, niin C on myös prefiksikoodi ja siis myös normalisoinnin määritelmän ehto (i) täyttyy. □

Edellisen lemman perusteella siis pelkät koodisanojen pituudet määrittelevät normalisoidun koodauskaavion. Lisäksi, jos jokin symboli ei ole ollenkaan käytössä, niin se voidaan ilmaista koodisanan pituudella 0. Siis jos joukon $X = \{x_1, \ldots, x_n\}$ symboleista on käytössä osajoukko $X' \subset X$, niin vektori $(l_1, \ldots, l_n) \in \mathbb{N}^n$, missä $l_i > 0$ jos ja vain jos $x_i \in X'$, määrittelee yksikäsitteisesti normalisoidun koodauskaavion $\phi : X' \to C$. Laajennetaan tässä kuitenkin funktion ϕ määrittelyjoukko koko joukkoon X asettamalla $\phi(x) = \varepsilon$ kaikilla $x \in X \setminus X'$.

Kohdassa 3.4.3 mainittiin, että koodauskaaviot ϕ_E ja ϕ_D ovat pakattu sanaan Φ. Pakkaus perustuu koodauskaavioiden määrittelemiseen koodisanojen pituuksien avulla. Koodauskaavioiden ϕ_E ja ϕ_D koodisanojen pituuksien koodauksessa käytetään aakkostoa $Q = \{q_0, q_1, \ldots, q_{18}\}$. Sen koodauskaaviota merkitään tässä $\phi_Q \colon Q \to C_Q$. Symbolit $q_0, \ldots, q_{15}$ tulkitaan koodisanapituuksina $0, \ldots, 15$. Kolmella viimeisellä symbolilla on erityismerkitys. Symbolia q_{16} seuraa kaksi lisäbittiä ja näiden tulkinta on, että edellistä koodisanapituutta toistetaan 3–6 kertaa. Symbolia q_{17} seuraa kolme lisäbittiä ja symbolia q_{18} seuraa 7 lisäbittiä. Nämä tulkitaan koodipituuden 0 toistoina 3–10 kertaa symbolilla q_{17} ja 11–138 kertaa symbolilla q_{18}.

3.4.5 Esimerkki. Bittijono

$$\phi_Q(q_{17})\overline{0}\overline{0}\overline{1}\phi_Q(q_5)\phi_Q(q_{16})\overline{0}\overline{1}\phi_Q(q_3)\phi_Q(q_{18})\overline{0}\overline{0}\overline{0}\overline{0}\overline{0}\overline{0}\overline{0}\phi_Q(q_{10})\phi_Q(q_0)\phi_Q(q_2)$$

tulkitaan koodisanapituuksien jonona

$$0,0,0,0,\ 5,\ 5,5,5,5,\ 3,\ 0,0,0,0,0,0,0,0,0,0,0,\ 10,\ 0,\ 2.$$

Nyt voidaan kuvailla miten koodauskaaviot ϕ_E ja ϕ_D saadaan dekoodauttua sanasta $\Phi \in \beta^+$.

3.4.6 Algoritmi. Deflate-menetelmässä käytettävä *koodauskaavioiden dekoodausalgoritmi* on seuraava:

1. Lue koodien E, D ja Q koodisanojen määrät m_E, m_D ja m_Q. Näiden arvovälit ovat $m_E \in \{257, \ldots, 286\}$, $m_D \in \{1, \ldots, 30\}$ ja $m_Q \in \{4, \ldots, 19\}$. Luvut ovat koodattu kiinteillä bittipituuksilla: m_E:lle 5 bittiä, m_D:lle 5 bittiä ja m_Q:lle 4 bittiä.

2. Lue m_Q kappaletta kolmebittisiä koodisanojen pituuksia järjestyksessä 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15 ja muodosta näistä vektori $\lambda_Q \in \mathbb{N}^{|Q|}$, jossa loput $|Q| - m_Q$ alkiota ovat 0.

3. Muodosta normalisoitu koodauskaavio ϕ_Q vektorista λ_Q.

4. Dekoodaa koodauskaaviolla ϕ_Q tietovirrasta m_E kappaletta koodisanojen pituuksia ja muodosta vektori λ_E. Dekoodaa samoin m_D kappaletta koodisanojen pituuksia ja muodosta vektori λ_D.

5. Muodosta normalisoidut koodauskaaviot ϕ_E ja ϕ_D vektoreista λ_E ja λ_D.

Menetelmä asettaa koodauskaavioiden ϕ_E ja ϕ_D koodisanojen pituudelle ylärajan 15. Täten Huffmanin koodausalgoritmia ei voi soveltaa ihan suoraan, koska sillä saattaisi tulla jollekin symbolille yli 15 bittinen koodisana. Muunneltua Huffmanin koodausalgoritmia ei kuitenkaan esitetä tässä. Sen löytää esimerkiksi zlib-toteutuksen lähdekoodista lähteestä [RA12].

4 Tiedostojen erojen vertailu

Versiohallinnassa tiedostojen erojen vertailu on tärkeä toiminto. Sitä tarvitaan esimerkiksi saman tiedoston vanhemman ja uudemman version vertailemisessa tai siihen, että voidaan listata ne muutokset tiedostoihin, joita ei ole vielä tallennettu versiohallintaan. Eräs paljon käytetty tapa lähettää muutoksia Git-versiohallintaa käyttävään projektiin on tehdä muutoksista korjaustiedosto (patch) ja lähettää se sähköpostilla projektin muille kehittäjille. Korjaustiedoston muodostamisessa tarvitaan tiedostojen erojen vertailua. Kahden eri kehityshaaran yhdistämisessä (branch merging) joudutaan toisinaan tilanteeseen, jossa samaa tiedostoa on muutettu kahdella eri tavalla. Myös tällöin erojen vertailu pitää pystyä tekemään, jotta voidaan havaita ovatko muutokset tehty tiedoston eri kohtiin, jolloin molempien kehityshaarojen muutokset voidaan yhdistää automaattisesti.

Tiedostojen erojen vertailu tapahtuu yleensä riveittäin. Tällöin tiedoston sisältö $a \in \mathcal{B}^*$, joka on siis tavujono, mallinnetaan sanana

$$a = a_1 a_2 \cdots a_n \in \Sigma^*,$$

missä aakkosto $\Sigma = \mathcal{B}^*$ ja merkit $a_1, a_2, \ldots, a_n \in \Sigma$ ovat tiedoston rivit, joista kaikki paitsi mahdollisesti a_n päättyvät rivinvaihtomerkkiin. Aakkosto Σ on siis tässä tapauksessa ääretön. Tämän mallinnuksen avulla tiedostojen erojen vertailu voidaan käsitellä yleisesti sanojen erojen vertailuna, joka on puolestaan kiinnostava aihe myös versiohallinnan ulkopuolella (esimerkiksi DNA-ketjujen vertailussa), joten siihen liittyvää tutkimusta on tehty paljon [Mye86].

4.1 Sanojen vertailun määritelmiä

Tässä osiossa on käytössä aakkosto Σ, joka voi olla äärellinen tai ääretön. Olkoon sanat $a, b \in \Sigma^*$ ja asetetaan $n = |a|$ sekä $m = |b|$.

4.1.1 Määritelmä. Sanan $a \in \Sigma^*$ *muokkauskaavio* (edit script) on pari (e_D, e_I), missä e_D on joukko

$$e_D \subset \{1, 2, \ldots, |a|\}$$

ja e_I on funktio

$$e_I \colon \{0, 1, 2, \ldots, |a|\} \to \Sigma^*.$$

Sanaa a sanotaan tällöin muokkauskaavion (e_D, e_I) *lähtösanaksi*.

Määritellään seuraavaksi miten muokkauskaaviota käytetään.

4.1.2 Määritelmä. Olkoon $e = (e_D, e_I)$ jokin sanan a muokkauskaavio. Sanan a *muokkaus* muokkauskaaviolla e on sana

$$p(a, e) = e_I(0)d_1e_I(1)d_2e_I(2)\cdots d_ne_I(n) \in \Sigma^*,$$

missä $n = |a|$ ja

$$d_i = \begin{cases} \varepsilon, & \text{jos } i \in e_D \\ a[i], & \text{jos } i \notin e_D \end{cases}$$

kaikilla $i = 1, \ldots, n$.

Sanan a muokkauskaavion (e_D, e_I) joukko e_D kertoo siis mitkä sanan a merkit poistetaan ja funktio e_I kertoo mitä merkkejä sanan a päihin ja merkkien väleihin lisätään.

4.1.3 Merkintä. Muokkauskaavion lisäysfunktiolle käytetään myös merkintää

$$e_I = \{i_1 \to u_1, i_2 \to u_2, \ldots, i_k \to u_k\},$$

jolloin $e_I(i_j) = u_j \in \Sigma^*$ kaikilla $j = 1, \ldots, k$ ja $e_I(i) = \varepsilon$ kaikilla $i \notin \{i_1, \ldots, i_k\}$.

4.1.4 Esimerkki. Muodostetaan muokkauskaavio e, joka muokkaa sanan $a =$ `aakkosto` sanaksi $b =$ `pakkaus`. Limitetään sanat seuraavasti:

sanan a kohta	1		2	3	4	5			6	7	8
$a =$	a		a	k	k	o			s	t	o
$b =$		p	a	k	k		a	u	s		

Tästä nähdään, että muokkauskaavio

$$e = (\{1, 5, 7, 8\}, \{1 \to \texttt{p}, 5 \to \texttt{au}\})$$

antaa halutun lopputuloksen: $p(a, e) = b$.

4.1.5 Määritelmä. Muokkauskaavion $e = (e_D, e_I)$ *koko* on luku

$$|e| = |e_D| + \sum_{i=0}^{|a|} |e_I(i)|\,,$$

missä a on muokkauskaavion e lähtösana.

4.1.6 Määritelmä. Sanojen a ja b välinen muokkauskaavio e on *optimaalinen*, jos sen koko on pienin mahdollinen eli kaikkien yhtälön $p(a, e') = b$ toteuttavien muokkauskaavioiden e' koko $|e'|$ on suurempi tai yhtä suuri kuin $|e|$. Sanojen a ja b välinen *muokkausetäisyys* (edit distance) on niiden välisen optimaalisen muokkauskaavion koko.

4.1.7 Esimerkki. Esimerkin 4.1.4 muokkauskaavion e koko on $|e| = 7$. Myös esimerkiksi muokkauskaavio

$$e' = (\{1, 2, 3, 4, 5, 6, 7, 8\}, \{1 \to \texttt{pakkaus}\})$$

muokkaa edellisen esimerkin sanasta a sanan b, mutta se on kooltaan suurempi, sillä $|e'| = |a| + |b| = 15$.

Näiden määritelmien avulla sanojen erojen vertailu voidaan formalisoida seuraavaan muotoon.

4.1.8 Ongelma. *Etsi sanojen $a \in \Sigma^*$ ja $b \in \Sigma^*$ välinen optimaalinen muokkauskaavio.*

Esimerkistä 4.1.7 nähdään, että ainakin voidaan aina muodostaa muokkauskaavio e', joka on kooltaan $|e'| = |a| + |b|$ ja toteuttaa yhtälön $p(a, e') = b$. Jos sanoista a ja b löytyy yhteisiä merkkejä, jotka ovat samassa järjestyksessä, niin niitä ei tarvitse poistaa ja lisätä uudelleen, jolloin saadaan kooltaan pienempi muokkauskaavio. Kyseessä on siis toisaalta sanojen a ja b pisimmän yhteisen alijonon (longest common subsequence, LCS) etsimisestä.

4.1.9 Määritelmä. Sanojen a ja b *yhteinen alijono* on sellainen sana

$$a[i_1] \cdots a[i_k] = b[j_1] \cdots b[j_k],$$

missä $1 \leq i_1 < \ldots < i_k \leq |a|$ ja $1 \leq j_1 < \ldots < j_k \leq |b|$.

4.1.10 Algoritmi. Yhteisten merkkien sijainneista $(i_1, j_1), \ldots, (i_k, j_k)$ saadaan helposti tehtyä muokkauskaavio, joka muokkaa sanasta a sanan b:

$(e_D, e_I) \leftarrow (\{\}, \{\})$
$(i_0, j_0) \leftarrow (0, 0)$
$(i_{k+1}, j_{k+1}) \leftarrow (|a| + 1, |b| + 1)$
kaikilla $x \leftarrow 1, 2, \ldots, k + 1$ **tee**
    $e_D \leftarrow e_D \cup \{i_{x-1} + 1, i_{x-1} + 2, \ldots, i_x - 1\}$
    $e_I \leftarrow e_I \cup \{i_{x-1} \rightarrow b[j_{x-1} + 1 \ldots j_x - 1]\}$

Näin muodostettu muokkauskaavio $e = (e_D, e_I)$ toteuttaa yhtälön $p(a, e) = b$.

4.2 Pisin yhteinen alijono dynaamisella ohjelmoinnilla

Sanojen a ja b pisin yhteinen alijono on helppo löytää rekursiivisesti:

funktio PYAREKURSIO(a, b)
    jos $|a| = 0$ tai $|b| = 0$ **niin**
        palauta $\{\}$
    muuten jos $a[|a|] = b[|b|]$ **niin**
        palauta PYAREKURSIO$(a[1 \ldots |a| - 1], b[1 \ldots |b| - 1]) \cup \{(|a|, |b|)\}$
    muuten
        $p_1 \leftarrow$ PYAREKURSIO$(a[1 \ldots |a| - 1], b)$
        $p_2 \leftarrow$ PYAREKURSIO$(a, b[1 \ldots |b| - 1])$
        jos $|p_1| > |p_2|$ **niin**
            palauta p_1
        muuten
            palauta p_2

Pahimmassa tapauksessa ($a[i] \neq b[j]$ kaikilla $i = 1, \ldots, n$ ja $j = 1, \ldots, m$) rekursiivisen ratkaisu tekee rivin 4 vertailuja $\binom{n+m}{n} - 1$ kertaa. Jos sanojen a ja b pituudet ovat samaa suuruusluokkaa $O(n)$, niin Stirlingin kaavalla algoritmin aikakompleksisuus saadaan muotoon $O(4^n/\sqrt{\pi n})$ [Wik12a]. Eksponentiaalisesta aikakompleksisuudesta päästään kuitenkin huomattavasti käytännöllisempään $O(n^2)$-kompleksisuuteen käyttämällä dynaamista ohjelmointia (dynamic programming).

Dynaamisen ohjelmoinnin idea on, että ongelma jaetaan osaongelmiin, jotka

ratkaistaan yksitellen järjestyksessä pienimmästä suurimpaan. Tällöin ratkaistaessa suurempaa ongelmaa voidaan hyödyntää jo ratkaistujen pienempien ongelmien tuloksia. [DPV06, s. 170]

Dynaamisella ohjelmoinnilla toteutettu pisimmän yhteisen alijonon hakeva algoritmi on esitetty Koodilistauksessa 4.1. Se laskee taulukon soluun (i, j) sanojen $a[1 \ldots i]$ ja $b[1 \ldots j]$ pisimmän yhteisen alijonon pituuden ja lopuksi selvittää pisimmän yhteisen alijonon käymällä tarvittavat taulukon solut vielä kertaalleen läpi. Algoritmin taulukon täyttävä osa on esitetty ja todistettu oikeaksi myös lähteessä [Hir75]. Algoritmin aika- ja tilakompleksisuus on esitetyssä muodossa $O(nm)$ tai $O(n^2)$ sanojen a ja b pituuksien ollessa samaa suuruusluokkaa. Taulukon soluun olisi voitu laskea suoraan itse pisin yhteinen alijono, mutta silloin taulukon tarvitsema tila olisi $O(n^3)$ (neliöpyramidiluku).

Daniel S. Hirschberg on kehittänyt algoritmista version, jonka aikakompleksisuus on sama $O(nm)$, mutta tilakompleksisuus on lineaarinen $O(n+m)$ [Hir75]. Sitä ei kuitenkaan esitetä tässä, sillä seuraavassa kohdassa esitetty versio on yleensä vielä käytännöllisempi.

Koodilistaus 4.1 Pisin yhteinen alijono dynaamisella ohjelmoinnilla

```
1:  funktio PYADYNAAMINEN(a, b)
2:      t ← LUOTAULUKKO(|a| + 1, |b| + 1)      ▷ Soluun t[i, j] lasketaan sanojen
                                                  a[1 ... i] ja b[1 ... j] pya:n pituus
3:      kaikilla i ← 0, 1, 2, ..., |a| tee
4:          t[i, 0] ← 0
5:      kaikilla j ← 1, 2, ..., |b| tee
6:          t[0, j] ← 0
7:      kaikilla i ← 1, 2, ..., |a| tee
8:          kaikilla j ← 1, 2, ..., |b| tee
9:              jos a[i] = b[j] niin
10:                 t[i, j] ← t[i − 1, j − 1] + 1
11:             muuten
12:                 t[i, j] ← max(t[i − 1, j], t[i, j − 1])
13:     L ← {}
14:     (i, j) ← (|a| , |b|)
15:     niin kauan kun i ≠ 0 ja j ≠ 0 tee
16:         jos a[i] = b[j] niin
17:             L ← {(i, j)} ∪ L
18:             (i, j) ← (i − 1, j − 1)
19:         muuten jos t[i − 1, j] ≥ t[i, j − 1] niin
20:             i ← i − 1
21:         muuten
22:             j ← j − 1
23:     palauta L
```

4.3 Erojen vertailu O(nd)-ajassa

Git käyttää tiedostojen vertailuun hieman muunneltua LibXDiff-kirjastoa [Tor06], jonka tiedostojen erojen vertailun toteutus perustuu Eugene Myersin artikkeliin "An O(ND) Difference Algorithm and Its Variations" [Libx12]. Seuraavassa on esitetty artikkelissa johdettu pisimmän yhteisen alijonon laskeva algoritmi ja perustelut sen toimivuudesta. Esitys perustuu kyseiseen artikkeliin eli lähteeseen [Mye86].

Monessa sanojen vertailun käytännön sovelluksessa verrattavat sanat a ja b poikkeavat toisistaan vain vähän, eli sanojen a ja b välinen muokkausetäisyys d on pieni suhteessa sanojen pituuksiin n ja m, jolloin n ja m ovat samaa suuruusluokkaa, sillä $|n - m|$ on korkeintaan d. Osion lopussa esitettävä algoritmi sopii juuri tämäntyyppisiin tilanteisiin, sillä sen aikakompleksisuus on $O(nd)$.

4.3.1 Muokkausgraafi ja muita käsitteitä

Algoritmin johtamiseen käytetään muutamia seuraavaksi määriteltäviä käsitteitä, jotka yhdistävät sanojen vertailun ongelman graafiteoriaan.

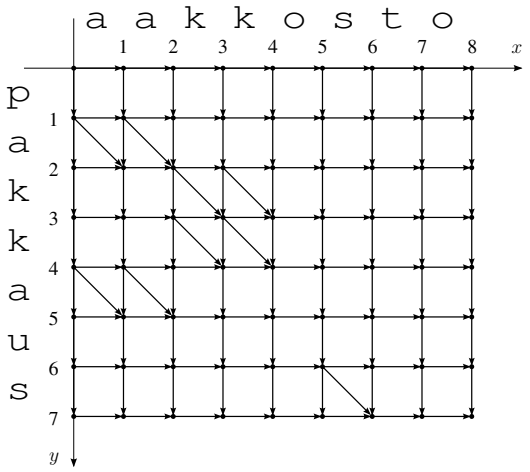
4.3.1 Määritelmä. Sanaparin (a, b) *muokkausgraafi* on suunnattu graafi (V, E), joka määräytyy seuraavasti:

(i) Solmujen joukko V koostuu tason pisteistä (x, y), missä $x = 0, 1, 2, \ldots, |a|$ ja $y = 0, 1, 2, \ldots, |b|$.

(ii) Kaarien joukko E sisältää kaikki kaaret vasemmalta oikealle ja ylhäältä alas eli kaaret $(x, y) \to (x+1, y)$ kaikilla $x = 0, \ldots, |a|-1$ ja $y = 0, \ldots, |b|$ sekä kaaret $(x, y) \to (x, y + 1)$ kaikilla $x = 0, \ldots, |a|$ ja $y = 0, \ldots, |b| - 1$. Lisäksi kaarien joukko sisältää diagonaaliset kaaret $(x - 1, y - 1) \to (x, y)$ kaikilla x ja y, joilla $a[x] = b[y]$.

4.3.2 Esimerkki. Kuvassa 4.1 on esitetty sanaparin (`aakkosto`, `pakkaus`) muokkausgraafi.

Jokaisesta sanaparin (a, b) muokkausgraafin polusta, joka alkaa origosta ja päättyy pisteeseen (n, m), voidaan johtaa muokkauskaavio, joka muokkaa sanasta a sanan b. Polun vaakasuorat kaaret vastaavat sanan a merkkien poistoja



Kuva 4.1: Muokkausgraafi

ja pystysuorat kaaret vastaavat sanan b merkkien lisäyksiä. Diagonaaliset kaaret eivät tule muokkauskaavioon ollenkaan, joten muokkauskaaviosta saadaan mahdollisimman pieni käyttämällä suurin mahdollinen määrä diagonaalisia kaaria eli pienin mahdollinen määrä ei-diagonaalisia kaaria.

4.3.3 Määritelmä. Muokkausgraafin polkua, jossa on tarkalleen d ei-diagonaalista kaarta, kutsutaan *d-poluksi*. Selvästi jokainen d-polku koostuu $(d-1)$-polusta, jota seuraa yksi ei-diagonaalinen kaari ja diagonaalikaarien jono, jota kutsutaan *madoksi*. Mato voi olla siis pituudeltaan 0 kaarta tai enemmän.

4.3.4 Määritelmä. Origosta alkava ja pisteeseen (n, m) päättyvä d-polku on *optimaalinen*, jos luvun d arvo on pienin mahdollinen, eli millään d'-polulla, joka alkaa origosta ja päättyy pisteeseen (n, m), ei ole $d' < d$.

Näiden määritelmien avulla Ongelma 4.1.8 voidaan muotoilla seuraavaan ekvivalenttiin muotoon:

4.3.5 Ongelma. *Etsi optimaalinen d-polku sanaparin (a, b) muokkausgraafista.*

4.3.2 Muokkausetäisyyden laskeva algoritmi

Optimaalisen d-polun etsimiseen tarvitaan vielä kuitenkin muutamia aputuloksia. Seuraavien kolmen lemman avulla saadaan johdettua algoritmi, jolla

voidaan helposti laskea muokkausetäisyys ja joka toimii perustana esitettävälle pisimmän yhteisen alijonon selvittävälle algoritmille.

4.3.6 Määritelmä. Muokkausgraafin *diagonaali* k koostuu pisteistä (x, y), joille $x - y = k$.

4.3.7 Lemma. *Origosta alkava d-polku päättyy diagonaalille $k \in \{-d, -d + 2, \ldots, d - 2, d\}$.*

Todistus. Origosta alkava 0-polku koostuu pelkästään diagonaalikaarista ja alkaa diagonaalilta 0, joten se päättyy diagonaalille 0. Jokainen $(d + 1)$-polku koostuu d-polusta, ei-diagonaalisesta kaaresta ja madosta. Jos kyseinen d-polku päättyy diagonaalille $k \in \{-d, -d+2, \ldots, d-2, d\}$, niin ei-diagonaalinen kaari ja sitä seuraava mato päättyvät diagonaalille $k - 1$ tai $k + 1$. Silloin $(d + 1)$-polku päättyy diagonaalille $k' \in \{-d \pm 1, -d + 2 \pm 1, \ldots, d - 2 \pm 1, d \pm 1\} = \{-d - 1, -d + 1, \ldots, d - 1, d + 1\}$, joten väite seuraa induktiolla. □

4.3.8 Määritelmä. d-polku on *pisimmälle ylettyvä* diagonaalilla k, jos sen päätepiste on diagonaalilla k ja millään muulla diagonaalille k päättyvällä d-polulla ei alku- ja loppupisteiden välinen etäisyys ole suurempi.

4.3.9 Lemma. *Origosta alkava pisimmälle ylettyvä 0-polku päättyy pisteeseen (x, x), missä $x = \min\{\, i \mid i \geq \min(n, m)$ tai $a[i + 1] \neq b[i + 1]\,\}$.*

Todistus. Selvästi x on sanojen a ja b pisimmän yhteisen prefiksin pituus. Muokkausgraafissa on origosta alkaen tarkalleen x peräkkäistä diagonaalista kaarta, joten väite on selvä. □

4.3.10 Lemma. *Kun $d > 0$, niin diagonaalilla k pisimmälle ylettyvän d-polun päätepisteeseen päästään myös joko*

(a) diagonaalilla $k-1$ pisimmälle ylettyvällä $(d-1)$-polulla ja sitä seuraavalla vaakasuuntaisella kaarella ja pisimmällä mahdollisella madolla, tai

(b) diagonaalilla $k+1$ pisimmälle ylettyvällä $(d-1)$-polulla ja sitä seuraavalla pystysuuntaisella kaarella ja pisimmällä mahdollisella madolla.

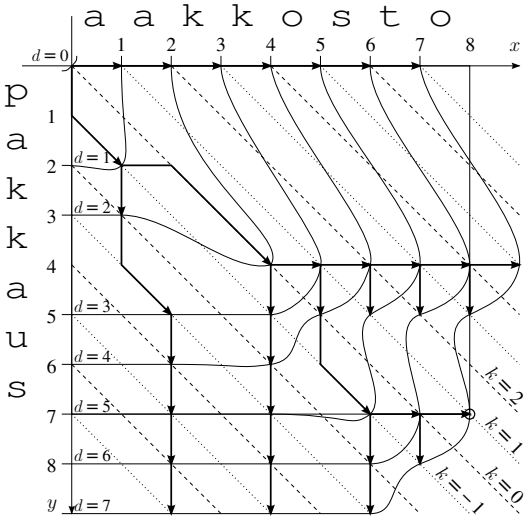
Todistus. Kuten aiemmin todettiin, d-polku koostuu $(d-1)$-polusta, ei-diagonaalisesta kaaresta ja madosta. Jos d-polku päättyy diagonaalille k, niin alun $(d-1)$-polku päättyy diagonaalille $k-1$ tai $k+1$ riippuen onko d-polun lopun matoa ennen pystysuuntainen vai vaakasuuntainen kaari. Vaikka tuo $(d-1)$-polku ei olisi pisimmälle ylettyvä diagonaalillaan, niin samalla diagonaalilla pisimmälle ylettyvä $(d-1)$-polku voidaan kuitenkin sopivalla ei-diagonaalisella kaarella jatkaa d-poluksi, joka päättyy diagonaalille k. Kun näin muodostettu d-polku jatketaan vielä pisimmällä mahdollisella madolla, niin sen päätepiste on välttämättä sama kuin alkuperäisen d-polun, sillä se ei voi olla pidemmällä alkuperäisen ollessa pisimmälle ylettyvä eikä lähempänä aloituspistettä, koska viimeinen ei-diagonaalinen kaari päättyy alkuperäisen d-polun lopun madolle. □

Käyttäen hyödyksi Lemmoja 4.3.7, 4.3.9 ja 4.3.10 on helppo muodostaa ahnas algoritmi, joka laskee sanojen muokkausetäisyyden: Lasketaan järjestyksessä arvoilla $d = 0, 1, \ldots$ kaikkien mahdollisten origosta alkavien pisimmälle ylettyvien d-polkujen päätepisteet jatkamalla aina edellisistä päätepisteistä. Ensimmäisenä tarvittava pisimmälle ylettyvän 0-polun päätepiste saadaan laskettua Lemman 4.3.9 avulla. Laskettaessa pisimmälle ylettyviä d-polkuja Lemman 4.3.7 mukaan niitä on on tarkalleen diagonaaleilla $-d$, $-d+2$, …, $d-2$ ja d ja Lemman 4.3.10 mukaan niiden päätepisteet voidaan laskea pisimmälle ylettyvien $(d-1)$-polkujen päätepisteistä. Lopulta jokin pisimmälle ylettyvä d-polku diagonaalilla $n-m$ päättyy pisteeseen (n, m), jolloin tiedetään, että muokkausetäisyys on d.

Tätä hyödyntävä muokkausetäisyyden laskeva algoritmi on esitetty Koodilistauksessa 4.2 ja sen toimintaa havainnollistava esimerkki Kuvassa 4.2.

Koodilistaus 4.2 Muokkausetäisyyden laskeva ahnas algoritmi

```
1: funktio MUOKKAUSETÄISYYS(a, b)
2:     v_1 ← 0
3:     kaikilla d ← 0, 1, ..., |a| + |b| tee
4:         kaikilla k ← −d, −d + 2, ..., d − 2, d tee
               ▷ Jatka pisimmälle ylettyvästä (d − 1)-polusta diagonaalilta k − 1 tai
                 k + 1 riippuen siitä kummalla pääse pidemmälle:
5:             jos k = −d tai (k ≠ d ja v_{k+1} > v_{k−1}) niin
6:                 x ← v_{k+1}
7:             muuten
8:                 x ← v_{k−1} + 1
9:             y ← x − k
               ▷ Muodosta pisin mahdollinen mato:
10:            niin kauan kun x < |a| ja y < |b| ja a[x + 1] = b[y + 1] tee
11:                (x, y) ← (x + 1, y + 1)
12:            v_k ← x    ▷ Tallenna diagonaalilla k pisimmälle ylettyvän d-polun
                            päätepisteen x-koordinaatti.
13:            jos x = |a| ja y = |b| niin
14:                palauta d
```



Kuva 4.2: Pisimmälle ylettyvät polut

4.3.3 Pisin yhteinen alijono keskimadon avulla

Jos Koodilistauksen 4.2 algoritmi muutetaan säilyttämään kaikki aiemmat v_k-arvot, niin sen avulla voidaan laskea myös pisin yhteinen alijono aivan kuten sivulla 43 Koodilistauksessa 4.1 esitetyssä dynaamista ohjelmointia käyttävässä algoritmissa. Täten toteutettu algoritmi olisi aikakompleksisuudeltaan $O(nd)$ ja tilakompleksisuudeltaan $O(d^2)$. Seuraavaksi johdettavalla algoritmilla päästään kuitenkin samaan aikakompleksisuuteen, mutta huomattavasti käytännöllisempään lineaariseen $O(d)$ tilakompleksisuuteen.

Esitettävä algoritmi perustuu hajota ja hallitse -menetelmään. Muokkausgraafin optimaalinen d-polku jaetaan kahteen osaan etsimällä polun keskeltä ns. keskimato, ja siten ongelma voidaan hajottaa kahteen osaan, jotka ratkaistaan rekursiivisesti.

4.3.11 Määritelmä. d-polun *keskimato* on polun alussa olevan $\lceil d/2 \rceil$-polun viimeinen mato.

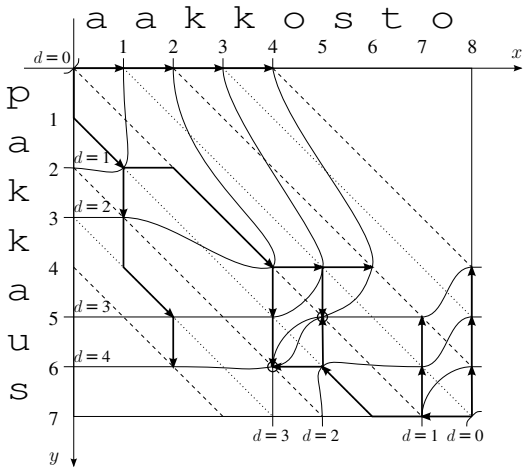
Koska d-polussa on tarkalleen $d+1$ matoa, joista osa voi olla tyhjiä, niin d-polun keskimatoa ennen on $\lceil d/2 \rceil$ matoa ja jälkeen $\lfloor d/2 \rfloor$ matoa. Keskimadon löytämiseksi käytetään hyödyksi symmetriaa. Kun muokkausgraafin jokaisen kaaren suunta käännetään ja tutkitaan origon sijaan pisteestä (n, m) alkavia polkuja, niin voidaan käyttää samoja menetelmiä pisimmälle ylettyvien polkujen löytämiseen kuin edellä. Tämän käänteisen muokkausgraafin polkuja kutsutaan *käänteisiksi d-poluiksi* ja aiemman määritelmän mukaisia *normaaleiksi d-poluiksi*, kun on tarpeen erotella kummista on kyse.

Keskimadon hakeva algoritmi saadaan muodostettua Koodilistauksen 4.2 algoritmista, kun se muutetaan muodostamaan pisimmälle ylettyvien normaalien d-polkujen lisäksi pisimmälle ylettyvät käänteiset d-polut sekä tarkistamaan milloin nämä kohtaavat. Käänteisissä poluissa pitää ottaa huomioon, että ne keskittyvät diagonaalin 0 sijaan diagonaalin $\Delta = n - m$ ympärille. Koska optimaalinen d-polku päättyy diagonaalille Δ, niin Lemman 4.3.7 perusteella muokkausetäisyys d on pariton, jos ja vain jos Δ on pariton. Siis kun Δ on pariton, niin riittää tarkistaa polkujen päällekkäisyys normaalin polun jatkamisen jälkeen, ja kun Δ on parillinen, niin riittää tarkistaa polkujen päällekkäisyys käänteisen polun jatkamisen jälkeen. Keskimadon hakeva algoritmi on esitetty Koodilistauksessa 4.3 ja todistettu oikeaksi Lemmassa 4.3.14.

Koodilistaus 4.3 Keskimadon hakeva algoritmi

```
1:  funktio KESKIMATO(a, b)
2:      Δ ← |a| − |b|
3:      v_1 ← 0      ▷ v_k = normaalin polun päätepisteen x diagonaalilla i
4:      u_1 ← |a| + 1      ▷ u_k = käänteisen polun päätepisteen x diagonaalilla Δ + k
5:      kaikilla d ← 0, 1, 2, ..., ⌈(|a| + |b|)/2⌉ tee
6:          kaikilla k ← −d, −d + 2, ..., d − 2, d tee
7:              jos k = −d tai (k ≠ d ja v_{k+1} > v_{k−1}) niin
8:                  x ← v_{k+1}
9:              muuten
10:                 x ← v_{k−1} + 1
11:             y ← x − k
12:             (x_0, y_0) ← (x, y)
13:             niin kauan kun  x < |a| ja y < |b| ja a[x + 1] = b[y + 1] tee
14:                 (x, y) ← (x + 1, y + 1)
15:             v_k ← x
16:             jos Δ on pariton ja −(d − 1) ≤ k − Δ ≤ (d − 1) niin
17:                 jos x ≥ u_{k−Δ} niin
18:                     palauta (x_0, y_0, x, y)
19:         kaikilla k ← −d, −d + 2, ..., d − 2, d tee
20:             jos k = −d tai (k ≠ d ja u_{k+1} − 1 < u_{k−1}) niin
21:                 x ← u_{k+1} − 1
22:             muuten
23:                 x ← u_{k−1}
24:             y ← x − (k + Δ)
25:             (x_0, y_0) ← (x, y)
26:             niin kauan kun  x > 0 ja y > 0 ja a[x] = b[y] tee
27:                 (x, y) ← (x − 1, y − 1)
28:             u_k ← x
29:             jos Δ on parillinen ja −d ≤ k + Δ ≤ d niin
30:                 jos x ≤ v_{k+Δ} niin
31:                     palauta (x, y, x_0, y_0)
```

4.3.12 Esimerkki. Sanaparille (`aakkosto`, `pakkaus`) keskimadon hakeva algoritmi palauttaa tyhjän madon pisteessä $(4, 6)$. Kuvassa 4.3 on havainnollistettu algoritmin toimintaa näillä sanoilla.



Kuva 4.3: Keskimadon hakeminen

Keskimadon hakevan algoritmin oikeaksi todistamiseen käytetään seuraavaa lemmaa:

4.3.13 Lemma. *On olemassa d-polku origosta pisteeseen (n, m), jos ja vain jos on olemassa $\lceil d/2 \rceil$-polku origosta pisteeseen (x, y) ja $\lfloor d/2 \rfloor$-polku pisteestä (x', y') pisteeseen (n, m), missä pisteille (x, y) ja (x', y') on voimassa seuraavat ehdot:*

(soveltuvuus) $x' + y' \geq \lceil d/2 \rceil$ *ja* $n - x + m - y \geq \lfloor d/2 \rfloor$, *ja*

(päällekkäisyys) $x - y = x' - y'$ *ja* $x \geq x'$.

Lisäksi molemmat $d/2$-polut sisältyvät d-polkuihin origosta pisteeseen (n, m).

Todistus. Oletetaan aluksi, että on olemassa d-polku origosta pisteeseen (n, m). Olkoon d-polun keskimadon alku pisteessä (x, y), jolloin d-polku voidaan jakaa $\lceil d/2 \rceil$-polkuun origosta pisteeseen (x, y) ja $\lfloor d/2 \rfloor$-polkuun pisteestä (x', y') pisteeseen (n, m), missä $(x', y') = (x, y)$. Polulla origosta pisteeseen (x', y') voi olla korkeintaan $x' + y'$ ei-diagonaalista kaarta ja koska on $\lceil d/2 \rceil$-polku

pisteeseen (x', y'), niin $x' + y' \geq \lceil d/2 \rceil$. Polulla pisteestä (x, y) pisteeseen (n, m) voi olla korkeintaan $n - x + m - y$ ei-diagonaalista kaarta ja koska on $\lfloor d/2 \rfloor$-polku pisteestä (x, y) pisteeseen (n, m), niin $n - x + m - y \geq \lfloor d/2 \rfloor$. Lisäksi $x - y = x' - y'$ ja $x \geq x'$, koska $(x, y) = (x', y')$.

Oletetaan sitten, että väitteen ehdot täyttävät $\lceil d/2 \rceil$- ja $\lfloor d/2 \rfloor$-polut ovat olemassa. Ehdosta $x \geq x'$ seuraa, että on k-polku origosta pisteeseen (x', y'), missä $k \leq \lceil d/2 \rceil$. Lemman 4.3.7 perusteella $\delta = \lceil d/2 \rceil - k$ on parillinen, sillä sekä k-polku että $\lceil d/2 \rceil$-polku päättyvät samalle diagonaalille. Lisäksi k-polulla on $(x' + y' - k)/2 \geq \delta/2$ diagonaalikaarta, sillä $x' + y' \geq \lceil d/2 \rceil$. Korvaamalla k-polusta tarkalleen $\delta/2$ diagonaalikaarta kukin aina vaakasuuntaisella ja pystysuuntaisella kaarella saadaan $\lceil d/2 \rceil$-polku origosta pisteeseen (x', y'). Silloin on myös origosta pisteeseen (n, m) kulkeva d-polku, joka koostuu tästä $\lceil d/2 \rceil$-polusta ja annetusta $\lfloor d/2 \rfloor$-polusta. Samantyyppisellä päättelyllä nähdään, että myös $\lceil d/2 \rceil$-polku sisältyy johonkin d-polkuun origosta pisteeseen (n, m). □

4.3.14 Lemma. *Koodilistauksen 4.3 algoritmi palauttaa sanaparin (a, b) optimaalisen d-polun keskimadon.*

Todistus. Oletetaan, että Δ on parillinen. Algoritmi pysähtyy päästyään pienimpään muuttujan d arvoon, jolla pisimmälle ylettyvä käänteinen d-polku kohtaa pisimmälle ylettyvään normaalin d-polun samalla diagonaalilla. Selvästi nämä kohtaavat polut täyttävät Lemman 4.3.13 päällekkäisyysehdon. Näytetään, että kohtaavat polut täyttävät myös lemman soveltuvuusehdon. Oletetaan, että pisimmälle ylettyvä käänteinen polku päättyy pisteeseen (x', y'), missä $x' + y' = k$. On olemassa k-polku ei-diagonaalisia kaaria origosta pisteeseen (x', y'), jotka yhdistettynä käänteiseen d-polkuun muodostavat $(k + d)$-polun origosta pisteeseen (n, m). Lemman 4.3.13 perusteella on siis olemassa kohtaavia h-polkuja, missä $h = (k + d)/2$. ($k + d$ on jaollinen kahdella, koska Δ on parillinen.) On siis varmasti kohtaavia pisimmälle ylettyviä h'-polkuja, missä $h' \leq h$. Jos olisi $k < d$, niin olisi myös $h < d$, mistä seuraisi ristiriita sen kanssa, että pisimmälle ylettyvät d-polut ovat ensimmäisiä, jotka kohtaavat. On siis oltava $x' + y' = k \geq d$. Samantyyppisellä päättelyllä nähdään, että myös pisimmälle ylettyvä normaali d-polku toteuttaa soveltuvuusehdon. Voidaan siis soveltaa Lemmaa 4.3.13 näihin kohtaaviin polkuihin. Tällä perusteella on siis $2d$-polku origosta pisteeseen (n, m). Tämä $2d$-polku on optimaalinen, sillä

jos jollain $j < d$ on $2j$-polku origosta pisteeseen (n, m), niin Lemmasta 4.3.13 seuraa, että on j-polut, jotka kohtaavat. Tämä kuitenkin johtaisi siihen, että olisi pisimmälle ylettyvät kohtaavat j_1- ja j_2-polut, missä $j_1 \leq j$ ja $j_2 \leq j$, mikä olisi ristiriita, sillä d-polut ovat ensimmäisiä, jotka kohtaavat. Molemmat kohtaavat polut ovat siis optimaalisen $2d$-polun osia. Lisäksi palautettava mato on selvästi kyseisen optimaalisen $2d$-polun keskimato, sillä sen molemmin puolin on tarkalleen d matoa. Vastaava päättely voidaan tehdä myös, kun Δ on pariton. □

Keskimadon hakevan algoritmin avulla on nyt helppo muodostaa algoritmi, joka hakee pisimmän yhteisen alijonon hajota ja hallitse -menetelmällä. Koodilistauksessa 4.4 on esitetty tällainen algoritmi. Se mukailee LibXDiff-kirjaston toteutusta[Lib03, Funktio `xdl_recs_cmp`] ja poikkeaa siinä lähteen [Mye86] vastaavasta algoritmista hieman. Näiden välinen ero on, että esitetty algoritmi – kuten LibXDiff-kirjaston versio – eliminoi sanojen pisimmän yhteisen prefiksin ja suffiksin ensimmäisenä ennen KESKIMATO-funktion kutsua, mutta lähteessä [Mye86] näin ei ole tehty.

Koodilistauksessa 4.3 esitetyn keskimadon hakevan algoritmin suurin tilan käyttö on arvojen v_k ja u_k tallentaminen. Koska näitä arvoja on asetettu kullakin kierroksella vain arvoilla $k = -d, -d+1, \ldots, d-1, d$, niin algoritmi on tilakompleksisuudeltaan $O(d)$. Algoritmin aikakompleksisuus on $O((n+m)d)$. Vaikka Koodilistauksen 4.4 PYA-funktio kutsuu itseään rekursiivisesti, niin sen aikakompleksisuus on $O((n+m)d)$ ja tilakompleksisuus $O(n+m)$. Perustelut näille kompleksisuuksille on esitetty lähteestä [Mye86].

Koodilistaus 4.4 Pisin yhteinen alijono keskimadon avulla

```
1:  funktio PYA(a, b)
2:      L ← {}      ▷ muuttuja johon paluuarvo muodostetaan
3:      s ← 1       ▷ aloituskohta
4:      (e_a, e_b) ← (|a|, |b|)      ▷ lopetuskohdat
        ▷ Siirrä aloituskohtaa s eteenpäin kunnes sanat eroavat
          ja tallenna yhteinen prefiksi muuttujaan L:
5:      niin kauan kun s ≤ e_a ja s ≤ e_b ja a[s] = b[s] tee
6:          L ← L ∪ {(s, s)}
7:          s ← s + 1
        ▷ Siirrä lopetuskohtia e_a ja e_b alkuunpäin kunnes sanojen loput
          eroavat ja tallenna yhteinen suffiksi muuttujaan L_T:
8:      L_T ← {}
9:      niin kauan kun e_a ≥ s ja e_b ≥ s ja a[e_a] = b[e_b] tee
10:         L_T ← {(e_a, e_b)} ∪ L_T
11:         (e_a, e_b) ← (e_a − 1, e_b − 1)
        ▷ Jos vielä jäi jäljelle vertailtavaa, niin jaa ongelma keskimadon
          kohdasta kahteen rekursiivisesti ratkaistavaan aliongelmaan:
12:     jos s ≤ e_a ja s ≤ e_b niin
13:         (x', y', x, y) ← KESKIMATO(a[s ... e_a], b[s ... e_b])
            ▷ Keskimatoa ennen olevat osasanat:
14:         kaikilla (i, j) ∈ PYA(a[s ... s − 1 + x'], b[s ... s − 1 + y']) tee
15:             L ← L ∪ {(s − 1 + i, s − 1 + j)}
            ▷ Keskimato:
16:         kaikilla i ← x' + 1, x' + 2, ..., x tee
17:             j ← y' + i − x'
18:             L ← L ∪ {(s − 1 + i, s − 1 + j)}
            ▷ Keskimadon jälkeen olevat osasanat:
19:         kaikilla (i, j) ∈ PYA(a[s + x ... e_a], b[s + y ... e_b]) tee
20:             L ← L ∪ {(s + x − 1 + i, s + y − 1 + j)}
21:     palauta L ∪ L_T
```

4.3.4 Lineaarisen suoritusajan takaava heuristiikka

Kun KESKIMATO-funktio palauttaa madon, joka on muokkausgraafin optimaaliselta polulta, niin funktion PYA palauttamat merkkien sijaintien parit muodostavat pisimmän yhteisen alijonon. Vaikka jakokohtana käytetty mato ei olisi optimaaliselta polulta, niin lopputuloksena on kuitenkin jokin yhteinen alijono. Tämä on tärkeää, sillä Gitin käyttämään LibXDiff-kirjaston keskimadon hakevaan algoritmiin on lisätty heuristiikka, joka voi palauttaa epäoptimaalisen polun madon, jos muokkausetäisyys d on liian suuri [Lib03, Rivi 154]. Heuristiikka takaa, että erojen vertailun voi aina suorittaa lineaarisessa ajassa, sillä se asettaa ylärajan muuttujan d arvolle.

Tämäntyyppinen optimointi on mahdollista, sillä minkä tahansa sanojen a ja b yhteisen alijonon avulla muodostettu muokkauskaavio e toteuttaa ehdon $p(a, e) = b$. Optimaalinen muokkauskaaviosta tulee kuitenkin ainoastaan silloin, kun se muodostetaan pisimmästä yhteisestä alijonosta.

Hieman suuremman muokkauskaavion käyttäminen saattaa tehdä erojen vertailusta vähän vaikeampaa Git-käyttäjälle, mutta tiedostojen erojen vertailu on yleensä kuitenkin paljon helpompaa epäoptimaalisellakin muokkauskaaviolla kuin vertailemalla tiedostojen sisältöjä ilman minkäänlaista muokkauskaaviota. Lisäksi koska epäoptimaalinenkin muokkauskaavio kuitenkin muokkaa sanasta a sanan b, niin sellaisen käyttäminen korjaustiedostossa ei aiheuta, että korjaustiedoston tuottama lopputulos poikkeaisi halutusta. Epäoptimaalisen muokkauskaavion käyttäminen kehityshaarojen yhdistämisessä aiheuttaa korkeintaan turhia konflikteja, joita ei voida selvittää automaattisesti, mutta ei aiheuta automaattisesti yhdistettyjen muutosten virheellisyyttä.

5 Pakettitiedosto

Kun irrallisten objektien lukumäärä kasvaa tarpeeksi suureksi, niin Git kerää irralliset objektit pakettitiedostoon. Pakettitiedostossa on objekteja pakattuna kahdella eri tavalla. Suurin osa objekteista on pakattu ensin deltapakkauksella (delta compression) ja tämän jälkeen vielä deflate-pakkauksella. Loput objekteista on pakattu pelkästään deflate-pakkauksella. [Cha11b, Kohta "Packfiles"]

Deltapakkauksessa on aina pakattavan kohdeobjektin (target object) lisäksi käytössä myös lähdeobjekti (source object). Sama lähdeobjekti pitää olla saatavilla myöhemmin deltapakatun tiedon dekoodauksessa. Jos kohdeobjektin ja lähdeobjektin tavujonoissa on samoja osajonoja, niin kohdeobjektin pakkauksessa voidaan tallentaa vain osajonon pituus ja viittaus lähdeobjektin kohtaan, josta se alkaa. Hyvin lyhyet tai kohdeobjektille uniikit alijonot tallennetaan sellaisenaan. [Mac00]

Deltapakkaus voi pienentää objektien tallennukseen tarvittavaa tilaa huomattavasti, sillä tyypillisesti versiohallintaan tallennetaan useita eri versioita samoista tiedostoista, missä kahden eri version erot ovat pieniä suhteessa tiedoston kokoon. Esimerkiksi lähteessä [Wel07] on tehty testi, jossa Gnumeric-taulukkolaskentaohjelman 172 eri version julkaisupakettien sisältö tallennettiin Git-tietovarastoon ja lopputuloksena Git-tietovaraston koko oli kooltaan vain alle kymmenesosa erikseen pakattujen julkaisupakettien yhteiskoosta.

5.1 Pakettitiedoston rakenne

Objektin deltapakattua muotoa kutsutaan tässä *deltaobjektiksi*. Se ei kuitenkaan ole varsinainen Gitin objektityyppi kuten blob, tag, tree ja commit. Deltaobjektin rakenne on kuvattu kohdassa 5.3.

Gitin teknisen dokumentaation [Ham08] perusteella pakettitiedoston rakenne on seuraava:

- Alussa on 12 tavun otsikko, joka koostuu neljän tavun tunnisteesta `"PACK"` sekä neljällä tavulla koodatuista versionumerosta ja pakettitiedoston objektien lukumäärästä. Gitin versiossa 1.7.5 hyväksytyt pakettitiedoston versionumerot ovat 2 ja 3, mutta tuotetut pakettitiedostot ovat aina versiota 2.

- Otsikkotietojen jälkeen on koodattu jokainen objekti erikseen.
  - *Deltapakkaamattomasta* objektista on koodattu ensin objektin tyyppi ja (pakkaamaton) koko vaihtelevalla tavupituudella. Näiden jälkeen on objektin sisältö deflate-pakattuna.
  - *Deltapakatusta* objektista on koodattu ensin deltaviittauksen tyyppi ja deltaobjektin (pakkaamaton) koko vaihtelevalla tavupituudella. Deltaviittauksen tyyppejä on kaksi: ofs-delta ja ref-delta. Jos deltaviittauksen tyyppi on ofs-delta, niin seuraavana on koodattu lähdeobjektin etäisyys vaihtelevalla tavupituudella, ja jos deltatyyppi on ref-delta, niin lähdeobjektin etäisyyden sijaan on tähän kohtaan koodattu 20 tavuinen lähdeobjektin tunniste. Lähdeobjektin osoitetietojen jälkeen seuraa deltaobjektin sisältö deflate-pakattuna. Lähdeobjektin etäisyys kertoo montako tavua ennen kyseistä objektia lähdeobjekti on pakettitiedostossa, joten ofs-delta tyyppisellä viittauksella voi viitata vain tiedostossa aiemmin oleviin objekteihin.
- Lopussa on vielä 20 tavun SHA1-tarkistussumma kaikesta edeltävästä.

Pakettitiedoston lisäksi Git tallentaa myös paketti-indeksitiedoston, jonka avulla voidaan nopeasti löytää tietyn objektin sijainti pakettitiedostossa objektin tunnisteen perusteella. Yksi tärkeä huomio pakettitiedoston rakenteessa on, että jokainen objekti on pakattu erikseen vaikka pakkaamalla kaikki objektit yhdessä olisi mahdollista saavuttaa parempi pakkaussuhde. Erikseen pakkaamisen syy on, että se mahdollistaa yksittäisten objektien purkamisen purkamatta koko pakettitiedostoa, jossa voi olla tuhansia objekteja. Deltapakatun objektin purkaminen vaatii myös lähdeobjektin purkamisen ja jos lähdeobjektikin sattuu olemaan deltapakattu, niin myös sen lähdeobjektin jne., mutta tällaisten *deltaketjujen* pituus on rajoitettu pakkausvaiheessa, joten yksittäisen objektin hakeminen voidaan tehdä aina käytännössä vakioajassa.

5.2 Pakettitiedoston muodostaminen

Pakettitiedoston koon kannalta on suuri merkitys sillä, että mitkä objektit deltapakataan ja mitä objekteja käytetään kunkin objektin deltapakkauksessa

lähdeobjektina. Periaatteessa voitaisiin vain käydä kaikki objektiparit läpi ja verrata sitten, että millä objektipareilla tulee pienimmät deltaobjektit. Tämä on kuitenkin aivan liian hidasta, koska objekteja saattaa olla paljon. (Esimerkiksi Gitin lähdekoodin omassa Git-tietovarastossa oli n. 150 000 objektia tätä kirjoittaessa.)

Toisaalta pelkästään pakettitiedoston pieni koko ei ole ainoa tavoiteltava asia. Gitin käytettävyyden kannalta tärkeämpi ominaisuus on, että tarpeelliset objektit ovat saatavissa nopeasti pakettitiedostosta.

Objektien nopea saatavuus saavutetaan tallentamalla objektit pakettitiedostoon *tuoreusjärjestyksessä* (recency order). Tuoreusjärjestys vastaa objektien muodostaman dag-graafin solmujen topologista järjestystä. Objektien oikealla järjestyksellä saavutetaan mahdollisimman pieni määrä hitaita levyhakuja, koska silloin yleensä yhdellä kertaa tarvittavat objektit ovat pakettitiedostossa lähellä toisiaan. [Loe06; Tor08]

Deltapakkausta varten pitää muodostaa lähde- ja kohdeobjektien pareja. Deltaobjektin rakenteesta ja niiden muodostamisesta on kerrottu tarkemmin kohdissa 5.3 ja 5.4. Pakettitiedoston muodostamisen kannalta riittää tietää, että deltaobjektista saadaan mahdollisimman pieni silloin, kun lähde- ja kohdeobjektit ovat sisällöltään mahdollisimman samanlaiset. Suuremman objektin on yleensä järkevämpää olla lähdeobjektina, sillä silloin yleensä saadaan tallennettua enemmän kohdeobjektin tavuja viittauksina lähdeobjektiin.

Kun Git hakee lähde- ja kohdeobjektien pareja deltapakkaukselle, se järjestää objektit ns. deltajärjestykseen. Deltajärjestys tehdään objektin tyypin, polkunimen (path name) ja koon perusteella: ensimmäisenä tyypin mukaan, koska erityyppisiä objekteja ei deltapakata toisiaan vasten. Polkunimestä tehdään ns. nimi-hash (namehash), jonka arvot pyritään muodostamaan siten, että nimi-hash-järjestyksessä saman tiedoston eri versiot ovat lähellä toisiaan ja lisäksi myös samantyyppiset tiedostot ovat lähellä toisiaan. Objektin koko on viimeinen järjestysperuste. Suuremmat objektit järjestetään ennen pienempiä, mikä johtaa siihen, että suurempia objekteja käytetään deltapakkauksessa useammin lähdeobjekteina kuin pienempiä. Toisaalta, koska tiedostot yleensä kasvavat ajan myötä, niin samalla uudempien objektien deltaketjuista tulee yleensä lyhyempiä kuin vanhempien. [Loe06]

Kunkin objektin deltapakkauksen lähdeobjekti valitaan liikuttamalla kiinteän kokoista ikkunaa tämän deltajärjestyksessä olevan objektilistan yli ja laskemalla deltoja ikkunaan viimeisenä tulleelle objektille käyttäen lähdeobjektina ikkunan muita objekteja järjestyksessä. Ikkunaan viimeisenä tulleen objektin lähdeobjektiksi valitaan sitten se, jolla delta on pienin. Deltaketjujen pituudet ovat kuitenkin rajattu syvyysparametrilla, jotta pitkät deltaketjut eivät aiheuttaisi objektien purkamisen liiallista hidastumista. Myös muutamalla muulla heuristiikalla voidaan päättää hylätä delta, jolloin voidaan päätyä käyttämään deltapakkaamatonta muotoa. Esimerkiksi deltan suuri koko suhteessa deltapakkaamattoman kohdeobjektin kokoon on yksi tällainen heuristiikka. [Pit+10]

Lopuksi, kun jokaiselle objektille on joko valittu lähdeobjekti tai päätetty jättää objekti deltapakkaamattomaan muotoon, Git tallentaa objektit pakettitiedostoon tuoreusjärjestyksessä aloittaen tuoreimmasta objektista. Järjestykseen tehdään kuitenkin aina poikkeus silloin, kun jotain objektia ollaan tallentamassa deltapakatussa muodossa, mutta lähdeobjektia ei ole vielä tallennettu. Siinä tapauksessa lähdeobjekti tallennetaan aina juuri ennen kyseistä objektia. Tämä tehdään rekursiivisesti niin, että jokaisen deltapakatun objektin lähdeobjekti tallennetaan aina ennen kohdeobjektia. Deltajärjestys on kuitenkin suunniteltu niin, että yleensä se vastaa hyvin tuoreusjärjestystä, jolloin näitä poikkeuksia ei tule montaa. [Loe06]

5.3 Deltaobjektin rakenne

5.3.1 Deltan matemaattinen määritelmä

Matemaattisesti deltapakkaus voidaan mallintaa esimerkiksi seuraavien määritelmien avulla.

5.3.1 Merkintä. Otetaan käyttöön merkinnät I, C_k ja D_k joukoille, jotka määräytyvät seuraavasti: Joukko $I = \{ (0, w) \mid w \in \Sigma^+ \}$ sekä kaikilla $k \in \mathbb{N}$

$$C_k = \left\{ (p, l) \in \mathbb{N}^2 \;\middle|\; 1 \leq p \leq k \text{ ja } 1 \leq l \leq k - p + 1 \right\}$$

ja $D_k = I \cup C_k$.

5.3.2 Määritelmä. Sanaan $a \in \Sigma^*$ perustuva *delta* on sana $d_1 d_2 \cdots d_n$, missä $n \geq 0$ ja $d_i \in D_{|a|}$ kaikilla $i = 1, \ldots, n$.

5.3.3 Määritelmä. Deltan $d = d_1 d_2 \cdots d_n$ soveltaminen sanaan $a \in \Sigma^*$ tuottaa sanan $x_1 x_2 \cdots x_n \in \Sigma^*$, missä

$$x_i = \begin{cases} w, & \text{jos } d_i = (0, w) \in I \\ a[p \ldots p + l - 1], & \text{jos } d_i = (p, l) \in C_{|a|} \end{cases}$$

kaikilla $i = 1, \ldots, n$. Näin muodostettua sanaa $x_1 x_2 \cdots x_n$ merkitään $P(a, d)$.

Deltan $d = d_1 d_2 \cdots d_n$ alkioita d_i sanotaan *deltakomennoiksi*. Deltakomentoa $d_i \in I$ sanotaan *lisäyskomennoksi* ja deltakomentoa $d_i \in C_k$ sanotaan *kopiointikomennoksi*. Kun deltaa koodataan tavujonoksi, niin yleensä lisäyskomennot koodautuvat paljon pidemmiksi tavujonoiksi kuin kopiointikomennot, koska lisäyskomennossa on mukana mielivaltaisen pitkä sana, mutta kopiointikomennot ovat esitettävissä kahdella kokonaisluvulla. Deltapakkaus perustuu siihen, että jos sana b voidaan jakaa osasanoihin, joista iso osa on myös sanan a osasanoja, niin voidaan muodostaa delta d, jolle $P(a, d) = b$ ja jonka deltakomennoista suurin osa on kopiointikomentoja.

5.3.4 Esimerkki. Olkoon $a =$ `JokinTodellaPitkaSana` ja

$$d = (1, 5)(13, 3)(0, \texttt{empi})(18, 4)(6, 7)(3, 3),$$

jolloin $P(a, d) =$ `JokinPitempiSanaTodellakin`.

Deltapakkaus muistuttaa aika paljon kohdassa 3.3 käsiteltyä LZ77-koodausta. Deltakomennot ovat hyvin samantyyppisiä kuin LZ77-koodauksen tuottamat (p, l, x)-kolmikot. Suurin ero deltapakkauksen ja LZ77-koodauksen välillä on se, että LZ77-koodauksessa käytetään liukuvaa ikkunaa, johon viitataan, kun taas deltapakkauksessa viitataan kiinteän sanan mihin tahansa kohtaan.

Toisaalta delta ja kohdassa 4.1 määritelty muokkauskaavio ovat myös hyvin samantyyppisiä käsitteitä. Molemmat kuvailevat tavan miten sanasta a saadaan sana b. Delta on kuitenkin siinä mielessä yleisempi käsite, että muokkauskaavion perusteella on aina mahdollista muodostaa delta, mutta ei päin vastoin. Tämä

johtuu siitä, että deltan kopiointikomennot eivät välttämättä viittaa lähdesanan kohtiin järjestyksessä.

5.3.5 Esimerkki. Esimerkin 4.1.4 muokkauskaavion

$$e = (\{1, 5, 7, 8\}, \{1 \to \mathtt{p}, 5 \to \mathtt{au}\})$$

ja lähtösanan pituuden $|a| = 8$ perusteella muodostettu delta on

$$d = (0, \mathtt{p})(2, 3)(0, \mathtt{au})(6, 1).$$

Näille on siis $p(a, e) = P(a, d)$. Huomaa, että Esimerkin 5.3.4 deltaa ei voi esittää muokkauskaaviona.

5.3.2 Gitin käyttämä deltan esitysmuoto

Kuvaillaan seuraavaksi tapa, jolla Git koodaa deltaobjektin sisällön. Kuvaus perustuu Gitin lähdekoodiin [Pit07, Funktio ”create_delta”].

Oletetaan, että aakkostona on tavut eli $\Sigma = \mathcal{B}$ ja olkoon $d = d_1 d_2 \cdots d_n$ tavujonoon $a \in \mathcal{B}^*$ perustuva deltakomentojen jono, jota ollaan koodaamassa. Oletetaan, että jonon d jokaisen kopiointikomennon (p, l) arvo $p < 2^{32}$. Muuten deltakomentojen jono ei ole koodattavissa Gitin menetelmällä. Jaetaan jonon d jokainen lisäyskomento, jonka lisäämä sana on yli 127 tavua pitkä, useaksi lisäyskomennoksi, joista kaikki lisäävät 127 tavua tai vähemmän. Lisäksi jaetaan jokainen kopiointikomento (p, l), jonka pituus l on yli 65536 tavua useaksi kopiointikomennoksi, joista jokainen kopioi 65536 tavua tai vähemmän. Merkitään näin muodostettua deltakomentojonoa $d' = d'_1 d'_2 \cdots d'_m$. On siis voimassa $P(a, d) = P(a, d')$.

Olkoon $h \in \mathcal{B}^*$ otsikkotavujono, johon on koodattu lähdeobjektin koko $|a|$ sekä kohdeobjektin koko $|P(a, d)|$ vaihtelevalla tavupituudella. Tarkkaa otsikkotavujonon koodausta ei kuvailla tässä. Jokainen deltakomento d'_i koodataan erikseen omaksi tavujonokseen $c_i \in \mathcal{B}^*$. Deltaobjektin sisältö on otsikkotavujonon ja koodattujen deltakomentojen katenaatio $hc_1 \cdots c_n$. Yksittäisen lisäyskomennon $d'_i = (0, w)$ koodaus on $\mathbf{b}_L w$, missä $L = |w| \leq 127$. Kopiointikomento

$d'_i = (p, l)$ koodataan tavujonoksi seuraavalla algoritmilla:

funktio KOODAAKOPIOINTIKOMENTO(p, l)

▷ Olkoon $p = 2^{24}q_3 + 2^{16}q_2 + 2^8q_1 + q_0$ ja $l = 2^{16}q_6 + 2^8q_5 + q_4$, missä $0 \leq q_i < 256$.

$(k, q) \leftarrow (128, \varepsilon_{\mathcal{B}})$

kaikilla $i \leftarrow 0, 1, 2, 3, 4, 5$ **tee**

jos $q_i \neq 0$ **niin**

$q \leftarrow q \cdot$ KOODAATAVUKSI(q_i)

$k \leftarrow k + 2^i$

palauta $\mathbf{b}_k q$

Kopiointikomennon koodattu pituus on siis yhdestä seitsemään tavua. Pituus $l = 65536$ on koodauksessa erikoistapaus, sillä se koodataan samoin kuin $l = 0$, mutta koska $l = 0$ ei ole käytössä, niin koodaus on yksikäsitteinen.

5.4 Deltaobjektin muodostaminen

Gitin deltapakkauksen toteutuksen perusidea on peräisin LibXDiff-kirjastosta [Pit07], jonka deltapakkauksen toimintaperiaate perustuu puolestaan Joshua P. MacDonaldin raporttiin ”File System Support for Delta Compression” [Libx12]. Raportissa on kuvailtu algoritmi nimeltään Xdelta [Mac00], joka on pääpiirteittäin sama kuin Gitin deltapakkauksen käyttämä algoritmi. Se etsii yhteneviä osasanoja käyttäen avuksi epäkryptografista hash-funktiota. Xdeltassa on käytetty hash-funktiona Adler-32-tarkistussummaa, mutta Gitissä on otettu käyttöön Rabinin sormenjälki [Pit07; Mac00]. Rabinin sormenjäljestä on kerrottu tarkemmin kohdassa 5.5.

Xdelta-algoritmi on kaksivaiheinen: ensin tehdään hash-funktion avulla ns. *deltaindeksi* lähdesanasta ja sen jälkeen itse delta muodostetaan etsimällä lähde- ja kohdesanan yhteisiä osasanoja käyttäen avuksi deltaindeksiä [Mac00]. Kuvaillaan molemmat vaiheet seuraavaksi tarkemmin.

Hash-arvot lasketaan aina kiinteän kokoisesta ikkunasta, jonka kokoa merkitään tässä symbolilla w. Deltaindeksiin lasketaan hash-arvot lähdesanan a kaikista w-pituisista osasanoista $a[i \ldots i + w - 1]$, joiden aloituskohta i on jaollinen luvulla w. Deltan muodostuksessa puolestaan hash-arvot lasketaan kohdesanan b kaikista w-pituisista osasanoista. Gitissä $w = 16$ [Pit07]. Deltaindeksi on

jaettu lokeroihin, joiden lukumäärä määräytyy lähdesanan pituuden ja ikkunan koon perusteella. Gitin toteutuksessa kuhunkin lokeroon tallennetaan hash-arvojen ja lähdesanan sijaintien lista, jonka koko lopuksi rajoitetaan 64 alkioon poistamalla listasta tasaisin välein alkioita [Pit07]. Xdelta-algoritmissa kuhunkin lokeroon tallennetaan vain yksi alkio [Mac00]. Deltaindeksin muodostus on esitetty pseudokoodina Koodilistauksessa 5.1.

Koodilistaus 5.1 Deltaindeksin muodostaminen

```
1:  funktio TEEDELTAINDEKSI(a)    ▷ a=lähdesana
2:      L ← LOKEROIDENLUKUMÄÄRÄ(|a|, w)
3:      kaikilla k ← 1, ..., L tee
4:          X_k ← {}
5:      i ← 1
6:      niin kauan kun i + w − 1 ≤ |a| tee
7:          h ← HASH(a[i ... i + w − 1])
8:          k ← LOKERO(h, L)    ▷ hash-arvon h lokero, k ∈ {1, ..., L}
9:          X_k ← X_k ∪ {(h, i)}
10:         i ← i + w
11:     kaikilla k ← 1, ..., L tee
12:         X_k ← KARSILISTA(X_k)    ▷ Rajoita listan alkioiden lukumäärää
13:     palauta (X_1, ..., X_L)
```

Kun deltaindeksi on tehty, niin itse deltan laskeminen onnistuu liu'uttamalla kiinteän kokoista ikkunaa merkki kerrallaan kohdesanan yli ja etsimällä ikkunan hash-arvoa deltaindeksistä. Aina kun ikkunan hash-arvo löytyy deltaindeksistä, niin saadaan samalla myös lähdesanan sijainti, josta kannattaa etsiä yhteistä osasanaa. Hash-arvojen yhtäsuuruus kertoo, että kyseisissä sijainneissa on mahdollista olla ikkunan koon pituinen yhteinen osasana, mutta siitä kohdasta voi löytyä myös pitempi yhteinen osasana, joten sellaista kannattaa etsiä. Osasanojen yhtäsuuruus pitää tarkistaa joka tapauksessa, sillä hash-arvot voiva myös törmätä vaikka ikkunoiden sisällöt poikkeaisivat. [Mac00]

Deltan muodostava algoritmi on esitetty Koodilistauksessa 5.2.

Koodilistaus 5.2 Deltan laskeminen deltaindeksin avulla

```
1: funktio TEEDELTA(a, b, (X_1, ..., X_L))   ▷ a=lähdesana, b=kohdesana,
                                                (X_1, ..., X_L)=deltaindeksi
2:     d ← ε    ▷ muuttuja johon delta lasketaan
3:     u ← ε    ▷ seuraavan lisäyskomennon sana
4:     i ← 1
5:     niin kauan kun i ≤ |b| tee
6:         (p, l) ← ETSIVASTAAVUUS(a, b, i, (X_1, ..., X_L))
7:         jos l < w niin
8:             u ← u · b[i]    ▷ Lisää merkki seuraavaan lisäyskomentoon.
9:             i ← i + 1
10:        muuten
11:            jos |u| > 0 niin
12:                d ← d · (0, u)    ▷ lisäyskomento
13:                u ← ε
14:            d ← d · (p, l)    ▷ kopiointikomento
15:            i ← i + l
16:    jos |u| > 0 niin
17:        d ← d · (0, u)    ▷ lisäyskomento
18:    palauta d
19: funktio ETSIVASTAAVUUS(a, b, i, (X_1, ..., X_L))
20:    h ← HASH(b[i ... i + w − 1])
21:    k ← LOKERO(h, L)
22:    (p, l) ← (0, 0)
23:    kaikilla (h', i') ∈ X_k tee
24:        jos h' = h niin
25:            l' ← VASTAAVUUDENPITUUS(a, i', b, i)
26:            jos l' > l niin
27:                (p, l) ← (i', l')
28:    palauta (p, l)
```

5.5 Rabinin sormenjälki

Deltapakkauksessa käytetään yhtenevien osasanojen etsimisen nopeuttamiseen hash-funktiota. Git käyttää deltapakkauksen hash-funktiona Rabinin sormenjälkeä, joka vaihdettiin Adler-32-funktion tilalle vuonna 2006. [Pit06]

Bittijonon Rabinin sormenjälki lasketaan tulkitsemalla bittijonon bitit polynomin $a(x) \in \mathbb{Z}_2[x]$ kertoimina ja laskemalla jakojäännös $q(x) \equiv a(x) \pmod{p(x)}$ ennalta valitun jaottoman polynomin $p(x) \in \mathbb{Z}_2[x]$ suhteen. Mene-

telmä vastaa pitkälti perinteistä menetelmää, jossa sormenjälki bittijonosta tehdään tulkitsemalla se suurena kokonaislukuna a, ja laskemalla jakojäännös $q \equiv a \pmod{p}$ ennalta valitun suuren alkuluvun p kanssa. Sormenjäljen laskeminen suurilla luvuilla vaatii vähintään summan ja erotuksen toteuttamisen $\lceil \log_2 p \rceil$-bittisillä kokonaisluvuilla. Rabinin menetelmän etuna on, että modulo $p(x)$ -aritmetiikka voidaan toteuttaa operaatioilla, jotka ovat nopeita suorittaa nykyisillä prosessoreilla. Tarvittavat operaatiot ovat bittien siirtäminen (bit shift) ja biteittäinen poissulkeva-tai (bitwise exclusive or, XOR). Lisäksi menetelmä mahdollistaa hash-arvon päivittämisen, kun lähdesanan alusta poistetaan merkki ja loppuun lisätään merkki, mikä tekee liukuvan ikkunan sisällön hash-arvojen laskemisesta nopeaa. [Rab81]

Esitetään seuraavaksi miten Rabinin sormenjälki voidaan laskea käytännössä. Ensin, kohdassa 5.5.2, johdetaan tapa laskea sormenjälki luettaessa lähdettä bitti kerrallaan. Sen jälkeen, kohdassa 5.5.3 näytetään vielä miten laskentaa voidaan nopeuttaa, kun lähdettä luetaan tavu kerrallaan ja koko lähteen sormenjäljen sijaan halutaankin laskea lähteen kaikkien w-pituisten osasanojen sormenjäljet. Esitettävät algoritmit toimivat millä tahansa jakajapolynomilla $p(x) \neq 0$, mutta jotkin jakajapolynomit ovat sormenjälkien laadun kannalta parempia kuin toiset. Jakajapolynomin valintaa pohditaan vielä kohdassa 5.5.4.

5.5.1 Merkintöjä ja määritelmiä

Polynomimerkintöjen helpottamiseksi määritellään polynomirenkaan $\mathbb{Z}_2[x]$ kanssa isomorfinen rengas $\mathcal{R}$ seuraavasti. Renkaan $\mathcal{R}$ alkioita merkitään vektoreilla $[g_1, g_2, \ldots, g_m]$, missä $m \geq 1$ ja $g_1, \ldots, g_m \in \mathbb{Z}_2$. Alkio $[g_1, g_2, \ldots, g_m]$ vastaa isomorfiassa polynomia

$$g(x) = g_1 x^{m-1} + g_2 x^{m-2} + \ldots + g_m \in \mathbb{Z}_2[x].$$

Lisäksi määritellään $x = [1, 0] \in \mathcal{R}$, jolloin polynomilla x^n kertominen vastaa n nollan lisäämistä vektoriesityksen loppuun. Jos vielä samaistetaan alkio $z \in \mathbb{Z}_2$ ja vektori $[z] \in \mathcal{R}$, niin näillä merkinnöillä on myös

$$[g_1, \ldots, g_m] = g_1 x^{m-1} + g_2 x^{m-2} + \ldots + g_m \in \mathcal{R}.$$

Alkion $g \in \mathcal{R}$ aste $\deg g$ on yhtä suuri kuin alkion g kanssa isomorfisen polynomin $g(x) \in \mathbb{Z}_2[x]$ aste. Siis kun $g_1 \neq 0$, niin $\deg[g_1, \ldots, g_m] = m - 1$ ja $\deg[0] = -\infty$.

Kiinnitetään alkio $p = [p_0, p_1, \ldots, p_k] \in \mathcal{R}$, missä $k \geq 1$ ja $p_0 = 1$, joten alkio p vastaa siis isomorfiassa astetta k olevaa polynomia. Käytetään alkion $g \in \mathcal{R}$ jakojäännöksestä alkion p suhteen merkintää $\overline{g}$. Toisin sanoen

$$\overline{g} \equiv g \pmod{p} \quad \text{ja} \quad \deg \overline{g} < \deg p = k.$$

5.5.2 Sormenjäljen laskeminen bitti kerrallaan

Johdetaan algoritmi bittijonon $a_0 a_1 \cdots a_n \in \beta^*$ sormenjäljen laskemiseen. Esitys perustuu Michael O. Rabinin raporttiin "Fingerprinting By Random Polynomials" [Rab81].

Merkitään $A_i = [a_0, \ldots, a_i]$, jolloin bittijonon sormenjälki on siis vektorin $\overline{A_n}$ alkioiden muodostama k-pituinen bittijono. Määritelmästä seuraa, että kaikilla $i = 1, \ldots, n$ on $A_i = A_{i-1}x + a_i$, joten

$$\overline{A_i} = \overline{\overline{A_{i-1}}x} + a_i.$$

Jos tässä $\overline{A_{i-1}} = [r_1, \ldots, r_k]$, niin saadaan

$$\overline{A_i} = r_1\overline{x^k} + [r_2, \ldots, r_k, a_i] \tag{5.1}$$

ja koska $\overline{p} = \overline{x^k + p_1 x^{k-1} + \ldots + p_k} = 0$, niin

$$\overline{x^k} = -(p_1 x^{k-1} + \ldots + p_k) = -[p_1, \ldots, p_k] = [p_1, \ldots, p_k].$$

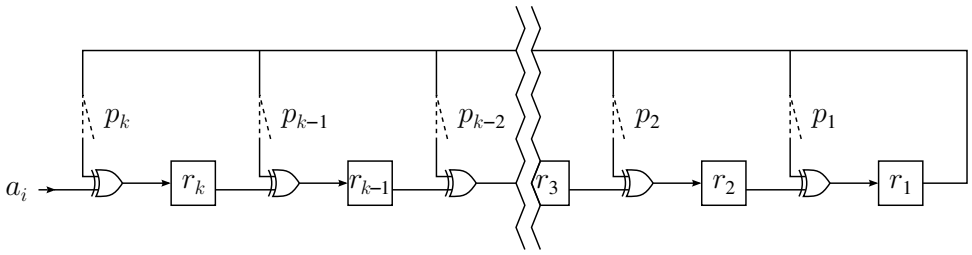
Viimeinen yhtäsuuruus seuraa siitä, että kunnassa $\mathbb{Z}_2$ on $-1 = 1$. Täten yhtälö (5.1) saadaan muotoon

$$\overline{A_i} = [r_2, \ldots, r_k, a_i] + r_1[p_1, \ldots, p_k]. \tag{5.2}$$

Yhtälön (5.2) perusteella sormenjälki $\overline{A_n}$ saadaan siis muodostettua iteratiivisesti seuraavasti: Asetetaan ensin vakiopituinen vektori $r = [r_1, \ldots, r_k] = 0$. Luetaan sanaa $a_0 \cdots a_n$ bitti kerrallaan vasemmalta oikealle ja aina bitin a_i

kohdalla muodostetaan arvo $\overline{A_i}$ siirtämällä vektorin r bittejä yhden bitin vasemmalle ja asetetaan sen oikeanpuoleisimmaksi bitiksi juuri luettu bitti a_i. Jos bittien siirtämisessä yli vuotanut bitti oli 1, niin lasketaan vielä vektorin r ja vektorin $[p_1, \ldots, p_k]$ välinen biteittäinen poissulkeva-tai ja tallennetaan tulos vektoriin r. Kun kaikki sanan $a_0 \cdots a_n$ kaikki bitit on käsitelty, niin vektorissa r on haluttu sormenjälki.

Esitetty sormenjäljen muodostamistapa kuvailee järjestelmän lineaarisena takaisinkytkettynä siirtorekisterinä (linear feedback shift register, LFSR) [Kra94]. Tätä on havainnollistettu Kuvassa 5.1.



Kuva 5.1: Lineaarinen takaisinkytketty siirtorekisteri

Kuvan laatikot $r_1, r_2, \ldots, r_k$ ovat kiikkuja. Jokainen kiikku tallentaa rekisterin yhden bitin. Kiikkujen vasemmalla puolella olevat portit ovat XOR-portteja, joiden ulostulo on siis poissulkeva-tai sisäänmenevistä biteistä. Katkoviivalla piirretyt kytkimet ovat kytkettyjä tai avonaisia riippuen bittien $p_1, p_2, \ldots, p_k$ arvoista. Kohdassa p_i on kytkentä silloin ja vain silloin, kun $p_i = 1$.

Järjestelmä toimii vaiheissa. Aluksi kaikki kiikut ovat tilassa 0. Jokaisen kiikun oikealla puolella on ulostulo, joka on arvoltaan sama kuin kiikun tila. Kaikki kiikut lukevat yhtä aikaa vasemmalta sisääntulevan bitin, josta tulee kiikun seuraava tila. Rekisteri lukee siis vasemmalta tulevia bittejä a_i yksi kerrallaan ja laskee polynomijakojäännöksen kiikkuihinsa.

5.5.3 Sormenjäljen laskeminen ikkunasta tavu kerrallaan

Kun sormenjälki halutaan laskea tavujonosta, niin tavujen käsittely bitti kerrallaan on tarpeettoman hidasta. Johdetaan vielä tapa nopeuttaa sormenjäljen laskemista, kun bittejä on kahdeksalla jaollinen määrä. Lisäksi samalla näytetään miten liukuvan ikkunan sisällöstä laskettava sormenjälki saadaan helposti

päivitettyä.

Nopeutus perustuu tiettyjen jäännösarvojen taulukointiin. Samaa menetelmää käytetään myös Gitin deltapakkauksessa, mutta Gitin lähdekoodissa ei ole esitetty perusteluja menetelmälle tai edes tapaa muodostaa lähdekoodissa olevia esilaskettuja taulukoita. [Pit07]

Oletetaan, että bittien määrä $n+1$ on jaollinen kahdeksalla, ikkunan koko $w \geq 1$ ja polynomin p aste $k \geq 8$. Olkoon $N = (n+1)/8 - 1$. Jaetaan bittijono $a_0 \cdots a_n$ tavuihin B_i asettamalla

$$B_i = [a_{8i}, a_{8i+1}, \ldots, a_{8i+7}]$$

kaikilla $i = 0, 1, \ldots, N$ ja sovitaan lisäksi, että

$$B_{-1} = B_{-2} = \ldots = B_{-w+1} = [0, \ldots, 0]\,.$$

Ikkunoiden sisällöt W_i, $i = 0, 1, \ldots, N$, joista sormenjäljet lasketaan, ovat

$$W_i = \sum_{j=0}^{w-1} B_{i-j} x^{8j},$$

jotka voidaan esittää myös rekursiomuodossa, kun $i = 1, 2, \ldots, N$:

$$W_i = (W_{i-1} - B_{i-w} x^{8(w-1)}) x^8 + B_i.$$

Merkitään rekursiokaavassa suluissa esiintyvää termiä symbolilla W_i' eli

$$W_i' = W_{i-1} - B_{i-w} x^{8(w-1)}.$$

Muodostetaan funktio U, joka määritellään kaikille mahdollisille tavuille $B = [b_0, \ldots, b_7]$ asettamalla

$$U(B) = \overline{Bx^{8(w-1)}} = \sum_{i=0}^{7} b_i \overline{x^{8(w-1)+7-i}}. \tag{5.3}$$

Nyt saadaan termin W'_i sormenjälki esitettyä muodossa

$$\overline{W'_i} = \overline{W_{i-1}} + U(B_{i-w}). \tag{5.4}$$

Määritellään vielä funktio T tavuille $B = [b_0, \ldots, b_7]$ asettamalla

$$T(B) = \overline{Bx^k} = \sum_{i=0}^{7} b_i \overline{x^{k+7-i}} \tag{5.5}$$

ja merkitään $\overline{W'_i} = [r_{i,1}, \ldots, r_{i,k}]$. Näillä merkinnöillä saadaan

$$\overline{W_i} = \overline{\overline{W'_i} x^8} + B_i = T([r_{i,1}, \ldots, r_{i,8}]) + [r_{i,9}, \ldots, r_{i,k}]\, x^8 + B_i. \tag{5.6}$$

Esitetään vielä johdettuihin kaavoihin perustuva ikkunoiden sisältöjen sormenjäljet laskeva algoritmi pseudokoodina:

Koodilistaus 5.3 Rabinin sormenjälkien laskeminen liukuvasta ikkunasta

Vaatimus: Funktiot U ja T ovat määritelty kuten kaavoissa (5.3) ja (5.5).

```
1:  funktio RABINSORMENJÄLJET(w, B_0, B_1, ..., B_N)
2:      kaikilla j ← −1, −2, ..., −w tee
3:          B_j ← [0, ..., 0]
4:      R ← {}                                   ▷ sormenjälkien lista
5:      [r_1, ..., r_k] = [0, ..., 0]
6:      kaikilla i ← 0, 1, ..., N tee
7:          [r_1, ..., r_k] ← [r_1, ..., r_k] + U(B_{i−w})      ▷ Ks. kaava (5.4)
8:          [r_1, ..., r_k] ← T([r_1, ..., r_8]) + [r_9, ..., r_k] x^8 + B_i      ▷ Ks. kaava (5.6)
9:          R ← R ∪ {[r_1, ..., r_k]}
10:     palauta R
```

Koodilistauksen 5.3 algoritmin toteutus vaatii vain kolme operaatiotyyppiä: bittien siirtämisen, poissulkevan-tain ja taulukkohaun. Funktioiden U ja T arvot on hyvä esilaskea taulukoihin ja pitää muistissa. Niiden vaatima tila on $256k$ bittiä kummallekin, tai käytännössä $256b$ bittiä, kun käytetään b-bittistä muuttujaa sormenjäljen laskemisessa. Rivillä 7 tehdään taulukkohaku ikkunasta poistuneelle tavulle ja lasketaan taulukkoarvon ja edellisen sormenjäljen välinen poissulkeva-tai. Rivillä 8 taulukkohaku tehdään tavulle, joka on sormenjäljen ylimmissä biteissä. Käytännössä tämä taulukkoon osoittavan tavun

arvo saadaan siirtämällä sormenjälkimuuttujan bittejä $k - 8$ bittiä oikealle. Polynomilla x^8 kertominen voidaan toteuttaa siirtämällä sormenjälkimuuttujan bittejä kahdeksan bittiä vasemmalle. Tässä kuitenkin yläbitit $r_1, \ldots, r_8$ pitäisi nollata, jos sormenjälkimuuttujan bittileveys on suurempi kuin k. Jos esimerkiksi sormenjälkimuuttuja on 32-bittinen ja $k = 31$, niin sormenjälkimuuttujan siirtäminen kahdeksan bittiä vasemmalle jättää bitin r_8 ylimmäksi bitiksi, joka pitää nollata. Tämän yläbittien nollaamisen voi kuitenkin tehdä kätevästi asettamalla sopivat yläbitit jo taulukkoon T, jolloin ne nollaantuvat laskettaessa poissulkeva-tai taulukkoarvon kanssa. Juuri näin on tehty myös Gitin toteutuksessa [Pit07].

5.5.4 Jakajapolynomin valinta

Rabinin menetelmässä käytetyn jakajapolynomin $p(x)$ valinta vaikuttaa sormenjäljen laatuun. Mahdollisten sormenjälkien arvojen joukko on sitä suurempi, mitä suurempi on polynomin aste, joten pääsääntöisesti suuremman asteen polynomilla saadaan parempi erottelukyky kuin pienemmän asteen polynomilla.

Rabinin raportissa "Fingerprinting By Random Polynomials" käytetään vain jaottomia polynomeja, joiden aste k on alkuluku. Raportissa osoitetaan, että tällöin todennäköisyys sille, että n-pituisen bittijonon sormenjälki törmäisi jonkin m-pituisen bittijonon n-pituisen osajonon sormenjäljen kanssa, on alle ε, jos jakajapolynomi valitaan satunnaisesti ja sen aste $k > \log_2(nm/\varepsilon)$. [Rab81]

Raportissa myös todistetaan, että alkulukuastetta k olevia jaottomia polynomeja on $(2^k - 2)/k$ kappaletta ja esitetään menetelmä, jolla näistä voidaan valita satunnaisesti yksi, kunhan on jo löydetty jokin astetta k oleva jaoton polynomi [Rab81]. Deltapakkauksen kannalta ei ole kuitenkaan tarpeen vaihtaa jakajapolynomia, kun sellainen on kerran valittu, joten jaottoman polynomin valitseminen satunnaisesti ei ole tarpeen.

Gitissä jakajapolynomia ei vaihdeta satunnaisesti vaan Gitin deltapakkaus käyttää aina yhtä ja samaa polynomia Rabinin sormenjälkien laskemisessa. Sen käyttämän jakajapolynomin kertoimista muodostetun binääriluvun heksadesimaaliesitys on ab59b4d1 [Pit07]. Heksadesimaaliesityksestä saadaan polynomin

kertoimet, kun se muutetaan binääriluvuksi. Polynomi on siis

$$x^{31}+x^{29}+x^{27}+x^{25}+x^{24}+x^{22}+x^{20}+x^{19}+x^{16}+x^{15}+x^{13}+x^{12}+x^{10}+x^{7}+x^{6}+x^{4}+1,$$

joka on jaoton. Jaottomuuden voi tarkistaa esimerkiksi menetelmällä, joka on kuvailtu lähteessä [Rab80, Lemma 1].

Koska jakajapolynomia ei vaihdeta satunnaisesti, niin on helppoa tehdä tiedosto, jonka sisällössä on paljon sormenjälkien törmäyksiä. Tällä ei ole kuitenkaan suurta merkitystä tietoturvan kannalta, koska Rabinin sormenjälkeä ei käytetä Gitissä mihinkään deltapakkausta kriittisempään toimintoon. Deltapakkauksessakaan sormenjälkien törmäykset eivät estä kokonaan deltapakkausalgoritmin toimintaa vaan korkeintaan hidastavat sitä, sillä sormenjälkiin ei luoteta sokeasti vaan sisältöjen vastaavuus tarkistetaan aina lisäksi.

Rabinin raportissa ei esitetä perusteluja sille, että sormenjälkien törmäysten todennäköisyyden pienentämisen kannalta jaottomat polynomit olisivat parempia kuin jaolliset, tai että polynomin asteen olisi parempi olla alkuluku kuin yhdistetty luku. Ainakin jotkin pieniin tekijöihin jakautuvat polynomit olisivat erittäin huonoja valintoja jakajapolynomiksi. Esimerkiksi polynomin x^k käyttäminen jakajana antaisi sormenjäljeksi aina viestin k alinta bittiä eikä pidemmän viestin yläbitit näin ollen vaikuttaisi ollenkaan sormenjäljen arvoon.

Tein tietokoneella kokeita sormenjälkien laskemisesta erilaisilla jakajapolynomeilla. Näytti siltä, että jaottomat polynomit antavat varmemmin tasaisen jakauman kuin muut polynomit. Kuitenkin jaollisten polynomien joukossa oli paljon hyviä polynomeja, jotka näyttivät suoriutuvan sormenjäljen laskemisesta ihan yhtä tasaisella jakaumalla kuin jaottomatkin.

Vaikka erilaisten jakajapolynomien käyttämisestä Rabinin menetelmässä ei näytä olevan julkaistu juuri mitään, niin erään vertailukohdan saa CRC-virheentarkistukseen liittyvistä tutkimuksista. Nimittäin Rabinin menetelmä on pitkälti sama kuin CRC-virheentarkistuksessa käytetään [Kra94]. Ero CRC-menetelmään on mahdollisesti poikkeavan aloitusarvon käyttäminen ja syötteen loppuun lisättävät nollabitit. CRC-tarkistukseen käytettäviä polynomeja on tutkittu jonkin verran. Esimerkiksi lähteessä [Koo02] on löydetty jaollinen 32-bittinen polynomi, joka on virheentarkistuksessa perinteistä jaotonta CRC32-polynomia parempi.

6 Johtopäätökset

Tutkimus osoitti, että Git-versiohallintajärjestelmän taustalla on paljon matemaattista teoriaa. Tämä ei välttämättä kuitenkaan tule mieleen vastaavan järjestelmän rakentajalle, sillä tämä matemaattinen teoria on hyvin piilotettu helposti käytettävien ohjelmakomponenttien, kuten zlib tai LibXDiff, toteutukseen. Aiheen laajuuden vuoksi kaikkea Gitin matemaattista taustaa ei mitenkään pystynyt käsittelemään tässä tutkielmassa. Tämän työn ulkopuolelle jätettiin muun muassa kryptografisten hash-funktioiden matemaattinen tarkastelu, merge-algoritmit, Git-objektien muodostamaan graafiin liittyvät graafialgoritmit sekä algoritmien kompleksisuustarkastelut.

Gitin käyttämän pakettitiedoston muodostamisen algoritmit saatiin avattua ja siltä pohjalta on aiempaa helpompi ymmärtää mihin Gitin tehokas pakkaus historian tallentamisessa perustuu. Olennaisia tekijöitä ovat deltapakkaus, sopivien deltaparien löytäminen ja yksittäisten objektien deflate-pakkaus.

Oli mielenkiintoista huomata miten kolmessa pääalueessa oli liitoskohtia toisiinsa: Deltapakkauksessa käytetyt etäisyys-pituus-parit ovat saman tyyppisiä kuin LZ77-koodauksessa. Toisaalta ensin esitelty muokkauskaavion käsite osoittautuikin yleisemmän käsitteen, deltan, erikoistapaukseksi.

Sanasto

Käännös	**Alkuperäinen termi**	**Sivu**
ahnas algoritmi	greedy algorithm	19
alimoduuli	submodule	6
alkukuvan vastustavuus	pre-image resistance	9
biteittäinen poissulkeva-tai	bitwise exclusive or, XOR	65
bittien siirtäminen	bit shift	65
blob-objekti	blob object	6
commit-objekti	commit object	7
deltaikkuna	delta window	59
deltapakkaus	delta compression	56
dynaaminen ohjelmointi	dynamic programming	41
hajautettu versiohallintajärjestelmä	distributed version control system, DVCS	4
hajota ja hallitse	divide and conquer	49
ikkunan koko	window size	13
irrallinen objekti	loose object	5
jakson pituuden koodaus	run-length encoding, RLE	23
kehityshaarojen yhdistäminen	branch merging	38
keskimato	middle snake	49
kohdeobjekti	target object	56
korjaustiedosto	patch	12
lineaarinen takaisinkytketty siirtorekisteri	linear feedback shift register, LFSR	67

Käännös	**Alkuperäinen termi**	**Sivu**
liukuva ikkuna	sliding window	22
lähdeobjekti	source object	56
mato	snake	45
muokkausetäisyys	edit distance	40
muokkausgraafi	edit graph	44
muokkauskaavio	edit script	38
muuttumaton	immutable	5
nimi-hash	namehash	58
pakettitiedosto	packfile	5
pakkaustaso	compression level	13
pisin yhteinen alijono	longest common subsequence, LCS	40
polkunimi	path name	58
suunnattu syklitön graafi	directed acyclic graph, DAG	1
syvyysparametri	depth parameter	59
tag-objekti	tag object	7
tavujono	byte sequence	2
tietovarasto	repository	4
tree-objekti	tree object	6
tuoreusjärjestys	recency order	58
työskentelyhakemisto	working tree	5
törmäyksiä vastustava	collision resistant	9
valmistelualue	staging area	5
versionhallintajärjestelmä	version control system, VCS	4
viittaus	reference, ref	5
yhden komplementti	ones' complement	28

Lähteet

[Cha09] Scott Chacon. *Pro Git*. New York: Apress, 2009. ISBN: 1-4302-1833-9. URL: http://git-scm.com/book/ (viitattu 14. 09. 2012) (ks. s. 11).

[Cha11a] Scott Chacon. *Pro Git – Getting Started*. 27. 11. 2011. URL: https://github.com/progit/progit/blob/33e9d8/en/01-introduction/01-chapter1.markdown (viitattu 13. 09. 2012) (ks. s. 4).

[Cha11b] Scott Chacon. *Pro Git – Git Internals*. 29. 10. 2011. URL: https://github.com/progit/progit/blob/75cb3b/en/09-git-internals/01-chapter9.markdown (viitattu 15. 07. 2012) (ks. s. 5, 6, 13, 26, 56).

[DPV06] Sanjoy Dasgupta, Christos Papadimitriou ja Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2006. ISBN: 0-07-352340-2. URL: http://www.cs.berkeley.edu/~vazirani/algorithms.html (ks. s. 19, 42).

[Duy12] Nguyễn Thái Ngọc Duy. *Document format of basic Git objects*. Sähköpostiviesti. 19. 02. 2012. URL: http://thread.gmane.org/gmane.comp.version-control.git/190829/focus=191017 (viitattu 14. 09. 2012) (ks. s. 5).

[ECR11] Steve Babbage et al. *ECRYPT II Yearly Report on Algorithms and Keysizes (2010-2011)*. Versio 1.0. European Network of Excellence in Cryptology II, 06/2011. URL: http://www.ecrypt.eu.org/documents/D.SPA.17.pdf (viitattu 29. 03. 2012) (ks. s. 9).

[Git12a] *Git*. Gitin kotisivu. 2012. URL: http://gitscm.com/ (viitattu 13. 09. 2012) (ks. s. 4).

[Git12b] *Git – About – Branching and Merging*. 2012. URL: http://gitscm.com/about/branching-and-merging (viitattu 13. 09. 2012) (ks. s. 4).

[Git12c] *Git – About – Data Assurance*. 2012. URL: http://gitscm.com/about/info-assurance (viitattu 13. 09. 2012) (ks. s. 4).

[Git12d] *Git – About – Small and Fast*. 2012. URL: http://gitscm.com/about/small-and-fast (viitattu 13. 09. 2012) (ks. s. 4).

[Ham08] Junio C. Hamano. Dokumentaatiotiedosto pack-format.txt Git 1.7.5'n lähdekoodissa. 2008. URL: https://github.com/git/git/blob/v1.7.5/Documentation/technical/pack-format.txt (viitattu 24. 07. 2012) (ks. s. 56).

[Hir75] Daniel S. Hirschberg. "A Linear Space Algorithm for Computing Maximal Common Subsequences". *Communications of the ACM* 18.6 (06/1975)., s. 341–343 (ks. s. 42).

[Huf52] D.A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". *Proceedings of the I.R.E.* 40.9 (09/1952)., s. 1098–1101. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1952.273898 (ks. s. 13).

[Koo02] Philip Koopman. "32-Bit Cyclic Redundancy Codes for Internet Applications". Teoksessa: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, s. 459–468. ISBN: 0-7695-1597-5. DOI: 10.1109/DSN.2002.1028931 (ks. s. 71).

[Kra94] Hugo Krawczyk. "LFSR-based Hashing and Authentication". Teoksessa: *Advances in Cryptology – CRYPTO '94*. Toim. Yvo Desmedt. Vol. 839. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1994, s. 129–139. ISBN: 978-3-540-58333-2. DOI: 10.1007/3-540-48658-5_15 (ks. s. 67, 71).

[Lib03] Davide Libenzi. LibXDiff-kirjaston tiedosto xdiffi.c Git 1.7.5'n lähdekoodissa. 2003. URL: https://github.com/git/git/blob/v1.7.5/xdiff/xdiffi.c (viitattu 11. 07. 2012) (ks. s. 53, 55).

[Libx12] *LibXDiff Home Page*. 2012. URL: http://www.xmailserver.org/xdiff-lib.html (viitattu 08. 06. 2012) (ks. s. 44, 62).

[Loe06] Jon Loeliger, toim. Dokumentaatiotiedosto pack-heuristics.txt Git 1.7.5'n lähdekoodissa. 2006. URL: https://github.com/git/git/blob/v1.7.5/Documentation/technical/pack-heuristics.txt (viitattu 24. 07. 2012) (ks. s. 58, 59).

[Mac00] Joshua P. MacDonald. *File System Support for Delta Compression*. Tekninen raportti. Berkeley, California, USA: University of California at Berkeley, 2000. URL: `http://xmailserver.org/xdfs.pdf` (ks. s. 56, 62, 63).

[Mye86] Eugene Myers. "An O(ND) Difference Algorithm and Its Variations". *Algorithmica* 1 (1 1986)., s. 251–266. ISSN: 0178-4617. DOI: `10.1007/BF01840446` (ks. s. 38, 44, 53).

[Net+69] Network Working Group et al. *RFC 20: ASCII format for Network Interchange*. 10/1969. URL: `http://tools.ietf.org/pdf/rfc20` (ks. s. 3).

[Net+96a] Network Working Group et al. *RFC 1950: ZLIB Compressed Data Format Specification*. Versio 3.3. 05/1996. URL: `http://tools.ietf.org/pdf/rfc1950` (ks. s. 13).

[Net+96b] Network Working Group et al. *RFC 1951: DEFLATE Compressed Data Format Specification*. Versio 1.3. 05/1996. URL: `http://tools.ietf.org/pdf/rfc1951` (ks. s. 13, 26).

[NIST95] *Secure Hash Standard, FIPS PUB 180-1*. U.S. Department of Commerce / National Institute of Standards and Technology, 17. 04. 1995. URL: `http://www.itl.nist.gov/fipspubs/fip180-1.htm` (viitattu 14. 09. 2012) (ks. s. 6, 9).

[Pit+10] Nicolas Pitre et al. Tiedosto `pack-objects.c` Git 1.7.5'n lähdekoodissa. 2010. URL: `https://github.com/git/git/blob/v1.7.5/builtin/pack-objects.c` (viitattu 24. 07. 2012) (ks. s. 59).

[Pit06] Nicolas Pitre. *Replace adler32 with Rabin's polynomial in diff-delta*. Commit Gitin lähdekoodiin. 29. 04. 2006. URL: `https://github.com/git/git/commit/3dc5a9e4cd` (viitattu 30. 07. 2012) (ks. s. 64).

[Pit07] Nicolas Pitre. Tiedosto `diff-delta.c` Git 1.7.5'n lähdekoodissa. 2007. URL: `https://github.com/git/git/blob/v1.7.5/diff-delta.c` (viitattu 24. 07. 2012) (ks. s. 61–63, 68, 70).

[RA12] Greg Roelofs ja Mark Adler. *zlib Home Site*. 05. 02. 2012. URL: `http://www.zlib.net/` (viitattu 28. 04. 2012) (ks. s. 13, 37).

[Rab80] Michael O. Rabin. ”Probabilistic algorithms in finite fields”. *SIAM Journal on Computing* 9.2 (1980)., s. 273–280. URL: http://www.bradblock.com/Probabilistic_Algorithms_in_Finite_Fields.pdf (viitattu 29. 07. 2012) (ks. s. 71).

[Rab81] Michael O. Rabin. *Fingerprinting By Random Polynomials*. Tekninen raportti TR-15-81. Center for Research in Computing Technology, Harvard University, 1981. URL: http://www.xmailserver.org/rabin.pdf (ks. s. 65, 66, 70).

[Sal06] David Salomon. *Data Compression: The Complete Reference*. 4. painos. Springer, 12/2006. ISBN: 1-84628-602-6 (ks. s. 13, 22, 23, 26).

[Ter09] Mike Terzza. *Linux Compression Comparison (GZIP vs BZIP2 vs LZMA vs ZIP vs Compress)*. 30. 08. 2009. URL: http://blog.terzza.com/linux-compression-comparison (viitattu 27. 03. 2012) (ks. s. 13).

[Tor06] Linus Torvalds. *Use a real built-in diff generator*. Commit Gitin lähdekoodiin. 25. 03. 2006. URL: https://github.com/git/git/commit/3443546f6e (viitattu 08. 06. 2012) (ks. s. 44).

[Tor07] Linus Torvalds. *Tech Talk: Linus Torvalds on git*. Videoitu esitelmä. Transkriptio saatavissa osoitteesta https://git.wiki.kernel.org/index.php/LinusTalk200705Transcript. 03. 05. 2007. URL: http://www.youtube.com/watch?v=4XpnKHJAok8 (viitattu 14. 09. 2012) (ks. s. 9).

[Tor08] Linus Torvalds. *Achieving efficient storage of weirdly structured repos*. Sähköpostiviesti. 04. 04. 2008. URL: http://thread.gmane.org/gmane.comp.version-control.git/78774/focus=78825 (viitattu 19. 07. 2012) (ks. s. 58).

[Wel07] Morten Welinder. *10x+ Better Compression Than Gzip*. 2007. URL: http://blogs.gnome.org/mortenw/2007/01/07/10x-better-compression-than-gzip/ (viitattu 24. 07. 2012) (ks. s. 56).

[Wik12a] Wikipedia. *Central binomial coefficient – Wikipedia, The Free Encyclopedia*. 2012. URL: http://en.wikipedia.org/wiki/Central_binomial_coefficient (viitattu 05. 06. 2012) (ks. s. 41).

[Wik12b] Wikipedia. *Entropy encoding – Wikipedia, The Free Encyclopedia*. 2012. URL: http://en.wikipedia.org/wiki/Entropy_encoding (viitattu 14. 04. 2012) (ks. s. 13).

[Wik12c] Wikipedia. *Git (software) – Wikipedia, The Free Encyclopedia*. 2012. URL: http://en.wikipedia.org/wiki/Git_(software) (viitattu 13. 09. 2012) (ks. s. 4, 5, 10).

[Wik12d] Wikipedia. *LZ77 and LZ78 – Wikipedia, The Free Encyclopedia*. 2012. URL: http://en.wikipedia.org/wiki/LZ77_and_LZ78 (viitattu 24. 04. 2012) (ks. s. 22, 23).

[Yli06] Kari Ylinen. *Informaatioteoria*. Luentomoniste. 2006 (ks. s. 14, 15, 34).

[ZL77] Jacob Ziv ja Abraham Lempel. "A Universal Algorithm for Sequential Data Compression". *IEEE Transactions on Information Theory* 23.3 (05/1977)., s. 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714 (ks. s. 22, 23).