Toteutusdokumentti Pihla Toivanen

Ohjelmani koostuu kolmesta isommasta osa-alueesta:

1. RGB-tiedoston luku ja erottelu 8*8 -blokkeihin sekä muu alustus.

- Separate(): Luetaan .rgb -tiedosto ja erotellaan se blocks[][][][] -taulukkoon. Kehitin indeksointiin systeemin, jossa 1. ulottuvuus on 8*8-blokin indeksi, 2 seuraavaa ulottuvuutta 8*8 -blokin indeksit ja neljäs ulottuvuus on tarkoitettu kolmea RGB -arvoa varten. Aikavaativuus on O(n), tilavaativuus n. N on kuvan x-akseli * kuvan y-akseli * 3 RGB -arvojen kanavien määrästä.

- DecreaseBlocks() ja increaseBlocks() ovat algoritmiin kuuluvia lisäys -ja vähennysfunktioita. Aikavaativuus O(n) ja tilavaativuus 3-arvoinen int-taulukko.

- ConvertToRgb ja convertToYCbCr kääntävät rgb-arvoja YCbCr -arvoiksi ja toisin päin. Aikavaativuus O(n) ja tilavaativuus 3-arvoinen int-taulukko.

Tehtyjä parannuksia tai parannusehdotuksia osioon:

- Minulla meni pitkään tajuta, että YCbCr – RGB -konversioissa välillä voi konvertoitua negatiivisiksikin arvoiksi, parannus on siis validointifunktiot.

2. Diskreetti kosinitransformaatio ja käänteistransformaatio

- doForBlocks(): yleinen pohja, jota Main kutsuu. Tämä käy kaikki kuvan arvot läpi kutsuakseen komentona annettavaa funktiota jokaiselle arvolle sovellettavaksi. Aikavaatisuus O(n) ja tilavaativuus 0.

- DCT tekee diskreetin kosinitransformaation. Aikavaativuus c*O(64^4) → O(64^4), missä c on kuvan 8*8 blokkien määrä.

    - Koska tehtävä operaatio on optimoitu mahdollisimman nopeaksi dynaamisella ohjelmoinnilla ja kuvat eivät koskaan ole äärettömän suuria, ohjelma toimii siedettävässä ajassa. Tilavaativuus 0.

- ApplyIDCT, käänteinen transformaatio. Aikavaativuus c*O(64^4) → O(64^4) samoin kuin edellisessä, tilavaativuus 0.

- kvantisointi ja dekvantisointi: Kvantisoinnissa kerrotaan jokainen kuvan 8*8 blokki JPEGin speksissä määritetyllä kvantisointimatriisilla.

Tehtyjä parannuksia tai parannusehdotuksia osioon:

- Tein ensin transformaation ilman dynaamista ohjelmointia eli kosiniarvot laskettiin aina uudelleen. Sitten tein dynaamisen ohjelmoinnin taulukon (giveDpvalue -metodi), joka tallentaa ja ottaa aiemmin laskettuja arvoja käyttöön. Dynaaminen ohjelmointi muutti ison kuvan (1024*1024px) pakkausaikaa 1 minuutista 4 sekuntiin, ja pienen kuvan (256*256px) pakkausaikaa 4 sekunnista 0,5 sekuntiin.

3. Huffman-koodaus

Tein muutoin “perus-Huffmanin”, paitsi että en tallenna nollia ollenkaan (transformoidusta kuvadatastani parhaimmillaan 92% nollia esim. kokonaan mustassa kuvassa). Tallennan siis vain nollasta eroavat arvot ja jokaista sellaista ennen, kuinka monta nollaa datassa on ennen sitä.

- Frekvenssien laskeminen frekvenssitaulukkoon. Aikavaativuus O(n) ja tilavaativuus 2*x*y*3, missä x on kuvan x-akselin koko ja y kuvan y-akselin koko. Luku 3 tulee RGB-arvojen 3 kanavasta.

- Trie-rakenteen muodostus minimikeon ja Node-luokan avulla. Trie-rakenteen muodostuksen aikavaativuus on O(n*log n), suoraan käyttämästäni kekojärjestämisestä solmujen lisäämisessä trie:en. Minimikeon tilavaativuus on n, ja trie-rakenteen myös.

- Tiedostoonkirjoitus

  - Toteutan tiedostoonkirjoituksen tallentamalla ensin puun tiedostoon, sitten itse datan. Tiedostoon ei voi kirjoittaa haluamansa pituisia binääriarvoja, joten käytän kahta kirjastotiedostoa, jotka toteuttavat bufferit jotka kirjoittavat sitten 8-bittisiä arvoja.

  - Koska diskreetin kosinimuunnoksen tuloksena suurin osa datasta on nollia, tallennan nollat siten, että jokaista ei-nolla-arvoa edeltää tieto siitä, kuinka monta nollaa arvoa ennen on datassa.

  - Käytän tallennuksen apuna trie:n muodostuksen yhteydessä luotua 2*x*y*3 -kokoista Koodiarvot -taulukkoa. Siitä saa suoraan tiedon, mikä kyseisen arvon koodiarvo on.

- Tiedostostaluku

  - Ensin luetaan tiedostosta trie-puu ja luodaan uusi blocks -taulukko kuvadataa varten. Sen jälkeen jos kyseessä on nollien määrää kuvaava arvo, täytetään blocksiin nämä nollat. Muuten aletaan käymään trie-prefiksipuuta läpi sen ominaisuuksia hyödyntäen niin, että tiedetään tiedostoluvun koodiarvoa vastaava arvo löydettäessä, että voi siirtyä seuraavaan arvoon.

  - Pisin koodiarvo on pituudeltaan eri koodiarvojen määrän binääriesityksen pituus

eli log 2 (k), missä k on erilaisten koodiarvojen määrä. Pahimmassa tapauksessa uutta arvoa tiedostossa luettaessa aikavaativuus on siis log 2 (k). Tämä kerrottuna kaikilla arvoilla on siis x*y*3*log 2(k).

- Implementoin minimikeon (PQ.java) itse. Minimikeon aikavaativuudet ovat lisäys log n ja poisto log n. Tilavaativuus on n ArrayListillä toteutettuna.

- Implementoi ArrayListin (Arraylist.java) itse minimikekoa varten. ArrayListin aikavaativuudet ovat lisäys O(1), elementin hakeminen O(1), ja elementin poistaminen pahimmassa tapauksessa O(n).

Tehtyjä parannuksia tai parannusehdotuksia osioon:

- En ehdinyt tehdä Zigzag -järjestyksessä läpikäyntiä arvojen tallennuksessa. Se parantaa pakkaussuhdetta siksi, että diskreetissä kosinimuunnoksessa nolla-arvot kasautuvat oikean alakulman puolelle kuvaa, ja koska nollista tallennetaan vain niiden lukumäärä niin kerralla olevat lukumäärät kasvavat.