Maisterintutkielma

Tietojenkäsittelytiede

Tilatiivis toteutus tiedon tiivistämiseen osamerkkijonoja luettelemalla

Aleksi Hankalahti

1.6.2020

Matemaattis-luonnontieteellinen tiedekunta
Helsingin yliopisto

Ohjaaja(t)

prof. Veli Mäkinen, FT Juha Kärkkäinen

Tarkastaja(t)

prof. Veli Mäkinen, dos. Niko Välimäki

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Aleksi Hankalahti

Työn nimi — Arbetets titel — Title

Tilatiivis toteutus tiedon tiivistämiseen osamerkkijonoja luettelemalla

Ohjaajat — Handledare — Supervisors

prof. Veli Mäkinen, FT Juha Kärkkäinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	1.6.2020	66 sivua

Tiivistelmä — Referat — Abstract

Häviöttömässä tiedon tiivistämisessä annetusta datasta luodaan tiiviste, joka vie mahdollisimman vähän tilaa suhteessa alkuperäiseen dataan. Tiivisteestä on voitava palauttaa identtinen kopio alkuperäisestä datasta.

Tutkielmassa käsitellään häviötöntä tiivistysmenetelmää, joka tutkii tiivistettävää dataa, eli merkkijonoa tai tekstiä, kokonaisuutena, eikä esimerkiksi pieni osa kerrallaan. Menetelmä välittää tiivisteen purkajalle osamerkkijonojen esiintymismääriä tekstissä. Osamerkkijonot käsitellään ennalta tunnetussa järjestyksessä lyhyimmästä pisimpään, jolloin kumpikin osapuoli osaa liittää esiintymismäärän oikeaan osamerkkijonoon. Jotkut esiintymismäärät voivat olla nollia kertomassa, ettei osamerkkijono esiinny tekstissä. Tiivistyvyys saavutetaan huomaamalla, että aiemmin välitetyt osamerkkijonot rajaavat millaisia pidemmät merkkijonot voivat olla. Tällöin osa esiintymismääristä voidaan jättää välittämättä, tai välittämiseen käyttää vähemmän tilaa. Osamerkkijonoja, joiden esiintymismäärä täytyy välittää, karakterisoidaan maksimaalisuuden käsitteen avulla.

Maksimaalisten osamerkkijonojen etsiminen ja osamerkkijonojen esiintymismäärien laskeminen paljaasta tekstistä on hidasta. Siksi teksti täytyy tallettaa tietorakenteeseen, joka tukee tarvittuja operaatioita tehden niistä nopeita. Tällaiset tietorakenteet vievät enemmän tilaa kuin paljas teksti. Koska tutkittavassa tiivistysmenetelmässä koko tiivistettävä teksti käsitellään kokonaisuutena, muistinkäytön tehokkuus korostuu.

Tutkielmassa toteutetaan tiivistysmenetelmä käyttäen tilatiiviistä tietorakennetta nimeltä kaksisuuntainen BWT-indeksi. Tilatiiviit tietorakenteet vievät vain vähän enemmän tilaa, kuin niihin talletettu data. Tästä huolimatta ne toteuttavat talletettua dataa käsitteleviä operaatioita tehokkaasti. Toteutukselle suoritetut kokeet osoittavat muistinkäytön pysyvän kohtuullisena, jolloin suurempienkin tietomäärien tiivistys on mahdollista.

ACM Computing Classification System (CCS)
Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Data compression
Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Sorting and searching

Avainsanat — Nyckelord — Keywords

häviötön tiedon tiivistys, tilatiiviit tietorakenteet

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Algoritmien, data-analytiikan ja koneoppimisen erikoistumislinja

Sisällys

1 Johdanto

Tiivistysmenetelmän (engl. *compression method*) tavoitteena on välittää viesti yhdeltä taholta toiselle tiivistetyssä muodossa. Tiivistetyn viestin tulisi viedä vähemmän tilaa, eli bittejä, kuin alkuperäisen viestin. Jos tiivistysmenetelmä on häviötön (engl. *lossless*), tiivisteestä voidaan aina palauttaa identtinen kopio alkuperäisestä viestistä. Tässä työssä tarkastellaan häviötöntä tiivistysmenetelmää nimeltä *tiivistys osamerkkijonoja luettelemalla* (engl. *compression via substring enumeration*, CSE).

1.1 Esimerkki osamerkkijonojen luettelusta

Oletetaan että tehtävänä on välittää kahden tahon välillä viesti tai teksti *aabba*. Tekstiä ei välitetä suoraan, vaan luetellaan sen osamerkkijonoja ja niiden esiintymismääriä tekstissä. Esimerkkinä taulukossa 1.1 on listattu joitain merkkijonon "aabba" osamerkkijonoja ja niiden esiintymismäärät.

osamerkkijono	*esiintymismäärä*
a	3
b	2
aa	1
ab	1
ba	1
bb	1
aab	1

Taulukko 1.1: Eräitä merkkijonon aabba osamerkkijonoja esiintymismäärineen. Osamerkkijonot on lueteltu ensisijaisesti pituusjärjestyksessä ja saman pituiset osamerkkijonot aakkosjärjestyksessä.

Vastaanottajan tehtävänä on löytää lyhin merkkijono, jossa esiintyy taulukossa 1.1 listatut osamerkkijonot. Kukin esiintyy täsmälleen niin monta kertaa, kuin taulukossa mainitaan.

1.2 Tiivistysmenetelmien ominaisuuksia

Koska käsittelemme tiivistysmenetelmää, tavoitteena on välittää lähettäjän ja vastaanottajan välillä vähemmän bittejä kuin alkuperäinen teksti vie. Edellisen esimerkin taulukko vie selvästi enemmän tilaa kuin pelkkä teksti aabba. Pidemmällä tekstillä asetelma saattaa kuitenkin kääntyä päinvastaiseksi.

Työssä esiteltävä tiivistysmenetelmä menee pidemmälle. Lyhyemmät osamerkkijonot sisältävät informaatiota myös pidemmistä osamerkkijonoista. Edellisen esimerkin tapauksessa osamerkkijonon ab esiintymismäärä ei voi ylittää komponenttiensa, osamerkkijonojen

a tai b, esiintymismääriä. Lisäksi esimerkiksi osamerkkijonoa baa ei voi esiintyä alkuperäisessä tekstissä, koska merkkijono aab esiintyy. Viimeisen b symbolin täytyy seurata tätä osamerkkijonoa, sillä tiedetään osamerkkijonon bb esiintyvän tekstissä kerran. Jäljelle jää vain yksi symboli a, joka voidaan lisätä osamerkkijonon aabb perään. Tämä ei riitä muodostamaan osamerkkijonoa baa.

Osamerkkijonot voidaan välittää vastaanottajalle järjestyksessä lyhimmästä pisimpään ja saman pituiset merkkijonot aakkosjärjestyksessä. Tällöin edellisen kaltaisia rajoitusta voidaan hyödyntää rajaamaan välitettävän datan määrää, ja välittää vain ne osamerkkijonot, joiden esiintymismäärät eivät ole edellisten esiintymismäärien pohjalta ilmeisiä.

Implisiittinen aakkojärjestys vähentää välitettävän datan määrää entisestään, koska itse osamerkkijonoja ei tarvitse välittää. Kun aakkosto, josta merkkijonot muodostetaan, on ennalta sovittu, kumpikin taho tuntee kaikkien potentiaalisesti tekstistä löytyvien osamerkkijonojen järjestyksen. Siksi voidaan välittää vain niiden esiintymismäärät. Joillain osamerkkijonoilla esiintymismäärä saattaa olla 0, merkiten ettei sitä löydy alkuperäisestä tekstistä.

Jotta tiivistysmenetelmä olisi käytännössä hyödyllinen, tekstin tiivistämisen ja alkuperäisen tekstin palautuksen tiivisteestä täytyy onnistua riittävän vähällä laskennalla. Esiteltävä menetelmä pyrkii olemaan mahdollisimman suoraviivainen. Alkuperäisen tekstin koko kerrotaan aluksi viestin vastaanottajalle. Myös kaikkien osamerkkijonojen esiintymismäärät ilmoitetaan järjestyksessä, kuten aiemmin kuvattiin. Välitettyjen osamerkkijonojen määrästä muodostuu kaavio, kuten taulukossa 1.2. Jotta esimerkin koko pysyisi tarpeeksi pienenä, oletetaan, että aakkosto koostuu vain symboleista a ja b.

a (3)	b (2)						
aa (1)	ab (1)	ba (1)	bb (1)				
aaa (0)	aab (1)	aba (0)	abb (1)	baa (0)	bab (0)	bba (1)	bbb (0)
aaaa (0)	aaab (0)	⋯	aabb (1)	⋯	abba (1)	⋯	bbbb (0)
aaaaa (0)	aaaab (0)	⋯	aabba (1)	⋯	bbbbb (0)		

Taulukko 1.2: Kaikki symboleista a ja b muodostetut merkkijonot, joiden pituus on korkeintaan 5 symbolia. Suluissa on merkkijonon esiintymismäärä tekstissä aabba. Joitain tekstissä esiintymättömiä merkkijonoja on jätetty pois.

Pelkän aikavaativuuden lisäksi tiivistysmenetelmän käytännöllisyyteen vaikuttaa sen muistivaativuus. Tiivistettävän datan kokoon verrattuna moninkertaisesti muistia kuluttava tiivistysmenetelmä rajoittaisi kuinka suuria datamääriä sillä voi tiivistää. Työssä sovelletaan kaksisuuntaista BWT-indeksiä tiivistyalgorimiin. Tämä indeksi on tilatiivis tietorakenne, johon tiivistettävä teksti säilötään. Tietorakenne tarjoaa joukon operaatioita, joilla voidaan etsiä tekstistä sen osamerkkijonoja ja laskea niiden esiintymismääriä.

Tilatiiviys tarkoittaa, että n symbolin pituisella tekstillä tietorakenne vaatii $n \log \sigma(1+o(1))$ bittiä tilaa [27, s. 13]. Edellisessä σ on tekstin aakkoston koko. Yleisesti pelkän tekstin tallettaminen muistiin vaatii tilaa $n \log \sigma$ bittiä. Kerroin $(1 + o(1))$ pysyy erittäin lähellä lukua 1, kun tekstin pituus n kasvaa. Tässä työssä tietorakenteen avulla tehdään CSE-tiivistysmenetelmän toteutus, jolla voidaan tiivistää suurempia tiedostoja.

1.3 Tutkielman sisältö

Seuraavassa luvussa 2 käydään ensin lävitse jatkossa tarvittavia määritelmiä. Niiden jälkeen kuvataan tarkemmin minkälaisten osamerkkijonojen esiintymismäärät täytyy välittää vastaanottajalle. Erillinen aliluku 2.4 esittelee kaksi erilaista menetelmää, joiden avulla esiintymismäärät voidaan koodata tiivisteeseen. Luvussa 2.2 esitellään kaksisuuntainen BWT-indeksi. Tämän tilatiiviin tietorakenteen avulla myöhemmin, luvussa 2.3, luetellaan kaikki välitettävät osamerkkijonot. Lisäksi tietorakenteen avulla lasketaan osamerkkijonojen esiintymismäärät tiivistettävässä tekstissä. Luku 2.5 esittelee algoritmin, jonka avulla alkuperäinen teksti voidaan purkaa tiivisteestä.

Teoreettisessa osuudessa esiteltyjä algoritmeja hyödynnetään CSE-menetelmän toteutuksessa, joka esitellään luvussa 3. Toteutettu ohjelma tiivistää annetun tiedoston CSE-menetelmällä ja osaa purkaa tiivisteestä alkuperäisen tiedoston. Ohjelman esittelyn jälkeen, luvussa 3.1, tiivistyvyyttä, nopeutta ja muistinkulutusta verrataan muihin tiivistysmenetelmiin. Lopuksi luvussa 4 tehdään johtopäätöksiä kokeellisista tuloksista. Niiden lisäksi ehdotetaan aiheita, joita CSE-menetelmästä kannattaisi tutkia enemmän. Alla käydään vielä lävitse CSE-menetelmän yhtymäkohtia muihin häviöttömiin tiivistysmenetelmiin. artikkeleita, joita on julkaistu tiivistyksestä osamerkkijonoja luettelemalla.

1.4 Vertailu muihin tiivistysmenetelmiin

Tiivistys osamerkkijonoja luettelemalla poikkeaa monista tiivistysmenetelmästä siten, ettei se etene lineaarisesti tiivistettävän tekstin alusta loppuun. Esimerkiksi LZ77-algoritmi etenee pitkin tekstiä ja muodostaa sanakirjaa näkemistään osamerkkijonoista. Vastaantullut, ennestään tuttu osamerkkijono, koodataan viitteenä sanakirjaan [40]. Sen sijaan CSE-menetelmä tutkii koko tiivistettävän tekstin, ja muodostaa kokonaisuudesta kuvauksen, joka talletetaan tiivisteeksi.

Burrows ja Wheeler ovat esitelleet Burrows-Wheeler-muunnokseen perustuvan tiivistysmenetelmän artikkelissa [5]. Burrows-Wheeler-muunnosta hyödyntää myös yleinen tiivistysohjelma bzip2 [35]. Kuten CSE-menetelmä, nämä menetelmät voivat käsitellä tiivistettävää merkkijonoa kokonaisuutena. Mikäli merkkijono on liian pitkä, se voidaan pilkkoa lohkoiksi ja kukin lohko tiivistää erikseen muista riippumatta. Esimerkiksi bzip2-ohjelma tekee näin [35]. Samaa tekniikkaa on mahdollista käyttää CSE-menetelmän kanssa, mutta tässä työssä tiivistettävää merkkijonoa käsitellään yhtenä kokonaisuutena.

Antisanakirjakoodaus (engl. *antidictionary coding*) on häviötön tiivistysmenetelmä, jossa muodostetaan automaatti, joka hyväksyy merkkijonot, joiden osamerkkijonoja ei löydy antisanakirjasta [7]. Antisanakirjasta löytyvällä merkkijonolla on selvä yhteys CSE-menetelmän alimerkkijonoihin, joiden esiintymismäärä on 0 – kumpikin vastaa osamerkkijonoa, joka ei löydy alkuperäisestä tekstistä.

Myöhemmin luvussa 3.1 CSE-menetelmän tuottaman tiivisteen kokoa verrataan LZ77-menetelmään ja muihin tiivistysmenetelmiin. Verrattuna LZ77-menetelmään, CSE-menetelmä

hyötyy tiivistettävän datan suuresta koosta.

1.5 Kirjallisuuskatsaus

Alunperin tiivistäminen osamerkkijonoja luettelemalla on esitelty Danny Dubén ja Vincent Beaudoin artikkelissa [12]. Siinä menetelmä kuvataan binääriaakkostolle. Kuvauksen lisäksi artikkelissa verrataan menetelmän tuottaman tiivisteen tiiveyttä muihin tiivistysmenetelmiin.

Edellinen artikkeli jättää osoittamatta oletuksen, että tiivistysmenetelmä on universaali kaikilla k-asteen markovilaisilla informaatiolähteillä (engl. *order-k Markovian source*). Tämä osoitetaan todeksi Danny Dubén ja Hidetoshi Yokoon artikkelissa [13]. Universaaliudella he tarkoittavat, että tiivisteen koko suhteessa alkuperäisen tekstin kokoon lähenee lähteen informaatioteoreettista entropiaa, kun alkuperäisen tekstin pituus kasvaa rajatta. Lisäksi he osoittavat, että binäärisellä aakkostolla tiivis osamerkkijonoverkko on aina kooltaan $2N - 1$ solmua, missä N on merkkijonon pituus. Lisäksi oletetaan, että merkkijono ei ole toisteinen.

Hidetoshi Yokoo ehdottaa parannuksia edellisen menetelmän tapaan koodata esiintymismäärät tiivisteeseen [39]. Hän osoittaa, että uudella koodaustavalla varustettu menetelmä on universaali stationaarisilla ergodisilla informaatiolähteillä (engl. *stationary ergodic source*). Lopuksi artikkelissa esitellään tiivisteen purkumenetelmä, jolle tarvitsee välittää vähemmän osamerkkijonojen esiintymismääriä, kuin Dubén ja Beaudoin menetelmässä.

Sho Kanai, Hidetoshi Yokoo, Kosumo Yamazaki ja Hideaki Kaneyasu esittelevät uuden toteutusmenetelmän CSE-tiivistykselle binäärisellä aakkostolla [24]. Menetelmä hyödyntää Burrows-Wheeler-muunnosta ja LCP-taulukkoa (engl. *longest common prefix*).

Dubén ja Beaudoin artikkelissa [12] havaitaan, että tiivistysmenetelmä toimii paremmin englanninkieliselle tekstille kuin binääridatalle. Hypoteesina Dubé esittää tämän johtuvan siitä että menetelmä on bittipohjainen, eikä osaa mukautua dataan, jossa bitin sijainti tavussa, tai pidemmässä sanassa, on merkityksellinen [11].

Danny Dubén artikkeli [11] jatkaa ongelman tutkimista lisäämällä tiivistettävään dataan synkronointibittejä, jotka auttavat tiivistysmenetelmää kohdistamaan samassa *vaiheessa* (engl. *phase*), eli samassa kohtaa tavua, sijaitsevat bitit keskenään.

Esimerkkinä vaiheen merkityksestä artikkelissa annetaan seuraavat kaksi 11 bittiä pitkää osamerkkijonoa, joilla on sama ydin:

$$\begin{array}{ccccccccccc} a_2 & 0_3 & 1_4 & 1_5 & 1_6 & 0_7 & 0_8 & 1_1 & 0_2 & 0_3 & b_4 \\ c_7 & 0_8 & 1_1 & 1_2 & 1_3 & 0_4 & 0_5 & 1_6 & 0_7 & 0_8 & d_1 \end{array}$$

Oletetaan, että osamerkkijonot ovat peräisin konekoodista, jossa jokainen käsky on yhden kahdeksanbittisen tavun mittainen. Esimerkin vuoksi oletetaan edelleen, että käskytavun bitit 1-5 kertovat operaation, ja bitit 6-8 lähde- tai kohderekisterin. Kunkin bitin alaindeksinä on merkitty sen vaihe, eli sijainti kokonaisessa tavussa. Vaikka osamerkkijonoilla on

sama ydin, ne ovat selvästi eri vaiheessa. Bitti a on käskyn osassa, joka ilmaisee suoritettavaa operaatiota. Seuraavan osamerkkijonon bitti c ilmaisee jonkin toisen käskyn kohde- tai lähderekisteriä.

Ei ole syytä olettaa, että bitti operaatiota koodaavasta osasta noudattaisi samaa todennäköisyysjakaumaa kuin rekisteriä ilmaiseva bitti. Edellisessä esimerkissä tiivistysmenetelmällä ei kuitenkaan ole mahdollisuutta erotella näitä tapauksia, ja tämä saattaa kasvattaa tiivisteen kokoa. Vastaava ongelma pätee myös biteille b ja d.

Kirjoittajien hypoteesin mukaan englanninkielinen teksti tiivistyy paremmin, koska lähes kaikissa ASCII-merkistön tavallisissa merkeissä ylin bitti on 0. Artikkeli antaa seuraavan esimerkin:

$$\begin{array}{cccccccc} a_6 & a_7 & a_8 & 0_1 & a_2 & a_3 & a_4 & a_5 \\ b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & 0_1 & b_2 \end{array}$$

Oletetaan, että tässä tilanteessa muut kuin ylimmät bitit ovat täysin satunnaisia. Silloin todennäköisyys, että ylintä 0-bittiä vastaava merkki toisessa osamerkkijonossa on myös 0 on $\frac{1}{2}$. Ylin bitti toimii synkronointikoodina, vaikkakaan se ei takaa osamerkkijonojen kohdistuvan oikein. Kirjoittajat kuitenkin arvelevat tämän pienen vihjeen riittävän siihen, että samassa vaiheessa olevat osamerkkijonot ryhmittäytyvät yhteen, kun niiden pituus kasvaa.

Kirjoittajat pitävät eksplisiittiset synkronointikoodit mahdollisimman yksinkertaisina, jotta toteutus olisi yksinkertaista, ja tiivistysmenetelmällä olisi parhaat mahdollisuudet oppia ne. Oppimisella artikkelissa tarkoitetaan, että mahdollisimman varhain, samaa ydintä laajentavia osamerkkijonoja laskettaessa, merkkijonot ovat samassa vaiheessa.

Artikkelissa kokeillaan tiivistysmenetelmää ilman synkronointibittejä ja käyttäen 1-5 synkronointibittiä jokaista alkuperäisen testidatan tavua kohden. Kun käytössä on 5 synkronointibittiä, synkronointi on luotettava, eli kaksi alkuperäisen datan tavua eivät voi olla keskenään eri vaiheessa. Kokeissa havaitaan, että lähes poikkeuksetta synkronointibitit johtavat pienempään tiivisteeseen. Suurin etu saavutetaan jo vähemmällä kuin 5 synkronointibitillä, mikä viittaa siihen, että tiivistysmenetelmä oppii vaiheen jo pienistä vihjeistä. Kirjoittajat olettavat tämän olevan epäformaali viite aikaisemman hypoteesin paikkaansapitävyydestä.

Danny Bubén posteriesityksessä esitellään lisää kokeita, joissa kokeillaan vielä vahvempia synkronointitapoja, kuin edellisessä artikkelissa. Vahvemman synkronoinnin havaitaan parantavan tuloksia vain marginaalisesta [10].

Danny Vohl, Claude-Guy Quimper ja Danny Dubé esittelevät kaksi eri rajoitemallia (engl. *constraint model*), joilla voidaan muodostaa lyhimmät mahdolliset synkronointikoodit [38]. Toisin sanoen muodostaa synkronointikoodit, jotka kasvattavat tiivistettävää dataa mahdollisimman vähän.

Synkronointikoodien lisääminen tiivistettävään dataan saa muokkaamattoman tiivistysmenetelmän erottelemaan muuten samanlaiset, mutta eri vaiheessa, olevat osamerkkijonot toisistaan implisiittisesti. Mathieu Béliveau ja Danny Dubé esittelevät muokatun tiivistys-

menetelmän, jossa tietoisuus vaiheesta on rakennettu suoraan menetelmään [4]. Se ei siten tarvitse erikseen lisättyjä synkronointikoodeja.

Artikkelin kokeissa huomataan, että erillisiä synkronointikoodeja käyttävä ja menetelmään rakennettua vaihetietoisuutta hyödyntävät CSE-tiivistäjät saavuttavat käytännössä yhtä hyvän tiivistyvyyden. Eksplisiittisesti vaihetietoinen menetelmä tuottaa aina pienemmän tai saman kokoisen tiivisteen kuin verrokki. Tiivisteiden kokoerot ovat kuitenkin hyvin pieniä.

Laajennos äärelliselle aakkostolle. Takahiro Ota ja Hiroyoshi Morita soveltavat CSE-menetelmän tekniikoita antisanakirjakoodaukseen [30]. Kyseisessä artikkelissa käytetään vain binääriaakkostoa. Myöhemmin he esittelevät laajennoksen, jolla tiivistämistä osamerkkijonoja luettelemalla voidaan käyttää merkkijonoon, jonka aakkoston Σ koko $\sigma \geq 2$ on äärellinen [29].

Artikkeli esittelee, miten maksimaalista merkkijonoa w, $w \in \Sigma^*$, vasemmalle ja oikealle puolelle laajentavien merkkijonojen awc, $a, c \in \Sigma$, esiintymismäärämatriisin $\sigma \times \sigma$ alkiota voidaan koodata tiivisteeseen. Samalla he ehdottavat menetelmää, jolla lasketaan alkioiden arvoille ala- ja ylärajoja, joiden avulla alkion arvo voidaan koodata tiivisteeseen käyttäen vähemmän tilaa.

Takahiro Ota, Hiroyoshi Morita ja Akiko Manada esittelevät kaksi uutta menetelmää, joilla tiivisteen kokoa voidaan pienentää [32]. Ensimmäiseksi he ehdottavat uutta alarajaa lukumäärämatriisin alkioille, joka korvaa Otan ja Moritan esittelemän alarajan [29]. Uudessa alarajassa huomioidaan enemmän entuudestaan tunnettuja arvoja rajan laskussa. He osoittavat alarajan päteväksi, sekä näyttävät miten tietyissä tilanteissa se on aidosti parempi kuin aiempi alaraja.

Toinen parannus saavutetaan järjestelemällä esiintymismäärämatriisin sarakkeet laskevaan suuruusjärjestykseen sarakesummien mukaan. Sarakesummat vastaavat aiemmin välitettyjen osamerkkijonojen esiintymismääriä, joten ne ovat jo purkajan tiedossa. Järjestämisellä pyritään saamaan suurin osa matriisin kokonaissummasta vasempaan laitaan, josta matriisin koodaus tiivisteeseen aloitetaan. Kirjoittajien olettamuksen mukaan tekniikalla voidaan parantaa ala- ja ylärajoja alkioiden arvoille. Artikkelissa julkaistut kokeet näyttävät, että tekniikasta on apua joissain tapauksissa.

Sovellus useampiulotteiselle datalle. Binääriaakkostolle toteutetun CSE-menetelmän voidaan ajatella käsittelevän tiivistettävää dataa yksiulotteisena. Tiivistettävä data on renkaaksi taivutettu binäärimerkkijono, jonka osamerkkijonojen esiintymismääristä ollaan kiinnostuneita. Takahiro Ota ja Hiroyoshi Morita esittävät miten menetelmää voidaan laajentaa kaksiulotteisen binääridatan tiivistämiseen [31]. Kaksiulotteista binääridataa voidaan ajatella $n \times m$ kokoisena matriisina, jossa on n riviä ja m saraketta. Artikkelissa matriisi taivutetaan litteäksi torukseksi (engl. *flat torus*), jolloin rivit muodostavat renkaan, eli riviltä n voidaan edetä riville 1 ja päinvastoin. Vastaavasti sarakkeet taivutetaan renkaaksi siten, että sarakkeesta m pääsee sarakkeeseen 1 ja päinvastoin.

Kaksiulotteinen CSE on myös vaihtoehtoinen tapa käsitellä äärellisen aakkoston merkkijonoja. Tällöin matriisin kukin sarake edustaa yhtä merkkijonon symbolia. Rivejä matriisissa on yhtä monta kuin aakkostossa on symboleita. Kunkin merkkijonon indeksin kohdalla tasan yhdellä rivillä on bitti 1 merkitsemässä symbolia, joka merkkijonossa on sillä kohtaa.

Danny Dubé laajentaa ideaa k-ulotteiselle datalle [9], missä $k \in \mathbb{N}^+$. Artikkelissa laajennosta sovelletaan kaksiulotteisen harmaasävykuvan tallentamiseen kolmiulotteisena datana. Sovelluksessa kuvan tasogeometriset ulottuvuudet vastaavat matriisin leveyttä ja korkeutta. Pikseleiden tummuus muodostaa kolmannen ulottuvuuden, kuten edellä aakkoston symbolit muodostivat toisen ulottuvuuden. Dubén mukaan useampiulotteisille-CSE menetelmille ei vielä ole olemassa toteutusta.

2 Menetelmä

Selvyyden vuoksi jatkossa käytetään merkintää $\mathbb{N}^0$ kaikkien luonnollisten lukujen joukosta – sisältäen nollan, eli luvuista $0, 1, 2, \ldots$. Vastaavasti $\mathbb{N}^+$ on kaikkien positiivisten luonnollisten lukujen joukko eli $\mathbb{N}^+ = \mathbb{N}^0 \setminus \{0\}$.

Työssä tutustutaan yksinomaan äärelliselle aakkostolle kehitettyyn tiivistysmenetelmään. Aakkoston koko on jokin äärellinen vakio $\sigma \in \mathbb{N}^+$ siten, että $\sigma \geq 2$. Kaikki alla esitellyt menetelmät toimivat myös binääriaakkostolle, jolloin $\sigma = 2$. Mutta, kuten luvussa 1 mainittiin, on tiivistysmenetelmästä erityisesti binääriaakkostolle kehitettyjä versioita, jotka saattavat toimia sillä paremmin, tai ovat yksinkertaisempia toteuttaa.

Koko σ symbolin kokoista aakkostoa merkitään symbolilla Σ ja aakkoston symboleita luvuilla $1, 2, \ldots \sigma$. Olkoon w jokin merkkijono eli peräkkäinasettelu tai konkatenaatio $n \in \mathbb{N}^0$ kappaleesta aakkoston Σ symboleita. Merkkijono voidaan kirjoittaa auki n symbolin konkatenaationa $w = w_1 w_2 \cdots w_n$. Merkkijonon w pituutta n merkitään $|w| = n$.

Kaikkien $n \in \mathbb{N}^0$ symbolin pituisten merkkijonojen muodostamaa joukkoa merkitään Σ^n. Vastaavasti kaikkien äärellisen pituisten merkkijonojen joukkoa merkitään $\Sigma^* = \cup_{i \in \mathbb{N}^*} \Sigma^i$. Merkintä Σ^∞ vastaa kaikkien aakkostosta Σ muodostettujen *äärettömien* merkkijonojen joukkoa.

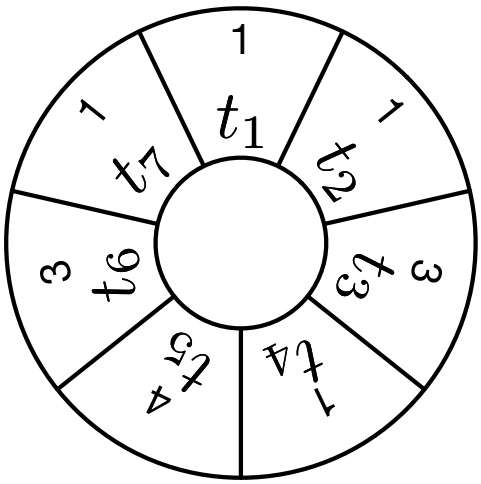
Kahden merkkijonon $v = v_1 v_2 \cdots v_n \in \Sigma^*$ ja $u = u_1 u_2 \cdots u_m \in \Sigma^*$ konkatenaatio on $vu = v_1 v_2 \cdots v_n u_1 u_2 \cdots u_m = w$. Sanotaan että merkkijonot v ja u ovat merkkijonon w *osamerkkijonoja* (engl. *substring*). Edelleen osamerkkijono v on merkkijonon w *alkuosa* (engl. *prefix*) ja osamerkkijono u on merkkijonon w *loppuosa* (engl. *suffix*).

Merkkijonon w, jonka pituus on $n \in \mathbf{N}^*$ symbolia, osamerkkijonoa voidaan merkitä lyhyesti $w[i..j]$, missä $1 \leq i, j \leq n$. Jos $j < i$ on kyseessä tyhjä osamerkkijono ϵ.

Tiivistettävä teksti on jonkin aakkoston Σ merkkijono $T \in \Sigma^*$. Tässä työssä vaaditaan, että merkkijono T ei ole toistojaksoinen, eli ei ole olemassa merkkijonoa $w \in \Sigma^*$ siten, että $T = w^n$ jollain luonnollisella luvulla $n \geq 2$. Toistojaksoinen merkkijono aiheuttaisi ongelmia purettaessa tiivistettä, kuten selitetään luvussa 2.5. Myöhemmin ongelma kierretään lisäämällä tiivistettävän merkkijonon loppuun erityinen symboli #, joka ei esiinny merkkijonon aakkostossa. Näin syntynyt merkkijono ei voi olla toistojaksoinen. Toiseksi tiivistettävän merkkijonon pituuden täytyy olla vähintään 3 symbolia, jotta tiivistysmenetelmä kykenee käsittelemään sitä. Tämä ei ole oleellinen rajoite.

Tavalliset merkkijonot ovat ongelmallisia, kun lasketaan osamerkkijonojen esiintymismääriä merkkijonoissa. Esimerkkinä otetaan kaksi merkkijonoa $w = 11344$ ja $v = 41134$. Merkkijono v on luotu merkkijonosta w siirtämällä kaikkia symboleita yhden askeleen oikealle ja siirtämällä viimeinen symboli 4 merkkijonon alkuun. Edelliset ovat siis *kiertoja* samasta merkkijonosta.

Osamerkkijono 44 esiintyy kierrossa w kerran ja kierrossa v ei kertaakaan. Tämän pe-



Kuva 2.1: Tekstistä $T = 1131431$ muodostettu merkkijonorengas T_r.

rusteella merkkijonon alulla ja lopulla on erityisasema laskettaessa osamerkkijonojen esiintymismääriä ja ne pitäisi huomioida jotenkin.

2.1 Merkkijonorenkaat

Esiteltävä tiivistysmenetelmä kiertää ongelman käsittelemällä tavallisten merkkijonojen sijaan renkaan muotoisia merkkijonoja: tiivistettävä teksti $T = t_1 t_2 \cdots t_n$ taivutetaan merkkijonorenkaaksi T_r. Renkaassa oikealle kuljettaessa symbolia t_n seuraa symboli t_1. Vastaavasti vasemmalle kuljettaessa symbolia t_1 seuraa symboli t_n. Kuvassa 2.1 on esimerkkinä tekstistä $T = t_1 t_2 t_3 t_4 t_5 t_6 t_7 = 1131431$ muodostettu rengas. Alla Määritelmä 2.1 esittelee mitä tässä työssä tarkoitetaan osamerkkijonon sijainnille merkkijonorenkaassa.

Määritelmä 2.1. *Olkoon Σ äärellinen aakkosto ja $T \in \Sigma^*$ mielivaltainen merkkijono, jonka pituus on $n \in \mathbb{N}^*$. Merkkijonorengas T_r on muodostettu merkkijonosta T.*

Merkkijono $w \in \Sigma^*$ esiintyy merkkijonorenkaassa T_r kohdassa tai indeksissä i, jos on olemassa merkkijonot $u \in \Sigma^*$ ja $v \in \Sigma^\infty$ siten että $|u| = i - 1 < n$ ja $uwv = T_r$.

Kuvassa 2.1 symboleiden indeksit $1, 2, \ldots 7$ on merkitty selvyyden vuoksi muodossa $t_1, t_2, \ldots t_7$. Renkaan koko $n \in \mathbb{N}^*$ on sama kuin alkuperäisen tekstin T koko, eli $n = |T|$.

Osamerkkijonon $w = 13$ voisi tulkita esiintyvän edellisen esimerkin merkkijonorenkaassa T_r äärettömän monta kertaa. Ensimmäinen esiintymä alkaa indeksistä 2. Toinen esiintymä on tasan yhden kierroksen eli 7 merkin päässä ensimmäisestä esiintymästä. Tällä tavalla voitaisiin aina edetä yksi kokonainen kierros ja päätyä uudelleen ja uudelleen osamerkkijonon 13 alkuun. Tällä määritelmällä jokainen merkkijono esiintyy merkkijonorenkaassa joko ei kertaakaan tai äärettömän monta kertaa. Jotta esiintymismäärät olisivat toimiva työkalu seuraava määritelmä samaistaa yllä mainitut äärettömän monta esiintymiskertaa yhdeksi esiintymäksi:

Määritelmä 2.2. *Olkoon T_r merkkijonorengas jonka koko on $n \in \mathbb{N}^*$ symbolia. Merkkijono $w \in \Sigma^*$ esiintyy renkaassa $C_w(T_r)$ kertaa, jos on olemassa $C_w(T_r)$ erillistä kohtaa, joissa merkkijono w esiintyy.*

Edellisellä määritelmällä 2.2 merkkijono 13 esiintyy nyt merkkijonorenkaassa $T_r = 1131431$ tasan kerran, eli $C_{13}(T_r) = 1$. Merkkijono 11 esiintyy renkaassa kahdesti, eli $C_{11}(T_r) = 2$. Ensimmäinen esiintymä alkaa indeksistä 1 ja toinen indeksistä 7. Toisen esiintymän toinen symboli on indeksissä 1, eli renkaassa on menty seuraavalle kierrokselle. Huomattavaa on että merkkijonorenkaassa esiintyy myös sen pituutta $n = 7$ pidempiä merkkijonoja. Esimerkiksi merkkijono 311131431 esiintyy renkaassa kerran. Tyhjän merkkijonon ϵ esiintymismääräksi määritellään merkkijonorenkaan koko, eli alkuperäisen merkkijonon pituus. Tällöin $C_\epsilon(T_r) = |T|$. Jatkossa merkintä $C_{11}(T_r)$ lyhennetään C_{11}, kun kontekstista on selvää mitä merkkijonorengasta T_r tarkastellaan.

Kuten aiemmin on mainittu, tiivistäjä välittää purkajalle merkkijonoja lyhimmästä pisimpään ja samankokoiset merkkijonot aakkosjärjestyksessä. Esimerkin vuoksi olkoon kaksi merkkijonoa $w = 12$ ja $v = 112$. Perinteisen aakkosjärjestyksen mukaan $v < v$. Kätevyyden nimissä määritellään uusi aakkojärjestys:

Määritelmä 2.3. *Merkkijono $w \in \Sigma^*$ on aakkosjärjestyksessä pienempi kuin merkkijono $v \in \Sigma^*$ jos ja vain jos toinen seuraavista ehdoista pätee:*

1. *$|w| < |v|$, tai*
2. *$|w| = |v|$ ja merkkijono w on tavanomaisessa aakkosjärjestyksessä pienempi kuin merkkijono v.*

Jatkossa aakkosjärjestyksestä puhuttaessa tarkoitetaan nimenomaan määritelmän 2.3 mukaista järjestystä.

Tiivistäjän ei tarvitse välittää purkajalle kaikkien osamerkkijonojen esiintymismääriä. Välitettävää osaa osamerkkijonoista voidaan kuvata maksimaalisuuden käsitteen avulla. Alla maksimaalisuus määritellään ensin tarkasti ja sen jälkeen osoitetaan, että niiden avulla voidaan välittää kaikki tarvittava informaatio merkkijonorenkaasta T_r.

Ensimmäiseksi määritellään oikealle maksimaalisuus:

Määritelmä 2.4. *Merkkijono w on oikealle maksimaalinen renkaassa T_r jos $C_w > C_{wc}$ kaikilla symboleilla $c \in \Sigma$.*

Oikealle maksimaaliset merkkijonot ovat sellaisia renkaassa esiintyviä osamerkkijonoja, joita voidaan jatkaa oikealle vähintään kahdella eri symbolilla. Kääntäen, jos merkkijono $w \in \Sigma^*$ ei ole oikealle maksimaalinen, on olemassa vain yksi symboli $c \in \Sigma$, jolla sitä voidaan jatkaa oikealle siten, että uusi merkkijono wc esiintyy renkaassa. Esimerkkikuvan 2.1 merkkijonorenkaassa $T_r = 1131431$ oikealle maksimaaliset osamerkkijonot ovat $\mathcal{R}_{T_r} = \{1, 11, 31\}$.

Määritelmä 2.4 vaatii implisiittisesti, että merkkijonon w täytyy esiintyä merkkijonorenkaassa. Jos näin ei ole, eli $C_w = 0$, mikään yhdellä symbolilla sitä jatkava merkkijono ei voi myöskään esiintyä renkaassa. Edelleen jos $C_w = 1$ ei voi olla kahta erilaista symbolia $c, d \in \Sigma$ siten että $C_{wc} > 0$ ja $C_{wd} > 0$. Eli merkkijonon täytyy esiintyä vähintään kaksi kertaa renkaassa, jotta se voisi olla oikealle maksimaalinen.

Oletetaan siksi, että w esiintyy merkkijonorenkaassa. Koska määritelmä käsittelee rengasta, on aina olemassa vähintään yksi symboli $c \in \Sigma$, jolla merkkijonoa w voidaan jatkaa oikealle siten, että näin syntynyt merkkijono wc edelleen esiintyy renkaassa.

Vastaavalla tavalla määritellään vasemmalle maksimaaliset merkkijonot:

Määritelmä 2.5. *Merkkijono w on vasemmalle maksimaalinen renkaassa T_r jos $C_w > C_{aw}$ kaikilla symboleilla $a \in \Sigma$.*

Yllä luetellut oikealle maksimaalisten merkkijonojen ominaisuudet pätevät myös vasemmalle maksimaalisiin merkkijonoihin. Merkkijonoa vain laajennetaan vasemmalle eikä oikealle. Kuvan 2.1 merkkijonorenkaassa vasemmalle maksimaaliset osamerkkijonot ovat $\mathcal{L}_{T_r} = \{1, 3, 11, 31\}$.

Edelliset määritelmät yhdistämällä saadaan seuraava määritelmä maksimaalisuudelle:

Määritelmä 2.6. *Merkkijono w on maksimaalinen renkaassa T_r jos se on sekä oikealle että vasemmalle maksimaalinen.*

Edellä oli lueteltu kuvan 2.1 renkaaseen liittyvien oikealle ja vasemmalle maksimaaliset merkkijonot. Näiden joukkojen leikkaus on kaikki maksimaaliset merkkijonot $\mathcal{M}_{T_r} = \mathcal{R}_{T_r} \cap \mathcal{L}_{T_r} = \{1, 11, 31\} \cap \{1, 3, 11, 31\} = \{1, 11, 31\}$.

Maksimaalisen osamerkkijonon w esiintymismäärää C_w ei välitetä tiivisteen purkajalle. Sen sijaan luetteloidaan kaikki merkkijonot awc, missä $a, c \in \Sigma$ ja välitetään niiden esiintymismäärät. Tällaiset merkkijonot siis laajentavat maksimaalista osamerkkijonoa w yhdellä symbolilla sekä oikealle että vasemmalle. Merkkijonojen esiintymismäärät ilmaistaan *esiintymismäärämatriisilla*

$$\boldsymbol{C}_w = \begin{bmatrix} C_{1w1} & C_{1w2} & \cdots & C_{1w\sigma} \\ C_{2w1} & C_{2w2} & \cdots & C_{2w\sigma} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\sigma w1} & C_{\sigma w2} & \cdots & C_{\sigma w\sigma} \end{bmatrix}. \tag{2.1}$$

Matriisin $\boldsymbol{C}_w$ nimi viittaa merkkijonoon w jota laajennetaan kahdella symbolilla. Rivit vastaavat symboleita, joilla merkkijonoa laajennetaan vasemmalle. Sarakkeet taas vastaavat symboleita, joilla merkkijonoa w laajennetaan oikealle.

Matriisin notaatio poikkeaa jonkin verran totutusta. Normaalisti rivillä 2 ja sarakkeessa 3 sijaitseva alkio voitaisiin ilmaista merkinnällä $c_{2,3}$. Tässä tekstissä sama alkio merkitään C_{2w3}. Merkintä samaistaa alkion aiempaan määritelmään merkkijonon $2w3$ esiintymismääristä merkkijonorenkaassa.

Valitaan esimerkiksi kuvan 2.1 merkkijonorenkaan T_r maksimaalinen osamerkkijono $w = 1$. Laajennetaan se yhdellä symbolilla sekä oikealle että vasemmalle. Syntyneet merkkijonot ovat muodoa awc missä $a, c \in \Sigma = \{1, 2, 3, 4\}$ ja vastaava esiintymismäärämatriisi on

$$\boldsymbol{C}_w = \boldsymbol{C}_{11} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{2.2}$$

Mielivaltaiselle merkkijonolle $w \in \Sigma^*$ luodun esiintymismäärämatriisin $\boldsymbol{C}_w$ rivin a alkiot vastaavat merkkijonoja $aw1, aw2, \ldots, aw\sigma$. Nämä ovat kaikki merkkijonot, jotka voidaan saada laajentamalla merkkijonoa aw yhdellä symbolilla oikealle. Merkkijonorenkaassa merkkijonoa aw voidaan aina laajentaa oikealle vähintään yhdellä symbolilla siten, että syntyvä merkkijono esiintyy renkaassa. Rivi a sisältää kaikki tällaiset symbolit, joten $C_{aw} = \sum_{c \in \Sigma} C_{awc}$.

Havainto liittää esiintymismäärämatriisin rivisummat sellaisten merkkijonojen esiintymismääriin, jotka ovat yhtä symbolia lyhyempiä kuin merkkijonot, joita matriisin alkiot vastaavat. Vastaava pätee myös sarakesummilla, joilla sarakkeen c sarakesumma on $C_{wc} = \sum_{a \in \Sigma} C_{awc}$.

Jatkossa esiintymismäärämatriisin rivi- ja sarakesummavektorit saatetaan tarvittaessa merkitä matriisin yhteyteen seuraavasti:

$$\boldsymbol{C}_w = \left[\begin{array}{cccc|c} C_{1w1} & C_{1w2} & \cdots & C_{1w\sigma} & C_{1w} \\ C_{2w1} & C_{2w2} & \cdots & C_{1w\sigma} & C_{2w} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ C_{\sigma w1} & C_{\sigma w2} & \cdots & C_{\sigma w\sigma} & C_{\sigma w} \\ \hline C_{w1} & C_{w2} & \cdots & C_{w\sigma} & C_w \end{array}\right]. \tag{2.3}$$

Pystysuoran viivan oikealla puolella on matriisin rivisummat. Vaakasuoran viivan alle on merkitty matriisin sarakesummat. Lisäksi osamerkkijonon w esiintymismäärä C_w on merkitty rivisummavektorin alimmaksi alkioksi. Luku C_w on samalla rivisummavektorin alkioiden summa $C_w = \sum_{a \in \Sigma} C_{aw}$ ja sarakesummavektorin alkioiden summa $C_w = \sum_{c \in \Sigma} C_{wc}$.

Tiivistettäessä n symbolin pituista merkkijonoa T, pitää tiivisteen purkajalle välittää kaikkien korkeintaan n symbolin mittaisten osamerkkijonojen esiintymismäärät. Määrät toimitetaan esiintymismäärämatriiseina, jolloin pisimmät osamerkkijonot, joille matriisi muodostetaan ovat pituudeltaan $n - 2$ symbolia. Seuraava lemma 2.1 osoittaa, että riitää välittää vain maksimaalisista osamerkkijonoista muodostetut esiintymismäärämatriisit. Ei-maksimaalisten osamerkkijonojen matriisit eivät lisää informaatiota.

Lemma 2.1. *Olkoon $T \in \Sigma^*$ mielivaltainen syklitön merkkijono, jonka pituus $n = |T| > 2$ ja olkoon merkkijonorengas T_r muodostettu merkkijonosta T. Lisäksi olkoon $w \in \Sigma^*$ osamerkkijono joka esiintyy renkaassa T_r.*

Kaikkien renkaassa T_r esiintyvien, merkkijonoa w lyhyempien, osamerkkijonojen esiintymismäärät ovat sekä tiivistäjän että tiivisteen purkajan tiedossa.

Tällöin osamerkkijonon w laajennosten awc, $a, c \in \Sigma$, esiintymismäärät välittävät ennestään tuntematonta informaatiota merkkijonorenkaasta T_r vain jos osamerkkijono w on maksimaalinen.

Todistus. Jos osamerkkijono w ei ole maksimaalinen, se on joko vain oikealle maksimaalinen, vain vasemmalle maksimaalinen tai ei kumpaakaan. Lemma osoitetaan todeksi käymällä läpi nämä kolme eri tapausta. Kunkin tapauksen kohdalla osoitetaan miten laajennosten awc, $a, c \in \Sigma$, esiintymismäärät voidaan laskea $|w| - 1$ pituisten renkaan T_r osamerkkijonojen avulla.

Olkoon osamerkkijono w oikealle maksimaalinen, mutta ei vasemmalle maksimaalinen. Tällöin siitä muodostettu esiintymismäärämatriisi on seuraavan kaltainen:

$$\boldsymbol{C}_w = \left[\begin{array}{cccc|c} 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ C_{aw1} & C_{aw2} & \cdots & C_{aw\sigma} & C_{aw} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \\ \hline C_{w1} & C_{w2} & \cdots & C_{w\sigma} & C_w \end{array}\right].$$

Matriisin sarake- ja rivisummat vastaavat merkkijonoja, joiden pituus on yhtä symbolia lyhyempi kuin merkkijonojen awc pituus. Nämä summat ovat siten tunnettuja.

Koska osamerkkijono ei ole vasemmalle maksimaalinen, on olemassa vain yksi symboli $a \in \Sigma$ jolla sitä voidaan laajentaa vasemmalle. Tämä symboli tunnistetaan etsimällä ainoa positiivinen rivisumma C_{aw}.

Vain matriisin rivillä a esiintyy positiivisia alkioita. Kaikkien muiden matriisin alkioiden arvo on 0. Siten rivin a alkiot voidaan täydentää matriisin sarakesummien avulla: alkioille pätee $C_{awc} = C_{wc}$ kaikilla symboleilla $c \in \Sigma$.

Toisessa tapauksessa osamerkkijono w on vasemmalle maksimaalinen, muttei oikealle maksimaalinen. Silloin tilanne muistuttaa edellistä, mutta tällä kertaa esiintymismäärämatriisissa

$$\boldsymbol{C}_w = \left[\begin{array}{ccccc|c} 0 & \cdots & C_{1wc} & \cdots & 0 & C_{1w} \\ 0 & \cdots & C_{2wc} & \cdots & 0 & C_{2w} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & C_{\sigma wc} & \cdots & 0 & C_{\sigma w} \\ \hline 0 & \cdots & C_{wc} & \cdots & 0 & C_w \end{array}\right]$$

voi olla vain yksi positiivisia alkioita sisältävä sarake. Riveistä sen sijaan vähintään kahdella on positiivisia lukuja. Sarake c voidaan löytää etsimällä ainoa merkkijono wc joka esiintyy renkaassa T_r. Tämän jälkeen sarakkeen alkiot saadaan matriisin rivisummista kaavan $C_{awc} = C_{aw}$ avulla.

Kolmannessa tapauksessa osamerkkijono w ei ole oikealle eikä vasemmalle maksimaalinen.

Tällöin sen esiintymismäärämatriisissa

$$\boldsymbol{C}_w = \left[\begin{array}{ccccc|c} 0 & \cdots & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & C_{awc} & \cdots & 0 & C_{aw} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & 0 \\ \hline 0 & \cdots & C_{wc} & \cdots & 0 & C_w \end{array}\right]$$

on vain yksi rivi ja yksi sarake, joissa esiintyy positiivisia alkioita. Toisin sanoen vain yksi matriisin alkioista on positiivinen. Tämä voidaan löytää etsimällä ainoat aw ja wc, $a, c \in \Sigma$ muotoiset merkkijonot joilla $C_{aw} > 0$ ja $C_{wc} > 0$. Alkion arvo $C_{awc} = C_{aw} = C_{wc} = C_w$.

Yhtäsuuruudet $C_{aw} = C_{wc} = C_w$ ovat ymmärrettävissä, koska tarkastelemme merkkijonorengasta. Renkaassa mitä tahansa siinä esiintyvää osamerkkijonoa voidaan aina jatkaa vähintään yhdellä symbolilla oikealle ja vasemmalle. Edellä oli määritelty, että kumpaankin suuntaan löytyy vain yksi symboli, jolla osamerkkijonoa w voidaan jatkaa. Siksi osamerkkijonot aw ja wc esiintyvät renkaassa tasan yhtä monta kertaa kuin ydin w.

Kaikissa tapauksissa esiintymismäärämatriisi oli laskettavissa jo tunnettujen osamerkkijonojen avulla. Tämä osoittaa lemman todeksi. □

Yllä esitellyn pohjalta voidaan esitellä Algoritmi 1, joka ottaa syötteenä tiivistettävän tekstin $T \in \Sigma$ jollain äärellisellä aakkostolla Σ. Tekstistä algoritmi tuottaa tiivisteen CSE-menetelmällä. Aluksi tiivisteeseen koodataan tiivistettävän tekstin pituus ja yksittäisten symboleiden esiintymismäärät riveillä 2 ja 4. Sen jälkeen koodataan jokaisen maksimaalisen osamerkkijonon esiintymismäärämatriisit rivillä 10.

Algorithm 1 Merkkijonon T tiivistys osamerkkijonoja luettelemalla.

```
1:  procedure CSE(T)
2:      Print(C_ε)                                   ▷ C_ε = |T|
3:      for all c ∈ Σ do
4:          Print(C_c)
5:      end for
6:      I_T#                          ▷ luo kaksisuuntainen BWT-indeksi merkkijonosta T
7:      M ← EnumerateMaximalSubstrings(I_T#)
8:      for all w ∈ M do
9:          C_w ← CreateCountMatrix(I_T#, w)
10:         Print(C_w)
11:     end for
12: end procedure
```

Algoritmin yleinen rakenne on selvä, mutta yksityiskohdat eivät. Tiivistettävän tekstin kokonaispituus C_ϵ voidaan tallettaa käyttäen Peter Eliaksen luomaa [14] Elias gamma -koodausta. Tällä koodauksella pituuden koodaus vie $2\lfloor \log_2 C_\epsilon \rfloor + 1$ bittiä tilaa. Kun

kokonaispituus C_ϵ tunnetaan, ensimmäisen aakkoston symbolin esiintymismäärä voidaan tallettaa tavanomaisena etumerkittömänä binäärilukuna. Tämä luku vaatii $\lfloor \log_2 C_\epsilon \rfloor + 1$ bittiä tilaa tiivisteestä, sillä sen arvo on maksimissaan C_ϵ. Jokaiselle aakkoston symbolin $i \in \Sigma$ esiintymismäärälle C_i pätee yläraja $\max C_i = C_\epsilon - \sum_{j<i} C_j$. Ylärajan avulla symbolin esiintymismäärä voidaan koodata tiivisteeseen käyttäen $\lfloor \log_2 \max C_i \rfloor + 1$ bittiä tilaa.

Tämän luvun loppuosassa esitellään loput yksityiskohdat, kuten kaksisuuntainen BWT-indeksi, sitä hyödyntävät proseduurit EnumerateMaximalSubstrings ja CreateCountMatrix sekä miten esiintymismäärämatriisit voidaan koodata tiivisteeseen. Lopuksi luvussa 2.5 esitellään miten tiivisteestä voidaan palauttaa alkuperäinen teksti.

2.2 Kaksisuuntainen BWT-indeksi

Yllä on esitetty miten tiivistäminen osamerkkijonoja luettelemalla toimii pääpiirteittäin. Tähän mennessä ei kuitenkaan ole esitelty algoritmeja, joiden avulla tiivistysmenetelmä voitaisiin toteuttaa. Myöhemmin esiteltävät algoritmit tukeutuvat tietorakenteeseen, jota kutsutaan kaksisuuntaiseksi BWT-indeksiksi. Alla esitellään indeksi CSE-menetelmän kannalta oleellisin osin aloittaen ensin Burrows-Wheeler-muunnoksen ja muutaman muun aputietorakenteen määrittelystä.

Michael Burrowsin ja David Wheelerin esittelemä merkkijonon $T = t_1 t_2 \dots t_n \in \Sigma^*$ Burrows-Wheeler-muunnos (engl. *Burrows-Wheeler transform*, BWT) [5] voidaan määritellä kaikkien sen kiertojen avulla. Kierrossa merkkijonon $k \in \mathbb{N}^*$ oikeanpuoleisinta merkkiä siirretään merkkijonon alkuun. Samalla $n - k$ vasemmanpuoleisinta merkkiä siirtyvät k askelta oikealle. Tällöin muodostuu kierto $t_{n-k} t_{n-k+1} \dots t_n t_1 t_2 \dots t_{k-1}$. Merkkijonolla, jossa on n merkkiä, on tasan n eri kiertoa. Esimerkkinä taulukossa 2.1 on listattu merkkijonon $T = \texttt{mississippi\#}$ kaikki 12 kiertoa.

Seuraavaksi kaikki merkkijonon kierrot $t_1 t_2 \dots t_n, t_n t_1 t_2 \dots t_{n-1}, \dots, t_2 \dots t_{n-1} t_n t_1$ järjestetään aakkosjärjestykseen. Taulukkoa, jossa kierrot on aakkosjärjestyksessä, kutsutaan myös BWT-matriisiksi. Matriisin viimeinen sarake. eli kiertojen viimeiset symbolit, muodostaa merkkijonon T Burrows-Wheeler-muunnoksen L. Sarake on korostettu esimerkkitaulukossa 2.1 josta nähdään merkkijonon $T = \texttt{mississippi\#}$ Burrows-Wheeler-muunnoksen olevan $L = \texttt{ipssm\#pissii}$.

Kahden eri kierron välinen järjestys on hyvin määritelty vain jos merkkijono T ei ole syklinen. Esimerkiksi syklisen merkkijonon $T = 1212$ kaikkien kiertojen joukossa on on kaksi eri kiertoa $t_1 t_2 t_3 t_4 = 1212$ ja $t_3 t_4 t_1 t_2 = 1212$ joiden välillä ei ole selvää aakkosjärjestystä.

Ongelma korjataan lisäämällä tekstin T loppuun loppusymboli #. Tämä symboli ei ole mukana tekstin aakkostossa, eli $\# \notin \Sigma$. Lisäksi määritellään, että uusi symboli # edeltää aakkosjärjestyksessä mitä tahansa aakkoston Σ symbolia. Kun Burrows-Wheeler-muunnos muodostetaan merkkijonosta $T\#$, voidaan olla varmoja merkkijonon syklittömyydestä, ja jokaiselle kahdelle eri kierrolle löytyy keskinäinen järjestys.

Alkujaan Paolo Ferraginan ja Giovanni Manzinin esittelemä BWT-indeksi [15] voidaan

#	m	i	s	s	i	s	s	i	p	p	**i**
i	#	m	i	s	s	i	s	s	i	p	**p**
i	p	p	i	#	m	i	s	s	i	s	**s**
i	s	s	i	p	p	i	#	m	i	s	**s**
i	s	s	i	s	s	i	p	p	i	#	**m**
m	i	s	s	i	s	s	i	p	p	i	#
p	i	#	m	i	s	s	i	s	s	i	**p**
p	p	i	#	m	i	s	s	i	s	s	**i**
s	i	p	p	i	#	m	i	s	s	i	**s**
s	i	s	s	i	p	p	i	#	m	i	**s**
s	s	i	p	p	i	#	m	i	s	s	**i**
s	s	i	s	s	i	p	p	i	#	m	**i**

Taulukko 2.1: Merkkijonon $T = \texttt{mississippi\#}$ kaikki kierrot listattuna aakkosjärjestyksessä. Viimeisen sarakkeen korostetut kirjaimet muodostavat Burrows-Wheeler-muunnoksen L_T. Merkkijonon T loppuosat on alleviivattu taulukossa.

toteuttaa hyödyntäen wavelet-puuta (engl. *wavelet tree*) Burrows-Wheeler-muunnoksen tallettamiseen. Tässä työssä ei käydä tietorakennetta sen tarkemmin läpi, mutta alla luetellaan joitain sen perusominaisuuksia.

Roberto Grossin, Ankur Guptan ja Jeffrey Scott Vitterin esittelemä wavelet-puu [20] on tilatiivis tietorakenne, joka muodostetaan jonkin äärellisen aakkoston Σ jollekin merkkijonolle $T = t_1 t_2 \cdots t_n \in \Sigma^*$. Tietorakenne toteuttaa seuraavat neljä operaatiota:

- $\textsc{Rank}(c, i)$ palauttaa kokonaisluvun $k \in \mathbb{N}^*$, joka ilmaisee montako symbolia $c \in \Sigma$ merkkijonossa T on ennen indeksiä $i \in \{1, 2, \ldots, n\}$,
- $\textsc{Select}(c, k)$ palauttaa indeksin $i \in \{1, 2, \ldots, n\}$, jonka kohdalla on k:des symboli $c \in \Sigma$ merkkijonossa T,
- $\textsc{RangeCount}(\langle i, j\rangle, \langle l, r\rangle)$ palauttaa luvun $k \in \mathbb{N}^*$, joka kertoo kuinka monta suljetulle välille $[l, r]$ osuvaa symbolia esiintyy merkkijonossa T välillä $i..j$ ja
- $\textsc{RangeList}(\langle i, j\rangle, \langle l, r\rangle)$ palauttaa aakkostetun listan symboleista $a \in \Sigma$, $l \leq a \leq r$, jotka löytyvät merkkijonosta T välillä $i..j$.

Edellisistä operaatioista RangeCount ja RangeList ovat Veli Mäkisen ja Gonzalo Navarron esittelemiä laajennoksia alkuperäiseen tietorakenteeseen [28].

Oletetaan, että käytössä oleva laskennan malli on RAM-malli ja tietokoneen sanan koko on $w = \Omega(\log_2 n)$. Kun aakkoston koko on σ symbolia, merkkijonolle T muodostettu wavelet-puu vie $n \log_2 \sigma(1 + o(1))$ bittiä tilaa. Operaatioiden Rank, Select ja RangeCount suoritusaika on $O(\log_2 \sigma)$. Operaation RangeList suoritusaika on $O(d \log(\sigma/d))$, missä d on palautetun listan koko. [28]

Toinen tarvittava aputietorakenne on yksinkertainen taulukko $C\,[1 \ldots \sigma]$. Se muodostetaan jonkin äärellisen aakkoston Σ merkkijonolle $T \in \Sigma^*$. Taulukon indeksissä $c \in \{1, 2, \ldots, \sigma\}$ olevan alkion arvo $C\,[c] \in \mathbb{N}^*$ ilmaisee montako symbolia c *aidosti pienempää* symbolia merkkijonossa T on. Huomattavaa on että $C\,[1]$ on aina arvoltaan 0.

Määritelmä 2.7. *Olkoon annettu äärellisen aakkoston $\Sigma = \{1, 2, \ldots \sigma\}$ merkkijono $T \in \Sigma^*$. Merkkijonolle T muodostettu BWT-indeksi on seuraavista aputietorakenteista koostuva tietorakenne:*

- *merkkijonolle $T\#$ muodostettu Burrows-Wheeler-muunnos $L_{T\#}$ joka on talletettu wavelet-puuhun ja*
- *kokonaislukutaulukko $C\,[0..\sigma]$ jonka alkion c arvo $C\,[c]$ kertoo montako symbolia c aidosti pienempää symbolia merkkijonossa $T\#$ esiintyy.*

Burrows-Wheeler-muunnos muodostetaan merkkijonosta $T\#$, jonka aakkosto on laajennettu loppusymbolilla #. Yllä sovittiin, että symbolista # käytetään myös merkintää 0, jolloin aakkostoksi tulee $\sigma + 1$ symbolin kokoinen aakkosto $\Sigma = \{0, 1, \ldots, \sigma\}$. Tämän takia taulukko C sisältää indeksit $0, 1, \ldots, \sigma$. Jatkossa BWT-indeksiä merkitään lyhyesti $\mathcal{I}_{T\#}$ algoritmien parametreissa ja vastaavissa tilanteissa. Myöhemmin samaa merkintää käytetään myös kaksisuuntaisesta BWT-indeksistä. Sekaannuksen vaaraa ei ole, koska tämän osuuden jälkeen työssä käytetään vain kaksisuuntaista indeksiä ja tarvittaessa siitä löytyisi myös BWT-indeksin operaatiot.

BWT-indeksin merkitys selviää tarkastelemalla taulukkoa 2.1. Siinä jokainen rivi alkaa jollain merkkijonon `mississippi#` loppuosalla. Selkeyden vuoksi ne on merkitty alleviivauksella, jolloin huomaa miten kiertojen järjestäminen aakkosjärjestykseen vastaa loppuosien järjestämistä aakkosjärjestykseen. Kutakin riviä vastaava alkio Burrows-Wheeler-muunnoksessa $L = l_1 l_2 \cdots l_n$ on rivin aloittavan loppuosan vasemmanpuoleinen konteksti. Toisin sanoen merkkijonossa T alkion l_i symboli edeltää rivillä i olevaa loppuosaa.

Havainnon avulla voidaan toteuttaa algoritmi 2 joka laskee montako kertaa mielivaltainen merkkijono $m \in \Sigma$ esiintyy merkkijonossa $T\#$. Algoritmi etenee etsittävän merkkijonon m lopusta alkuun etsien ne merkkijonon $T\#$ loppuosat jotka alkavat merkkijonolla m. Tätä varten pidetään yllä indeksiparia $\langle s, e\rangle$ Burrows-Wheeler-muunnokseen $L_{T\#}$. Indeksit rajaavat muunnoksesta alueen, joka vastaa sen hetkistä loppuosaa. Aluksi pari alustetaan arvolla $\langle 1, n\rangle$ joka vastaa tyhjää merkkijonoa ϵ. Jokainen loppuosa alkaa tyhjällä merkkijonolla, joten indeksien rajaama alue vastaa kaikkia loppuosia BWT-matriisissa.

Indeksi i pitää kirjaa, missä kohtaa merkkijonoa m ollaan, ja se aloittaa viimeisestä symbolista m_k. Indeksiparia $\langle s, e\rangle$ päivitetään riveillä 6-7. Operaatio kaventaa indeksien osoittaman alueen tyhjästä merkkijonosta ϵ osoittamaan BWT-matriisin aluetta, jonka sisältämät loppuosat alkavat yhden pituisella osamerkkijonolla m_k.

Esimerkin vuoksi oletetaan että haettava merkkijono $m =$ `si` ja BWT-matriisi on edelleen taulukossa 2.1 esitetty. Symboli m_k on silloin `i`, joka asetetaan muuttujan c arvoksi. Ensimmäisellä kierroksella $C\,[c] = 1$, sillä vain loppusymboli # on pienempi kuin symboli `i`

Algorithm 2 Laske montako kertaa merkkijono $m = m_1 m_2 \cdots m_k$ esiintyy merkkijonossa $T\# = t_1 t_2 \cdots t_n$.

```
1:  procedure CountSubstrings(I_{T#}, m)
2:      i ← k
3:      ⟨s, e⟩ ← ⟨1, n⟩                          ▷ Tyhjä osamerkkijono ϵ
4:      while s < e and i ≥ 1 do
5:          c ← m_i
6:          s ← C[c] + Rank(c, s) +1
7:          e ← C[c] + Rank(c, e + 1)
8:          i ← i − 1
9:      end while
10:     if e < s then
11:         return 0
12:     else
13:         return e − s + 1
14:     end if
15: end procedure
```

merkkijonossa $T\#$. Kutsu $\textsc{Rank}(c, s)$ palauttaa tuloksen 0 koska ennen indeksiä s ei esiinny yhtään symbolia `s`. Alueen ylärajaksi tulee siten $1 + 0 + 1 = 2$. Loppuindeksi e saa uuden arvon 5, koska kutsu $\textsc{Rank}(c, e + 1)$ palauttaa tuloksen 4, joka on symboleiden `i` lukumäärä ennen indeksiä $n + 1$. Näin rajautunut alue $\langle 2, 5\rangle$ vastaa taulukossa 2.1 kaikkia osamerkkijonoja, jotka alkavat symbolilla `i`.

Seuraavalla kierroksella edetään symboliin `s`, josta tulee muuttujan c uusi arvo. Indeksiparin $\langle s, e\rangle$ arvo on edellisen kierroksen jäljiltä $\langle 2, 5\rangle$, ja taulukon C alkion $C[c]$ arvo on 8. Burrows-Wheeler-muunnoksessa L ei esiinny symbolia `s` ennen indeksiä 2, joten operaatiokutsu $\textsc{Rank}(c, s)$ palauttaa arvon 0. Alkuindeksin s uudeksi arvoksi asetetaan $8 + 0 + 1 = 9$.

Ennen kohtaa $e + 1 = 5 + 1 = 6$ muunnoksessa L esiintyy 2 kappaletta symbolia `s`. Tämän johdosta operaatio $\textsc{Rank}(c, e + 1)$ palauttaa arvon 2, ja loppuindeksin e arvoksi tulee $8 + 2 = 10$. Algoritmin 2 viimeisen kierroksen jälkeen Taulukosta 2.1 on rajattu rivit $\langle 9, 10\rangle$. Tämä alue sisältää kaikki loppuosat, jotka alkavat merkkijonolla `si`. Algoritmin 2 lopullinen tulos on $e - s + 1 = 10 - 9 + 1 = 2$, eli tekstissä T esiintyvien osamerkkijonojen $m =$ `si` esiintymismäärä.

Käyttäen merkkijonosta T muodostettua BWT-indeksiä algoritmi 2 löysi loppuosat, jotka alkoivat annetulla merkkijonolla m. Kirjoitetaan merkkijono $T = t_1 t_2 \cdots t_n$ käänteisessä järjestyksessä, jolloin saadaan merkkijono $\underline{T} = t_n t_{n-1} \cdots t_1$. Tämän merkkijonon loppuosat ovat merkkijonon T alkuosia kirjoitettuna käänteisessä järjestyksessä. Muodostetaan merkkijonosta $\underline{T}$ BWT-indeksi, ja otetaan jokin toinen merkkijono $m = m_1 m_2 \cdots m_k$. Myös merkkijono m kirjoitetaan käänteisessä järjestyksessä, jolloin syntyy merkkijono $\underline{m} = m_k m_{k-1} \cdots m_1$. Jos sovelletaan algoritmia 2 BWT-indeksiin $\mathcal{I}_{\underline{T}\#}$ ja merkkijonoon $\underline{m}$ voidaan laskea kuinka monta merkkijonon T alkuosaa loppuu merkkijonolla m.

Thomas Schnattingerin, Enno Ohlebuschin ja Simon Gogin esittelemä [34] *kaksisuuntainen BWT-indeksi* (engl. *bidirectional BWT index*) merkkijonolle $T \in \Sigma^*$ luodaan yhdistämällä kaksi BWT-indeksiä. Ensimmäinen luodaan merkkijonolle T ja toinen sen käänteismerkkijonolle $\underline{T}$. Kaksisuuntaiselle indeksille oleellista on kuusi operaatiota, jotka se toteuttaa. Ne on lueteltu alla Määritelmässä 2.8.

Määritelmän avuksi määritellään funktio $\mathbb{I}(\mathcal{I}_{T\#}, w)$ joka saa syötteenä merkkijonolle T luodun kaksisuuntaisen BWT-indeksin ja merkkijonon w. Funktio palauttaa parin $\langle i, j \rangle$ jossa $i, j \in \mathbb{N}^*$ ja $1 \leq i \leq j \leq n$. Pari rajaa merkkijonon $T\#$ BWT-matriisista rivit, jotka vastaavat merkkijonon $T\#$ merkkijonolla w alkavia loppuosia.

Määritelmä 2.8. *Merkkijonolle T muodostettu kaksisuuntainen BWT-indeksi tarjoaa seuraavat kuusi operaatiota:*

- IsLeftMaximal($\langle i, j \rangle$) *palauttaa totuusarvon* ***true*** *jos ja vain jos osamerkkijono $L_{T\#}[i..j]$ sisältää vähintään kahta eri merkkiä. Muuten paluuarvo on* ***false****.*
- IsRightMaximal($\langle i, j \rangle$) *palauttaa totuusarvon* ***true*** *jos ja vain jos osamerkkijono $L_{\underline{T}\#}[i..j]$ sisältää vähintään kahta eri merkkiä. Muuten paluuarvo on* ***false****.*
- EnumerateLeft($\langle i, j \rangle$) *palauttaa listan erillisistä symboleista osamerkkijonossa $L_{T\#}[i..j]$ järjestettynä aakkosjärjestykseen.*
- EnumerateRight($\langle i, j \rangle$) *palauttaa listan erillisistä symboleista osamerkkijonossa $L_{\underline{T}\#}[i..j]$ järjestettynä aakkosjärjestykseen.*
- ExtendLeft($c, \mathbb{I}(w, T), \mathbb{I}(\underline{w}, \underline{T})$) *palauttaa syötesymbolilla $c \in \{0, 1, \ldots, \sigma\}$ parin $\langle \mathbb{I}(cw, T), \mathbb{I}(\underline{w}c, \underline{T}) \rangle$.*
- ExtendRight($c, \mathbb{I}(w, T), \mathbb{I}(\underline{w}, \underline{T})$) *palauttaa syötesymbolilla $c \in \{0, 1, \ldots, \sigma\}$ parin $\langle \mathbb{I}(wc, T), \mathbb{I}(c\underline{w}, \underline{T}) \rangle$.*

Operaatio IsLeftMaximal vastaa suoraan aiemmin määriteltyä vasemmalle maksimaalisuuden käsitettä: olkoon $\langle i, j \rangle = \mathbb{I}(\mathcal{I}_{T\#}, w)$, jollain osamerkkijonolla w. Indeksien rajaama osa Burrows-Wheeler-muunnoksesta $L_{T\#}$ vastaa osamerkkijonojen w vasemmanpuoleista kontekstia merkkijonossa $T\#$. Mikäli kontekstissa on vain yhtä symbolia, voidaan osamerkkijonoja w jatkaa vain yhdellä symbolilla, eivätkä ne siten ole vasemmalle maksimaalisia. Teknisesti määritelmä toimisi myös vaikka rajattu alue ei vastaisi yhtä osamerkkijonoa w; tässä työssä ei operaatiota kuitenkaan käytetä muunlaisessa yhteydessä.

Symmetrisesti operaatiolla IsRightMaximal voidaan selvittää onko osamerkkijono w oikealle maksimaalinen merkkijonossa $T\#$. Kumpikin operaatio vastaa wavelet-puun operaatiota RangeCount ja voidaan siten suorittaa ajassa $O(\log_2 \sigma)$.

Operaatio EnumerateLeft listaa kaikki symbolit joilla annettua osamerkkijonoa $w \in \Sigma^*$ voidaan jatkaa vasemmalle. Olkoon $\langle i, j \rangle = \mathbb{I}(\mathcal{I}_{T\#}, w)$ merkkijonoa w vastaavat rivit Burrows-Wheeler-muunnoksessa $L_{T\#}$. Operaatio toteutetaan kutsumalla wavelet-puun operaatiota RangeList($\langle i, j \rangle, \langle 0, \sigma \rangle$). Operaatio palauttaa suoraan halutun listan, koska

symbolit on rajattu suljetulle välille $[0, \sigma]$, eli koko aakkosto hyväksytään. Vastaava operaatio EnumerateRight toteutetaan samalla tavalla, mutta käyttäen Burrows-Wheeler-muunnosta $L_{\underline{T}\#}$. Operaatioiden aikavaativuus on RangeList-operaation aikavaativuus $O(d \log(\sigma/d))$.

Operaatio ExtendLeft ottaa syötteenään osamerkkijonoa w vastaavat rajaukset kaksisuuntaisessa BWT-indeksissä sekä symbolin $c \in \Sigma$. Se palauttaa uudet rajaukset $\langle \mathbb{I}(cw, T), \mathbb{I}(\underline{w}c, \underline{T}) \rangle$ jotka vastaavat osamerkkijonoa cw tekstissä $T\#$.

Osamerkkijonon laajentaminen yhdellä merkillä vasemmalle onnistuu algoritmissa 2 esitetyllä idealla. Algoritmia suoritetaan yhden kierroksen verran indeksille $\mathcal{I}_{T\#}$, jolloin osamerkkijono w laajenee yhdellä merkillä vasemmalle. Tämä vastaa tulosparin alkiota $\mathbb{I}(cw, T)$. Tuloksen toista puolta ei saada samalla algoritmilla, koska siinä osamerkkijono laajenee oikealle.

Merkitään parilla $\langle i, j \rangle = \mathbb{I}(w, T)$, ja yllä laskettua tulosta $\mathbb{I}(cw, T) = \langle i', j' \rangle$. Toinen tulosparia olkoon $\langle p', q' \rangle = \mathbb{I}(\underline{w}c, \underline{T})$, ja syöte $\mathbb{I}(\underline{w}, \underline{T}) = \langle p, q \rangle$. Parin $\langle p, q \rangle$ rajaama väli sisältää kaikki loppuosat $\underline{w}$ tekstistä $\underline{T}$. Vastaavasti väli sisältää kaikki alkuosat w tekstistä T, ja seurauksena myös kaikki alkuosat cw ovat jossain tällä välillä. Waveletpuun operaatio RangeCount$(\langle i, j \rangle, \langle 0, c-1 \rangle)$ palauttaa luvun k, joka ilmaisee moniko välillä $\langle i, j \rangle$ oleva loppuosa alkaa symbolia c pienemmällä symbolilla. Tämän avulla voidaan laskea $p' = p + k$.

Yllä indeksi i' vastaa loppuosan cw alkua tekstin $T\#$ BWT-matriisissa, ja j' näiden loppukohtaa. Vastaavasti indeksit p' ja q' rajaavat loppuosan $\underline{w}c$ tekstin $\underline{T}\#$ BWT-matriisissa. Rajattujen alueiden pitää olla saman kokoiset, minkä ansiosta saadaan laskettua loppupää $q' = p' + j' - i'$.

Yksi kierros algoritmin 2 suoritusta vaatii kaksi Rank-operaatiota, joiden yhdistettu aikavaativuus on $O(\log_2 \sigma)$. Myös operaatio RangeCount onnistuu samassa yhteisaikavaativuudessa $O(\log_2 \sigma)$, joten operaatioiden ExtendLeft ja ExtendRight aikavaativuus on $O(\log_2 \sigma)$.

2.3 Osamerkkijonojen luettelu

Tässä luvussa hyödynnetään edellä esiteltyä kaksisuuntaista BWT-indeksiä. Tietorakenteen avulla toteutetaan tehokkaasti tekstin T maksimaalisten osamerkkijonojen luetteleminen, ja maksimaalisen osamerkkijonon w esiintymismäärämatriisin $\boldsymbol{C}_w$ alkioiden laskeminen.

Maksimaalisten osamerkkijonojen tuottaminen. Tiivistysalgoritmin ytimessä on algoritmi, joka luettelee tiivistettävän tekstin $T\#$ maksimaaliset osamerkkijonot aakkosjärjestyksessä. Algoritmin 3 proseduuri EnumerateMaximalSubstrings esittelee tavan toteuttaa luettelu hyödyntäen kaksisuuntaista BWT-indeksiä.

Proseduuri saa syötteenään vain merkkijonosta $w = T\#$ tuotetun kaksisuuntaisen BWT-indeksin. Tuloksenaan proseduuri palauttaa listan kaikista maksimaalisista osamerkkijo-

Algorithm 3 Syötemerkkijonon T maksimaaliset osamerkkijonot v aakkosjärjestyksessä

```
1:  procedure EnumerateMaximalSubstrings(I_{T#})
2:      n ← |T#|
3:      I ← ∅
4:      R ← {⟨⟨1, n⟩, 0⟩}                          ▷ tyhjä merkkijono ϵ pituudella 0
5:      while not IsEmpty(R) do
6:          ⟨⟨i, j⟩, ℓ⟩ ← Pop(R)
7:          if ℓ ≤ n − 2 then
8:              R ← EnumerateRight(⟨i, j⟩)
9:              for all c ∈ R do
10:                 ⟨i′, j′⟩ ← ExtendRight(c, ⟨i, j⟩)
11:                 if IsRightMaximal(⟨i′, j′⟩) then
12:                     Push(R, ⟨⟨i′, j′⟩, ℓ + 1⟩)
13:                 end if
14:             end for
15:         end if
16:         if IsLeftMaximal(⟨i, j⟩) then
17:             I = I ∪ {⟨i, j⟩}
18:         end if
19:     end while
20:     return I
21: end procedure
```

noista aakkosjärjestyksessä.

Rivillä 4 alustetaan oikealle maksimaalisten osamerkkijonojen jono tyhjällä merkkijonolla ϵ. Tiivistettävässä merkkijonossa w täytyy esiintyä vähintään kahta eri symbolia, jotta se ei olisi toisteinen. Seurauksena tyhjä merkkijono on aina oikealle maksimaalinen.

Jokaisella kierroksella jonosta käsitellään etummaisin osamerkkijono v, kunnes jono on tyhjä, ja palautetaan löydetyt maksimaaliset osamerkkijonot. Riveillä 7-15 osamerkkijonoa v laajennetaan oikealle kaikilla symboleilla, joilla niin voi tehdä. Mikäli syntyvä osamerkkijono on oikealle maksimaalinen, se lisätään jonoon $\mathcal{R}$.

Suorituksen aikana jonoon $\mathcal{R}$ tullaan lisäämään vuorollaan kaikki oikealle maksimaaliset osamerkkijonot, joiden pituus on korkeintaan $n - 2$ symbolia. Osamerkkijonot lisätään aiemmin määritellyssä aakkosjärjestyksessä, koska ensin laajennetaan 0 symbolin pituiset osamerkkijonot 1 symbolin pituisiksi osamerkkijonoiksi, jotka lisätään jonon loppuun. Käsiteltäessä k symbolin pituista osamerkkijonoa, siitä voi syntyä pituudeltaan $k + 1$ osamerkkijonoja, jotka asetetaan jonoon kaikkien k symbolia pitkien tai lyhyempien osamerkkijonojen perään. Operaation EnumerateRight tuottama lista symboleista on aakkosjärjestyksessä, minkä seurauksena samanpituiset osamerkkijonot lisätään jonoon keskenään aakkosjärjestyksessä.

Rivillä 16 testataan onko osamerkkijono v myös vasemmalle maksimaalinen. Mikäli näin

on, se on maksimaalinen ja voidaan lisätä jonoon $\mathcal{I}$.

Algoritmin 3 tuloslista $\mathcal{I}$ saattaa viedä paljon tilaa, koska kaikki maksimaaliset osamerkkijonot talletetaan siihen. Koko lista on kuitenkin tosiasiallisesti turha, sillä aakkosjärjestyksessä luodun listan alkiot voidaan käsitellä sitä mukaa, kun ne luodaan. Kuten aiemmin on esitetty, maksimaalisesta osamerkkijonosta v luodaan esiintymismäärämatriisi C_{avc} kaikilla symboleilla $a, c \in \Sigma \cup \{\#\}$. Tämä matriisi talletetaan tuotettavaan tiivisteeseen. Talletuksen jälkeen osamerkkijonoa v tai matriisia $\boldsymbol{C}_v$ ei enää tarvita. Toisin sanoen, maksimaalisen merkkijonon lisäys jonoon $\mathcal{I}$ rivillä 17 voidaan korvata kutsulla proseduuriin, joka käsittelee maksimaalisen osamerkkijonon.

Algoritmin yhdelle kierrokselle voidaan laskea pahimman tapauksen aikavaativuus huomaamalla, että pahimmillaan käsiteltävä osamerkkijono $\langle i, j \rangle$ voidaan jatkaa oikealle kaikilla aakkoston merkeillä:

Lemma 2.2. *Algoritmin 3 yhden kierroksen aikavaativuus on pahimmassa tapauksessa* $O(\sigma \log \sigma)$.

Todistus. Kierroksella suoritetaan yksi EnumerateRight-operaatio, jonka aikavaativuus on $O(\sigma \log(\sigma/\sigma))$, jossa d on tulosjoukon koko. Siten tämän operaation aikavaativuus ei ole koskaan huonompi kuin $O(\sigma \log \sigma)$.

Pahimmillaan osamerkkijonoa $\langle i, j \rangle$ voidaan jatkaa oikealle kaikilla aakkoston symboleilla, joita on σ kappaletta. Silloin ExtendRight ja IsRightMaximal operaatiot suoritetaan σ kertaa. Yhteensä tämä onnistuu $O(\sigma \log \sigma)$ ajassa.

Lopuksi, aikavaativuudeltaan $O(\log \sigma)$, operaatio IsLeftMaximal suoritetaan kerran.

Edelliset yhdistäen saadaan aikavaativuudeksi $O(\sigma \log \sigma)$. □

Koko algoritmin 3 aikavaativuudelle voidaan muodostaa yläraja seuraavalla teoreemalla 2.3.

Lemma 2.3. *Kun syötemerkkijonon T pituus on n symbolia, algoritmin 3 aikavaativuus on korkeintaan* $O(n\sigma \log \sigma)$.

Todistus. Oikealle maksimaalisten osamerkkijonojen lukumäärä on korkeintaan $n - 1$ [2], joten algoritmin 3 silmukka suoritetaan korkeintaan $n - 1$ kertaa. Yhdistämällä edellinen lemmaan 2.2 saadaan suoritusajalle yläraja $O(n\sigma \log \sigma)$. □

Osamerkkijonojen esiintymismäärien laskeminen. Osamerkkijonojen esiintymismäärien laskeminen voidaan toteuttaa tehokkaasti kaksisuuntaiselle Burrows-Wheeler-indeksillä. Sho Kanai kanssakirjoittajineen [24] ovat kehittäneet menetelmän esiintymismäärien laskemiseen binäärimerkkijonoista Burrows-Wheeler-matriisin avulla. Algoritmi 4 laajentaa menetelmää äärellisen aakkoston merkkijonoille, ja hyödyntää kaksisuuntaista BWT-indeksiä.

Algoritmi aloittaa luomalla esiintymismäärämatriisin $\boldsymbol{C}_w$, jonka kaikkien alkioiden arvo on 0. Seuraavaksi matriisiin täytetään esiintymismäärät kaikkiin alkioihin, jotka vastaavat merkkijonorenkaasta löytyvää osamerkkijonoa. Täyttäminen tapahtuu riveillä 4-12,

Algorithm 4 Esiintymismäärämatriisin luominen osamerkkijonolle w.

```
1: procedure CreateCountMatrix(I_{T#}, w)
2:     C_w ← 0                                   ▷ luo σ × σ nollamatriisi
3:     ⟨i, j⟩ ← I(I_{T#}, w)
4:     for all a ∈ EnumerateLeft(⟨i, j⟩) do
5:         ⟨i′, j′⟩ ← ExtendLeft(a, ⟨i, j⟩)
6:         for all c ∈ EnumerateRight(⟨i′, j′⟩) do
7:             ⟨i″, j″⟩ ← ExtendRight(c, ⟨i′, j′⟩)
8:             if j″ ≥ i″ then
9:                 C_awc ← j″ − i″ + 1
10:            end if
11:        end for
12:    end for
13:    return C_w
14: end procedure
```

joissa syötemerkkijono w ensin laajennetaan kaikilla symboleilla $a \in \Sigma$ vasemmalle, joilla laajentaminen on mahdollista. Seuraavaksi kaikki aw muotoiset osamerkkijonot laajennetaan yhdellä symbolilla oikealle. Jälleen käydään läpi kaikki symbolit $c \in \Sigma$, joilla merkkijono awc löytyy merkkijonorenkaasta.

Rivillä 9 lasketaan osamerkkijonon awc esiintymismäärät ja talletetaan luku matriisiin. Esiintymismäärät saadaan laskettua indeksien avulla, sillä ne rajaavat yhtenäisen alueen BWT-matriisista, joka sisältää kaikki loppuosat awc.

Esiintymismäärämatriisin luominen vaatii pahimmassa tapauksessa matriisin jokaisen alkion läpikäynnin. Lemma 2.4 osoittaa algoritmin aikavaativuuden tässä tapauksessa.

Lemma 2.4. *Algoritmin 4 pahimman tapauksen aikavaativuus on* $O(\sigma^2 \log \sigma)$.

Todistus. Lemman 2.2 todistuksessa osoitettiin, että operaatiot EnumerateLeft ja EnumerateRight vievät korkeintaan $O(\sigma \log \sigma)$ aikaa.

Pahimmassa tapauksessa kummankin operaation palauttaama symbolijoukko on kooltaan σ. Tällöin esiintymismäärämatriisin jokainen alkio tulee saamaan positiivisen arvon.

Operaatio ExtendLeft algoritmin 4 rivillä 5 suoritetaan σ kertaa. Sen sijaan operaatio ExtendRight rivillä 7 suoritetaan σ^2 kertaa. Luvussa 2.2 selvitettiin, että operaatioiden ExtendLeft ja ExtendRight aikavaativuus on $O(\log \sigma)$, jolloin koko algoritmin pahimman tapauksen aikavaativuus on $O(\sigma^2 \log \sigma)$. □

Tyypillisesti tiivistettävän tekstin pituus n on paljon suurempi kuin aakkoston koko σ. Esimerkki tyypillisestä aakkostosta on 8-bittinen tavuaakkosto, jolloin $\sigma = 256$. Sen sijaan pituus n voi olla satoja tai tuhansia miljoonia symboleita. Lemman 2.3 osoitti kaikkien maksimaalisten osamerkkijonojen etsinnän vievän $O(n\sigma \log \sigma)$ aikaa, eli käytännössä aikavaativuus on lineaarinen suhteessa tekstin pituuteen.

$$\begin{bmatrix} C_{1w1} & \cdots & C_{1wc} & \cdots & C_{1w\sigma} & C_{1w} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ C_{aw1} & \cdots & C_{awc} & \cdots & C_{aw\sigma} & C_{aw} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ C_{\sigma w1} & \cdots & C_{\sigma wc} & \cdots & C_{\sigma w\sigma} & C_{\sigma w} \\ C_{w1} & \cdots & C_{wc} & \cdots & C_{w\sigma} & C_{w} \end{bmatrix}$$

Kuva 2.2: Esiintymismäärätaulukossa on esiintymismäärämatriisin $\boldsymbol{C}_w$ lisäksi matriisin rivi- ja sarakesummat. Taulukkoon on merkitty alkiot, jotka tunnetaan käsiteltäessä alkiota C_{awc}.

Koska oikealle maksimaalisten osamerkkijonojen lukumäärä on korkeintaan $n-1$, myös maksimaalisia osamerkkijonoja on korkeintaan tämän verran. Lemman 2.4 perusteella voidaan arvioida, että kaikkien esiintymismäärämatriisien koodaamiseen kuluvaan aikaa tekstin n pituus vaikuttaa vain lineaarisesti. Yksittäisen matriisin kohdalla maksimiaika $O(\sigma^2 \log \sigma)$ on hillitty, koska edes kerroin σ^2 ei tavanomaisilla aakkostojen koilla kasva suureksi.

2.4 Esiintymismäärämatriisin koodaaminen

Tuotetut maksimaalisten osamerkkijonojen esiintymismäärämatriisit pitää koodata tiivisteeseen. Naiivilla koodauksella jokainen matriisin alkio vaatisi $\lfloor \log_2 n \rfloor + 1$, jossa n on merkkijonon T pituus, bittiä tilaa, kun aakkoston koko on σ symbolia. Seuraavaksi esitellään kaksi erilaista tapaa koodata matriisi käyttäen vähemmän bittejä. Kumpikin menetelmä hyödyntää seuraavassa esitettyä laajennosta esiintymismäärämatriisista $\boldsymbol{C}_w$.

Esiintymismäärätaulukko 2.2 laajentaa esiintymismäärämatriisin $\sigma + 1 \times \sigma + 1$ kokoiseksi taulukoksi. Taulukon oikeanpuoleisin sarake sisältää matriisin $\boldsymbol{C}_w$ rivisummat, ja alin rivi matriisin $\boldsymbol{C}_w$ sarakesummat.

Ensimmäinen tapa tallettaa matriisin alkiot järjestyksessä vasemmalta oikealle ja ylhäältä alas. Se ei kuitenkaan käytä jokaiselle alkiolle $\lfloor \log_2 n \rfloor + 1$ bittiä tilaa. Sen sijaan esiintymismäärälle C_{awc} lasketaan alaraja l_{awc} ja yläraja u_{awc}, joiden avulla se talletetaan potentiaalisesti pienempään tilaan. Tässä esitetyt ala- ja ylärajat ovat alunperin Ken-ichi Iwatan ja Mitsuharu Arimuran esittelemiä [23].

Tiivistysalgoritmilla on käytössään alkuperäinen merkkijonorengas T_r, joten se tietää taulukon kaikkien alkioiden arvon. Tiivistettyä tekstiä purettaessa alkion C_{awc} kohdalla tunnetaan vain merkkijonoa awc lyhyempien tai aakkosjärjestyksessä pienempien merkkijonojen esiintymismäärät. Nämä tunnetut esiintymismäärät on rajattu taulukossa 2.2 harmaalla taustalla. Rajattua aluetta voidaan siten hyödyntää etsittäessä ala- ja ylärajoja esiintymismäärälle C_{awc}.

Tarkasteltaessa tiettyä alkiota C_{awc} taulukko 2.2 voidaan jakaa 16 alueeseen kuten taulu-

$$\begin{bmatrix} S_{11} & S_{12} & S_{13} & S_{1.} \\ S_{21} & C_{awc} & S_{23} & C_{aw} \\ S_{31} & S_{32} & S_{33} & S_{3.} \\ S_{.1} & C_{wc} & S_{.3} & C_{w} \end{bmatrix}$$

Kuva 2.3: Esiintymismäärätaulukon 2.2 alkiot yhdistettynä suuremmiksi alueiksi suhteessa alkioon C_{awc}.

kossa 2.3. Alueiden arvot lasketaan taulukon 2.2 alkioista seuraavilla kaavoilla:

$$S_{11} = \sum_{\substack{c(<a)\in\Sigma,\\ d(<b)\in\Sigma}} C_{cwd} \qquad S_{12} = \sum_{c(<a)\in\Sigma} C_{cwb} \qquad S_{13} = \sum_{\substack{c(<a)\in\Sigma,\\ e(>b)\in\Sigma}} C_{cwe}$$

$$S_{21} = \sum_{d(<b)\in\Sigma} C_{awd} \qquad S_{23} = \sum_{e(>b)\in\Sigma} C_{awe}$$

$$S_{31} = \sum_{\substack{f(>a)\in\Sigma,\\ d(<b)\in\Sigma}} C_{fwb} \qquad S_{32} = \sum_{f(>a)\in\Sigma} C_{fwb} \qquad S_{33} = \sum_{\substack{f(>a)\in\Sigma,\\ e(>b)\in\Sigma}} C_{fwe}$$

Kuten alkuperäisessä taulukossa 2.2, myös taulukon 2.3 oikeanpuoleisin sarake sisältää rivien alkioiden kokonaissummat. Kaavoina esitettynä:

$$S_{1.} = S_{11} + S_{12} + S_{13}, \qquad C_{aw} = S_{21} + C_{awb} + S_{23}, \qquad S_{3.} = S_{31} + S_{32} + S_{33}.$$

Taulukon alin rivi sisältää kaikkien sarakkeiden alkioiden kokonaissummat:

$$S_{.1} = S_{11} + S_{21} + S_{23}, \qquad C_{wb} = S_{12} + C_{awb} + S_{32}, \qquad S_{.3} = S_{13} + S_{23} + S_{33}.$$

Taulukossa 2.3 harmaalla taustalla rajatut alkiot ovat purkualgoritmin tiedossa, kun sen pitää päätellä esiintymismäärä C_{awc}. Siksi näitä rajattuja alkioita voidaan käyttää laskemaan ala- ja ylärajoja merkkijonon esiintymismäärälle. Niin tekevät seuraavaksi esiteltävät rajat.

Olkoon $S_{23} = \sum_{d>c} C_{awd}$, eli taulukossa 2.2 kaikkien alkiota C_{awc} samalla rivillä seuraavien alkioiden summa. Tämän summan avulla alkion C_{awc} arvo voidaan kirjoittaa

$$C_{awc} = C_{aw} - S_{21} - S_{23}. \tag{2.4}$$

Käsitellessään alkiota C_{awc}, purkualgoritmi ei tunne alkion S_{23} arvoa, joten se ei voi käyttää kaavaa 2.4. Sen sijaan taulukon 2.3 alkion $S_{.3}$ arvo on tunnettu, ja se sisältää myös kaikki summan S_{23} alkiot. Havainnon avulla saadaan alkion C_{awc} arvolle alaraja

$$C_{awc} \geq C_{aw} - S_{21} - S_{.3}. \tag{2.5}$$

Jos edellinen alaraja on positiivinen, täytyy loput jäljellä olevasta rivisummasta C_{aw} olla alkiossa C_{awc}.

Vastaavasti saadaan toinen alaraja

$$C_{awc} \geq C_{wb} - S_{12} - S_{3.}. \tag{2.6}$$

Kun edellisen rajan kohdalla tarkasteltiin riviä, jolla alkio C_{awc} sijaitsee, niin tämän rajan kohdalla sama tarkastelu tehdään sarakkeelle, jossa alkio on.

Nämä voidaan yhdistää ottamalla niistä suurin alaraja, jolloin lopulliseksi rajaksi tulee

$$l_{awc} = \max\left(0, C_{aw} - S_{21} - S_{\cdot 3}, C_{wb} - S_{12} - S_{3\cdot}\right). \tag{2.7}$$

Mikäli kummankin kaavan 2.5 ja 2.6 tulos on ei-positiivinen, eli ne eivät onnistu kaventamaan alarajaa, kaava 2.7 käyttää triviaalia alarajaa 0.

Ensimmäisessä ylärajassa hyödynnetään sitä, että alkion C_{awc} kohdalla rivin a rivisumma on tunnettu. Samoin kaikkien samalla rivillä sitä edeltävien alkioiden yhteissumma S_{21} tiedetään. Näiden erotus

$$C_{awc} \leq C_{aw} - S_{21} \tag{2.8}$$

on yläraja, koska $C_{awc} + S_{23} = C_{aw} - S_{21}$.

Toinen yläraja vastaa ensimmäistä, mutta nyt tarkastellaan rivin sijaan saraketta, jolla alkio C_{awc} sijaitsee:

$$C_{awc} \leq C_{wb} - S_{12}. \tag{2.9}$$

Edellisistä ylärajoista 2.8 ja 2.9 valitaan pienin, jolloin saadaan yläraja

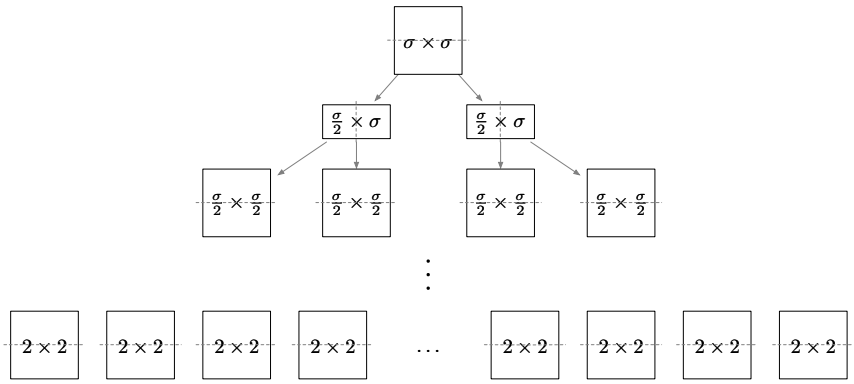
$$u_{awc} = \min\left(C_{aw} - S_{21}, C_{wb} - S_{12}\right). \tag{2.10}$$

Merkitään luvun C_{awc} ylä- ja alarajojen erotusta

$$N_{awc} = u_{awc} - l_{awc}.$$

Jos erotus $N_{awc} = 0$, osamerkkijonon esiintymismäärä tunnetaan edellisten osamerkkijonojen esiintymismäärien perusteella. Sitä ei tarvitse silloin koodata tiivisteeseen lainkaan. Muussa tapauksessa ala- ja yläraja rajaavat mahdollisten lukujen määrää ja riittää koodata kuinka mones luku C_{awc} on suljetulla välillä $[l_{awc}, u_{awc}]$. Tähän riittää $\lfloor \log_2 N_{awc} \rfloor + 1$ bittiä.

Esiintymismäärämatriisin koodaus puolittamalla. Seuraavassa esitetään rekursiivinen tapa tallettaa matriisi $\boldsymbol{C}_w$ käyttäen $(\sigma - 1)^2$ lukua, kun rivi- ja sarakesummat tunnetaan. Lopuksi selitetään, miten jokaiselle näistä luvuista voidaan laskea ala- ja yläraja. Rajojen avulla luvut voidaan koodata pienempään tilaan, vastaavasti kuin edellisessä menetelmässä. Esityksen yksinkertaisuuden vuoksi seuraavassa selityksessä oletetaan, että aakkoston koko $\sigma = 2^k$, jollain kokonaisluvulla $k \in \mathbb{N}^+$. Tekniikka on mahdollista toteuttaa myös matriiseille, joiden koko ei ole kahden potenssi.



Kuva 2.4: Kooltaan $\sigma \times \sigma$ esiintymismäärämatriisi $\boldsymbol{C}_w$ jaettuna rekursiivisesti vuorotellen vaaka- ja pystysuunnassa. Alimatriiseihin on merkitty matriisin koko.

Jaetaan matriisi $\boldsymbol{C}_w$ vaakasuunnassa kahteen saman suuruiseen osamatriisiin seuraavasti:

$$\boldsymbol{C}_w = \begin{bmatrix} C_{1w1} & C_{1w2} & \cdots & C_{1w\sigma} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\frac{\sigma}{2}w1} & C_{\frac{\sigma}{2}w2} & \cdots & C_{\frac{\sigma}{2}w\sigma} \\ \cdots & \cdots & \cdots & \cdots \\ C_{\frac{\sigma}{2}+1w1} & C_{\frac{\sigma}{2}+1w2} & \cdots & C_{\frac{\sigma}{2}+1w\sigma} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\sigma w1} & C_{\sigma w2} & \cdots & C_{\sigma w\sigma} \end{bmatrix}.$$

Merkitään matriisin ylemmästä puolikkaasta syntyvää uutta matriisia kirjaimella $\boldsymbol{A}$ ja alemmasta puolikkaasta syntyvää matriisia kirjaimella $\boldsymbol{B}$. Matriisin $\boldsymbol{A}$ sarakesummat voidaan luetella vektorina $\vec{s}_{\boldsymbol{A}}$. Matriisin $\boldsymbol{B}$ sarakesummia ei tarvitse erikseen tallettaa, koska ne voidaan laskea alkuperäisen matriisin $\boldsymbol{C}$ sarakesummien $\vec{s}_{\boldsymbol{C}}$ ja matriisin $\boldsymbol{A}$ sarakesummien $\vec{s}_{\boldsymbol{A}}$ avulla, eli

$$\vec{s}_{\boldsymbol{B}} = \vec{s}_{\boldsymbol{C}} - \vec{s}_{\boldsymbol{A}}. \tag{2.11}$$

Puolittamista voidaan jatkaa rekursiivisesti matriiseissa $\boldsymbol{A}$ ja $\boldsymbol{B}$, kunnes puolituksen tuloksena syntyneissä matriiseissa on vain yksi rivi. Puolitusprosessia voidaan kuvata binääripuuna. Jokainen puun solmu v vastaa yhtä puolitusta. Solmun vasemmanpuoleinen lapsi v_l vastaa puolituksessa syntynyttä ylempää alimatriisia $\boldsymbol{A}$, ja oikeanpuoleinen lapsi v_r alempaa alimatriisia $\boldsymbol{B}$. Solmuun v talletetaan matriisin $\boldsymbol{A}$ sarakesummavektori. Puun lehdet vastaavat jakoja, joissa 2 riviset matriisit jaetaan 1 rivisiksi matriiseiksi.

Alkuperäinen matriisi $\boldsymbol{C}_w$ voidaan palauttaa yllä luodusta binääripuusta ja entuudestaan tunnetusta matriisin sarakesummavektorista. Puun juuri vastaa ensimmäistä matriisin $\boldsymbol{C}_w$ jakoa kahteen osaan. Kun kaikista solmuista on luettu sarakesummavektorit, on käytössä $\sigma - 1$ sarakesummavektoria ja alkuperäinen, koko esiintymismäärämatriisin, sarakesummavektori.

Sekä tiivistys- että purkualgoritmi tuntevat matriisista $\boldsymbol{C}_w$ sarakesummien lisäksi myös rivisummat. Tätä voidaan hyödyntää jättämällä jokaisesta sarakesummavektorista viimei-

nen alkio tallettamatta. Oletetaan, että ollaan jakamassa $n \times \sigma$ kokoista matriisia, jonka rivisummat ovat $r_1, \ldots, r_n$. Ylemmän jaossa syntyneen matriisin $\boldsymbol{A}$ rivit olkoon $1, \ldots, m$, jossa $m = n/2$. Tämän alimatriisin rivisummien kokonaissumma $r = \sum_{i=1}^{m} r_i$ on kaikkien matriisin $\boldsymbol{A}$ alkioiden yhteissumma. Matriisin $\boldsymbol{A}$ viimeisen sarakkeen sarakesumma s_σ voidaan kirjoittaa kokonaisrivisumman r avulla seuraavasti:

$$s_\sigma = r - \sum_{i=1}^{\sigma-1} s_i.$$

Vaakasuoran jaon lisäksi matriisi voidaan jakaa myös pystysuoraan. Seuraavassa matriisin $\boldsymbol{C}_w$ vaakasuorassa jaossa syntynyt matriisi $\boldsymbol{A}$ on jaettu pystysuoraan matriiseiksi $\boldsymbol{E}$ ja $\boldsymbol{F}$:

$$\boldsymbol{A} = \left[\begin{array}{ccc:ccc} C_{1w1} & \cdots & C_{1w\frac{\sigma}{2}} & C_{1w\frac{\sigma}{2}+1} & \cdots & C_{1w\sigma} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ C_{\frac{\sigma}{2}w1} & \cdots & C_{\frac{\sigma}{2}w\frac{\sigma}{2}} & C_{\frac{\sigma}{2}w\frac{\sigma}{2}+1} & \cdots & C_{\frac{\sigma}{2}w\sigma} \end{array}\right].$$

Vasemmanpuoleinen puolikas on alimatriisi $\boldsymbol{E}$ ja oikeanpuoleinen alimatriisi $\boldsymbol{F}$.

Matriisille $\boldsymbol{E}$ voidaan laskea uudet rivisummat vastaavasti, kuin laskettiin matriisin $\boldsymbol{A}$ sarakesummat. Lisäksi jako voidaan kuvata binääripuun solmuna, jonka arvoksi talletetaan matriisin $\boldsymbol{E}$ rivisummat. Edelleen rivisummista voidaan jättää viimeinen alkio tallettamatta, koska se voidaan laskea sarakesummien ja muiden rivisummien avulla.

Vaaka- ja pystysuoraan jakoa voidaan vuorotella. Tällöin syntyvän binääripuun ensimmäisen tason solmu, eli juurisolmu, vastaa ensimmäistä vaakasuoraa jakoa. Toisen tason solmut vastaavat pystysuoria jakoja, ja kolmannen tason solmut taas vaakasuoria jakoja. Kun alkuperäisen matriisin koko on $\sigma \times \sigma$, juurisolmun jaossa syntyvät matriisit ovat kooltaan $\sigma/2 \times \sigma$, ja toisen tason jaoissa syntyvät matriisit $\sigma/2 \times \sigma/2$ kokoisia. Vuorottelevaa jakoa kuvaavat algoritmi 5 ja algoritmi 6.

Algoritmeja käyttäen $\sigma \times \sigma$ kokoisesta matriisista $\boldsymbol{C}_w$ voidaan muodostaa binääripuu kutsumalla algoritmia 5. Rivisummavektorina $\vec{r}$ ja sarakesummavektorina $\vec{s}$ käytetään matriisin rivi- ja sarakesummia. Algoritmit kutsuvat rekursiivisesti toisiaan, kunnes juurisolmuja vastaavat jaot jakavat 2×2 kokoisia alimatriiseja kahtia. Näistä jaoista muodostuvat puun lehtisolmut.

Syntynyt puu voidaan koodata tiivisteeseen käymällä puu läpi syvyyssuuntaisella läpikäynnillä ja kirjoittamalla kunkin solmun jakovektori, alkio kerrallaan, tulosteeseen. Jokaisen solmun lapset käsitellään aina samassa järjestyksessä, jatkossa vasen lapsi ensin. Tällöin solmujen järjestys tunnetaan implisiittisesti, kun puu luetaan takaisin purkualgoritmin syötteestä.

Jotkin jaot voivat tuottaa summavektorin, jonka kaikki alkiot ovat nollia. Esimerkiksi jos vaakasuuntaisen jaon tuloksena on nollavektori, tiedetään, että ylemmän alimatriisin kaikkien alkioiden summa on 0. Koska esiintymismäärämatriisien alkiot ovat aina epänegatiivisia, tiedetään tällöin, että kyseinen alimatriisi on nollamatriisi — eli sen jokaisen alkion arvo on 0. Käytetään tätä seuraavaksi hyväksi tekniikassa, joka saattaa vähentää esiintymismäärämatriisin koodaukseen tarvittavaa tilaa.

Oletetaan, että $n \times m$ kokoista alimatriisia $\boldsymbol{M}$ ollaan jakamassa vaakasuunnassa puoliksi. Merkitään matriisin rivien indeksejä luvuilla $1, 2, \ldots, n$. Rivejä vastaava rivisummavektori $\vec{r} = [r_1, r_2, \cdots, r_n]$ on tunnettu jaon aikana. Jaossa syntyvä ylempi alimatriisi $\boldsymbol{A}$ käsittää rivit $1, \ldots, k$ ja alempi alimatriisi $\boldsymbol{B}$ rivit $k+1, \ldots, n$. Olkoon jaon tuloksena syntyvä alimatriisin $\boldsymbol{A}$ sarakesummavektori nollavektori $\vec{s}_{\boldsymbol{A}} = [0, 0, \ldots, 0]$. Tällöin alimatriisi $\boldsymbol{A}$ on nollamatriisi, ja myös sitä vastaavat rivisummat $r_1, r_2, \ldots, r_k$ ovat arvoltaan 0.

Myös tiivisteen purkaja tuntee rivisummat $r_1, r_2, \ldots, r_k$, kun se käsittelee yllä mainittua alimatriisin $\boldsymbol{M}$ jakoa. Siten jaossa syntynyt sarakesummavektori $\vec{s}_{\boldsymbol{A}}$ ei sisällä yhtään uutta informaatiota, ja vektorin, eli jakoa vastaavan solmun, koodaaminen tulosteeseen on turhaa. Esiintymismäärämatriisia koodatessa kyseiset solmut jätetään koodaamatta.

Optimointi voi säästää huomattavasti tilaa mikäli esiintymismäärämatriisi on harva, eli sisältää paljon alkioita, joiden arvo on 0. Pelkkä harvuus ei kuitenkaan riitä, vaan 0 alkioiden pitää osua sellaisiin kohtiin, että jaossa syntyy nolla-alimatriiseja.

Algorithm 5 Matriisin $\boldsymbol{M}$ jako puoliksi vaakasuunnassa

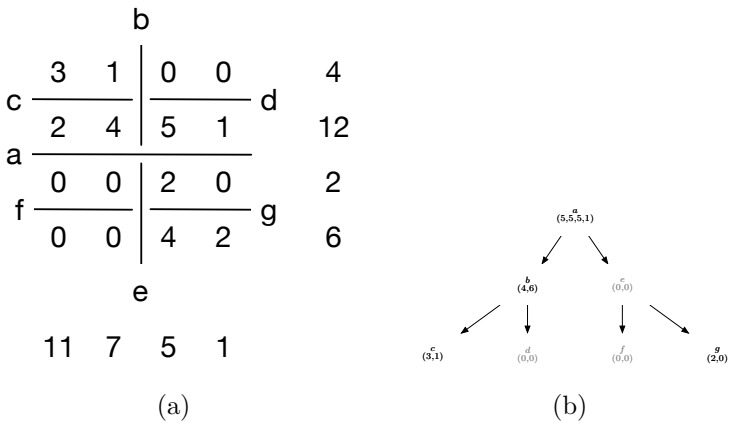
```
1:  procedure DivideHorizontal(M, r, s)
2:      n_rows ← NumRows(M)
3:      n_columns ← NumColumns(M)
4:      r_m ← n_rows/2                          ▷ Ylemmän puolikkaan viimeinen rivi
5:      i ← 1
6:      while i ≤ n_columns do                  ▷ Laske matriisien A ja B sarakesummat
7:          s^A[i] ← Σ_{j=1}^{r_m} M_ji
8:          s^B[i] ← s[i] − s^A[i]
9:          i ← i + 1
10:     end while
11:     w ← s^A[1 ... n_columns − 1]
12:     if n_rows = 2 and n_columns = 2 then
13:         return Node(w, ∅, ∅)
14:     else
15:         A ← M[1 ... r_m][1 ... n_columns]
16:         B ← M[r_m + 1 ... n_rows][1 ... n_columns]
17:         t_l ← DivideVertical(A, r[1 ... r_m], s^A)
18:         t_r ← DivideVertical(B, r[r_m + 1 ... n_rows], s^B)
19:         return Node(w, t_l, t_r)
20:     end if
21: end procedure
```

Esimerkki matriisin koodaamisesta tulosteeseen on kuvassa 2.5. Esimerkissä aakkoston koko on $\sigma = 2^2 = 4$ merkkiä. Koodattavan matriisin $\boldsymbol{C}$ rivisummat $\vec{r}_{\boldsymbol{C}} = [4, 12, 2, 6]$ ja sarakesummat $\vec{s}_{\boldsymbol{C}} = [11, 7, 5, 1]$ ovat sekä tiivistäjän että tiivisteen purkajan tiedossa. Kuvassa oikealla on puu, joka syntyy, kun proseduuria DivideHorizontal kutsutaan matriisille $\boldsymbol{C}$ ja vektoreille $\vec{r}$ ja $\vec{s}$.

Vuorottelevien proseduurien DivideHorizontal ja DivideVertical tekemät jaot on

Algorithm 6 Matriisin $\boldsymbol{M}$ jako puoliksi pystysuunnassa

procedure DIVIDEVERTICAL($\boldsymbol{M}, \vec{r}, \vec{s}$)
  $n_{\text{rows}} \leftarrow$ NUMROWS($\boldsymbol{M}$)
  $n_{\text{columns}} \leftarrow$ NUMCOLUMNS($\boldsymbol{M}$)
  $c_m \leftarrow n_{\text{columns}}/2$ ▷ Vasemman puolikkaan viimeinen sarake
  $i \leftarrow 1$
  while $i \leq n_{\text{rows}}$ **do** ▷ Laske matriisien $\boldsymbol{A}$ ja $\boldsymbol{B}$ rivisummat
    $\vec{r}^{A}[i] \leftarrow \sum_{j=1}^{c_m} \boldsymbol{M}_{ij}$
    $\vec{r}^{B}[i] \leftarrow \vec{r}[i] - \vec{r}^{A}[i]$
    $i \leftarrow i + 1$
  end while
  $w \leftarrow \vec{r}^{A}[1 \ldots n_{\text{rows}} - 1]$
  if $n_{\text{rows}} = 2$ **and** $n_{\text{columns}} = 2$ **then**
    return NODE($w, \emptyset, \emptyset$)
  else
    $\boldsymbol{A} \leftarrow \boldsymbol{M}[1 \ldots n_{\text{rows}}][1 \ldots c_m]$
    $\boldsymbol{B} \leftarrow \boldsymbol{M}[1 \ldots n_{\text{rows}}][c_m + 1 \ldots n_{\text{columns}}]$
    $t_l \leftarrow$ DIVIDEHORIZONTAL($\boldsymbol{A}, \vec{r}^{A}, \vec{s}[1 \ldots c_m]$)
    $t_r \leftarrow$ DIVIDEHORIZONTAL($\boldsymbol{B}, \vec{r}^{B}, \vec{s}[c_m + 1 \ldots n_{\text{columns}}]$)
    return NODE(w, t_l, t_r)
  end if
end procedure



(a) (b)

Kuva 2.5: Esimerkki puolitettavasta matriisista (a) ja puolituksia vastaavasta binääripuusta (b). Puuhun on harmaalla merkitty solmut joita ei talleteta tiivisteeseen.

merkitty janoin kuvan 2.5 matriisiin. Samoin vastaavat solmut puussa on nimetty jaon mukaan. Todellisuudessa jakoja ei tarvitse erikseen yhdistää summavektoriin, koska jakojen järjestys on ennalta tunnettu.

Selvyyden vuoksi jokaisessa kuvan 2.5 puun solmussa on täydellinen, kyseisessä jaossa syntynyt, rivi- tai sarakesummavektori. Ne siis sisältävät myös viimeisen luvun, jota ei tarvitse välittää tiivisteen purkajalle. Puussa on myös solmut, joita vastaavissa jaoissa syntyvä ylempi matriisin puolikas (vaakasuora jako) tai vasemmanpuoleinen matriisinpuolikas (pystysuora jako) ovat nollamatriiseja. Kuvassa ne ovat harmaita muistuttamassa ettei niitä tarvitse koodata tiivisteeseen.

Kuvan 2.5 puussa harmaat solmut d, e ja f vastaavat nollasummavektorin tuottaneita jakoja. Huomattavaa on, että vaikka solmua e ei tulla liittämään tulosteeseen, sillä on oikeanpuoleinen lapsisolmu g, jonka summavektori pitää koodata tulosteeseen. Sen sijaan solmun e vasemmasta lapsesta f alkava alipuu edustaa nollamatriisin jakoa osiin, eikä mitään tämän alipuun solmua tulla koodaamaan tulosteeseen.

Naiivisti koodattuna jokainen vektorialkio vaatii $\lfloor \log_2 n \rfloor + 1$ bittiä tilaa, missä n on tiivistettävän merkkijonon T pituus. Aiemmin esitetyn suoraviivaisen matriisin koodauksen tapaan tässäkin voidaan hyödyntää ala- ja ylärajoja vähentämään kunkin alkion vaatimaa tilaa.

Luettaessa tiivisteestä (ja koodattaessa tiivisteeseen) summavektorin $\vec{x}$ alkioita, tunnetaan kaikki tätä edeltävät solmut polulla puun juuresta solmuun $\vec{x}$. Seuraavassa ala- ja ylärajat lasketaan hyödyntäen solmun $\vec{x}$ vanhempaa $\vec{p}$ ja isovanhempaa $\vec{g}$. Näiden kolmen summavektorin välinen suhde on kuvattu kuvassa 2.6.

Summavektori $\vec{x}$ saattaa vastata vanhempansa $\vec{p}$ vasemman- tai oikeanpuoleista lasta. Kuvaan 2.6 on piirretty kumpikin tilanne siten, että $\vec{x}$ vastaa vasemmanpuoleista ja $\vec{y}$ oikeanpuoleista lasta. Vektorin $\vec{y}$ jakaman matriisin rivisummat ovat summavektorissa $\vec{p}'$. Tämä on laskettu kaavan 2.11 avulla summavektorista $\vec{p}$ ja sen isovanhemmasta, kuten ylempänä on selitetty.

$$
\begin{array}{ccccccccccccc}
 & & & & & & \vec{p} & & & & & & \\
 & & & & & & p_1 & & & & & & \\
 & & & & & & p_2 & & & & & & \\
\vec{x} & x_1 & x_2 & x_3 & x_4 & & y_1 & y_2 & y_3 & y_4 & \vec{y} & & \\
 & & & & & & p_3 & & & & & & \\
 & & & & & & p_4 & & & & & & \\
\vec{g} & g_1 & g_2 & g_3 & g_4 & & g_5 & g_6 & g_7 & g_8 & & &
\end{array}
$$

Kuva 2.6: Kirjoitettaessa (tai luettaessa) vektoria $\vec{x}$ tunnetaan sen vanhempi $\vec{p}$ ja isovanhempi $\vec{g}$. Vektori $\vec{y}$ vastaa tilannetta, jossa vektori $\vec{x}$ on vanhempansa oikeanpuoleinen lapsi.

Jakopuun juurella ei ole puussa vanhempaa eikä isovanhempaa. Vanhempana voidaan kuitenkin käyttää alkuperäisen esiintymismäärämatriisin $\boldsymbol{C}_w$ rivisummavektoria. Vastaavasti isovanhempana toimii matriisin sarakesummavektori. Ongelma toistuu puun toisen tason solmujen kanssa. Niille vanhempana toimii puun juuri, ja isovanhempana käytetään matriisin $\boldsymbol{C}_w$ rivisummavektoria.

Kuvassa 2.6 vektori $\vec{x}$ jakaa alimatriisin $\boldsymbol{C}_w$ ylempään alimatriisin $\boldsymbol{A}$ ja alempaan alimatriisiin $\boldsymbol{B}$. Summavektorin $\vec{p}$ alkioiden yhteissumma on alimatriisin $\boldsymbol{C}_w$ alkioiden yhteissumma. Edelleen vektorin $\vec{p}$ ensimmäisen puolikkaan yhteissumma $A_{\text{total}} = \sum_{i=1}^{|\vec{p}|/2}$ vastaa ylemmän alimatriisin $\boldsymbol{A}$ alkioiden yhteissummaa, joka on myös summavektorin $\vec{x}$ alkioiden yhteissumma.

Eräs alaraja alkion x_i arvolle on

$$x_i \geq A_{\text{total}} - \sum_{j=1}^{i-1} x_j - \sum_{j=i+1}^{n_x} g_j \tag{2.12}$$

Tämä vastaa alkiot suoraan tallettavan menetelmän alarajaa 2.5. Purettaessa, vektorin alkion i kohdalla, tunnetaan kaikki sitä edeltävät alkiot $x_1, \ldots, x_{i-1}$, mutta ei alkiota i tai sitä seuraavia alkioita. Alaraja approksimoi tuntemattomia alkoita käyttämällä isovanhemman vastaavissa kohdissa olevia alkioita. Nämä ovat vähintään yhtä suuria kuin vektorin $\vec{x}$ alkiot.

Alarajaa 2.12 vastaava alaraja vanhemman oikeanpuoleiselle lapselle on

$$y_i \geq A'_{\text{total}} - \sum_{j=1}^{i-1} y_j - \sum_{j=i+1}^{n_x} g_{n_p/2+j}. \tag{2.13}$$

Yksinkertaisin yläraja alkion x_i arvolle on isovanhempaa vastaavan summavektorin $\vec{g}$ arvo samassa kohdassa. Eli aina pätee, että

$$x_i \leq g_i. \tag{2.14}$$

Vastaavasti oikeanpuoleiselle lapselle pätee

$$y_i \leq g_{n_p/2+i}. \tag{2.15}$$

Toinen yläraja perustuu siihen, että matriisin $\boldsymbol{A}$ kokonaissumma ja vektorin $\vec{x}$ edeltävät alkiot on tunnettuja. Tiedot yhdistämällä saadaan epäyhtälö

$$x_i \leq A_{\text{total}} - \sum_{j=1}^{i-1} x_j, \tag{2.16}$$

tai oikeanpuoleiselle lapselle

$$y_i \leq A'_{\text{total}} - \sum_{j=1}^{i-1} y_j. \tag{2.17}$$

Ylärajoista käytetään parasta, eli pienintä:

$$u_i^x = \min\left(g_i, A_{\text{total}} - \sum_{j=1}^{i-1} x_j\right). \tag{2.18}$$

Tai oikeanpuoleiselle lapselle vastaavaa

$$u_i^y = \min\left(g_{n_p/2+i}, A'_{\text{total}} - \sum_{j=1}^{i-1} y_j\right). \tag{2.19}$$

Esiintymismäärämatriisi $\boldsymbol{C}_w$ voidaan lukea tiivisteestä algoritmin 7 avulla, proseduurikutsulla ReadMatrixHorizontal($\boldsymbol{C}_w, \langle 1, \sigma\rangle, \langle 1, \sigma\rangle, \vec{r}, \vec{s}$). Kutsussa $\boldsymbol{C}_w$ on $\sigma \times \sigma$ kokoinen matriisi, jonka kaikki alkiot on alustettu arvolla 0. Pari $\langle r_t, r_b\rangle$ ilmoittaa jakoa vastaavan matriisin ylimmän ja alimman rivin koordinaatit alussa annetussa $\sigma \times \sigma$-kokoisessa matriisissa. Ensimmäisessä jaossa parin arvo on $\langle 1, \sigma\rangle$, eli kaikki σ riviä. Vastaavasti pari $\langle c_l, c_r\rangle$ ilmoittaa vasemman- ja oikeanpuoleisimman sarakkeen, ja saa alkuarvon $\langle 1, \sigma\rangle$. Vektorit $\vec{r}$ ja $\vec{s}$ ovat esiintymismäärämatriisin $\boldsymbol{C}_w$ rivi- ja sarakesummavektorit.

Proseduuri ReadMatrixHorizontal ja sen tekemät rekursiiviset kutsut proseduureihin ReadMatrixVertical ja ReadMatrixHorizontal täydentävät tiivisteestä matriisiin $\boldsymbol{C}_w$ kaikki positiiviset alkiot.

Proseduurin ReadMatrixHorizontal riveillä 2-4 lasketaan jaetun matriisin tai alimatriisin koko, sekä ylemmän alimatriisin viimeinen rivi r_m. Seuraavaksi lasketaan jaetun matriisin kaikkien alkioiden summa rivisummavektorin $\vec{r}$ avulla. Mikäli matriisi on nollamatriisi, ei ole muuta tekemistä, ja rekursion tämä haara päättyy.

Riveillä 9-15 jakoa vastaava sarakevektori luetaan tiivisteestä, pois lukien viimeinen alkio, joka lasketaan ylemmän alimatriisin kokonaissumman $c_{\boldsymbol{A}}$ ja aiempien sarakevektorin arvojen avulla.

Rivillä 17 tutkitaan onko jaetun matriisin korkeus 2 riviä. Mikäli näin on, kyseessä on jakopuun lehtisolmu, ja kaikki 4×4 alimatriisin alkiot voidaan täydentää. Muussa tapauksessa kutsutaan rekursiivisesti proseduuria ReadMatrixVertical kummallekin alimatriisille. Riveillä 27-30 sarakesummavektori, eli jakovektori, käännetään vastaamaan jaon alempaa alimatriisia.

Algoritmin 8 proseduuri ReadMatrixVertical on lähes identtinen proseduurin ReadMatrixHorizontal kanssa. Rivi- ja sarakesummavektorit vaihtavat asemaansa samalla, kun jakosuunta vaihtuu vaakasuunnasta pystysuuntaan.

Algorithm 7 Matriisin $\boldsymbol{M}$ luku rekursiivisesti syötteestä, vaakasuunta.

procedure ReadMatrixHorizontal($\boldsymbol{M}, \langle r_t, r_b \rangle, \langle c_l, c_r \rangle, \vec{r}, \vec{s}$)
  $n_{\text{rows}} \leftarrow r_b - r_t + 1$
  $n_{\text{columns}} \leftarrow c_r - c_l + 1$
  $r_m \leftarrow r_t + n_{\text{rows}}/2$
  $c \leftarrow \sum_{i=1}^{n_{\text{rows}}} \vec{r}[i]$
  if $c = 0$ **then** ▷ alimatriisi on nollamatriisi
    return
  end if
  $\vec{s}_m$ ▷ n_{columns} alkion pituinen jakovektori
  $i \leftarrow 1$
  while $i \leq n_{\text{columns}} - 1$ **do**
    $\vec{s}_m[i] \leftarrow$ ReadNumber()
    $i \leftarrow i + 1$
  end while
  $c_{\boldsymbol{A}} \leftarrow \sum_{i=1}^{r_m} r[i]$
  $\vec{s}_m[n_{\text{columns}}] \leftarrow c_{\boldsymbol{A}} - \sum_{i=1}^{n_{\text{columns}}-1} \vec{s}_m[i]$
  if $n_{\text{rows}} = 2$ **then**
    $i \leftarrow 1$
    while $i \leq n_{\text{columns}}$ **do**
      $\boldsymbol{M}_{r_t,i} \leftarrow \vec{s}_m[i]$
      $\boldsymbol{M}_{r_b,i} \leftarrow \vec{s}[i] - \vec{s}_m[i]$
      $i \leftarrow i + 1$
    end while
  else
    ReadMatrixVertical($\boldsymbol{M}, \langle r_t, r_m \rangle, \langle c_l, c_r \rangle, \vec{r}, \vec{s}_m$)
    $i \leftarrow 1$
    while $i \leq n_{\text{columns}}$ **do**
      $\vec{s}_m[i] \leftarrow \vec{s}[i] - \vec{s}_m[i]$
      $i \leftarrow i + 1$
    end while
    ReadMatrixVertical($\boldsymbol{M}, \langle r_{m+1}, r_b \rangle, \langle c_l, c_r \rangle, \vec{r}, \vec{s}_m$)
  end if
end procedure

Algorithm 8 Matriisin $\boldsymbol{M}$ luku rekursiivisesti syötteestä, pystysuunta.

procedure ReadMatrixVertical($\boldsymbol{M}, \langle r_t, r_b \rangle, \langle c_l, c_r \rangle, \vec{r}, \vec{s}$)
  $n_{\text{rows}} \leftarrow r_b - r_t + 1$
  $n_{\text{columns}} \leftarrow c_r - c_l + 1$
  $s_m \leftarrow s_l + n_{\text{columns}}/2$
  $c \leftarrow \sum_{i=1}^{n_{\text{columns}}} \vec{c}[i]$
  if $c = 0$ **then** ▷ alimatriisi on nollamatriisi
    return
  end if
  $\vec{r}_m$ ▷ n_{rows} alkion pituinen jakovektori
  $i \leftarrow 1$
  while $i \leq n_{\text{rows}} - 1$ **do**
    $\vec{r}_m[i] \leftarrow$ ReadNumber()
    $i \leftarrow i + 1$
  end while
  $c_{\boldsymbol{A}} \leftarrow \sum_{i=1}^{c_m} \vec{c}[i]$
  $\vec{r}_m[n_{\text{rows}}] \leftarrow c_{\boldsymbol{A}} - \sum_{i=1}^{n_{\text{rows}}-1} \vec{r}_m[i]$
  if $n_{\text{columns}} = 2$ **then**
    $i \leftarrow 1$
    while $i \leq n_{\text{rows}}$ **do**
      $\boldsymbol{M}_{i,c_l} \leftarrow \vec{r}_m[i]$
      $\boldsymbol{M}_{i,c_r} \leftarrow \vec{r}[i] - \vec{r}_m[i]$
      $i \leftarrow i + 1$
    end while
  else
    ReadMatrixHorizontal($\boldsymbol{M}, \langle r_t, r_b \rangle, \langle c_l, c_m \rangle, \vec{r}_m, \vec{s}$)
    $i \leftarrow 1$
    while $i \leq n_{\text{rows}}$ **do**
      $\vec{r}_m[i] \leftarrow \vec{r}[i] - \vec{r}_m[i]$
      $i \leftarrow i + 1$
    end while
    ReadMatrixHorizontal($\boldsymbol{M}, \langle r_t, r_b \rangle, \langle c_{m+1}, c_r \rangle, \vec{r}_m, \vec{s}$)
  end if
end procedure

2.5 Tiivisteen purkaminen

Edellä esitelty CSE-menetelmä tuottaa alkuperäisestä tekstistä $T \in \Sigma$ tiivisteen, joka koostuu seuraavista osista:

- tekstin T pituus C_ϵ,
- aakkoston symboleiden $\#, 1, 2, \ldots, \sigma$ esiintymismäärät $C_\#, C_1, C_2, \ldots, C_\sigma$ ja
- esiintymismäärämatriisi $\boldsymbol{C}_w$ kaikille merkkijonorenkaan T_r maksimaalisille osamerkkijonoille w, joiden pituus on pienempi kuin $C_\epsilon - 2$, aakkosjärjestyksessä.

Luvussa esitetään miten tiivisteestä voidaan luoda tiivis loppuosaverkko, joka vastaa alkuperäistä merkkijonorengasta T_r. Verkko on versio loppuosatrie-puusta (engl. *suffix trie*), mutta tavanomaisen merkkijonon sijaan siihen talletetaan merkkijonorengas. Toisin sanoen verkko esittää joukon osamerkkijonoja, jotka voivat olla äärettömän pitkiä, koska merkkijonorenkaassa osamerkkijonoa voidaan aina jatkaa loputtomiin. Syntynyt verkko on kuitenkin äärellisen kokoinen.

Alla esiteltävän tiivin loppuosaverkon on alunperin esitellyt Danny Dubé ja Vincent Beaudoin binääriselle aakkostolle [12]. Samoin välivaiheena esiteltävä ääretön osamerkkijonopuu on esitelty samassa artikkelissa. Tiivisteen purun suorittava Algoritmi 9 on esitelty binääriaakkosille Sho Kanain ja kanssakirjoittajien artikkelissa [24]. Tässä luvussa algoritmi laajennetaan käsittelemään äärellisen aakkoston merkkijonoja. Lopuksi kerrotaan miten alkuperäinen teksti T voidaan lukea luodusta tiiviistä loppuosaverkosta.

Ääretön osamerkkijonopuu. Olkoon olemassa jokin äärellinen aakkosto Σ ja merkkijonorengas $T_r \in \Sigma^*$. Renkaalle muodostettu *ääretön osamerkkijonopuu nollaesiintymin* (engl. *infinite substring tree with null counts*) sisältää tasan yhden solmun n_w jokaista merkkijonoa $w \in \Sigma^*$ kohden; solmuja on siten äärettömän monta. Jokaisella solmulla on tasan σ lasta, ja suunnatut kaaret solmusta sen lapsiin on nimetty aakkoston Σ symbolein $\{1, 2, \ldots, \sigma\}$. Puun juuri n_ϵ on ainoa solmu jolla ei ole vanhempaa ja se vastaa tyhjää merkkijonoa ϵ.

Kuljettaessa puun juuresta n_ϵ johonkin toiseen solmuun v_w, polun

$$n_\epsilon \xrightarrow{w_1} n_a \xrightarrow{w_2} n_b \xrightarrow{w_3} n_c \cdots n_d \xrightarrow{w_k} n_w$$

muodostavien kaarten nimet voidaan yhdistää merkkijonoksi $w = w_1 w_2 \cdots w_k$. Ensimmäisen kuljetun kaaren nimi on merkkijonon ensimmäinen merkki, toisena kuljetun kaaren nimi on toinen merkki ja niin edelleen. Lopulta viimeisen kaaren nimi on merkkijonon w viimeinen merkki. Solmu johon polkua pitkin päädyttiin vastaa merkkijonoa w ja solmua merkitään n_w.

Jokaiselle solmulle on määritelty paino joka vastaa solmua vastaavan merkkijonon esiintymismäärää merkkijonorenkaassa T_r. Eli juurisolmun n_ϵ paino on C_ϵ ja mielivaltaista

merkkijonoa w vastaavan solmun n_w paino on C_w. Suurin osa joukon Σ^* merkkijonoista ei esiinny renkaassa T_r kertaakaan. Siten suurimmalla osalla solmuista paino on 0.

Seuraava lemma 2.5 osoittaa yllä määritellyn puun olevan ehyt siinä suhteessa että jokainen polku juuresta solmuun, jonka paino on positiivinen, sisältää vain positiivispainoisia solmuja.

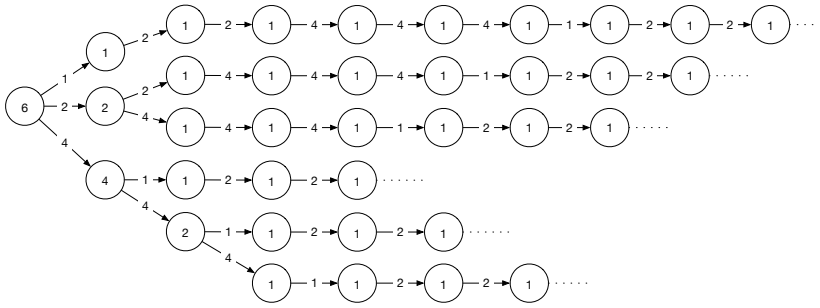
Lemma 2.5. *Olkoon edellä määritellyssä puussa polku* $n_\epsilon \xrightarrow{w_1} n_1 \xrightarrow{w_2} n_2 \cdots n_{k-1} \xrightarrow{w_k} n_w$ *juuresta mielivaltaiseen solmuun* n_w*, jonka paino* C_w *on positiivinen.*

Jokaisen tällä polulla sijaitsevan solmun $n_\epsilon, n_1, n_2, \ldots, n_{k-1}$ *paino on positiivinen.*

Todistus. Määritelmän mukaan merkkijonorengas voidaan muodostaa vain epätyhjästä merkkijonosta. Siten solmun n_ϵ paino on positiivinen.

Oletetaan että polulla on solmu n_v jonka paino on 0. Koska on olemassa polku $n_\epsilon \rightsquigarrow n_v \rightsquigarrow n_w$ merkkijono v on merkkijonon w aito etuosa. Tällöin merkkijono w löytyy merkkijonorenkaasta T_r vähintään kerran mutta sen etuosa v ei kertaakaan. Tämä on mahdotonta, joten vastaoletus on väärä, ja lemma on todistettu oikeaksi. □

Koska lemma 2.5 pätee voidaan puusta jättää pois kaikki solmut joiden paino on 0, eli solmut jotka vastaavat merkkijonoja joita ei esiinny renkaassa T_r. Niiden poistaminen jättää jäljelle ehyen puun. Kutsutaan tällaista puuta *äärettömäksi osamerkkijonopuuksi* (engl. *infinite substring tree*). Kuvassa 2.7 on muodostettu ääretön osamerkkijonopuu tekstin $T = 224441$ merkkijonorenkaasta T_r.



Kuva 2.7: Ääretön osamerkkijonopuu muodostettuna merkkijonorenkaasta $T_r = 224441$. Jokainen haara menee toiselle kierrokselle, joka alkaa $22\cdots$. Näiden merkkien jälkeen haarat jatkuvat äärettömyyteen toistaen renkaan T_r sisältö.

Ääretön osamerkkijonopuu on edelleen ääretön, koska jos osamerkkijono v löytyy merkkijonorenkaasta T_r sitä voidaan aina jatkaa vähintään yhdellä merkillä siten että uusikin merkkijono esiintyy renkaassa. Lopulta puun kaikki haarat vain toistavat renkaan T_r sisältöä loputtomiin.

Tiivis osamerkkijonoverkko. *Tiivis osamerkkijonoverkko* (engl. *compacted substring tree*) on suunnattu verkko, joka on äärellinen, isomorfinen esitysmuoto äärettömästä osamerkkijonopuusta. Tässä tapauksessa isomorfisuus tarkoittaa että äärettömän osamerkkijonopuun solmuista on kuvaus tiiviin osamerkkijonoverkon solmuille, joka säilyttää solmujen

väliset yhteydet, solmujen painot ja kaarien nimet solmujen välisillä poluilla. Verkko on lähes syklitön: ainoan syklin muodostaa polku joka vastaa merkkijonorengasta T_r. Ainoa solmu johon ei saavu lainkaan kaaria on verkon juuri n_ϵ, ja se vastaa tyhjää osamerkkijonoa.

Olkoon osamerkkijonot $w \in \Sigma^*$ ja $aw' \in \Sigma^*$, $a \in \Sigma$, sellaiset, että $w = w'$ ja $C_w = C_{aw'} > 0$. Tällöin osamerkkijono w esiintyy merkkijonorenkaassa T_r vain siten, että sitä aina edeltää symboli a. Tiiviin osamerkkijonoverkon kontekstissa osamerkkijonoja w ja w' kutsutaan isomorfisiksi. Verkossa tulee olemaan solmu n_w, joka vastaa osamerkkijonoa w. Sen sijaan isomorfiselle osamerkkijonolle w' ei luoda solmua, vaan kaari solmuun n_w. Kaaren nimeksi tulee symboli a.

Tiivis osamerkkijonoverkko voidaan muodostaa CSE-tiivistysalgoritmin tulosteesta algoritmin 9 avulla. Algoritmi lisää vuorollaan kaikkien vasemmalle maksimaalisten osamerkkijonojen $w \in \Sigma^*$ laajennokset awc, $a, c \in \Sigma$, verkkoon, jos ne esiintyvät renkaassa T_r.

Tässä ja seuraavissa algoritmeissa käytetään selkeyden vuoksi esiintymismäärämatriisin $\boldsymbol{C}_w$ laajennettua muotoa, kuten yhtälön 2.3 matriisissa, joka sisältää myös rivi- ja sarakesummat. Symboli w on mielivaltainen, ei välttämättä maksimaalinen, osamerkkijono merkkijonorenkaassa T_r. Ohjelmakoodissa esiintymismäärämatriisi voidaan toteuttaa $\sigma + 1 \times \sigma + 1$ kokoisena taulukkona etumerkittömiä kokonaislukuja.

Algoritmin 9 proseduurin DecodeCST syöte on tiivistysalgoritmin 1 tuottama tiiviste S. Lopputuloksena, kun koko tiiviste on luettu, on muodostettu tiivis osamerkkijonoverkko, joka vastaa alkuperäistä tiivistettyä merkkijonorengasta T_r. Purkualgoritmi aloittaa riveillä 2-12 ensin luomalla juurisolmun n_ϵ, jonka paino luetaan tiivisteestä. Seuraavaksi tiivisteestä luetaan jokaisen aakkoston symbolin esiintymismäärät ja luodaan symbolia vastaavat solmut juurisolmun n_ϵ lapsiksi. Solmu luodaan vain siinä tapauksessa että symboli ylipäänsä esiintyy merkkijonorenkaassa. Huomattavaa on miten proseduurilla CreateSuffixLink luodaan solmuille loppuosakaari (engl. *suffix link*) juurisolmuun. Vastaisuudessa tällainen kaari luodaan kaikille solmuille n_{aw}, jossa $a \in \Sigma$ ja $w \in \Sigma^*$. Kaari osoittaa solmusta n_{aw} solmuun n_w, eli merkkijonoon josta puuttuu ensimmäinen symboli a.

Algoritmin 9 rivillä 13 alustetaan w_queue-jono yhdellä solmulistalla. Tämä lista sisältää kaikki verkon alustuksessa luodut n_a, $a \in \Sigma$, solmut. Kyseinen lista kokonaisuudessaan vastaa maksimaalista osamerkkijonoa ϵ. Listan alkiot ovat solmuja, jotka laajentavat osamerkkijonoa yhdellä symbolilla vasemmalle, eli muotoa $n_{a\epsilon}$, $a \in \Sigma$.

Yleisemmin jonon w_queue alkiot vastaavat aina jotain osamerkkijonoa $v \in \Sigma^*$. Alkio on lista joka koostuu solmuista $n_{i_1 v}, n_{i_2 v}, \ldots, n_{i_k v}$ jossa kukin $i_j \in \Sigma$ ja $i_1 < i_2 < i_j < i_k$. Myöhemmin luotaessa osamerkkijonolle v esiintymismäärämatriisia $\boldsymbol{C}_v$ tarvitaan kaikki verkosta löytyvät n_{av} muotoiset solmut, koska nämä vastaavat matriisin rivisummia. Näiden solmujen tunteminen nopeuttaa algoritmia, koska muuten esimerkiksi solmu $n_{i_1 v}$ jouduttaisiin etsimään verkosta kulkemalla polku $n_\epsilon \rightsquigarrow n_{i_1 v}$. Tämä, solmua vastaavan osamerkkijonon pituuteen nähden lineaarisaikainen, kulku jouduttaisiin toistamaan jokaiselle solmulle $n_{i_j v}$. Nämä solmut ovat kuitenkin tiedossa silloin kun osamerkkijono v lisätään jonoon ja siten niistä koostuva lista on helppo lisätä jonoon pelkän solmun n_v sijaan.

Koko purkualgoritmin suorituksen aikana kukin mahdollinen osamerkkijono tulee lisätyksi

Algorithm 9 Tiivis osamerkkijonoverkko tiivistysalgoritmin tulosteesta

```
 1: procedure DecodeCST(S)
 2:     C_ε ← ReadNumber(S)
 3:     CreateNode(n_ε, C_ε)
 4:     node_list ← ()
 5:     for all a ∈ Σ do
 6:         C_a ← ReadNumber(S)
 7:         if C_a > 0 then
 8:             CreateNode(n_a, C_a)
 9:             CreateSuffixLink(n_a → n_ε)
10:             AddList(node_list, n_a)
11:         end if
12:     end for
13:     w_queue ← {node_list}
14:     while w_queue is not empty do
15:         w ← Dequeue(w_queue)
16:         C_w ← InitCountMatrix(w)
17:         if C_w triviaalisti laskettavissa then
18:             C_w ← ComputeCountMatrix(C_w)
19:         else
20:             C_w ← ReadCountMatrix(C_w, S)
21:         end if
22:         CreateNodes(C_w)
23:         QueueCores(C_w)
24:     end while
25: end procedure
```

jonoon korkeintaan kerran. Lisääminen tapahtuu aina purkualgoritmin 9 rivillä 23 kutsuttavassa aliohjelmassa 13. Kutsuttaessa aliohjelmaa on juuri käsitelty osamerkkijono w ja lisätty kaikki osamerkkijonoja awc, $a, c \in \Sigma$, vastaavat solmut puuhun.

Aliohjelma 13 käy läpi tällä kierroksella lisätys solmut. Jokainen jonoon lisättävä lista vastaa osamerkkijonoa wc. Jotta lista lisättäisiin jonoon sen täytyy sisältää vähintään kaksi alkiota. Tämä tarkoittaa että kyseistä listaa vastaavaa osamerkkijono wc on vasemmalle maksimaalinen.

Purkualgoritmin 9 riveillä 16-21 muodostetaan osamerkkijonolle w esiintymismäärämatriisi $\boldsymbol{C}_w$. Matriisi alustetaan algoritmilla 10, joka luo nollamatriisin ja asettaa oikeat rivi- ja sarakesummat paikoilleen. Listan aw_nodes avulla matriisin $\boldsymbol{C}_w$ positiiviset rivisummat saadaan suoraan. Seuraavaksi edetään loppuosakaaren avulla jostain listan aw_nodes solmusta solmuun n_w. Positiiviset sarakesummat täydennetään käymällä läpi kaikki solmun n_w olemassa olevat lapset n_{wc}, jossa $c \in \Sigma$.

Alustuksen jälkeen esiintymismäärämatriisin käsittely jakaantuu kahteen tapaukseen. Yksinkertaisimmassa tapauksessa matriisin $\boldsymbol{C}_w$ sarakesummista vain yksi on positiivinen.

Algorithm 10 Alustaa esiintymismäärämatriisin $\boldsymbol{C}_w$ rivi- ja sarakesummat kaikkien n_{aw}, $a \in \Sigma$, muotoisten solmujen listasta.

```
1: procedure InitCountMatrix(aw_nodes)
2:     C_w ← ZeroMatrix(σ + 1, σ + 1)          ▷ luo σ + 1 × σ + 1 nollamatriisi
3:     for all n_aw ∈ aw_nodes do
4:         C_aw ← Weight(n_aw)
5:     end for
6:     n_w ← FollowSuffixLink(aw_nodes[1])
7:     for all n_wc ∈ Child(n_w) do
8:         C_wc ← Weight(n_wc)
9:     end for
10:    return C_w
11: end procedure
```

Algorithm 11 Laske esiintymismäärämatriisin $\boldsymbol{C}_w$ kaikki alkiot kun vain yksi sarakesumma on positiivinen.

```
1: procedure ComputeCountMatrix(C_w)
2:     c ← 0
3:     for all c ∈ Σ do
4:         if C_wc > 0 then
5:             break
6:         end if
7:     end for
8:     for all a ∈ Σ do
9:         C_awc ← C_aw
10:    end for
11:    return C_w
12: end procedure
```

Tämä tarkoittaa että osamerkkijonoa w voidaan laajentaa oikealle vain yhdellä symbolilla siten että laajennettu osamerkkijono esiintyy merkkijonorenkaassa T_r. Tässä tapauksessa koko esiintymismäärämatriisi voidaan täydentää pelkkien rivi- ja sarakesummien avulla lukematta tiivisteestä S mitään. Täydennys tapahtuu kutsumalla algoritmin 11 proseduuria ComputeCountMatrix. Proseduuri etsii sarakkeen c, jonka sarakesumma on positiivinen. Sarakkeessa olevat alkiot täydennetään rivisummien avulla, kuten lemman 2.1 todistuksessa selitetään.

Toisessa tapauksessa esiintymismäärämatriisilla $\boldsymbol{C}_w$ on useampi positiivinen sarakesumma. Sillä on myös useampi positiivinen rivisumma koska jonoon w_queue ei koskaan lisätä osamerkkijonoa joka voidaan laajentaa oikealle vain yhdellä symbolilla. Toisin sanoen osamerkkijono w on maksimaalinen ja sen alkioiden arvot joudutaan lukemaan tiivisteestä S. Proseduuri ReadCountMatrix lukee matriisin tiivisteestä kuten aiemmin luvussa 2.4 on selitetty.

Algorithm 12 Lisää n_{awc} muotoisten solmut puuhun

```
1: procedure CreateNodes(aw_nodes, C_w)
2:     for all n_aw ∈ aw_nodes do
3:         node ← n_aw
4:         for all c ∈ Σ do
5:             if C_awc > 0 then
6:                 if C_awc = C_wc then
7:                     n_w ← FollowSuffixLink(node)
8:                     n_wc ← Child(n_w, c)
9:                     CreateLink(node → n_wc, c)
10:                else
11:                    CreateNode(n_awc, C_awc)
12:                    n_w ← FollowSuffixLink(node)
13:                    n_wc ← Child(n_w, c)
14:                    CreateSuffixLink(n_awc → n_wc)
15:                    CreateLink(node → n_awc, c)
16:                end if
17:            end if
18:        end for
19:    end for
20: end procedure
```

Seuraavassa esimerkissä puretaan tekstistä $T = 224441$ luodusta tiivisteestä vastaava tiiviis osamerkkijonoverkko, joka vastaa merkkijonorengasta T_r. Tässä aakkosto on nelisymbolinen: $\Sigma = \{1, 2, 3, 4\}$.

Purkualgoritmin syöte eli tiiviste alkaa vektorilla

$$\vec{c} = [C_\epsilon, C_1, C_2, C_3, C_4] = [6, 1, 2, 0, 3]\,,$$

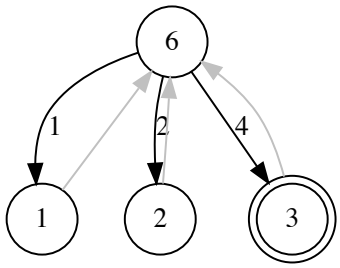
joka sisältää merkkijonon T kokonaispituuden C_ϵ sekä kaikkien aakkoston merkkien esiintymismäärät renkaassa T_r. Algoritmin 9 riveillä 2–12 tämä vektori luetaan tiivisteestä. Jokaiselle symbolille $a \in \Sigma$ luodaan puuhun solmu jos symbolin esiintymismäärä C_a on positiivinen. Lopuksi juurisolmun painoksi asetetaan vektorin kokonaissumma. Alustettu verkko on kuvassa 2.8. Tavallisten kaarten lisäksi puuhun on merkitty loppuosakaaret harmailla nuolilla. Symboli 3 ei esiinny merkkijonorenkaassa T_r kertaakaan, joten sitä vastaavaa solmua ei luotu. Viimeisenä luotu solmu on merkitty kaksinkertaisella renkaalla.

Vektorin $\vec{c}$ jälkeen tiivisteessä on maksimaalisia osamerkkijonoja, joiden pituus on korkeintaan $C_\epsilon - 2 = 4$ symbolia, vastaavat esiintymismäärämatriisit:

$$\boldsymbol{C}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix},$$

Algorithm 13 Lisää nykyistä osamerkkijonoa w oikealle jatkavat osamerkkijonot wc jonoon, jos wc on vasemmalle maksimaalinen.

```
1: procedure QueueCores(w_queue, C_w)
2:     for all c ∈ Σ do
3:         aw_list ← ()                          ▷ uusi osamerkkijono on wc
4:         for all a ∈ Σ do
5:             if C_awc > 0 then
6:                 AddList(aw_list, n_awc)
7:             end if
8:         end for
9:         if Length(aw_list) ≥ 2 then           ▷ onko wc vasemmalle maksimaalinen?
10:            Push(w_queue, aw_list)
11:        end if
12:    end for
13: end procedure
```



Kuva 2.8: Tiivis osamerkkijonoverkko, joka on alustettu tiivisteestä lisäämällä kaikki renkaassa T_r esiintyvät aakkoston symbolit ja niiden esiintymismäärät.

$$\boldsymbol{C}_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\boldsymbol{C}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

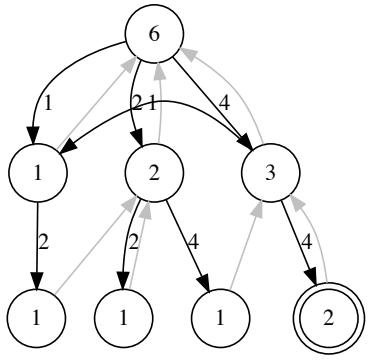
ja

$$\boldsymbol{C}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Nämä vastaavat merkkijonorenkaan T_r maksimaalisia osamerkkijonoja $\mathcal{M} = \{\epsilon, 2, 4, 44\}$ lyhimmästä pisimpään, aakkosjärjestyksessä. Tässä vaiheessa purkua algoritmi ei kuitenkaan tiedä mitä merkkijonoja maksimaaliset osamerkkijonot ovat. Ne selviävät vasta kun osamerkkijono liitetään rakennettavaan verkkoon.

Seuraavaksi purkualgoritmi laajentaa tyhjää osamerkkijonoa ϵ kummaltakin puolelta kaikin mahdollisin merkein. Eli muodostaa kaikki merkkijonot $a\epsilon c$, jossa $a, c \in \Sigma$ ja $C_{a\epsilon c} > 0$. Esimerkin tapauksessa joukko solmuja $\{n_{1\epsilon}, n_{2\epsilon}, n_{4\epsilon}\}$ on tallessa jonossa. Solmun $n_{1\epsilon}$ ja loppuosakaaren avulla algoritmi 9 löytää osamerkkijonoja $\epsilon 1, \epsilon 2$ ja $\epsilon 4$ vastaavat solmut $n_{\epsilon 1}, n_{\epsilon 2}$ ja $n_{\epsilon 4}$. Koska ϵc-mallisia osamerkkijonoja on useampi kuin yksi, osamerkkijono ϵ on maksimaalinen. Siten esiintymismäärämatriisin $\boldsymbol{C}_\epsilon$ sisältöä ei voida päätellä puussa olevasta tiedosta vaan se pitää lukea tiivisteestä.

Nyt tiedetään että $\boldsymbol{C}_1 = \boldsymbol{C}_\epsilon$. Tämä ensimmäisenä tiivisteessä oleva esiintymismäärämatriisi $\boldsymbol{C}_1$ luetaan ja sen perusteella muodostetaan $n_{a\epsilon c}$-muotoiset solmut. Nämä solmut on lisätty kuvan 2.8 puuhun kuvassa 2.9. Koska $C_{4\epsilon 1} = C_{\epsilon 1}$ merkkijonolle $4\epsilon 1$ ei luoda omaa solmua vaan sitä merkitsevä kaari $n_{4\epsilon} \xrightarrow{1} n_{\epsilon 1}$.

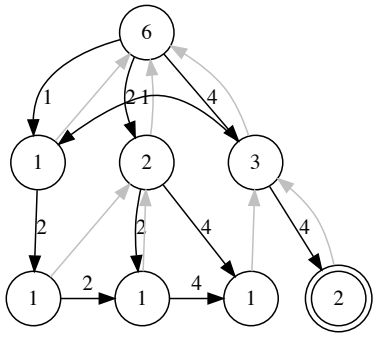


Kuva 2.9: Kuvan 2.8 verkko, kun osamerkkijonoa ϵ laajentavat osamerkkijonot on lisätty siihen.

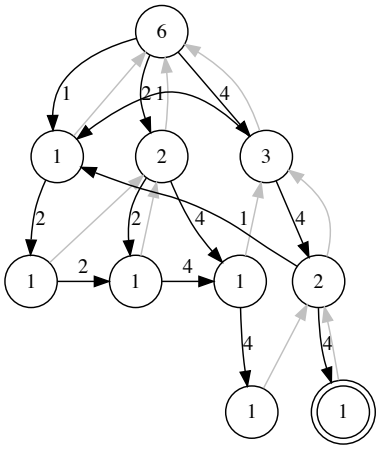
Lopuksi proseduuri QueueCores lisää jonoon w_queue solmut $\langle n_{1\epsilon 2}, n_{2\epsilon 2}\rangle$ ja solmut $\langle n_{2\epsilon 4}, n_{4\epsilon 4}\rangle$. Näistä ensimmäinen vastaa osamerkkijonoa $\epsilon 2$ ja toinen osamerkkijonoa $\epsilon 4$.

Seuraavaksi käsitellään jonossa ensimmäisenä oleva osamerkkijono $\epsilon 2$ eli osamerkkijono 2. Koska puusta löytyvät solmut n_{22} ja n_{24} täytyy tätäkin osamerkkijonoa vastaava esiintymismäärämatriisi lukea tiivisteestä. Matriisin $\boldsymbol{C}_2$ perusteella puuhun ei lisätä yhtään uutta solmua. Niiden sijaan luodaan kaksi kaarta $n_{1\epsilon 2} \xrightarrow{2} n_{\epsilon 22}$ ja $n_{2\epsilon 2} \xrightarrow{4} n_{\epsilon 24}$ kuten kuvasta 2.10 selviää. Tällä kierroksella jonoon w_queue ei ilmaannu uusia osamerkkijonoja.

Tällä hetkellä viimeinen jonossa oleva osamerkkijono $\epsilon 4$ käsitellään lukemalla matriisi $\boldsymbol{C}_3 = \boldsymbol{C}_{\epsilon 4}$, koska verkosta löytyy solmut n_{41} ja n_{44}. Kuvassa 2.11 on tiivis osamerkkijonoverkko tämän operaation jälkeen. Verkkoon on lisätty solmu $n_{2\epsilon 44}$, koska $C_{2\epsilon 44} = 1 \neq 2 = C_{\epsilon 44}$, ja solmu $n_{4\epsilon 44}$, koska $C_{4\epsilon 44} = 1 \neq 2 = C_{\epsilon 44}$. Osamerkkijonoa $4\epsilon 41$ vastaamaan luotiin kaari $n_{4\epsilon 4} \xrightarrow{1} n_{\epsilon 41}$, koska $C_{4\epsilon 41} = 1 = C_{\epsilon 41}$.



Kuva 2.10: Tiivis osamerkkijonoverkko, kun osamerkkijonoa $\epsilon 2$ laajentavat osamerkkijonot on lisätty siihen.

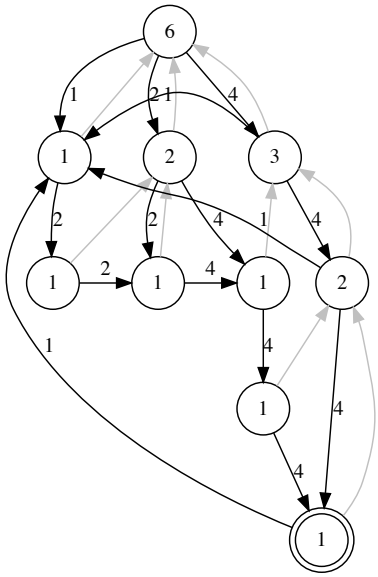


Kuva 2.11: Tiivis osamerkkijonoverkko, kun osamerkkijonoon $\epsilon 4$ liittyvät osamerkkijonot on lisätty siihen.

Kierroksella puuhun lisätyistä osamerkkijonoista löytyy yksi merkkijono jota voidaan jatkaa vasemmalle kahdella eri symbolilla. Tätä osamerkkijonoa $\epsilon 44$ vastaavat solmut $\langle n_{2\epsilon 44}, n_{4\epsilon 44} \rangle$ lisätään jonoon w_queue.

Yllä lisätyt solmut edustavat osamerkkijonoa $\epsilon 44$ joka on ainoana jonossa w_queue. Siten se käsitellään seuraavaksi. Tässäkin tapauksessa ei ole vain yhtä yksittäistä merkkiä jolla osamerkkijonoa $\epsilon 44$ voidaan jatkaa oikealle, joten esiintymismäärämatriisi pitää lukea tiivisteestä. Seuraava ja viimeinen matriisi tiivisteessä on C_4. Kierros ei lisää puuhun yhtään uutta solmua. Sen sijaan lisätään kaari $n_{244} \xrightarrow{4} n_{444}$ koska $C_{2\epsilon 444} = 1 = C_{444}$ ja kaari $n_{444} \xrightarrow{1} n_1$ koska $C_{4\epsilon 441} = 1 = C_{\epsilon 441}$.

Tällä kierroksella ei lisätty yhtään osamerkkijonoa jonoon w_queue joten kierroksen jälkeen jono on tyhjä. Seurauksena Algoritmin 9 WHILE-silmukan ehto rivillä 14 on epätosi ja algoritmin suoritus päättyy. Purkualgoritmin muodostama tiivis osamerkkijonoverkko on



Kuva 2.12: Merkkijonorengasta T_r vastaava tiivis osamerkkijonoverkko, joka on luotu tiivisteestä Algoritmin 9 avulla.

kokonaisuudessaan kuvassa 2.12.

Alkuperäisen merkkijonon T lukeminen. Algoritmi 9 tuottaa tiivisteestä tiiviin osamerkkijonoverkon, mutta ei suoraan alkuperäistä tekstiä T. Seuraavassa käydään läpi miten verkosta voidaan lukea teksti. Verkon jokainen solmu n_w, jonka paino on 1, vastaa yhtä tai useampaa osamerkkijonoa, joka esiintyy renkaassa T_r tasan kerran. Jokaisesta tällaisesta solmusta n_w lähtee tasan yksi kaari, koska mitä tahansa renkaassa esiintyvää osamerkkijonoa voidaan aina jatkaa vähintään yhdellä symbolilla oikealle. Toisaalta kaaria voi olla vain yksi, koska merkkijonorenkaassa ei voi esiintyä osamerkkijonoja wc ja wd millään symboleilla $c(\neq d), d \in \Sigma$, kun osamerkkijono w esiintyy renkaassa vain kerran. Saman päätelmän mukaan kaaren kohdesolmun painon tulee olla 1, muuten osamerkkijonoja wc, jollain symbolilla $c \in \Sigma$, olisi enemmän kuin osamerkkijonoja w.

Edellisen perusteella verkon kaikkien painolla 1 varustettujen solmujen tulee muodostaa suljettu suunnattu rengas. Rengas on ainoa mahdollinen rakenne, jossa jokaisella solmulla voi olla tasan yksi suunnattu kaari johonkin toiseen, samassa rakenteessa olevaan, solmuun.

Kuvan 2.12 tiivis osamerkkijonoverkko purettiin edellisessä esimerkissä Algoritmilla 9 tekstin $T = 224441$ tiivisteestä. Viimeisenä luotu solmu on kuvan verkossa korostettu kaksinkertaisella renkaalla. Solmun paino on 1 joten se kuuluu yllä kuvatunlaiseen renkaaseen. Kiertämällä rengas ja konkatenoimalla kaarien nimet yhteen saadaan merkkijono 122444. Tämä on yksi alkuperäisen tekstin T mahdollisista kierroista.

Danny Dubé ja Vincent Beaudoin ehdottavat, että tiivistysvaiheessa kaikki tiivistettävän

tekstin T mahdolliset kierrot voidaan laittaa aakkosjärjestykseen [12]. Luettelosta selviää kuinka mones teksti T on kaikkien kiertojensa aakkosjärjestyksessä. Tämä järjestysluku voidaan lisätä tiivisteen loppuun, josta se purun lopussa luetaan. Lopullinen teksti voidaan palauttaa tiiviistä osamerkkijonoverkosta tämän luvun avulla.

Tässä työssä esitelty menetelmä, joka toteuttaa CSE-tiivistysalgoritmi kaksisuuntaisen BWT-indeksin avulla, ei tarvitse yllä esiteltyä järjestyslukua. BWT-indeksi vaatii, että käsiteltävä teksti T päättyy erityisellä loppusymbolilla, joka ei esiinny muualla tekstissä. Tämän loppusymbolin avulla voidaan tiiviistä osamerkkijonoverkosta luetusta tekstin T kierrosta löytää viimeinen symboli ja sitä kautta alkuperäisen tekstin alku.

3 Toteutus

Tässä luvussa esitellään luvussa 2 kuvatun tiivistysmenetelmän toteuttava ohjelma *cse*. Äärellisen aakkoston kooksi on valittu vakio $\sigma = 256$, eli ohjelma käsittelee syötetietoa jonona 8-bittisiä tavuja tai symboleita. Kummatkin luvussa 2.4 esitetyt vaihtoehdot esiintymismäärämatriisien koodaamiseksi tiivisteeseen on toteutettu. Tiivistyksen lisäksi on toteutettu tiivisteen purku luvussa 2.5 kuvatulla menetelmällä.

Luvussa käydään ensin läpi perusasioita ohjelman toteutuksesta, ja muutamia kirjastoja, joita toteutus hyödyntää. Niiden jälkeen esitellään oleelliset yksityiskohdat toteutuksesta, sekä puhutaan ohjelman testauksesta. Lopuksi alaluvussa 3.1 esitellään ohjelmalla tuotettuja kokeellisia tuloksia, ja selitetään miten kokeet on suoritettu. Tutkielman toteutuksen lisäksi esitellään tuloksia myös muista CSE-menetelmän toteutuksista, sekä kokonaan toisista tiivistysmenetelmistä.

Aiemmasta poiketen tässä luvussa symboleiden arvo on suljetulla välillä $[0, 255]$, tai ilmaistuna heksadesimaalilukuina $[00_{16}, \mathrm{ff}_{16}]$, koska nollasta alkavat lukuvälit ovat luonnollisempia käsitellä käytössä olevilla tietokoneilla. Jatkossa aakkosto Σ on siis joukko $\{00_{16}, 01_{16}, \ldots, \mathrm{ff}_{16}\}$.

Toteutus on kirjoitettu C++-ohjelmointikielen versiolla C++17 [22]. Käytetyt kirjastot on kirjoitettu joko C++-ohjelmointikielellä tai C-ohjelmointikielellä, tai näiden sekoituksella.

Ohjelma on toteutettu tavanomaisena komentorivityökaluna. Esimerkiksi tiedoston *book1* voi tiivistää komennolla

```
$ cse book1
```

Edellinen komento tuottaa tiivistetiedoston *book1.cse*, joka voidaan purkaa alkuperäiseksi tiedostoksi komennolla

```
$ cse -d book1.cse
```

Purkukomento tuottaa tiedoston *book1.cse.out*, jonka sisältö on identtinen tiedoston *book1* kanssa.

Oletuksena esiintymismäärämatriisien tallettamiseen käytetään alkio kerrallaan tallettavaa menetelmää. Komentorivivalitsimella `--serialize sums` voidaan valita käyttöön matriisit rekursiivisesti puolittava menetelmä.

Toteutus tukeutuu C++-kielen standardikirjaston lisäksi seuraaviin valmiisiin kirjastoihin:

- Jarno Alangon *BD_BWT_index* [1] toteuttaa kaksisuuntaisen BWT-indeksin,
- *CLI11* [21] tarjoaa komentorivivalitsinten käsittelyn ja
- *Catch2* [25] on yksikkötestauskirjasto.

BD_BWT_index-kirjasto perustuu Djamal Belazzouguin, Fabio Cunialin, Juha Kärkkäisen ja Veli Mäkisen artikkeliin [3]. Kirjasto hyödyntää sdsl-lite-kirjaston [19] tarjoamia tilatiiviitä tietorakenteita, sekä libdivsufsort-kirjastoa [26] Burrows-Wheeler-muunnoksen luomiseen.

BD_BWT_index-kirjaston käsittelemä aakkosto on 8-bittinen: syöte otetaan vastaan taulukkona 8-bittisiä tavuja, eli symboleita. Kirjasto vaatii, ettei syötteessä esiinny symboleita, joiden arvo on 00_{16} tai 01_{16}. Symbolia 00_{16} käytetään merkitsemään syötteen loppua, ja symbolia 01_{16} käytetään Burrows-Wheeler-muunnoksessa loppumerkkinä, eli samassa asemassa kuin tässä työssä on aiemmin käytetty merkkiä #.

Jotta ohjelmalla voidaan tiivistää mielivaltaisia syötetiedostoja, pitää tiedostoissa esiintyvät 00_{16}- ja 01_{16}-symbolit poistaa. Toteutus esikäsittelee tiedoston sisällön, tässä merkkijonon $T \in \Sigma^*$, ennen BWT-indeksin muodostusta. Esikäsittelyssä jokainen 00_{16}-symboli korvataan kahden symbolin mittaisella merkkijonolla $02_{16}03_{16}$. Vastaavasti symbolit 01_{16} korvataan symbolipareilla $02_{16}04_{16}$.

Edellisen symbolien korvausmallin ongelma on, ettei se osaa erotella seuraavia kahta tilannetta toisistaan:

1. merkkijonossa T esiintynyt symboli 00_{16} on korvattu merkkijonolla $02_{16}03_{16}$ ja
2. merkkijonossa T esiintyy alunperin osamerkkijono $02_{16}03_{16}$.

Ongelma korjataan korvaamalla jokainen alkuperäisessä merkkijonossa T esiintyvä symboli 02_{16} merkkijonolla $02_{16}02_{16}$. Tällöin kohdan 2 osamerkkijono $02_{16}03_{16}$ korvautuu osamerkkijonolla $02_{16}02_{16}3_{16}$. Luettaessa korvaukset sisältävää merkkijonoa vasemmalta oikealle, voidaan yllä esitellyt kaksi tilannetta erottaa toisistaan. Kun vastaan tulee symboli 02_{16}, tutkitaan aina seuraava symboli, ja palautetaan sen mukaan symboliparin tilalle joko symboli 00_{16}, 01_{16} tai 02_{16}.

Mikäli tiivistettävä merkkijono T sisältää paljon symboleita 00_{16}, 01_{16} tai 02_{16}, korvausoperaatio voi kasvattaa sen pituutta paljon, sillä jokainen luetelluista symboleista korvataan kahdella symbolilla. Pahimmassa tapauksessa merkkijonon T pituus kaksinkertaistuu ennen varsinaista tiivistystä. Tämän havaittiin kasvattavan tiivisteen kokoa huomattavasti joissain tapauksissa.

Pituuden kasvun välttämiseksi on lisätty esikäsittelyvaihe, joka etsii kolme vähiten merkkijonossa T esiintyvää symbolia, jotka eivät kuulu joukkoon $\{00_{16}, 01_{16}, 02_{16}\}$. Merkitään löydettyjä symboleita x_1, x_2 ja x_3. Niiden esiintymismäärille pätee $C_{x_1} \leq C_{x_2} \leq C_{x_3}$, eli x_1 on harvinaisin. Parhaimmassa tapauksessa yksikään näistä symboleista ei esiinny merkkijonossa T kertaakaan.

Ennen symboleiden korvaamista kahden symbolin mittaisilla merkkijonoilla, vaihdetaan

merkkijonon T symbolit seuraavasti:

$$\begin{aligned} 02_{16} &\to x_1, & x_1 &\to 02_{16}, \\ 00_{16} &\to x_2, & x_2 &\to 00_{16}, \\ 01_{16} &\to x_3, & x_3 &\to 01_{16}. \end{aligned}$$

Eniten korvauksessa käytetty symboli 02_{16} korvataan merkkijonon T harvinaisimmalla symbolilla x_1.

Tiivisteen alkuun lisätään kolmen symbolin mittainen merkkijono $x_1x_2x_3$, jonka avulla purkaja osaa vaihtaa symbolit takaisin oikeiksi, ja palauttaa alkuperäisen merkkijonon T.

Tukeakseen kumpaakin luvussa 2.4 esiteltyä esiintymismäärämatriisin koodausmenetelmää, ohjelma kirjoitettiin aluksi käyttäen erillistä luokkaa esiintysmimäärämatriisin esittämiseen. Kummassakin tapauksessa yhteinen osuus tuottaa maksimaaliset osamerkkijonot algoritmin 3 avulla. Maksimaaliset merkkijonot tuotetaan yksi kerrallaan, jotta ne eivät vie turhaan muistia. Kun seuraava maksimaalinen merkkijono löydetää, muodostetaan sitä vastaava esiintymismäärämatriisi algoritmin 4 avulla. Tämän matriisin käsittelee joko alkio kerrallaan koodaava tai rekursiivisesti matriisin pilkkova menetelmä. Riippuen ohjelman komentovalitsimista.

Ohjelman suoritusajan profilointi osoitti erillisen matriisiluokan käytön hitaaksi: tiivistäessä koko suoritusajasta vajaa 80 % kului matriisin käsittelyyn. Esiintymismäärämatriisin alkio kerrallaan koodaava osuus optimointiin siten, että maksimaalista osamerkkijonoa laajennettaessa esiintymismäärämatriisiksi, matriisin alkiot kirjoitetaan suoraan tiedostoon. Vastaava optimointi puolittavaan koodaukseen ei onnistu suoraan, sillä matriisin alkioiden arvoja tarvitaan puolitusvektoreiden laskemiseen. Optimointi ei siten näy puolittavan koodauksen toteuttavan tiivistysmenetelmän suoritusajoissa.

Tiivisteen purku on toteutettu suoraviivaisesti luvun 2.5 purkualgoritmin 9 mukaan. Tiiviin osamerkkijonopuun jokaisella solmulla on hajautustaulu, johon talletetaan osoittimet solmun lapsiin. Oletuksena on, että keskimääräisellä solmulla on paljon vähemmän lapsia kuin aakkoston koko σ. Silloin hajautustaulu vie vähemmän muistia, kuin σ alkion kokoinen taulukko.

Tiiviin osamerkkijonoverkon muodostamisen jälkeen siitä luetaan alkuperäinen merkkijonorengas. Renkaasta etsitään BWT-indeksin käyttämä loppusymboli 01_{16}. Symbolin avulla löydetään alkuperäisen merkkijonon alku. Ennen tiedostoon kirjoittamista merkkijonosta poistetaan ennen tiivistystä luodut symboliparit, ja palautetaan korvatut symbolit oikeiksi. Toimenpidettä varten joudutaan käyttämään vähintään merkkijonon pituuden verran ylimääräistä tilaa.

Ohjelman testaus perustuu pääosin yksikkötestaukseen. Siinä pieniä toiminnallisia yksiköitä on testattu erillään toisista. Testattavalle yksikölle on kirjoitettu joukko testejä, jotka varmistavat sen reagoivan oikein annettuihin syötteisiin ja komentoihin. Testaus on kohdistettu erityisesti loogisesti monimutkaisiin osiin, kuten esiintymismäärämatriisin puolitusalgoritmiin, puolitetun matriisin koostamiseen takaisin kokonaiseksi matriisiksi ja maksimaalisten osamerkkijonojen luettelualgoritmiin. Yksikkötestit on rakennettu

Catch2-testaussovelluskehyksen (engl. *test framework*) avulla.

Integraatiotestaus on suoritettu tiivistämällä seuraavassa luvussa esiteltäviä testidatatiedostoja, ja purkamalla tiivisteitä. Ohjelman toimivuus täysin integroituna kokonaisuutena on osoitettu varmistamalla, että alkuperäinen tiedosto ja sen tiivisteestä purettu tiedosto ovat identtiset.

3.1 Kokeelliset tulokset

Kaikki tämän luvun kokeet on suoritettu tietokoneella, jossa on Intelin Core i5-2500K 3.30 GHz suoritin ja 16 Git keskusmuistia. Ohjelma ja kirjastot on käännetty samassa tietokoneessa GNU Compiler Collection -kääntäjän versiolla 9.3.0, käyttäen optimointivalitsimia `-O3`, `--arch=native` ja `--flto`. Valitsimista ensimmäinen ottaa käyttöön agressiivisimmat optimoinnit. Toinen valitsin sallii kääntäjän käyttää kaikkia konekäskyjä, joita tietokone, jossa kääntö tapahtuu, tukee. Viimeinen valitsin `--flto` ottaa käyttöön linkityksen aikaisen optimoinnin. [17]. Kaikki kokeet on ajettu yksi kerrallaan, jotta ne eivät häiritsisi toisiaan, ja erityisesti, eivät kilpaile muistista.

Kokeissa käytetty data on peräisin kahdesta lähteestä. Calgary-korpus [33] sisältää joukon erilaisia tiedostoja, jotka on lueteltu taulukossa 3.1. Koska jokainen Calgary-korpuksen tiedosto on pienempi kuin 1 MiB, suurempia testitiedostoja on haettu Pizza&Chili-korpuksesta [16]. Mahdollisista korpuksista valittiin *english*, koska sen aakkoston koko on kaikista vaihtoehdoista suurin: 239 symbolia. Toinen valittu korpus on *dna*, jonka aakkostossa on 16 eri symbolia, eli vaihtoehdoista pienin. Kummastakin tekstistä on erikseen neljä tiedostoa, jotka sisältävät ensimmäiset 128 kiB, 1 MiB, 50 MiB ja 200 MiB tekstin alusta. Tiedostot on lueteltu taulukossa 3.2.

tiedosto	*sisältö*	*koko tavuissa*
bib	bibliografia (refer-tiedostomuoto)	111 261
book1	fiktiokirja	768 771
book2	tietokirja (troff-tiedostomuoto)	610 856
geo	geofysikaalista dataa	102 400
news	USENET-arkisto	377 109
obj1	konekoodia VAX-tietokoneesta	21 504
obj2	konekoodia Apple Mac -tietokoneesta	246 814
paper1	tekninen artikkeli	53 161
paper2	tekninen artikkeli	82 199
pic	mustavalkoinen kuva	513 216
progc	lähdekoodia, C-ohjelmointikieli	39 611
progl	lähdekoodia, LISP	71 646
progp	lähdekoodia, PASCAL	49 379
trans	terminaali-istunnon litterointi	93 695

Taulukko 3.1: Calgary-korpuksen sisältämät tiedostot kokoineen.

tiedosto	*sisältö*	*koko tavuissa*
dna.128kB	DNA-sekvenssejä	131 072
dna.1MB	DNA-sekvenssejä	1 048 576
dna.50MB	DNA-sekvenssejä	52 428 800
dna.200MB	DNA-sekvenssejä	209 715 200
english.128kB	englannin kielistä tekstiä	131 072
english.1MB	englannin kielistä tekstiä	1 048 576
english.50MB	englannin kielistä tekstiä	52 428 800
english.200MB	englannin kielistä tekstiä	209 715 200

Taulukko 3.2: Pizza&Chili-korpuksesta valikoidut tiedostot ja niiden koot.

Kokeissa on käytetty useita menetelmiä. Näistä *CSE ($\sigma = 256$)* on työssä toteutettu CSE-tiivistysmenetelmä, jossa esiintymismäärämatriisi koodataan lineaarisesti, matriisin alkio kerrallaan. Menetelmä *CSE-jako ($\sigma = 256$)* on esiintymismäärämatriisit rekursiivisesti jakamalla koodaava. *CSE ($\sigma = 2$)* on Dubén ja Beaudoin alkuperäinen CSE-menetelmä binääriselle aakkostolle, sen tiivistyvyysluvut on otettu artikkelista [12].

Kolmanneksi äärellisen aakkoston CSE-menetelmäksi on valittu *CSE-PS ($\sigma = 256$)*. Menetelmä on esitetty Otan ja kanssakirjoittajien artikkelissa [32]. Tämä menetelmä järjestää esiintymismäärämatriisin sarakkeet laskevaan suuruusjärjestykseen sarakesummien mukaan. Lisäksi he tallettavat esiintymismäärämatriisien alkiot käyttäen adaptiivista aritmeettista koodausta. Edelliset ja toteutusmenetelmä pois lukien, menetelmä vastaa *CSE ($\sigma = 256$)*-tiivistysmenetelmää.

Vertailtavaksi perinteiseksi tiivistysmenetelmäksi on valittu GNU Gzip -ohjelman versio 1.10. Ohjelma toteuttaa DEFLATE-algortimin [18], joka yhdistää LZ77-tiivistysmenetelmän Huffmanin koodaukseen [8]. Kaikissa kokeissa Gzip-ohjelmaa käytetään komentirivivalitsimella `--best`, eli `-9`, joka tuottaa tiiveimmän tiivisteen, mutta käyttää eniten aikaa.

Pizza&Chili-korpuksen teksteillä vertailussa on mukana Julian Sewardin bzip2-ohjelma. Ohjelma hyödyntää tiivistyksessä Burrows-Wheeler-muunnosta [35], eli käyttää koko tiivistettävää merkkijonoa, kuten CSE-menetelmä. Tiedostoa ei kuitenkaan välttämättä tiivistetä kokonaisena kerrallaan, vaan se saatetaan jakaa lohkoiksi, jotka tiivistetään erikseen. Lohkoiksi jako tapahtuu, jos valittu tiedosto ei mahdu yhteen lohkoon. Kaikissa tämän työn kokeissa bzip2-ohjelmaa käytetään komentorivivalitsimella `-9`. Valitsin määrittää lohkokooksi 900 000 tavua, mikä on suurin tuettu lohkokoko. Pizza&Chili-korpuksesta valikoitujen tiedostojen tapauksessa lohkokoosta seuraa, että vain 128 kiB kokoiset tiedostot tiivistetään yhtenä lohkona. Suuremmat tiedostot jakautuvat useampaan lohkoon.

Taulukossa 3.3 esiintyvä *BWT* on Burrowsin ja Wheelerin esittelemä Burrows-Wheeler-muunnokseen perustuva tiivistysmenetelmä [5]. Menetelmän tiivistyvyysluvut on otettu samasta artikkelista. PPM-menetelmän tiivistyvyysluvut ovat vuorostaan peräisin John G. Clearyn ja William J. Teahanin artikkelista [6].

Jatkossa kaikissa koetulostaulukoissa on kunkin koetekstin kohdalla vahvennettu paras tulos. Jos useamman menetelmän tulokset ovat yhtä hyviä, kaikkien tulos on vahvennettu.

Taulukossa 3.3 on esitetty eri tiivistysmenetelmien tiivistystehokkuus Calgary-korpuksen tiedostoilla. Tiivistystehokkuus, tai tiivistyvyys, on luvun kokeissa tiivisteen koon suhde alkuperäisen datan kokoon. Toisin sanoen, jos alkuperäinen data vaatii l_a tavua tilaa ja siitä tehty tiiviste l_t tavua, tiivistyvyys on $\frac{l_t}{l_a}$. Pizza&Chili-korpuksen tiedostojen tiivistyvyysluvut on taulukossa 3.4. Työssä esiteltyjen CSE-menetelmien kahden variaation lisäksi mukana on vain Gzip- ja bzip2-ohjelmien tulokset. Muista CSE-menetelmistä ei ole saatavilla toteutuksia, tai tuloksia muista, kuin Calgary-korpuksen tiedostoista.

data	Gzip	BWT	PPM	CSE ($\sigma = 2$)	CSE-PS ($\sigma = 256$)	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
bib	0.31	0.26	**0.24**	0.25	**0.24**	0.34	14.02
book1	0.41	0.31	0.30	**0.28**	0.30	0.40	22.50
book2	0.34	0.27	**0.25**	**0.25**	**0.25**	0.36	18.92
geo	0.67	**0.56**	0.60	0.67	0.58	1.57	8.43
news	0.38	0.32	**0.30**	0.31	0.31	0.50	16.27
obj1	**0.48**	0.50	0.50	0.56	0.51	1.92	25.75
obj2	0.33	0.33	**0.30**	0.34	0.31	0.91	8.78
paper1	0.35	0.32	**0.30**	0.32	0.31	0.47	18.41
paper2	0.36	0.31	**0.29**	0.30	0.30	0.43	20.63
paper3	0.39	-	-	**0.34**	-	0.49	21.21
paper4	0.42	-	-	**0.40**	-	0.57	20.79
paper5	**0.42**	-	-	**0.42**	-	0.63	19.33
paper6	0.35	-	-	**0.33**	-	0.49	17.98
pic	**0.10**	**0.10**	0.11	**0.10**	0.12	0.22	12.23
progc	0.33	0.32	**0.30**	0.32	0.31	0.50	14.87
progl	0.23	0.22	**0.21**	**0.21**	**0.21**	0.30	12.58
progp	0.23	0.22	**0.20**	0.22	0.21	0.35	11.47
trans	0.20	0.20	**0.18**	0.20	0.19	0.32	9.23

Taulukko 3.3: Tiivisteen koko suhteessa alkuperäisen datan kokoon. Datajoukkona on Calgary-korpukseen sisältyvät tiedostot.

Tiivistykseen kuluva aika on esitetty taulukossa 3.5 Calgary-korpuksen tiedostoille, ja taulukossa 3.6 Pizza&Chili-korpuksen tiedostoille. Kummastakin taulukosta näkee, että erillistä esiintymismäärämatriisiluokkaa käyttävä CSE-jako ($\sigma = 256$)-menetelmä on huomattavasti CSE ($\sigma = 256$)-menetelmää hitaampi. Tämä tukee aiemmassa luvussa 3 kuvattua päätöstä optimoida CSE ($\sigma = 256$)-menetelmä suoraviivaisemmaksi.

Ohjelmien muistinkulutus on mitattu Valgrind-ohjelman massif-työkalulla [36]. Massif mittaa ohjelman keosta, esimerkiksi operaatioita new, new[] ja vastaavia käyttäen, varaamaa muistia. Lisäksi sillä voidaan mitata paljonko muistia varataan pinoista. Tässä työssä muistinkulutuksen mittaukseen on käytetty komentorivivalitsinta `--stacks=yes`, eli Valgrind mittaa myös ohjelman pinoista varaaman muistin. Keosta ja pinosta varatut muistimäärät on luvun taulukoissa summattu yhteen. Ilmoitettu muistinkulutus on maksimi muistimäärä, joka on ollut varattuna jossain kohtaa ohjelman suoritusta. Taulukossa 3.7 listataan tiivis-

data	Gzip	bzip2	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
dna.128kB	0.28	**0.27**	0.31	29.51
dna.1MB	0.27	**0.26**	0.30	29.52
dna.50MB	0.27	**0.26**	0.29	29.32
dna.200MB	0.27	**0.26**	0.28	29.28
english.128kB	0.37	**0.31**	0.44	21.04
english.1MB	0.36	**0.27**	0.36	20.62
english.50MB	0.38	**0.28**	0.29	17.31
english.200MB	0.38	**0.28**	0.29	18.18

Taulukko 3.4: Tiivisteen koko suhteessa alkuperäisen datan kokoon. Datajoukkona on valikoituja tiedostoja Pizza&Chili-korpuksesta.

tyksen muistinkulutukset Calgary-korpuksen tiedostoille. Pizza&Chili-korpuksen tiedostojen tiivistyksen muistinkulutukset on taulukossa 3.8. Gzip-ohjelman pieni muistinkulutus selittyy sen käyttämällä algoritmilla, joka ei missään vaiheessa pidä koko tiivistettävää tiedostoa, eikä tiivistettä, muistissa [8]. Vastaava toiminta selittää bzip2-ohjelman pienen muistinkulutuksen: tiedoston käsittely lohkoissa vaatii vain yhden lohkon pitämistä muistissa kerrallaan.

Gzip-ohjelman muistinkulutuksen selvittämiseen käytetään massif-työkalun komentorivivalitsinta `--pages-as-heap=yes`. Tällöin kaikki ohjelman tekemät sivuvaraukset (engl. *page allocation*), myös ohjelman oma konekoodi, näkyvät muistinkulutuksessa [36]. Ilman valitsinta Gzip-ohjelman muistinkulutus pyöristyy taulukoissa nollaan, eli olisi alle 10 kiB. Oletettavasti ohjelma varaa käyttämänsä muistin tavalla, joka ei näy massif-ohjelman normaalissa tarkkailussa. Epätarkemmasta, ja Gzip-ohjelmalle epäreilusta, mittaustekniikasta huolimatta, sen muistinkulutus on minimaalista, kuten esimerkiksi taulukosta 3.8 näkyy. Työssä toteutetussa cse-ohjelmassa ei käytetä suoria mmap-järjestelmäkutsuja tai muita massif-ohjelman ohittavia muistinvarauksia.

Tiivisteen purku. Tiivisteen purkuun kuluva aika Calgary-korpuksen tiedostojen osalta on lueteltu taulukossa 3.9. Purkuajat Pizza&Chili-korpuksen tiedostoille on taulukossa 3.10. Pizza&Chili-korpuksen tiedostoista mukana ovat vain 128 KiB ja 1 MiB kokoiset tiedostot, sillä tietokoneen 16 GiB muisti ei riitä suuremmista tiedostoista tehtyjen tiivisteiden purkamiseen työssä toteutetulla CSE-purkuohjelmalla.

Purun muistinkulutus on esitetty Calgary-korpuksen tiedostoille taulukossa 3.11. Taulukossa 3.12 on muistinkulutus Pizza&Chili-korpuksen tiedostoilla. Pienetkin tiedostot riittävät näyttämään, että purkualgoritmi vaatii paljon muistia.

data	Gzip	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
bib	**0.01**	0.34	3.63
book1	**0.06**	2.87	40.70
book2	**0.04**	2.27	25.77
geo	**0.02**	0.98	5.59
news	**0.02**	1.43	15.27
obj1	**0.00**	0.18	1.35
obj2	**0.03**	1.32	9.84
paper1	**0.01**	0.19	2.29
paper2	**0.01**	0.29	3.72
paper3	**0.00**	0.17	2.23
paper4	**0.00**	0.06	0.67
paper5	**0.00**	0.06	0.58
paper6	**0.00**	0.15	1.67
pic	**0.07**	1.56	19.21
progc	**0.00**	0.15	1.64
progl	**0.01**	0.19	2.26
progp	**0.01**	0.14	1.49
trans	**0.01**	0.24	2.28

Taulukko 3.5: CSE-ohjelman ajoajat sekunteina eri lukumäärämatriisien käsittelyalgoritmeilla. Datajoukkona on Calgary-korpukseen sisältyvät tiedostot.

data	Gzip	bzip2	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
dna.128kB	0.08	**0.01**	0.25	9.90
dna.1MB	0.64	**0.09**	2.01	82.09
dna.50MB	32.33	**4.31**	124.85	3961.05
dna.200MB	133.14	**17.14**	504.86	15701.53
english.128kB	**0.01**	**0.01**	0.49	6.27
english.1MB	0.09	**0.08**	3.88	49.57
english.50MB	4.34	**4.19**	226.26	1899.52
english.200MB	16.68	**16.74**	1074.95	8306.73

Taulukko 3.6: Tiivistyksen ajoajat CSE-ohjelmalla sekunteina eri lukumäärämatriisien käsittelyalgoritmeilla. Datajoukkona on valikoituja tiedostoja Pizza&Chili-korpuksesta.

data	Gzip	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
bib	5.02	**4.36**	**4.36**
book1	**5.02**	13.73	13.73
book2	**5.02**	12.95	12.95
geo	**5.02**	10.65	10.65
news	**5.02**	13.10	13.10
obj1	5.02	**3.81**	**3.81**
obj2	**5.02**	13.11	13.11
paper1	5.02	**3.12**	**3.12**
paper2	5.02	**3.63**	**3.63**
paper3	5.02	**3.01**	**3.01**
paper4	5.02	**1.82**	**1.82**
paper5	5.02	**1.84**	**1.84**
paper6	5.02	**2.75**	**2.75**
pic	**5.02**	11.74	11.74
progc	5.02	**3.00**	**3.00**
progl	5.02	**3.41**	**3.41**
progp	5.02	**2.88**	**2.88**
trans	5.02	**3.77**	**3.77**

Taulukko 3.7: Gzip- ja CSE-ohjelmien muistinkulutus megatavuina (1024^2 tavua). Datajoukkona on Calgary-korpukseen sisältyvät tiedostot.

data	Gzip	bzip2	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
dna.128kB	5.02	7.18	**2.38**	2.38
dna.1MB	**5.02**	7.18	11.57	11.57
dna.50MB	**5.02**	7.18	510.53	510.53
dna.200MB	**5.02**	7.18	2041.48	2041.49
english.128kB	**5.02**	7.18	5.20	5.20
english.1MB	**5.02**	7.18	17.91	17.91
english.50MB	**5.02**	7.18	534.43	534.43
english.200MB	**5.02**	7.18	2137.27	2137.27

Taulukko 3.8: Gzip-, bzip2- ja CSE-ohjelmien muistinkulutus megatavuina (1024^2 tavua). Datajoukkona on valikoituja tiedostoja Pizza&Chili-korpuksesta.

data	Gzip	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
bib	**0.00**	2.48	2.86
book1	**0.01**	24.25	27.46
book2	**0.01**	15.75	19.00
geo	**0.00**	3.29	3.75
news	**0.00**	9.07	11.48
obj1	**0.00**	0.63	0.90
obj2	**0.00**	6.70	7.29
paper1	**0.00**	1.38	1.71
paper2	**0.00**	2.28	2.65
paper3	**0.00**	1.38	1.58
paper4	**0.00**	0.37	0.46
paper5	**0.00**	0.34	0.41
paper6	**0.00**	0.96	1.26
pic	**0.01**	10.51	17.10
progc	**0.00**	0.88	1.28
progl	**0.00**	1.52	1.98
progp	**0.00**	0.95	1.37
trans	**0.00**	1.71	2.33

Taulukko 3.9: Tiivisteen purun aikavaativuus Gzip- ja CSE-ohjelmilla. Luvut ovat sekunteja, ja datajoukkona on Calgary-korpukseen sisältyvät tiedostot.

data	Gzip	bzip2	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
dna.128kB	**0.00**	0.01	4.27	6.95
dna.1MB	**0.01**	0.06	33.97	54.17
dna.50MB	-	-	-	
dna.200MB	-	-	-	
english.128kB	**0.00**	0.01	3.90	4.49
english.1MB	**0.01**	0.05	29.31	35.40
english.50MB	-	-	-	
english.200MB	-	-	-	

Taulukko 3.10: Tiivisteen purun aikavaativuus Gzip-, bzip2- ja CSE-ohjelmilla. Luvut ovat sekunteja, ja datajoukkona on Pizza&Chili-korpuksesta valitut tiedostot.

data	Gzip	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
bib	**5.02**	39.77	39.77
book1	**5.02**	315.67	315.67
book2	**5.02**	180.42	180.42
geo	**5.02**	40.91	40.91
news	**5.02**	159.92	159.92
obj1	**5.02**	5.91	5.91
obj2	**5.02**	79.62	79.62
paper1	**5.02**	20.06	20.06
paper2	**5.02**	23.72	23.72
paper3	**5.02**	20.20	20.20
paper4	**5.02**	5.25	5.25
paper5	**5.02**	5.30	5.30
paper6	**5.02**	11.56	11.56
pic	**5.02**	160.74	160.74
progc	**5.02**	11.72	11.72
progl	**5.02**	22.53	22.53
progp	**5.02**	20.13	20.13
trans	**5.02**	39.77	39.77

Taulukko 3.11: Tiivisteen purun tilavaativuus Gzip- ja CSE-ohjelmilla. Luvut ovat sekunteja, ja datajoukkona on Calgary-korpuksen tiedostot.

data	Gzip	bzip2	CSE ($\sigma = 256$)	CSE-jako ($\sigma = 256$)
dna.128kB	5.02	**3.51**	43.12	43.12
dna.1MB	5.02	**3.51**	344.95	344.95
dna.50MB	-	-	-	-
dna.200MB	-	-	-	-
english.128kB	5.02	**3.51**	40.94	40.94
english.1MB	5.02	**3.51**	328.81	328.81
english.50MB	-	-	-	-
english.200MB	-	-	-	-

Taulukko 3.12: Tiivisteen purun muistivaativuus Gzip-, bzip2- ja CSE-ohjelmilla. Luvut ovat megatavuja (1024^2 tavua), ja datajoukkona on Pizza&Chili-korpuksesta valitut tiedostot.

4 Johtopäätökset

Tiivistäminen osamerkkijonoja luettelemalla on suhteellisen uusi, kymmenen vuotta sitten esitelty, menetelmä häviöttömään tiedon tiivistykseen [12]. Siitä ei ole olemassa montaa toteutusta. Dubén ja Beaudoin alkuperäinen toteutus ei toimi lineaarisessa ajassa suhteessa syötteen kokoon [12]. Kanai kanssakirjoittajineen on toteuttanut menetelmän tehokkaasti binääriaakkostolle [24]. Toteutus tuottaa osamerkkijonojen esiintymismäärät, mutta ei niiden koodausta tiivisteeseen. Tiivistysmenetelmän teoreettiseksi muistivaativuudeksi on laskettu $O(n \log n)$, mutta toteutuksen empiiristä muistinkulutusta ei mainita.

Ainoa toteutus menetelmästä binääriaakkostoa suuremmille äärellisille aakkostoille on Otan ja kanssakirjoittajien artikkelissa [32]. Artikkelin kokeellisessa osiossa tutkitaan vain menetelmän tiivistystehokkuutta. Aika- tai muistivaatimuksia ei mainita.

Tässä työssä on toteutettu tiivistys osamerkkijonoja luettelemalla käyttäen tilatiiviitä tietorakenteita. Toteutus on äärellisille aakkostoille, erityisesti 8 bittisille tai pienemmille. Tiivistettävän merkkijonon maksimaalisten osamerkkijonojen etsintä ja osamerkkijonojen lukumäärän laskeminen toteutettiin käyttäen tilatiivistä kaksisuuntaista BWT-indeksiä. Tiivistyksen muistivaativuuden mittaus taulukoissa 3.7 ja 3.8 osoittaa, että muistinkulutus on maltillinen suhteessa tiivistettävän merkkijonon kokoon. Maltillisuudesta huolimatta, suurilla tiedostoilla muistinkulutus on noin kymmenkertainen tiivistettävän merkkijonon kokoon nähden. Burrows-Wheeler-muunnoksen tekevä libdivsufsort-kirjasto käyttää muistia $5n+O(1)$ tavua n tavun mittaisella tekstillä [26]. Tämä viittaa siihen, että muistinkulutusta on mahdollista optimoida lisää.

Tilankäytön tehokkuus korostuu CSE-menetelmä kanssa, koska koko tiivistettävä merkkijono joudutaan tallettamaan muistiin sen tiivistämiseksi. Dubé ja Beaudoin ovat ehdottaneet ratkaisuksi merkkijonon tiivistämistä lohkoina [12]. Lohkoa tiivistäessään algoritmi ei kuitenkaan voisi käyttää hyödykseen mitään informaatiota edellisistä lohkoista.

Edellisen luvun kokeissa pitkillä teksteillä saavutettiin parempi tiivistyvyys. Erityisesti Pizza&Chili-korpuksen tapauksessa taulukosta 3.4 nähdään miten *dna* tai *english* datajoukkojen tiivistyvyys paranee, kun dataa on enemmän. Tämän perusteella voidaan olettaa pienten lohkojen heikentävän pakkaustehoa. Myös intuitio tukee väitettä, koska on todennäköistä, että samassa tekstissä on samanlaisia, ja samalla tavalla laajenevia, maksimaalisia osamerkkijonoja eri osissa tekstiä. Lisäksi samanlaisten maksimaalisten osamerkkijonojen oikea ja vasen konteksti on todennäköisesti sama.

Esimerkiksi englanninkielisessä tekstissä maksimaalinen osamerkkijono ”he” vasen konteksti on usein joko merkki ’t’ tai välilyönti. Tällaisista osamerkkijonoista muodostuvissa esiintymismäärämatriiseissa saattaa olla paljon tyhjiä rivejä ja sarakkeita, ja positiiviset luvut painottuneita harvoihin alkioihin. Tämän kaltaiset matriisit vievät vähän tilaa tiivisteessä, sillä tyhjät rivit ja sarakkeet voidaan ohittaa suoraan. Hyödyistä huolimatta todella suuria merkkijonoja on epäkäytännöllistä tiivistää pilkkomatta niitä lohkoiksi. Lohkojen

koolle saattaa kuitenkin olla löydettävissä raja, joka on hyvä kompromissi tiivistyvyyden ja muistinkulutuksen välillä.

Eri tiivistysmenetelmien tiivistyvyyslukuja vertaillaan edellisen luvun taulukossa 3.3 Calgary-korpuksen tiedostoilla. Sama vertailu Pizza&Chili-korpuksen tiedostoille on taulukossa 3.4. Mitä pienempi luku tiivistyvyys on, sitä pienemmän tiivisteen menetelmä on saanut aikaan. Parhaimmilla menetelmillä on siten pienin luku. Taulukoista nähdään suoraan, ettei CSE-jako-menetelmä saa tiivistettyä yhtään tiedostoa alkuperäistä pienempään tilaan. Seuraavassa analyysissä keskitytään siksi esiintymismäärämatriisin alkiot yksi kerrallaan tallettavaan CSE ($\sigma = 256$) menetelmään. CSE-jako-menetelmän ongelmia käsitellään myöhemmin.

Calgary-korpuksen pienillä tiedostoilla CSE ($\sigma = 256$) pärjää huonosti verrattuna muihin tiivistysmenetelmiin. Tiedostojen geo ja obj1 kanssa tiiviste on jopa alkuperäistä tiedostoa suurempi. Muut tiivistysmenetelmät pärjäävät suhteellisesti huonommin samoilla tiedostoilla, joka viittaa siihen, että näissä tiedostoissa on vähemmän säännönmukaisuuksia.

Binäärisellä CSE-($\sigma = 2$)-menetelmällä on yksi etu, jota äärellisen aakkoston menetelmällä ei suoraan ole. Oletetaan, että tunnetaan jonkin osamerkkijonon w esiintymismäärä C_w ja sitä jatkavan osamerkkijonon $w0$ esiintymismäärä C_{w0}. Tällöin osamerkkijonon $w1$ esiintymismäärä $C_{w1} = C_w - C_{w0}$, sillä osamerkkijono w voi jatkua vain symbolilla 0 tai 1. Äärellisen aakkoston menetelmissä vastaavassa tilanteessa pitää tuntea osamerkkijonon jatko ensimmäisellä $\sigma - 1$ symbolilla, jotta jatko σ symbolilla tunnetaan suoraan. Esiintymismäärämatriisien koodauksessa käytetyt alkioiden ala- ja ylärajojen laskumentelmät ovat äärellisen aakkoston menetelmien todellinen vastine binääriaakkoston tekniikalle. Niissä käsitellään kerralla suurempaa määrää informaatiota. Ei tiedetä kumpi menetelmä vähentää välitettävän datan määrää tehokkaiten.

Toisena äärellisen aakkoston menetelmänä taulukossa 3.3 on Otan ja kanssakirjoittajien CSE-PS ($\sigma = 256$) [32]. Menetelmä on kaikissa tapauksissa parempi kuin Dubén ja Beaudoin alkuperäinen binäärisen aakkoston CSE. Erityisesti ei-tekstimuotoisilla tiedostoilla, kuten geo ja obj1, menetelmä suoriutuu hyvin. Tämä tukee Danny Dubén ajatusta, että binäärinen CSE-menetelmä tarvitsee apua, kun bitin sijainti tavussa vaikuttaa bitin arvojen todennäköisuusjakaumaan [12]. Toisaalta tulos myös viittaa siihen, että äärettömän aakkoston CSE-menetelmä on kilpailukykyinen binäärisen menetelmän kanssa.

Työssä esitelty CSE ($\sigma = 256$)-menetelmä koodaa yksittäisen esiintymismäärämatriisin alkion C_{awc}, eli esiintymismäärän, sille laskettujen ala- ja ylärajojen l ja u avulla. Alkion arvon koodaamiseksi riittää kertoa kuinka mones arvo on suljetulla välillä $[l, r]$, eli koodata luku $N_{awc} = C_{awc} - l$. Välin koko määrää montako bittiä luvun N_{awc} koodaamiseen käytetään. Dubén ja Beaudoin CSE-($\sigma = 2$)-menetelmä koodaa luvun N_{awc} käyttäen *adaptiivista aritmeettista koodausta* (engl. *adaptive artihmetic coding*) [37]. Tällöin ei hyödytä siitä, että tunnetaan välin $[l, r]$ koko, mutta se ei välttämättä haittaa.

Dubé ja Beaudoin havaitsevat artikkelissaan, että CSE-menetelmä hyötyy suuremmasta lohkokoosta, eli kerralla käsiteltävän merkkijonon koosta [12]. Pizza&Chili-korpuksen tiedostojen tiivistyvyydestä taulukossa 3.4 voidaan päätellä, että tämä näyttäisi pätevän

myös työssä esitellylle CSE ($\sigma = 256$)-menetelmälle. Tiedostokoon suurentaminen parantaa tiivistyvyyttä, erityisesti 50 MiB ja suuremmalla tiedostokoolla menetelmä pärjää vertailukohteena olevalle Gzip-ohjelmalle. Näillä tiedotoilla CSE-menetelmä pääsee myös lähelle bzip2-ohjelman tiivistystehokkuutta, erityisesti tiedostoilla *english.50MB* ja *english.200MB*.

Vertailun perusteella adaptiivisen aritmeettisen koodauksen käyttöä esiintymismäärien N_{awc} koodaamiseen on syytä tutkia enemmän. Koodaamista kuitenkin vaikeuttaa eri lukujen N_{awc} suuri skaala. Esimerkiksi voidaan valita tiivistettävässä merkkijonossa useimmin esiintynyt symboli $a \in \Sigma$ ja sen esiintymismäärä C_a. Kaikki luvut N_{awc}, kaikilla $w \in \Sigma^*$ ja $a, c \in \Sigma$, ovat suljetulla välillä $[0, C_a]$. Välin koko saattaa kuitenkin olla lähes sama kuin tiivistettävän tekstin pituus n. Binäärisen aritmeettisen koodauksen avulla voitaisiin edelleen käyttää luvun N_{awc} ala- ja ylärajojen tuntemista hyväksi. Tällöin 0 ja 1 bittien jakauman pitäisi kuitenkin olla hyvin epätasainen, ainakin suurilla yhtenäisillä alueilla.

Muihin tiivistysmenetelmiin vertailu, Calgary-korpuksen tiedostoilla taulukossa 3.3, osoitaa että CSE-menetelmä on hyvinkin kilpailukykyinen niiden kanssa. Tässä työssä toteutettu versio ei vertailun mukaan pärjää muille tiivistysmenetelmille, eikä muille CSE-menetelmille. Suuremmilla tiedostoilla, Pizza&Chili-korpuksen taulukossa 3.4, toteutettu CSE-($\sigma = 256$)-menetelmä pärjää Gzip-ohjelman tiivistyvyydelle. Tämä on todennäköisesti sen ansiota, että CSE-menetelmä käsittelee tiivistettävää merkkijonoa kokonaisuudessaan.

Esitelty toteutus tukee 256 symbolin aakkostoa. Mikäli tiivistettävässä tekstissä esiintyy vähemmän eri symboleita, tiivisteeseen koodattavat esiintymismäärämatriisit ovat näennäisesti tarpeettoman suuria tekstin tiivistämiseen. Yksinkertaisuuden vuoksi oletetaan seuraavassa, että matriisit koodataan tiivisteeseen käyttäen ensimmäisenä esitettyä menetelmää. Siinä matriisin alkiot koodataan lähtien ylämmältä riviltä, vasemmanpuoleisimmasta alkiosta. Alkiot luetellaan edeten rivi kerrallaan, vasemmalta oikealle, ylhäältä alas.

Mikäli jokin symboli ei esiinny tiivistettävässä tekstissä, sitä vastaavan rivin ja sarakkeen alkiot ovat aina arvoltaan 0. Silloin myös vastaavat rivi- ja sarakesummat ovat arvoltaan 0. Sekä tiivistäjä että tiivisteen purkaja tuntevat nämä rivi- ja sarakesummat. Lisäksi summat toimivat triviaaleina ylärajoina rivien ja sarakkeiden alkioiden arvolle. Koska yläraja puristaa kunkin alkion arvon suljetulle välille $[0, 0]$, yksikään alkio ei vie tilaa tiivisteessä.

Menetelmässä, jossa esiintymismäärämatriisi koodataan tiivisteeseen rekursiivisesti puolittamalla, tapahtuu vastaava tiivistyminen. Tarkastellaan jakovektoria, jonka jokin alkio vastaa symbolia, joka ei esiinny tiivistettävässä tekstissä. Tällöin myös vektorin isovanhemman, vastaavassa kohdassa sijaitsevan, alkion arvo on 0. Tämä on yläraja jakovektorin alkion arvolle. Vastaavasti kuin ensimmäisessä menetelmässä rivi- ja sarakesummat. Ensimmäisen jakovektorin isovanhempana toimii esiintymismäärämatriisin sarakesummavektori, joka on kummankin osapuolen tiedossa. Siten käyttämättömät symbolit eivät vie ylimääräistä tilaa matriisien koodauksessa.

Ainoa kohta tiivisteessä, jossa tilaa kuluu tarpeettoman suuren aakkoston takia, on alussa oleva luettelo kaikkien aakkoston symboleiden esiintymismääristä. Kokonaisuuden kannalta nämä σ lukua eivät vie oleellisesti tilaa.

Vaihtelevankokoisen aakkoston tukeminen saattaa olla järkevää, jos sillä voidaan nopeuttaa ohjelman toimintaa. Edellä käytiin lävitse matriisin rekursiivisesti puolittavan menetelmän hitautta. Esitettiin, miten esiintymismäärämatriisin käsittely muodostaa suurimman osan tiivistykseen kuluvasta ajasta. On mahdollista, että pienempi matriisi nopeuttaa käsittelyä oleellisesti.

Työssä esitelty CSE-tiivistyksen toteutus ei tue 256 merkin aakkostoa puhtaasti, koska käytetty, kaksisuuntaisen BWT-indeksin toteuttava, kirjasto ei salli symboleiden 00_{16} ja 01_{16} käyttöä tiivistettävässä tekstissä. Indeksi tulisi vaihtaa toteutukseen, joka sallii kaikki symbolit. Toisenlaisesta toteutuksesta saattaa olla hyötyä erityisesti teksteissä, joissa esiintyy kaikkia aakkoston symboleita suunnilleen yhtä paljon. Tällöin symboleiden korvaamisesta toisella symbolilla ei ole hyötyä, ja kiellettyjen symboleiden korvaaminen kahden symbolin mittaisilla merkkijonoilla kasvattaa tekstin pituutta ennen tiivistystä. Aikavaativuuden kannalta korvaus ei ole raskasta, mutta on silti ylimääräinen työvaihe.

Tiivistyvyystaulukoista 3.3 ja 3.4 huomataan miten esiintymismäärämatriisin koodaus tiivisteeseen rekursiivisesti puolittamalla tuottaa huomattavasti tiivistettävää tekstiä suurempia tiivisteitä. Jakovektoreiden alkiot vievät sitä vähemmän tilaa, mitä lähempänä niille lasketut ala- ja ylärajat ovat toisiaan. Tiivistyvyyttä saattaa olla mahdollista parantaa löytämällä uusia tapoja laskea rajoja alkioiden arvolle. On myös mahdollista, ettei tiivisteen purkajalla ole tarpeeksi informaatiota riittävän tiukkojen rajojen laskemiseen, jolloin menetelmällä ei voida saavuttaa hyvää tiivistyvyyttä.

Matriisien koodaustavat eroavat rajojen suhteen siten, että lineaarisesti matriisin alkiot koodaavalla menetelmällä on enemmän informaatiota käytössään. Kun purkualgoritmi käsittelee alkiota C_{awc}, kaikki samalla rivillä sijaitsevat, käsiteltävää alkiota edeltävät alkiot on tunnettuja. Lisäksi samassa sarakkeessa, alkion C_{awc} yläpuolella olevien, alkioiden arvot tunnetaan. Kumpienkin arvojen summaa, eli lukuja S_{21} ja S_{12}, hyödynnetään sekä ala- että ylärajan laskemisessa.

Purettaessa puolittavan koodauksen jakovektorin $\vec{x}$ alkiota x_i, tunnetaan kaikkien vektorin edeltävien alkioiden, $x_j, j < i$, arvot. Tilanne on kuvattu luvun 2.4 kuvassa 2.6. Sen sijaan alkion x_i yläpuolella olevien matriisin, tai edes jakovektorin, alkioiden arvoja ei tunneta. Jakovektorin yläpuoliseen alimatriisiin voidaan edetä rekursiivisesti vasta, kun jakovektorin alkioiden arvot on selvillä. Yläpuolisesta alimatriisista tunnetaan vain kokonaissumma, joka voidaan laskea tunnetun vanhemman $\vec{p}$ avulla.

Jakovektorin alkioille saattaa olla mahdollista kehittää parempia rajoja kokonaan toisenlaisen lähestymistavan kautta. Esimerkiksi artikkelissaan [32] Ota kanssakirjoittajineen parantaa lineaarisesti esiintymismäärämatriisin alkiot koodaavan menetelmän tiiveyttä, järjestelemällä matriisin sarakkeet uudelleen. Vastaavalla uudelleenjärjestely voi toimia matriisia jakavan menetelmän kanssa, jos jakovektoreiden kokonaismassa saadaan painottumaan vektorin alkuun. Myös suuret, pelkistä nolla-alkioista koostuvat, alimatriisit ovat tavoiteltavia, koska niitä ei tarvitse jakaa pienempiin osiin.

Luvun 3.1 kokeissa käsitellään myös alkuperäisen tekstin purkamista tiivisteestä. Tässä työssä toteutettu purkumenetelmä ei hyödynnä tilatiiviitä tietorakenteita, vaan luvussa 2.5

kuvattua tiivistä osamerkkijonoverkkoa. Tiiveys tietorakenteen nimessä tarkoittaa vain, ettei se ole ääretön. Purun muistinkulutus on mitattu Calgary-korpuksen tiedostoille taulukossa 3.11, ja Pizza&Chili-korpuksen pienille tiedostoille taulukossa 3.12. Kaikissa tapauksissa muistinkulutus on moninkertainen verrattuna tiivisteen tai alkuperäisen tekstin kokoon. Suurempia tiedostoja ei edes pystytty purkamaan.

Koe korostaa tarpeen purkumenetelmälle, joka hyödyntää tilatiiviitä tietorakenteita. Purussa on kaksi tietorakennetta, jotka voivat vaatia erityisen paljon tilaa. Toinen niistä on tiivis loppuosaverkko ja toinen purkualgoritmin 9 w_queue-jono. Purkuvaiheessa kaikki alkuperäisen tekstin vasemmalle maksimaaliset osamerkkijonot kulkevat jonon kautta. Jokaista tällaista osamerkkijona varten jonoon lisätään lista, jossa on yhtä monta osoitinta tiiviiseen loppuosaverkkoon, kuin on osamerkkijonoa vasemmalle jatkavia symboleita. Pahimmassa tapauksessa kerralla voidaan lisätä σ, eli aakkoston symboleiden verran, uutta listaa. Jokaisessa listassa on myös mahdollista olla σ alkiota. Keskimääräinen lisäys lienee kuitenkin paljon pienempi. Tietorakenteista tiivis loppuosaverkko viekin todennäköisemmin eniten tilaa: siihen lisätyt osamerkkijonot eivät poistu verkosta ennen purun valmistumista.

Tiivis osamerkkijono on otollisin kohde, jota korvaamaan kannattaa kehittää tilatiivis tietorakenne. Toisaalta CSE-tiivistysmenetelmän purkualgoritmeja ei ole juurikaan tutkittu, joten on mahdollista, että nopea ja vähän muistia kuluttava purkualgoritmi on hyvin erilainen kuin tässä työssä esitetty.

Kirjallisuus

[1] J. Alanko. *BD_BWT_index*. URL: https://github.com/jnalanko/BD_BWT_index (viitattu 20. 05. 2020).

[2] D. Belazzougui ja F. Cunial. "Space-Efficient Detection of Unusual Words". Teoksessa: *String Processing and Information Retrieval*. Toim. C. Iliopoulos, S. Puglisi ja E. Yilmaz. Cham: Springer International Publishing, 2015, s. 222–233. ISBN: 978-3-319-23826-5.

[3] D. Belazzougui, F. Cunial, J. Kärkkäinen ja V. Mäkinen. "Versatile succinct representations of the bidirectional Burrows-Wheeler transform". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8125 LNCS (2013), s. 133–144. ISSN: 03029743. DOI: 10.1007/978-3-642-40450-4_12.

[4] M. Béliveau ja D. Dubé. "Improving compression via substring enumeration by explicit phase awareness". Teoksessa: *Data Compression Conference Proceedings*. Institute of Electrical ja Electronics Engineers Inc., 2014, s. 399. ISBN: 9781479938827. DOI: 10.1109/DCC.2014.68.

[5] M. Burrows ja D. J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Tekninen raportti. Digital Equipment Corporation, 1994.

[6] J. G. Cleary ja W. J. Teahan. "Unbounded Length Contexts for PPM". *The Computer Journal* 40.2_and_3 (tammikuu 1997), s. 67–75. ISSN: 0010-4620. DOI: 10.1093/comjnl/40.2_and_3.67. URL: https://doi.org/10.1093/comjnl/40.2_and_3.67.

[7] M. Crochemore, F. Mignosi ja A. Restivo. "Automata and forbidden words". *Information Processing Letters* 67.3 (1998), s. 111–117. ISSN: 0020-0190. DOI: https://doi.org/10.1016/S0020-0190(98)00104-5. URL: http://www.sciencedirect.com/science/article/pii/S0020019098001045.

[8] L. P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. RFC Editor, toukokuu 1996. URL: http://www.rfc-editor.org/rfc/rfc1951.txt.

[9] D. Dubé. "Lossless compression of grayscale and colour images using multidimensional CSE". *International Symposium on Image and Signal Processing and Analysis, ISPA* 2019-Septe (2019), s. 222–227. ISSN: 18492266. DOI: 10.1109/ISPA.2019.8868744.

[10] D. Dubé. "On the use of stronger synchronization to boost compression by substring enumeration". *Data Compression Conference Proceedings* (2011), s. 454. ISSN: 10680314. DOI: 10.1109/DCC.2011.58.

[11] D. Dubé. "Using synchronization bits to boost compression by substring enumeration". *ISITA/ISSSTA 2010 - 2010 International Symposium on Information Theory and Its Applications* (2010), s. 82–87. DOI: 10.1109/ISITA.2010.5649565.

[12] D. Dubé ja V. Beaudoin. "Lossless data compression via substring enumeration". *Data Compression Conference Proceedings* (2010), s. 229–238. ISSN: 10680314. DOI: 10.1109/DCC.2010.28.

[13] D. Dubé ja H. Yokoo. "The universality and linearity of compression by substring enumeration". *IEEE International Symposium on Information Theory - Proceedings* 5 (2011), s. 1519–1523. ISSN: 21578104. DOI: 10.1109/ISIT.2011.6033796.

[14] P. Elias. "Universal Codeword Sets and Representations of the Integers". *IEEE Transactions on Information Theory* 21.2 (1975), s. 194–203. ISSN: 15579654. DOI: 10.1109/TIT.1975.1055349.

[15] P. Ferragina ja G. Manzini. "Opportunistic data structures with applications". Teoksessa: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. Marraskuu 2000, s. 390–398. DOI: 10.1109/SFCS.2000.892127.

[16] P. Ferragina ja G. Navarro. *Pizza&Chili Corpus*. URL: http://pizzachili.dcc.uchile.cl/ (viitattu 28. 05. 2020).

[17] F. S. Foundation. *Using the GNU Compiler Collection (GCC)*. 2019. URL: https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/ (viitattu 28. 05. 2020).

[18] J.-l. Gailly ja Free Software Foundation. *GNU Gzip*. 2018. URL: https://www.gnu.org/software/gzip/manual/gzip.html.

[19] S. Gog, T. Beller, A. Moffat ja M. Petri. "From Theory to Practice: Plug and Play with Succinct Data Structures". Teoksessa: *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 2014, s. 326–337.

[20] R. Grossi, A. Gupta ja J. S. Vitter. "High-Order Entropy-Compressed Text Indexes". *Technical Report* 2068 (2003), s. 841–850. DOI: 10.5555/644108.644250.

[21] Henry Schreiner ja kanssakehittäjät. *CLI11*. URL: https://github.com/CLIUtils/CLI11 (viitattu 20. 05. 2020).

[22] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. Joulukuu 2017, s. 1605. URL: https://www.iso.org/standard/68564.html.

[23] K.-i. Iwata ja M. Arimura. "Lossless Data Compression via Substring Enumeration for k-th Order Markov Sources with a Finite Alphabet". *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E99.A.12 (2016), s. 2130–2135. ISSN: 0916-8508. DOI: 10.1587/transfun.E99.A.2130. URL: https://www.jstage.jst.go.jp/article/transfun/E99.A/12/E99.A_2130/_article.

[24] S. Kanai, H. Yokoo, K. Yamazaki ja H. Kaneyasu. "Efficient Implementation and Empirical Evaluation of Compression by Substring Enumeration". *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E99.A.2 (2016), s. 601–611. ISSN: 0916-8508. DOI: 10.1587/transfun.E99.A.601. URL: https://www.jstage.jst.go.jp/article/transfun/E99.A/2/E99.A_601/_article.

[25] Martin Hořeňovský ja kanssakehittäjät. *Catch2*. URL: https://github.com/catchorg/Catch2 (viitattu 20. 05. 2020).

[26] Y. Mori. *libdivsufsort*. URL: https://github.com/y-256/libdivsufsort (viitattu 28. 05. 2020).

[27] V. Mäkinen, D. Belazzougui, F. Cunial ja A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015. DOI: 10.1017/CBO9781139940023.

[28] V. Mäkinen ja G. Navarro. "Rank and select revisited and extended". *Theoretical Computer Science* 387.3 (2007), s. 332–347. ISSN: 03043975. DOI: 10.1016/j.tcs.2007.07.013.

[29] T. Ota ja H. Morita. "On a universal antidictionary coding for stationary ergodic sources with finite alphabet". *Proceedings of 2014 International Symposium on Information Theory and Its Applications, ISITA 2014* C (2014), s. 294–298.

[30] T. Ota ja H. Morita. "On antidictionary coding based on compacted substring automaton". *IEEE International Symposium on Information Theory - Proceedings* (2013), s. 1754–1758. ISSN: 21578095. DOI: 10.1109/ISIT.2013.6620528.

[31] T. Ota ja H. Morita. "Two-dimensional source coding by means of subblock enumeration". *IEEE International Symposium on Information Theory - Proceedings* (2017), s. 311–315. ISSN: 21578095. DOI: 10.1109/ISIT.2017.8006540.

[32] T. Ota, H. Morita ja A. Manada. "Compression by Substring Enumeration with a Finite Alphabet Using Sorting". Teoksessa: *2018 International Symposium on Information Theory and Its Applications (ISITA)*. C. IEEE, lokakuu 2018, s. 555–559. ISBN: 978-4-88552-318-2. DOI: 10.23919/ISITA.2018.8664360. URL: https://ieeexplore.ieee.org/document/8664360/.

[33] M. Powell, T. Bell, J. Horlor ja R. Arnold. *The Canterbury Corpus*. URL: http://corpus.canterbury.ac.nz/ (viitattu 28. 05. 2020).

[34] T. Schnattinger, E. Ohlebusch ja S. Gog. "Bidirectional Search in a String with Wavelet Trees". Teoksessa: *Combinatorial Pattern Matching*. Toim. A. Amir ja L. Parida. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 40–50. ISBN: 978-3-642-13509-5.

[35] J. Seward. *bzip2 and libbzip2, version 1.0.8*. 2019. URL: https://www.sourceware.org/bzip2/manual/manual.html (viitattu 01. 06. 2020).

[36] Valgrind Developers. *Valgrind User Manual*. 2020. URL: https://valgrind.org/docs/manual/manual.html (viitattu 28. 05. 2020).

[37] I. H. Witten, R. M. Neal ja J. G. Cleary. "Arithmetic Coding for Data Compression". *Commun. ACM* 30.6 (kesäkuu 1987), s. 520–540. ISSN: 0001-0782. DOI: 10.1145/214762.214771. URL: https://doi.org/10.1145/214762.214771.

[38] D. Vohl ja D. Dub. "Finding Synchronization Codes for Compression by Substring Enumerations" (). arXiv: arXiv:1605.08102v1.

[39] H. Yokoo. "Asymptotic optimal lossless compression via the CSE technique". *Proceedings - 1st International Conference on Data Compression, Communication, and Processing, CCP 2011* (2011), s. 11–18. DOI: 10.1109/CCP.2011.32.

[40] J. Ziv ja A. Lempel. "A universal algorithm for sequential data compression". *IEEE Transactions on Information Theory* 23.3 (1977), s. 337–343. ISSN: 1557-9654 VO - 23. DOI: 10.1109/TIT.1977.1055714.