Section 1 – Literature review of compression algorithms

Data compression have played a crucial role in our daily lives, whether transferring a large file to a team member, or streaming music from the internet, even when it comes to downloading your favourite video games from the cloud. Compression algorithms have assisted us to save spaces on the disk and reduce data usage when downloading files, this happens while keeping changes to the file minimal.

Compression algorithms can be mainly divided into two types, namely lossy compression and lossless compression. Lossy compression, or irreversible compression, refers to compression algorithms that discard some of the original data to represent the content, this results in a smaller file size but some data from the original file is lost. It is widely used to compress files such as JPEG images, MPEG videos and MP3 audio formats. Lossless compression, on the other hand, means that the compression algorithm compresses the original file while keeping all data from it, therefore it is also known as reversible compression.

The focus of this coursework is to implement a lossless compression algorithm that compresses a text file. Out of many compression techniques, Huffman Coding has been used in this coursework. In the following paragraphs, Huffman Coding, along with two other lossless compression algorithms will be evaluated.

Huffman Coding

The first lossless compression algorithm to be explored is Huffman Coding. This compression algorithm is created by David A. Huffman in 1951 for his term paper, with the idea of compressing a file using a frequency-sorted binary tree (Stix, 1991, p.56). Normally, each character uses fixed-length encoding, meaning every character is stored using 8-bits regardless of their frequency, this may be inefficient for symbols that appear frequently. The algorithm uses an idea of variable-length encoding, meaning that characters that appear frequently can be represented using a lesser number of bits (Huffman Coding Compression Algorithm, n.d.).

The compression using Huffman Coding can be achieved by choosing a representation for each symbol, containing a prefix code and a bit string. Before evaluating the steps of Huffman Coding, there are serval keywords that are essential to understand how the compression works. To start, a node is a datapoint for larger networks, such as binary trees. An internal node refers to nodes that contain child nodes, where, by convention, bit 0 represents the left child and bit 1 represents the right child. A leaf node, on the other hand, refers to nodes that do not have any child nodes. To compress a given text using Huffman Coding, nodes will be created for each distinct symbol in the text, the frequency of that symbol will also be stored in the node for reference. A new internal node will then be created, with the children of the node being the two nodes with the least frequency, the frequency of the created node will be the sum of the frequency of the two child nodes. This process is applied until there is only one node remaining, making it the root of the binary tree (Wikipedia contributors, 2021).

After creating the binary tree, it will then be traversed to generate a dictionary, mapping symbols to binary codes. This process starts with setting the root node as the current node and labelling the edge to the left child as 0, and the right child as 1 if the current node is not a leaf node. This process is repeated for the left and right child of the current node until the current node is a leaf node, meaning that the algorithm has reached the bottom of the binary tree. The text can then be encoded using the dictionary, translating every character to the binary code that represents it. The dictionary generated and the encoded string will then be stored in a file, which contains the information required for decompressing the string, and have a smaller file size than the original text file since the encoded string takes up less storage after encoding.

Decompression of an encoded string is relatively straight forward. The decoder only requires the dictionary and the encoded string to translate bits back to symbols. This is done by translating the stream of binary codes to individual byte values, meaning that 0s and 1s from the encoded string are translated back to symbols. After decompression, the output should be identical to the original text passed into the compression algorithm.

Huffman Coding is a compression algorithm that generally has a good compression ratio and is easy to implement. Using this algorithm allows us to compress a text file that is smaller than the original file since binary codes generated are variable in length, with symbols that appear more frequently represented by shorter binary codes, this can benefit us since we can store a text file with less storage space than the original file while not encountering any data loss to the file. However, using Huffman Coding comes with some disadvantages. For instance, lossless compression algorithms, like Huffman Coding, often achieve a lower compression ratio when compared with lossy compression algorithms, the compression is also generally slower since the algorithm requires one pass to build the statistical model before another pass to actually encode the text. The decoding of Huffman Coding may not be reliable as well since it is difficult for the decoder to detect data corruption since the binary codes are different in length, resulting in outputting an incorrect string (Huffman Coding Algorithm, n.d.). The optimality of Huffman Coding may also vary depending on the content of the original text. For instance, the algorithm may not be optimal when symbols are not independent and identically distributed, resulting in a similar result as storing characters with fixed-length encoding since the compressed binary codes cannot benefit from encoding with fewer bits for more frequent characters (Wikipedia contributors, 2021).

In terms of real life applications, Huffman Coding is a data compression method that is independent from the data type, it is therefore widely used in compression files. For example, it is used as a back end for other compression methods such as PKZIP, as well as compressing multimedia codecs like MP3 and JPEG files.

Arithmetic Coding

Another lossless compression algorithm to be evaluated is Arithmetic Coding. This compression algorithm is sometimes used since it addresses some problems and inefficiencies that come with Huffman Coding, especially when the probability model is biased. Arithmetic coding is originally an idea by Elias, who is one of Huffman's classmates in the early 1960s. However, the first practical approach was not published until 1976 by Rissanen from IBM (Tsai, 2014). Similar to Huffman Coding, Arithmetic Coding compresses a text file by representing symbols with a higher frequency with a lower number of bits. However, different from Huffman Coding, Arithmetic Coding encodes the entire message into a single number rather than separating the input into symbols and replacing each of them with a binary code (Glen, 1984, p. 144).

Before encoding the text, a model will first be defined. This model is a prediction of what symbols will appears in the message and is often implemented using a frequency table which is similar to Huffman Coding. The frequency table is first created by counting the frequency of each distinct character that appear in the original text, it is then translated into a probability table, containing the probability of each character which can be found by dividing the frequency of a particular symbol with the sum of the frequency of all symbols from the text. This probability table will be essential in determining the sub-range of each character in the encoding process (Witten et al., 1987, p. 535).

The encoding of Arithmetic Coding requires two inputs, namely the message to be encoded, and the probability table which contains the probability of all distinct characters. In the beginning, a line that ranges from 0.0 to 1.0 is created, this line is used to represent the cumulative probability of all symbols. To start, the first symbol is passed into the encoder, assigning a sub-interval for every symbol passed. In convention, the start position of the passed character will be assigned to the lower limit S of the current interval, and the upper limit can be calculated by the formula S+(P(C)*R), where S is the starting point of the sub-interval, P(C) is the probability of character C, and R is the range of the line (Gad, 2021). After the interval is created, the interval will be restricted depending on the range of the sub-interval of the first character. For instance, if the sub-interval of the first character lies between 0.0 and 0.2, the range of the interval will be restricted, with 0.0 being the lower limit, and 0.2 being the upper limit, the lower and upper limit of other sub-intervals can be obtained using the formula above. This process is repeated until there are no more remaining characters to be encoded. Afterwards, the average value of the final interval is obtained and stored, along with the frequency table and the number of symbols in the original message, for decompression.

To perform decoding using Arithmetic Coding, a total of three inputs is required. Firstly, the single value that encodes the message is required since it contains the encoded text. The frequency table which is identical to the one used for encoding, along with the number of symbols in the original message, is required to reconstruct the probability table, representing the probability of each symbol. Similar to the encoding function, the decoding first constructs a line with interval 0.0 and 1.0, then assigning sub-intervals to each distinct character, creating the sub-intervals as compression function (Gad, 2021). The decoding algorithm will then compare the encoded value with sub-intervals, deciding which interval does the value falls into, the character of the sub-interval will then be appended to the result string. This process will then be followed by restricting the range of the interval to be the range of the current sub-interval, similar to the compression function. This process will be repeated until the number of iteration reaches the value of the number of symbols in the original message, meaning that the whole string is built and decoded.

Using Arithmetic Coding to compress a text file is optimal when the probabilities of characters are known since we can build an accurate model resulting in better compression performance. One of the advantages of using Arithmetic Coding over other compression algorithm is it's separation of coding and modelling since it allows changing the modeller while not requiring to change the coder. Although Arithmetic Coding is sometimes used as an alternative to Huffman Coding, it still has some disadvantages. For instance, using this compression algorithm is relatively slower and is more complicated to implement than other compression algorithms due to its complexity. Moreover, the property that it does not use prefix codes may lead to some technical difficulties and errors while compressing and decompressing the codes (Howard & Vinter, 1994).

LZ77 Compression Algorithm

The last lossless compression algorithm to be evaluated is the LZ77 compression algorithm. This compression algorithm was first published by Abraham Lempel in 1977 in the paper. The compression works by replacing redundant information of the input data with metadata, which is used to indicate how to expand the compressed sections again (Wikipedia contributors, 2021). The theory of LZ77 compression will be explored below.

The compression of text using LZ77 compression is first done by setting the coding position to the start of the input stream, followed by finding the longest match in the window for the lookahead buffer. A pointer P will then be outputted, and the coding position will be moved L bytes forward if a match is found. Otherwise, a null pointer, along with the first byte in the lookahead buffer will be outputted, the coding position will also be moved one byte forward. This process will be continued until the lookahead buffer becomes empty. After the compression, a series of bytes and metadata will be outputted and saved as the compressed file, where the metadata indicates whether that byte is preceded by bytes that are already stored (Microsoft, 2020).

To decode the compressed stream using LZ77 compression, the decompression algorithms start with processing it from start to end. Whenever a null pointer is encountered, the algorithm appends the associated byte to the end of the output stream. For a non-null pointer, it reads back the specified offset from the end of the output stream and appends a specific number of bytes to the end of the output stream (Microsoft, 2020). The output stream will be the decoded stream and should be identical to the text pre-compression.

Similar to LZ77 compression algorithm, a similar algorithm is LZ78 compression. When comparing these two compression algorithms, one of the main differences they have is the fact that LZ77 algorithm works on past data or data that is present in the file to be compressed. However, the LZ78 compression algorithm attempts to work on future data, predicting the pattern of data that will occur. Moreover, LZ77 generally compresses the file faster than LZ78 algorithm (Choudhary et al., 2015).

Currently, the LZ77 compression algorithm is an open-source and is widely used in compressing files. One of the examples being compressing ZIP files, which is a very popular compression file that is used frequently. This is also used in multimedia formats such as PNG, TIFF and PDF files.

Section 2 – List of all data structures and algorithms used in implementation

Serval data structures have been used in implementing Huffman Coding. They play a critical role in creating the binary tree and the actual encoding of the text.

To start, for the compression function, a hash map is used in the beginning to store the frequency count of every character from the text, this is critical for the compression algorithm since characters with the highest frequency will be added to the tree last, resulting in a shorter binary representation for the character. This is done using the poll() method of the priority queue since it will retrieve and remove the head of the queue, obtaining the character with the least frequency. To implement Huffman Coding, a binary tree with nodes is used to determine the binary codes used to encode the text file, in which each leaf node corresponds to a character. The binary code for each character can be found by working down the tree from the root and add a 0 if you progress to the left child node, or 1 if you progress to the right child node until you have reached the leaf node representing the character. In addition to the binary tree, another hash map is used in the compression function to map the binary code to the specific character by traversing the Huffman tree. This hash map, together will the encoded string, will be stored in files for decoding the string afterwards.

As for decompression using Huffman Coding, a hash map, which is retrieved from the compressed file, is used to reference and acts as a 'dictionary' to translate binary codes back to characters. The translated character will then be appended to a StringBuilder object in a for loop. This process will be performed until all binary codes from the compressed file have been read and translated, the StringBuilder object will therefore contain the decoded string, which should be identical to the original file before compression if the compressed file has not been corrupted.

In addition to the data structures used during the compression/ decompression. A hash map is used to store the Huffman codes in a file which will be used for decoding the encoded string,. The encoded string, on the other hand, is stored in a separate file in binary format in order to reduce the size of the compressed file. These two files are linked together by their name.

Section 3 – Weekly log of progress

Week	Work Done
T2/W5 (w/c 8/2)	Watching the explanation video of the coursework on ELE. Doing research of the theory and terminology of Huffman Coding on the internet.
T2/W6 (w/c 15/2)	Understanding how Huffman Coding works, data structures that a Huffman Coding requires and planning on how to implement it. Deciding on whether to use Java or Python for the coursework.
T2/W7 (w/c 22/2)	Watching a number of explanation videos online to get a better understanding of Huffman Coding and the data structures. Started programming the compression part of the algorithm, implemented the Node class and created a Hash Map to store the frequencies of characters.
T2/W8 (w/c 1/3)	Continued to program the compression algorithm, implemented a priority queue which contains a comparator to compare the frequency of characters. Browsed examples of implementing methods (Priority Queue, Comparator, Hash Map) to get a practical understanding, fixed a bug of implementing priority queue afterwards. Implementing a method to create a file containing the encoded string and Huffman codes. Doing research of other lossless compression algorithms for the literature review.
T2/W9 (w/c 8/3)	Started implementing the decompression function, which translates the binary codes back to characters using the hash map stored in the file. Finalise the program, writing JavaDocs, comments and creating a scanner for user input. Doing benchmarks for different books/ datasets, recording the performance. Writing the PDF specification for the compression algorithms, including plotting charts to express the performance of compression on different files. Recording the 2 min video to show how the compression algorithm works.

Section 4 – Performance analysis

As the performance of using Huffman Coding to compress a text file varies with the distribution of characters, and also the size of the file. Some files tend to do better than other files, therefore it is essential to benchmark the performance of Huffman Coding when different text files are compressed or decompressed. To test the performance of Huffman Coding, serval text files have been tested and benchmarked. They include two books in three languages – English, French and Portuguese, the books selected for the test are Alice's Adventures in Wonderland and Oliver Twist. These books are downloaded from project Gutenberg. Four datasets have been downloaded from repetitive corpus dataset, which covers the category of artificial, pseudo-real and real.

The benchmarking of these files have been done using a test script, which runs the compression and decompression function for all the files to be tested. During the benchmark, the following data will be collected and be further analysed: original file name, original file size, compressed file size and decompressed file size, as well as recording the compress and decompress time. These tests have been done 3 times and an average value has been taken to ensure that the test is accurate.

To begin, the compression time and decompression time is recorded and figures 1 and 2 have been plotted to reflect these results. Figure 1 is a plot of compression time in millisecond over the size of the original file, a linear trend line has also been added to the plot and we have a R^2 value of 0.94, reflecting that the larger size the original file has, the longer it takes for Huffman Coding to be compressed. The decompression time over size of the original file plot (Figure 2) also has a similar result, suggesting that the larger size the original file has, the longer it takes for the compressed file to be decompressed. We can observe that a 40MB files take around 2000ms to compress and 3500ms to decompress.

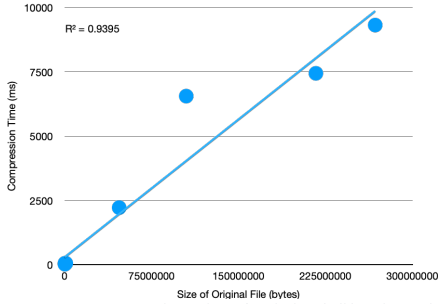

Figure 1 - Compression time/Original file size plot.

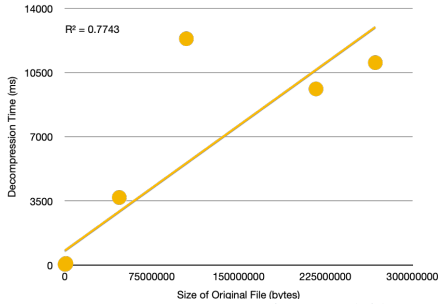

Figure 2 – Decompression time/Original file size plot

In addition to analysising the compression time of files, another way to analyse the efficiency of compression can be done by finding the compression ratio for those files by dividing the original file size with the size of compressed file. The higher the compression ratio, the more compressed the file is and as a result a higher efficiency is achieved. According to Figure 3, we can observe that larger files tend to have a higher compression ratio, meaning that larger files tend to do better in terms of level of compression.

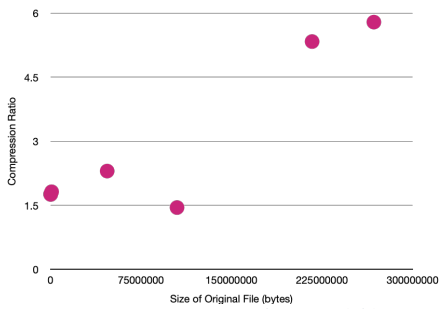

Figure 3 - Compression ratio/Original file size plot

The results above can be expressed in a theoreticalway. For instance, an original file of larger size implies that there are more characters to compress and there more iterations are required to go through the whole text file, this can also be a result that larger files take longer to write to storage than smaller file. Larger files tend to have a higher compression ratio since there are more data for the compressing algorithm to compress.

Note: size of compressed file/size of original file plot is not included here since it can be expressed in terms of compression ratio.

References

Choudhary, S. M., Patel, A. S., & Parmar, S. J. (2015, May). Study of LZ77 and LZ78 Data Compression Techniques. International Journal of Engineering Science and Innovative Technology (IJESIT), 4(3). https://www.ijesit.com/Volume%204/Issue%203/IJESIT201503_06.pdf

Gad, A. (2021, February 25). Lossless Data Compression Using Arithmetic Encoding in Python and Its Applications in Deep Learning. Neptune Blog. https://neptune.ai/blog/lossless-data-compression-using-arithmetic-encoding-in-python-and-its-applications-in-deep-learning

Glen, G. (1984). An Introduction to Arithmetic Coding. In J. R. Langdon (Ed.), IBM Journal of Research & Development (Vol. 28, pp. 135–149). IBM.

Howard, P. G., & Vinter, J. S. (1994). Arithmetic Coding for Data Compression. Proceedings of the IEEE, 82(6). http://www.ittc.ku.edu/~jsv/Papers/HoV94.arithmetic_codingOfficial.pdf

Huffman Coding Algorithm. (n.d.). Studytonight. Retrieved March 10, 2021, from https://www.studytonight.com/data-structures/huffman-coding

Microsoft. (2020, October 30). [MS-WUSP]: LZ77 Compression Algorithm. Microsoft Docs. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb

Stix, G. (1991). Profile: David A. Huffman. Scientific American, September, 54–58.
Huffman Coding Compression Algorithm. (n.d.). Techie Delight. Retrieved March 10, 2021, from https://www.techiedelight.com/huffman-coding/

Tsai, C.-J. (2014, September 10). Arithmetic Coding [Slides]. National Chiao Tung University. https://people.cs.nctu.edu.tw/~cjtsai/courses/imc/classnotes/imc14_04_Arithmetic_Codes.pdf

Wikipedia contributors. (2021, February 19). Huffman coding. Wikipedia. Retrieved March 10, 2021, from https://en.wikipedia.org/wiki/Huffman_coding

Wikipedia contributors. (2021, January 20). LZ77 and LZ78. Wikipedia. https://en.wikipedia.org/wiki/LZ77_and_LZ78

Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic Coding for Data Compression. Communications of the ACM, 30(6), 520–540. https://web.stanford.edu/class/ee398a/handouts/papers/WittenACM87ArithmCoding.pdf