Text Compression Report

1. Brief description of the implementation

- **huff-compress.py**

  In this program, it can be divided into three parts: load input file, build Huffman tree, and encode the file. In the first section, I load the input file line by line and calculate the frequencies of every character or word at the same time, in the end, I also add extra symbol, '|', to be a pseudo-EOF. After counting the all frequencies, their probabilities can be derived and be put on correct positions in the node list in ascending order. In the next section, I build the Huffman tree through adding the two smallest probabilities as a new node, removing the old two, and putting the new one on correct positions of the original list. According to this tree, I write a recursive function, called traversal, to trace all the nodes in order to find out the Huffman codes for the all symbols. In the last section, I transfer the whole text to the Huffman code and even combine them because I have to convert each 8 characters into a one-byte binary representation. If the rest characters are not filled in a byte, I add extra '1' characters to finish off the last one. Finally, the result of encoding can be exported a file, and the Huffman tree also must be stored in another file for decoding using pickle package.

- **huff-decompress.py**

  In this program, I load the two input files 'infile.bin' and 'infile-symbol-model.pkl' firstly. The infile.bin file can be converted back to the characters which only consist of '0' and '1', and the other can reconstruct the Huffman tree which is quite important to make the program recognise the character length of each symbol. After that, I trace the all characters which means that a variable 'current node', which is used to point out the current node, can be assigned to its left child node if the character is '0', and be assigned to its right child node if the character is '1'. When the symbol of pseudo-EOF is found, tracing character is terminated and the rest characters should be ignored. Finally, the result of decoded file can be exported and it must be similar to the original file.

2. Comparison of the compression performance of character-based vs. word-based

How big (in bytes) are the different files?

	Compressed text	Symbol model
Character-based	690,363	6,157
Word-based	392,564	1,720,133

How long (second) do different steps take to?

	Build symbol model	Encode the input file	Decode the compressed file
Character-based	1.39	1.63	3.31
Word-based	1.75	1.06	1.95

3. How could the performance be improved?

Based on my observation, there are two aspects taking the program much time: loading the input file and encoding the whole text, which take between one and two seconds. However, the other sections, such as computing the probabilities and sorting, building the Huffman tree and traversal, all just take nearly zero second.

It can be seen that the program has to check whether each character or word has ever appeared or not when it reads a new line, and then it determines to create a new dictionary or just add one time. Unfortunately, I think this loading process is essential and may be hard to improve its performance though another approaches. In addition, it is quite time-consuming to convert each 8 characters into a one-byte binary representation because it needs to do so many times for whole text. Similarly, I also think that is the best way to be addressed about its optimization.

On the other hand, for the whole program, I have improved the search approach through the 'Binary Search' which can divide and conquer those large-scale elements in list, compared with general way to search all of them. Especially in word-based Huffman, the execution time with general way takes approximately 60 seconds and that with binary search just takes about 3 seconds.

4. Conclusion

In conclusion, it is obvious that the difference between character-based and word-based is their Huffman tree. Indeed, word-based Huffman tree has more leaf nodes and more level (deeper) than character-based. That causes the program to take plenty of time to search all the nodes if I do not use binary search. Furthermore, dividing a word which is combined only with alphabets and other character which is regarded as a word using the regular expression is also different between them.