

Документация за JPEG компресия на изображения

Проект по СДА практикум 2020/2021

Автор: Николай Георгиев

ФН: 62307

Съдържание

Съдържание	2
1. Описание на проекта.....	3
1.1. Описание на задачата.....	3
1.1.1. Color space transformation.....	3
1.1.2. Разделяне на блокове с размер 8x8	3
1.1.3. Косинусова трансформация.....	3
1.1.4. Quantization	4
1.1.5. Хъфман енкодинг	5
1.2. Описание на решението	5
1.2.1. Клас Image	6
1.2.2. Клас HuffmanTree	7
1.2.3. Външни библиотеки.....	7
1) boost::dynamic_bitset	7
2) stb_image.....	8
2. Направени тестове	8
3. Резултати от тестване.....	10
3.1. Тест на изображение 16x8.....	10
3.2. Тест на изображение 8x16.....	10
3.3. Тест на изображение 8x8.....	11
3.4. Тест на изображение 16x12	11
3.5. Тест на изображение 12x16	11
3.6. Тест на изображение 12x12	12
3.7. Тест на изображение 7x4.....	12
3.8. Тест на изображение 4x7.....	12
3.9. Тест на изображение 7x7.....	12
3.10. Тест на изображение 32x16	13
3.11. Тест на изображение 16x32	13
3.12. Тест на изображение 32x32	14
3.13. Тест на изображение 32x20	15
3.14. Тест на изображение 20x32	15
3.15. Тест на изображение 36x36	16
3.16. Тест на несъществуващо изображение	16

1. Описание на проекта

1.1. Описание на задачата

Целта на проекта е да представи процесът на компресиране на изображения чрез JPEG алгоритъма. Той включва следните етапи:

- 1) Color space transformation
- 2) Разделяне на блокове с размер 8x8
- 3) Косинусова трансформация
- 4) Quantization
- 5) Хъфман енкодинг

1.1.1. Color space transformation

Изображението се транслира от RGB в YCbCr пространството. Y компонента представя яркостта на пиксела, а Cb и Cr - цветността съответно в синята и червената гама. За транслирането съществуват различни формули, като в този проект е ползван JPEG стандарта. Формулата представлява функция на R, G и B компонентите, като за Y, Cb и Cr функциите има различни коефициенти.

Целта на тази стъпка е алгоритъма да се възползва от факта, че човешкото око е по-чувствително към яркостта на дадено изображение отколкото към точността на неговите цветове.

1.1.2. Разделяне на блокове с размер 8x8

Тази стъпка цели да се раздели изображението на стандартни блокове с еднакъв размер, като блоковете с размер по-малък от 8x8 обикновено се допълват с някакъв вид запълваща информация. В текущия проект тези блокове се изрязват от изображението.

Това означава, че изображение с размери по-малки от 8x8 няма да се обработват.

1.1.3. Косинусова трансформация

Данните се центрират около нулата. По този начин RGB информацията, която е в интервала [0, 255] се транслира в интервала [-128, 127]. След това се прилага дискретна косинусова трансформация от втори тип.

Тя има следната формула

$$G_{u,v} = \frac{1}{4} \alpha(u) \alpha(v) \sum_{x=0}^7 \sum_{y=0}^7 g_{x,y} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

където

u, v са числа между 0 и 7, представящи съответно реда и колоната в матрицата от пиксели с размери 8x8

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{ако } u = 0 \\ 1, & \text{иначе} \end{cases}$$

$g_{x,y}$ е стойността на пиксела на позиция (x, y)

$G_{x,y}$ е коефициента на координати (u, v), получен от формулта

Като резултат по-големите стойности се концентрират в горния ляв ъгъл на блока и с приближаването към долния десен ъгъл постепенно намаляват.

1.1.4. Quantization

Както вече беше споменато човешкото око е по-чувствително към яркостта на изображението, но то разпознава разлика в яркостта само в сравнително големи части от изображението. Това позволява да се намали разликата в яркостта между съседни пиксели в отделни части от изображението. За целта се използва процеса quantization, ползваш следната формула:

$$B_{j,k} = \text{round}\left(\frac{G_{j,k}}{Q_{j,k}}\right)$$

където

j, k са числа между 0 и 7, съответстващи на реда и колоната в блоковете

$G_{j,k}$ е същото G от миналата стъпка

$Q_{j,k}$ е предварително дефинираната матрица

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}.$$

1.1.5. Хъфман енкодинг

Използа се дървото на Хъфман за съсдаване на битово представяне на всяко уникално число в текущия блок и в цялото изображение. За целта си използа дъро на Хъфман, което се построява по следния начин:

- 1) Създаване на масив от уникалните стойности в общата матрица на изображението, който се сортира по честотата на тяхното срещане в нарастващ ред
- 2) За всеки елемент се създава възел със неговата честота и се добавят във вектор
- 3) Докато има елементи във вектор, се изваждат последните два. От тях се създава нов възел с честота, равна на събита им. Той става тъхен родител, като елементът с по-малка честота е ляво дете. След това новия възел се добавя във вектора. Векторът се сортира, след което стъпката се повтаря от начало.

1.2. Описание на решението

Проектът се състои от 6 файла. Те включват **Image.cpp** и **HuffmanTree.cpp** за класовете със съответните имена както и техни хедър файлове **Image.h** и **HuffmanTree.h**.

Source.cpp е файлът, съдържащ **main** функцията и функция **testImageCompression**, използвана за извършване на тестове върху примерни изображения.

stb_image.h е локално копие на **stb_image** от библиотеката **stb**.

Папката **boost_1_54_0** е локално копие на библиотеката **boost**.

Също така е предоставена папка **img**, съдържаща изображенията използвани за тестовете.

1.2.1. Клас Image

Класът `Image` служи за обработка на изображението и свързване на отделните стъпки на JPEG алгоритъма.

Главните методи на `Image` са:

- **colorTransformAndInit** – извършва трансляцията от RGB в YCbCr пространството, като инициализира вектор от вектори, представлящи матриците на всяка една от трите компоненти на пространството.
- **cosTransform** – извършва косинусовата трансформация, използвайки метода `g`. След получаване на резултати се обновяват стойностите на трите матрици.
- **g** – извършва изчисленията, свързани с косинусовата трансформация за един пиксел и връща резултата като вектор с три елемента – по един за всяка компонента на пространството.
- **quantization** – извършва преобразуванията свързани с процеса quantization, като обновява стойностите на трите матрици `y`, `cb` и `cr`.
- **initFrequencies** – намира честотата на срещане на всяка уникална стойност и в трите матрици на компонентите. Извиква се за всяка от тях, като резултатите се пазят в общ `unordered_map`, наречен `freq`.
- **encodeHuffman** – извършва Хъфман енкодинг-а като ползва метода `getBitRepresentations` на класа `HuffmanTree` за създаване на дърво на Хъфман от сортирания масив на честотата на срещане на елементите `freq`. Като резултат се получава `unordered_map`, наречен `table`, с ключ уникален елемент от трите матрици и стойност булев вектор на битовото му представяне според дървото на Хъфман.
- **getBitRepresentation** – използва `table`, получен от предишната стъпка, за да създаде последоватленост от битове отговаряща на тройки от вида `y[x][y]cb[x][y]cr[x][y]`. (`x`, `y`) са координатите на пикселите в оригиналното изображение, т.е. в този метод се заменят RGB десетични тройките с YCbCr битови тройки, като се запазва оригиналната подредба. Получената последователност се запазва в структурата `dynamic_bitset` от библиотеката `boost`.
- **compress** – свързва останалите методи. За целта извършва определени преобразувания на данните между извикването на различните методи.

1.2.2. Клас HuffmanTree

Класът **HuffmanTree** служи за извршване на операциите, свързани с конструирането на дървото на Хъфман. Представяла двумерно дърво, чито възли съдържат стойността на дадения елемент и неговата честота на срещане. Родителските възли се създават като комбинация на децата си. За стойност се избира -130, тъй като след косинусовата трансформация стойностите на данните са в интервала [-128, 127], т.е. по този начин се гарантира, че родителските възли ще бъдат лесно различими от възлите деца.

Честотата се получава като сбор на честотите на двете деца.

Векторът **nodes**, който пази оставащите стойности за добавяне в дървото, се използва като стек чрез методите `back()` и `pop_back()`, чрез които се достъпват само последните елементи. Векторът се сортира в намаляващ ред след всяка итерация. Така се гарантира, че листата на дървото ще са най-рядко срещаните елементи и с доближаването до корена ще се увеличава честотата на елементите. За получаването на битовото представяне на даден елемент се обхожда дървото, като за всяко преминаване на ляво се добавя 0, а на дясно - 1. Така колкото по-близо е до корена (с по-голяма честота) даден елемент, толкова по-малко бита ще са необходими за неговото представяне.

Методът `getBitRepresentations` инициализира `unordered_map` table, който се ползва в `Image` за заместването на десетичните стойности с тяхното битово представяне.

1.2.3. Външни библиотеки

1) boost::dynamic_bitset

За да има смисъл от дървото на Хъфман е необходимо използването на структура от данни, която запазва отделните данни като единични битове. Стандартната библиотека предлага `bitset` като такава структура, но тя има недостатък, свързан с условието за деклариране на размера като константа. В случая за всяко изображение за инициализиране на `bitset` трябва да се сметне размерът на последователността от битове, което се случва по време на компилация, т.е. не е предварително дефинирано.

Това налага използването на `dynamic_bitset` от библиотеката `boost`. Той предоставя възможност за динамично деклариране на размера на структурата, което решава проблема на `bitset`.

Алтернативен вариант е използването на вектор от булеви стойности, но той не е оптимален поради факта, че булевия тип е 8 бита, а проекта търси решение, използващо само 1 бит за отделните стойности в крайната последователност от битове. Такова решение е `dynamic_bitset`.

Документация:

https://www.boost.org/doc/libs/1_36_0/libs/dynamic_bitset/dynamic_bitset.html

2) stb_image

Проектът се занимава със същността на JPEG алгоритъма за компресия, което не включва начинът, по който изображението ще бъде прочетено от програмата. За целта се използва **stb_image** от библиотеката **stb**.

За интегрирането му е необходимо единствено да се добави файлът `stb_image.h` в папката с хедъри на проекта.

Използван е във файлът за клас `Image`. Чрез методите на `stb_image` изображението, намиращо се на подадения като входни данни път, се отваря и за всеки пиксел се създава тройка стойности, отговарящи на RGB стойностите му (на червения, зеления и синия цвят). Като резултат се създава вектор от символи(`char`), които в проекта се кастват до `double`, за да може да се използват в следващите обработващи данните методи.

Документация: <https://github.com/nothings/stb>

2. Направени тестове

За тестване на коректността на поведение на програмата са използвани 15 изображения с различни размери. Направен е тест и за опит за компресиране на несъществуващо изображение.

Така са тествани следните случаи:

- Изображение с размери, делими на 8, които са
 - с различни размери на дълчината и ширината
 - с еднакви размери на дълчината и ширината
- Изображение с размери, неделими на 8, които са
 - с различни размери на дълчината и ширината
 - с еднакви размери на дълчината и ширината

Изходът от всеки тест е следния:

- **Теоретично изчисление на използвана памет за съхранение на изображение** – изчислена е паметта, която би била нужна за съхранение на изображението като RGB последователност от десетични числа (тип `double`) и за съхранението по като последоватленост от битове, базирани на YCbCr формат. Изчислението е „теоретично“, защото се получава чрез умножение на броя данни с необходимата памет за съхранение на един брой от техния тип.

- **Битово представяне на данните** – принтиране на конзолата на последователността от битове след извършване на компресията
- **Сравняване на очакваната първа тройка числа с получената от компресията** – проверява се дали битовото представяне на първия пиксел е очакваното (полученото от дървото на Хъфман). Принтират се числата, съответстващи на Y, Cb и Cr стойностите на първия пиксел, получени от дървото на Хъфман. След всяка от тях се принтира съответно първата, втората и третата последоватленост от битове в крайния масив от битове. Така се доказва, че след компресията стойностите са в правилна последоватленост.

3. Резултати от тестване

3.1. Тест на изображение 16x8

3.2. Тест на изображение 8x16

3.3. Тест на изображение 8x8

3.4. Тест на изображение 16x12

3.5. Тест на изображение 12x16

3.6. Тест на изображение 12x12

3.7. Тест на изображение 7x4

```
Test for image with height: 0px width: 0px
Memory for storing with doubles: 0
Memory for storing with bits: 0

Bits of image with height: 0px width: 0px
```

3.8. Тест на изображение 4x7

```
|Test for image with height: 0px width: 0px  
|Memory for storing with doubles: 0  
|Memory for storing with bits: 0  
  
|Bits of image with height: 0px width: 0px
```

3.9. Тест на изображение 7x7

```
Test for image with height: 0px width: 0px  
Memory for storing with doubles: 0  
Memory for storing with bits: 0  
  
Bits of image with height: 0px width: 0px
```

3.10. Тест на изображение 32x16

3.11. Тест на изображение 16x32

3.12. Тест на изображение 32x32

```
Test for image with height: 32px width: 32px
Memory for storing with doubles: 24576
Memory for storing with bits: 10289
```

3.13. Тест на изображение 32x20

3.14. Тест на изображение 20x32

3.15. Тест на изображение 36x36

```
Test for image with height: 32px width: 32px
Memory for storing with doubles: 24576
Memory for storing with bits: 11042
```

3.16. Тест на несъществуващо изображение

Error loading image

```
Test for image with height: 0px width: 0px  
Memory for storing with doubles: 0  
Memory for storing with bits: 0
```

Bits of image with height: 0px width: 0px