Specification Document for ECM1414 Coursework

Part 1 - Literature review of compression algorithms:

Data compression is a process in which the amount of space needed to store data is reduced by removing excessive, non-essential information. The goal of this is to be able to represent a digital file with the fewest bits possible. When a file has been compressed, a version similar to the original should be able to be obtained. There are two types of data compression - lossy and lossless.

Lossy data compression algorithms irreversibly remove parts of data and only an approximation of the original file can be reconstructed. The best files for lossy compression are image, video and audio files as people will often not detect the differences between the original and the compressed version. For example, MP3 files use lossy data compression in which some of the higher frequency data, not audible by humans, is discarded (the perceived frequency range of the human ear is 20 Hz to 20 kHz - MP3 files don't contain frequencies above 18 kHz).

Where lossy compression permanently removes data from the file, lossless compression differs by not losing any data in the compression process. Instead, data is encoded in a way that takes up less space but also means that the original file can be restored when decompressed. If it is not possible to exactly reconstruct the original data from the compressed version, then the compression is not lossless.

For this project, I need to implement a compression algorithm to encode a text file. I need to use a lossless compression algorithm to restore the original data from a compressed file. In the context of a text file, a lossy algorithm would result in losing several letters and words, causing complete disorganisation. This is why lossless compression is the only option.

There are many lossless compression algorithms to choose from. I will be going over the following algorithms and discussing what they do, how they work, and the advantages and disadvantages of each with the given task:

- Run-length Encoding (RLE)
- Lempel Ziv Algorithms (LZ77/LZ78)
- Huffman Encoding

Run-length Encoding:

Run Length encoding is one of the most basic lossless compression algorithms. It stores multiples of the same value as a single data value and a count. For example, the string "AAAAABBCAAAA" would become '5A2B1C4A'. Run-length encoding is most effective when dealing with lots of repetitive data - such as simple, low colour-depth images. With few colours to choose from, there will be lots of repeated pixels meaning an image can easily be compressed into taking up much less storage space.

However, when it comes to text files, RLE may not be the best choice, this is because in the worst case (where every character is unrepeated) the size of the compressed file will be twice the size of the input file. "ABC" for example, becomes "1A1B1C". You could further encode the output with another

compression algorithm, such as Huffman Encoding (which is discussed later), to mitigate this issue. However, this may be less effective than just running Huffman Encoding on the original text file.

Lempel Ziv Algorithms:

The Lempel Ziv Algorithms are a set of lossless data compression algorithms originating from Jacob Ziv and Abraham Lempel's 1977 and 1978 papers. There are two main algorithms, LZ77 and LZ78 which have spawned numerous other similar algorithms derived from themselves.

LZ77 works with repeated words and phrases in a text file. When there is a repetition, a pointer to an earlier occurrence is added to the file alongside the number of characters matched. This dramatically reduces the file size as multiple bytes of data can be represented as a 2-byte pointer.

The pointers are denoted in the form <j, l> in which 'j' represents the relative jump of how many characters back a word or phrase was and 'l' represents the length of the word or phrase. For example, the 'wood' of 'woodchuck' in the phrase "how much wood could a woodchuck chuck" could be replaced by the pointer <13, 4>, and the word 'chuck' could be replaced by <6, 5> and so on. The pointer is comprised of 16 bits. In most cases, programmers allocate more bits to the jump than the length. A split of 12 bits:4 bits means that you can point to a word up to 16 characters long from 4096 characters away.

LZ78 is a compression algorithm that features a dictionary that is used for both encoding and decoding. The algorithm creates a code of two elements (i, c) where 'i' is an index referring to the longest matching dictionary entry and 'c' is the first non-matching symbol. When a symbol is not yet found in the dictionary, the 'i' value equals 0 and it is added to the dictionary as well. Over time the dictionary grows until every value in the text file has been encoded. The LZ78 algorithm works well when dealing with repeating patterns, but if there is not a clear pattern then the dictionary may grow to an incredibly large size as there is nothing bounding the size of the dictionary.

Given the string "ababcbababaa" LZ78 would iterate through the string, outputting: (0,a)1, (0,b)2, (1,b)3, (0,c)4, (2,a)5, (5,b)6, (1,a)7. To decompress, you take the value at the index and add the symbol. So at 1, you add index 0 (nothing) to 'a', at 2 you add index 0 to 'b', at 3 you add index 1 ('a') to 'b' (at this point you have "abab") and so on until you reach the last code (add index 1 ('a') to 'a') and get your output string "ababcbababaa".

Huffman Encoding:

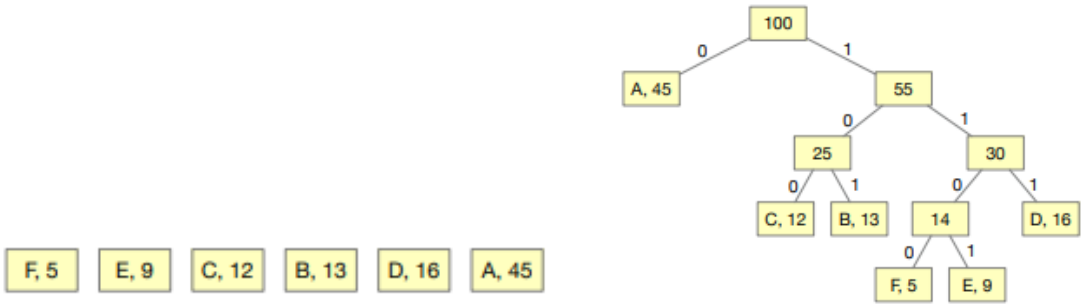
The Huffman Encoding algorithm was developed in the 1950s by MIT student David Huffman and is a very successful method for text compression.

In a regular text file, each letter is represented by a string of binary digits, its size depending on the character set used (for example, if Extended ASCII is being used, then each character takes up 8 bits). Now, Huffman's idea was to replace these long fixed-length codes with shorter variable-length codes, in which the shortest codes would be assigned to the most frequently occurring symbols. This would result in the size of the data decreasing.

These codes are uniquely decipherable, removing the need for a separator to show where one code starts and another code ends. To create these codes, Huffman Encoding uses a binary tree generated by the following steps:

1. For each distinct character create a node with its frequency stored in that node, store them in a list.
2. Select two free nodes with the lowest weight from the list.
3. Create a parent node for these two nodes with its weight is equal to the sum of the two child nodes.
4. Remove the two child nodes from the list and add the parent node to the list of free nodes.
5. if until only a single tree remains:
   done
   else:
   goto step 2

Once the Huffman tree is been created, the algorithm creates a code for each character by traversing the binary tree from the root to the character's node. It assigns '0' for a left branch and '1' for a right branch.



I will be using Huffman Encoding for this project due to its high levels of simplicity but also decent results when compressing. A dictionary containing the Huffman codes for all the characters in a text file will most likely be smaller than the dictionary of an LZ78 algorithm for the same file. In addition, Huffman Encoding is often much faster than RLE and LZ77 as you do not need to spot patterns in the text file in order to encode. You simply generate your Huffman codes and then replace each byte with your given code.

Part 2 - Implementation:

In my implementation, I used 2 different types of data structure, the Binary Tree and the Hash Table.

My Binary Trees have been implemented using a class called 'NodeTree' which I created. This allows for the creation of the Huffman tree, needed for the Huffman Encoding algorithm. I have also used numerous Hash Tables in my implementation as I have used many Python dictionary abstract data types which are secretly Hash Tables in disguise. These python dictionaries have been used for the character frequency nodes and storing the Huffman codes (.hufftree files).

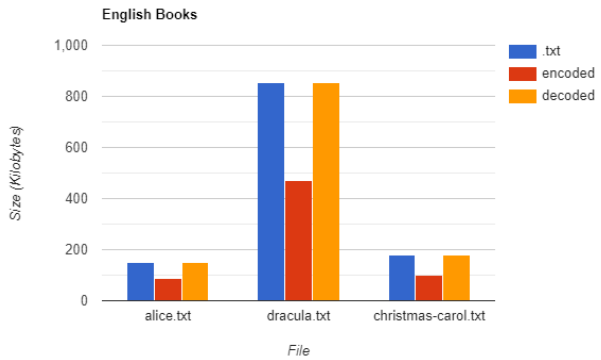
The only main algorithm I have implemented is Huffman Encoding, however, I have needed to code an algorithm for tree traversal in order to generate the Huffman codes from the Huffman binary tree.

Part 3 - Weekly progress log:

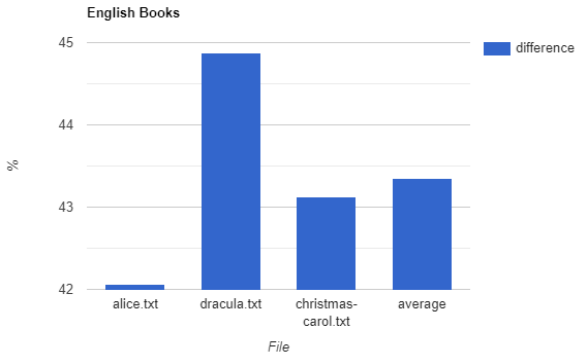
Week/Date:	What Got Done:
w/b 8th February 2021	Researched Huffman Encoding and found other algorithms to look at.
w/b 22nd February 2021	Created functions ‘open_file()’ and ‘get_character_freq()’. These open a .txt file and calculate the number of occurrences of each character in the text.
w/b 1st March 2021	Created the Huffman Tree using a class ‘NodeTree’. Created everything needed to encode/compress the .txt file.
8th March 2021	Implemented everything needed to read and decompress the .bin file. Implemented saving the Huffman Codes separately as .hufftree files.
9th March 2021	Added User Interface to allow for functionality in the program.
10th March 2021	Did research for Part 1 of the .pdf document. Added feature to use different Huffman Trees to encode for Part 4 of the .pdf.
11th March 2021	Creation of the README file. Write-up of the .pdf document.

Part 4 - Performance analysis:

My program is effective as it creates a compressed file that is smaller than the original. My program works best when dealing with files larger than ~4KB as the .hufftree file it creates is approximately 2KB and so any .txt file smaller will be ineffective to compress. Below is a bar chart to represent how my program handled three different books in English. As you can see, the .bin and .hufftree files together are ~60% the size of the uncompressed file. When the file is subsequently uncompressed, it returns to the same size as before. I have tested my program with at least 3 books from 3 different languages; English, French, and German.

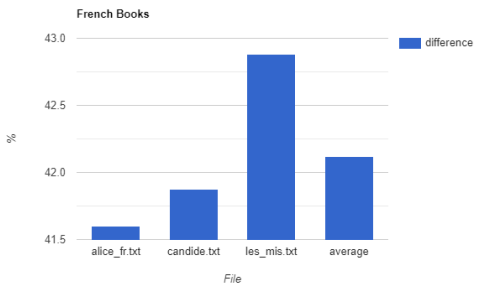


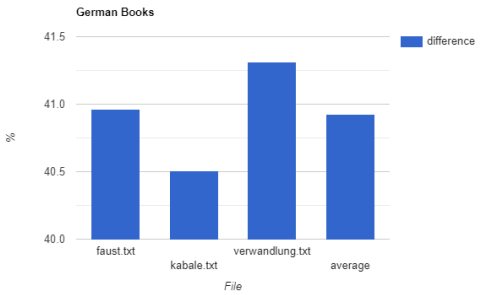
The difference as a percentage of the original file is as follows:



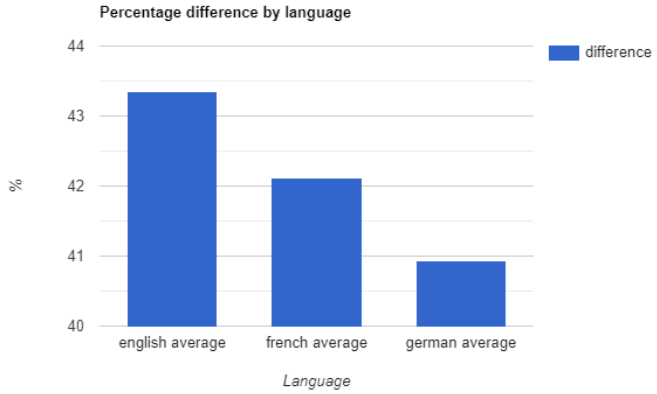
Lewis Carroll - Alice's Adventures In Wonderland = 42.063%
Bram Stoker - Dracula = 44.869%
Charles Dickens - A Christmas Carol = 43.122%
Average = 43.352%

I generated bar graphs to show the same information for the French and German books.



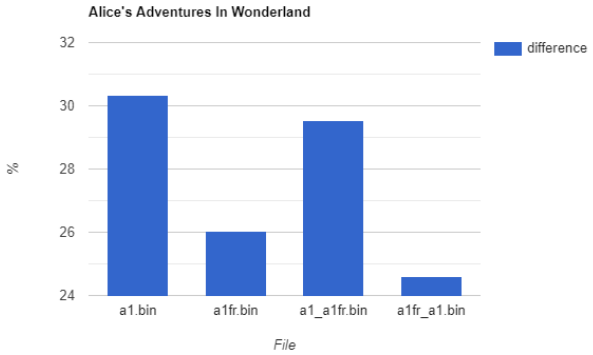


You can compare the averages. The higher the percentage, the better the compression.

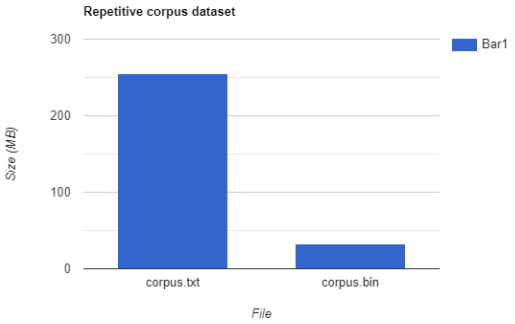


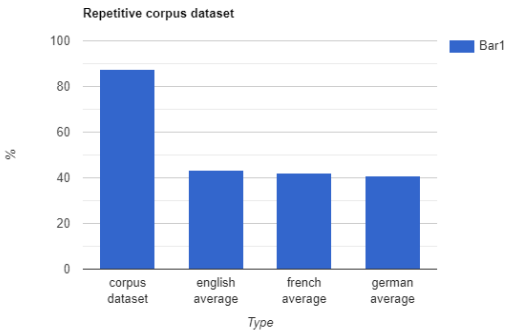
As you can see, English books have the best compression. This is because they do not contain as many special characters. Languages such as French and German contain special characters such as letters with accents and so the Huffman tree is bigger resulting in longer Huffman codes.

I took the first chapter of Alice's Adventures In Wonderland in both English and French and compared the compression of English, French, English encoded with French, and French encoded with English. These were the results.



I also used the program to compress a dataset file from the repetitive corpus dataset. The file was 200MB which is much larger than any file I had compressed previously. This took my program about 10 minutes to compress but dramatically reduced the size of the file by over 223MB.





Part 5 - References:

D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in Proceedings of the IRE, vol. 40, no. 9, 1098-1101, Sep 1952,

D. Salomon, "Data compression: the complete reference," Springer Science & Business Media, Feb 2004,

S. Shanmugasundaram, & R. Lourdusamy, "A comparative study of text compression algorithms," International Journal of Wisdom Based Computing, 1(3), 68-76. Dec 2011,

T. H. Cormen, C. E. Leiserson, R. L. Rivest, & C. Stein, "Introduction to algorithms," MIT Press, Jul 2009,

J. Nugent, "WAV or MP3: What's the Difference?" audiobuzz.com, Mar 2019, accessed 10/03/2021 - (https://www.audiobuzz.com/blog/wav-or-mp3-whats-the-difference/#:~:text=Lossy%20Compression,-As%20I%20touched&text=The%20perceived%20frequency%20range%20that,cuts%20off%20at%20around%2018KHz)

Computerphile, "Elegant Compression in Text (The LZ 77 Method) - Computerphile," youtube.com, Nov 2013, accessed 11/03/2021 - (https://www.youtube.com/watch?v=goOa3DGezUA)

D. Budhrani, "How Data Compression Works: Exploring LZ78," medium.com, Jan 2020, accessed 11/03/2021 - (https://medium.com/swlh/how-data-compression-works-exploring-lz78-e97e539138)