Computer Vision, Spring 2017, Seoul National University

Term Project: Research into image compression techniques

Horner Benjamin, Steiner Patric, Penas Pastilla Francisco Javier

Table of Contents

1. Introduction

The purpose of this project is to research and understand the approaches to and underlying theory behind modern compression techniques. Subsequently, to understand the differences in these techniques by implementation, experimentation, and evaluation.

2. Approach

Our initial approach will be to attempt develop our own image compression format as a way to understand the problems in the topic first hand. After evaluating this process and any results obtained, we will then proceed to research around the most prominently used compression formats (such as jpeg and png) to discover how these formats tackled the problems that we faced.
We will then attempt to evaluate the practical implications and trade-offs that these commonly used formats offer. With this new knowledge in hand, we will return to our initial algorithm and either improve upon what we created or implement one of the more common formats.

3. Goals

Firstly and most obviously, obtain knowledge and expertise around the topic of image compression.
Secondly and more importantly, obtain first-hand experience in tackling problems faced in modern computer science and learn from the efforts of previous work.

4. Potential Datasets for experimentation

For this project, we will use a variety of different images to experiment what type of images are easier to compress and what type of images cannot be compressed as well.

5. Potential Extension

If the main part of the project goes sufficiently to plan and in good time, the natural extension would be to perform the same process for video compression techniques.

6. Image representation

An image is basically a matrix of pixel values. A pixel can be represented in many different ways.

Three common ways are described in the following sections.

6.1. Additive color model (RGB)

This model is commonly used to represent images on a screen. Each pixel is represented by three different values:

- One value for the red component of the pixel
- One value for the green component of the pixel
- One value for the blue component of the pixel

Every value can be in the range from 0 to 255. If we want to get the color yellow for example, we mix 255 red with 255 green. This way, all possible colors can be created.

6.2. Subtractive color model (CMYK)

This model is usually used for printing. In a printer, there are usually inks of the colors cyan (C), magenta (m), yellow (y) and black (k). To get black, all three colors could just be mixed, but in practice it’s better to have separate black ink, so a clearer black is achieved.

6.3. YCbCr model

This model uses a completely different approach to represent colors. Here every color is represented by a value between 0 and 255 for the luminance (Y) and 2 values between -128 and 127 (Cb and Cr) to represent a chroma component (blueness and redness or greenness and purpleness when negative respectively).

Example use of this model is TV broadcasting, because it allows to easily downsample the color components, which are not as important as the luminance component for the human eye.
The jpeg compression algorithm also makes use of this advantage and uses this format to remove the high frequency data of an image (which is barely noticeable for the human eye).

6.4. Color model and image representation for this project

Our goal is to create and experiment with image compression algorithms. To simplify this task, we decided to only use grayscale images at first, this means that each pixel only consists of one single value between 0 and 255 that represents the grayness (0 = black, 255 = white) of the pixel.
In later stages of the project, when we got our algorithm running, we were easily able to extend it to the RGB model by just running it for every color component once.

7. Idea for compression algorithm

Our basic idea to compress images is to find pixel groups that are equal and replacing them by a placeholder. The original values are stored in a dictionary together with the placeholder, so no information is lost (the method is lossless) and we are fully able to reconstruct the original image.

It is quite obvious that this method would work better or worse for different kinds of images. The best case scenario would be a fully one-colored image. Such an image could be represented by one single group of pixels that occurs exactly once, namely as the whole image. If the image contains repeating patterns, the algorithm would also work quite well, since it could just store the pattern once and then replace each occurrence of the pattern with a single placeholder value.

If the image does not contain many patterns, like for example very colorful and noisy pictures, the algorithm would not be as successful, meaning the compressed size would barely be lower than the original size.

8. Problems & Solutions

8.1. Representing Placeholders

The first problem we faced was how to represent placeholders as they replace the pixels. The solution we used was to represent them as codes starting at 256 and increasing by 1 for each code found. This ensures that, when decompressing, the codes can never be mistaken for a pixel value, as they must be between 0 and 255.

8.2. Finding Patterns

The first and main problem that we had to overcome was the method of finding repeated pixels to replace. For simplicity, we compressed the image into a 2D vector, considering the separate colors for a pixel as three separate values. This technique, whilst saving complexity, has an obvious drawback of losing spatial information which is an issue that we will consider in section 9.

Next we would look for pairs of repeated values to replace, repeating this process on the new vector so that any patterns longer than two values will again be replaced until no more patterns are present. As an additional parameter, we introduced a threshold of a minimum number of repetitions for a pair to be replaced to experiment if different thresholds increased performance.

Now as for practically looking for pairs, the naïve implementation would be to simply compare each pair to each other. While spatially efficient, the time complexity of this algorithm is O(n^3), far too inefficient to be seriously considered. Instead we decided to perform an initial pass over the vector and hash each pair to a bucket, on a second pass if the value of the bucket is equal to at least the threshold, we know that the pair is to be replaced. This new algorithm is at worst O(n^2) and best case O(n(logn)) – in the case of an all-white image for example.

To ensure that the hash function never hashes any pair to the same value, the size of the hash table must be equal to at least the square of the number of codes found. With a low threshold and large image, this quickly becomes too large for MATLAB to store. We then discovered how to represent matrixes as sparse, only storing non-zero values, which enabled us to continue with this algorithm.

8.3. Replacing Pairs with Codes

Another problem we had was that if three consecutive values could be replaced with two codes. If one code was replaced in one location, and the other code in a second, on the next iteration over the vector, the patterns in these locations would not be extended. In other words, pattern replacement must be deterministic.

While we initially had three passes – the second pass for assigning codes from the hash, and the third for replacing values in the vector – we realized that only two was needed. On the second pass, if a code is found, replace the hash with the negative of that code. Now if a bucket contains a negative number, we know to replace the value in the vector with that code immediately, and the next value with NaN. Whilst saving time, this also ensured determinism in replacing codes. The values that are NaN can easily be removed from the vector by only taking those indexed by "~*isnan*", a MATLAB function.

8.4. Storing the Compressed Image

To complete the process, we needed a method to physically represent the compressed file. We decided that using Huffman encoding would be an effective approach.

9. Results

To test the process we used four images that can be seen in Figure 1. Figure 1a is a small simple image that we shall call "SNU Paint"; Figure 1b is an unremarkable picture of a cat that we shall call "Cat"; Figure 1c is an image of a night sky that we shall call "Night"; Figure 1d is a completely white image that we shall call "White".

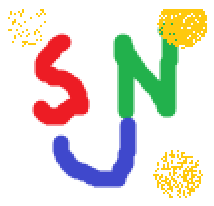

Figure 1a: "SNU Paint"
100 x 100

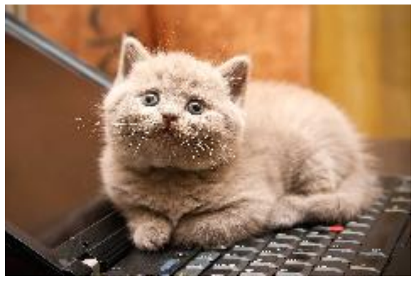

Figure 1b: "Cat"
160 x 230

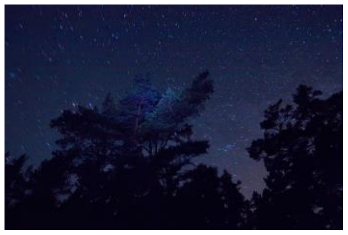

Figure 1c: "Night"
426 x 640

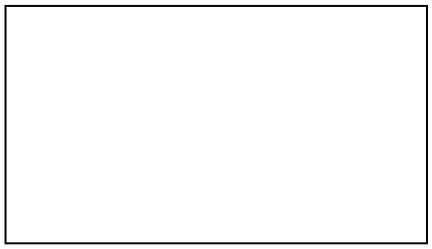

Figure 1d: "White"
648 x 1152

Figure 1: Pictures Used

SNU Paint provided a simple test for our algorithm, Cat provided a test on a typical picture, Night was a typical picture that we believed our algorithm would perform well on, White provided an ideal example.

9.1. Initial Compression

Our first tests where to determine which threshold produced the greatest reduction in number of values to represent the image. The results can be seen below.

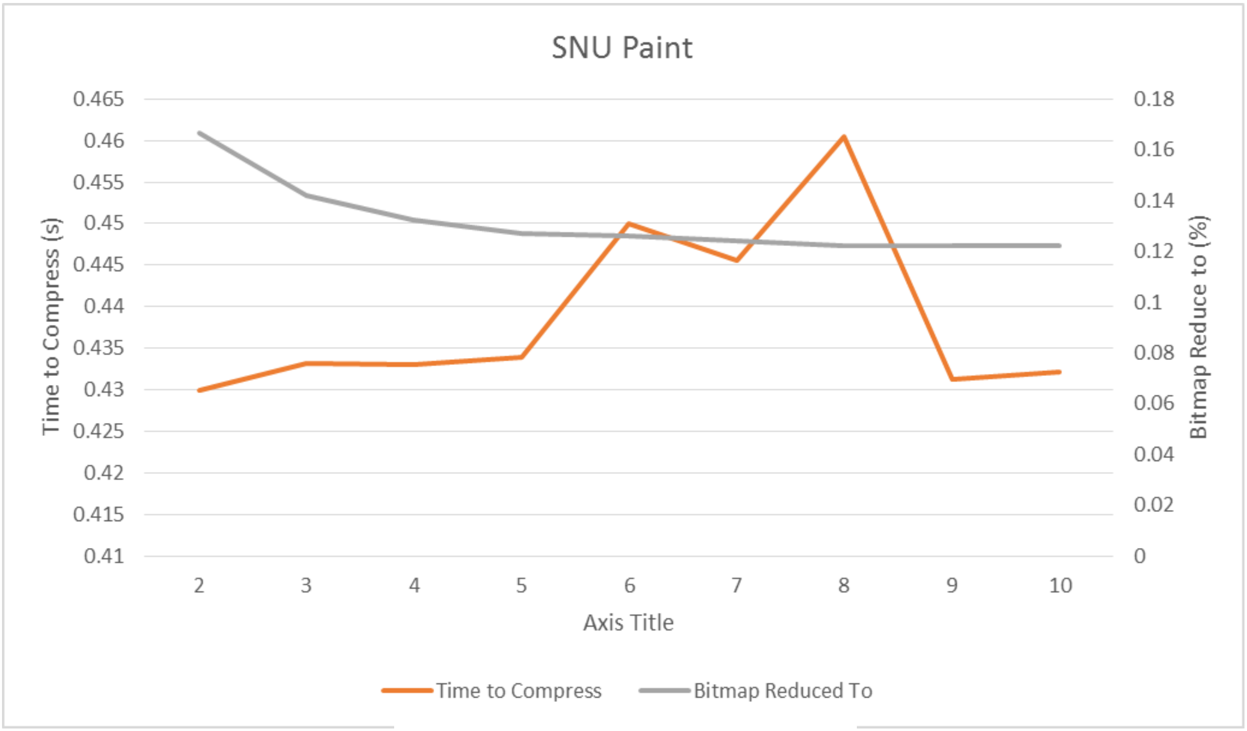

Figure 2a: SNU Paint Compression

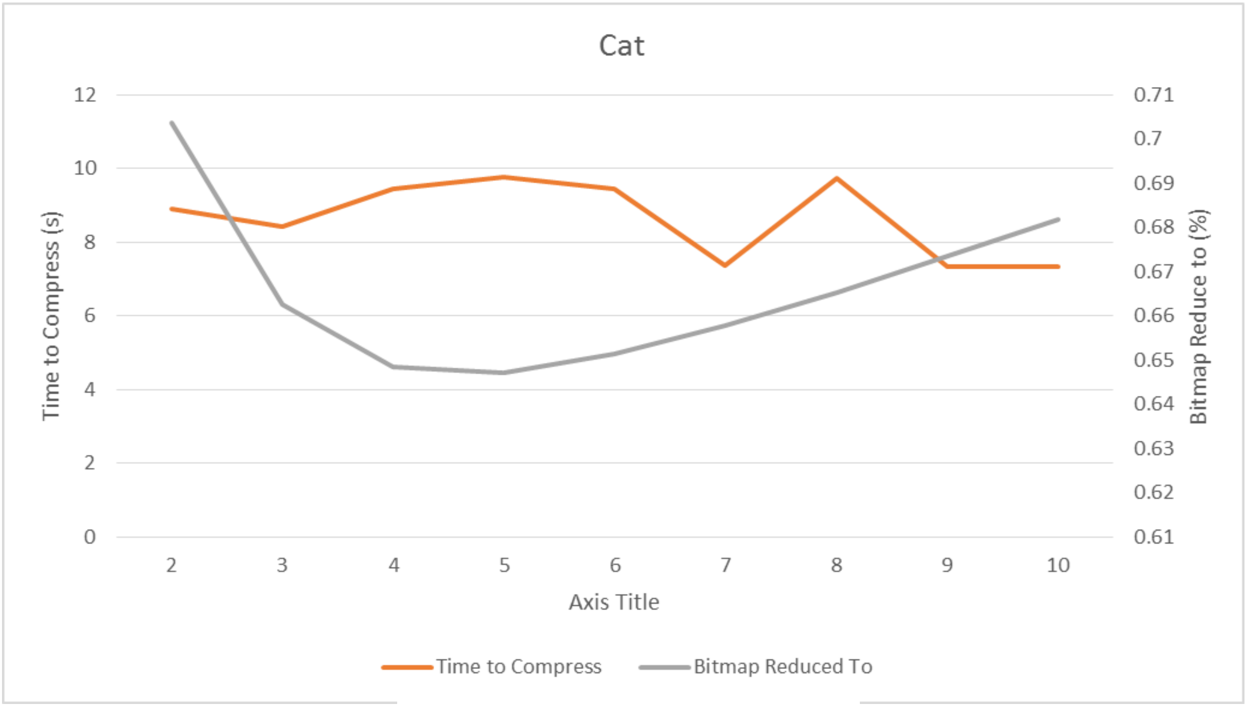

Figure 2b: Cat Compression

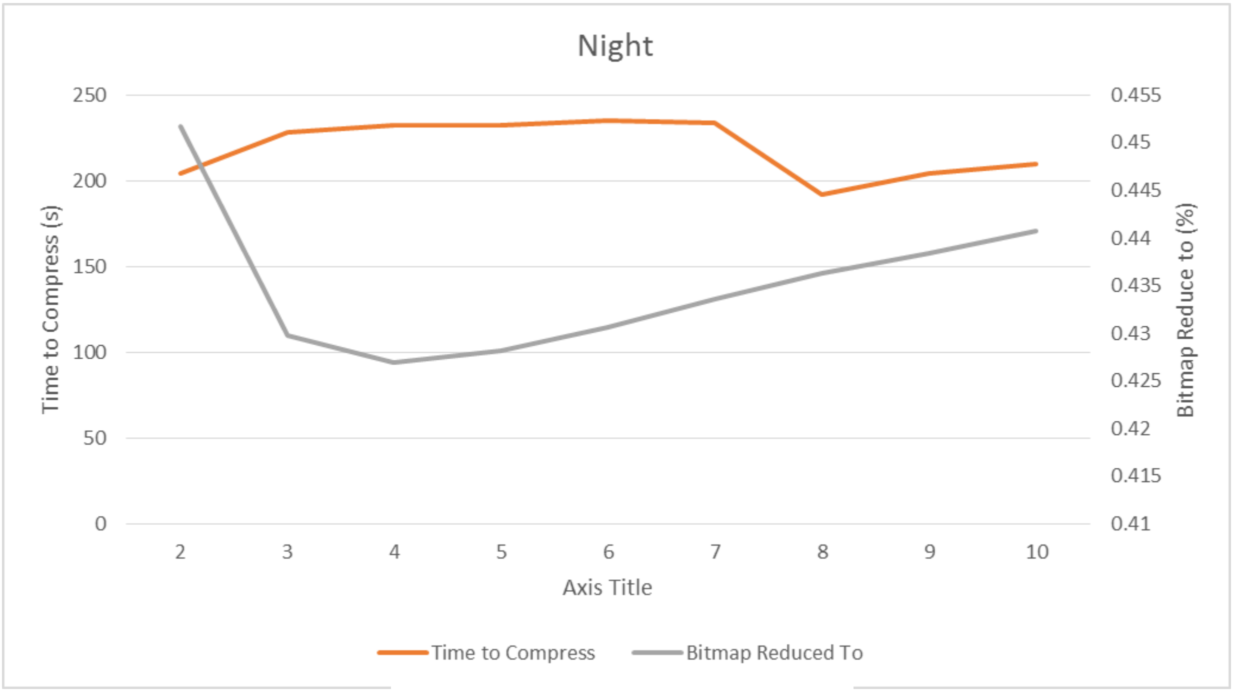

Figure 2c: Night Compression

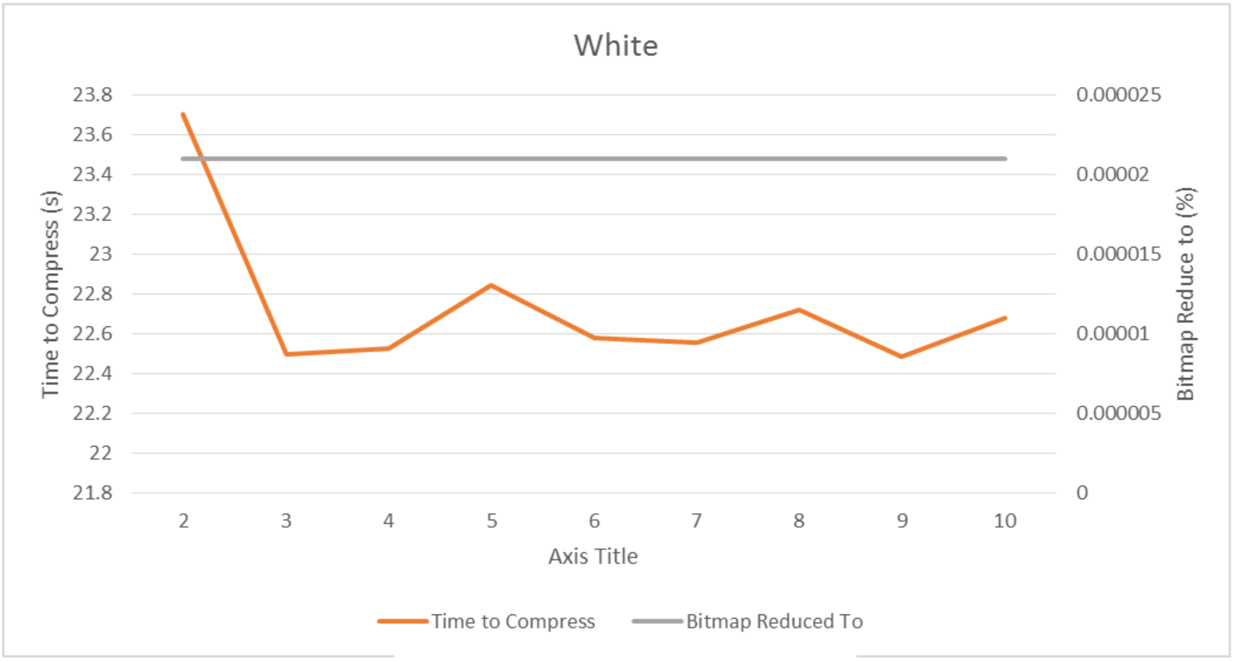

Figure 2d: White Compression

As it can be clearly seen, using 2 (and to some extent 3) as a threshold, the increases in dictionary sizes outweigh the reduction in image size. For the two conventional pictures – Cat and Night – a threshold of 4 provided a mean reduction to 0.5378, while 5 provided a mean reduction to 0.5376.

Some other interesting points are: the length of time that compression of Night took, an issue we discuss in the conclusion; the effectiveness of the algorithm compressing White.

Some additional experiments that we would like to have conducted was testing different thresholds relative to the size of the image.

9.2. Encoding

While using a threshold of five provided, on average, the greatest reduction of values to describe an image, this was not the case after encoding the results. This was due to the large number of unique codes requiring many bits to represent them. After some experimentation, we discovered that on average, a threshold of 25 produced the best results. The final sizes that we reduced the images to can be seen in Figure 3.

The method we used to calculate the size was a sum of the following values converted to bytes:

- The bits in the Huffman bit-stream
- The bits used in the Huffman dictionary
- The minimum number of bits required to represent the keys of the dictionary – e.g. if the largest key is 255, all keys would be stored with 8 bits.
- 8 bits to store the size used above – in the same example, this value would be 8.
- 4*16 bits to store the dimensions of the image and Huffman dictionary.

White this calculation is not quite exact, the Huffman bit-stream far outweighed the other terms and so the error is negligible.

	Huffman on Raw Image	**Our Size**	**PNG Size**
SNU Paint	9.12 KB	4.25 KB	3 KB
Cat	113 KB	104 KB	72.9 KB
Night	637 KB	538 KB	79.0 KB
White	279.94 KB	39.38 B	3.52 KB

Figure 3: Encoded Sizes

We also included the size when Huffman encoding the image before compression to compare to our result and the size of the PNG file.

In the first three pictures, while our compressed and encoded size was larger than the PNG size, it was a large reduction from simply using Huffman encoding which we were pleased with. We were hoping to achieve a better result on the Night picture this was in fact the largest in relation to its PNG size. This is because, due to the larger image size, the bit stream was simply far too large, taking up 532KB of the final size.

However, it should be noted that, on the White image, the algorithm performed extremely well, reducing the image to a fraction of its PNG file size.

10. Improving the algorithm

10.1. Reduce compression size by preprocessing the image

As described in section 7, the proposed algorithm works better if the image is less random basically. So our initial idea was to preprocess the images to reduce noise and gradients so there are more repeating pixel patterns to be found and thus the image can be compressed more efficiently. Additionally, fewer codes would be required and so the encoding would be more effective also. However, there is the tradeoff that now the whole process is not lossless anymore but lossy instead. We will never be able to reproduce the original image 1:1 after we compress the preprocessed image.

The following sections describe our experiments and results with this idea to preprocess the image.

10.1.1 Using a small Gaussian filter

Our first idea was to smooth the image with a Gaussian filter with a low standard deviation, so noise is reduced and the whole picture is kind of evened out. We expected to be able to find more repeating patterns this way. However, it turns out that this idea makes it even harder for our algorithm to find patterns, since not only noise is removed, but also edges are smoothed, which means there are now small gradients. And gradients turned out to rarely be repeating.

So in conclusion, this idea did not work at all and we discarded it.

10.1.2 Sharpening the image

If smoothing produces a worse result, we tried to sharpen the images before applying our algorithm, to achieve the opposite of what the previous idea did. As it turns out, for certain images, using a sharpening filter can actually make it easier to find repeating patterns in the image. It works best for low-contrast images or images with gradients, but not so much if the image is already very basic because if we sharpen a perfect checker-board for example, we will end up with more color values!

10.1.3 Rounding the image

The most straightforward approach we came up with. We just preprocess the whole image by rounding all the pixels to a multiple of X. The bigger the X, the bigger the impact on the image quality (in a negative manner) and compression ratio. By using this tactic, we sacrifice some image quality but we can always increase the compression ratio.

10.2. Dictionary Reduction

Reducing the size of the compression dictionary would mean less unique values when performing Huffman encoding, resulting in smaller files. We did attempt a method to store codes that consist of the same value, x, as “2*x” instead of “x x”, meaning that we could possibly remove the code x. Unfortunately, due to a great deal of dependence within the dictionary, using this method there were very rarely any codes that could be removed.

Figure 4 shows the reduction in dictionary size with this method. Additionally, as we would use negative numbers to represent multipliers instead of values, it would cause there to be more unique values.

	SNU Paint	Cat	White	Night
Reduced	1	1	15	0

Figure 3: Encoded Sizes

Other approaches to this problem might be considering larger size groups of values, triplets instead of pairs for example.

10.3. Spatial Information

As mentioned before, because we reduced the image to one dimension, much spatial information was lost that could be used to aid results. A possible approach to this problem would be to consider pairs in multiple directions. This would also increase the amount of pairs found and reduce the number of codes required.

This would however require a different method of storing codes, perhaps with an indication for with dimension the code expands in. It would also require a different method of storing the compressed image as once an iteration of the algorithm has run, the rows and columns of the image would no longer be uniform.

10.4. Maximum Pair Selection

Instead of replacing the pairs in order along the image, we could instead first select codes with greater frequencies. This would likely slightly reduce the number of codes produced. Additionally, this policy is deterministic and so still represents a solution to the problem in section 8.3.

11. Conclusion

While our algorithm did indeed compress the images, the reduction and runtime left much to be desired, particularly when compared to modern compression algorithms. The main conclusion that we are left with is that it is simply not efficient (in terms of time and results) to consider images on a pixel by pixel basis, compression algorithms should act upon the image as a whole.