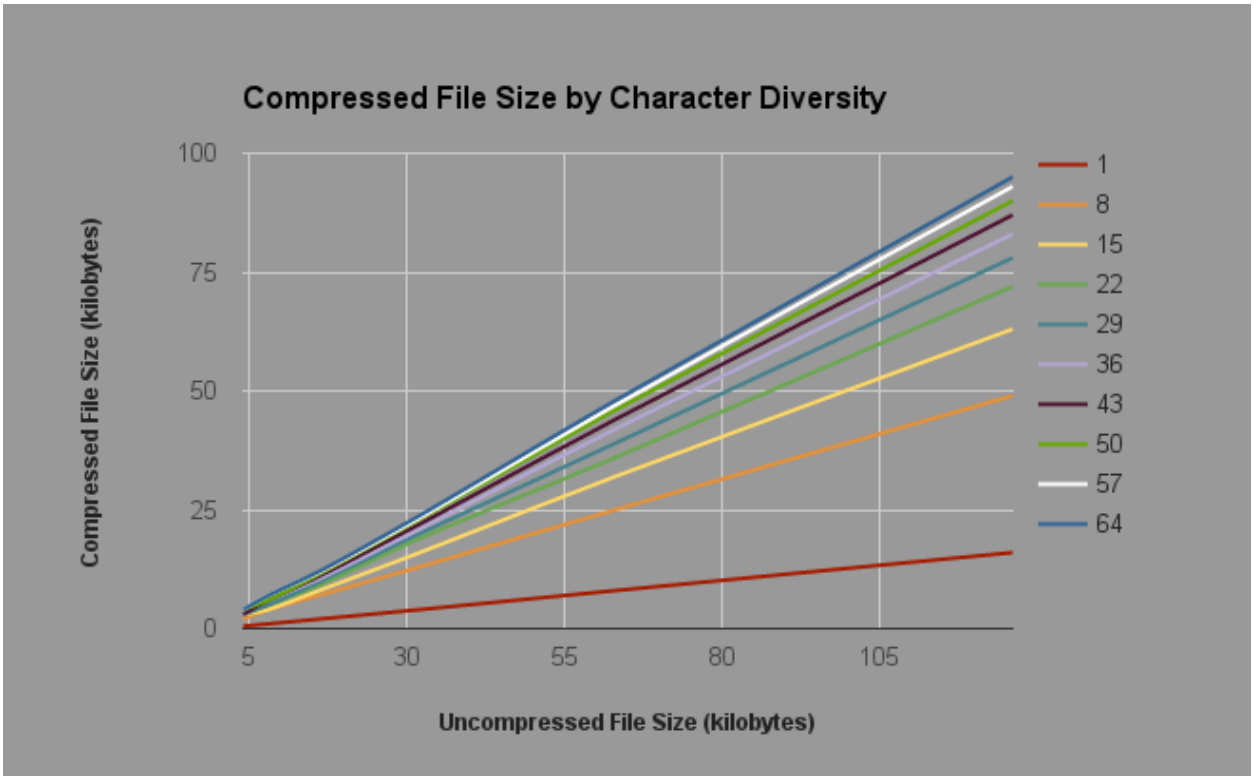
Elliot Carr-Lee
CS 2420
Fall 2016
Meyer

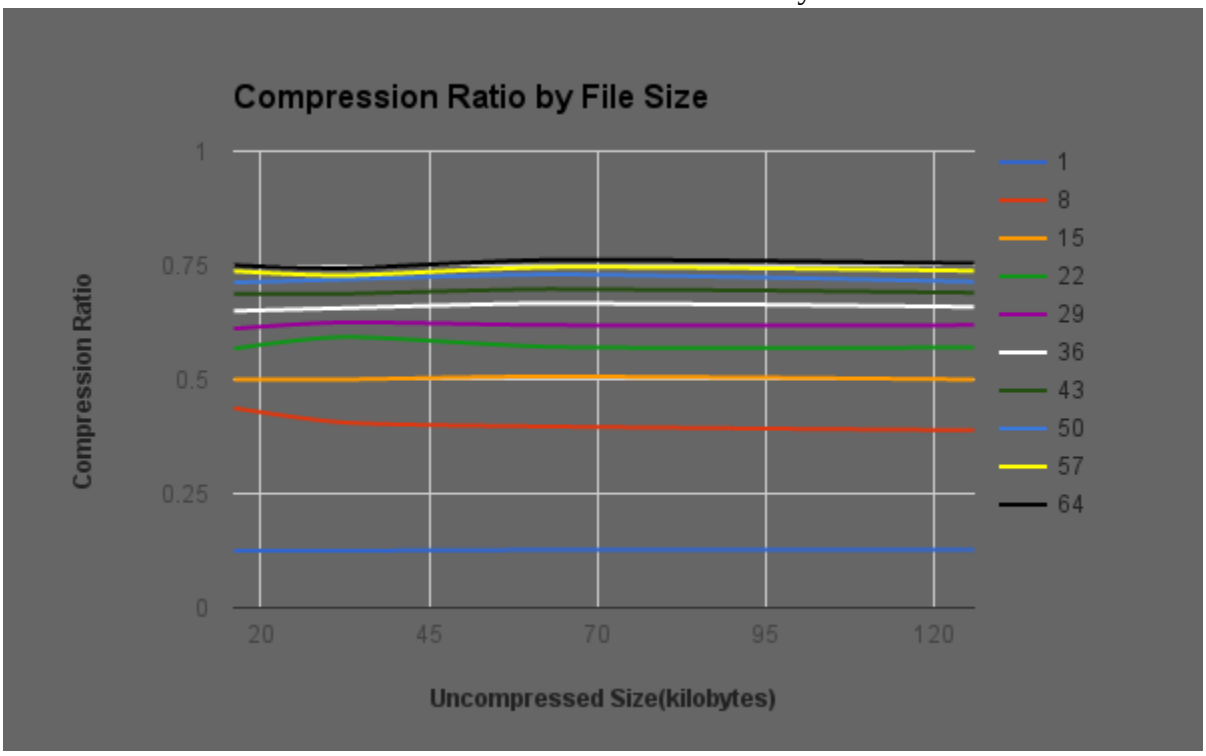
Analysis Assignment 12

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.



Each of the lines on this graph represent a different degree of character diversity: 1 indicates that the entire file consists of a single character, while 64 indicates that the file contains 64 different characters. **All characters that a file contains appear with equal frequency**: in the 64 case, they each occur $1/64^{th}$ of the time. This was achieved by creating a char array containing all eligible characters and looping across it until the total file size goal was achieved. Ascii character values in the range 48-112 were used.

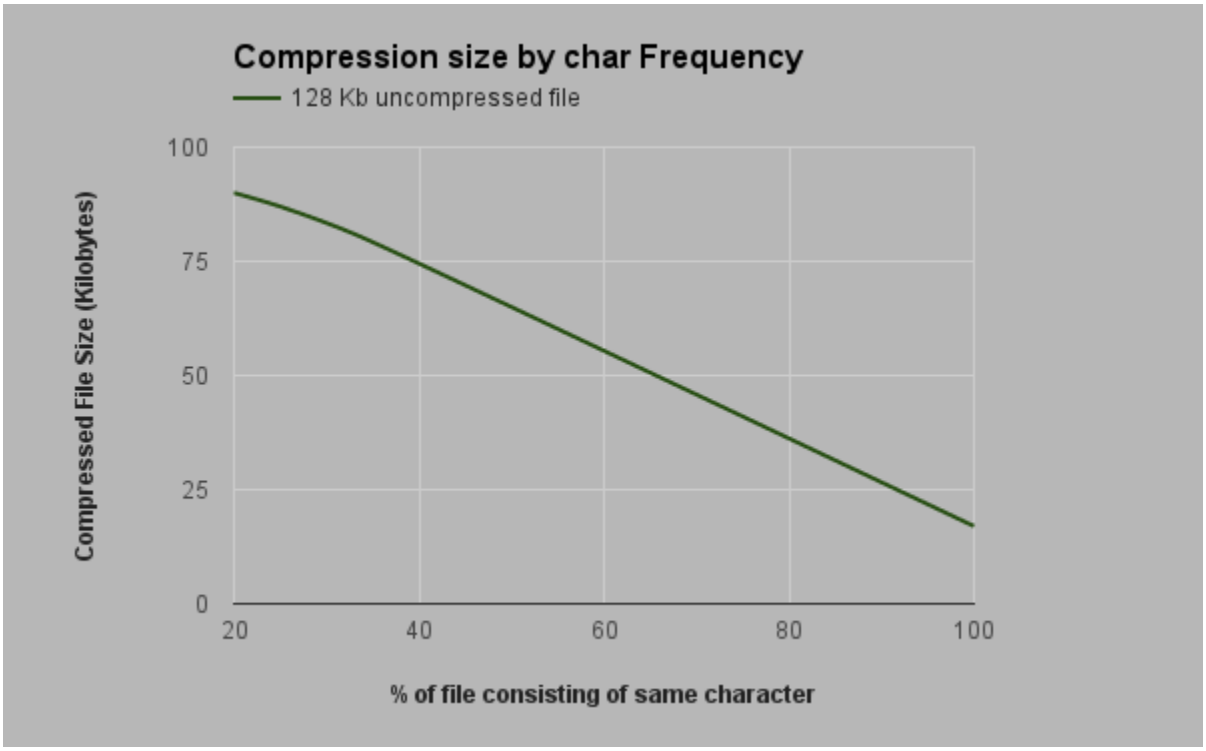
The following chart displays the same data, except that I divided the compressed file size by the uncompressed file size, to generate a compression ratio which replaces the compressed file size on the y-axis. I cut off some of the smaller data points for this graph as noise in file size (many files ended up being the same number of kilobytes because the files were so small) made the trend less clear. **The overall trend is that files with low character diversity compress well, while files with higher character diversity compress less well.** This is because when the character diversity is low all of the nodes in the binary Tri are near the top, allowing characters to be encoding in 1 bit instead of 8 bits. Indeed, the compression ratio for the file with exactly one type of character is almost exact .125, or $1/8^{th}$. When there are 64 characters in the file, it will take 6 bits to store most characters because 6 bits in binary can



Store 32+16+8+4+2+1 = 63 items. Once we account for the end of file symbol, there are a total of 65 items in our tri, which suggests that 2 of our items will be encoded with 7 bits instead of six. Indeed, generating a graphviz graph of our 64 character compressed file shows this is exactly what happened (its way too big to fit an image here unfortunately). **The fact that it takes 6 bits to store characters, instead of the standard 8 bits, suggests a compression ratio of 6/8 = .75. That is the exactly the compression ratio we actually see for the black line, a file with 64 different characters, regardless of file size.**

In the previous experiments I dealt with files that had equal number of each character. **If the character values are highly unequal(100k 'A', 1k 'B') then we should expect files with high character diversity to behave as if they had low character diversity. The more characters that are of one type, the better compression you will get, even if there are many types of characters in the file.**

To prove this, I created various files with 128,000 characters, and a total of 64 different characters in each file. But one of the characters had disporportionate representation:



The character in question ranged from being 20% of all characters in the file, to consisting of all characters in the file except for 100 characters at the end that were of the normal distribution of 64 elements. As we can see, **the compression size decreases as the frequency of a single character increases, even though there is high character diversity. This shows us that it is not merely the number of characters that are present in the file that matters, but rather how the overall frequency of characters is distributed.**

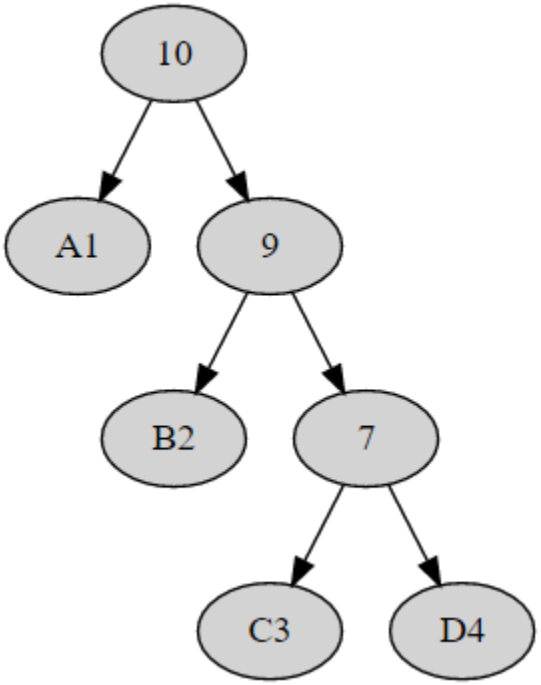
2. For what input files will using Huffman's algorithm result in a significantly reduced number of bits in the compressed file? For what input files can you expect little or no savings?

If there are a relatively small number of different characters in the file, you can expect a great degree of savings. This is because each char, uncompressed, takes 1 byte (8 bits) of memory. This is proven by the fact that an uncompressed file with 128,000 characters take 128 kilobytes (128,000) of memory. But a Huffman tree, for a file with only one character, can store a character in 1 bit ($1/8^{th}$ of a byte). Thus, you can, at best, achieve a savings of 87.5%, as demonstrated by my graphs under question 1. On the other hand**, if there are a large number of characters, occurring with relatively equal frequency, there will be little benefit.** This is because as the number of characters in the file grows, the size of the Huffman tree grows. Since the characters occur with equal frequency, they will all end up in the bottom row of the tree, and take more bits to encode.

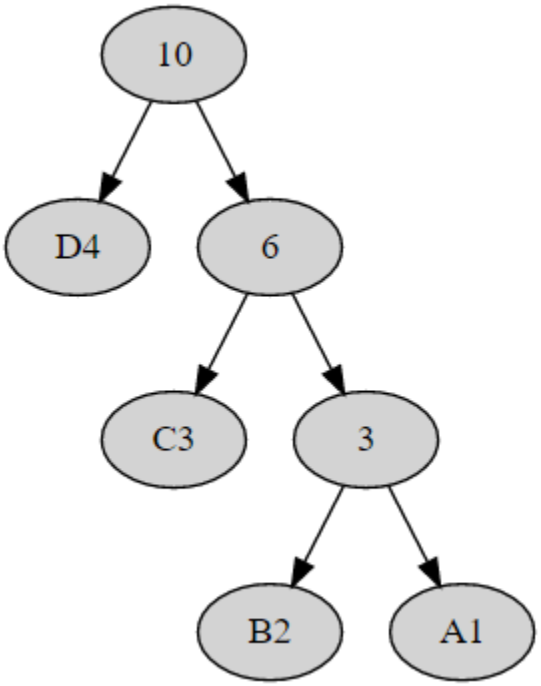
If there are many characters in the file, but a few of those characters are repeated a lot while the others occur rarely, there will still be substantial gains. This is because you will have an unbalanced tree with the commonly occurring characters near the top, taking less bits to encode, while the rare characters take more to encode but occur less often.

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

If it merged the two largest-weight trees you would get trees where the items that occur the least in the document would have the shortest encodings, and the items that occur the most have the longest. This would decrease the ability of Huffman compression to decrease the size of a file. Take a tree with 4 items, with frequencies of A(1) B(2), C(3) D(4). If we merged the largest trees, you'd get a root of 7 with C3 and D4 as children, and then a root of 9 with B2 and 7 as children, and finally a root of 10 with 9 and A1 as children:



This means encoding C and D, the two most frequently occurring characters would take 3 bits, while encoding the least commonly occurring character, A, takes only a single bit. Conversely, if we construct the tri by merging the smallest trees first we would get the following tree:



This tree encodes the most common item, D4, with only 1 bit, the second most common item, C, with 2 bits, etc. The least frequent items are at the bottom of the tree taking up the most bits. **Thus, constructing the tri by merging the smallest weight trees results in a better compression than merging the largest weight trees.**

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).

It performs lossless data compression: it returns, after compression, the compressed file exactly as it was before compression. This is verified by my test code which shows that post compression files are identical to those which were originally compressed. Thinking about how the algorithm works, it uses the binary tri to encode enough information to recover all the characters in the original file. No characters are lost, it's just that it uses information about character frequency in the file to record the exact characters in the file in a shorter format.

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

Approximately 10.