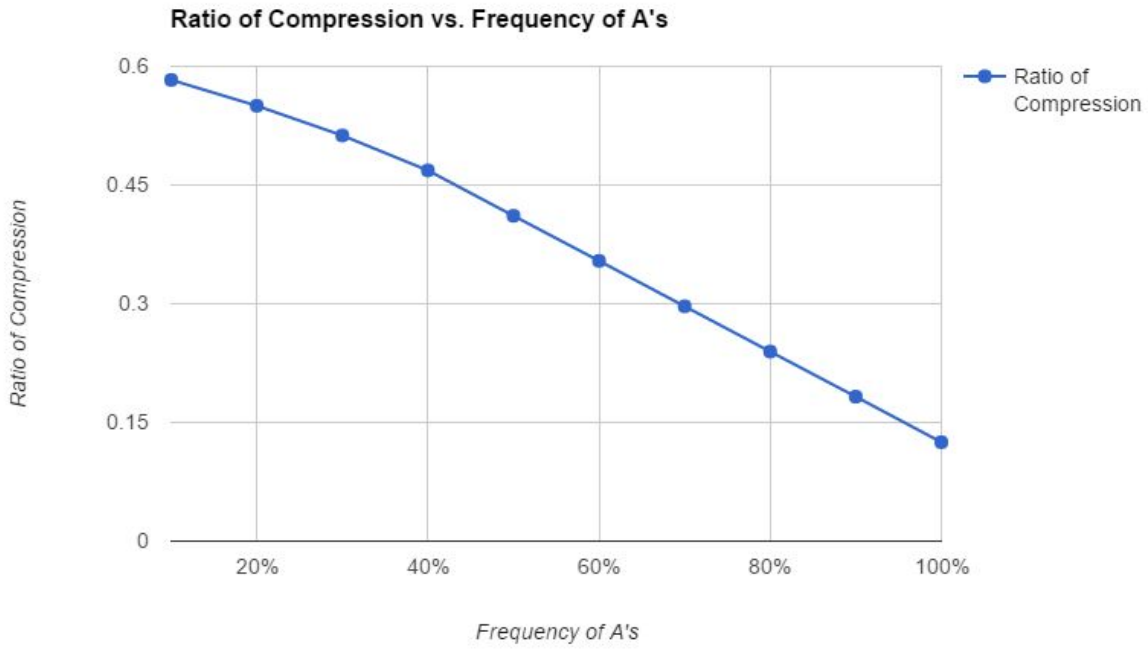
Osama Kergaye
CS 2420
11/21/2016
Dr. Meyer

Assignment 12 Analysis Document

1. Design and conduct an experiment to evaluate the effectiveness of Huffman's algorithm. How is the compression ratio (compressed size / uncompressed size) affected by the number of unique characters in the original file and the frequency of the characters? Carefully describe your experiment, so that anyone reading this document could replicate your results. Submit any code required to conduct your experiment with the rest of your program and make sure that the code is well-commented.Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plots(s), as well as, your interpretation of the plots is critical.

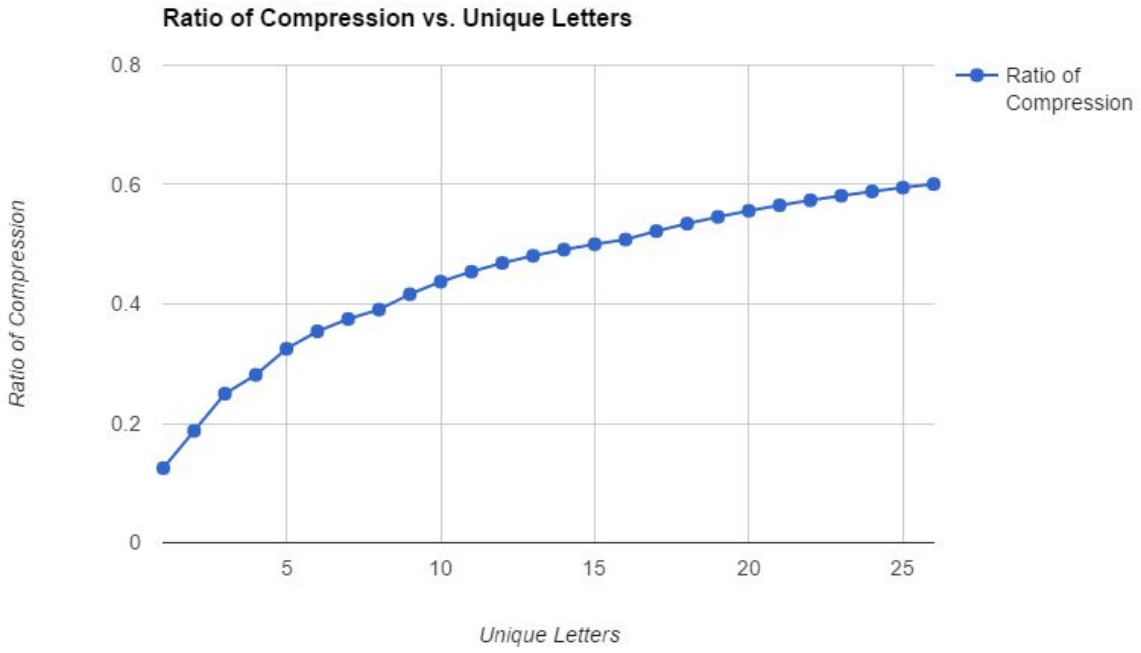
To conduct my first experiment examining the ratio of compression vs the frequency of a letter in a file to be compressed, I had to set up a few things. First I had to modify the timing template that was given to us by Ryan. I changed the exponent for loop to only run once and create an int of 2^15th (I will call this N). This number is then used to create a string that will be written to a file that always has 327680 KB in size before compression. I then changed the iter loop to only run 10 times instead of 100. Inside that loop, I created a file called “zHuffmanText.txt” this is where the string will be written.

So with that out of the way, this is where it gets interesting. I create a StringBuilder and append the letter A, N/ (10/11-iter) times. This means it will start with the string being filled with 100% A’s then 90% then 80% and so forth, the remainder is filled with random letters of the alphabet. Then I write that to the file, compress it and compute my ratio and write to my data file. For each time iter loops, I get data for a new ratio with less and less percent A frequencies.



Above is the plot I got after graphing my data. As you can clearly see, as the frequency of the same letter (letter A) is increased in the file, Huffman's algorithm gets a much better compression ratio. This is because it doesn't have to make a deeper binary trie and therefore it doesn't have to increase the number of bits needed to encode a letter.

For my second experiment testing the ratio of compression vs the number of unique characters in a file, I did pretty much the same setup with a few differences. First one being, that I changed iter to go from 1 to 27, to account for the 26 letters of the English alphabet. Then in my for loop that builds my string, I had it the number of different letters that could be added depend on the iter count. First, it adds the letter A only, then the second time A and B, then A, B and C, and so forth until the last loop allowed the entire alphabet to be added in the string. Then I wrote the string to file, and conducted the same ratio test.



As seen above, as the number of unique characters that could fill my file went up, so did the compression ratio. This was because the Huffman algorithm had less of the same letters and was therefore forced to create a deeper binary trie. Thus for each unique character that was added, the bit code had to get longer, therefore the compression ratio became larger :(.

2. For what input files will using Huffman's algorithm result in a significantly reduced number of bits in the compressed file? For what input files can you expect little or no savings?

For very large files, the algorithm will significantly reduce the number of bits in the compressed file. So, the larger the file the greater the savings. In very small files, the algorithm will provide little to no savings. So, the smaller the file the smaller the savings.

This is because the header accounts for some space, and if the file is only a sentence, with few repeated charters, the header could very well be bigger in bits then the actual file. But if the file is huge, then there will be a bunch of repeated letters and that means the header will be smaller in bits then the file to be compressed. Thus, giving the header more value for the space it uses.

3. Why does Huffman's algorithm repeatedly merge the two smallest-weight trees, rather than the two largest-weight trees?

Huffman's algorithm merges the two smallest weight trees to keep uncommon letters at the bottom of the tree, which is paramount to the functionality of the algorithm. If the two largest trees were merged, then the most common letters would be at the bottom, and there new bit encoding could very well be larger than the original letter. Therefore, possibly creating a larger "compressed" file.

4. Does Huffman's algorithm perform lossless or lossy data compression? Explain your answer. (A quick google search can define the difference between lossless and lossy compression).

Huffman's algorithm performs lossless compression because it can decompress the file to exactly the way it was before, without any changes.

5. How many hours did you spend on this assignment?

I spent about 20 hours coding the assignment I think, but i'm really garbage at keeping track of how much time I spent.