Huffman Coding for PGM Files

Abdel-Salam Waleed[1], Mahmoud Mohammed Abdel-Moneim[2], Omar Sayed[3], Zeyad Khaled[4]

Department of Biomedical Engineering, Cairo University

SBE 201: Data Structures and Algorithms

[1] abdel.sayed98@eng-st.cu.edu.eg

[2] mahmoud.qalouby14@eng-st.cu.edu.eg

[3] omar.ibrahim98@eng-st.cu.edu.eg

[4] zeyad.abouyoussef99@eng-st.cu.edu.eg

Abstract

Data compression is an important part aspect of modern-day computing. Files taking large disk space can be hard to send using emails and can take a lot of time to upload to the cloud. Huffman encoding is a popular compression algorithm developed by David A. Huffman in 1951. The main advantage of this algorithm is the absence of data loss after compression and the ability to retrieve the data in the original format. In this use case, we implemented the Huffman algorithm on images in portable gray map (PGM) format and created an encoded file storing the compressed data and a frequency file storing the frequency table of the original file. Our main goal was to calculate the effect of compression on 8 different PGM image files and calculate the compression ratio of each one of them. We created a simple console application in C++ and all the source code can be found on Github.

Keywords: Huffman encoding, data compression

Introduction

Data compression techniques can either be lossy or lossless. Lossy techniques present in jpeg image files and audio files use inexact approximations to discard data from the original file. This leads to a significant reduction in the file's size but also, a degradation in the quality of the compressed file. The main disadvantage of lossy techniques is the inability to reverse the compression, as the discarded data is was lost forever during compression. Meanwhile, lossless techniques reduce bits by eliminating statistical redundancy. The original file can be retrieved without any data loss. Lossless techniques are mainly used in Text files and high-quality images that cannot tolerate any data loss.

David A. Huffman was born on the 9th of August 1925. He got his bachelor's degree from Ohio State University in 1944 in Electrical Engineering, a master's degree from Oh and a doctorate in 1953 from MIT. During his time at MIT, he developed a data compression technique that calculates the frequency of each character in a given file and constructs a binary tree where every character in the file is represented by a Binary Character Code of 0s and 1s. As the frequency of a character increases, its Binary Character Code shortens hence, saving space in the compressed file. This algorithm can save from 20% up to 90% of space depending on the original data characteristics.

Algorithm

The Huffman algorithm creates a binary tree of nodes where leaf nodes are the unique characters in the file and internal nodes are the sum of frequencies of the smallest two nodes. Considering a file with **N** unique characters, the Huffman tree is created as follows:

1. **N** leaf nodes whose values are the frequency of each character in the given file are created.
2. These leaf nodes are pushed into a highest priority queue and the two smallest nodes are merged to form an internal node. The internal node replaces its child nodes in the queue.
3. We repeat step 2 until there is only one node in the queue.
4. This node is the root of the tree.

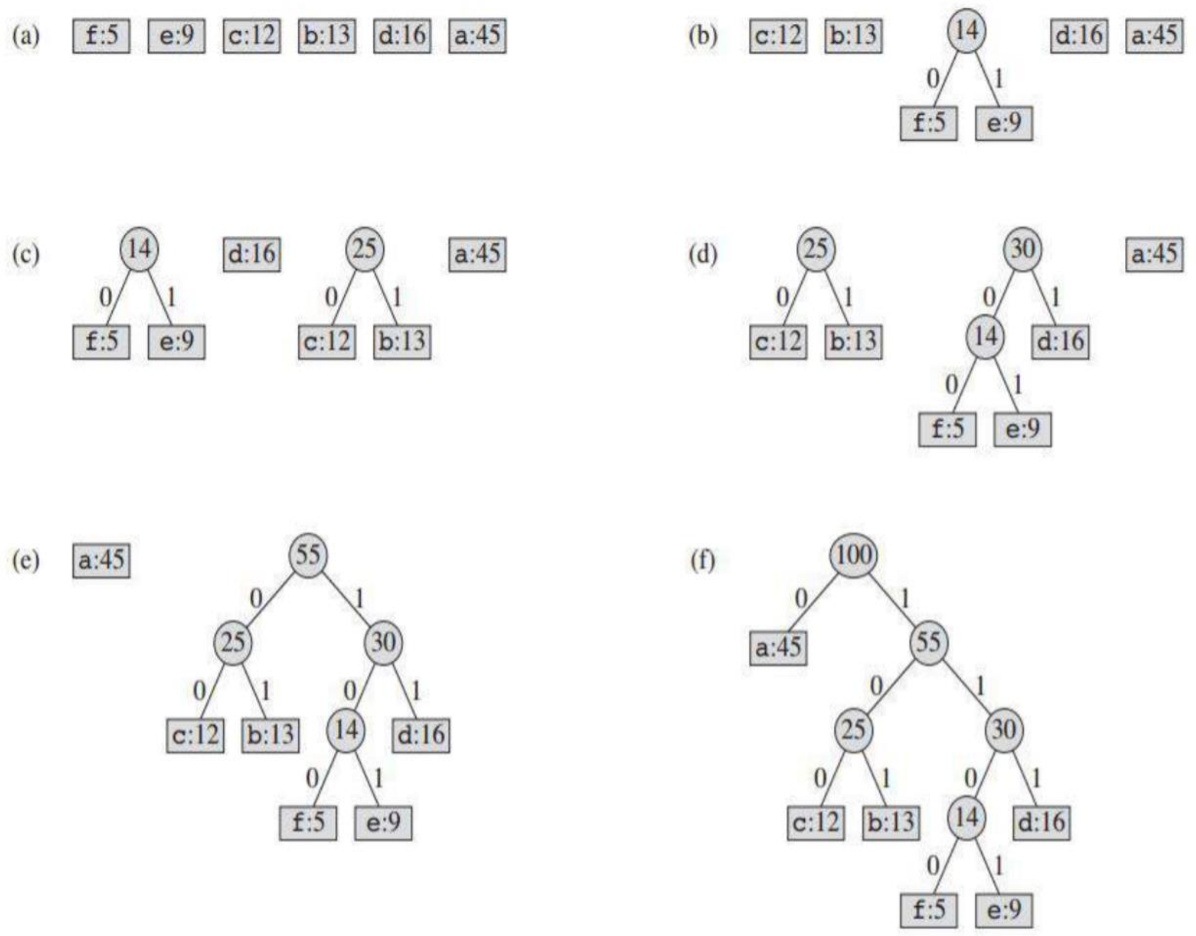
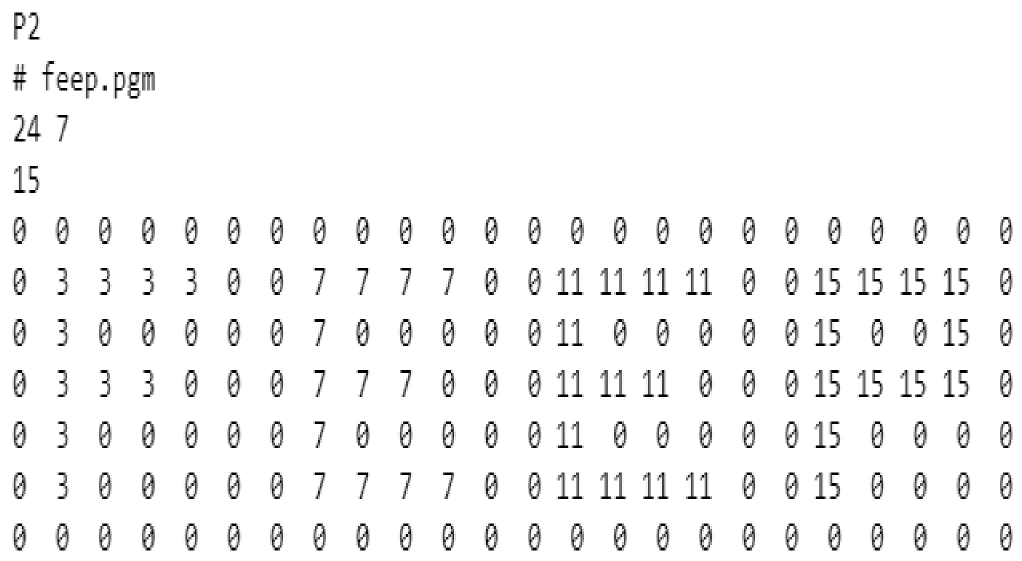


Figure 1. Visualization of Huffman tree

After forming the tree, each character is assigned a Binary Character Code according to its position in the tree. By convention, left nodes are labeled as 0s and right nodes are 1s. So, in the previous example, character **e:9** is represented by **1101** meanwhile, **a:45** is represented by **0** only.

Implementation

In our use case, we implemented the algorithm on images in PGM format. Images in PGM format are represented either by ASCII text format or binary format and hold the values of each pixel in the image. They consist of 3 header lines holding the format P2 or P5 (ASCII or Binary), width and height, and the maximum value of pixels. Then, a grid of pixels is formed with each value representing the value of the pixels on the grayscale with 0 mapping to white and the maximum value mapping to black.



```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figure 2. ASCII representation of PGM file

In the figure above, the first line is P2 (ASCII), then any line starting with the "#" symbol is ignored. Width is 24 and height is 7, the maximum value of a pixel is 15 then the pixel values are formed. 7 rows are formed, with each row holding 24 pixel values.

Firstly, we began by reading the PGM image files. It is worth noting that all 8 PGM files had high resolution and were stored in P5 format (Binary) so the files were relatively small. We stored the pixel values in a vector and built a frequency table for each value. The frequency table was built using a map data structure. Then, we fed the frequency table to the Huffman algorithm to start building the binary tree. As mentioned before, the Huffman tree is built with the help of a highest priority queue data structure, so, we created nodes of all pixel values and pushed them into the queue. Then, we decided to construct another queue to hold the merged nodes as we believed that it could prevent the delay from constantly sorting the elements in the original queue but no tests were carried to further prove this hypothesis.

After constructing the tree and created the Binary Character Code, we started encoding the input stream. The maximum value was 255 which meant that we only needed one byte for each value. We wrote the Binary Character Code of each value in the stream into the encoded file (.enc) and the frequency table in a separate file (.frq) in binary.

In the decompression phase, we first read the frequency table file (.frq) and constructed the frequency table and Huffman tree. Then we read the entire encoded file and copied every byte into a vector. Then, we converted every byte into a string of 0s and 1s to access every bit and started traversing the tree until reaching a leaf node. After reaching a leaf node, we pushed the leaf's character into a vector creating the original file's stream. Lastly, we create the PGM file.

Results

The results were promising. The compressed and decompressed successfully with no loss in data. The compression on average took 2-3 seconds while the decompression took only about 1 seconds.

Contributions

The workload was evenly distributed among team members. Abdel-Salam Waleed was responsible for reading and writing the PGM files, the map implementation and building the frequency table. Mahmoud Abdel-Moneim built the Huffman tree. Zeyad Khaled was responsible for the encoding and decoding the input streams and the program's main. We only used one external library cxxopts and all data structures were built in-house.

References

https://brilliant.org/wiki/huffman-encoding/

https://courses.engr.illinois.edu/cs225/sp2020/labs/huffman/

Algorithms Notes for Professionals book (pp. 84-87)