Dylan Peters
Computer Science 201
Prof. Azhar
21 April 2016

Huffman Assignment Analysis

1. Based on the code I wrote, the compression rate, as a function of file length L and alphabet size A, is O(log(A)). The compression rate is independent of the file length because the compression rate is based solely on how often each of the 256 possible characters occurs. If some characters occur very often, and others are very rare, the alphabet will be small, and the tree will also be small, thus allowing you to represent the characters with smaller sequences. If the alphabet is full (all 256 characters represented with approximately equal frequency), the tree will be very large, and the compression rate will be lower. The reason it is log(A) is because the characters are stored in a tree, so doubling the number of characters only increases the length of the encoded strings by 1.

   The data that I got when I benchmarked the code backed this up, showing a compression rate of O(log(A)). This data can be seen in a table below. Of course, the data is not perfect, and does have a few inconsistencies, but in general, the larger alphabet sizes do result in lower percent savings. One of the reasons this data is difficult to determine the relationship from is because of the O(log(A)) savings. Because the relationship is not linear, and the domain ranges only from 0 to 256, the output does not have a large range (barring the 79.19% compression achieved from compressing "pic.")

File Name	Alphabet Size	Original File Length	Compressed File Length	Percent Saved	Runtime (ms)
bib	81	111261	72880	34.5	53
book1	82	768771	438495	42.96	77
book2	96	610856	368440	39.68	39
geo	256	102400	72917	28.79	7
news	98	377109	246536	34.62	25
obj1	256	21504	16411	23.68	2
obj2	256	246814	194456	21.21	16
paper1	95	53161	33475	37.03	3
paper2	91	82199	47748	41.91	4
paper3	84	46526	27398	41.11	3
paper6	93	38105	24158	36.6	3
pic	159	513216	106778	79.19	20
progc	92	39611	26048	34.24	4
progl	87	71646	43110	39.83	7
progp	89	49379	30344	38.55	5
trans	99	93695	65361	30.24	7

Based on the code I wrote, the runtime of the code is O(L+log(A)). The reason for this runtime is that the code runs through every bit of the input file to count the frequencies, which takes O(L) time. The code then turns the frequencies into a tree, which takes log(A) time because of the tree structure. The code then traverses the tree, which takes log(A) time, in order to turn the tree into a map for faster access. Finally, the code runs through every 8 bits of the input file and uses the map (which has constant access time), to convert to the compressed file. This last operation takes O(L) time. Adding up all these operations results in O(L+log(A)) theoretical runtime.

The data that I got when I benchmarked the code backed this up, showing a compression rate of O(L+log(A)). This data can be seen in the table above. As you can see, for longer files, the runtime is nearly linearly longer. Furthermore, for files with larger alphabets, the runtime is slightly longer than for others. One problem with this method of collecting data is that the files are so small that the runtimes are all under 100 milliseconds. A more precise way of measuring runtime (in microseconds, for example), and longer files with more variance in alphabet size would likely result in more precise results for the runtime as a function of file length and alphabet size.

2. Text files compress much more than binary files. For example, the total compression for the calgary folder, which was all text, was 43.76% space saved. For the waterloo folder, which was all binary files, the compression was 20.97% space saved. The reason text files compress well is that each character is stored using 8 bits. This gives 256 possible characters. However, most text files use some characters more than others, so these characters could be represented using shorter sequences of bits. In fact, many of the possible 256 characters are never used, thus making some bits irrelevant. Binary files, by contrast, are not necessarily broken down by every 8 bits. Instead, if you sample every 8 bits of a binary file, you are much more likely to get an even distribution between all 256 possible 8-bit combinations. Therefore, you cannot remove as many possible combinations, and are therefore left with less compression.

3. Compression is usually much less efficient after the first compression. Some empirical data showing this is given in the table below.

Folder	First Compression Percent Saved	Second Compression Percent Saved
Calgary	43.76%	2.57%
Canterbury	45.66%	3.41%
Waterloo	20.97%	1.80%

The reason for this is that the compression algorithm is supposed to take advantage of the redundancies in the file and store them in a more compact way. After the redundancies that a given compression algorithm searches for are removed, that compression algorithm will be unable to further compress that file.
Put another way, when you compress a text file, it gets significantly smaller, and then becomes a .txt.hf file. In other words, it was converted from a text to a binary file. We know the Huffman algorithm does not compress binary files as well as text files, so it

makes sense that the compression algorithm would not work effectively on already-compressed files.
If the compression algorithm were able to effectively compress already-compressed files, this would mean that each time you run the algorithm, it would remove at least one bit from the file length. Therefore, if you were to run the compression algorithm continuously, you would decrease the file length until the file was just one bit. This is nonsensical, as you cannot decompress a one-bit file into all the possible original files, without losing information.

4. The way the Huffman Tree is stored, as specified by the assignment, is the most efficient way to store the information contained in the tree. However, there are other ways to store the information. For example, you could store the Huffman Codes of each of the 256 possible character values. This would essentially equate to storing an array, wherein the index of each element is the ascii code, and the value at the index is the code that tells you how to traverse the Huffman Tree to reach the node with that ascii value. This storage method would be less efficient than the current way the program works because this would involve storing a special character for each of the possibilities that is not represented. An example of this method of storing a tree, for GO GO GOPHERS, is shown below:

ASCII Code	**HuffTree Code**
00000000	Null
00000001	Null
…	…
01000101 (E)	0110 (left, right, right, left)
01000110 (F)	Null
01000111 (G)	10 (right, left)
…	…
11111111	Null

Where Null is a special sequence denoting that this ascii code is not in the tree.

Binary Representation: 00000000-NULL-00000001-NULL-…-01000101-0110-0100110-NULL-01000111-10-…-11111111-NULL