Specification

Literature Review of Compression Algorithms

Data compression algorithms compress a file with their counterpart, decompressing algorithms doing the opposite operation. Data compression reduces the size of files by encoding information in a particular way. This compression can be lossy or lossless. Lossy data compression reduces the size of a file by removing less critical information. The discarding of data is often done to avoid human's perceptions to be able to notice. However, this will become obvious when done to higher degrees or to more sensitive media like text documents. This also benefits from a more significant reduction in file size as data is being deleted, but this data loss makes the compression irreversible. On the other hand, lossless data compression decreases files' size by locating and eliminating statistical redundancy. This makes the compression reversible as no bits are lost, although it is significantly worse at compressing the file. In general, data compression is required since the need to transmit files quickly (smaller files mean fewer packets need to be sent) over networks.

Huffman coding is a lossless compression algorithm that David Huffman developed while he was at MIT. It was published in 1952 in "A Method for the Construction of Minimum-Redundancy Codes". This does not just have to be used on text documents but can be used on images. It uses a binary tree (known as the Huffman tree) generated using the frequencies of a list of different characters. Various characters' frequencies can be generated using some text, with longer texts being ideal for creating a more precise representation of the text's language. The more frequently used characters should be closer to the root of the tree. This is done since the encoding of a separate text document will require you to traverse through the tree for each character to find the character. A binary code will be produced from this traversal, with any traversals left adding a 0 to the code and to the right adding a 1. These binary codes can then be written to another file. To decode a compressed file, you need to do the reverse, with the code provided which route to take on the tree to get the character. Huffman coding is not always optimal. This problem mainly occurs when the text is used to generate the tree; it is dramatically separate from the encoding text, the text used to generate the tree is too short, or the document has lots of infrequently used letters. Overall, Huffman coding is beneficial since it is easy to implement (with only the usage of a binary tree). It can find a code in linear time when implemented efficiently and is a lossless compression algorithm. However, Huffman coding has only a compression ratio of about 1.5:1, meaning only a slight change in file size should be expected. This smaller compression ratio is not too bad since it is difficult to achieve over 2:1 due to data entropy (the natural randomness and unpredictability to data).

JPEG (Joint Photographic Experts Group) is a digital image format and compression algorithm. It is used to compression digital images and is the most widely used image compression in the world. JPEG is an example of block-based compression. The original specification for JPEG was published in 1992 and was formed from multiple earlier research papers. The algorithm's primary basis came from the discrete cosine transform (DCT) first proposed by Nasir Ahmed in 1972. A DCT articulates a finite sequence of discrete data points in terms of a cosine function at different frequencies. The cosine is used since it allows for fewer cosine functions to approximate a signal. JPEG compression goes through several different stages before being able to compress an image.

The algorithm starts by dividing the RGB data for each pixel into luminance and colour components. This is done since human perception is significantly more sensitive to the intensity of information (RGB) than the colour information. This allows for more colour information to be removed with a slight reduction in quality. The image is then split into eight by 8-pixel blocks before being transformed using the DCT. This means that the image's pixels now store the spatial frequencies for vertical and horizontal, not the binary value they had before. After this, the image is quantized. This process removes information, with more or less being lost depending on the size of the steps. This process will reduce many coefficients to zero, with higher coefficients being more probable to be zero. These values with zero can further reduce the file size by removing duplicates. This is a very effective compression algorithm that has been so popular due to its effectiveness at compression while not impacting quality. Even in these cases, it has a compression ratio of about 10 to 12 times, saving much space and reducing transmission time. However, it still has drawbacks due to being lossy and can only be performed on images.

Lempel-Ziv compression is made up of two compression algorithms published by Abraham Lempel (1977) and Jacob Ziv (1978). They are both lossless and make up the basis for many other algorithms. They are both dictionary coders, which are compression algorithms that operate by searching for matches between a set of the string in a dictionary and the text being compressed. When a match is found, the value associated with the text replaces it. Therefore, they are sometimes referred to as substitution coders. These two compression algorithms are pretty similar but do have some differences. Lempel's (LZ77) and Ziv's algorithm (LZ78) will spot matches and replace the data slightly differently. LZ77 will replace repeats of data with a single copy of that data from the uncompressed file, but LZ28 will replace repeats with references to a dictionary. This dictionary is built based upon the uncompressed file.

Similarly, LZ78 will use the dictionary to spot any matches, while LZ77 will use a length-distance pair, and the encoder would keep track of some amount of most recent data. Dictionary encoders work similarly to how a Huffman coder would work, using a data structure to store single characters that can be used to replace duplicates. An advantage of this type of compression is that it is pretty simple and requires a smaller data structure to compress the text. The algorithms' ability to not need first generate a binary tree but be creating a dictionary as it goes would reduce storage needs but could make dictionary compression slower on average due to the constant manipulation of the dictionary.

To sum it all up, the most effective compression algorithms at purely compressing would always be lossy forms of compression due to their ability to remove data, but this is often not possible with text files and with files being edited. Lossless files are much more flexible than lossy files since they can be run on more sensitive file types. Compression algorithms have been vital to increasing the ability to transmit files quickly over a network.

List of all Data Structures and Algorithms

* Binary Tree:

This is a specific type of tree where each node only has two children. This data structure was created using the Nodes.java and BinaryTree.java files. This is an important data structure used to compress and decompress a file using Huffman coding.

* Map<Character, Integer> (Dictionary):

This is a map, which is Java's equivalent to a dictionary. This data structure was used to store the frequency of each character in a training file. I used this since Java can only return one type at once, so this was a way of organizing both frequency and its character in one structure.

* Arrays:

These are primitive type arrays used to store several primitives. These were used to store many different values ranging from storing a binary in a list to be iterated. They are storing the list of characters to be checked and put into the dictionary one by one.

* String:

This is an example of an array, with a String being an array of characters.

Weekly Log

Week 1 – 1st March:

Creation of the Node.java file on the 5th of March. This is the file for the nodes in the binary tree. It was fully complete by the 6th of March.

Week 2 – 8th March:

Creation of the BinaryTree.java file on the 8th of March. This is the file that uses the nodes to create a binary tree.

Comments and Javadoc's comments were started on the 8th of March with a few comments detailing things to do. Javadoc's comments were started on the 11th of March, finishing on the same day.

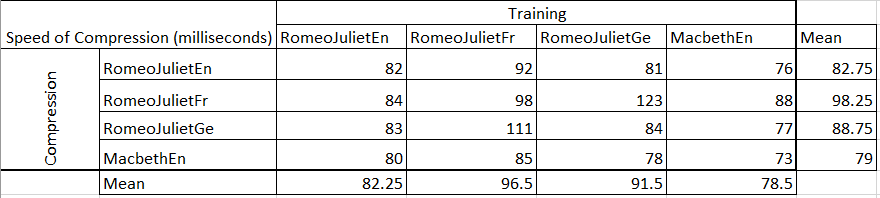
Creation of the HuffmanCoding.java file on the 8th of March. This is the file that will create, read, and write the files and create a dictionary between the frequency of characters in a file.

Creation of the HuffmanApp.java file on the 10th of March. This is the application where the main program will be run, tying all the other programs together. This was completed on the 11th of March.

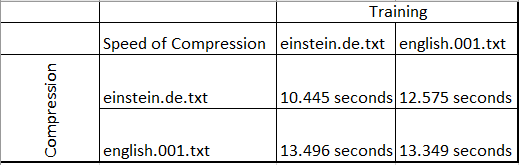
Literature Review was started on the 10th of March, with the introduction being completed. By the 11th of March, it was completed.

The list of all Data Structures and Algorithms was started on the 10th of March and was completed on the same day.

Performance Analysis

With the books, the Huffman coder was very fast across all the, with no compression taking more than 0.1 of a second. The slowest on average was with the French and the fastest way with the English Macbeth. The English Macbeth was likely the fastest since it is about 40kb less then Romeo and Juliet. I think the French book was slowest due to having more characters because of accents. This would make it slower to traverse the binary tree and would mean more character would be added when using a different training file to the file being compressed.

The Huffman code was relatively quick for different files over the 100MB size. In general, the algorithm was faster when the training code was the same as the file to be compressed.



A significant problem with the Huffman code is that it reduces or compress the size of the file. It generally increases the size of the file by between 2 to 4 times the original size. This is likely due to a fault with the allocation of leaves on the tree being incorrect. A leaf tends to go left in the tree resulting in rarely used characters having very long binary codes.

This is the compression file for MacbethEn.txt after being trained on the same file. The compressed file has increased by 252kbs, about three times.

