Okko Vainonen:

Tekstinpakkaus Lempel-Ziv-Welch-algoritmilla

Ohjelmointiharjoitustyöni tarkoituksena on java-ohjelmointikielellä rakentaa vuonna 1984 kehitetty Lempel-Ziv-Welch-algoritmi, lyhennettynä LZW, joka on luotu häviötöntä tiedonpakkausta varten. Tämä tarkoittaa, että pakattua tietoa purkaessa tuloksena on alkuperäistä vastaava tiedosto, mikä on tekstimuotoista tietoa käsiteltäessä ehdoton vaatimus. LZW-algoritmi rakennettiin parannuksena muutama vuotta aikaisemmin julkaistulle Lempel-Ziv-algoritmille, josta on versiot LZ77 ja LZ78. Vertailun vuoksi, jos aika sallii, voisin yrittää toteuttaa LZW:n pohjalta tehtyjä variantteja, kuten LZMW, LZAP tai LZWL.

LZW:ssä luodaan ensin ns. sanakirja, johon tallennetaan käytössä olevat merkit ja annetaan niistä jokaiselle indeksinumero. Tämän jälkeen syötettä ruvetaan käymään läpi merkki kerrallaan. Jos merkki on jo sanakirjassa, siirrytään seuraavaan merkkiin ja lisätään se jonona edelliseen merkkiin. Jos tällaista merkkijonoa ei vielä ole sanakirjassa, se lisätään sanakirjaan uuden indeksinumeron kera. Muussa tapauksessa jatketaan merkkijonon kasvattamista, kunnes löytyy sellainen jono, jota ei vielä ole sanakirjassa ja joka voidaan sinne siten lisätä. Tällä prosessilla tekstistä pyritään löytämään useamman merkin kaavamaisuuksia, joita voidaan merkitä bitteinä lyhyemmän mittaisella tietoyksiköllä.

Aika- ja tilavaativuuteen LZW:ssä vaikuttaa käsittääkseni eniten se, millä tietorakenteella sanakirja muodostetaan. En ole tätä vielä miettinyt loppuun, mutta jonkunlainen hashmap-rakenne voisi tulla kyseeseen. Tekstisyöte käydään läpi merkki kerrallaan ja vain yhden kerran, joten sen pohjalta aikavaativuus on vain O(n) n merkkiä sisältävässä tiedostossa. Vastaantulevia merkkijonoja vertaillaan aikaisemmin sanakirjaan tallennettuihin merkkijonoihin ja etsimisoperaation pitäisi olla hashmap-tyylisessä tietorakenteessa aikavaativuudeltaan keskimäärin O(1). Muistivaativuus taas on paljon suurempi kuin esimerkiksi Huffmanin pakkausalgoritmissa. Sanakirja joudutaan tallentamaan muistiin ja jos teksti on täysin satunnainen, sanakirjasta voi tulla jopa suurempi kuin tekstisyötteen koko, koska jokainen uusi merkkiyhdistelmä joudutaan tallentamaan sanakirjaan.

Pakkauksen purkaminen ei ole aika- ja tilavaativuudeltaan suurempi, koska purkaminen tapahtuu samalla logiikalla kuin pakkaaminenkin. Muodostetaan aluksi samanlainen oletussanakirja merkistöstä ja lähdetään käymään pakattua syötettä merkki – tässä tapauksessa bittijono – kerrallaan. Kun vastaan tulee ensimmäinen tuntematon merkki, se lisätään sanakirjaan, ja tutkimalla syötteen aiemmin esiintyneitä merkkejä, lisätään ensimmäinen sanakirjalle tuntematon merkkijono sanakirjaan.

Welchin artikkelissa on käsitelty lähinnä vain pakkauksessa muodostetun tiedoston suuruutta alkuperäiseen verrattuna eikä ole otettu kantaa aika- ja tilavaativuuksiin. Pakkauksessa käytetään aluksi 9 bitin kokoisia jonoja, jos pakataan 8 bitin kokoisia merkkejä. Kun vastaan tulee ensimmäinen bittijono, jossa on pelkkiä ykkösiä, käsittelykokoa kasvatetaan yhdellä bitillä sekä pakkauksessa että sen purkamisessa. Alussa pakattavan tekstijonon merkit ovat vain 8-bittisiä ja pakattavat symbolit 9-bittisiä, joten aivan lyhyen tekstinpätkän pakkauksessa alkuperäinen teksti on lyhyempi. Pakkaustehokkuus kuitenkin kasvaa teksti koon kasvaessa asymptoottisesti kohti tiettyä maksimirajaa, kun pakkaus pääsee hyödyntämään tekstistä löytyviä kaavamaisuuksia. Pakkauksen tehokkuus riippuu siitä, minkälaista tietoa pakataan. Welchin artikkelissa testituloksissa englanninkielinen teksti pakkautuu 1,8 kertaa alkuperäistä pienempään kokoon, kun taas ohjelman lähdekoodi kutistuu 2,3-kertaisesti, kun taas satunnaisemmassa ainestossa (esim. liukuluvut) pakkaustehokkuus lähenee nollaa. Oman ohjelmani testisyötteinä ajattelin käyttää eri kielisiä tekstejä, esim. suomi, englanti ja ranska ja kenties jotain eksoottisempaa kieltä sekä näiden lisäksi ainakin äärimmäistapauksena täysin satunnaista merkkigeneroitua tekstiä.

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