Toteutusdokumentti

1. Ohjelman yleisrakenne

Ohjelmani toteuttaa LZW-pakkaus- ja purkualgoritmit. Ensin tiedoston tavut luetaan, sitten LZW-algoritmin mukaisesti luodaan sanakirja merkeistä. Tässä tapauksessa päätin tallentaa tiedoston syötteen tavuina LZW-sanakirjaan. Yksi merkki esim. UTF-8-koodauksessa saattaa muodostua kahdesta tavusta. Ajattelin kuitenkin, että ohjelman kannalta on yksinkertaisempaa käsitellä tiedostosyötettä tavuina enkä usko tämän heikentävän ohjelman suorituskykyä. Näin ei tarvitse arvailla, että sisältääkö tekstitiedosto erikoisempia merkkejä. Jos ajateltaisiin, että oletuksena pitäisi muodostaa LZW-sanakirja kaikista UTF-8-merkeistä, sanakirjan koko olisi päälle 60000. Nyt kuitenkin riittää tavun koon mukaisesti kahdeksan bittiä ja 256 merkin sanakirja pohjaksi sekä pakkaukselle että purkamiselle. Yksi merkki voi ohjelman kannalta olla kaksi eri tavua, mutta lopputuloksena purettu pakkaus kuitenkin saa saman muodon kuin alkuperäinen.

Pakkauksessa käydään läpi tekstitiedoston tavut yksi kerrallaan. Syötettä käydään läpi merkki kerrallaan ja tutkitaan, onko merkkijono esiintynyt jo aikaisemmin tiedostossa. Jos löydetään uusi aikaisemmin esiintymätön merkkijono, se tallennetaan LZW-sanakirjaaan ja annetaan sille tunnukseksi uusi koodinumero. Purkamisessa on myös sama periaate, vaikka koodattuja merkkijonoja tulkittaessa ollaankin hieman jäljessä. Muutamassa tapauksessa koodattua merkkijonoa ei ole vielä purkusanakirjassa vaan se joudutaan päättelemään edellisestä koodista.

Testauksen vuoksi siirsin lopulliseen versioon tallennuksen algoritmin loppuun, jotta testitulokset kuvaisivat paremmin itse algoritmin tehokkuutta.

2. Saavutetut aika- ja tilavaativuudet (m.m. O-analyysit pseudokoodista)

Ohjelma käy läpi tekstitiedoston vain yhden kerran tallentaen merkkijonoja sanakirjaan samalla, joten tiedoston läpikäynnin aikavaativuus on O(n) tiedostokoolla n. Kriittisin osa liityy LZW-algoritmin sanakirjaan ja erityisesti sanakirjahakuihin. Sanakirjasta pitää tehdä tekstitiedoston joka merkin kohdalla hakuja. Toteutin algoritmin aluksi Javan HashMap-rakenteella. En ehtinyt tässäkään kohtaa tutustua HashMapin periaatteisiin ja aikavaativuuksiin. Toteutin oman korvaavan tietorakenteen hajautustaululla. Hajatustaululla toteutettuun LZW-sanakirjaan tallennetaan sekä pakattaessa että purettaessa ja tässä vaativiin osuus on tutkia, onko tekstipätkä jo tallennettu sanakirjaan. Pakkaaminen on tiedetysti vaativin osuus: täytyy tutkia merkki merkiltä, onko merkkijono LZW-sanakirjassa. Toteutin hajautustaulun ensin kiireessä tyhmällä algoritmilla joka laski vain merkkijonojen arvot yhteen. Tässä tapauksessa esim. merkkijonot ”AB” ja ”BA” saivat saman arvon. Sain kuitenkin helposti algoritmia nopeutettua lisäämällä merkkijonojen merkkeihin kertoimen järjetysnumeron mukaan eli esim. 2. merkki sai kertoimen 2. Tällä tavalla pakkaus nopeutui huomattavasti (kts. Testausdokumentti). Huonommalla tavalla samoja merkkejä sisältävät tekstitiedostot pakkaavat merkkijonoja hajautustaulussa samaan ylivuotolistaan, joten haku voi viedä huonoimmassa tapauksessa (esim. samoja merkkejä sisältävässä tiedostossa ”ABABA”) todella paljon aikaa, kun joudutaan käymään ylivuotolista kokonaan läpi käyden myös merkkijonon joka merkki läpi.

Varasin hajautustaulukolle tilaa päähänpistosta hajautusalgoritmin alkuluvun 772663 verran. Sanakirja voi paisua tiedoston koon mukana varsinkin, jos tiedostossa ei ole toistuvuuksia. Hajatusalgoritmilla saadut samanarvoiset tulokset tallentuvat taulukossa ylivuotolistalle (termi Tietorakenteet ja algoritmit -kurssin luentomateriaalista), joka on linkitetty lista. Ideaalitapauksessa muisti olisi rajaton, joten jokaiselle syötteelle voisi antaa eri hajautusluvun ja kohdan taulukosta, jolloin haettu data löytyisi heti taulukosta. Kuitenkin nyt suuremmilla tiedostoilla ylivuotolistat täyttyvät koko ajan, joten aikavaativuus heikkenee reilusti tiedostokoon kasvaessa.

3. Suorituskyky- ja O-analyysivertailu (mikäli työ vertailupainotteinen)

Testaustuloksista päätellen (kts. Testausdokumentti) Javan HashMap-rakenteella etsintä LZW-sanakirjasta käy O(1)-ajassa, kuten Tietorakenteet ja algoritmit -kurssin luentomateriaalissa sanotaan. Tiedostokoon ja pakkausajan suhde on suunnilleen sama tiedostokoosta riippumatta. Omassa hajautustaulutietorakenteessani taulukon koko oli kiinteä, mikä hidastaa tiedostokoon kasvaessa hakua, kun taulukkoon pakkaantuu avain-arvo-pareja vähitellen.

4. Työn mahdolliset puutteet ja parannusehdotukset

Kuten edellä olevassakin olen kertonut, ohjelmani tehokkuutta parantaisi parempi hajautusalgoritmi. Myöskin muistia voisi käyttää joustavemmin ja hajautustaulun muistivaraus pitäisi olla tiedostokoon mukainen. Ehkä ohjelmaa hidastaa myös tavujen ja muiden käsiteltävien yksikköjen muuttaminen String-olioiksi Integer.toBinaryString-metodilla. Bittimuunnosten tekeminen oli kuitenkin sen verran rasittavaa pelkillä numeroilla sumplittuna, joten tässä kohdin tingin ohjelman tehokkuudesta.

Lähteet:
Welch, Terry (1984). "A Technique for High-Performance Data Compression" (PDF). Computer 17 (6): 8–19.
https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
https://www.youtube.com/watch?v=p4HnBe_KzKM