Gillian Schmitz

1835892

Abstract

This document contains the description and comparison of 2 different image compression algorithms, comparing both in their strong and weak points.

Image compression

For AZERTY

Contents

[Assignment 2](#_Toc138521945)

[The Plan 2](#_Toc138521946)

[Algorithm 1 – simple 2](#_Toc138521947)

[Colour Compression 2](#_Toc138521948)

[Run Length Encoding 4](#_Toc138521949)

[Rebuilding the image 4](#_Toc138521950)

[Algorithm 2 – complex 4](#_Toc138521951)

[Colour Space Conversion 5](#_Toc138521952)

[Chrominance Down Sampling 5](#_Toc138521953)

[Discrete Cosine Transform 5](#_Toc138521954)

[Quantization 6](#_Toc138521955)

[Run Length Encoding 7](#_Toc138521956)

[Comparison 7](#_Toc138521957)

[Conclusion 7](#_Toc138521958)

[Sources 7](#_Toc138521959)

# Assignment

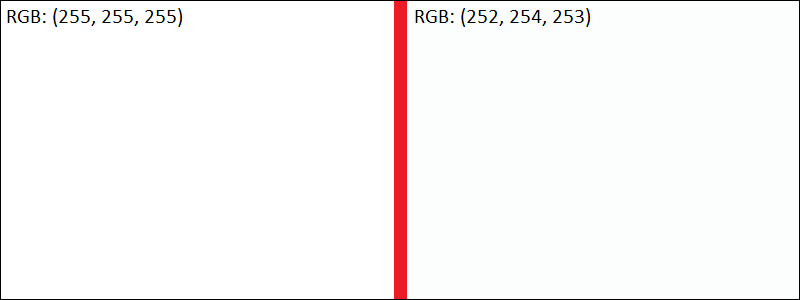
# The Plan

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.

# Algorithm 1 – simple

The first algorithm uses a very simple technique, it mainly uses the Run Length Encoding (RLE) algorithm. But before RLE you can use colour compression 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”, you would save “white x 4”. Understanding the concept of this helps with why we use colour compression.

## Colour Compression

With colour compression we mean to generalize the colour values of the pixels, 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 (0, 1, 0) is almost indistinguishable from the (0, 0, 0) RGB value. They are both so close to being black that you would really have to look for it to find it, and even then, it’s still very difficult to do. 

So, if we round each RGB value down we can generalize the colour value, 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 4. 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” 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.

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 1070KB, which is a lot less data.

## Run Length Encoding

As said before, using 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) 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, essentially 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 van 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.

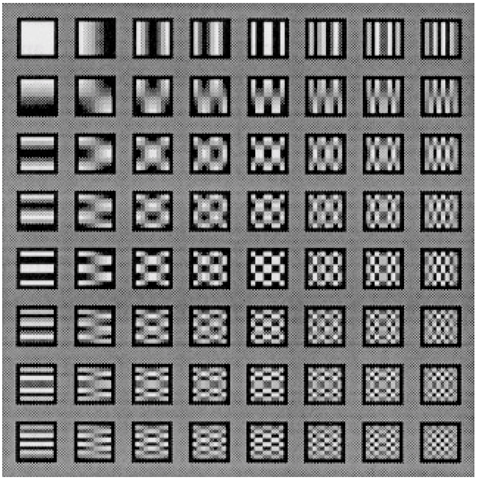
Y + Cb + Cr → 1 + 1 + 1 = 3

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 4 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, creating the full image.

## Discrete Cosine Transform

Discrete Cosine Transform (DCT) 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 8x8 pixels. DCT uses a set of base images and uses that set of images to rebuild that block of 8x8 pixels.



This table Contains the base 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.

By doing this you end up with 8x8 blocks that have the frequency of the base image that was used. With this you can separate the high frequency parts of an image from the low frequency parts. 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 8x8 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 our table of frequencies, we end up with a lot of zeros in the bottom right where all the high frequency data is stored. Essentially deleting that high frequency data of the image, deleting only the data form the image that wasn’t very noticeable to us anyway.

## Run Length Encoding

Now that we have blocks that contain a lot of zeros, we can save those zeros efficiently using RLE. Since we made every base block the same (8x8) format, we can flatten the block to one list of 64 elements which further increases the effectiveness of the RLE algorithm.

# Comparison

# Conclusion

# Sources