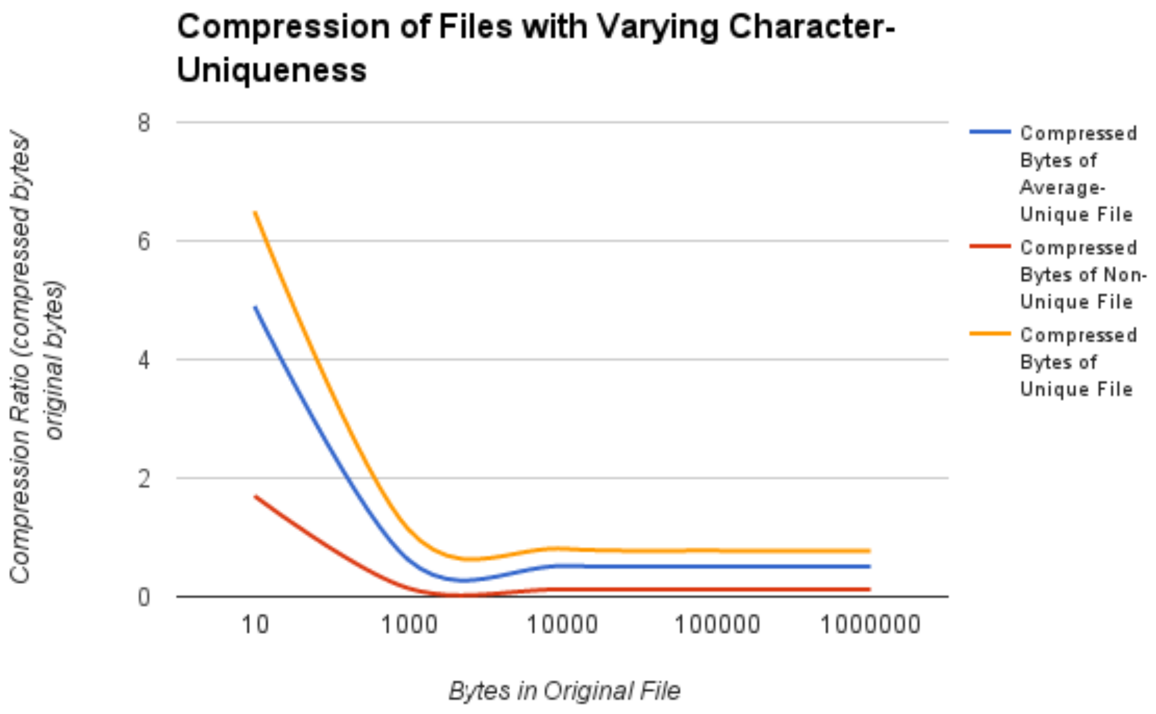
1. In order to determine how the compression ratio is affected by the number of unique characters in the original file and the frequency of the characters, I made 15 different files. 5 of these files were completely non-unique, all of their information was the same character (red line in my graph). 5 other files were averagely-unique, and by this I mean that I took a random character generator from the internet and generated 100 characters, then copy and pasted those characters until my file had the desired byte size (blue line in my graph). The remaining 5 files are "unique" because they were comprised of every possible character, copy and pasted over and over (yellow line in my graph).

   As you can see, all 3 file types perform better as the size of the file increases. The non-unique file was always the best, followed by averagely-unique, then unique. Which is what I expected. If there is only one character repeated over and over, then the Huffman tree that the program has to store is very simple. And an "average" amount of characters should take more space since it will have to write more characters to the tree, but not as much space as the "unique" files since they have all possible characters and therefore have to write the most information. The frequency of the characters is not as important as the number of unique characters. This is because the program doesn't write each character to the compressed file, it actually only writes it once and then records its frequency only once. So having a high frequency will increase the overall bytes of your original file, thus increasing the overall bytes of your compressed file by roughly the same amount.



2. In order for Huffman's algorithm to be effective, you want to have a compression ratio that is less than 1. That way you are actually making you file smaller. According to my

experiments, as long as your text file is around 1000 bytes or longer, then you will be compressing your file length by running my program on it.

3. Huffman's algorithm repeatedly merges the two smallest weight trees because we want to have the characters with the highest frequency at the top of the tree. If we merged the two trees with the highest weights every time, this would push the nodes with high frequency down the tree, which is the opposite of what we want. So when we merge the two smallest trees from our priority queue every time, the last tree's we merge will be the nodes with the highest frequency, thus leaving them as close to the root as possible.
4. Huffman's algorithm performs lossless compression. Absolutely no information is lost from the compression. Or in other words, when you decompress the compressed file, it will be exactly the same as your original file.
5. I spent a total of 3 hours on this assignment.