Lossless Monochromatic Image Compression Algorithms

Alexandre van Velze, Mário Lemos, Guilherme Almeida

Departamento de Engenharia Informática, Universidade de Coimbra, Coimbra, Portugal

20192126618, 2019216792, 2019224555

***Abstract* — With the amount of data used increasing every day, it is fundamental that we find good ways of minimizing data size. In this paper we take a look at some of the techniques used to compress images, a process that aims to reduce the size of the image.**

***Keywords* — Compression, Lossless Image Compression, Monochromatic Image, Compression Ratio**

I. Introduction

Everyday thousands of Terabytes of data are produced and transmitted all through the web, so it's of extreme importance to create a way to compress it as it would be near to impossible to move this much data in its raw format. Compressing the amount of data required to represent digital images reduces the cost of storage and cables transmission over a limited bandwidth channel.

There are two methods of compression that can be done on images: "lossless" "lossy".

Lossless compression is done by representing a file's data using less bits, without losing any information. We can achieve this by using mathematical and statistical tools like entropy coding and transforms. This type of compression is used when it is necessary to keep all the information intact, as such in cases of executable files.

Lossy compression techniques remove unnecessary bits with reduces the size of the file. Once compressed the file can no longer be converted back to the original, so when decompressing it what we get is an approximation to the original file. This type of compression has a higher compression ratio when compared to lossless. It's also a great candidate to compress images since the human eye usually can't notice the difference.

In this paper we are going to address lossless compression algorithms only, so any lossy techniques won't be mentioned in here.

II. State of the Art

In order to find the best method to compress monochromatic images, first we need to understand and know more about the compression algorithms that are being used to this day.

A. Compression Algorithms

- Run-length Encoding (RLE): The idea behind RLE is to replace sequences of the same identical symbol with pairs (run, value), where the value is the symbol being repeated and run is the number of times that symbol is repeated. There are multiple ways to implement this method, based on the format used.

  For example:

  0 0 1 0 0 0 0 0 1 1

  Using Run-length Encoding:

  (2, 0) (1,1) (5, 0) (2, 1)

  Figure 1: RLE Algorithm

  This method is effective to compress graphic images, as they contain sufficient long bytes series of repetitive sequences. It is badly suited to code images like photographs. It's used in ILBM and PCX format files.

- Huffman Encoding: This method removes coding redundancy from the image and is more efficient compared to RLE. It uses the frequency of the characters correlating it with code words, so symbols with a higher probability of occurrence are represented by a smaller code word than the ones that occur less often. A tree called Huffman Tree is created as the symbols are analyzed, and it is complete when the total probability reaches 100%.

  This method is used in the last step of JPEG and TIFF formats. It requires a greatly amount of calculations, this results in a slow and resource-intensive coding process.

- Arithmetic Coding: Developed by Jorma Rissanen and Richard Pasco, it's a method that usually has a higher compression ratio than Huffman coding. This method is mainly useful when the alphabets of the sources we are dealing with are small, like a binary source. It also comes in handy when we need to keep the encoding and decoding procedures separate from modeling aspects, which allows the source statistical model to be changed without requiring any additional steps.

  The first step is to create a table from a statistical model, where we list the probability of each symbol in the alphabet being the next read symbol. From that we represent the probabilities in intervals between [0, 1[, where the first symbol starts at 0, and the last symbol ends in 1. Using that interval, we identify the sub-interval that corresponds to the first symbol. Now for each symbol following, we divide this interval in the same proportions as the original, and find the sub-interval that matches the next symbol. After all these steps, we end up with an interval that corresponds to the probability of each symbol's occurrence in correct order.

By calculating the average of the upper limit and the lower limit of this interval, we end up with a number with the name of Tag. The tag is then used to generate the binary code.

- Lempel-Ziv Coding (LZ77): Developed by Abraham Lempel and Jacob Ziv in 1977, it's different to the previous ones mentioned because it does not require prior knowledge of the source statistical information. It's based on exploiting the repetition of words and phrases in a file. The source file is analyzed one character at a time, while maintaining a sliding window (buffer) which stores the occurrences of previous symbols. As we go through the file, when a repeated symbol is found, it's stored a reference to the previous occurrence of that exact symbol, along with the number of occurrences of that symbol.

  There are multiple variations of this method, such as LZ78 and LZW. The main difference in these variations are the way the dictionary is implemented.

- Lempel Ziv Welch (LZW): This method is known for being simple to implement and for its very high throughput in hardware implementations. It works based on the amount of multiplicity of bit sequences in the pixel to be encoded. It also takes an error free approach that focus in removing spatial redundancy.

- Lempel-Ziv Markov Chain Algorithm (LZMA): Variant of LZ77, features a high compression ratio and a dynamic dictionary which lowers the memory usage. It has good decompression speeds, but it takes a hit on compression time. It was first used in the 7-zip archiver. It uses the sliding window algorithm based in the LZ77, a delta filter and a range encoder. Firstly, the delta filter is used to transform the data making it more suitable for the sliding window mechanism. To the output data from the delta filter it is applied the sliding dictionary algorithm. After this step it is finally used the range encoder, which encodes the symbols with numbers based on the frequency at which these occur. This algorithm is suitable for real time data compression.

- Deflate: This algorithm was developed by Philip W. Katz. It combines LZ77 and Huffman, and nowadays it's used in software such as gzip and 7-zip. The input data is divided into blocks, and LZ77 is used to find duplicated series of characters and replace them with a reference to the previous occurrence in each block. Huffman encoding is then used to assign code words to the symbols based on their occurrences. The Huffman trees for each block are independent of those for previous or subsequent blocks.

  A big disadvantage of this method is it only finds duplicates in a 32Kbytes window.

- Prediction by partial matching (PPM): This method is used an adaptive data statistical compression mechanism that is context and, as its name suggests, prediction based. The prediction comes in play when it uses sets of previous symbols in the original uncompressed stream to predict the next symbol. This works thanks to the rank assign to each symbol before the compression occurs, and its correspondent codeword will be determined by a ranking system. PPM is now mostly used for compressing natural language texts, since it is one of the best lossless compression methods for these cases.

B. *Image-Based Compression Algorithms*

- FELICS (Fast Efficient Lossless Image Compression System): As the name indicates this algorithm as very fast run time at a low loss of compression efficiency. It performs five-times faster than JPEG codec while having a similar compression ratio. The first two pixels in the first row are directly packed in to the bitstream. For the following pixels it is attributed two neighbors' pixels to create a prediction for the probability distribution of its intensity. While calculating the intensity of the current pixel the probability is going to be higher in range of the maximum and minimum intensity of both the neighbor's pixels. With this in mind it is then needed to apply binary adjustments code in range and Golomb-Rice code below and above range. This algorithm is VLSI-oriented.

- Context Based Adaptive Lossless Image Coder (CALIC): This method obtains a higher level of lossless compression of an image than other lossless compression codecs, this being achieved with high efficiency, given it's accomplished in relatively low time and space, thanks to efficient techniques for forming and quantizing modeling contexts. One of its main features consists of the use of a large number of contexts to create a nonlinear predictor that learns from its mistakes under a give context and corrects itself, and adapt it to the various source statistics.

- Low Complexity Lossless Compression for Images (LOCO-I): This algorithm is mainly used for continuous-tone images, as JPEG-LS, that have a low entropy. It offers one of the best compression ratios at a very low complexity level. This algorithm as two essential steps: decorrelation/prediction and error modeling. In the prediction stage it is utilized the Median Edge Detection (MED) predictor to determinate the neighbor's pixel to use and it is predicted the pixel X according to the value of the neighbor pixel.

$$X = \begin{cases} \min(A,B) & \text{if } C \geq \max(A,B) \\ \max(A,B) & \text{if } C \leq \min(A,B) \\ A+B-C & \text{otherwise.} \end{cases}$$

Figure 2: LOCO-I Algorithm

- Portable Network Graphics (PNG): When the LZW algorithm was patented by Unisys, the community decided to create a new patent-free method to replace GIF. It was released in 1996 and has the name of PNG. Unlike GIF, PNG uses Deflate (combination of LZ77 and HUFFMAN Encoding), that has already been explained, but prior to applying Deflate the data is transformed using a prediction method (Filtering). The reason why PNG uses a filtering method first is because an image after being filtered is often more compressible than the raw image will ever be.

  PNG currently uses only one filtering method for the entire image, using one of five different filter types for each line in the image. The filter predicts the value of each pixel based on the values of surrounding pixels and subtracts the predicted color from the actual value.

Compared to GIF, PNG has a higher compression ratio in almost every case, but it does not support multiple image format, like animations.

PNG also supports a wide variety of colors and multiple levels of transparency.

C. *Conclusion*

Throughout this paper we have analyzed multiple lossless compression methods along with advantages and disadvantages for each one of them. With this we can conclude that some algorithms are better for text and other are better for different types of images.

III. Exploring the Methods and Analysing the Experimental Results

In order to create our own compression method, we first need to experiment with already existing methods, by analyzing what they do and the outcome result.

The images used to experiment with these methods were the following:

TABLE I. FILES USED IN THE EXPERIMENT

Name of the File	File Type	Size (MB)
egg	bmp	17.7
landscape	bmp	11
pattern	bmp	48
zebra	bmp	16.7

Every test in this experiment was done on PyCharm IDE on a Windows Machine. The computer running Windows has an Intel i5 8400 CPU and 16GB of RAM.

A. *Methodology*

We used BZIP2, LZMA and Deflate Python modules to compress the images and the results are showed in Tables II to IV.

For RLE, we created a simple Python program and the results are presented in Table V.

B. *Calculations*

- Compression Ratio

The compression ratio tells us how effective an algorithm is. When compressing an image, the bigger the compression ratio the bigger the compression, and we can calculate it with the following expression:

$$compression\ ratio = \frac{original\ size}{compressed\ size}$$

As we can interpret from the Tables, the image with the biggest compression ratio is “pattern.bmp” because of its characteristics (this is a binary image, and since it only has two possible pixel values it is very easy to compress). LZMA and BZIP2 have a very similar average compression ratio while RLE’s algorithm has the lowest ratio.

- Compression Speed

The time it takes to compress a file is a very important aspect when it comes to analyzing if an algorithm is good or not. Even if the algorithm has a very high compression ratio, if it has a low compression speed it might not be as useful for some types of tasks. We can calculate the compression speed using the following expression:

$$compression\ speed = \frac{original\ size}{compression\ time}$$

Deflate has a much higher compression speed than the others, topping the chart at 90 MB/s. LZMA and RLE have the lowest speeds.

- Decompression Speed

It’s also important to consider the time it takes to get the original file after it has been compressed. If this step is very slow, the algorithm most likely won’t be very useful in most cases. The expression used to calculate this is pretty much the same as the one above:

$$decompression\ speed = \frac{original\ size}{decompression\ time}$$

Deflate also has the highest decompression speed, making it one of the best algorithms mentioned here. LZMA and BZIP2 have average speeds. RLE decompression speed is not mentioned in our table because we were unsuccessful at creating the decompressing algorithm.
Since there is also more than one way to implement all of these methods, speed values can change based on the implementation.

Overall, the methods shown here are all good, and choosing one should be based on the scenario. Deflate is useful if you need really fast compression and decompression speeds, while BZIP2 is good if you’re interested in a good compression with normal speeds.

TABLE II. BZIP2 IMPLEMENTATION

Name of the File	Original Size (MB)	Comp. Size (MB)	Comp. Ratio	Comp. Speed (MB/s)	Decomp. Speed (MB/s)
egg	17.7	4.53	3.91	19.03	24.58
landscape	11	3.33	3.30	15.28	19.30
pattern	48	1.62	27.91	85.71	75
zebra	16.7	5.36	3.12	16.7	19.42
Average			9.56	34.07	34.58

TABLE III. LZMA IMPLEMENTATION

Name of the File	Original Size (MB)	Comp. Size (MB)	Comp. Ratio	Comp. Speed (MB/s)	Decomp. Speed (MB/s)
egg	17.7	4.59	3.85	1.33	31.05
landscape	11	3.08	3.57	1.24	28.95
pattern	48	1.84	26.08	4.34	94.12
zebra	16.7	5.40	3.09	1.09	24.93
Average			9.15	2	44.76

TABLE IV. DEFLATE IMPLEMENTATION

Name of the File	Original Size (MB)	Comp. Size (MB)	Comp. Ratio	Comp. Speed (MB/s)	Decomp. Speed (MB/s)
egg	17.7	6.30	2.80	61.03	252.86
landscape	11	4.24	2.59	47.83	220
pattern	48	2.41	19.92	200	218.18
zebra	16.7	7.26	2.30	53.87	208.75
Average			6.90	90.68	224.95

TABLE V. RLE IMPLEMENTATION

Name of the File	Original Size (MB)	Comp. Size (MB)	Comp. Ratio	Comp. Speed (MB/s)
egg	17.7	10.2	1.7	4.99
landscape	11	8.23	1.34	4.18
pattern	48	3	16	14.63
zebra	16.7	10.3	1.62	4.69
Average			5.17	7.12

IV. TRYING TO IMPLEMENT OUR OWN METHOD

We tried to implement our own compression algorithm based on the algorithms we have mentioned over this paper. After mainly looking at the paper mentioned in reference [3], we started by looking at possible transformations available that might be useful for these images. We quickly found the Move to Front Transform. Eventually we decided it would not be very useful for the specific types of images we mainly want to compress, that being grey-scale images and binary images. Since there aren't many possible different values of pixels on these images, this transform would not be very useful, especially in binary images.

Then we decided to implement Burrows-Wheeler Transform, which is used to rearrange the data to enable other methods to compress it more efficiently. The BWT alone does not compress the file, but when used with, for example, Run-Length Encoding, it improves the compression.

There are very simple implementations of the BWT over the internet, but after trying to apply them to real case scenarios it's clear those implementations aren't efficient and can't be used for what we intend. Eventually we found a good method using suffix arrays (reference [14]). The code used to create the suffix array using induced sorting is from reference [15].

Run-Length Encoding looked like a good simple option to combine with the BWT, so we also decided to implement it. We used reference [16] as an example to create our own methodology.

TABLE VI. OUR OWN IMPLEMENTATION

Name of the File	Original Size (MB)	Comp. Size (MB)	Comp. Ratio	Comp. Speed (MB/s)
egg	17.7	9.54	1.86	0.14
landscape	11	7.38	1.50	0.14
pattern	48	2.73	17.58	0.17
zebra	16.7	10.1	1.65	0.13
Average			5.65	0.15

As you can clearly see from the table, this method has an okay compression ratio but the compression speed is slow. We had some success at increasing the compression speed compared to what we had in the beginning, but it's still not great for most scenarios.

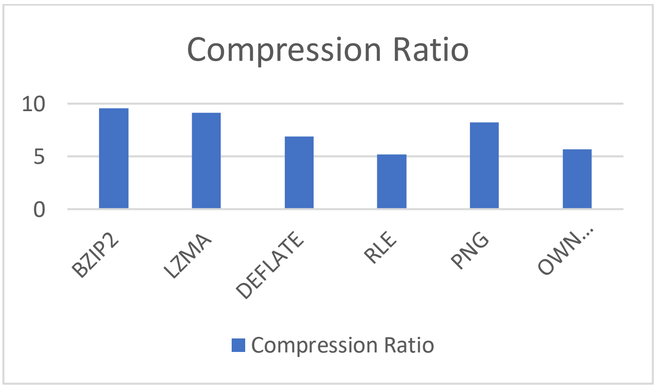

Figure 3: Compression Ratio of all the algorithms used on this experiment

As observed in figure 3, our method has a slightly worst compression ratio than the supplied PNG images.

V. CONCLUSION AND FUTURE WORK

Our method is far from perfect, but as we understood from doing this research, there is no perfect compression method. There are ones that fit different needs, and you need to pick the one that suits your needs.

With the increasing development of quantum computers, in the near future there will be new compression algorithms based on quantum methods. An increase in compression and decompression speeds are expected, making some currently unreliable methods suitable.

References

[1] Apoorv Gupta, Aman Bansal and Vidhi Khanduja, “Modern Lossless Compression Techniques: Review, Comparison and Analysis”, 2016.

[2] Ms. Shilpa Ijmulwar and Deepak Kapgate, “A Review on - Lossless Image Compression Techniques and Algorithms”, 2014.

[3] Lina J. Karam, Lossless Image Compression, Chapter 16,pp. 385-417.

[4] Morgan Kaufmann, Introduction to Data Compression, 4th ed., Khalid Sayood, 2012.

[5] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[6] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[7] B. Rusyn, O. Lutsyk, Y. Lysak, A. Lukenyuk and L. Pohreliuk, "Lossless image compression in the remote sensing applications", 2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP), Lviv, 2016.

[8] M. J. Weinberger, G. Seroussi and G. Sapiro, "LOCO-I: a low complexity, context-based, lossless image compression algorithm," Proceedings of Data Compression Conference - DCC '96, Snowbird, UT, USA, 1996, pp. 140-149, doi: 10.1109/DCC.1996.488319.

[9] https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_aritm%C3%A9tica

[10] https://en.wikipedia.org/wiki/Prediction_by_partial_matching

[11] https://en.m.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch

[12] https://en.wikipedia.org/wiki/Portable_Network_Graphics

[13] https://medium.com/@duhroach/how-png-works-f1174e3cc7b7

[14] https://github.com/dsnet/compress/blob/master/doc/bzip2-format.pdf

[15] https://zork.net/~st/jottings/sais.html

[16] https://stackabuse.com/run-length-encoding/