Huffman Code File Compression
159.333 Programming Project

Gray Salmond
Student ID: 15366621
Supervisor: Dr. Andre Barczack

Summer School
2017/2018

Contents

Chapter 1

Literature Review

1.1 What is data compression

In the context of data files, data compression is the process of encoding information using a smaller representation than the original file [1]. For data files the size of the representation can be defined in bits or other derived unit such as bytes.

Benifits of data compression occur from reducing the resources required to store and transmit data [1]. However, in the process of compressing and decompressing data computational resources which may impact benifits gained by compression.

The ratio of compression can be defined as:

$$C_r = \frac{d_1}{d_2}$$

The compressed ratio (C_r) is equal to the compressed file size (d_1) over the orginal file size (d_2) [2].

1.2 Lossless Compression

Lossless compression is a method of encoding a file with the goal of representing the file using a smaller amount of bits but not losing any information from the decoding process. The opposite being lossly compression where this is not the case. Huffman coding, of which is the subject of this report is used to implement lossless compression [3].

Lossless compression works because most real world data contains stastical redundancy [1]. If a file contains symbols that occur more frequently than others we can use smaller representations to represent the frequent symbols and larger representations to represent the less frequent symbols. If a symbol doesn't occur at all in a file then no representation for that symbol is needed.

The two most common ways of constructing statistical models for the purposes of data compression are static and adaptive models . Static models build a model after reading all the data and store the model representation in the encoded file. Adapative models update a model as the file is compressed [4]. The implemtation of Huffman coding presented in this report is an example of a static model.

1.3 Lossy Compression

Lossy compression works by removing less important information from a data file. Audio and Image data are examples of where lossy compression is applied. Human eyes are more sensitive to varitions in luminance than the are to variations in colour [1]. Because of this fact image compression can remove information from files that humans can not percieve therby improving compression ration for a file by reducing the number of bits needed to represent said file. Similiarly human limitations for hearing various audio signals present opportunties to decrease the need representation of audio files.

1.4 Use cases

Various uses for data compression can be defined. General purpose data compression algorithms generally refer to compression algorithms that apply to standard text or binary files. Of these algorithms some may be optimised for a particular type of input file such as text compression . As mentiioned in previous sections specialized audio and image compression algorithms exist. Indeed for any particular domain there may be a case for developing a special purpose compression algorithm, such is the case of HAPZIPPER a compression application for the purpose of compressing genetic data [4].

Many of the lossless compression algorithms in use today combine Huffman Coding and other compression algorithms. DEFLATE combines Lempel-Ziv with Huffman coding and is used in zip files, gzip files and PNG images.

1.5 Entropy

Information theory underpins the theoritical background of data compression. Claude Shannon published pioneering papers on the topic in the 1940s and 1950s [1]. Shannon came up with the idea information entropy which can be defined by the following equation in regards to binary data [2]:

$$H = -\left(\sum_{i=0}^{n} p_i log_2(p_i)\right)$$

The negative of the sum of: The probability of the symbol (p_i) occuring in a file times log base 2 of the probability of the symbol occuring in a file.

The implications of information entropy impose a limit on the potential compression ratio. If the likelihood of any symbol is the same entropy will equal the number of bits it takes to represent each symbol in the original file. A lower entropy allows more frequent symbols to be represented in a smaller number of bits.

There is no single lossless data compression algorithm that can compress any and all data. In fact it is provably impossible to create such an algorithm [4].

1.6 Huffman Coding

Huffman coding is an algorithm developed by David Huffman in 1951. He came up with his method as a student studying information theory at the Massachusetts Institute of Technology [3].

A Huffman code refers to a prefix code, an encoded representation of a symbol whereas Huffman coding is the process of creating and decoding Huffman codes. Huffman coding is a lossless data compression algorithm. There are two major tasks for the algorithm to perform. Encoding a data file and decoding an encoded file with the goal being achieving an optimal compression ratio.

Encoding a file involves building a Huffman tree, a type top down binary tree used to create prefix codes for symbols. Symbols that occur more frequently are closer to the root of the tree and ones that occur less frequently are found at lower levels of the tree.

Decoding a file involves using a built Huffman tree to decode a sequence of encoded symbols into their original form. A Huffman tree for any given file is not necessarily unique so a Huffman tree must also be extracted in the decoding process.

Chapter 2

Implementation

2.1 Background

The implementation of Huffman coding describe in this report was implemented in the C++ programming language. There is a wide range of reference material available on Huffman coding and variations on Huffman coding. The application is a simple command line application that runs on OSX or Linux type systems although it could easily be ported to Windows if required. The application described in this report compresses a single file but once again could be extended to allow compressing multiple files into a single file.

Huffman coding comprises of two distinct parts. Encoding, taking a uncompressed file and compressing it by building a Huffman tree. And decoding, using a Huffman tree to decompress a file previously compressed using the application.

The project is available as open source via GitHub [5] and made use of CMake to build the project and the CLion Integrated Development Environment during development.

The implementation has the option of encoding a file and decoding a file at the same time for demonstartion purposes. Encoded file filenames are suffixed with .huffCode and decoded files replace the .huffCode suffix with .decoded, the original file is retained for comparison purposes.

2.2 Encoding

Encoding a file using Huffman code requires two passes of the original file. The first pass count occurrences of a unique symbol within a file and uses this information to build a Huffman tree. A symbol could be any size of bits, for this implementation the initial symbol size is a single byte (8 bits) of which there are 256 possible combinations. The second pass uses the built Huffman tree to build the encoded file by comparing the symbols in the original file to the newly created Huffman codes as defined by the Huffman tree.

2.2.1 Counting Symbol Frequencies

As mention previously the first pass involves counting the frequencies of a unique symbol in a file. The implementation represents symbols as a struct, which contain data on which symbol and how many occurrences of the symbol there are. The application to be compressed is examined linearly and the count values are updated as symbols are examined. Much of the literature on Huffman coding uses frequencies instead of counts, however, this implementation uses integer counts instead of percentage frequencies.

2.2.2 Building a Huffman tree

To build a Huffman tree a min heap is required, a data structure which always has the smallest (in this case the least occurring symbol) element at its root. All the symbols are inserted into the Min Heap. The project implementation uses it's own version of a Min Heap instead of the standard library for the purpose of gaining experience with data structure implementations.

The next step in building a Huffman tree is actually building the tree. To build a Huffman tree we need to create new internal nodes that point to other nodes, either other symbols or other nodes. Upon completion of the Huffman tree there will be a single root internal node for a binary tree with all symbol nodes as leaves.

To build the a new internal node the least occurring symbol is removed from the min heap (the root node) but pointed to as a left pointer by the new internal node. Then the next least occurring symbol is then also removed from the min heap (also now the root node) and pointed to by the right pointer of the new internal node. The count of occurrences of the two least occuring symbols are summed and become the count value for the new internal node. Finally, the new internal node is placed back in the min heap effectively replacing 2 nodes with one. This process is continued until there is the single root node for the Huffman tree.

Once the Huffman tree is built we now have our Huffman codes, this is done by traversing the tree. Going left on the tree is represented as a 0 and going right on the tree is represented as a 1. The project implementation traverses the tree recursively to create a map of symbols to Huffman codes for the final step in the encoding process of actually encoding the file.

There are some challenges in creating an encoded file. The most significant being representing the Huffman tree in the encoded file. For this implementation a system whereby an internal node is represented as a 0 and a leaf is represent by a 1 and followed immediately by its symbol representation was used. The implemtation can work out from this algorithm when there are no more internal nodes left to traverse.

After the tree has been stored in the encoded file the next step is to make a final pass through the file and use the map of symbols and codes created earlier from the Huffman tree to append Huffman codes to the encoded file. A final challenge is presented as the proposition that an encoded file might not necessarily a multiple of 8 bits after the encoding process. To mitigate this a 3 bit header is attached to the front of the encoded file signifying how many bits to ingnore (0-7) in the decoding process.

2.3 Decoding

Decoding a file is relatively simple compared to the process of encoding a file as the most computational effort is performed by the encoded process. To decode an encoded file first the 3 bit header is read, then the Huffman tree representation is read in and built in memory. Finally every bit is read in one at a time, if a 0 is encountered we traverse down the left of the tree, if a 1 is encountered we traverse down the right of the tree. If we encounter a leaf node whilst traversing the tree the corresponding symbol is appended to a new decoded file and the traversing process starts from the root node again. This process is continued until the entire file is decoded and the original file is recovered bit for bit.

Chapter 3

Results

Bibliography

[1] "Data compression — Wikipedia." `https://en.wikipedia.org/wiki/Data_compression`. Online; accessed 29th November 2017.

[2] "Image compression — Massey Univeristy image processing study guide." Chapter 6.

[3] "Huffman coding — Wikipedia." `https://en.wikipedia.org/wiki/Huffman_coding`. Online; accessed 29th November 2017.

[4] "Lossless compression — Wikipedia." `https://en.wikipedia.org/wiki/Lossless_compression`. Online; accessed 29th November 2017.

[5] "Project github link." `https://github.com/gsalmond/summerProject`.