Lossless Compression Techniques For Grayscale Images

Amélie Butler
Emily Dormody
Anders Cornect
Ethan Denny

1. Introduction

Image compression is a technology we all use during our everyday computing tasks, often without even being aware of it. It presents an interesting challenge: how can data be "shrunk" efficiently, both in terms of time and space? We aim to quantitatively explore some of the methods and techniques available to do so, including the LZ77 algorithm, Huffman coding, run-length encoding and arithmetic coding. Each of these algorithms is explained in detail later in this paper, along with some fascinating data and conclusions.

2. Description

Before explaining our methods, it would be wise to briefly examine compression. Compression takes advantage of the fact that the *entropy* of some set of data is likely to be significantly smaller than the size of the data itself. By entropy, we refer to the term as it was used by Shannon [1]: the irreducible information contained within data. Entropy is as much what something *is* as what it *isn't* - the 26 letters of the English alphabet can be represented with just 5 bits, but the ASCII standard uses 8 bits, in order to allow for a greater amount of symbols to be encoded. *Lossless* compression algorithms allow pictures to be compressed and then later fully reconstructed without distortion or loss of information, by not reducing the inherent entropy of the data.

For this project, we only considered lossless image compression, and only used grayscale images, to keep the scope of the endeavour managable and easily testable. We compared several different algorithms: the LZ77 algorithm, Huffman coding, run-length encoding and arithmetic coding. The first three techniques were implemented and tested on a subset of the UCS-SIPI image collection in order to compare and contrast their time and space efficiency. We also wrote an implementation for arithmetic coding, but technical difficulties involving floating-point precision led us to conclude that it was not a competitive technique for image encoding.

There are many algorithms for lossless image compression, but it is a proven fact that for any lossless compression algorithm which successfully *decreases* the size of some file then there must exist another file for which that algorithm *increases* the size. This is a result of the pigeonhole principle [2], and is one of the primary reasons that there is no "best compression algorithm". This is also why the purpose of our project is not to find the best algorithm for lossless image compression. Instead, it is to explore a few of the best-known algorithms for the task and compare the strengths and weaknesses of each.

3. Solutions

3.1. Run-Length Encoding (RLE)

Run-length encoding records *runs* of bytes (sequences in which only a single value is repeated). Rather than each byte being stored individually, a single byte and a count are recorded instead. For

files with many runs, this can result in large space savings. However, the opposite may also be true, with many files suffering from increased space usage after performing RLE. [3]

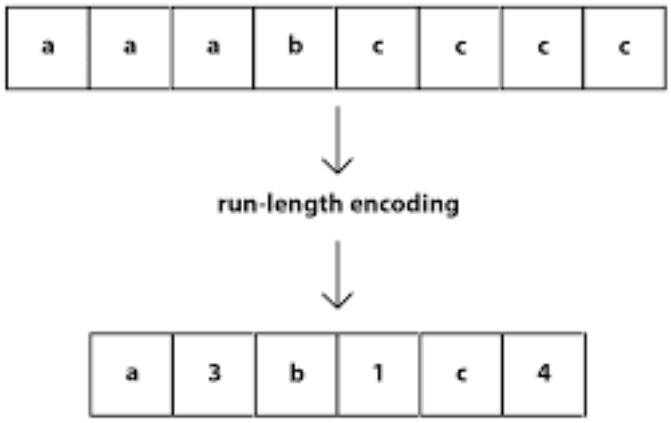

Figure 1: An example of RLE performed on a string.

Our code performs RLE exactly as shown in Figure 1. It iterates over each row of pixels in a given image and attempts to compress them, but only stores the "compressed" row when it leads to space savings.

3.2. Huffman Coding

Huffman coding is an algorithm that generates variable-length codes assigned to intensities. This is done in such a way that no codes share a prefix, which prevents ambiguity. Encoding takes place by constructing a binary tree based on the frequencies of the intensities. The tree is constructed as follows:

- Every intensity is represented as an independent node, with no parent-child relationships.
- We then consider all top-level nodes (nodes without parents), and pick the two which have the lowest frequencies. A node's frequency is either the frequency of the intensity it represents (if it is a leaf node), or the sum of the frequencies of its children.
- The selected nodes are made the children of a new node. For example, if we select two leaf nodes representing intensities 100 and 212, with frequencies 12 and 34 (respectively) then they become the children of a new node (entitled, for instance, "100/212") with frequency 46.
- We repeat this process of combining top-level nodes until there are no remaining independent nodes.

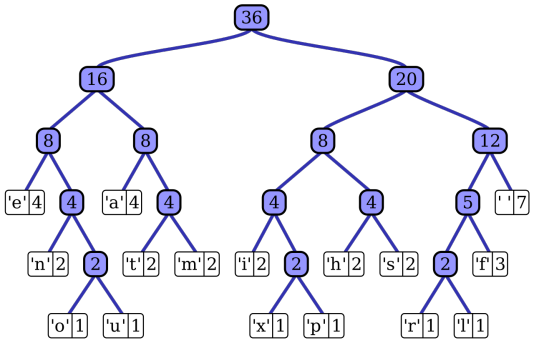

Figure 2: An example of Huffman coding performed on a string. The same principles apply for an image, but with intensities rather than characters. [4]

As seen in Figure 2, an intensity's "code" is derived from the set of edges connecting it to the root node. Starting from the root node, a 0 indicates that the leaf node is within the left subtree, and a 1 indicates that the leaf node is within the right subtree. Thus, a code is really just a set of instructions for traversing the tree. Since higher frequency intensities are added to the tree later, they will have shorter codes, and this is what allows Huffman coding to compress data. [4]

So, with reference to Figure 2:

- "tame" ⇒ `0110 010 0111 000`
- "sith" ⇒ `1011 1000 0110 1010`.

3.3. LZ77

LZ77 is the most complicated technique we studied. It provides an algorithm for searching an image for matching sequences of bytes, and then compressing it by creating pointers to duplicate sequences. Implementing it proved difficult, and our result was not optimal, but we succeeded in capturing the core functionality from the original paper **[Source needed]** in a more approachable manner.

Given a small search buffer of unprocessed bytes (typically ~2^8 bytes in length), it searches a much larger "dictionary" of already processed bytes (typically ~2^{15} bytes in length), and attempts to find the longest match within the dictionary. By a match, we mean a substring starting at the beginning of the buffer which is also present somewhere in the dictionary.

Typically, LZ77 outputs a tuple containing three fields:

- The distance from the start of the buffer to the start of the match in the dictionary
- The length of the match
- The next byte after the match

However, in our implementation, we have two types of tuples:

1. If a match is found:
   - The distance from the start of the buffer to the start of the match in the dictionary
   - The length of the match
2. If a match is not found:
   - A zero byte
   - The first byte in the buffer

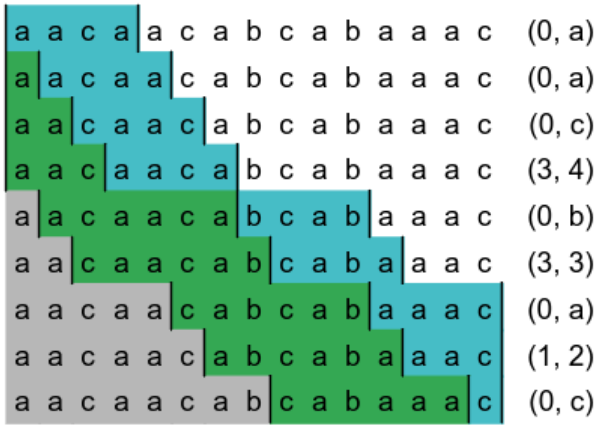

Figure 3: An example of LZ77 encoding performed on a string. Blue represents the buffer (length = 4), green the dictionary (length = 6), and grey already compressed data. [5]

In either implementation, once a tuple is returned, the process is repeated for the next unprocessed byte in the input. [6] Decompression reverses the method described above, iterating over the tuples and copying from previously decompressed bytes or inserting new bytes. Our implementation was sourced from [7] and [8].

3.4. Arithmetic Coding

Arithmetic coding uses fractional intervals to represent intensities. [9] It can be thought of as partitioning a number line between 0 and 1 into intervals based on the normalized frequency of each intensity, then progressively partitioning those intervals for every byte in the image. The compressed version of an image is some value which lies within the final interval.

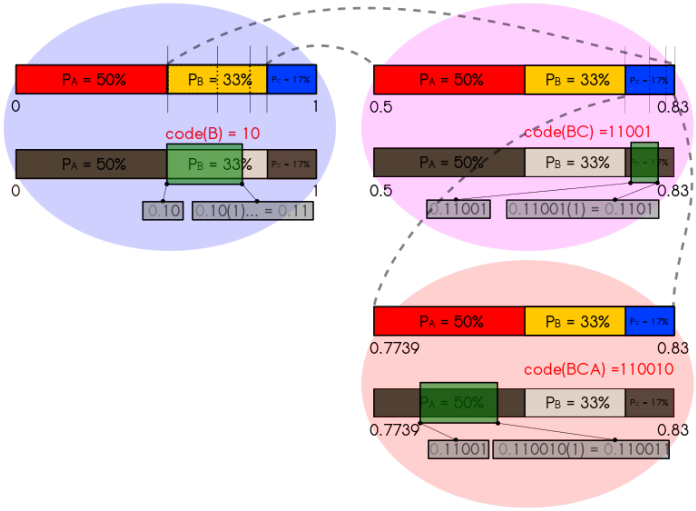

Figure 4: An example of arithmetic coding [9]

We originally tried to implement arithmetic coding using floating-point numbers to represent the intervals, but these did not give us the precision or speed we needed. Next, we tried using fractions. This gave us the precision we needed, but was extremely slow, taking several hours to compress an average-size image and space savings were extremely poor. For this reason, it is not included in our final results. Our implementation is sourced from [10].

4. Experiments and Results

Initially, we had plotted all of our results onto a box plot, with ‘x’-markings to denote the outliers. However, the general results are much easier to parse with the outliers removed. Therefore, graphs representing both sets of results (with and without outliers) will be presented. Overall, there is a significant difference in the average performance and effectiveness in the three algorithms.

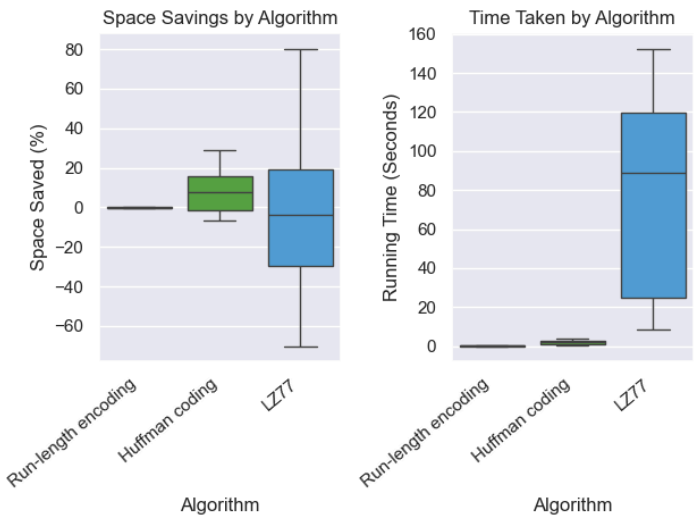

Figure 5: Results from running RLE, Huffman, and LZ77 algorithms on the UCS-SIPI image collection, with the outliers removed.

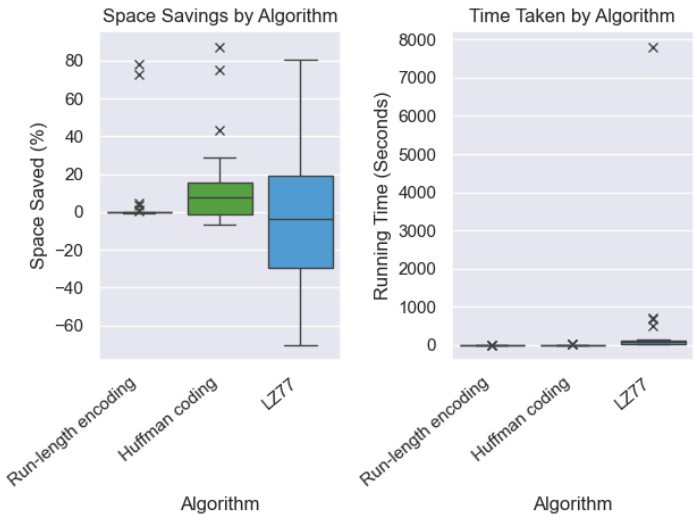

Figure 6: Results from running RLE, Huffman, and LZ77 algorithms on the UCS-SIPI image collection, including the outliers.

4.1. Run-Length Encoding (RLE)

Run-length encoding was not at all effective for the majority of the images we observed. There were, however, certain outliers. Some images had space savings in the 5-10% range and two, *resolution chart* and *pixel ruler*, had significant space savings of 73% and 78%, respectively, as seen in Figure 7.

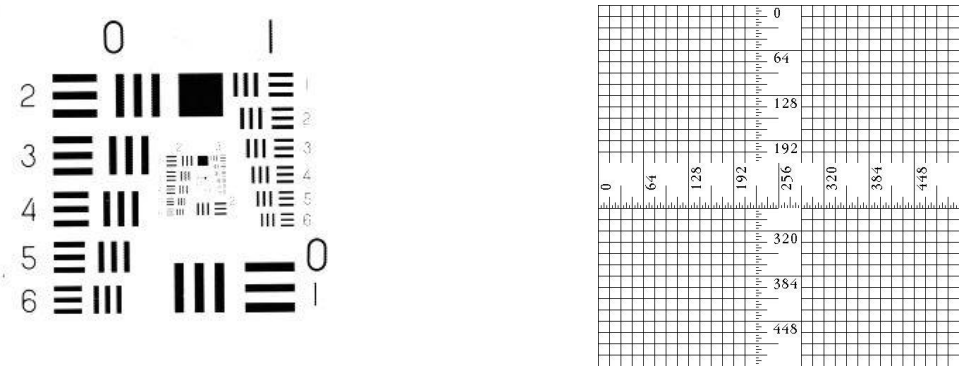

Figure 7: The two images on which run-length encoding led to large space savings, namely "resolution chart" (5.1.13.tiff) and "pixel ruler" (ruler.512.tiff), respectively.

These two images are perfect candidates for RLE, containing large sections of solid colours. Despite these two successes, RLE was extremely ineffective in compressing the types of images the majority of people would see in everyday use. Despite its lack of space savings RLE ran very quickly.

4.2. Huffman Coding

Being a less naive approach to data compression, Huffman coding had slightly more interesting and varied results than RLE. The resulting space savings were consistently greater, with the average being closer to 10%. A tradeoff, however, is that there were a number of images where the amount of space was *increased* rather than decreased. As we mentioned at the start of the paper, this is inevitable in some cases, but this happened to a more significant degree in our Huffman coding results than our RLE results.

As with RLE, there were certain outliers where the amount of space saved was very significant. Again, the two images *resolution chart* and *pixel ruler* had extremely large space savings (75% and 88%, respectively). However, there was another significant outlier: *airplane*, as shown in Figure 8.

Figure 8: "Airplane" (7.1.02.tiff), an image with significant space saving results from Huffman Coding that did not appear in Run-Length Encoding.

This image has large amounts of similar colours, but they are not grouped together as much as in *resolution chart* and *pixel ruler*. This means that certain intensities appear have very high

frequencies, but do not necessarily cause large runs of bytes, meaning that in this case, Huffman coding is ideal for compressing this specific image, while the more naive approach of RLE is not.

As for time, Huffman coding was very efficient, without any major issues or outliers. The time taken was much higher than that of RLE, but only in the sense that it was more readily measured in seconds than milliseconds. However, it was still not a significant amount of time for the size and amount of images which we were aiming to compress.

Overall, it seems that Huffman coding was a reasonably useful algorithm when it comes to saving a noticeable, if somewhat small, amount of space in a "safe" way — i.e. with a low chance of high compresson times or greatly increased space usage.

4.3. LZ77

The results from LZ77 were by far the most varied, and the most interesting. It is may be quite easy to look at an image and predict whether there will be significant space savings when using RLE or Huffman, but the same cannot be said for LZ77.

LZ77 had no true outliers when it came to space-saving data. However, results were greatly varied. We experienced space reductions of up to 80%, but there were also space increases as high as 70%. In fact, our data seems to show that on average LZ77 *increased* the size of the given images rather than decreased them. However, this algorithm produced far better results overall for the images whose size it did decrease, when compared to RLE and Huffman. High-percentage space-saving was a common trend with LZ77, instead of an outlying rarity.

It is interesting to note that LZ77 did not compress the *resolution chart* and *pixel ruler* images as significantly as either of the previous algorithms. This demonstrates effectively the non-transitive nature of comparing image compression algorithms, i.e. different compression algorithms are effective for different kinds of images.

The major issue with LZ77 (at least in our implementation), was time. Looking at Figure 5, the average time taken was close to 90 seconds. This is a significant increase from the seconds or milliseconds of our other two algorithms. As can be seen in Figure 6, there were also outliers that landed in the 10-15 minute range, but the most significant were *pixel ruler* and *step wedge* (Figure 9).

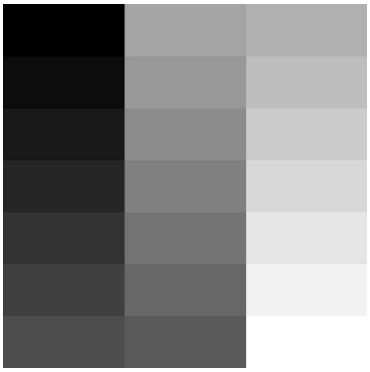

Figure 9: "Step Wedge" (gray21.512.tiff), the image that we had to remove from our dataset because of LZ77.

LZ77 took a staggering 2 hours and 10 minutes to compress *pixel ruler*. It is not exactly clear why, but we suspect there is an issue with our implementation (or an improvement that can be made) that causes images with many adjacent pixels of the same value to cause the algorithm to tend towards worst-case time complexity. However, this does not fully explain why images like *resolution chart* actually compressed rather quickly. Therefore, this theory would require more data and a much deeper look at our implementation of LZ77.

Step wedge is simply a cascading gradient of 21 rectangles in various shades of grey. For reasons we suspect are similar to those outlined above, *step wedge* took many hours to compress, to the point where we could not realistically collect our data so long as *step wedge* was a part of the data set. This provides some additional evidence to our theory that long stretches of similar colours cause LZ77 issues, but more research and analysis would have to be done to understand further.

5. Conclusion

Throughout this paper, we have explored how inconclusive exploring image compression algorithms can be. While it was not possible to label any one technique as the best, we observed that there are situations in which it may be optimal to choose one algorithm over another when attempting to increase compression, decrease run time, or both.

When it comes to images with many sequences of identical bytes, we concluded that run-length encoding would perform the best. This algorithm was often ineffective, but for some images, its fast compression makes it the best choice. The speed did not vary greatly between images and it was always the quickest algorithm. Due to our implementation, RLE did not increase the space of the image if there were no runs to compress, so while there would often be no savings, it would take up no more space then that of the original.

For images where there are some intensities which are very common relative to others (regardless of location in the image), Huffman coding stood out. This algorithm was consistently able to compress most images, even if that compression was often $< 10\%$. Huffman coding had decent (but not amazing) run time, making it the safe choice since it did not trade its space savings success for extreme run times.

LZ77 performed well in the right conditions and had the best space savings, even for more complex images. However, it could also result in increased space usage to the same degree as its savings. The other drawback of this algorithm was its run time, taking up to a minute on average to compress an image, with some images taking several minutes or even hours. In general, the long run times might make selecting other algorithms a better option in some cases.

We were unable to find or implement an algorithm which clearly outshone the others. Since LZ77 is the algorithm behind common file formats like PNG and ZIP, we suspect that it may have been able to eventually perform well on a larger set of images. Unfortunately, lack of time prevented us from developing it further, and our choice to work in Python torpedoed performance. Overall, however, we can say conclusively that each algorithm has its own strengths and weaknesses, and demonstrates the importance of fitting your algorithm to your data set to achieve optimal results.

Bibliography

[1] C. Shannon, “A Mathematical Theory of Communication”.

[2] Wikipedia, “Lossless compression --- Wikipedia, The Free Encyclopedia”, 2023.

[3] Wikipedia, “Run-length encoding --- Wikipedia, The Free Encyclopedia”, 2023.

[4] Wikipedia, “Huffman coding --- Wikipedia, The Free Encyclopedia”, 2023.

[5] P. Shi, B. Li, P. H. Thike, and L. Ding, “A knowledge-embedded lossless image compressing method for high-throughput corrosion experiment”, *International Journal of Distributed Sensor Networks*, vol. 14, 2018, doi: 10.1177/1550147717750374.

[6] E. Erdal and A. Ergüzen, “An Efficient Encoding Algorithm Using Local Path on Huffman Encoding Algorithm for Compression”, *Applied Sciences*, vol. 9, no. 4, p. 782, Feb. 2019, doi: 10.3390/app9040782.

[7] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression”, *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[8] G. E. Blelloch, “LZ77 15-853:algorithms in the real world - CMU school of computer science”. [Online]. Available: https://www.cs.cmu.edu/~guyb/realworld/slidesF08/suffixcompress.pdf

[9] Wikipedia, “Arithmetic coding --- Wikipedia, The Free Encyclopedia”. 2023.

[10] M. Yang and N. Bourbakis, “An overview of lossless digital image compression techniques”, in *48th Midwest Symposium on Circuits and Systems, 2005.*, 2005, pp. 1099–1102. doi: 10.1109/MWSCAS.2005.1594297.

Contributions

Amélie
- Wrote most of the report except the results section.
- No involvement in the presentation.
- Wrote arithmetic coding source code.

Ethan
- Proofread and edited the report.
- Wrote the majority of the source code, including 3/4 of the algorithms and implementing fractions for arithmetic coding.
- Wrote and presented the algorithms portion of the slides.

Emily
- Proofread and edited the report, wrote conclusion.
- Wrote source code and documentation in terms of running the algorithms.
- Structured slideshow for the presentation.

Anders
- Wrote the source code for data collection, including logging the data for runtime and space-saving, as well as plotting the data using the Python package, *Seaborn*.
- Wrote the “Experiments and Results” section of the paper.
- Wrote and presented the results portion of the slides.