Ohjelman toteutus

Yleisrakenne

Ohjelma on jaettu useampaan moduliin, joista jotkut ovat riippuvaisia toisistaan. Funktioiden nimeämiskäytäntönä on käytetty tyyliä `moduuli_funktionimi`. Ohjelmassa on tekstipohjainen käyttöliittymä, joka käyttää compress-moduulin tarjoamia toimintoja kuvien lataukseen ja tallennukseen. Compress-moduuli käyttää lähes kaikkien muiden moduulien tarjoamia ominaisuuksia, sillä se sisältään pakkauksen korkean tason toimintalogiikan. Varsinaiset algoritmit (DCT, ZRL-pakkaus ja Huffman-koodaus) on toteutettu erillisissä moduuleissaan. Itse toteuttamani tietorakenteet ovat util-moduulissa.

Moduulit

- util/bitbuf.h
  - bitin tarkkuudella toimiva dynaaminen taulukko
- util/stack.h
  - yksinkertainen dynaamisesti kasvava osoitinpino
- util/trie.h
  - binääripuun toteutus
- block.h
  - 8x8 pikselin ruutujen käsittely
- compress.h
  - korkean tason kuvanpakkausfunktiot
- dct.h
  - diskreetin kosinimuunnokset toteuttavat funktiot
- eks_math.h
  - C99-standardista puuttuvien matemaattisten funktioiden määrittelyt
- huffman.h
  - huffmanin-koodaukseen liittyvät funktiot
- jpeg.h
  - jpeg-kohtaiset vakiot ja toiminnot

- stat/stat.h
  - testauksessa käytetyt tilastolliset toiminnot
- tiraimg.h
  - ohjelmakohtaiset vakiot

Saavutetut aika- ja tilavaativuudet

Aika- ja tilavaativuudet ovat aikalailla samat mitä määrittelydokumentissa suunnittelin.

DCT Toteutin DCT:stä ja IDCT:stä naiivit ratkaisut, joissa jokaisen taajuusyhdistelmän kohdalla tarkastellaan kaikki 8x8 ruudun pikselit. Algoritmin aikavaativuus on siis $O(N^2)$. Ruudut ovat kuitenkin vakiokokoisia, joten voimme ajatella DCT:n laskemisen olevan ruutujen lukumäärän vakiokerroin. DCT:n laskemisen aikavaativuus on siis $O(N)$, missä N on kuvan pikseleiden lukumäärä.

Syötekuva kopioidaan uuteen taulukkoon samalla kun se jaetaan ruutuihin DCT:n laskemista varten, joten algoritmin tilavaativuus on $O(N)$.

ZRL, zero run-length coding ZRL:llä tarkoitetaan tässä tapauksessa ruutujen loppuosassa sijaitsevien nollarivien tunnistusta. Tämä onnistuu yksinkertaisesti käymällä ruudun kaikki pikselit lävitse lopusta alkuun, ja lopettamalla kun esiintyy ensimmäinen nollasta poikkeava tavu. [1] Pahimmassa tapauksessa nollia ei ole ollenkaan, jolloin käydään lävitse koko ruutu, eli aikavaativuus on $O(N)$.

Jos ruudun lopussa ei ole ollenkaan nollia, joudutaan muistiin tallentamaan ruudun arvojen lisäksi niiden pituus, jolloin tilavuus hieman kasvaa. Siis $O(N + 1) = O(N)$, missä N on yksittäisen ruudun pikseleiden määrä, joka on tässä tapauksessa 64.

Huffman-koodaus

Pakkaus Huffman-koodauksessa on kaksi vaihetta: Huffman-puun luominen ja syötteen pakkaaminen. Puun luomisessa on ensin selvitettävä kaikkien symbolien (tässä tapauksessa tavujen) esiintymistiheydet, joka on aikavaativuudeltaan $O(N)$. Puun rakentamisessa ei ole käytössä prioriteettijonoa tai muuta vastaavaa edistynyttä tietorakennetta, vaan pelkkä symbolitaulukko

[1] Sillä kaikki arvot on tiraimg:issä tallennettu 8-bittisinä etumerkillisinä kokonaislukuina vakiovähennyksellä, on nollatavu itseasiassa 0x80, ei 0x00.

johon tehdään lineaarihakuja. Tämän vuoksi rakentamisvaiheen kokonainen aikavaativuus on $O(N + |L|^2)$, jossa N on syötteen pituus tavuina, ja $|L|$ on aakkoston symbolien lukumäärä.

Varsinaisessa pakkauksessa syötedata käydään läpi, ja jokaisen symbolin kohdalla tulostetaan oikea koodi esilasketusta taulukosta. Rutiinin aikavaativuus on $O(N)$.

Pakattu data pidetään muistissa yhtäaikaa syötteen ja puurakenteen kanssa. Jos jokainen 256 tavusta esiintyisi syötteessä yhtä monta kertaa, olisi puu täydellinen eli siinä olisi $2^{8+1} - 1 = 511$ solmua. joten tilavaativuus on $O(511 + N) = O(N)$.

Purkaminen Huffman-koodauksen purkamisoperaatiossa jokaisen syötedatan bitin kohdalla tehdään yksi liike alaspäin puussa. Näin ollen aikavaativuus on $O(NH)$, missä N pakatun datan pituus *symboleina* ja H on Huffman-puun korkeus. Tilavaativuus on sama kuin pakatessa.

Puutteet ja jatkokehitysmahdollisuudet

Ohjelman sisäiset tietorakenteet ja tallennuksessa käytettävä tiedostoformaatti perustuvat oletukseen, että jokainen DCT-kerroin pystytään tallentamaan kvantisoinnin jälkeen 8-bitin mittaiseen muuttujaan. Matalamilla laatutasoilla kertoimet kvantisoidaan suuremmilla jakajilla, jolloin tallennettavien kertoimien itseisarvo pienenee, eikä tämä rakenteellinen rajoite koidu ongelmaksi. Kuitenkin käyttäessä korkeampaa laatua [2] artefaktit ovat ilmiselviä: kerroinmatriisin matalin taajuus (DC) on joissakin ruuduissa väärä, joka näkyy kirkkauden tai värisävyn suurena paikallisena vaihteluna.

Pakkausalgoritmi on myös varsin hidas, sillä trigonometristen funktioiden laskemista ei ole optimoitu millään tavalla. Jo yksinkertainen taulukointiratkaisu (mikä on käytössä useimmissa JPEG-toteutuksissa) toisi suuren hyödyn. Koko algoritmi on mahdollista toteuttaa myös pelkillä kokonaislukuoperaatioilla, mikä nopeuttaisi prosessointia huimasti.

Pakkaussuhdetta voisi parantaa tallentamalla kuvan värikomponentit heikommalla laadulla kuin kirkkausdatan. Tästä ei aiheudu juuri näkyviä artefakteja, ja esim. JPEG-pakkaus tukee tätä ominaisuutta (chroma subsampling). Tämä kuitenkin vaatisi muutoksia tiedostoformaattiin ja pakkauskoodiin.

[2] Tarkka arvo riippuu syötekuvasta, mutta yleensä virheitä alkaa esiintyä kun arvo on yli 80.