Ohjelman yleisrakenne

Ohjelmassa on kolme eri tiedostonpakkausalgoritmia, Huffmanin koodaus, LZW jotka toimivat niin kuin pitääkin ja joiden suorituskyky on myöskin ihan hyvä. Lisäksi ohjelmassa on pahasti kesken oleva minimal-viable-product LZ77. .

Huffmanin koodauksen olen toteuttanut käyttäen hyväksi hajautustaulua ja prioriteettijonoa. Hajautustaulun teossa on myös käytettyhyväksi ArrayList tietorakennetta. Nämä kaikki olen tehnyt itse ja vain niin laajana että Huffmanin koodaus toimisi. Lisäksi Huffmanin koodaus tarvitsee toimiakseen Huffmanin puun, jonka olen toteuttanut Node luokan avulla. Huffmanin koodauksen toiminta perustuu siihen että merkit joita tiedostossa on tiheämmin koodataan binäärikoodilla joka vie vähemmän tilaa kuin koodi jolla koodataan merkit joita on tiedostossa harvemmin.

Huffmanin koodauksessa pakatessa käytettään pääasiassa luokkaa ”HuffmanCompression”. Ensin käydään teksti läpi ja lisätään kaikkien merkkien esiintyvyydet hajautustauluun. Tästä hajautustaulusta muodostettaan prioriteettijonon avulla Huffmanin puu. Puu käydään läpi ja merkitään jokaista mahdollista merkkiä vastaava Huffmanin koodi hajautustauluun. Tämän jälkeen kirjoitettaan merkkien esiintyvyydet pakatun tiedoston alkuun. Sitten käydään alkuperäinen tiedosto läpi merkki merkiltä ja kirjoitettaan merkkejä vastaavat koodit hajautustaulusta pakattuun tiedostoon. Pakkauksen suoritus on nyt valmis.

Huffmanin koodauksessa purettaessa käytettään pääasiassa luokkaa ”HuffmanDecompression”. Ensin luetaan pakatun tiedoston alkuosasta merkkien esiintyvyydet hajautustauluun. Tästä hajautustaulusta muodostettaan prioriteettijonon avulla Huffmanin puu, samalla tavalla kuten tiedostoa pakatessa. Tämän jälkeen luetaan pakatun tiedoston loppuosa läpi ja puretaan Huffmanin koodi edellä mainitun puun avulla.

LZW algoritmissa olen toteuttanut käyttäen hyväksi hajautustaulua ja ArrayListiä, jotka olen itse toteuttanut niin laajana että LZW toimisi.

LZW algoritmin pakkaus käyttää luokkaa ”LZWCompression”. LZW tiedoston pakkauksessa aluksi alustetaan sanakirja hajautustaulu kaikilla 256 ascii merkillä ja niitä vastaavalla koodilla. Sitten tämän jälkeen käydään tiedosto läpi, jos sanakirja sisältää wc(missä c on luettumerkki) niin w=wc, muuten koodi ArrayListaan lisätään w vastaava koodi sanakirjasta ja sanakirjaan lisätään wc ja uusi w = c. Tämän jälkeen kirjoitettaan koodisto tiedostoon 24 bittiä kerrallaan.

LZW algoritmin purkaus käyttää luokkaa ”LZWDecompression”. Purkaminen sujuu melkein samalla tavalla kuin pakkaaminen. Ensin luetaan tiedostoa 24 bittiä kerallaan ja tästä saadaan tätä vastaava koodi joka lisätään ArrayListaan. Seuraavaksi alustetaan sanakirja kuten pakatessakin. Nyt voidaan koodit purkaa sanakirjan avulla.

LZ77 toiminta perustuu siihen että liikkuvalla ikkunalla etsitään tiedostosta toistoja. Sitten kun kyseinen toisto löytyy niin tämän jälkeen kirjoitetaan sen paikalle etäisyys ja kuinka pitkä tämä toisto on. Purku tapahtuu sitten niin että kun tullaan kohdalle mihin on kirjoitettu toiston etäisyys ja pituus pari se voidaan purkaa siirtämällä pointteri kohtaan missä alkuperäinen match on ja kirjoitettaan se etäisyyspituus/parin tilalle.

Saavutetut aika- ja tilavaativuudet (m.m. O-analyysi pseudokoodista)

Huffmanin koodauksen aika ja tilavaativuudet ovat seuraavanlaiset.

Tekstin frekvenssien laskenta: Aikavaativuus O(n), tilavaativuus O(n).

Huffmanpuun muodostus prioriteettijonon avulla: Aikavaativuus O(n log n), tilavaativuus O(n)

Lähdetiedoston lukeminen vie aikaa O(n), frekvenssien kirjoituskohdetiedoston alkuun O(1) ja itse kohdetiedoston kirjoitus O(n) ne vievät myös tämän määrän tilaa.

Purkaessa lähdetiedoston lukeminen vie O(n) ja merkkien haku lähdetiedoston puusta vie aikaa O(log n). Kohde tiedoston kirjoittaminen vie aikaa O(n).

Joten Huffmanin koodauksen aikavaativuus on O(n log n) ja tilavaativuus O(n), missä n on tiedostossa olevien merkkien määrä.

LZW algoritmissa aika- ja tilavaativuudet ovat sekä pakatessa että purkaessa samat.

Sanakirjan alustus vie molemmissa tapauksissa aikaa O(1) ja tilaa O(1). Koodien luomisen aika ja tilavaativuus on O(n) missä n on tiedostossa olevien merkkien määrä. Koodin purku vie myöskin aikaa ja tilaa O(n) missä n on pakatussa tiedostossa olevien koodien määrä. Tiedoston kirjoittamisen ja lukemiset vievät aikaa taas O(n).

Joten LZW algoritmin aika ja tilavaativuudet ovat O(n).

LZ77 aikavaativuus tällä hetkellä on O(n^2) koska siinä on kaksinkertainen for loopi. Koska algoritmi on vielä tosi pahasti kesken, mielestäni tässä vaiheessa on turhaa analysoida sen aika ja tilavaativuuksia tämän enempää.

Suorituskyky- ja O-analyysivertailu (mikäli työ vertailupainotteinen)

Suoritus kykyyn liittyvää aineistoa löytyy enemmän testaus dokumentaatiosta. Tässä voisin mainita sen että LZW algoritmilla käytän erikokoista hashmapia kuin huffmanin koodauksessa. Huffmanin koodauksen hashmapin koko on 256. Huomasin suorituskyky testauksessa, että parhaan tuloksen saan LZW algoritmilla jos hashmapin koko on 2^16 alkiota. Tämä meinaa että LZW:n tilavaativuus on pienissäkin tiedostoissa melko suuri, mutta vakio. Mutta tämä on mielestäni tämän uhrauksen arvoista, koska 2^16 kokoinen hashmap parantaa huimasti suorituskykyä isoja tiedostoja pakatessa. ArrayListissä parhaan suorituskyvyn taas sain kun aluksi alustin sen kooksi 256 ja sitten aloin kasvattamaan sitä aina kaksinkertaiseksi.

O-analyysivertailuin mukaan LZW:n pitäisi olla parempi algoritmi kun Huffmanin oma, mutta kuten suorituskyky testauksesta voidaan huomata, näin ei kuitenkaan ole. LZW algoritmi on kuitenkin ylivertainen jos tiedostossa on paljon toistoja tai copypaste tekstiä. Erittäin pienissä tiedostoissa taas LZW:llä taas tulee liikaa overheadiä 24-bittisen koodien kirjoituksen takia ja tiedoston koko pakatessa kasvaa pienimiseen sijasta, tämä ongelma voitaisiin korjata lisäämällä algoritmiin keino jolla saataisiin tiedoston koko selville ennen kuin pakkauksen suoritus alkaa. Ja käytettäisiin tällöin 12-bittistä koodien kirjoitusta. Jos tällainen systeemi olisi algoritmissa mukana voitaisiin myöskin hashmap alustaa sopivamman kokoiseksi.

Työn mahdolliset puutteet ja parannusehdotukset

LZ77 algoritmin toiminta on pahasti kesken. Lisäksi luultavasti hieman voisi parantaa sekä Huffmanin koodauksen että LZW algoritmien suorituskykyä. Lisäksi ohjelmaa voisi laajentaa toimimaan myös muilla tiedostotyypeillä kuin tekstitiedostoilla. LZW:n tapauksessa tämä luultavasti onnistuisi helpoiten, algoritmia pystyy jo ajamaan muilla kuin tekstitiedostoilla ilman että se kaatuu, mutta tiedoston koko kasvaa eikä pienene. Asia mikä luultavasti lisäisi algoritmien suorituskykyä, olisi että kovakoodaisin binääritiedoston lukemista ja kirjoittamista käsittelevän koodin itse algoritmeihin, mutta tämä olisi kaikkea Matti Luukkaisen kursseilla oppimaani vastaista(clean code, srp-periaate etc), joten tähän en ala.

Lähteet

http://en.wikipedia.org/wiki/Huffman_coding

http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch

https://en.wikipedia.org/wiki/LZ77_and_LZ78