Toteutus ja testaus

June 16, 2013

1 Ohjelman rakenne ja aikavaativuus

Harjoitusyössä on toteutettu yksinkertainen aallokemuunnos sekä sen tallentaminen.

Ohjelma koostuu viidestä luokasta: GUI, Compression, BitmapIO, HaarTransform ja WTFIO.

GUI ja Compression ovat käyttöliittymäluokkia ja BitmapIO:ssa on bmp-tiedostomuodon lukemiseen ja kirjoittamiseen tarvittavat metodit. Varsinainen sisältö on HaarTransform-luokassa, jossa tehdään muunnos ja käänteismuunnos, sekä WTFIO-luokassa, jossa tiedosto tallennetaan. Käydään seuraavasti lyhyesti läpi, miten oleelliset luokat toimivat.

1.1 BitmapIO

Data luetaan kolmiulotteiseksi byte-taulukoksi, jossa `data[c][x][y]` kertoo, mikä perusvärin `c` sävy pikselillä paikassa (`x,y`) on. Lukemiseen kuluva aika on $O(mn)$, missä m on rivien ja n sarakkeiden määrä.

1.2 HaarTransform

Luokan tehtävä on ottaa vastaan BitmapIO:n lukemia datataulukoita ja muuntaa niitä, sekä ottaa vastaan muunnoksia WTFIO luokasta ja muuntaa niitä BitmapIO:n datataulukoiksi.

Muunnos tehdään jokaiselle kuvan sarakkelle erikseen, ja lopputulos riippuu häviökertoimesta `lol` (level of loss).

Jos sarakkeen pituus on jokin kahden potenssi $n = 2^k$, tehdään ensin binäärinen summapuu, jonka jokainen lehti on $2^{\texttt{lol}}$ sarakkeen peräkkäisen alkion summa. Lehtien vanhemmat ovat aina kahden lehden summia, jne. kunnes lopulta puun juuri on koko sarakkeen summa.

Summapuusta on helppo laskea Haarin muunnos kaavalla

`transform[i] = sumTree[2i] -sumTree[2i+1]` , `i` $\leq$ `sumTree.length / 2`

saadussa taulukossa on ensin hienoimman tason aallokkeiden kertoimet, sitten seuraavaksi hienoimman jne. Taulukon `transform` loppuun lisätään summapuun juuri, eli syötetaulukon alkioiden summa sellaisenaan. (Summataulukon pituus on pariton, joten ylläolevan kaavan yläraja `i`:lle on kelvollinen).

Summapuun laskemisen aikavaativuus on $O(n)$: jokainen syötteen alkio on käytävä läpi lehtien laskemista varten, siihen kuluu aikaa $O(n)$. Sen jälkeen lasketaan jokaiselle muulle puun solmulle yhteenlasku, ja näitä solmuja on vähemmän kuin lehtiä, joten aikavaativuus ei kasva.

Itse muunnoksen aikavaativuus on selvästi myös $O(n)$. Muunnoksen koko on $2^{-\texttt{lol}}$ kertaa syötteen koko lukujen määrällä mitattuna. Todellisuudessa kuitenkin syötteen bytet muuttuvat integereiksi, joten jos `lol` on pienempi kuin 2, lopputuloksen koko kasvaa.

Jos sarakkeen pituus ei ole kakkosen potenssi, etsitään suurin siihen sisältyvä alijono, jonka koko on, ja tehdään sille muunnos. Sitten tehdään sama jäljellä olevalle osalle sarakkeesta jne., kunnes koko sarake on muunnettu. Saadut muunnokset laitetaan vain peräkkäin yhdeksi pitkäksi taulukoksi, jonka tulkitsemista varten tarvitaan tieto sarakkeen alkuperäisestä pituudesta sekä häviökertoimesta.

Aikavaativuus on tällekin $O(n)$ ja muunnoksen koko on vieläkin suunnilleen $2^{-\texttt{lol}}n$ lukujen määrällä mitattuna.

Kuva kokonaisuutena muunnetaan tekemällä yllämainittu muunnos jokaiselle sen sarakkeelle. Jos kuvan koko on $m \times n$, on muunnoksen aikavaativuus $O(mn)$, ja se vie tilaa suunilleen $2^{-\texttt{lol}}mn$ integeriä.

Käänteismuunnos tehdään myös sarakkeittain. Syötteenä saadaan muunnoksen lisäksi alkuperäinen sarakkeen pituus ja level of loss. Algoritmi laskee, minkä mittaisiin osiin sarake on muunnoksessa jaettu, ja minkä kokoisiksi ne ovat kutistuneet. Jokaiselle pätkälle tehdään käänteismuunnos erikseen ja lopputulokset liimataan yhteen. Käänteismuunnoksessa kirjoitetaan jokainen taso aallokkeita erikseen, joten siihen kuluu aikaa $O(n)$ jokaisella säästetyllä tasolla allokkeita. Aikavaativuus sarakkeelle on siis $O(n[\log n - \log \texttt{lol}])$ kaikille järkeville häviökertoimille. Koko kuvan käänteismuunnoksen kesto on siis $O(mn[\log n - \log \texttt{lol}])$.

Huomaa: käänteismuunnoksessa on pari oudolta näyttävää asiaa: aallokkeita summattaessa ne kerrotaan muuttujalla `howMany` ja lopuksi niihin sovelletaan metodia `addDivideConvert`. Tämä johtuu siitä, että käytetty muunnos ei ole itseasiassa ihan aallokemuunnos. Tämä selitetään tarkemmin GitHubissa olevassa dokumentissa "Wavelets and wtf" osassa "3. How can they be applied to image processing".

1.3 WTFIO

Tämä luokka kirjoittaa muunnoksen wtf-tiedostoksi. Tiedoston alussa on seuraavanlainen header:

Alku	tyyppi	sisältö
0	byte	Tiedostomuoto
1	short	Värien määrä
3	int	Alkup. leveys
7	int	Alkup. korkeus
11	short	Level of loss
13	int	Tiivistetyn sarakkeen pituus

Tässä värien määrä, alkuperäinen leveys ja alkuperäinen korkeus ovat muuntamattoman datataulukon dimensiot.

Tiedostomuotoja on vain kaksi, arvoa 0 vastaa integerData, jossa muunnos tallennetaan integereinä. Tämä vie tolkuttomasti tilaa, ja se on koodissa jäljellä kokeiluja varten.

Tiedostomuotoa 1 vastaa mixedData, jossa jokainen muunnoksen sarake on tallennettu muodossa

`(int shortOffset, int intOffset, [bytes], [shorts], [ints] )` .

Alussa olevat offsetit kertovat, monesko luku on sarakkeen ensimmäinen short/int. Sitten muunnoksen kertoimet on kirjoitetty byteinä, shortteina ja intteinä. Luettaessa ne tulkitaan headerissa olevan luvun perusteella, joka kertoo, kuinka monta lukua sarakkeen muunnoksessa on.

Tiedostot kirjoitetaan ja luetaan melko suoraviivaisesti, siihen kuluu aikaa $O(mn)$. Tiedoston koko muuttuu sen sijaan ennalta-arvaamattomasti. Jos huomioidaan, että int on neljä byteä, saadaan karkeat rajat:

$$2^{-\texttt{lol}} \text{ alkuperäinen koko } \leq \text{ muunnoksen koko } \leq 4 \cdot 2^{-\texttt{lol}} \text{ alkuperäinen koko } .$$

Todellisuus on vähän onnellisempi, sillä muunnokset tapaavat alkaa pienillä luvuilla ja kasvaa loppua kohti.

1.4 JUnit-testit

Oleellisille luokille on kirjoitettu JUnit-testejä. Osa niistä edellyttää, että kansiossa on luettava tiedosto, jonka nimi on `c.bmp`, mutta testien ajaminen onnistuu ilman sitäkin: testiluokkien alussa on oma boolean-muuttujansa, jonka voi muuttaa `true`ksi, jos tiedosto on kansiossa. Tällä hetkellä nämä muuttujat siis ovat `false`ja.

2 Kuinka pieniksi tiedostot käytännössä tiivistyvät?

Kometoriviltä voi tehdä testejä kuvan tiivistymisestä kirjoittamalla

java -jar Compression.jar writeStats kuva.bmp .

Tämä komento luo ja tuhoaa tilapäisiä tiedostoja, joten sitä kannattaa periaatteessa varoa. (Ks. "How to use it" GitHubissa. Vaaraa ei ole ajokansion ulkopuolella, ja sielläkin vain, jos olet nimennyt tiedostoja erityisen onnettomasti).

Järjestelmällisessä kokeilussa olen käyttänyt seuraavia tiedostoja:

Kuva	koko kb.	leveys	korkeus	Sisältö
maisema.bmp	23 887	3456	2304	Talvimaisema
nuutti.bmp	9 437	1536	2048	Nuutti-kissa
nuutti-1.bmp	9 432	1536	2047	Edellinen yhtä riviä lyhempänä
screenshot.bmp	5 292	1680	1050	Ruuttukaappaus Netbeansista
levea.bmp	36 744	8000	1531	Panoramakuva San Franciscosta
korkea.bmp	36 768	1531	8000	levea.bmp käännettynä 90 astetta

San Francisco -kuva on wikipediasta:

`http://en.wikipedia.org/wiki/File:SanFrancisco_from_TwinPeaks_dusk_MC.jpg`

Muut kuvat lataan GitHubiin, ellei siinä esiinny ongelmia.

Ensimmäinen kysymys on: **miten hyvin teoreettiset tiedostokoot pitävät paikkansa?** Tämän voi selvittää tallentamalla wtf-tiedostot integerDatana, jolloin tiedostokoon pitäisi olla 2^{2-lol} kertaa alkuperäinen. Tämä on täysin teoreettinen kokeilu, joten valmis ohjelma ei tarjoa mahdollisuutta sen tekemiseen. Kokeen toistamiseksi voit vaihtaa Compression.class:in metodin convertAndAnalyze riviltä 298 kutsun writeMixedData muotoon writeIntegerData.

Kokeilin tätä kahdella tiedostolla: `nuutti.bmp`, sillä sen korkeus on 2048, eli kakkosen potenssi, ja `nuutti-1,bmp`, jonka korkeus on 2047.Saadut tiedostokoot ovat:

lol	nuutti	n teoriassa	nuutti-1	n-1 teoriassa
alkup.	9 437		9432	
0	37 748	37 748	37 730	37 728
1	18 874	18 874	18 874	18 864
2	9 437	9 437	9 455	9 432
3	4 718	4 719	4 755	4 716
4	2 359	2 359	2 414	2 358
5	1 179	1 179	1 253	1 179
6	589	590	682	590
7	295	295	405	295
8	147	147	276	147
9	73	74	221	74
10	36	37	202	37
11-	18	18	202	37

Tässä ”n teoriassa” ja ”n-1 teoriassa” tarkoittavat tiedostojen kokoja kerrottuna luvulla $2^{\lceil 2-\texttt{lol}}$ (paitsi, kun 2^{lol} on suurempi kuin sarakkeen pituus, josta eteenpäin häviön kasvattaminen ei enää vaikuta wtf-tiedoston kokoon). Kaikki koot on ilmoitettu kilotavuina.

On hämmästyttävää, kuinka hyvin todellisuus vastaa teoriaa sarakkeen pituudella 2048. Kun muunnoksessa joudutaan pilkkomaan sarakkeita osiin, eroavat tiedostojen koot jo selvemmin teoreettisista.

Kuinka paljon tilanne paranee mixedDatan avulla? Samoille kuville saadaan:

lol	nuutti	n mixed	%	nuutti-1	n-1 mixed	%
alkup.	9 437			9432		
0	37 748	10 615	28	37 730	23 567	62
1	18 874	5 897	31	18 874	12 082	64
2	9 437	3 537	37	9 455	6 348	67
3	4 718	2 358	50	4 755	3 490	73
4	2 359	1 224	52	2 414	1 820	75
5	1 179	644	55	1 253	981	78
6	589	353	60	682	569	83
7	295	206	70	405	371	91
8	147	133	90	276	286	103
9	73	96	131	221	251	113
10	36	68	189	202	239	118
11-	18	55	306	202	239	117

Sarakkeissa siis on `lol`, nuutti-kuvan koko integerDatana, mixedDatana ja jälkimmäisen prosenttiosuus edellisestä. Sitten sama kuvalle `nuutti-1.bmp`.

Nuutti-tiedostolle säästö on jopa huomattava. Nuutti-1:lle pienempi, koska mixedData ei huomioi sitä, koostuuko sarakkeen muunnos useamman pätkän muunnoksesta vai ei. Tässä tapauksessa sarake on jaettu osiin, joiden pituudet ovat 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2 ja 1, eli kyseessä on huonoin mahdollinen tapaus. Ensimmäisen pätkän muunnoksessa kertoimet kasvavat loppua kohti, ja intOffset on melkein varmasti pienempi kuin 1024. Siksi jokaisen muun pätkän muunnoksen pienetkin ketroimet kirjoitetaan byteinä, ja tilaa hukataan melkotavalla.

Kannattaakin huomioida, että millään häviökertoimella pienempi tiedosto `nuutti-1.bmp` ei mene pienempään kokoon kuin alkuperäinen, jonka sarakkeen pituus vain sattuu olemaan sopiva. Tämä olisi ollut hyvä tajuta jo koddatessa muunnosta: sen sijaan, että jaoin kuvaa pienempiin osiin, sen olisi voinut suurentaa, ja silti säästää tilaa.

Miten kuvan orientaatio vaikuttaa tiivistykseen?

Kuva `korkea.bmp` on sama kuin `levea.bmp`, mutta käännettynä pystyasentoon. Eroaako niiden pakkauksen koot jotenkin toisistaan? Voisi odottaa, että korkea kuva menee pienempään tilaan, koska siinä muunnos tehdään pidemmälle sarakkeelle. Näin ei kuitenkaan käynyt, minkä otaksuin johtuvan kuvan sarakkeen pituudesta, joka ei ole kakkosen potenssi. Tein siksi kaksi uutta kuvaa, uusiLevea ja uusiKorkea, joiden dimensiot olivat (4096×1024) ja (1024×4096). Muunnosten tulokset ovat:

lol	levea	korkea	uusiLevea	uusiKorkea
alkup.	36 744	36 768	12 582	12 582
0	63 163	99 306	18 100	18 711
1	34 809	51 663	10 719	10 557
2	18 222	26 387	5 715	5 699
3	9 484	13 362	2 995	3 000
4	5 007	6 742	1 591	1 549
5	2 755	3 408	873	803
6	1 635	1 735	502	423
7	1 087	904	306	229

Näyttää, ettei korkeudesta ole huomattavaa hyötyä.

Miten kuvan sisältö vaikuttaa muunnoksen kokoon?

Jotkin kuvat ovat kuin tehtyjä aallokemuunnokselle: ruutukaappaus Netbeansista voisi esimerkiksi olla tälläinen: siinä on suuria tasaisia pintoja ja pieniä paikallisia muutoksia. Tätä kokeilua varten leikkasin kuvista `maisema.bmp`, `nuutti.bmp` ja `screenshot.bmp` (1024×1024) neliöt, joille tein muunnokset. Alla tulokset:

lol	maisema	nuutti	screenshot
alkup.	3 145	3 145	3 145
0	3 852	3 521	6 226
1	2 278	1 948	3 139
2	1 318	1 162	1 585
3	699	769	809
4	371	413	423
5	206	225	227

Näyttää siis siltä, että screenshot vie enemmän tilaa kuin kuvat oikeasta maailmasta. Eräs selitys tälle voisi olla, että ruutukaappauksen kirjaimet tekevät hienompien tasojenkin kertoimista suuria, joten short- ja intOffsetit ovat pienempiä.

Miten pakkaus vaikuttaa kuvan laatuun?

Tätä on helpointa tutkia komennoilla

```
java -jar Compression.jar 0 10 maisema.bmp temp
```

ja

```
java -jar Compression.jar inv 0 10 temp kuva
```

, jotka tuottavat tiedostot `kuva1.bmp`, `kuva2.bmp`, jne. sekä vastaavat wtf-tiedostot.

Kuvalle `nuutti.bmp` ensimmäinen selkeä muutos tulee häviökertoimella 4, `maisema.bmp` muuttuu jokaisella häviökertoimella, mutta ensimmäinen, jolla sen voi sanoa olevan huonompi on 3. Kuvan `screenshot.bmp` teksti muuttuu häviökertoimella 1, mutta ei ole paljoa alkuperäistä huonompi. Häviökertoimella 2 se on jo huomattavan vaikealukuinen ja arvolla 3 täysin lukukelvoton.

Kuvat `korkea.bmp` ja `levea.bmp` muuttuvat selvästi huonommiksi häviökertoimilla 3-4. Huomionarvoista on, kuinka rakennusten ikkunarivit sumenevat niissä eri tavoilla sen mukaan, ovatko ne vaaka- vai pystysuoria. Nämähän siis olivat sama kuva käännettynä eri asentoon.

3 Kuinka aikavaativuudet toteutuivat käytännössä?

Tämä on sinänsä sivuseikka, koska ensimmäinen tavoite oli tilan eikä ajan säästäminen. Ajan säästäminen on toki hyvä toissijainen tavoite, mutta siinä eivät paremmat algoritmit olisi nähdäkseni paljoa auttaneet. Suurin osa ajasta tuhlautuu tiedostojen lukemiseen ja kirjoittamiseen.

Kokeilin kuitenkin kuvasta `korkea.bmp` leikattuja pienempiä kuvia, joiden korkeudet olivat 1000, 2000, ..., 8000. Tässä kuluneet ajat hatusta vedetyllä häviökertoimella 2 (ajat millisekunteja):

korkeus	muunnos	wtf:n kirjoittaminen	käänteismuunnos
1000	41	19	83
2000	58	35	169
3000	95	40	280
4000	121	47	347
5000	149	59	477
6000	539	72	568
7000	246	73	681
8000	284	81	749

Voiko näistä luvuista tehdä jotain jähtopäätöksiä, en tiedä. Kestot näyttävät sopivan hyvin lineaarisiin ja $O(n \log n)$ aikavaatimuksiin, kun ottaa huomioon satunnaiset ajonaikaiset viivytykset, joista esimerkiksi johtunee 6000:n korkuisen kuvan muuntamiseen kulunut pitkä aika. Ainakaan kesto ei hyppää pilviin rivien lisääntyessä.

3.1 Kauanko tiedoston muuntaminen kestää?

Suraavassa taulukossa kuvien muuntamiseen kuluneita aikoja:

kuva	koko kb	bmp:stä wtf:ksi	wtf:stä - bmp:ksi
screenshot	5 292	877	860
nuutti	9 437	1 382	1 281
maisema	23 887	3 474	2 653
levea	36 744	5 503	3 804
korkea	36 768	5 549	3 962

Muunnoksissa on häviökertoimena 2 ja ilmoitettu aika on mediaani viidestä ajosta.

4 Toteutuksesta yleisesti

Harjoitustyöni pääajatus näyttää jälkikäteen katsottuna olevan teorian soveltaminen käytäntöön. En ottanut juurikaan selvää, miten tällaisia asioita käytännössä tehdään, osittain koska en halunnut toteuttaa jo valmiiksi olemassaolevia pakkausmuotoja. Tämä näkyy lopputuloksessakin: siinä olisi voinut tehdä montakin asiaa järkevämmin.

Jos nyt aloittaisin tämän tekemisen alusta, tekisin muunnoksen etukäteen valitun mittaisille pätkille (kuten Pekka demossa kertoi tehneensä omansa). Lisäksi valitsisin paremman allokkeen kuin Haarin, oleellisesti tekisin kai diskreetin kosinimuunnoksen.

Nämä kaksi asiaa myös liittyvät toisiinsa, sillä käytin vain Haarin aalloketta, koska se takasi muunnoksen pysymisen kokonaislukuna kaiken kokoisille sarakkeille. Paremmilla allokkeilla tästä ei välttämättä olisi ollut taetta, mutta alueen fiksaaminen, jolla muunnos tehdään, ratkaisee tämän ongelman.

Jonkinlainen wtf-tiedoston lisätiivistys olisi myös ollut paikallaan, esimerkiksi Kristiinan ehdottama Lempel-Ziv-Welch, mutta sitä en kuitenkaan ehtinyt aloittaakaan.

Käytetyt tietorakenteet ovat älyttömän yksinkertaisia. Summapuu taitaa olla monimutkaisin. Kaksiulotteista muunnosta kokeillessani tein vähän monimutkaisempia puita, mutta eivät nekään mitenkään ihmeellisiä olleet.

En varsinaisesti käyttänyt mitään kirjallisuutta harjoitustyön tekemiseen, mutta ainakin seuraavista olen aiemmin lukenut aallokkeista:

Strang, Gilbert: Wavelet transfroms versus Fourier transforms
Bulletin of AMS vol 28 no 2. April 1993

Bachman, Narici, Beckman: Fourier and Wavelet analysis
Springer 2000