Manuel Worlali Ziwu - 180128251

COM 6115: Text Processing

Assignment 2: Text Compression

Implementation

To obtain arguments from the command-line, I employed the use of two default python modules: **sys** and **argparse**. All files were checked for their existence before they were opened for reading, and an error was thrown if the file didn't exist to give the user a more descriptive message and prevent the program from "crashing".

Code Implementation

huff-compress.py

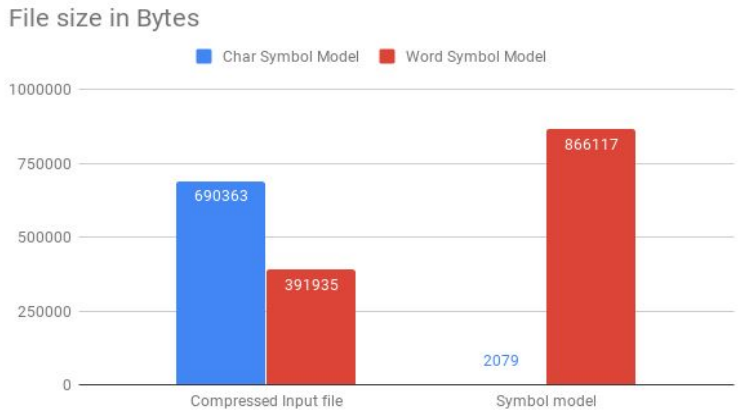
- A class called **HuffmanNode** was created with which its instances will serve as nodes in the Huffman Tree. It is a simple class with a constructor defining five attributes which are: **symbol, left_tree, right_tree, probability, binaryCode**.
- A method, **assignBinaryCode** which took as arguments, **node**(a node in the Huffman Tree) and **code**(binary code at that level of the node). This method was used to traverse the completed Huffman Code tree and to assign binary codes to symbols in the tree.
- The symbol model type(char/word) was checked to confirm what regex(char - **[\w\W]** and word - **[\w]+|[\W]**) will be used for obtaining the symbols to be used for building the Huffman Code Tree.
- EOF character was defined and I began going through the input file line by line, using the regex to obtain the symbols while keeping track of their frequencies in a dictionary.
- Using dictionary comprehension, the relative frequency of each symbol in the input file was computed and stored in a dictionary as before.
- A list of nodes was created using list comprehension. Instances of the class, **HuffmanNode** were initialized with symbols and their probability values to begin with. After which, the list was sorted using the Python's **sorted** function and using the node's probability as the **key** or criteria for sorting.
- A **while** loop is used to go through the list and **break** on condition that the list has less than 2 members. The two lowest nodes were **popped** from the list and a new node was created using the sum of their probabilities and assigning one of each of the nodes as the **left_tree** and **right_tree** of this new node(parent node). The parent node is then **appended** to the list and a resorting is done to the list again.
- Once the loop breaks, the list contains one node which will be the complete Huffman tree. The **assignBinaryCode** method is then used to assign binary codes to the symbols that are stored in a dictionaries, **symbol_model** and **dictionary_model**.
- The **symbol_model** is then **pickled** using Python's **pickle** library to be saved and used for decompressing the input file later.
- The input file is then encoded using the **dictionary_model** along with the assigned **pseudo-EOF** and the new encodings are stored in an array that stores binary data as a C-Type unsigned integer; after which the **binary_array** stores the new binary data to a file.

huff-decompress.py

- The program accepts as input the compressed binary file and obtains the **symbol_model** from the saved pickle data in the current working directory.
- The binary file is read and its data is obtained and converted back to a string of binary values.
- The obtained binary string is iteratively read and a comparison is made against the encodings in the **symbol_model** dictionary that contains the mappings between the binary code and their symbols.
- This process is repeated until the **pseudo-EOF** is reached and the complete decompressed file is stored in a new file.

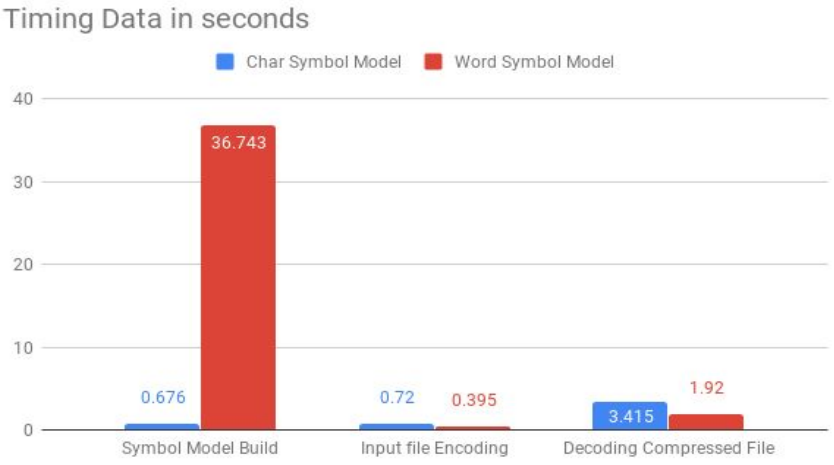
Result Analysis

The program was run on a Manjaro Arch Linux computer with 8GB of RAM and 16GB of Swap space and 110GB of Hard disk space. It was running a number of various processes as such the results shown below were affected by all these factors.



Compressed file size - We can observe that after compressing the input file, we obtain a file with smaller size using the word-based Huffman coding as compared to that of the character-based Huffman coding. This can be attributed to the fact that, many of the words in our input file occur multiple times. As such, the probabilities of such values in file increases, reducing the size of the Huffman codes for the words significantly. However, although the same can be said about character-based Huffman code, each character being replaced by its Huffman code could be problematic for uncommon characters that appear once or twice in the input file. Perhaps even taking more space than its original character byte.

Symbol model size - There is clearly a vast difference between the sizes of both symbol models. The character-based symbol model is extremely small as it holds all the unique characters of the input file along with their Huffman codes. These characters consists of mainly the lower and upper case alphabets, numbers, punctuations and the different types of space characters(i.e. new line, tab, etc.) which are comparatively not as much as the number of unique words in the input file. This causes huge increase in the size of the word-based symbol model as the number of unique words increases.



Symbol model creation time - It is evident that it takes a lot more time to create the word-based symbol model than that of the char-based and this is due to the large size of the word-based symbol model as compared to the char-based symbol model. The algorithm was written to resort after a new node had been appended to the list reducing performance with an especially large list of nodes.

Input file encoding time - In encoding the file, it takes more time for the character-based symbol model to perform encoding because it has to encode more tokens than the word-based symbol model at once.

Decoding compressed file time - In a similar fashion to the input file encoding, the decoding of a character-based compression takes almost twice as much time as the word-based model due to the fact that the compressed file for the character-based symbol model is larger and for every character, a decoding of its value has to be done unlike the word-based model where decoding of a huffman code results in a word of usually more than one character.

Conclusion

Based on the graphs displayed, it will be highly desired to reduce the time it takes to build the word-based symbol model. This can be achieved by using a binary insertion sort method instead of the regular python **sorted** method that returns a new list after sorting. This will make the algorithm faster and as such increase its performance. It will also be better to perform a combination of both the word-based and character-based symbol models which could greatly reduce the size of the symbol model. This can be done by creating Huffman codes of symbols that occur close to each other. For example, the symbol ‘q’ is usually found to be followed by a symbol such as ‘u’, hence creating a Huffman code of ‘qu’ may be better than creating one of just ‘q’. A combined size of the symbol model and compressed input file for both the word-based and character-based models are **1258052B** and **692442B** respectively as compared to the original input file of size **1220150B**. We can therefore say that the character-based model provides an overall better compression to the input file, since transmission of the compressed file and the symbol model is more cost-effective than the word-based model and raw input file comparatively.