Tietorakenteiden ja algoritmien harjoitustyö: Huffman-koodaus

Aaro Lehikoinen

06.12.2013

1 Määrittely

1.1 Huffman-koodaus

Huffman-koodaus on tiedon pakkaamiseen käytettävä algoritmi. Algoritmin idea on kartoittaa aakkoston eri merkkien esiintymisfrekvenssit ja koodata jokainen merkki mahdollisimman lyhyellä bittijonolla siten, että eniten käytetty merkki saa lyhyimmän bitti-ilmauksen ja vähiten käytetty pisimmän.

1.2 Algoritmin yksityiskohdat

Pakkauksen ensimmäinen askel on pakattavan datan aakkoston merkkien esiintymisfrekvenssien selvittäminen. Tämä vaatii merkkien esiintymien laskemisen datasta, siis koko pakattavan datan läpikäynnin. Tämän jälkeen rakennetaan Huffman-puu. Aluksi jokaista merkkiä käsitellään puun lehtenä, jonka arvona on sen esiintymisfrekvenssi. Valitaan ensin kaksi harvinaisinta merkkiä, eli pienimmät arvot omaavaa lehteä, ja tehdään niille emosolmu, jonka arvoksi tulee lastensa arvojen summa. Luotua solmua kohdellaan samoin kuin alkuperäisiä solmuja. Lapsiksi alennetut solmut eivät enää ole käsiteltävien solmujen listassa. Jatkamme samaa poimien aina kaksi pienimmät arvot omaavaa solmua ja luomalla niille emon, kunnes jäljellä on vain yksi solmu, jonka arvo on kaikkien merkkien esiintymistiheyksien summa eli 1. Tuloksena on puu, jonka lehtisolmuina ovat kaikki aakkoston merkit.

Merkkien koodaus on on bittijono, joka saadaan kulkemalla juuresta solmuun. Kuljettaessa aina oikealle lisätään bittijonoon 1 ja vasemmalle 0. Tällöin solmuun päästessä ollaan saatu bittijono, joka kertoo merkin bittiesityksen pakatussa datassa. Itse pakkaaminen tehdään käymällä data läpi merkki kerrallaan ja kirjoittamalla aina merkin koodaus tiedostoon.

Näin pakatun datan purkaminen käy rekonstruoimalla puu ja lukemalla pakattua dataa bitti kerrallaan. Lähtemällä liikkeelle juuresta ja valitsemalla aina luetun bitin mukainen siirtymä, eli 1-bitillä oikea ja 0-bitillä vasen, löydetään lehti, jossa on etsimämme merkki. Lehteen päästäessä tulostetaan löydetty merkki, siirrytään takaisin juurisolmuun ja jatketaan datan lukemista.

1.3 Tehokkuus

Koodauksen voi siis jakaa helposti vaiheisiin

1. Lue datan aakkosto ja laske merkkien esiintymisfrekvenssit
2. Rakenna Huffman-puu
3. Etsi kunkin merkin bittiesitys puusta
4. Pakkaa data

Merkitään jatkossa n = datan koko ja k = aakkoston koko. Periaatteessa k on vakio.

1.3.1 Esiintymisfrekvenssien selvitys

Ensimmäisen kohdan aikavaativuus on $O(n)$, koska koko data käydään kertaalleen läpi ja jokaisen luetun merkin kohdalla kasvatetaan sen laskuria yhdellä. Oletuksena, että merkistö on ennalta tiedossa (esim. aakkoset tai tavut) ja tietorakenne, jossa lukumäärät muistataan on valmiiksi rakennettu ja sen alkioiden arvojen muuttaminen tapahtuu vakioajassa. Käytännössä toteutus voi olla yksinkertainen taulukko, jossa on jokaiselle merkille alkio, jonka arvo on 0. Taulukon alustamiseen menee aakkoston koon verran aikaa. Vaiheen tilavaativuus on $O(k)$ olettaen, että datasta on muistissa kerrallaan vain yksi merkki.

Pseudokoodiesitys:

```
lukumaarat = []

while tiedostoa jaljella
  m = lue merkki
  lukumaarat[m]++
```

1.3.2 Huffman-puun rakentaminen

Toinen kohta voidaan jakaa vaiheisiin

a. Järjestä solmut

b. Poimi kaksi pienintä solmua

c. Luo uusi emosolmu

d. Palaa vaiheeseen a, jos solmuja jäljellä

ja esittää pseudokoodina näin:

```
while solmut.lukumaara > 1
  solmut.jarjesta()
  oikea=solmut.poistapienin()
  vasen=solmut.poistapienin()
  emo=uusisolmu(oikea+vasen)
  emo.vasenlapi=vasen
  emo.oikealapsi=oikea
  solmut.lisaa(emo)
```

Silmukka suoritetaan kunnes solmuja on jäljellä yksi. Koska jokaisessa suorituksessa solmuista poistetaan kaksi ja lisätään yksi, suoritetaan silmukka kerran solmua kohden, eli merkkien lukumäärän verran. Tästä nähdään myös, että syntyvässä puussa on noin tuplasti solmuja alkuperäiseen puuhun, eli akkostoon, verrattuna. Koska silmukka suoritetaan vain k-1 kertaa, lopullinen solmujen lukumäärä on $k + k - 1 = 2k - 1$. Tämä johtuu siitä, että *solmut*-joukosta poistettavat solmut ovat kuitenkin muodostuvan puun solmuja. Silmukan sisällä tapahtuvat operaatiot järjestä, poistapienin ja lisää. Nämä kaikki toimivat ajassa $O(log\ k)$ jos solmut säilytetään minimikeossa. Keon järjestäminen tarvitsee tehdä vain kerran ennen silmukkaan astumista, koska keko-operaatiot poistapienin ja lisää pitävät keon järjestyksessä suorittamalla korjaavat toimenpiteet ajassa $O(log\ k)$. Täytyy muistaa, että näiden operaatioiden aikavaativuus on verrannollinen aakkoston (k), ei pakattavan datan kokoon (n). Aakkoston koko on usein huomattavasti pienempi, muussa tapaukssa pakkaaminen ei edes ole hyödyllistä. Käsiteltävä puu voi kuitenkin olla pahasti epätasapainossa. Puun rakennusvaiheessa korkeus kasvaa aina kun toinen kahdesta solmusta kuuluu jo puuhun. Tällaisessa tapauksessa puun korkeus kasvaa yhdellä. Jokaisella silmukan suorituskerralla voi käydä näin, jos yhteenlaskettava arvo on aina pienempi kuin seuraavaksi pienin arvo. Tällöin jokaisen solmun arvo on aina kaksinkertainen edeltävän solmun

arvoon verrattuna. Esimerkiksi $arvo_i = 2^i$. Täten tällaisessa pahimmassa tapauksessa puun korkeudeksi tulee $k - 1$. Tilaa algoritmi vie eniten silmukan viimeisellä suorituskerralla, jolloin uusia solmuja on luotu $k - 1$ kappaletta. Tilavaativuutena siis $O(k)$.

1.3.3 Koodausten etsiminen puusta

Kolmannessa vaiheessa kuljetaan kaikki reitit juuresta lehtiin läpi löytääksemme koodaukset. Tämä on toteutettavissa yksinkertaisesti pinoa ja syvyyssuuntaista hakua käyttämällä.
Pseudokoodiesitys bittijonot selvittävälle läpikäynnille:

```
pino.push(huffpuu.juuri)

while not empty pino
  solmu = pino.pop()
  if solmu on lehti
    koodit[solmu.merkki]=solmu.koodaus
    continue //takaisin silmukan suorituksen alkuun

  if solmu has left child
    vasen=solmu.vasen
    vasen.koodaus=solmu.koodaus+0
    pino.push(vasen)

  if solmu has right child
    oikea=solmu.oikea
    oikea.koodaus=solmu.koodaus+0
    pino.push(oikea)
```

Pinon push- ja pop-operaatiot toimivat vakioajassa. Pinoon laitetaan aina kertaalleen jokainen solmu, joten silmukka suoritetaan niin monta kertaa kuin solmuja on. Kuten aiemmin todettiin, solmuja on $2k - 1$ kappaletta. Aikavaativuus siis on $O(k)$. Aiemmin todettiin myös, että puun maksimikorkeus on $k - 1$. Syvyyssuuntaisen läpikäynnin tilavaativuus riippuu pinossa kerrallaan olevien solmujen määrästä. Korkeimmalla paikalla olevaa lehteä tarkasteltaessa on siis kaikki polulla olevat solmut pinossa. Tilavaativuuskin on siis $O(k)$.

1.3.4 Pakkaus

Neljäs vaihe, datan pakkaus, voidaan esittää pseudokoodina:

```
while tiedostoa jaljella
  m=lue merkki
   pakattudata+=koodit[m]
```

Tämän vaiheen silmukka suoritetaan n-kertaa ja kaikki silmikassa suoritettavat operaatiot ovat vakioaikaisia. Pakkausvaiheen aikavaativuus on siis $O(n)$.

1.4 Purkaminen

Pakatun datan purkaminen tapahtuu saman Huffman-puun avulla kuin pakkaaminen. Puussa kuljetaan aina juuresta kohti lehteä jokaisen luetun bitin kohdalla. Aikavaativuutena luonnollisesti $O(n)$. Pseudokoodiesitys:

```
while tiedostoa jaljella
  b=lue bitti
  if bitti=1
    solmu=solmu.oikea
  if bitti=0
    solmu=solmu.vasen
  if solmu on lehti
    data+=solmu.data
    solmu=juuri
```

1.5 Yhteenveto

Koko algoritmin suorittamiseen menee siis käytännössä $O(n)$ aikaa, koska hitaimmat vaiheet kuuluvat tähän luokkaan.

1.6 Lähteet

`http://fi.wikipedia.org/wiki/Huffmanin_koodaus`
`http://en.wikipedia.org/wiki/Huffman_coding`

2 Toteutus

2.1 Tehokkuus

Toteutukseni ohjelmalogiikka on noudattaa määrittelyssä esiteltyä vaiheitusta.

2.1.1 Pakkaus

Metodi

- read_bytes käy tiedoston kerran läpi ja lukee taajuudet taulukkoon. $O(n)$
- load_heap rakentaa minimikeon, joka toteuttaa prioriteettijonon. $O(k\ log\ k)$
- build_huffman_tree rakentaa puun. $O(k\ log\ k)$
- huffman_codes käy rakennetun puun läpi. $O(k)$
- write_header ja write_data pakkaavat datan ja kirjoittavat datat tiedostoon. $O(n)$

2.1.2 Purkaminen

Metodi

- read_header lukee otsakkeen. $O(k)$
- rebuild_tree rakentaa puun uudestaan otsakkeen tiedoista. $O(k)$
- read_data lukee ja purkaa datan $O(n)$

3 Testaus

Kaikille metodeille toteutettiin kattavat yksikkötestit ja suorituskykytestit käyttäen modifioitua MinUnit (`http://www.jera.com/techinfo/jtns/jtn002.html`) yksikkötestauskehystä. Testit generoivat satunnaisia syötteitä ja käyttävät ennalta määriteltyjä syötetiedostoja. Suorituskykytesteissä vaiheet ajetaan usealla erikokoisella syötteellä ja ajoaikoja vertaillaan.

Lisäksi testattiin käsin pakkaustehoa, joka osoittautui tekstitiedostoilla erittäin hyväksi (noin 50% 30 kilotavun lähdekoodipaketilla). Validitytest.sh-skriptiä käytettiin ennen yksikkötestien perusteellista toteuttamista oikeellisuuden tarkastelemiseksi.

Suorituskykytestit antoivat odotettuja tuloksia. Keko-operaatioiden aikavaativuus oli $O(log\ n)$ ja pakkaamisen ja purkamisen aikavaativuus luokkaa $O(n)$.