Compression Algorithms

Files are pieces of data stored on a computer that encode information such as images, audio, text etc. Since files take up lots of space on computers, there is a desire for a more efficient way to store data in order to transmit them more quickly and reduce the space they take on hard-drives. Compression achieves this encoding data using fewer bits than the original representation, this can be done by reducing repeated information or by run-length encoding, where the length of the run is specified for identical files [1]. There are 2 main types of compression; lossless and lossy. In lossless compression the original data can be recovered exactly from the compressed data [2], whereas in lossy compression less important information is discarded [3]. This usually takes place in image and video compressions, as it would be less evident to lose a few pixels compared to text. There are various different examples of compression and in this literature review Huffman Coding, Arithmetic Coding and Relative Lempel-Ziv will be explained.

Huffman Coding

In the English-language each possible character is saved as 8 bits as a unique combination of 0's and 1's, allowing for 256 such-possible combinations. This includes spaces and special characters not in the alphabet. Since, the only thing that is visible to the computer is the stream of 0's and 1's it was difficult to try encompass everything into smaller bits [4]. David Huffman decided to create a greedy algorithm in 1950 that would assign codes to characters such that the length of the code depends on the relative frequency or weight of the corresponding character [5].

Huffman coding is implemented by creating a Huffman tree which is a binary tree where each leaf of the tree relates to a letter/character in the text. It is then counted how many times each letter/character appears and a list of this is put into an order using a letter frequency table for example. The 2 characters/letters that appear the least frequently are positioned as the bottom branches of the binary tree and they are then labelled with their frequency and connected together by the sum of their frequencies one level up [4]. We can label these as nodes and describe them as being connected to a parent node. This process is repeated for each and every character/letter by taking the next 2 least frequent characters/letters and connecting them to a parent node. The value of the parent node must be equal to the frequency count of the character/letter of the 2 nodes being connected. Once all of the letters/characters are connected, it is possible to say that the Huffman Tree/ Root Node has been built.

The left-hand side of the tree can be noted as '0,' and the right-hand is noted as '1.' In order to compress the file it is converted to the string of bits and in order to decompress it is converted from the string back to the original text. To decode a letter/character you would need to traverse the tree by using the path of branches. To begin, you have to start at the root node which is the main parent node of the entire Huffman Tree. If the bit being read is 0, you have to move to the left node of the tree, alternatively if the current bit is 1, you have to move to the right node of the tree. While traversing the tree, if a leaf node is encountered, it signposts the character/letter and it is decoded to represent it. This process repeats from the leaf node the entire file is decoded.

Some of the advantages of Huffman Coding involve it being lossless compression which means no data is lost or disregarded. It also generates shorter binary codes for encoding letters/characters that tend to appear more frequently in the file, making it extremely efficient [6]. Additionally, Huffman encoding is easy to decode because the decoder has to follow the tree method simply by going left for '0' and right for '1.' However, a disadvantage of Huffman Coding include that it is possible for the decoder to generate the wrong output or might be difficult for the decoder to detect if any of the encoded data is corrupt, because the length of all of the binary codes are different [6]. Also, it could be said to be slow when building the tree from scratch for each character/letter.

Arithmetic Coding

Arithmetic Coding was Pasco, Risannen and Elias in 1987. It is a popular alternative to Huffman Coding and is particularly useful for smaller and relatively skewed alphabet [7]. Therefore it could be more useful to implement arithmetic coding when applying it to different languages with lots of accented characters such as French or Spanish. The arithmetic coding algorithm essentially encodes a file containing data as a sequency of symbols into a single decimal number [7]. Arithmetic Coding is usually used for lossless data compression, therefore there is no data that is lost. It is also represented using a fixed number of bits per letter/character.

In order to encode a file using arithmetic coding you need to start with an interval that is initialised to (0,1). This range is selected as it is implied that because there are many possible numbers between 0 and 1, we can select a decimal value from this range and map it to a specific sequence of message from a text data set. This value is unique to the message and has its own binary form. The input message is encoded over multiple iterations by dividing the current interval into subintervals. These subintervals created are based on a probability distribution with each subinterval connected to the next input message that is selected for the next iteration [8]. As the message becomes longer and longer, the corresponding interval needed for it to be represented becomes smaller and smaller, therefore the number of bits that specify the interval grows [9]. It is similar to Huffman coding in the way that letters/characters that appear more frequently have fewer bits compared to those that appear less frequently because they reduce the length of the interval.

The main advantage of Arithmetic Coding include that it is dynamic and adaptable. This is because the frequency can quickly adapt whilst changing the data, whereas in Huffman coding the frequency has to be in-built into the method [8]. Additionally, it is easy to maintain the order of the messages without any loss in compression efficiency, making it efficient. However, a disadvantage of arithmetic coding is that it can be slow because of the complex nature of implementing. Additionally, if the sequency of the message is long the unique digit assigned to it will also be longer making it slower.

Relative Lempel-Ziv

Relative Lempel-Ziv is an algorithm used to compress databases when fast random access is desired [10]. It was created in 1978 by Abraham Lempel and Jacob Ziv as a lossless compression data algorithm and is usually used to compress image files. It relies on the patterns that frequently reappear to compress the data. As explained in the Huffman coding section above, there are 256 unique combinations within 8 binary bits for the English-language. However, in RLZ (Relative Lempel-Ziv) the algorithm tried to extend this to contain around 9-12 bits per letter/character. These new unique symbols are made up of combinations of symbols that already exist within the string [11].

The method of encoding involves having a base-line of frequently appearing symbols/messages that can be compressed to a few letters/characters. Each time a new letter/character is found it will be recorded as the first time it has been seen. The data is then continued to be read and then when the same letter/character is found again it will begin recording a message until a new letter/character if found and then the message chain is broken. In order to decode this data the method of encoding needs to be followed but reversed. The decoder needs to build its own dictionary of sorts to decode the data via the symbols.

RLZ is advantageous for compressing redundant data because it relies on already existing combination of symbols, which would make it quick and efficient to decode. There is also no need to have a selected range and intervals like in Arithmetic Coding or a binary tree like Huffman Coding so it can be easy to decode with its own dictionary. Additionally, it can be encoded and decoded within the same file itself. However, with shorter and more unique strings it does not compress as efficiently because there are no repeating patterns. Therefore, in order to send something unique it would cause RLZ to be inefficient and would take up lots of time. Additionally, a very large file could take very long because the decoder needs to keep referring back to its created dictionary.

Data Structures and Algorithms implemented

Data Structures

- **Arrays** (Used in Byte Arrays, Bits to file and frequency: To read the binary string and bits from the input file path and also write it into the compressed file path. The array is used to read each byte in the file.)
- **Files** (Included the scanner for the user to input the file paths for the original text, compression and decompression. These files were also traversed through in order to extract the data via strings. FileInputStream, FileOutputStream and FileWriter are part of the file data structures needed to traverse the string.)
- **Tree** (This data structure was needed as part of the nodes in order to retrieve the codes to traverse the tree. The left node meant a 0 bit and the right node mean a 1 bit.)
- **Priority Queue** (The priority queue data structure was implemented in order to build the Huffman tree from the lowest frequency and connecting it to the parent nodes. In this case the element with the high priority was served first in constructing the tree, these were the letter with the lowest frequency.)
- **Strings and String Buffer** (Strings represents the sequence of the text either in binary format or in ASCII and the String Buffer is utilised to ensure mutability for the string.)
- **Class** (This was used to create the objects for the node including left, right, frequency and character)

Algorithms

- **Get Character Frequency** (A frequency array generated that uses a for loop to find out the frequency of unique characters in the text file.)
- **Generating Byte Array** (Creating a new byte array from a string and dividing the length by 7, one less than the standard of 8 and parsing by 2 in order to generate the correct binary representation.
- **Writing to File** (Used when writing to a new file and writing bits into the file for effective compression and decompression.)
- **Reading to File** (Reading from a .txt or .bin and reading bytes from a file for effective compression and decompression.)
- **Build Huffman Tree** (Creation of the Binary tree using the priority queue method to connect nodes of lowest frequency to parent nodes.)
- **Compress** (With help of the Build Huffman Tree, it executes by reducing the bit size of most frequent letters/characters that appear.)
- **Decompress** (Retrieves the current bit and then traverses down the tree following the left or right depending on the binary code of 0 or 1 until a leaf node is reached. Once a leaf node is reached the character is extracted.)

Weekly/Daily Log

Week 1	Researched the background of Compression Algorithms and watched some videos on Huffman Coding to gain an idea of how to implement it in Java.
Day 2	Started implementing the Java program, decided the structure and implemented a rough draft of the main method.
Day 3	Created the Nodes class with the getter and setter methods and the frequency table. Implemented the build Huffman Tree method with help from Stack Overflow.
Day 4	Implemented the reading and writing method to files

Day 5	Created the byte array and the bits to file method.
Day 6	Implemented the compress and decompress method.
Day 7	Debugging the program with help from Stack Overflow; realised implemented a wrong method for decompression.
Day 9	Debugging the program
Day 10	Debugging the program and conducting the performance analysis.
Day 11	Researched for and completed the literature review.
Day 12	Completed the literature review and recorded the video for submission. Generated a license from GitHub.

Performance Analysis – Books

For the performance analysis I have compressed 3 books in 3 different languages; English, French and German respectively via the Gutenberg website. The table of results in below:

	Original Size (Bytes)	Compressed Size (Bytes)	Percentage of Compression (%)
English - Merchant of Venice (MOV)	145,637	94,890	34.84
French - Merchant of Venice	183,664	121,176	34.02
German - Merchant of Venice	153,455	100,431	34.55
English - A Midsummer Night's Dream (MND)	121,627	79,148	34.93
French - A Midsummer Night's Dream	157,543	104,351	33.76
German - A Midsummer Night's Dream	122,094	80,260	34.26
English - King Lear (KL)	185,962	117,006	37.08
French - King Lear	247,042	163,277	33.91
German - King Lear	212,853	137,144	35.57

The percentage of compression represents the compression as a percentage of the original file size.

From this table above, it is possible to generate the reports below:

Figure 1

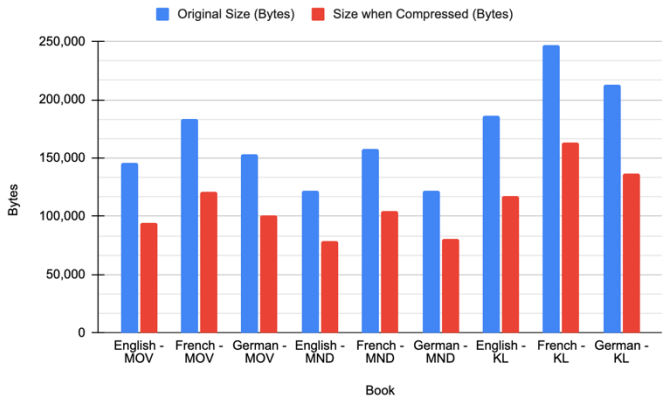


Figure 2.

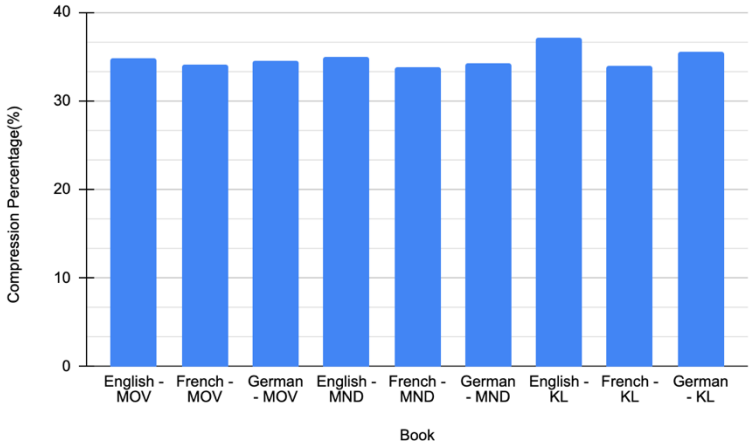
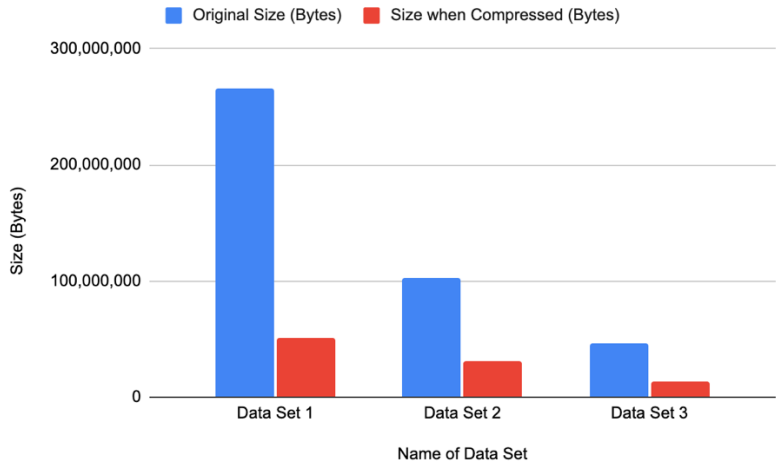


Figure 1 shows the original size and the size when compressed of the books in all 3 languages. It is evident that some files are originally bigger than others. For example, I am assuming that because French contains more accented characters and letters its file size is bigger than that of English and German therefore its compressed file is also slightly bigger. Overall, the compression seems to be efficient and working and has reduced the file size by 34% on average. Interestingly as shown in Figure 2 it does not matter which file it is the compression as a percentage of the original file size is almost the same for all of the languages.

Performance Analysis – Repetitive Corpus Data Set

Due to main error Heap File large data sets should have increased the ALPHABET_SIZE in Huffman_Coding.java and should be run in the terminal via java -Xmx2048m. Although I still received errors while processing the repetitive corpus data set it is possible to run a large data set using the algorithm as shown by the graph below:



The table for this data:

Type	Original Size (Bytes)	Size when Compressed	Percentage of Compression (%)
Data Set 1	265,900,291	51,109,845	19.22
Data Set 2	103,016,600	31,456,000	30.54
Data Set 3	45,885,814	13,496,421	29.41

Bibliography

[9]2021. [online] Available at: <https://www.researchgate.net/publication/2984808_Arithmetic_Coding_for_Data_Compression > [Accessed 26 March 2021].

[10]Cox, A., Farruggia, A., Gagie, T., Puglisi, S. and Sirén, J., 2021. *RLZAP: Relative Lempel-Ziv with Adaptive Pointers*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1605.04421#:~:text=Relative%20Lempel%2DZiv%20(RLZ),fast%20random%20access%20is%20desired.&text=We%20show%20experimentally%20that%20our,with%20comparable%20random%2Daccess%20times. https://www.youtube.com/watch?v=j2HSd3HCpDs> [Accessed 26 March 2021].

[4] Homes.sice.indiana.edu. 2021. *Huffman encoding*. [online] Available at: <http://homes.sice.indiana.edu/yye/lab/teaching/spring2014-C343/huffman.php> [Accessed 26 March 2021].

[3] Khan Academy. 2021. *Lossy compression (article) | Khan Academy*. [online] Available at: <https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:digital-information/xcae6f4a7ff015e7d:data-compression/a/lossy-compression> [Accessed 26 March 2021].

[7]Sciencedirect.com. 2021. *Arithmetic Coding - an overview | ScienceDirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/computer-science/arithmetic-coding> [Accessed 26 March 2021].

[6] Sciencedirect.com. 2021. *Lossless Compression - an overview | ScienceDirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/computer-science/lossless-compression> [Accessed 26 March 2021].

[5]Studytonight.com. 2021. *Huffman Coding Algorithm | Studytonight*. [online] Available at: <https://www.studytonight.com/data-structures/huffman-coding> [Accessed 26 March 2021].

[8]Web.stanford.edu. 2021. [online] Available at: <https://web.stanford.edu/class/ee398a/handouts/papers/WittenACM87ArithmCoding.pdf> [Accessed 26 March 2021].

[1] Youtube.com. 2021. [online] Available at: <https://www.youtube.com/watch?v=OtDxDvCpPL4> [Accessed 26 March 2021].

[2] Youtube.com. 2021. [online] Available at: <https://www.youtube.com/watch?v=JsTptu56GM8)> [Accessed 26 March 2021].

[11]Youtube.com. 2021. [online] Available at: <https://www.youtube.com/watch?v=zSsTG3Flo-I> [Accessed 26 March 2021]

Sources for Programming in Huffman_Coding.java

https://stackoverflow.com/
https://www.youtube.com/watch?v=zSsTG3Flo-I
https://www.youtube.com/watch?v=oNPyYF_Cz5I