**Data Structures and Algorithms**

**Literature review of compression algorithms**

The lossless data compression algorithm that I chose to implement is Huffman Coding. This algorithm analyses the text for the frequency that each character occurs, and stores this all in some sort of dictionary. This dictionary is then used to build a tree that is stored with the compressed data that is required to decompress the compressed (binary) data back into the original data. This compression algorithm can work very well for larger text files that are written in some sort of language (rather than entirely random letters), where there are some characters that occur more frequently than others, such as the vowels occur more frequently than the letter ‘z’ in the English language.

Huffman Coding works by building a binary tree, where the leaf nodes (nodes in a tree that have no children nodes themselves) are the characters, and they combine to form parent nodes. The lowest frequency characters are at the bottom of the tree and require more bits to store than characters that are closer to the root node in terms of nodes traversed that require less bits to store. Let us say that for an ASCII character it requires 8 bits (or 1 byte) to store. Huffman works by instead storing more frequently occurring characters with less than 8 bits, and characters that might occur much more rarely might take more than 8 bits. This tree is stored alongside the often-shorter binary data and is needed to decode the data.

This compression algorithm works can work very well for text file formats where there is a distribution of frequencies where some characters occur more than others. However, there is an upfront cost of size of storing the tree, so for small files, sometimes using Huffman Coding can increase the size of the compressed file, which is suboptimal. Therefore, it is fair to say that Huffman should only be used on files of a certain size where we can be sure that compression will be achieved, otherwise a dictionary-less compression algorithm like Lempel-Ziv-Welch might be more suitable.

One lossless data compression algorithm is called Lempel-Ziv-Welch (LZW). This uses a dictionary technique that relies on low entropy that is common in text file formats to achieve a good compression ratio. However, in the LZW compression algorithm, the dictionary is built incrementally, such that during the decoding process, the decoder can rebuild the original dictionary used to compress the file so that the original, decoded file can be recovered. This means that unlike a great many other compression algorithms, an advantage of the LZW algorithm is the ability to store the encoded data without needing to also store with it the encoding dictionary.

The way that the LZW algorithm works is by assigning the most commonly repeating series of characters with a code. For the sake of example, imagine our entire character list that we wanted to be able to store (read/write) consisted of only the English lower-case alphabet (a-z), and each is assigned a code (0-25), that is then stored in a byte. In a text file, we might encounter common sequences of characters more often based on the low entropy that is common in text file formats, so words such as ‘the’ or ‘it’ may appear more commonly. So, instead of storing ‘the’ as the individual values for ‘t’, ‘h’, and ‘e’, and taking 3 bytes, we can store the word ‘the’ as its own value in the dictionary, so ‘the’ would be stored with a value of 26.

However, there are two major considerations. The first is that when implementing such an algorithm, we must choose what dictionary to initialise with to encode. There are common text encoding formats that are used, such as ASCII. Often implementations will choose to assign an alphabet size of 256, (i.e., 0-255), so that it takes 1 byte. This means that the program would be unable to encode characters that do not appear in this alphabet (such as Chinese characters, for instance), but this is perfectly fine to encode English, numerical and common characters. The other consideration is that we need to build the dictionary incrementally so that during the decompression process, it can be decoded without needing the dictionary to be stored alongside the compressed file.

The LZW works by having a dictionary, D that maps Strings (such as a, b, aa, ab, bb, etc.) to Codes (such as 0, 1, 01, 011, etc.). To start with, we initialise an empty string that we can call S, and C, the current character that is being read in from some file. We also initialise D with all initial possible characters, in this example we will map a to 0, and b to 1, but in practice we would likely map the 256 ASCII characters to their corresponding numerical value. Then we will loop through each character in the file, and append C to the current S. We will then check if the dictionary does not contain S already, in which case we will map S to the next available code, so in our example 0 and 1 are both taken, so our next code is 2 (or in bit representation, 10). We then remove the last byte from S, and write the code representation of this updated S to our file as the encoded data, before finally setting S to C. Once the loop has concluded, we must remember to add the final dictionary entry to the encoded data, so if S is not empty, we write the code that S maps with to our encoded data file.

This has another problem though, in that currently, the description written above of a possible implementation of LZW would carry on defining codes infinitely, and the best compression of LZW comes when small segments repeat themselves, but if we allow longer segments of text to have a code, then this will provide diminishing returns, and could even create longer files, since we could have to deal with segments that rarely repeat that take up huge numbers of bits, especially for long files. So, we assign a fixed number of bits, such as 12 bits instead of the regular 8 bits to represent our data, and if the maximum possible 4096 codes are reached in the dictionary, then the dictionary is reset.

This explains the encoding process of the LZW algorithm, but what about the decoding process? As explained earlier, the LZW algorithm does not require the data to be stored alongside some dictionary or tree in order to decode the encoded data to recover the original data. This means LZW is a much better compression algorithm than Huffman Coding for example, since storing the tree can take a lot of data in of itself. In order to decode, we have our dictionary D that maps Codes to Strings in a similar fashion to the encoder, except in reverse. We also have S, a string which is initialised as empty. We also have our K value, which represents the Code, stored in binary form but that represents a number from 0 to 4097 (for a 12-bit implementation as discussed prior). We initialise our dictionary identically to our encoding dictionary, and then enter a loop, we read each code type from the encoded data file into K. We first check if K is greater than the dictionary length (i.e., K is 4096 or more for a 12-bit implementation), in which case we cannot decode as it is impossible to rebuild the original dictionary. If that is not the case, then we check if K is equal to the dictionary length, which is an edge case that we need to catch. In this case, we will map the next unused code to S + the first character of S. Otherwise, as long as S isn’t empty, we will map the next unused code to S + the first character of the string mapped to K in the dictionary. We will write the string that maps to K to our decoded data file, and set S equal to the string that maps to K.

Identical to the Huffman Coding compression algorithm that I implemented, LZW is also a greedy algorithm. This means that the goal of the algorithm is to find the best compression at any given stage of the compression process.

Another lossless data compression algorithm is called run-length encoding (RLE). This compression algorithm is incredibly simple and fast. The way it works is by assigning a number to repeated runs of characters, so for the sake of example, if you had a sequence of characters like this:

*AAAAAAAABBB*

It would be possible to shorten this to 8A3B, and storing that instead, so you are having to store less data.

However, in practice, this algorithm usually does not achieve very good compression ratios on text files, since in language, it is quite rare to see repeated characters, so this type of compression is not particularly applicable for that. However, for images, run length encoding can be extremely useful, since it is much more likely that if a pixel if a certain colour, the pixels next to it are also that colour. It can also be useful in video file formats, such as GIFs, since it is quite likely that if a pixel is a certain colour, in the next frame it will continue to be that colour.

Overall, it is fair to say that each of these compression algorithms have their uses on different file formats, for different file sizes, and it might be a good idea to run them simultaneously, or at least analytically to predict what compression algorithm is most likely to perform well with a given file.

**Data structures and algorithms used in my implementation**

*Map (HashMap)*

I used a HashMap which is an implementation of Map provided by the Java Collections Framework. Its use was to map characters to frequencies, and as the file was being read into memory, to update the frequencies that each character occurred. Since a “char” data type in Java can be up to 65535 inclusive (0-indexed), the maximum N number of Key, Value pairs that can exist is 65536. For HashMaps, there is no guarantee about ordering based on some sorting algorithm, so for this purpose I used a PriorityQueue, which I will discuss later. To access the value of some Key in the map, it takes O(1) time to get or put, which is useful as during the initial population of the map to determine frequencies, the map is frequently accessed to put updated data in. For example, a TreeMap might enforce some sorted ordering, but it takes O(log(n)) time to get or put items into the map, which would radically harm performance, since each character that we are reading in from the file could be anywhere in our map.

I tested the performance using both a TreeMap and a HashMap on several sample files, including three from the repetitive corpus dataset, as well as one book, and on average HashMap performed far better in terms of time taken to compress and decompress than TreeMap. Each file was sampled 10 times for each data structure (HashMap or TreeMap). This is why I decided to use HashMap data structure in my final implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Type** | **Compression Time (ms)** | **Decompression Time (ms)** |
| Fib41 | HashMap | 14419 | 5393 |
| Fib41 | TreeMap | 25184 | 6369 |
| World\_Leaders | HashMap | 3987 | 2368 |
| World\_Leaders | TreeMap | 6154 | 2549 |
| Sources.001.2 | HashMap | 13157 | 6582 |
| Sources.001.2 | TreeMap | 16214 | 8236 |
| Great-Gatsby | HashMap | 139 | 105 |
| Great-Gatsby | TreeMap | 145 | 110 |

*Queue (PriorityQueue)*

I used a PriorityQueue which is an implementation of Queue provided by the Java Collections Framework. PriorityQueues follow a First-In-First-Out algorithm, like all queues, but PriorityQueue follows it based on the priority heap, where items are ordered based on some comparison. I created Nodes that implemented Comparable<Node>, which means that when a Node is added to the PriorityQueue, the node with the lowest frequency is put at the top of the queue. I then use the poll() method to retrieve and remove this element from the queue and while there are two elements or more (a left and a right node), I create a parent node with these two nodes, which has its own frequency that is compared with the nodes on the queue when the parent node is added back to the queue. Once there is only one element left, I know this must be the root node. This is how I build my binary tree. Adding has a time complexity of O(N log(N)), since the input isn’t already sorted, and to poll the top element, there is a time complexity of O(1), since we are just fetching the element at the top of the list, so there is only 1 element to traverse.

*BinaryTree and tree traversal algorithm*

I created a Node object that essentially functioned like a binary tree, in that each node would store a left and right node (which can be null), each of which would in turn store their own left and right nodes (which can be null), etc. If the left and right nodes are null, then the Node can be considered a leaf object, in that it is the end of the tree, and does not store any more data. I constructed the tree by making the lowest frequency characters come first, and part of a new parent node which has a null character, and a frequency of the combined frequency of the previous two nodes, which is then combined with another character node to form a new parent node, etc. This process is repeated until there is exactly 1 node left in the queue, which is the parent node, and contains the whole (binary) Huffman Tree.

I then use a tree traversal algorithm when decompressing some text by starting with the root node, and iterating through each character in the encoded data, and if the current node was not a leaf (i.e., had a left AND right node), then I would get the bit (0 or 1 value) at that index. How Huffman works is that a 0 means we go to the node on the left (i.e., set the current node to the node’s left node), and a 1 means we go to the node on the right (i.e., set the current node to the node’s right node). Any value that is not either a 0 or 1 is invalid, and an error is thrown, since we are dealing with binary data. That works out how we traverse the tree, and we keep on traversing the tree until we get to our leaf node, at which point we will get the character that the node represents, and append it to our decoded string, before resetting the current node to the root node, and starting the process again. It is important to note however that whilst we do reset the node to the root node, we do not reset the index, so that we are continuing to decode the data.

**Progress log**

|  |  |
| --- | --- |
| **Week** | **Progress summary** |
| 18/01/2021 – 27/01/2021 | Worked on implementation of Lempel-Ziv-Welch (LZW) algorithm. |
| 28/01/2021 –  04/02/2021 | No progress – focusing on other assessments. |
| 05/02/2021 –  12/02/2021 | No progress – focusing on other assessments. |
| 13/02/2021 – 20/02/2021 | No progress – focusing on other assessments. |
| 21/02/2021 – 28/02/2021 | Started working on implementation of Huffman Coding. Able to get a working implementation of both encoding and decoding that would take a String and return a Huffman object that contained String and Huffman Tree to decode. |
| 01/03/2021 – 08/03/2021 | Started writeup of literature review. Wrote about LZW from my initial attempt at implementing LZW algorithm. Researched other compression algorithms for my literature review, including Run Length Encoding. |
| 09/03/2021 – 16/03/2021 | Finished writeup of literature review. Started and finished working on reading in raw data from file to compress, storing compressed data (including tree, binary data, and information about padding), rebuilding Huffman object from this data, and decoding and storing back as a decoded text file. Started performance analysis section of writeup. |
| 17/03/2021 –  18/03/2021 | Finished performance analysis section of writeup and began and finished data structures and algorithms section of writeup. Finalised and submitted document and source code for assessment. Recorded video demonstrating functionality of application. |

**Performance analysis**

*Two real books in English:*

For the first book, I used ‘The Great Gatsby’, which was 300 KB decompressed. After compression, the file size was 173 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 57.7% of the original file size. Compression took 143ms, and decompression took 139ms.

For the second book, I used ‘Romeo and Juliet’, which was 166 KB decompressed. After compression, the file size was 104 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 62.7% of the original file size. Compression took 116ms, and decompression took 91ms.

*Two real books in French:*

For the first book, I used ‘Germaine’, which was 426 KB decompressed. After compression, the file size was 244 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 57.3% of the original file size. Compression took 135ms, and decompression took 115ms.

For the second book, I used ‘Tolla’, which was 432 KB decompressed. After compression, the file size was 246 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 57.0% of the original file size. Compression took 187ms, and decompression took 166ms.

*Two real books in Portuguese:*

For the first book, I used ‘Lupe’, which was 147 KB decompressed. After compression, the file size was 87 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 59.2% of the original file size. Compression took 98ms, and decompression took 83ms.

For the second book, I used ‘Cintra’, which was 28 KB decompressed. After compression, the file size was 21 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 75.0% of the original file size. Compression took 87ms, and decompression took 81ms.

*Repetitive corpus dataset (Artificial):*

For this, I used the ‘fib41’ file, which was 261636 KB decompressed. After compression, the file size was 32707 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 12.5% of the original file size. Compression took 14419ms, and decompression took 5393ms.

*Repetitive corpus dataset (Pseudo-real):*

For this, I used the ‘sources.001.2’ file, which was 102400 KB decompressed. After compression, the file size was 70856 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 69.2% of the original file size.Compression took 13157ms, and decompression took 6582ms.

*Repetitive corpus dataset (Real):*

For this, I used the ‘world\_leaders’ file, which was 45868 KB decompressed. After compression, the file size was 19946 KB, including the Huffman tree and padding information. This meant that compression reduced the file size to approximately 43.5% of the original file size. Compression took 3987ms, and decompression took 2368ms.

**Performance claims**

Compression as a percentage (%) of original file size


**References**

*Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW)*. Web.mit.edu. (2011). Retrieved 18 March 2021, from <http://web.mit.edu/6.02/www/f2011/handouts/3.pdf>.

*LZW file compressor - C++ Articles*. Cplusplus.com. (2013). Retrieved 18 January 2021, from <http://www.cplusplus.com/articles/iL18T05o/#LZWAlgo>.

Saikia, A. (2019). *LZW (Lempel–Ziv–Welch) Compression technique - GeeksforGeeks*. GeeksforGeeks. Retrieved 18 March 2021, from <https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/>.

Singh, K. (2021). *Huffman Coding | Greedy Algo-3 - GeeksforGeeks*. GeeksforGeeks. Retrieved 21 Feburary 2021, from <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>.

*Run-length encoding*. En.wikipedia.org. (2021). Retrieved 18 March 2021, from <https://en.wikipedia.org/wiki/Run-length_encoding>.