Ethan Raymond

CS250 Extra Credit

Assignment:

In this assignment, you should program up 3 variants of Huffman codes,  
test them on varying forms of inputs, and write up the conclusions of your  
work.

In variant 1, you use straight Huffman coding, counting the number of times  
each symbol appears and creating a Huffman code based on this.

In variant 2, you use Huffman coding on pairs of symbols rather than individual  
symbols.

Variant 3 deals with the possibility that symbol frequency varies over time;  
for example, if you are sending text which may be books in different  
languages, the letter frequency changes when you switch from an English  
book to a German book. Therefore, you (either periodically or when you  
realize the frequency is off) divide the message into segments, and  
create optimal code for each segment.

Remember that there is a cost for these variations; you must send as well  
as the encoded message the encodings you are using. This is longer as  
you go from single to double letters, and longer if you divide the  
message into segments since the code must be sent for each segment.

Summary:

I wrote a program capable of encoding multiple variations of Huffman codes, including of codes varying symbol size (calculating the frequency of pairs, trios, etc.) of characters, and different codes for different parts of the input.

To run the program I typed “python3 huffmancoding1.py” into the terminal. The program is not compatible with python 2.7.

Program

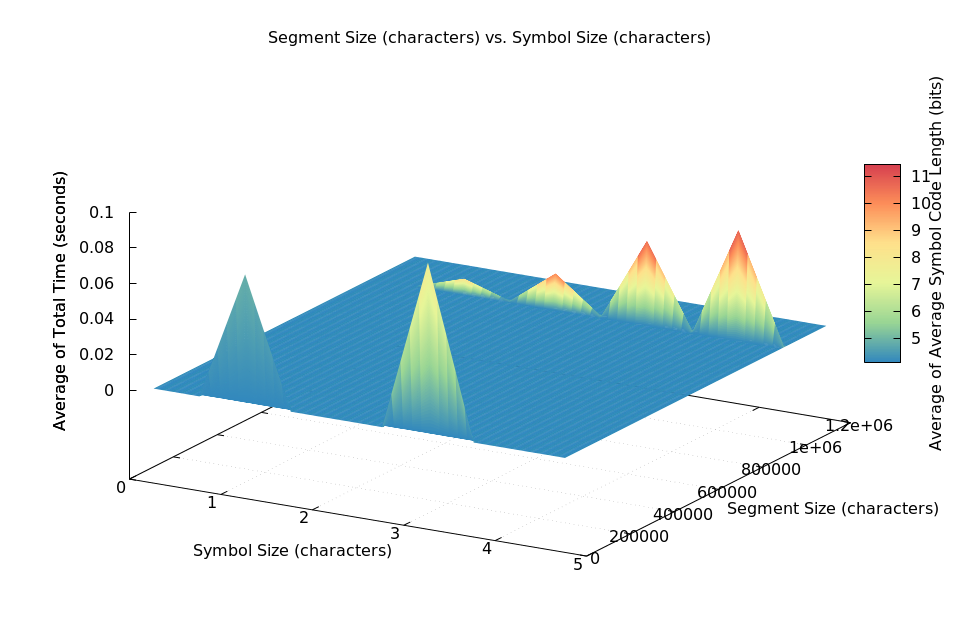
I defined a Node class to represent the nodes of the trees required to create Huffman codes. The program reads in a file as input, divides the file into different segments if required, and calculates the frequency of symbols based on the user’s choice of symbol length. I used a dictionary to store the frequency of each symbol. The program then creates a tree using a priority queue. I used a priority queue because the data structure automatically sorts the elements based on their frequency in ascending order. The program removes the two lowest frequency elements from the queue, creates a new node whose frequency is the sum of the two elements’ frequency, and inserts the new node into the queue. Once the queue has a size of 1, the only node in the queue is the parent node for a tree whose leaves are the symbols in the segment of the input. The code is created by adding a 0 or a 1 to each node’s parent node’s coded value. Since all symbols are leaves of the tree, no symbol’s code is a prefix for another’s. The program could be made more secure by randomly assigning a binary number to the right child, and the complement of that number to the left child. The message is then encoded by incrementing through the input and adding each symbol’s binary representation to an output string.

Input Files

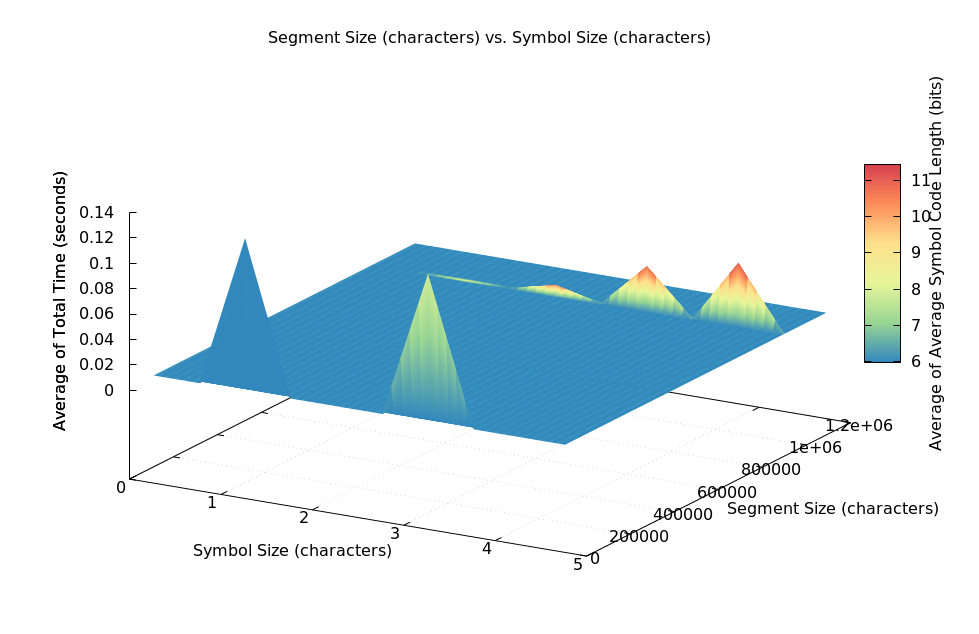
The program is tested with four different files. Test.txt is used to make sure the program outputs the simplest codes for a small input and was used to check the correctness of the program. The larger files were used to test the varying Huffman code versions’ performance on different inputs. Englishpaper.txt is part of a research paper I wrote this year. I used this as a control because the frequencies of characters remains relatively constant throughout the paper and there are no explicit patterns in the text. Thunderroad.txt contains three Bruce Springsteen songs with certain verses translated into other languages such as French, Spanish, or Bosnian. This file was used to test an input with varying symbol frequencies. Finally, limitedsymbols.txt only used 4 symbols and therefore had a lot of repeated characters. This file was used to test the Huffman code with pair characters’ performance on a file with many repeated characters.

Results

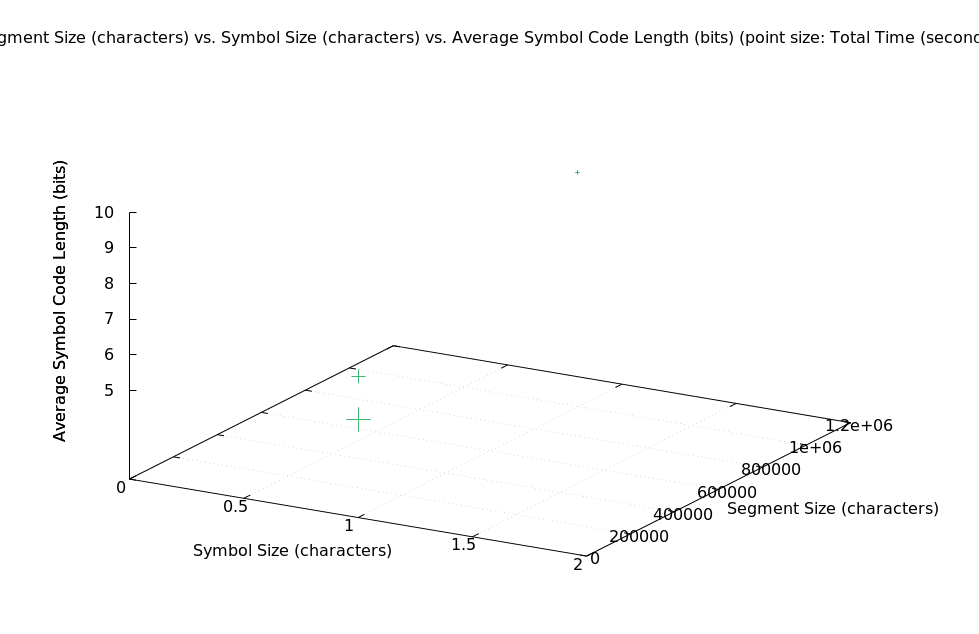
I recorded the size of the input, number of differently encoded symbols, average symbol code length, size of the encoded message, time to create the Huffman code, time to encode the message, and total time required. I graphed the results of the averages of all three inputs. From the graph I learned that time to create the Huffman code increases with symbol size and decreases with segment size. Average symbol code length positively correlates with the amount of time needed to create the code and symbol size. The time to create the code increases with symbol size because as the number of characters of symbols increases, the number of different symbols increases. Therefore, the tree used to create the code is larger and takes longer to build and traverse. As segment size increases, less trees are created for the different Huffman codes so the program takes less time to execute. Symbol code length increases as segment size increases because more symbols are included in the segment and longer codes are required to represent each symbol.



I also graphed each input file individually. This is the graph of the englishpaper.txt file. This graph shows that symbol size can decrease the time required to run the algorithm and larger segment sizes greatly reduce the time required to encode message but also tend to have larger average symbol code sizes.



The thunderroad.txt data graph shows that as segment size increases the average size of a symbol’s code increases, but the time required to run the Huffman code algorithm decreases (size of the point corresponds to the time required to run the algorithm). Smaller segment sizes results in smaller code lengths for symbols partly because the Huffman trees are more optimal for each segment and more frequently used characters are given smaller codes.



This is the graph of the limitedsymbols.txt file. It shows that the algorithm ran the fastest with a symbol size of two, so increasing symbol size can make the algorithm more efficient for inputs with patterns. A symbol size of two is the fastest because characters were frequently repeated in the input so increasing the symbol size by 1 reduced the size the number of bits required to represent an individual symbol without significantly increasing the number of different symbols.