Datan pakkausohjelma

Teemu Pitkänen

Helsinki 7.3.2015

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Teemu Pitkänen

Työn nimi — Arbetets titel — Title

Datan pakkausohjelma

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Aineopintojen harjoitustyö	7.3.2015	0 sivua + 16 liitesivua

Tiivistelmä — Referat — Abstract

Tässä dokumentissa esitellään aineopintojen harjoitustyönä toteutettu kolmitasoinen pakkausohjelma. Ohjelma pohjautuu kolmeen hyvin tunnettuun pakkaus- ja muunnosmenetelmään. Pakattavaa dataa käsitellään ensin Burrows-Wheeler- ja move to front -muunnoksilla, minkä jälkeen varsinainen pakkaus tapahtuu Huffman-symbolikoodauksella.

Varsinainen ohjelma on ladattavissa osoitteesta https://github.com/teempitk/2015-periodi-3 .

Avainsanat — Nyckelord — Keywords
pakkaus, Huffman-koodaus, move to front, Burrows-Wheeler -muunnos

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Määrittelydokumentti

Projektin tavoitteena oli toteuttaa Huffman-koodaukseen pohjautuva pakkausohjelma. Ohjelma saa syötteenä suoritettavan operaation (pakkaus/purku) ja pakattavan tiedoston nimen. Ohjelma käyttää muunnos- ja pakkausmenetelmiä parametritiedostolle, ja tallentaa tuloksen tiedostoon, jonka nimi on sama kuin alkuperäinen jatkettuna .teemuzip-päätteellä. Alkuperäinen tiedosto pysyy ennallaan.

Huffman-koodaus on ominaisuuksiensa vuoksi tehokas erityisesti tekstimuotoiselle datalle. Pakkaustehokkuuden parantamiseksi entisestään ohjelmaa laajennettiin vielä toteuttamalla datalle Burrows–Wheeler -muunnos ja move to front -muunnos ennen varsinaista symbolikoodausta. Ohjelman kompressiopino on siis kolmitasoinen:

1. Burrows–Wheeler -muunnos
2. move to front -muunnos
3. Huffman-koodaus

Kompressiopinoon valitut menetelmät sopivat projektin aiheeseen hyvin, sillä ne vaativat monipuolisesti erilaisten algoritmien ja tietorakenteiden toteuttamista ja soveltamista. Projektissa toteutetut keskeisimmät tietorakenteet ovat:

- Linkitetty lista (`MyLinkedList`)
- Hajautustaulu (`CodewordDictionary`)
- Huffman-puu (`HuffmanTree`)

Jokainen kompressiopinon elementti on jo itsessään oma algoritminsa, mutta niiden toteutus vaati monien pienempien ongelmien ratkaisemista. Ohjelmassa on alirutiineina toteutettu mm. seuraavat algoritmit:

- quicksort (Burrows–Wheeler -muunnoksen purussa)
- string quicksort (Burrows-Wheeler -muunnoksessa)
- insertion sort (Huffman-puun rakennuksessa, käytettäessä `MyLinkedList`iä järjestettynä `insertPreservingOrder`-metodilla)

Pakkausohjelmille on tärkeää toimia tehokkaasti niin tila- kuin aikavaativuuksien suhteen, koska syötteet ovat usein hyvin suuria. Käytetyt menetelmät toimivat pääosin lineaarisessa ajassa, mistä poikkeuksena Burrows-Wheeler -muunnos, joka vaatii keskimäärin $\mathcal{O}(n \log n)$-ajassa toimivia järjestämisalgoritmeja. Tarkemmin aikavaativuuksia on eriteltu toteutusdokumentissa.

2 Toteutusdokumentti

2.1 Ohjelman rakenne

Ohjelman rakenne on todella yksinkertainen, ja isojen linjojen toimintalogiikka on pitkälti Main-luokassa, joka tulkitsee annettujen parametrien kelpoisuuden, ja kutsuu pakkausoperaatioita tekeviä metodeja. Jokainen kolmesta pakkaus- ja muunnosoperaatioluokasta tarjoaa operaationsa staattisena metodina, joita Main-luokka kutsuu. Seuraavissa osioissa kuvataan yksityiskohtaisemmin eri menetelmien sisäistä toteutusta.

2.2 Pakkaus

2.2.1 Burrows-Wheeler -muunnos (BWT)

Datalle ensimmäisenä tehty Burrows-Wheeler -muunnos ei pienennä datan kokoa lainkaan, vaan itseasiassa kasvattaa sitä neljällä tavulla, ja muunnoksen idea pakkauksessa onkin pelkästään parantaa muiden menetelmien tehokkuutta. Muunnoksen jälkeen datassa esiintyy yleensä enemmän peräkkäisiä saman tavun toistoja. Muunnoksen voi ajatella toimivan seuraavalla tavalla. Datasta muodostetaan ensin matriisi, jonka jokaisella rivillä dataa on "kierretty"yksi pykälä eteenpäin. Jos alkuperäinen data olisi siis "banana", matriisi olisi seuraava:

$$\begin{bmatrix} \mathrm{b} & \mathrm{a} & \mathrm{n} & \mathrm{a} & \mathrm{n} & \mathrm{a} \\ \mathrm{a} & \mathrm{n} & \mathrm{a} & \mathrm{n} & \mathrm{a} & \mathrm{b} \\ \mathrm{n} & \mathrm{a} & \mathrm{n} & \mathrm{a} & \mathrm{b} & \mathrm{a} \\ \mathrm{a} & \mathrm{n} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{n} \\ \mathrm{n} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{n} & \mathrm{a} \\ \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{n} & \mathrm{a} & \mathrm{n} \end{bmatrix}$$

Seuraavaksi matriisin rivit järjestetään akkosjärjestykseen:

$$\begin{bmatrix} a & b & a & n & a & n \\ a & n & a & b & a & n \\ a & n & a & n & a & b \\ b & a & n & a & n & a \\ n & a & b & a & n & a \\ n & a & n & a & b & a \end{bmatrix}$$

Muunnettu data on tämän matriisin viimeinen sarake, ja datan oikeassa järjestyksessä sisältävän rivin numero, tässä siis "nnbaaa", 3. Tässä projektissa muunnos toteutettiin Wikipedian [1] artikkelin mukaisesti. Käytännön toteutuksissa datan koon suhteen neliöllistä matriisia ei tietenkään toteuteta kokonaan, vaan kunkin matriisin rivin voi esittää sanan ensimmäisen tavun indeksinä alkuperäisessä datassa.

Datan luku tiedostosta ja matriisia kuvaavien osoittimien alustus ovat luonnollisesti luokkaa $\mathcal{O}(n)$ niin aika- kuin tilavaativuudeltaan. Matriisin rivien järjestys tapahtuu [3] kuvatulla *string quicksort*illa. Pitkien merkkijonojen (tai tässä tavutaulukoiden) normaali pikajärjestäminen ei toimi $\mathcal{O}(n \log n)$-ajassa, sillä kahden merkkijonojen vertaaminen ei aina onnistu yhdellä operaatiolla, vaan voi vaatia jopa merkkijonon pituuden verran vertailuja. Merkkijonoille muokattu quicksort pitää rekursiossa mukana myös vertailun kohteena olevien merkkijonojen *pisimmän yhtenevän alkuosan* (longest common prefix, lcp) pituuden, jolloin vertailu osataan aloittaa merkkijonon järkevästä indeksistä. Tällöin quicksortin aikavaativuus saadaan merkkijonoille toteutumaan ajassa $\mathcal{O}(n \log n) + \Sigma LCP$. Tässä ΣLCP tarkoittaa summaa järjestetyn matriisin kunkin rivin yhteisestä etuliitteestä ylläolevaan riviin, siis *banana*-esimerkissä $0 + 1 + 3 + 0 + 0 + 2 = 6$. Satunnaisissa ja käytännön tapauksissa aakkosto (erilaisten esiintyvien tavujen määrä) on yleisesti suuri, jolloin yhteiset alkuosat ovat suhteellisen lyhyitä, ja aikavaativuus on käytännössä $\mathcal{O}(n \log n)$. Toteutettu algoritmi vaatii kussakin rekursion tasossa lineaarisen määrän tilaa, ja rekursion keskimääräinen syvyys on $\mathcal{O}(n)$, joten myös tilavaativuus on tässä $\mathcal{O}(n \log n)$, jota voisi luonnollisesti optimoida vielä paremmaksi.

Muunnetun datan lukeminen järjestetystä matriisista on nyt $\mathcal{O}(n)$-operaatio, ja samoin datan kirjoitus tiedostoon. Toteutetun Burrows–Wheeler -muunnoksen peräkkäisistä operaatioista raskain on siis matriisin rivien pikajärjestäminen, joka vie aikaa ja tilaa keskimäärin $\mathcal{O}(n \log n)$. Muunnetun tiedoston alkuun lisätään nyt vielä alkuperäisen datan viimeisen tavun osoitin. Tämä tallennetaan 32-bittisenä etumerkillisenä kokonaislukuna (int), joten suurimmat tiedostot, joille pakkaus voidaan tehdä ovat kooltaan $2^{31} - 1 \approx 2$ gigatavua.

2.2.2 Move to front -muunnos (MTF)

Move to front [8] ei vaikuta datan kokoon lainkaan, mutta on oleellinen Burrows–Wheelerin hyödyntämisessä Huffman-koodaukseen. Kuten edellä kuvattiin, BWT ei muuta datassa esiintyviä tavuja, vaan pelkästään niiden järjestystä. Jos tälle muunnetulle datalle ajettaisiin Huffman-koodaus, muunnoksella ei olisi mitään vaikutusta pakatun datan kokoon, sillä symbolikoodauksessa sama merkki korvattaisiin samalla bittijonolla sijainnista riippumatta. (Itseasiassa datan koko kasvaisi hieman lisätystä osoittimesta johtuen).

MTF:ssa luodaan ensin kaikki mahdolliset 256 tavua sisältävä linkitetty lista, niin että kukin tavu on (etumerkitöntä) arvoaan vastaavassa indeksissä. Dataa käsitellään tavu kerrallaan, ja kukin tavu korvataan ensin tavun indeksillä edellä kuvatussa listassa, minkä jälkeen tavu siirretään listan ensimmäiseksi. MTF hyötyy nyt BWT:n tuottamista saman tavun toistoista, sillä kaikki toiston tavut ensimmäistä lukuunottamatta korvataan nollatavulla. Yleisesti tekstissä usein esiintyvät tavut korvataan pienillä luvuilla, joita esiintyy muunnetussa datassa paljon. Huffman-koodaus puolestaan hyötyy tästä tavujen epätasaisesta jakaumasta.

Esimerkiksi, sanassa *banana* esiintyvien merkkien alkuindeksit listassa ovat

merkki	numeroarvo / indeksi
a	97
b	98
n	110,

ja muunnettu data olisi nyt 98, 98, 110, 1, 1, 1.

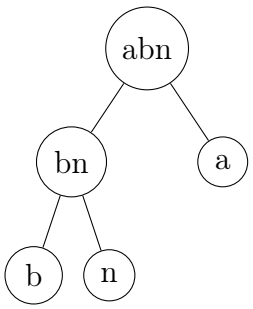
Tavulistan luonti alussa vaatii aikaa ja tilaa $\mathcal{O}(m)$, missä m on aakkoston koko, tässä projektissa kaikkien erilaisten tavujen määrä eli 256 (vakio). Dataa muuntaessa tavun indeksin selvitys listassa vaatii aikaa $\mathcal{O}(m)$, ja se tehdään jokaiselle (n kpl) tavulle. Operaation aikavaativuus on siis $\mathcal{O}(mn) = \mathcal{O}(256 \cdot n) = \mathcal{O}(n)$. Muunnos vaatii tilaa vain tavulistan ja muunnetun datan tallentamiseen, joten tilavaativuus on $\mathcal{O}(m) + \mathcal{O}(n) = \mathcal{O}(n)$.

2.2.3 Huffman-koodaus

Huffman-koodaus on optimaalinen symbolikoodaus datalle, jossa kukin tavu saa sitä pidemmän koodisanan, mitä harvinaisempi se on koodattavassa tekstissä. Huffman-

koodaukselle erityistä on koodisanojen luontitapa, muuten kyseessä on täysin tavallinen symbolikoodaus, jossa sama tavu korvataan aina samalla bittijonolla sijainnista riippumatta.

Koodauksen aluksi kunkin tavun esiintymismäärä täytyy laskea lähdedatassa, johon kuluu aikaa $\mathcal{O}(n)$ ja tilaa $\mathcal{O}(m)$. Tämän jälkeen luodaan lista, jossa tavut (niitä vastaavat Huffman-puun solmut) ovat esiintymismääriensä mukaisesti kasvavassa järjestyksessä. Puuta rakennetaan siten, että listasta poistetaan toistuvasti kaksi ensimmäistä (harvinaisinta) alkiota, luodaan uusi solmu, jonka esiintymismääräksi asetetaan poistettujen yhteenlaskettu esiintymismäärä, poistetut solmut tämän lapsiksi, ja laitetaan tämä uusi solmu oikealle paikalleen listaan. Toisto loppuu, kun listassa on enää yksi solmu. Esimerkkisanalle banana saadaan siis seuraava puu:



solmu	esiintymismäärä	koodisana
a	3	1
b	1	00
n	2	01

Koodisanat saadaan luettua puusta yksinkertaisesti rekursion avulla. Puuta läpikäydään juuresta alkaen, ja rekursion parametrina annetaan aina kertynyt koodisana. Siirryttäessä rekursiossa solmun lapseen, kertyneen koodisanan loppuun lisätään 0 jos siirrytään vasempaan lapseen, ja 1 jos siirrytään oikeaan lapseen. Lopulta, kun rekursio pääsee lehteen asti, lehteä vastaavan tavun koodisana on juuri rekursiossa kertynyt bittijono.

Puuta rakentaessa kahden harvinaisimman solmun poistaminen listan alusta ja uuden solmun luominen ovat vakioaikaisia. Uuden solmun sijoittaminen listaan vaatii kuitenkin $\mathcal{O}(m)$-ajan, sillä linkitetyssä rakenteessa ei voida etsiä paikkaa binäärihakutyylisesti, vaan lisääminen perustuu lisäysjärjestämiseen. Edellä kuvattu toiminta poistaa listasta kaksi alkiota ja lisää takaisin yhden, joten kullakin iteraatiolla listan solmujen määrä vähenee yhdellä, ja toistoja tarvitaan siis $m - 1$ kappaletta. Kokonaisaikavaativuus puun rakentamiselle on siis $\mathcal{O}(m^2)$. Tässä toteutuksessa m on aina korkeintaan 257 (kaikkien erilaisten tavujen määrä ja tiedoston loppumerkki), joten puun rakennus on itseasiassa vakioaikaista.

Koodisanoja puusta luettaessa puun läpikäynnissä kussakin solmussa käydään vain

kerran. Huffman-puu on aina täysi, joten solmuja on kaikkiaan $2m - 1$ kappaletta, siis aikavaativuus on $\mathcal{O}(2m - 1)$, ja tässä toteutuksessa siis jälleen vakioaikaista. Luetut koodisanat tallennetaan taulukkoon, josta kutakin tavua vastaava koodisana saadaan vakioajassa, kun tavua käytetään taulukon indeksinä. Kun siis suoritetaan varsinainen datan symbolikoodaus, kunkin datan tavun koodaus vie vakioajan, ja kokonaisaikavaativuus on siten $\mathcal{O}(n)$.

Koska Huffman-koodauksen tuottamat koodisanat ovat erilaiset pakattavasta tiedostosta riippuen, myös koodisanat täytyy tallentaa pakattuun tiedostoon, jotta pakkaus voidaan myös purkaa. Tässä ohjelmassa Huffman-pakatun tiedoston koko rakenne on seuraava:

1. Tiedot koodauksesta

koodinpituus(0)	koodi(0)	koodinpituus(1)	koodi(1)	koodinpituus(2)	...

...	koodi(255)	koodinpituus(256)	koodi(256)	koodinpituus(EOF)	koodi(EOF)

2. Varsinainen data koodattuna (tässä '+' tarkoittaa koodisanojen konkatenointia ja data[i] pakattavan datan tavua indeksissä i)

koodi(data[0])+koodi(data[1])+...+koodi(data[data.length-1])

3. Loppuosa

koodi(EOF)	täyttö tasatavuihin 0-tavuilla

2.3 Purku

Purkuvaihessa operaatiot täytyy suorittaa pakkaukseen nähden käänteisessä järjestyksessä. Käytetyt menetelmät ovat asymmetrisiä, eli purkuoperaatiot ovat erilaiset kuin pakkauksen vastaavat.

2.3.1 Huffman-koodaus

Huffman-koodauksen purku tapahtuu nyt täysin normaalin symbolikoodauksen tapaan. Ensin tiedoston alusta täytyy lukea koodisanat, joka tapahtuu toistamalla 257 kertaa seuraava proseduuri:

```
1. for tavu = 0 to 256
2.     koodinpituus = lue seuraavat 8 bittiä
```

```
3.      koodisana = lue seuraavat koodinpituus bittiä
4.      lisää (koodisana, tavu) -pari CodewordDictionaryyn
```

Koodisanojen määrä ja maksimipituus ovat vakioita, joten koko koodauksen tulkinta tiedoston alusta on vakioaikaista. Kun koodisanat ovat hajautustaulussa, datan purku tapahtuu pelkistetysti seuraavalla algoritmilla

```
1. while pakattua dataa jäljellä
2.      lue seuraava bitti puskuriin
3.      if (hajautustaulu sisältää puskurin bittijonon)
4.          aseta vastaava tavu puretun datan loppuun
5.          tyhjennä puskuri
```

Algoritmin while-silmukka suoritetaan selvästi $\mathcal{O}(n)$ kertaa. Silmukan lohkon operaatiot ovat kaikki keskimäärin vakioaikaisia, joten myös koko koodauksen purun aikavaativuus on $\mathcal{O}(n)$.

2.3.2 Move to front -muunnos

Move to front on menetelmistä samankaltaisin pakkaus- ja purkuvaiheessa. Myös purkuvaiheessa kaikki tavut alustetaan ensin listaan, mutta nyt toimitaan pakkausvaiheeseen nähden käänteisesti – luetaan ensin "oikean"tavun indeksi pakatusta tiedostosta, luetaan tavu listasta indeksin mukaisesti, ja siirretään sitten tavu taas listan alkuun ja lisätään se puretun datan loppuun. Kuten pakkauksessa, alustustoimenpiteet vaativat $\mathcal{O}(m) = \mathcal{O}(256)\mathcal{O}(1)$ ajan ja tilan. Dataa käsitellessä läpikäydään linkitettyä m-alkioista tavulistaa n-kertaa, ja kokonaisaikavaativuus on, kuten pakkauksessa, $\mathcal{O}(mn) = \mathcal{O}(n)$.

2.3.3 Burrows–Wheeler -muunnos

Burrows-Wheeler -muunnoksen purku on toteutettu kuten [2]:ssa Muistetaan, että pakattu data vastasi BWT-matriisin viimeistä saraketta. Koska matriisin jokainen sarake sisältää datan kaikki symbolit, ja matriisin rivit olivat aakkosjärjestyksessä, saamme ensimmäisen sarakkeen yksinkertaisesti järjestämällä muunnetun datan tavut. Viimeisen sarakkeen lukeminen on tietenkin lineaarinen operaatio, ja ensimmäinen sarake saadaan pikajärjestämällä ajassa $\mathcal{O}(n \log n)$. Aiemmin käytetyssä *banana*-esimerkissä tiedämme nyt siis:

$$\begin{bmatrix} a & ??? & n \\ a & ??? & n \\ a & ??? & b \\ b & ??? & a \\ n & ??? & a \\ n & ??? & a \end{bmatrix}$$

Koska sanat kiertävät matriisin riveillä "ympäri", saamme kaikki alkuperäisessä datassa esiintyneet kahden tavun parit nyt yhdistämällä (rivin viimeinen tavu)+(rivin ensimmäinen tavu). Esimerkissämme tämä tuottaa siis tavuparit na, na, ba, ab, an, an. Tavuparit saadaan oikeaan järjestykseen havaitsemalla, että tavun t i:s esiintymä ensimmäisessä sarakkeessa vastaa saman tavun i:nnettä esiintymää viimeisessä sarakkeessa – saman tavun esiintymien keskinäinen järjestys on kussakin sarakkeessa sama!

Datan purkamiseksi luodaan nyt indeksointi kunkin tavun esiintymispaikoista viimeisessä sarakkeessa. Tämä saadaan 256-paikkaisena taulukkona linkitettyjä listoja. Sarake käydään kertaalleen läpi, oikea lista löydetään vakioajassa käyttäen tavun arvoa indeksinä, ja esiintymän indeksi lisätään listan loppuun. Listoja on vakiomäärä 256 kappaletta, ja niihin asetetaan n linkitetyn listan alkiota, joten tilavaativuus on $\mathcal{O}(n + 256) = \mathcal{O}(n)$.

Indeksointia ja edellä mainittua vastaavuusominaisuutta käyttäen luodaan nyt n-paikkainen taulukko "seuraajat"(tilavaativuus $\mathcal{O}(n)$), jonka indeksissä i on ensimmäisen sarakkeen i:nnen alkion sijainti-indeksi viimeisessä sarakkeessa. Havaitaan itseasiassa, että tästä seuraa suoraan, että tavua seuraava tavu on *ensimmäisen* sarakkeen arvo samassa indeksissä. Tämä taulukko saadaan yksinkertaisesti toistamalla seuravaa: lue 1. sarakkeen i:s tavu ($\mathcal{O}(1)$), etsi oikea lista edellä luodusta indeksoinnista käyttäen tavun arvoa indeksinä ($\mathcal{O}(1)$), lisää luotavaan taulukkoon indeksiin i listan ensimmäinen alkio ja poista se samalla listasta.Vakioaikaisia operaatioita toistetaan n kertaa, joten vaiheen aikavaativuus on $\mathcal{O}(n)$.

Luoduilla rakenteilla alkuperäinen data saadaan nyt purettua helposti – muistetaan, että pakattuun dataan lisättiin myös osoitin, joka kertoo missä indeksissä alkuperäisen datan viimeinen merkki on viimeisessä sarakkeessa. Havaitaan, että alkuperäisen datan ensimmäinen merkki sijaitsee samassa indeksissä ensimmäisessä sarakkeessa. Nyt toistetaan vain seuraavaa:

```
1. indeksi = viimeisen tavun sijainti viimeisessä sarakkeessa
```

```
2. for i = 0 to data.length-1
3.     lisää purettuun dataan ensimmäinen_sarake[indeksi]
4.     indeksi = seuraajat[indeksi]
```

Operaation aikavativuus on selvästi lineaarinen datan kokoon nähden.

2.4 Jatkokehitysideoita

Ohjelman loppulinen toteutus saavuttaa halutut aikavaativuusluokat, mutta isommilla tiedostoilla pakkaus ja purku venyvät jopa kymmeniin sekunteihin. Aikavaativuusluokkatason parannusta on oikeastaan mahdoton saada, mutta käytännössä suuria eroja näkyi syntyvän esimerkiksi eri tiedoston luku- ja kirjoitusmenetelmiä käyttämällä. Etenkin lineaarisessa ajassa toimivaa Huffman-koodausta ja -purkua saisi varmasti optimoitua paljon nopeammaksi.

Burrows-Wheeler -muunoksen toteutus rajaa pakattavan datan kokoa, ja rajan olisi sinänsä voinut muuttaa isommaksikin. Tällä hetkellä en ole myöskään ohjelmoinut näihin tapauksiin mitään varoitusta, joten ohjelma aiheuttanee virheen jos liian ison tiedoston pakkausta kokeilee.

Kuten pakkausohjelmissa yleensä, toteutettavia menetelmiä voi aina lisätä aiempien perään, ja esimerkiksi alkuviikoilla suunnittelemani run length encoding jäikin lopulta toteuttamatta, ja voisi olla mukava toteuttaa vielä pakkaustehokkuuden parantamiseksi.

3 Käyttöohje

Ohjelman jar-tiedosto löytyy kohteesta `tiralabra/dist/tiralabra.jar`. Ajettaessa ohjelmalle annetaan:

- Parametri -c tai -d, eli pakataanko (compress) vai puretaanko (decompress). Toinen ja vain toinen näistä on aina annettava.
- Vapaaehtoisesti parametri -p (print), joka tulostaa tietoa suorituksen etenemisestä.
- Pakattavan tiedoston nimi.

Komennon yleisrunko on siis

```
> java -jar tiralabra.jar [halutut optiot joukosta {-c,-d,-p}] tiedoston_nimi,
```

ja pakkauksen esimerkkikomento voisi siis olla esimerkiksi:

```
> java -jar tiralabra.jar -c -p  ../sampleFiles/alice.txt
```

ja purkukomento esimerkiksi

```
> java -jar tiralabra.jar -d -p ../sampleFiles/alice.txt.teemuzip
```

Jos tulostusoptio on annettu, ohjelma tulostaa tietoa etenemisestään, edellä mainittu pakkauksen esimerkkikomento voi tulostaa esimerkiksi:

```
> java -jar tiralabra.jar c ../sampleFiles/alice.txt
Compression started.
Phase 1/3: Burrows-Wheeler transform started ...    Finished. Time: 0,369 sec.
Phase 2/3: Move to front transformation started ... Finished. Time: 0,286 sec.
Phase 3/3: Huffman encoding started ...             Finished. Time: 0,137 sec.
Compression finished. Total time: 0.818899 sec.
File size reduced from 167518 bytes to 55788 bytes (33,3% of original size).
```

4 Testausdokumentti

4.1 JUnit-testaus

Projektin automatisoitu testaus koostuu 92 JUnit-testistä. En käyttänyt mitään varsinaista testikattavuutta mittaavaa työkalua, mutta pyrin testaamaan silti toteutusta mahdollisimman kattavasti. Toteutin pakkaus- ja muunnosmenetelmille yksisuuntaisia testejä pieniin ja helposti käsin laskettaviin esimerkkeihin, ja toteutin myös edestakaisen oikeellisuuden testaavat testit sampleFiles-hakemiston tiedostoille alice.txt ja turing.jpg.

Tietorakenteiden testauksessa pyrin ottamaan mahdollisimman hyvin huomioon kaikki erikoistapaukset, joita ohjelman suoritus voisi aiheuttaa. Tässä testaus osoittautui todella hyödylliseksi, ja auttoi hyvin löytämään ohjelmointivirheitä korvatessa javan valmiita tietorakenteita omilla toteutuksilla.

4.2 Empiirinen pakkaustehokkuuden testaus

Testasin myös ohjelman pakkaustehokkuutta olemassaoleviin tekstitiedostoja tehokkasti pakkaaviin ohjelmiin. Vertailuun valikoituivat Lempel-Ziv -koodaukseen pohjautuva gzip [7] ja bzip2 [6], joka sisältää kaikki tässä projektissa toteutetut välivaiheet, mutta niiden lisäksi myös muita. Käytetyt testitiedostot löytyvät `tiralabra/sampleFiles`-hakemistosta.

- `turing.jpg` - Keskikoon jpg-pakattu kuvatiedosto [9]
- `alice.txt` - Alice's adventures in Wonderland Project Gutenbergista [4]
- `U00096.2.fas` - Escherichia coli -bakteerin genomi [5]

Alkuperäisten ja kullakin menetelmällä pakattujen tiedostojen koot löytyvät allaolevasta taulukosta.

	turing.jpg	alice.txt	U00096.2.fas
alkuperäinen	51183	167518	4697740
teemuzip	51185	55788	1330270
gzip	50722	60636	1424418
bzip2	50773	49037	1328000

Kuten oli odotettavissa, mikään menetelmistä ei saavuta merkittävää tulosta tiedostolle `turing.jpg`. Jo valmiiksi pakattu tiedosto sisältänee varsin heterogeenisesti kaikkia tavuja, eikä pakkauksessa hyödynnettävää säännöllisyyttä ole. Tiedosto oli otettu tähän lähinnä osoittamaan, ettei pakkauksen käyttö ei-optimaaliseen tiedostotyyppiinkään tuota mielivaltaisen suurta pakattua tiedostoa. Kuten havaitaan, "huonon"tiedoston pakkaus voi myös kasvattaa tiedoston kokoa, kuten toteuttamalleni ohjelmalleni tässä käy, onneksi kuitenkin hyvin vaatimattoman kahden tavun verran.

Tekstimuotoista dataa sisältävien tiedostojen suhteen ohjelmani vaikuttaa sijoittuvan pakkaustehokkuudeltaan gzip:n ja bzip2:n välimaastoon, ja sama tulos onkin nähtävillä myös yllä esitetyissä testitapauksissa. Luonnollista kieltä sisältävä data pakkautuu noin kolmannekseen alkuperäisestä koostaan, kun taas lähinnä neljää merkkiä (T, A, G, C) sisältävä genomidata jopa hieman paremmin.

4.3 Empiirinen suorituskyvyn testaus

Toteutusdokumentaatiossa operaatioille todettiin olevan käytännön tapauksissa seuraavat aikavaativuudet:

	pakkaus	purku
BWT	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
MTF	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Huffman	$\mathcal{O}(n)$	$\mathcal{O}(n)$

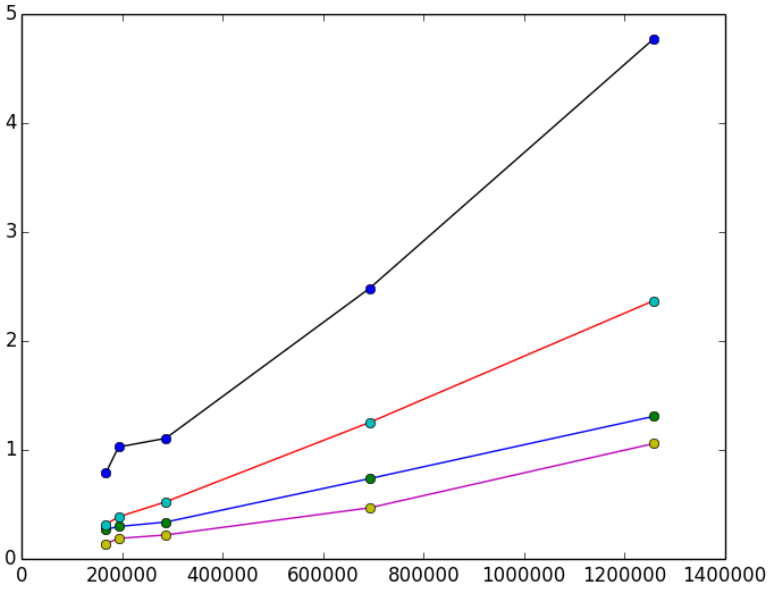
Ensinnäkin on todettava, että tämän pakkausohjelman tapauksessa pakkausaikaan vaikuttaa vahvasti myös muut tekijät pelkän tiedoston koon lisäksi. BWT hyötyy nopeudessa mahdollisimman satunnaisesta datasta, sillä BWT-matriisin rivien vertailu järjestämisessä ratkeaa todennäköisesti sitä aiemmin, mitä satunnaisempi data. Toisaalta esimerkiksi MTF hyötyy tasan päinvastaisesta tapauksesta, ja toimii nopeimmin jos koko data on samaa tavua (tavu asetetaan ensimmäisen tavun käsittelyn jälkeen tavulistan alkuun, eikä sitä tarvitse enää sen jälkeen etsiä listasta). On myös huomattava, että ajallisen suorituskyvyn etu ei välttämättä ole pakkaustehokkuuden etu, kuten edellä mainitussa BWT-esimerkissä (satunnaisessa datassa ei ole pakkauksessa hyödynnettävää säännöllisyyttä, ja pakkaustehokkuus on huono).

Kun pakkausohjelmalle annetaan ajettaessa parametri `-p`, ohjelma tulostaa tietoa suorituksen etenemisestä ja kuhunkin vaiheeseen kuluneesta ajasta. Pakataan nyt aluksi seuraavat tiedostot:

- `alice.txt`
- `hamlet.txt` William Shakespeare: Hamlet [14]
- `odysseia.txt` Homeros: Odysseia [11]
- `seitseman.txt` Aleksis Kivi: Seitsemän veljestä [10]
- `mobydick.txt` Herman Melville: Moby Dick [13]

Saatiin seuraavat tulokset:

Kulunut aika (s)	alice (167 518 B)	hamlet (193 083 B)	odysseia (287 062 B)	seitseman (692 967 B)	mobydick (1 257 294 B)
BWT	0,271	0,297	0,338	0,737	1,308
MTF	0,317	0,386	0,524	1,253	2,369
Huffman	0,139	0,188	0,220	0,469	1,058
Yhteensä	0,787231	1.02652	1.107054	2.484259	4.770644

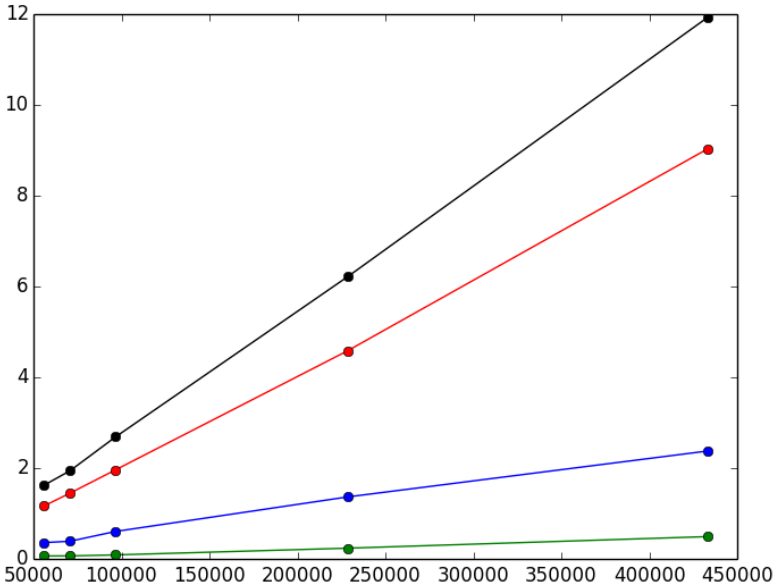


Kuva 1: Pakkausaika. Violetti = Huffman-koodaus, sininen = BWT, punainen = MTF. Pakkauksen kokonaisaika mustalla, edellämainittujen tiedostojen koot vaaka-akselilla.

Kuten kuvaajasta 1 havaitaan, MTF:n ja Huffman-koodauksen lineaarinen aikavaativuus toteutuu testissä kuten pitääkin. Burrows-Wheeler näkyy toimivan tässä testissä jopa "paremmin kuin pitäisi", ja näyttää jopa lineaariselta. Kuitenkin, kuten edellä mainittiin, pakkauksen nopeuteen vaikuttavat monet muutkin tekijät. Lisäksi ohjelma tulostaa koko suoritusajan, jolloin esimerkiksi laitteiston muu kuormitus vaikuttaa tuloksiin.

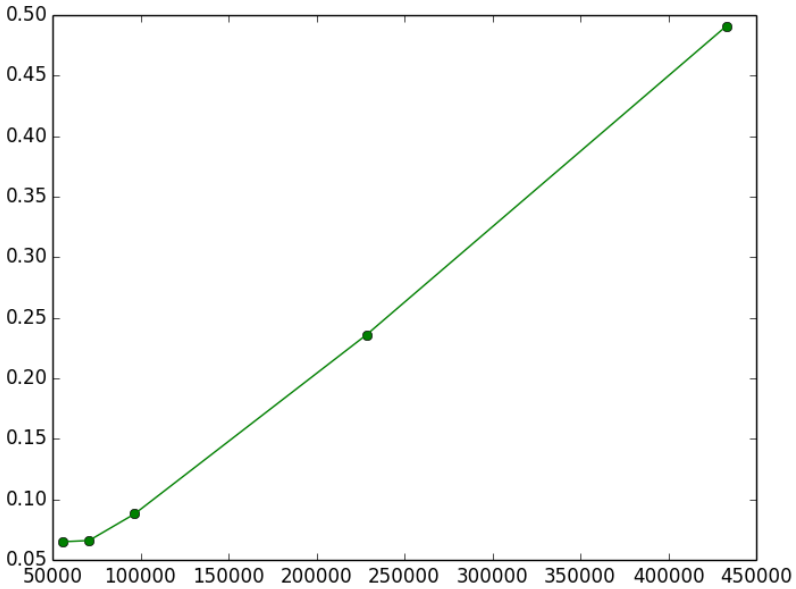
Vastaavasti purkua testattaessa saatiin seuraavat tulokset:

Kulunut aika (s)	alice (55 788 B)	hamlet (70 785 B)	odysseia (96 683 B)	seitseman (228 538 B)	mobydick (433 095 B)
Huffman	1,163	1,449	1,962	4,584	9,029
MTF	0,358	0,389	0,605	1,365	2,376
BWT	0,065	0,066	0,088	0,236	0,491
Yhteensä	1.617612	1.942284	2.688281	6.214634	11.925905



Kuva 2: Purkuaika. BWT vihreällä, MTF sinisellä, Huffman punaisella ja kokonaisaika mustalla. Pakatun tiedoston koko vaaka-akselilla.

Kuten kuvasta 2 nähdään, Huffman-koodaus ja MTF vaikuttavat noudattavan lineaarista aikavaativuutta. BWT:tä täytyy tarkastella erikseen, sillä sen käyttämä aika on tässä niin pieni, että vaikutus hukkuu. Kuvaajasta 3 voidaan nähdä aavistus $\mathcal{O}(n \log n)$-operaation suoritusajan loivasti kiihtyvästä kasvusta.



Kuva 3: Burrows-Wheeler -muunnoksen purkuaika

Lähteet

1 http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

2 Spencer Carroll "An Easy to Understand Explanation of the Burrows Wheeler Transform"http://spencer-carroll.com/an-easy-to-understand-explanation-of-the-burrows-wheeler-transform/

3 Juha Kärkkäinen
String Processing Algorithms, Kurssimateriaali (syksy 2014).
http://www.cs.helsinki.fi/u/tpkarkka/teach/14-15/SPA/lecture02.pdf

4 Project Gutenberg's Alice's Adventures in Wonderland, by Lewis Carroll
http://www.gutenberg.org/cache/epub/11/pg11.txt

5 University of Wisconsin-Madison
Escherichia coli K-12 MG1655 complete genome
http://www.genome.wisc.edu/pub/sequence/U00096.2.fas

6 http://en.wikipedia.org/wiki/Bzip2

7 http://fi.wikipedia.org/wiki/Gzip

8 http://en.wikipedia.org/wiki/Move-to-front_transform

9 http://en.wikipedia.org/wiki/Alan_Turing

10 The Project Gutenberg EBook of Seitsemän veljestä, by Aleksis Kivi
http://www.gutenberg.org/cache/epub/11940/pg11940.txt

11 The Project Gutenberg EBook of Odysseus, the Hero of Ithaca, by Homer
http://www.gutenberg.org/files/24856/24856-0.txt

12 The Project Gutenberg EBook of The King James Bible
http://www.gutenberg.org/cache/epub/10/pg10.txt

13 The Project Gutenberg EBook of Moby Dick; or The Whale, by Herman Melville
http://www.gutenberg.org/cache/epub/2701/pg2701.txt

14 Project Gutenberg Etext of Hamlet by Shakespeare
http://www.gutenberg.org/cache/epub/1524/pg1524.txt