The task was to complete two programs, one program to compress text files using Huffman Encoding and the other to decompress binary files using the symbol model generated in the Huffman Encoding program. Both programs have been successfully created with fully working methods for both 'Char' and 'Word' symbol types.

huff-compress.py:

Before generation of the Huffman Codes and subsequent compression of the text file, the inputted file must first be analysed to determine the probability of each of symbol within the file. This 'Tokenisation' is carried out by the *Tokeniser* class, where the text file is iterated through line by line and depending on the specified symbol type:

- If char is selected, then each character in the line is iterated through the character is added to the symbol model and symbol counts updated.
- If word is selected then the regex expression *"[\w]+|[^\s\w]+|[\n\r\s]"* is used to separate the line into the required symbols, these symbols are then added to the symbol model dictionary.

Once the text has been fully analysed, the final symbol model is created by dividing the count for each individual symbol by the total number of symbols in the text, giving probability.

The Huffman tree can now be generated using the *HuffmanTree* class. A seed tree is created by converting each symbol in the symbol model to a *Node* class instance. To build the Huffman tree the nodes are sorted in descending order of node probability. The final two least probable nodes are removed and combined into a new node. This new node is then added back into the tree, this continues until the tree array has one node remaining, the root node with 1.00 probability.

The Huffman codes for each symbol can now be attained by traversing the tree. This is implemented by the g*etHuffmanCodes* method where each node is checked whether a left or right branch from the node is present. If it is, then travel down the branch logging the branch path taken as a binary 0 for a left branch and binary 1 for right branch. Then call another instance of *getHuffmanCodes,* to check the new node for branches. If there are no branches, the node is a leaf node and contains a symbol, hence the symbol is assigned the binary path taken to reach the leaf.

Finally, the text file can be compressed by iterating through each symbol in the file and appending the respective Huffman code of the symbol to a binary array. Both the binary array and symbol model dictionary are saved to .bin and .pkl files.

huff-decompress.py

The symbol model is retrieved from the .pkl file and each byte within the .bin file is iterated through converting each byte to its bitwise value. Any blank bits are filled with 0's so that each byte is represented by 8 bits IE 0x08 = 00001000. These bits are then added to the compressed bit string for decompression.

The bit string is iterated through with each bit appended to a separate binary section string. During each iteration, the binary section string is checked whether it exists in the symbol model, if it does then add the corresponding symbol to the text string and reset the binary section string. Else continue adding bits from the compressed text string to the binary section and check against the symbol model, repeat the process until a symbol is found. After each bit in the compressed string has been iterated through and converted back to symbols the text has been decompressed back to the original .txt file.

Evaluation of Program Performance

Computer Specs: Intel Core i7-8700 CPU @ 3.19Ghz. 15.8GB usable installed RAM. Windows 10

Symbol	Compressed Text File Size (bytes)	Symbol Model Size (bytes)	Time to Build Model (seconds)	Time to Encode File (seconds)	Time to Decode File (seconds)
Char	690,359	1,402	0.18	1.34	1.62
Word	390,192	599,332	13.65	0.89	0.99

The original text file was 1,220,149 bytes, compressing using char symbols has achieved a text compression ratio of 0.57 compared to a ratio of 0.32 for word symbols. But if the symbol model is included into the ratio, word results in a far worse compression ratio of 0.81 compared to char which remains at a ratio of 0.57.

Char is over 50 times faster to generate the symbol model than word. This is because the number of symbols in the text file is positively correlated to time taken to build the symbol model. There are far fewer unique characters than words in a text file hence, the smaller symbol model build time for chars. Despite this, word has faster encoding and decoding times, this is because more of the text file is encoded/decoded in each iteration as words contains multiple characters. Each of these characters would have to be encoded individually in char mode taking more time. Nevertheless, the poor compression ratio when the symbol model is included combined with the overall run time shows that word symbol compression performs far worse than char. There would be no reason to select this type of compression unless you wanted fast decompression times or the symbol model was generated once and used for multiple compressions/decompressions.

However, the way the symbol model is created in this assignment is highly inefficient, the model for word contains thousands of nodes that simple contain pointers to other child nodes which computationally wasteful and traversing the tree to find the Huffman codes takes a lot time. Therefore, if a Canonical Huffman code system was used it is possible that the time required to generate the symbol model and size of the model for word based would be dramatically reduced and hence the performance enhanced.

Canonical Huffman does not require a code tree; this alone would vastly improve performance for both symbol methods. Furthermore, Canonical does not require the transmission of the entire symbol codes dictionary, just the number of symbols at each code length is enough to reconstruct the entire code. This would improve the compression ratio for Word mode if the size of the transmitted symbol model is included; because the average length of the Huffman codes symbols and compressed text file size doesn’t vary between Canonical and standard Huffman methods but the size of the Symbol Model does massively.

In conclusion if a Canonical system was implemented the Word symbol type would far out-perform char, the combined size of the compressed file and symbol model would be smaller than char. The time to build the word symbol model would be slightly slower than char due to the number of unique word symbols but this is negated by faster encoding and decoding times. The Word model would therefore give a greater compression ratio with potentially smaller run times over the full compression/decompression cycle than char.