Compression Algorithms

In recent times, computers are used for almost everything. Lots of data is stored on devices, taking up memory space. In order for the storage to be efficient, it is important that files are stored as compactly as possible. One way in which this can be achieved is with the use of compression.

Compression is used almost everywhere [5] – watching videos, surfing the web, taking pictures, listening to music – and because of this, many different compression algorithms have been invented. Compression is used to store data in a more compact form.

Compression Algorithms are used to reduce the size of large files when storing them in memory, so that they take up much less space. There are many different types of compression algorithm; Huffman Coding, Arithmetic Coding, Relative Lempel-Ziv, Fixes-Block Compression and Dictionary-Based Compression to name a few.

Compression algorithms can have different performance factors, but it is often seen that the larger the input data, the longer it will take to compress, and the larger the compressed file will be. They are often more than O(n).

Lossless compression is where, when the file is decompressed, it is exactly the same as the original file with no data loss. This ultimately takes up more space in memory but means that there are no errors or losses in the decompressed file. Examples of this include PNG and GIF. In Run-Length Encoding, if the file were, for example, *AAAAABBBCBBEFEEE*, the compression would be stored as 5xA, 3xB, 1xC, 2xB, 1xE, 2xF, 3xE, which will take up less space in memory, and when expanded again this returns the exact string of characters we had before. This is particularly important if you don't want any data to be lost.

Lossy compression is where some data is permanently deleted from the file in order to compress it. This will therefore take up less memory space, but when the file is decompressed, there is an issue involving data loss, however minute. Examples of this include JPEG and MP3. In Run-Length Encoding, if the file were, for example, *AAAAABBBCBBEFEEE*, it would be compressed differently to the lossless compression. Instead, it could be stored as 5xA, 6xB, 5xE. When this is decompressed, the resultant message is *AAAAABBBBBBEEEEE*, which is remarkably similar to the original message, however, it has lost data in the form of the characters 'C' and 'F'. If, as an example, this file represented pixel colours in an image, it would be hardly noticeable if one pixel that was slightly different in stream of single-coloured pixels was changed to be the same as a stream, but if this were a text file it would be bad as it would be much harder to read. Lossy compression is therefore used for audio and image compression, as it greatly reduces the large file sizes, so is very memory efficient. Lossless compression is used for data where it would be very noticeable if the data had changed, as this would be an issue upon decompression.

Huffman Coding

Huffman coding, developed by David Huffman, is a greedy algorithm used for lossless file compression. The algorithm takes the frequency of each unique character in a file, and the character with the highest frequency is given the binary code with the smallest number of bits. Each character code is a prefix code; that is, no code is the prefix of another code. This means that the file can be read without confusion [3].

Huffman is implemented using a binary tree, whose leaves represent the unique characters. Each non-leaf node has two branches and therefore two children (it is a full binary tree), and the code for the character is read using the path of branches needed to reach it. The left branch represents a 0 and the right branch represents 1, so if a node was found by following the right branch, then the left,

then the right again, the code would be 101. Each tree has enough leaves for the size of the character set, and the number of nodes is one less than this.

The tree is built using Huffman's algorithm. The process identifies the two least-frequent characters in the character list and merges them together to create a node. This node's frequency is the combined frequency of the two characters, *node.left + node.right*, and it has properties *left* and *right*, which contain the lowest and second-lowest frequency items, respectively. The node is added to the character list, and the two individual characters are removed from the list. The program then repeats this process, choosing the two objects with the lowest frequencies in the list, merges them to create a node, then removes the two separate objects. This is repeated until there is only one item – the root node – left in the character list, with two properties *left* and *right*, which themselves contain nodes or characters, as mentioned earlier.

The file to be compressed is then encoded, replacing the ASCII or Unicode encoding with the relevant Huffman codes for that character. The file could be up to 90% smaller after compression, however, Huffman is an O (n log n) algorithm, so the larger the file, the longer it takes. The running time can be reduced to O(n log log n) by using a van Emde Boas tree, instead of the binary tree.

Arithmetic Coding

Arithmetic Coding, created by Risannen and Pasco [5], uses arithmetic operations on the code in order to compress the file. [1] It encodes an entire file as a sequence of symbols into a decimal number [4]. The input symbols are processed over iterations, and the intervals are created according to a probability distribution, with the subinterval that links to the input symbol being selected for the next iteration. It is used in a variety of lossy and lossless compression algorithms.

The algorithm of Arithmetic Coding replies on the face that there are infinitely many numbers between 0 and 1, so it is possible to assign a unique number from the interval to a sequence of symbol from an alphabet [6]. This sequence can then be coded as a unique, single number, which has a binary representation.

To implement this, a mapping is needed, e.g., the cumulative density function of the character sequence, which is a number between 1 and 0. This number would be unique to the sequence and can be split into intervals. Any number in the interval can be used as a label for the sequence. For example, the interval could be [0.55, 0.64].

The problem with Arithmetic Coding is that as the character sequence length increases, so does the number of digits needed to represent it. However, it has a much better performance than other compression algorithms, such as Huffman Coding.

Dictionary-Based Compression

Dictionary-Based Compression is relatively common. It achieves a good balance between memory usage, compression ratio, speed, and simplicity [2]. This compression algorithm works by matching phrases in the file to be compressed to its own dictionary, with the dictionary implicitly transmitted. This is called on-line compression, where the phrase in the dictionary can be referenced. A drawback is that the dictionary may also have to be transmitted. There are several other ways in which this compression algorithm can be used.

Data Structures needed for this type of compression include an array to store the characters to be compressed, a hash table for each 'active pair'; two adjacent symbols, and a specialised priority queue recording active pairs.

This type of compression algorithm has the Big-O notation O(n), which is linear, so the larger the input size, the longer it will take. This means that it is very efficient for smaller input, but less so for larger inputs, although generally it is more efficient than a lot of other algorithms.

An added difficulty to this algorithm is that the dictionary must be encoded, which is an added overhead compared to other algorithms. This could take up more memory space.

Data Structures and Algorithms Used

Data Structures

- HashMap – this is used to record each unique character in the alphabet of the input text, along with its frequency in the said text. It is important that each character is unique, hence why a HashMap is used as hashes are different for each specific character.
- Tree – this is used in order to implement Huffman Coding. The unique character code is found by traversing the tree, with the left node representing a 0 and the right node representing a 1. The tree stores nodes, and each leaf represents a character of the alphabet. Each inner node has two branches. It is a full binary tree and is implemented using classes and ArrayLists.
- Class – this data structure is used in order to be able to create node objects, in order to implement the tree. The nodes have properties – *frequency*, *left*, *right* and *character* – which describe them in order for the tree to exist.
- ArrayList – this data structure is used to store the nodes in the tree and is sorted using the Quicksort algorithm, so the nodes are listed smallest frequency to largest.
- File – this data structure is used to store the text to be compressed (.txt), the encoded and compressed text (.bin), and the output of the decompression algorithm (.txt). These are stored in folders on the computer and are opened and closed using a FileReader and OutputStream in the program.
- String – this data structure is a sequence of characters. It is used to store the information to be compressed and the compression output of the files.
- Array (of bytes) – this is used to read the binary from and write the binary to the input and output files. The array of bytes is needed to read each byte in the file.

Algorithms

- Huffman Coding – this algorithm is a compression algorithm that uses a tree in order to reduce the bit size of the most frequent characters in a file. Huffman has been explained in more detail in the *Compression Algorithms* part of this literature review, please reference that for more information.
- Huffman Decoding – this algorithm takes the binary digits and traverses down the tree, left for 0 and right for 1, in order to find the corresponding character to the binary code. It is used to decompress Huffman files.
- Quicksort – this is a sorting algorithm used to sort a list into ascending order. It chooses a pivot point, then moves all items larger than the pivot to the right of the pivot, and all items smaller than the pivot to the left of the pivot. The pivot is then in the correct position. This process is repeated for all points as pivots to the left of the original pivot, and then again for all points to the right of the original pivot. All positions are then in order. This algorithm is O(log n) and therefore takes longer the larger the input, but the increase is logarithmic, so it is actually a very efficient and quick algorithm.
- Get Character Frequency – this algorithm is used to find out the frequency of unique characters in a String or text file. It uses a HashMap and for loops to record the frequencies.
- Write to File – this algorithm is used to write text or binary code to a .txt or .bin output file.

- Read from File – this algorithm is used to read from a .txt or .bin text or binary file, with the information being used in the program.
- Compare File Sizes – this algorithm uses comparative statements in order to compare the file sizes of the original file and the compressed file.
- Create Tree – this algorithm uses the HashMap to create an ArrayList of nodes from the input data in order to create the Huffman binary tree.

Weekly Log

- Week 1 - Starting 11/01 – This week I printed off the specification, read it through and made nodes. I researched Huffman Coding and wrote the literature review on it.
- Week 2 - Starting 18/01 – This week I researched two other compression algorithms – Arithmetic Coding and Dictionary-Based Compression and completed the literature review.
- Week 3 - Starting 25/01 – I started implementing the program in Java by creating a function that uses a HashMap to record each unique character in a text file, and its frequency in the same file.
- Week 4 - Starting 01/02 – I created the node class and set each character as a 'leaf' in that class so the tree could be implemented later. Each node was added to an ArrayList that will later become the tree.
- Week 5 - Starting 08/02 – I needed to sort the ArrayList in ascending order of frequencies and did not realise that Java had a .sort() algorithm, so programmed a Quicksort algorithm to sort the nodes.
- Week 6 - Starting 15/02 – I started implementing Huffman Coding to compress and decompress the algorithm, as well as saving it all to files.
- Week 7 - Starting 22/02 – I debugged my program as I ran into issues like Stack Overflow Errors and Scanners not reading lines correctly. I fixed these problems.
- Week 8 - Starting 01/03 – I started to test my program, fixed bugs, and wrote up and tested the performance analysis.
- Week 9 - Starting 08/03 – I completed the write up, created a README file, and uploaded the finished coursework.

Performance Analysis

The compression program overall is not as efficient as I would have liked due to time constraints and the pressure of other coursework – this meant that I did not have time to do the full extent of testing that I would have liked.

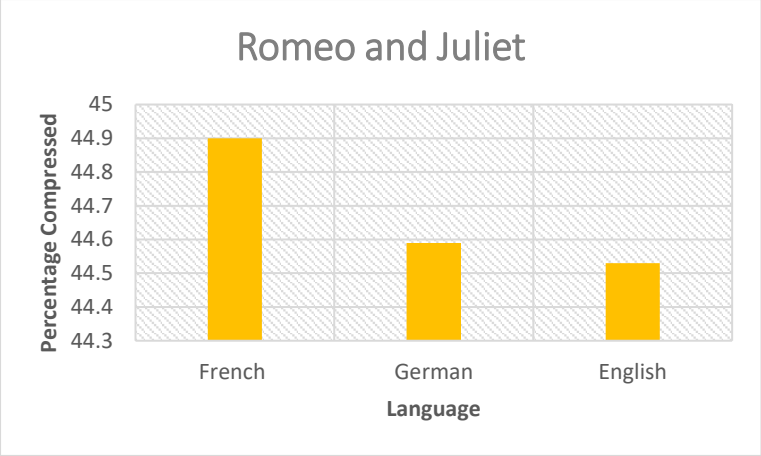
It was possible, however, for me to try extracts of text in different languages, as well as extracts of the large data sets, in order to produce some graphs for analysis of performance. I took the same sized extract in each language an ran it to produce the below graphs.

Each graph shows the three languages (or data sets), and the percentage compression when they were run through the program. Most differences were small, however, there was more variation in the different data sets.

Unfortunately, I did not make a record of the exact extracts used from the texts. This is something I would do if I had more time or were to do the project again, as this is much easier for testing. If someone else was to test it, I would recommend taking a small extract of the same passage themselves and to keep a record of it as this makes it much easier to analyse in the future.

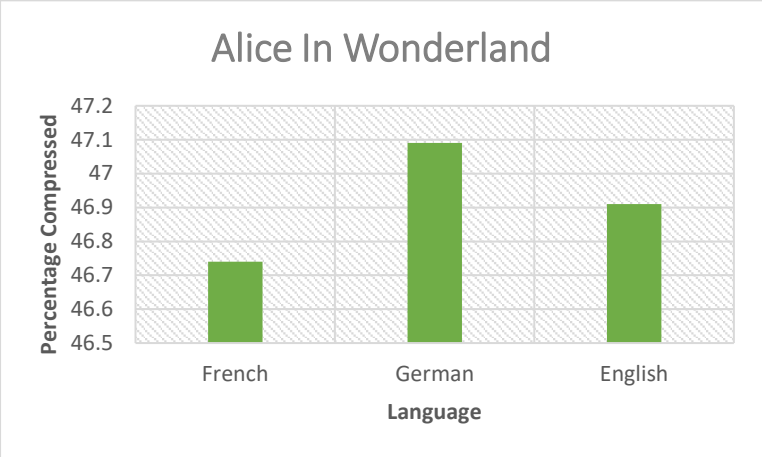
Romeo and Juliet

For Romeo and Juliet, the algorithm best compressed the French text, however there was a only a miniscule difference between each language.



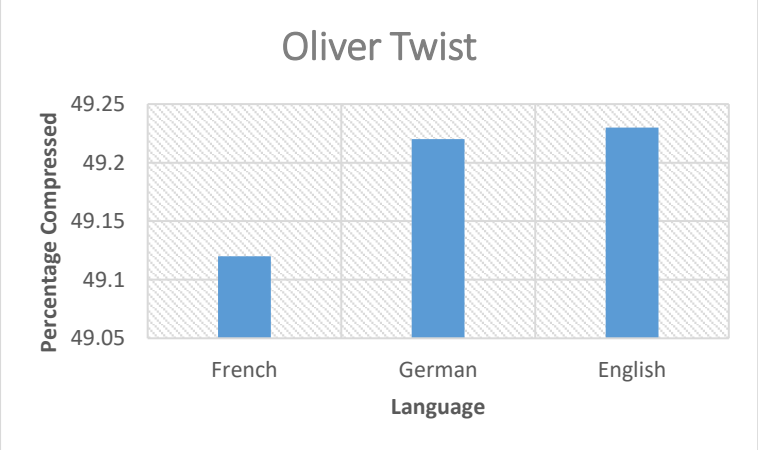
Alice in Wonderland

For Alice in Wonderland, German was the best compressed language, followed by English. There was more variation here than with Romeo and Juliet.



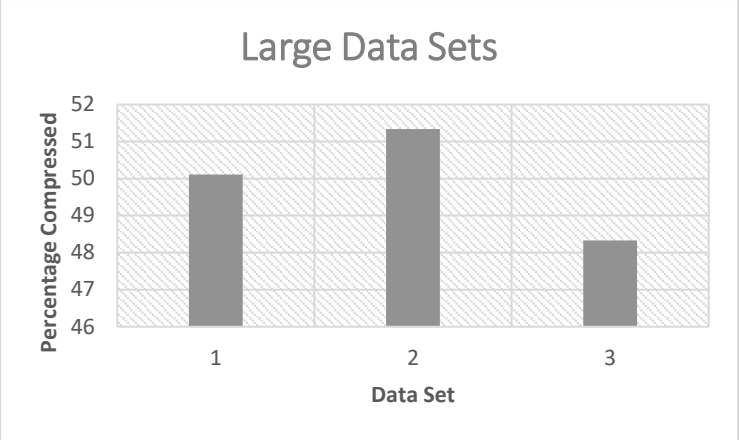
Oliver Twist

English was the best compressed language for Oliver Twist. There was a minuscule percentage difference here.



Large Data Sets

The large data sets showed a larger variation as some of them were a lot more repetitive than others.



References

[1] G. G. Langdon Jr, "An Introduction to Arithmetic Coding," *IEEEXPLORE*, vol. 28, no. 2, Mar. 1984, Accessed: Mar. 12, 2021. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5390377.
[2] N. J. Larsson and A. Moffat, "Off-Line Dictionary Based Compression," *IEEEXPLORE*, vol. 88, no. 11, Nov. 2000, Accessed: Mar. 12, 2021.
[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. Cambridge, Mass.: Mit Press, 2009, pp. 428–437.
[4] Ida Mengyi Pu, *Fundamental data compression*. Oxford ; Burlington, Ma: Butterworth-Heinemann, 2006.
[5] Khalid Sayood, *Introduction to data compression*, 5th ed. Cambridge, Ma: Mogan Kaufmann, 2018.
[6] Khalid Sayood, *Encyclopedia of Information Systems*. Academic Press, 2003.