# Black & White Image Compression and Decompression

Ruben Teimas, m47753

January 2022

## 1 Introduction

During the curricular unit of *Data Compression and Coding* we studied both lossy and lossless compression techniques and algorithms and, for the final project, we've been given the chance to choose a problem and present its solution.

In my final project I chose to implement a black  white image compressor and decompressor. By using only black  white images (*.pbm*) our alphabet is considerably smaller than it would be with *RGB* images, which might also lead to a smaller number of symbols.

## 2 Approach

Even though we're compressing an image and not text we must have a defined group of symbols. If we treat each pixel individually we won't have any gain since each pixel can be represented with 1 bit.

For that reason, I decided to process the image in squares of dimension 2, as described in Figure 1. Each square/tile will represent a symbol.

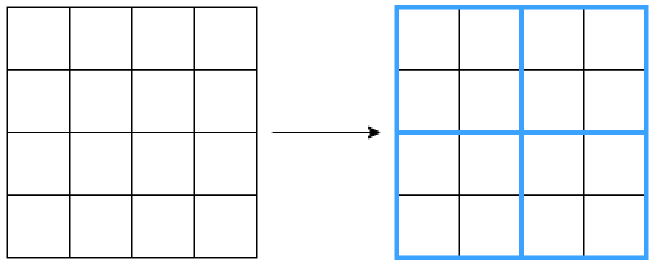

Figure 1: Tiled image.

Since a tile has 4 bits we'll have $2^4$ (16) symbols.

## 3 Huffman Code

Amongst the algorithms we were taught, *Huffman code* seems a right fit for this job. When compressing an image we can calculate the *p(x)* of each symbol, which satisfies one of *Huffman code* requirements. The algorithm also allow us to generate the best instantaneous code that can be generate from a particular alphabet. It satisfies the following expression:

$$H(X) \leq L(C) < H(X) + 1$$

Which means that the difference between the entropy and average length of the code is always smaller than 1.

Unfortunately, *Huffman code* has its downsides as well. One of the downsides is that, to decompress we need to look at the same tree we used to compress it, thus storing the tree.

### 3.1 Huffman tree

The *Huffman code* is implemented by using a binary tree. To build the tree we must satisfy some conditions:

- All the symbols are leafs.
- In each step, the 2 nodes with smaller frequency are used to create a parent node with the sum of the children frequency.
- The code for each symbol is its path from root to the leaf.

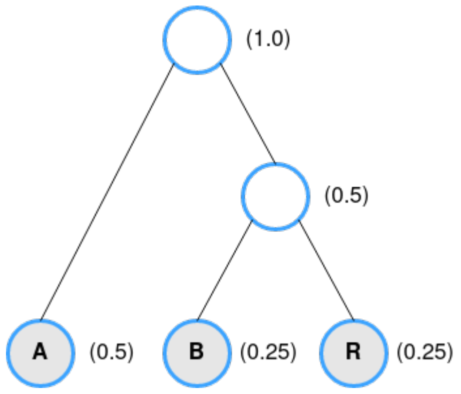

Figure 2: Sample Huffman tree.

By looking at the tree from Figure 2 we get the following codes for the symbols: {A: 0, B: 10, R: 11}.

### 3.2 Implementation's tweaks

My implementation of *Huffman code* also adds a Python dictionary to the tree. The dictionary stores the symbols and its corresponding code. That way, encoding a symbol has an complexity of *O(1)*.

I thought about another tweak to implement, which unfortunately did not see the light of day. When decoding the symbols, instead of travelling the tree we could have a look-up table. I've read [1] about a way of doing it however, after thinking about it, I realized that I would be scarifying a lot of memory for the sake of some time.

Another minor tweak that could have been done in my code was to use the number of occurrences of each symbol instead of its probability.

## 4 Program's flowchart

The program can be used easily by the user without knowing any details from the implementation. For the user to run the program he can, at the root of the project, run ***python3 main.py -f filepath --c/--e***.

The compression operation takes ***.pbm*** files as input whilst the extraction (or decompression) operation takes ***.huffman*** files.

Below, at Figure 3, the program's flowchart is represented .

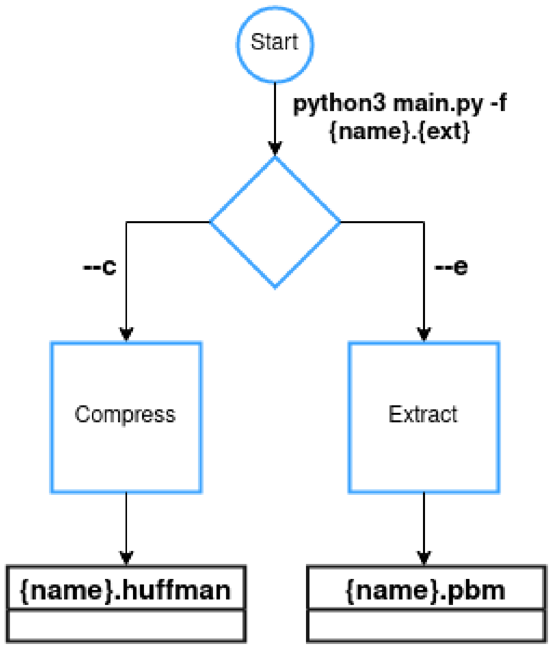

Figure 3: Overall program's flowchart

## 4.1 Compression flowchart

Before the image's symbols can actually be encoded, thus making the compressed image, some pre-processing must be done just as it can be seen in Figure 4

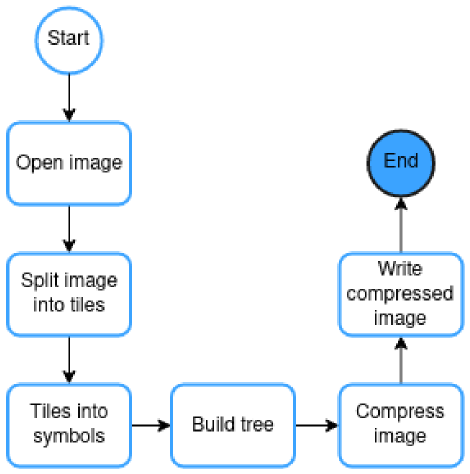

Figure 4: Compression flowchart.

After encoding the info from the original info we must store it in a binary file in order to compress it. The file has a specified format as described on Figure 5.

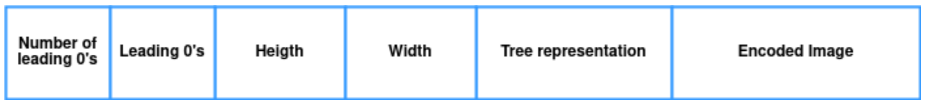

Figure 5: Compressed file structure.

Since we're writing a binary file the smallest unit of information that can be written to the file is a *byte*, which is composed by 8 *bits*, which meanings that our number of bits must be a multiple of 8.

To guarantee it I decided to insert a padding to the beginning of the file. We want a multiple of 8, which means that our padding will range from 0 to 7 bits. This amount of information can be represented by 3 bits, which is the length of the first section of the file.

The second section will have $n$ *bits*, in which $n$ is the number represented on section 1.

The third and forth section are used to store the height and width of the image. Due to a personal choice, they are not dynamic and have a size of 11 *bits* each, which means that images with either an height or width superior to 2047 pixels will not be properly compressed.

The file's fifth section is used to represent the *huffman's tree*. To represent the tree we do a *pre-order* traversal (Root, Left, Rigth), an interior node is represent by *0* and the leafs (symbols) are represented by *1*. When a leaf is found the symbol is written after the 1 representing it. This

The size, $S$, of the tree, $T$, is given by:

$$S(T) = (Ns_T \times 2) - 1 + Ns_T \times \text{L}(s_T)$$

In which:

- $Ns_T$ = Number of symbols of the tree.
- $L(s_T)$ = Length of a tree's symbol.

At the end of the file, in the sixth section, we have at last the image representation.

### 4.2 Extraction flowchart

In short, extracting the original image from the compressed file is reversing what we did when we compressed it. The flowchart is represent at Figure 6.

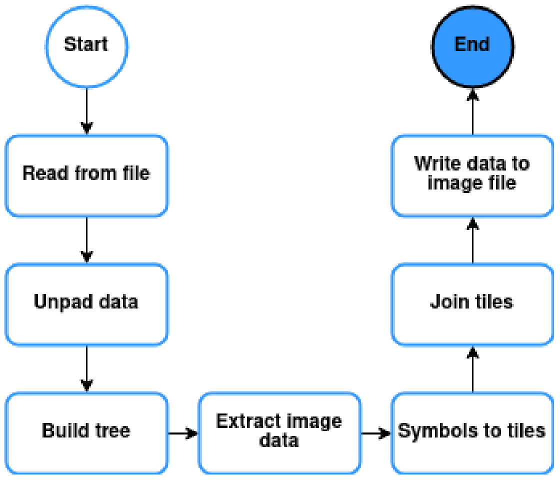

Figure 6: Extraction flowchart.

Most of these steps are very straight forward with the trickiest being the one to read the tree from the file. However, it easy to explain, just like we built the tree we will read it and reconstruct it, in a *pre-order*. This is possible to do because the *huffman tree* is a *full binary tree* (all the nodes, except the leafs, have 2 children), which allow us do it recursively without needing to know where our tree representation start or end.

Figure 7 represents this process.

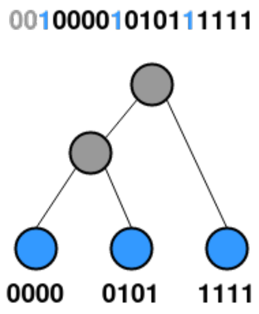

Figure 7: Reconstructing tree from file.

## 5 Results

To test my program I compressed some *.pbm* images. The results can be seen in the table below.

| **Image** | **Height** (pixels) | **Width** (pixels) | **H(X)** (bits) | **L(C)** (bits) | **Time** (s) | **Compression Rate** |
|---|---|---|---|---|---|---|
| baiao.pbm | 808 | 1200 | 2.907 | 2.976 | 7.1 | 1.344 |
| brunner.pbm | 658 | 1170 | 2.204 | 2.273 | 6.7 | 1.760 |
| lenna.pbm | 512 | 512 | 3.106 | 3.144 | 2.25 | 1.272 |
| linda.pbm | 1056 | 1300 | 1.243 | 1.521 | 10.72 | 2.629 |
| lof.pbm | 1920 | 1080 | 2.565 | 2.577 | 15,99 | 1.552 |
| sauron.pbm | 1920 | 1080 | 0.438 | 1.121 | 15,92 | 3.569 |

Table 1: Image compression benchmarks.

For most of the compressed images the average length of the code is very close to the entropy, which means the codes are very close to being optimal. The compression rate is also good with most of the rates varying between 1.5 and 3.0.

## 6 Conclusion

I think the work was successfully accomplished as I managed to compress an image losslessly using one of the algorithms studied in class.

Some optimizations could have been done, specially in the image processing part which is taking a lot of time in the compression operation.

## Final Notes

The project was made using *Poetry* as a dependency manager and it must be installed in order to run the program. After installing *Poetry* go to the root of the project and run ***poetry install –no-root***. Next type ***poetry shell***.

Inside the virtual environment you can run the program with the command mentioned in the section 4.

***P.S:*** “Always look at the bright side of light!”

## References

[1] JasonD. Huffman code with lookup table, Dec 2012.