Gillian Schmitz

1835892

Abstract

This document contains the description and comparison of two different image compression algorithms, comparing both in their strong and weak points.

Image compression

For AZERTY

Contents

[Assignment 2](#_Toc138708993)

[The Plan 2](#_Toc138708994)

[Algorithm 1 – simple 2](#_Toc138708995)

[Colour Compression 2](#_Toc138708996)

[Run Length Encoding 4](#_Toc138708997)

[Rebuilding the image 4](#_Toc138708998)

[Algorithm 2 – complex 5](#_Toc138708999)

[Colour Space Conversion 5](#_Toc138709000)

[Chrominance Down Sampling 5](#_Toc138709001)

[Discrete Cosine Transform 6](#_Toc138709002)

[Quantization 7](#_Toc138709003)

[Run Length Encoding 7](#_Toc138709004)

[Rebuilding the image 7](#_Toc138709005)

[Comparison 8](#_Toc138709006)

[Conclusion 8](#_Toc138709007)

[Sources 8](#_Toc138709008)

# Assignment

The assignment was to create an app that uses image compression, for the online webstore AZERTY. AZERTY wants to use this app to compress the product images, decreasing the storage and bandwidth needed for the servers to operate.

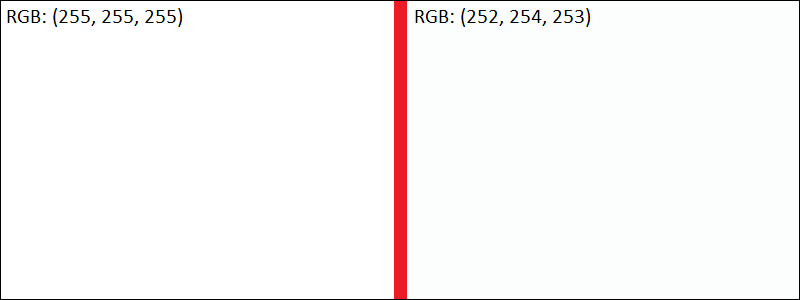
# The Plan

The plan is to compare two algorithms with each other, one being a simple but rough one. And the other being a more complicated one. Each of these algorithms have their pros and cons. The algorithms will be compared on different fronts, namely: compress/build time, general image quality, and file size.

# Algorithm 1 – simple

The first algorithm uses a very simple technique, it mainly uses the Run Length Encoding (RLE) algorithm. But before RLE, colour compression could be used to make RLE more effective. In short RLE uses a counter to count how many of the same colour is aligned after one another, for instance instead of saving “white, white, white, white”, “white x 4” is saved. Understanding the concept of this helps with why colour compression is used.

## Colour Compression

With colour compression, the colour values of the pixels are generalized into rougher values. This will make our RLE algorithm more effective. This is because a lot of the time a pixel has just a slightly different value, while it is seemingly the same colour. The RGB value (255, 254, 255) is almost indistinguishable from the (255, 255, 254) RGB value. They are both so close to being white that it takes a persona long time to distinguish the colours, and even then, it’s still very difficult to do. 

So, each RGB value is rounded down to generalize the colour values. The RLE algorithm is going to be better at picking up multiples of that rounded colour. This is especially effective for an object with the same colour, because each pixel will vary slightly because of the light/shading creating very small differences in the RGB values of the pixels.

The colour compression algorithm used in this algorithm is very simple, it takes the original value from 0 to 255 and floors it to the closest multiple of four. It achieves this using the modulo operator.

(6, 3, 1) → (6 – 6 % 4, 3 – 3% 4, 1 – 1 % 4)

= (6 – 2, 3 – 3, 1 – 1)

= (4, 0, 0)

(4, 1, 3) → (4 – 4 % 4, 1 – 1% 4, 3 – 3 % 4)

= (4 – 0, 1 – 1, 3 – 3)

= (4, 0, 0)

Thus:  
(6, 3, 1) → (4, 0, 0)

(4, 1, 3) → (4, 0, 0)

Listed below are some examples using the colour compression with different rates of compression. The npy-file is the original sized data, the json-file and pickle-files save the compressed data. Both the json- and pickle-files were saved to test which one was better at saving the data. It makes sense for the json-file to be larger, since it saves the data using text. The json-file was further only used to read out the data, so this is removed in the final version.

Using % 4:

A screenshot of a computer

Description automatically generated with medium confidence

Using % 2:

A screenshot of a computer

Description automatically generated with medium confidence

Without colour compression:

A screenshot of a computer

Description automatically generated with medium confidence

When you compare the pickle-file with the npy-file between the different rates of compression you can see the difference it makes for the RLE algorithm to be able to store more pixels in a single value. Without compression the RLE algorithm could make 1404KB from the original 1611KB. Whilst using the %4 made it 772KB, which is a lot less data.

## Run Length Encoding

As said before, the Run Length Encoding algorithm groups adjacent pixels, giving 4 white pixels that come after one another the value of “white x 4”.

In this example is shown how it would work on a 4×4 matrix of pixels.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[“Black”, “Black”, “Black”, “White”] [(“Black”, 3), (“White”, 1)]

[“White”, “Black”, “Red”, “Red”] → [(“White”, 1), (“Black”, 1), (“Red”, 2)]

[“White”, “White”, “White”, “Black”] [(“White”, 3), (“Black”, 1)]

[“White”, “Red”, “Black”, “Black”] [(“White”, 1), (“Red”, 1), (“Black”, 2)]

Especially for long rows of the same colour (which is very common in product pictures because of the background) this seems very effective. Sometimes compacting a row of more than 1000 values to just a tuple with one value and an integer.

## Rebuilding the image

To reconstruct the image all we must do in this case, is to fill each row of a 2-dimensional matrix with the values, repeating the values the number of times it was counted by the RLE algorithm earlier.

In short, the colour compression made the Run Length Algorithm go from reducing the data by 12,85% to 52,08%.

100-1404/1611\*100 ≈ 12,85%

100-772/1611\*100 ≈ 52,08%

# Algorithm 2 – complex

The second algorithm is based on the JPEG image compression algorithm, to reconstruct this method 5 main steps are used:

1. Colour Space Conversion
2. Chrominance Down Sampling
3. Discrete Cosine Transform
4. Quantization
5. Run Length Encoding

Compared to the simple algorithm, this is much more complex, but this algorithm also has its downsides. It generalizes groups of pixels a lot, now this is barely noticeable on pictures of nature, where there’s a lot of smooth shifting of the colour of the pixels. But in pictures of products, it is much more apparent. A sharp edge where one side of the edge is a solid white, and the other is the product (a lot of the times black), should have a sharp and thin edge. But as mentioned earlier, by generalizing the pixels in groups of 8×8 it creates a lot of opportunity to bleed out that edge, making it less sharp.

## Colour Space Conversion

The first step of the algorithm is Colour Space Conversion. A regular image is made from a matrix of pixels, with each pixel having 3 colour values: red, green, and blue (RGB). But RGB aren’t the only colours you can use to create an image. The JPEG algorithm uses the YCbCr colour space, this colour space consists of Luminance (Y), Blue Chrominance (Cb) and Red Chrominance (Cr). The Luminance layer has the black/white values from the image, whilst the Blue- and Red Chrominance layers consists of the colours, that altogether create the same image as when you would use the RGB colour space.

This conversion uses a matrix of constants to convert the RGB values to the YCbCr colour space.

## Chrominance Down Sampling

The chrominance layers can now be down sampled, keeping the sharpness by using the Luminance layer. This creates a – still good-looking – image when we deleted half of the data we started with. Both the Chrominance layers are 1/4th the original size.

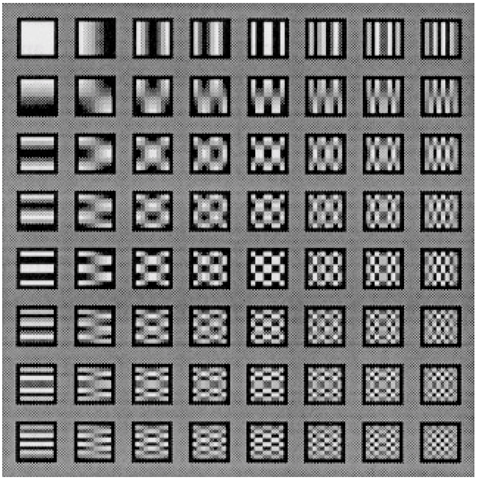
Y + Cb + Cr → 1 + 1 + 1 = 3

Y + Cb + Cr → 1 + ¼ + ¼ = 1.5

The down sampling on the chrominance layer works by taking a 2×2 square of pixels and putting the average of those four values in the new pixel. The old layer is deleted and when reconstructing the image, the layers that are down sampled get upscaled back to the original resolution when rebuilding the original image. It is important to remember that after this step, the colour channels are separated in each their own arrays.

## Discrete Cosine Transform

Discrete Cosine Transform (DCT), along with two other steps, is a way to remove unnecessary data from an image, DCT makes use of the fact that the human eye isn’t good at looking at high frequency elements. First the image is divided in blocks of 8×8 pixels. DCT uses a set of base images and uses that set of images to rebuild that block of 8×8 pixels.



This table contains the base (8×8) images that are being used to recreate any image. The lower frequency parts reside in the top left corner, while the higher frequency parts are in the bottom right corner.

A picture containing text, font, handwriting, white

Description automatically generated

By applying the calculation to a block of pixels, you end up with 8×8 blocks that have the frequency of which base image was used. With this, you can separate the high frequency parts of an image from the low frequency parts. This step is performed for all the colour channels individually. With the next step, we can remove a lot of the data that is less important.

## Quantization

Now that we separated the high frequencies with the low frequencies, we can remove the higher frequencies. To do this, we divide the 8×8 table of frequencies with our quantization table and then rounding to the closest integer.

Base table used:  
A screenshot of a computer

Description automatically generated with medium confidence

The table has higher values in the bottom right, and the top left have smaller values. This means when dividing it with the table of frequencies, the result of this is a matrix where there are a lot of zeros in the bottom right where all the high frequency data is stored. Essentially deleting only the high frequency data of the image, deleting only the data from the image that wasn’t very noticeable to the human eye anyway.

## Run Length Encoding

The last step gave a matrix which contains a lot of duplicate data (a lot of zeros), RLE is used to store those zeros efficiently. Since every base block was made with the same (8×8) format, the matrix can be flattened to one list of 64 elements which further increases the effectiveness of the RLE algorithm.

## Rebuilding the image

To rebuild the image the previous steps used must be used in reverse, to start the RLE algorithm is reversed. Reverting the RLE results in a list of 64 values, the block-size is then used to revert the 64-element list to a 2-dimensional, 8×8 matrix. Doing this for all the blocks within the compressed data, results in a 2-dimensional matrix of 8×8 blocks.

The next step is to revert the quantization and apply the inverse DCT on the blocks, the inverse of DCT is also known as DCT-III. To apply dequantization we mean to multiply every 8×8 block with the same quantization table that was used to reduce the data in the original DCT step in the compression algorithm. After a block is dequantized, DCT-III is used to revert the changes that the original DCT algorithm made. The DCT-III algorithm converts the (8×8) matrix of coefficients back to a matrix close to the original matrix of pixel values that was used to compress the image.

A picture containing text, font, handwriting, line

Description automatically generated

After DCT, the chrominance steps need to be rescaled to the original resolution, since the luminance channel was not down sampled, we use that as the base to rescale the chrominance channels. That is because the resolutions might differ slightly after rebuilding.

The last step to rebuild the image is to convert the values back to the RGB colour space. This is done with a matrix vector multiplication with a table of constants.

# Comparison

When comparing the two algorithms there are a couple main things to look at, namely: compress/build time, general image quality, and file size.

To start with compression- and build time, the compression- and build time are for each algorithm individually equal. Both the build- and compression time for the simple algorithm are very fast and done within seconds. While the complex algorithm is much longer than that. Depending on the image size it may take half a minute for small images, and multiple minutes for larger sized images.

The image quality for the small image is not very noticeable at first, but if you compare it next to the original you can see within seconds that the shift in colour/shading is

# Conclusion

# Sources