Mika Huttunen

Toteutusdokumentti

Ohjelman yleisrakenne

Ohjelman yleisrakenne on yksinkertainen ja selkeä. Se sisältää pakkaukset “Apuvalineet”, “Huffman”, “LZW” ja “Tietorakenteet”. “Huffman” ja “LZW” sisältävät kyseisten algoritmien vaadittavan toiminnallisuuden.
Apuvalineet sisältää mm. luokat tiedoston lukemiseen, kirjoittamiseen ja binäärimuunnoksiin, Tietorakenteet taas minimikeon ja hajautustaulun.
Näiden lisäksi on olemassa testipakkaukset ja -luokat, jotka on nimetty samalla lailla.

Saavutetut aika- ja tilavaativuudet

Lyhenne ‘M’ = “aihemäärittelyssä”

Huffman

Pakkaaminen
Aikavaativuus: O(n) M: O(n)
Tilavaativuus: O(n * log(m)) M: O(m)

Tilavaativuus pakatessa on suurempi, sillä ennen tekstin kirjoittamista muodostetaan kirjoitettava teksti kokonaisuudessaan kelaamalla alkuperäinen teksti läpi ja lisäten alkuperäisen tekstin jokaista merkkiä vastaava “bittijono” (muotoa: 00 01 01 00 01 00…) StringBuilder -oliolle (tilavaativuus: O(n * log(m))).

Purkaminen
Aikavaativuus: O(n)
Tilavaativuus: O(n)

Purun yhteydessä tehtäviä operaatioita ovat pakkaukseen kirjoitetun Huffman puun läpikäynti (aika- ja tilavaativuus O(m), missä m on merkkien lkm pakkaamattomassa tiedostossa), pakkauksen sisällön lukeminen (aikavaativuus: O(n), missä n sen sisällön koko) ja “binääritekstin” (00 01 01 01 00 00…) muodostus (aikavaativuus: O(n), tilavaativuus: O(n)). Näiden operaatioiden lisäksi sekä pakatessa että purettaessa kirjoitetaan tekstiä, joka on aikavaativuusluokkaa O(n).

Syy siihen, miksi pakatessa ei päästä optimaaliseen tilavaativuuteen johtuu siitä, että koodissa ei käsitellä bittijonoja kokonaislukuina, vaan String -formaatissa ja mitä enemmän erilaisia merkkejä tiedostossa esiintyy, sitä suuremmiksi bittijonot kasvavat.

Lempel-Ziv-Welch

Pakkaaminen
Aikavaativuus: O(n)
Tilavaativuus: O(n * log(m))

Pakkaamisen aikavaativuus on luokkaa O(n), sillä pakatessa luetaan teksti läpi ja kirjoitetaan se. Tilavaativuus on O(n * log(m)), sillä binääriteksti (00 01 01..), joka muutetaan ascii -koodiksi kirjoittamista varten, on tilavaativuudeltaan tätä suuruusluokkaa. “log (m)” tulee siitä että siihen lisättävien bittijonojen pituuden kasvu on logaritminen ja kasvaa aina kun lisätään aakkostoon (= hajautustaulu) uusi alkio, jota vastaava arvo on suurempi kuin mitä sillä hetkellä käytetyllä “bittipituudella” voidaan esittää.

Purkaminen
Aikavaativuus: O(n)
Tilavaativuus: O(n * log(m))

Purettaessa pätee samat suuruusluokat kuin pakatessakin (ja “log (m)” tilavaativuuskin johtuu täysin samasta syystä). Purettaessa käytetään enemmän tilaa kuin pakattaessa, sillä kirjoitettava teksti on luonnollisesti pidempi ja purkamiseen käytetään yhtä hajautustaulua enemmän kuin pakkaamiseen. Tämä ei kuitenkaan suuruusluokkaan vaikuta.

Suorituskykyvertailua -ja testausta

9.6. klo 12:00

Alkuperäinen teksti tyyppiä:

“Ensimmäinen runo

Mieleni minun tekevi, aivoni ajattelevi
lähteäni laulamahan, saa'ani sanelemahan,
sukuvirttä suoltamahan, lajivirttä laulamahan.
Sanat suussani sulavat, puhe'et putoelevat,
kielelleni kerkiävät, hampahilleni hajoovat.”

Alkuperäisen tekstin koko: 550kt
Tekstin pakkaamisen viemä aika vaihtelee n. 2s - 3s.
Pakatun tekstin koko: 432kt
Purkamisen viemä aika: 2.2s
Puretun tekstin koko: 556kt

Tyyppiä:

“Ensimmäinen runoMieleni minun tekevi, aivoni ajatteleviläheteäni laulamahan, saa'ani sanelemahan,sukuvirttä suoltamahan, lajivirttä laulamahan.Sanat suussani sulavat, puhe'et putoelevat,kielelleni kerkiävät, hampahilleni hajoovat.”

10.6. klo 16:20

Nyt tiedoston pakkaamisen ja purun jälkeen alkuperäisteksti näyttää tekstieditorilla täysin samalta kuin alkuperäinenkin. Ero kuitenkin löytyy: ohjelmani koodaa ääkköset (eli tässä tapauksessa merkit “Ä, Ö, ä ja ö” 2 tavun unicode -formaatissa alkuperäisen tekstin 1 tavun formaatin sijaan). Tästä johtuen kun alkuperäistekstin koko on n. 550kt, pakkaamisen ja purun jälkeen koko on 571kt. Muita ongelmia tässä ei näyttäisi tapahtuvan.

Pakatun tiedoston koko oli nyt 448kt ja tilankäyttö alkuperäiseen tekstiin verraten 81% ja purettuun tiedostoon 78%.

Huffman

Pakkaaminen

Tekstin lukeminen pakkaamisen alussa vie pienellä syötteellä (3kt) 200ms ja isolla syötteellä (550kt) n. 1200ms.

Huffman -puun sekä puun määrittämien merkkien bittiesitysten muodostaminen on nopea operaatio ja vie kaikenkokoisilla syötteillä vain vähän aikaa (enimmillään n. 10ms). Suurin kokeilemani tiedosto oli kalevalan .txt versio ja sillä tämä vei max. 7ms (puu) + 1ms (bittiesitykset).

Pakattava sisällön muodostaminen viei suhteellisen paljon aikaa. Pienillä syötteillä (< 70kt) pakkausaika on alle 150ms ja isommilla syötteillä (550kt) yli 600ms. Tämä käyttää selvästi enemmän aikaa kuin muut ohjelman osat, joten testaan sen metodien käyttämää suoritusaikaa kattavammin ja selvitän, mikä luokan metodeista vie eniten aikaa.

Pakattavan sisällön kirjoittaminen ei vienyt aikaa kovinkaan paljon. Suurilla syötteillä (550kt) aikaa kuluu hieman yli 30 millisekuntia.

Kokonaisajat pakatessa:

lorem ipsum (67kt)	n. 700ms
kalevala (550kt)	n. 1900ms

Pakkaamisen kokonaisaika riippuu siis suureksi osaksi tekstin lukemiseen sekä pakatun sisällön muodostamiseen kuluvasta ajasta. Tekstin lukua voi olla hankala nopeuttaa, sillä se toimii suhteellisen hyvin optimoidulla yksinkertaisella algoritmilla, mutta pakatun sisällön muodostaminen sen sijaan on monivaiheinen operaatio, mikä voi sisältää hitaita osuuksia.

Pakatun sisällön muodostaminen

Koko suoritusaika kuluu käytännössä sen pakkauksen osan muodostamiseen, joka sisältää ainoastaan tekstin ascii -merkkeinä. Tähän operaatioon kuluu kaksi osaa, joista toinen vie kalevala.txt:n kanssa suoritusajasta n. 140ms ja toinen n. 500ms.

Nopeampi näistä operaatioista on pakkauksen tekstin kelaaminen merkki merkiltä läpi, ottaen omatekoisesta hajautustaulusta merkkiä vastaava avain ja lisäten tämä avain StringBuilder-oliolle.
Hitaampi operaatio taas käyttää tästä nopeammasta operaatiosta saatua StringBuilderin sisältöä (11001…) ja kelaa sitä 8 merkin pätkinä muodostaen jokaisesta pätkästä ascii -merkin ja lisäten tämän toisella StringBuilder -oliolle. Operaatio on todennäköisesti hidas muutamien “turhien” algoritmien takia, jotka jouduin koodaamaan, koska Javaa käyttämällä oli ongelmia tehdä saada aikaan ascii -merkki mistä tahansa kokonaisluvusta (“(char) int” saattoi palauttaa jotain ihan muuta kuin ohjelman toiminnan kannalta olisi pitänyt).

Purkaminen ja hajautus

Pakkauksen purku koostuu käytännössä kahdesta vaiheesta; kirjoitettavan tekstin muodostamisesta pakkauksen sisällön pohjalta sekä ko. tekstin kirjoittamisesta. Kirjoittaminen ei kuitenkaan ollut pitkä operaatio (kuten em.), joten keskityn tässä ainoastaan kirjoitettavan tekstin muodostamiseen ja selvittämään, mitkä operaatiot sen muodostettaessa vievät eniten aikaa.

Tekstin muodostus sen sijaan vei todella paljon aikaa omaa hajautustauluani käyttäen. Toisaalta kun kokeilin vastaavaa Javan HashMap -tietorakennetta käyttäen, pakkaaminen sujui huomattavasti nopeammin.

Kokonaisajat omalla hajautustaululla (arvot 11.6. klo 21:00) sekä Javan HashMap:illä.

Pakkaus	HajautusTaulu	HashMap
huffman.txt.hemi	250ms	140ms
lorem ipsum.txt.hemi	2000ms	660ms
kalevala.txt.hemi	17000ms	3000ms

(Seuraavia kokeiluja varten käytän omaa tämän hetkistä hajautustauluani.)

Tekstin muodostaminen on nelivaiheinen prosessi;
1) pakkausen sisällön luku
2) sisällön “Huffman -puu” -osan läpikäynti
3) pakkausen teksti -osan (String teksti = 010110…) muodostaminen vastaavasta tavukoodista
4) kirjoitettavan tekstin muodostus kohtien 2) ja 3) avulla.

Tekstin luku on nopea prosessi (kestää kalevalan kohdalla alle 300ms) ja puun läpikäynti ei vie läheskään edes sitä vertaa (< 20ms).

Kohta 3 vie huomattavasti enemmän aikaa (lorem ipsum vie noin 300ms ja kalevala hieman alle 2s). Se ei käytä hajautustaulua mihinkään ja sen käyttämä aika on lineaarisesti suoraan verrannollinen tekstin (011010…) pituuteen pakatussa tiedostossa tavuina.

Hidas operaatio, joka tämän aikana suoritetaan on jokaisen tavun muuntaminen 8-merkkiseksi String -olioksi BinaariMuuntaja -luokkaa hyödyntäen. Jos sen koodaisi toimimaan eri lailla ja tietysti nopeammin, tulisi tässä ajansäästöä.

Kohta 4 taas vei loppuajan, ja koska kolme edeltävää vaiheita ovat nopeita suhteessa tekstin muodostamisessa kuluvaan aikaan ja purkamisen kokonaisaika on huomattavasti pienempi Javan valmista hajautusrakennetta käyttäen, kulutettu aika liittyy oman hajautustauluni hitaampaan toimintaan.

Kohta (4), oma haj.taulu (11.6 klo 21:00)

Pakkaus	HajautusTaulu	HashMap
huffman.txt.hemi	200ms	59ms

lorem ipsum.txt.hemi	1500ms	140ms
kalevala.txt.hemi	15000ms	550ms

12.6 klo 15:30

En tehnyt vielä hajautustauluun muutoksia, mutta aloin testaamaan sitä paremmin. Löysin hajautustaulusta ongelman; hajautustaulu voi olla todella suuri ja sisältää ainoastaan vähän avaimia. Tällöin sekä poisto -että “sisältääAvaimen” operaatiot ovat äärimmäisen hitaita mikäli avain ei ole taulussa. Tällöinhän koko suuri hajautustaulu käydään läpi etsiessä avainta joka ei siellä ole.

Lisäys hajautustauluun on nopea operaatio niin kauan kuin hajautustaulu ei ole todella iso (> 10000 avainta). Tämän jälkeen - vaikka käytetään kaksoishajautusta - syntyy todennäköisesti paljon yhteentörmäyksiä, joka hidastaa paikan etsimistä, minne avain voidaan lisätä.

Kokeilenkin siis seuraavaksi muokata hajautustauluni operaatioita siten että uudelleenhajautus toteutuu aina kun hajautustaulussa on paljon tavaraa (koon kasvatus) tai kun vastaavasti siellä on vain vähän tavaraa (koon pienennys).

15.6 klo 17:20

Käytin viimeiset 2 päivää kirjoittamalla hajautustietorakenteeni kokonaan alusta, jotta saisin sen operaatiot “lisää”, “poista” ja “etsi” toimimaan vakioajassa. Uuteen toteutukseeni käytin apuna verkkoluentoa[1]. Toteutus ei ole aikavaativuudeltaan optimaalinen, mutta se on nyt kuitnekin huomattavasti nopeampi kuin ennen ja toimii lähes ajassa O(1) pienillä syötekoilla (< 10000).

Pakkaus	**HajTaulu**	**HashMap**
huffman.txt.hemi	120ms	59ms
lorem ipsum.txt.hemi	550ms	140ms
kalevala.txt.hemi	1800ms	550ms

Keko

Testasin omaa tietorakennetta “MinKeko” verraten sitä Javan valmiiseen “PriorityQueue”:een. Toteutin testauksen luomalla while -silmukan sisällä uusia solmuja. Ensin 100kpl, sitten

1000kpl, 10000kpl ja lopuksi 100000kpl ja katsoin, miten lisäykseen kuluva aika näillä vaihtelee ja lisäyksen jälkeen poistin samat solmut yksitellen.

Solmujen lisääminen ja poisto / kuluva aika

Lisättyjä solmuja	MinKeko	PriorityQueue
10000	7ms	9ms
100000	20ms	25ms

Poistettuja solmuja	MinKeko	PriorityQueue
10000	24ms	17ms
100000	45ms	34ms

MinKeko tietorakenne näyttäisi toimivan moitteettomasti (myös siten että keko on alussa pieni ja sitä joutuu kasvattamaan, jotta uusia solmuja voidaan siihen lisätä) aikavaativuudessa O(nlogn).

16.6. klo 18:30

Sain viimein tiedoston pakkaamisen ja purkamisen toimimaan tekstitiedostoilla (sekä englannin- että suomenkielinen) oikein. Tämä onnistui käyttäen lukemiseen ja kirjoittamiseen “DataOutputStream” ja “DataInputStream” -olioita ja kävien tekstiä läpi tavu tavulta. Tällä tavoin menee kuitenkin todella paljon aikaa, mikä tässä on selvä haittapuoli. Esimerkiksi “kalevala.txt”:n pakkaaminen vie nyt n. 10s, vaikka ilman näiden käyttöä päästäisiin alle 2s aikaan.

Toiminnan muutos johti lisäksi siihen että pakkaustehokkuus parani huomattavasti, kun käytettiin eri “char encoding” muotoa. Nyt pakkaustehokkuus on usein välillä 50-75%, kun pakattavan tiedoston koko on jo muutamia kilotavuja. Mm. kalevalalle tämä on 56%, lorem ipsumille 54% ja huffmanille 74%.

Lempel-Ziv-Welch

En tehnyt algoritmia varten suorituskykytestausta laisinkaan, mutta algoritmin suorituksen aikavaativuus on optimaalinen (O(n)) ja sekä pakkaamisen että purun “päätoiminnallisuus” sisältyy yhteen while -toistolausekkeeseen.

Algoritmi toimisi nopeammin javan HashMap:iä käyttäen, sillä mm. purkaessa tarvitaan kolme eri hajautustaulua, joiden operaatiot pitäisivät olla vakioaikaisia, mutta ovat tätä ainoastaan keskimäärin.

Bugit ja parannusehdotukset

Lempel-Ziv-Welchin algoritmin toteuttamiseen käytin ainoastaan muutamia päiviä viimeisen työviikon aikana ja en saanut sitä toimimaan oikein vaikka yritystä kyllä oli. Sekä pakkaus- että purkuvaiheessa algoritmin aikana käsitellään bittijonoja, joiden koko kasvaa algoritmin suorituksen aikana sitä mukaa kun “merkkijonojen aakkkosto” täyttyy. Aakkostoon koodataan merkkijonot bittijoina ja kun tulee vastaan tilanne että seuraavaa lisättävää merkkijonoa ei voida esittää tähän mennessä käytetyllä määrällä bittejä, täytyy alkaa käyttää enemmän bittejä. Tämän toteutuksen olen sekä pakkaus- että purkuvaiheeseen tehnyt, mutta se ei toimi oikein.

Ongelmaa olen yrittänyt debugata parhaani mukaan, mutta en ole saanut yksinkertaisesti selville, missä vika tarkalleen ottaen on. Osasyy, siihen että ongelma lopullisessa palautuksessa on, on se että aikaa LZW algoritmin toteuttamiseen oli silti turhan vähän.

Esimerkkejä LZW algoritmin toiminnasta:

Toimii oikein	**Voi pakata ja purkaa, ei toimi**	**Hajoaa purettaessa**
HuffmanCodingTest.txt	lorem ipsum.txt	huffman.txt
kaannokset.txt	kalevala.txt	

Sen lisäksi että em. bugi pitäisi korjata, ohjelman koodia voisi muuttaa käsittelemään “tavutaulukoita” (byte array), sillä Kirjoittaja -luokan olio“DataOutputStream” muuttaa saamansa merkkijonon tavuiksi kirjoittamista varten, mikä kuluttaa merkittävän osan ohjelman suoritusajasta. En ollut tästä alun perin tietoinen, joten päätin esittää bittijonot String -formaatissa, kun ohjelmaani kirjoitin.

Tämän lisäksi omatekoisen hajautustaulun toimintaa voisi parantaa mm. uudelleenhajautuksella sen sijaan että tällä hetkellä hajautustalun sisältämiä String -taulukoita kasvatetaan tarvittaessa (keskim. O(1)).

Lähteet

[1] Todd Wittman, Lecture 23: Implementing A Hash Table, http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec23.pdf