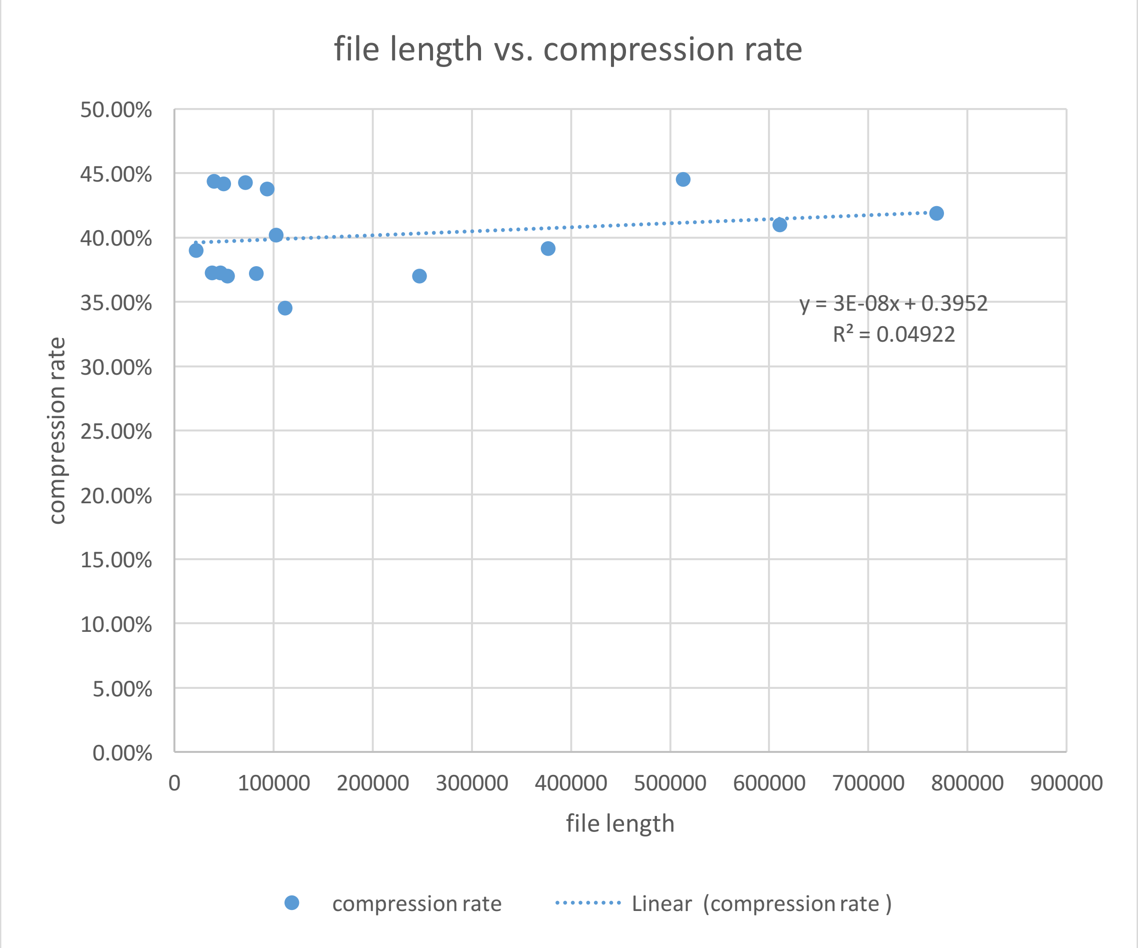
Liane Yanglian,
April 21, 2016

Huffman Analysis

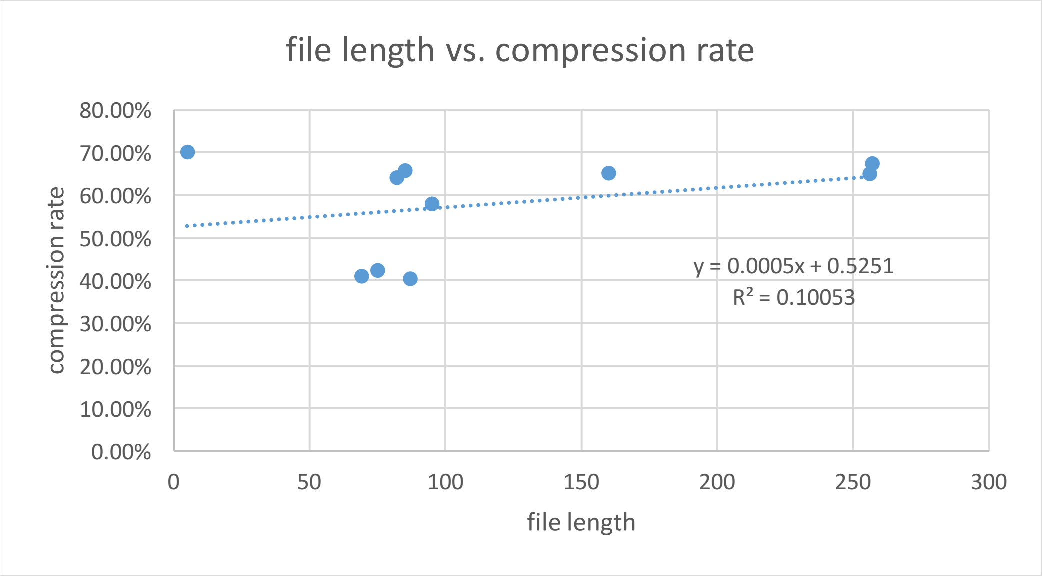
1. Benchmark your code on the given calgary and waterloo directories. Develop a hypothesis from your code and empirical data for how the *compression rate* and *time* depend on *file length* and *alphabet size*. Note that you will have to add a line or two of code to determine the size of the alphabet.

File Length vs. Compression Rate

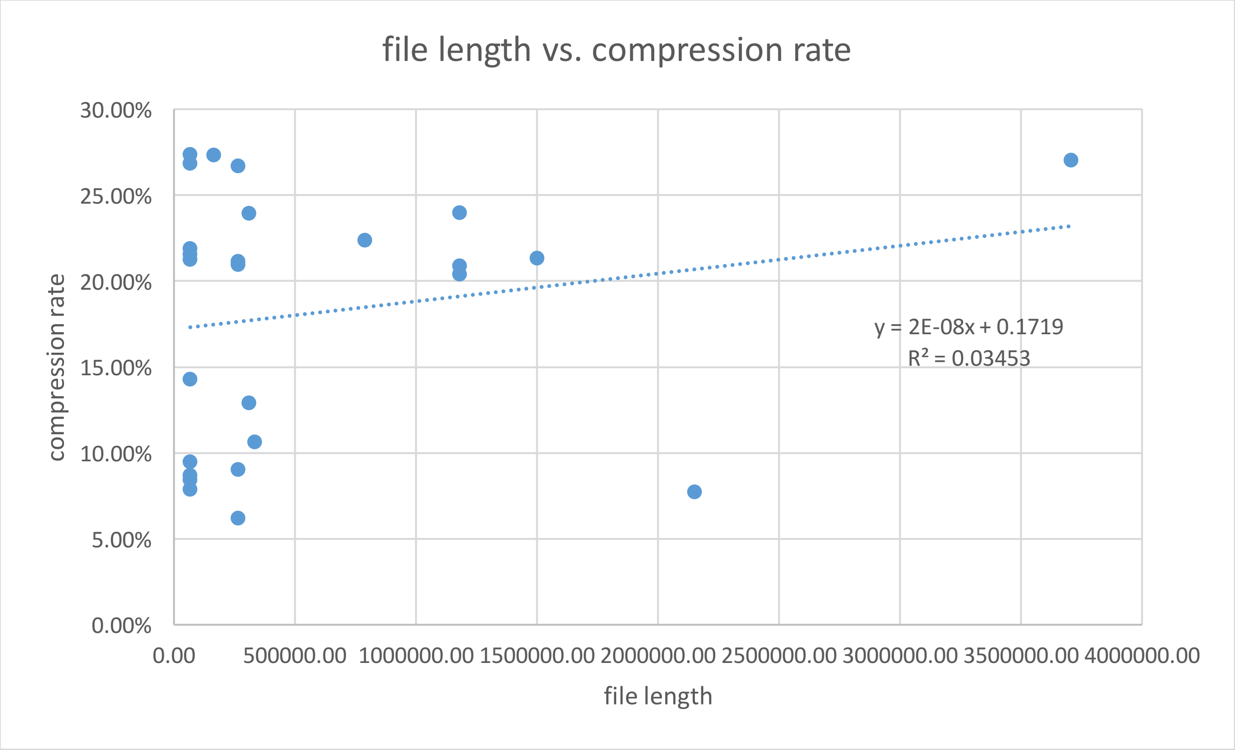
With Calgary:



With Canterbury:



With Waterloo:

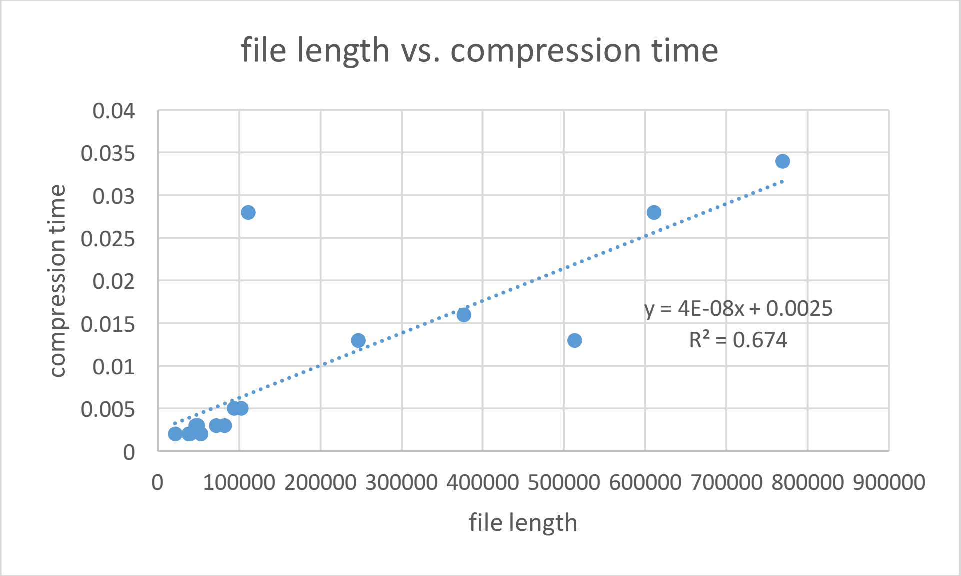


Analysis

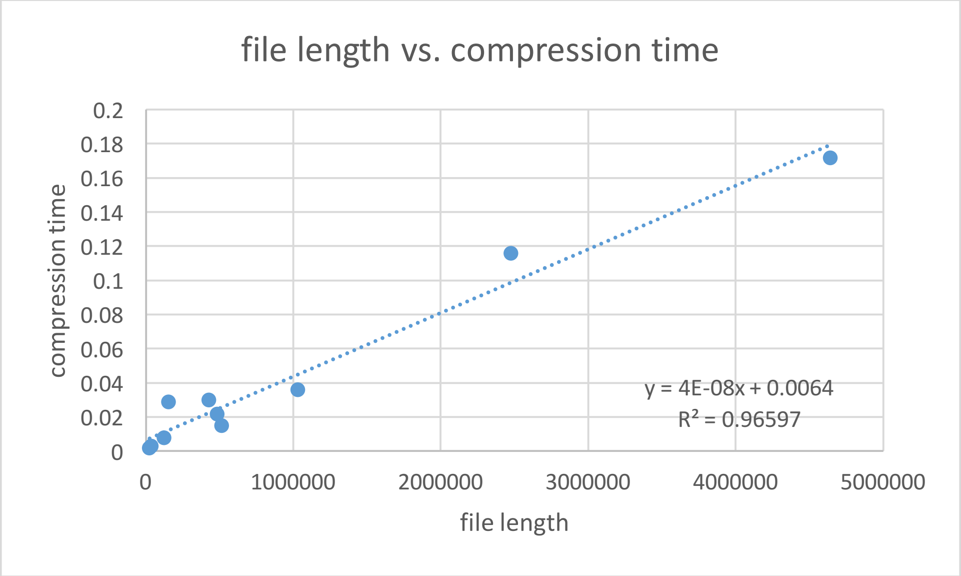
Compression rate only depends on how unbalanced the Huffman tree is, which has no relation to how long the file is. This is supported by the data from Calgary, Canterbury, and Waterloo, graphed above. As shown in the graphs, the relationship between file length and compression rate is nonexistent. Compression rate only increases slightly as file length increases. This is due to the fact that there's more order and less randomness with bigger files.

File Length vs. Compression Time

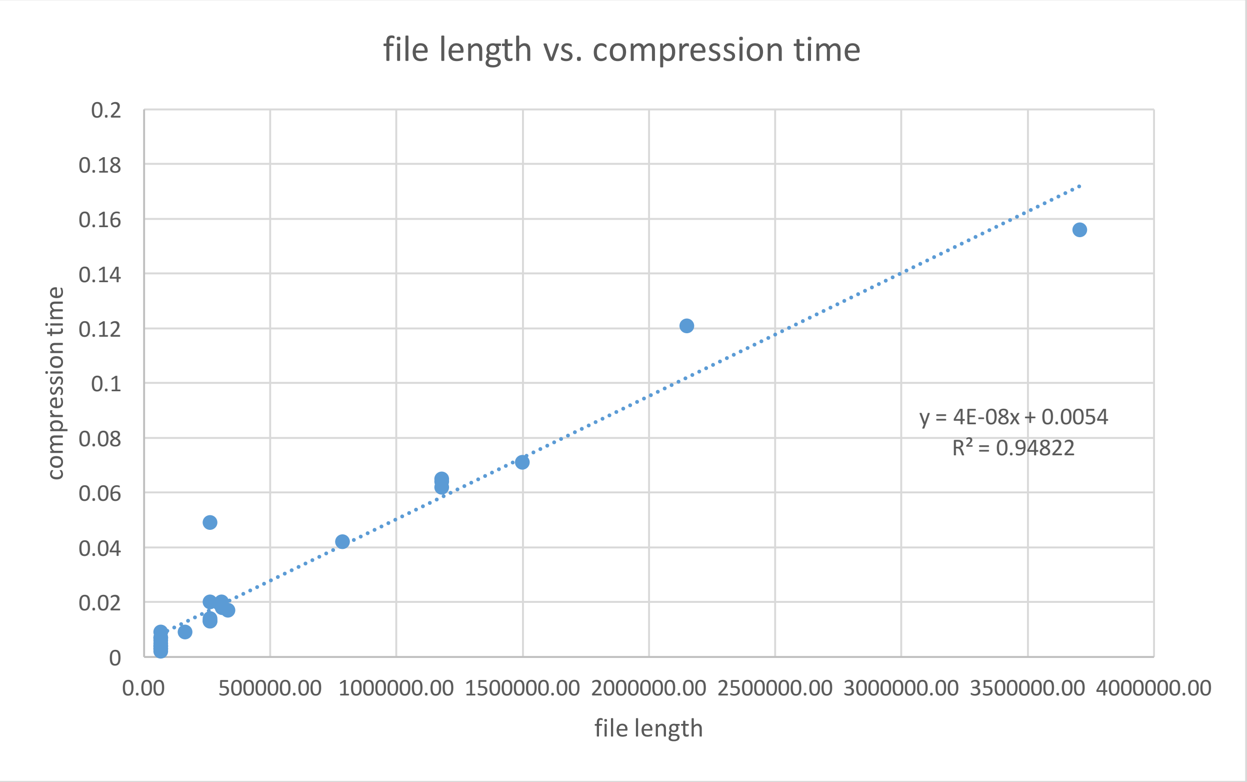
With Calgary:



With Canterbury:



With Waterloo:



Analysis

File length and compression time have a linear relationship. This is because in.readBits(*BITS_PER_WORD*) is used several times throughout Huffman Processor, and when the file length increases, compression time also increases because longer files take more time to be read in 8 bits.

The relevant code is when we count characters in the file:

```
 int var = in.readBits(BITS_PER_WORD);
        while (var != -1) {
              array[var]++;
              var = in.readBits(BITS_PER_WORD);
        }
```

And in the body of the compress method:

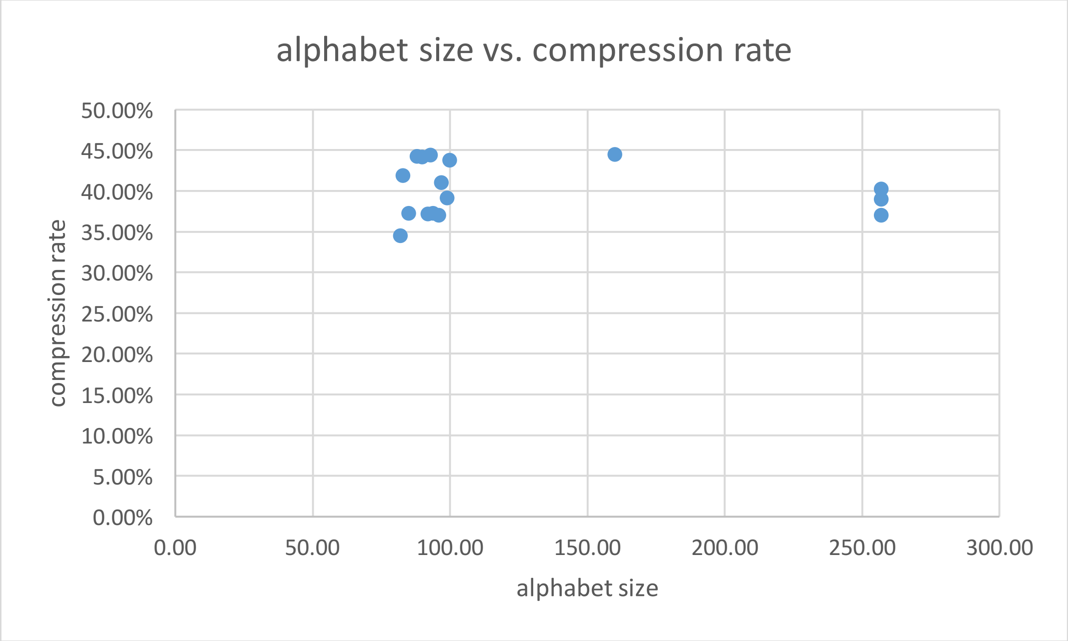
```
int temp = in.readBits(BITS_PER_WORD);
            while (temp != -1)
            {
                  String code = stringarray[temp];
                  out.writeBits(code.length(), Integer.parseInt(code,
2));

                  temp = in.readBits(BITS_PER_WORD);
            }
```

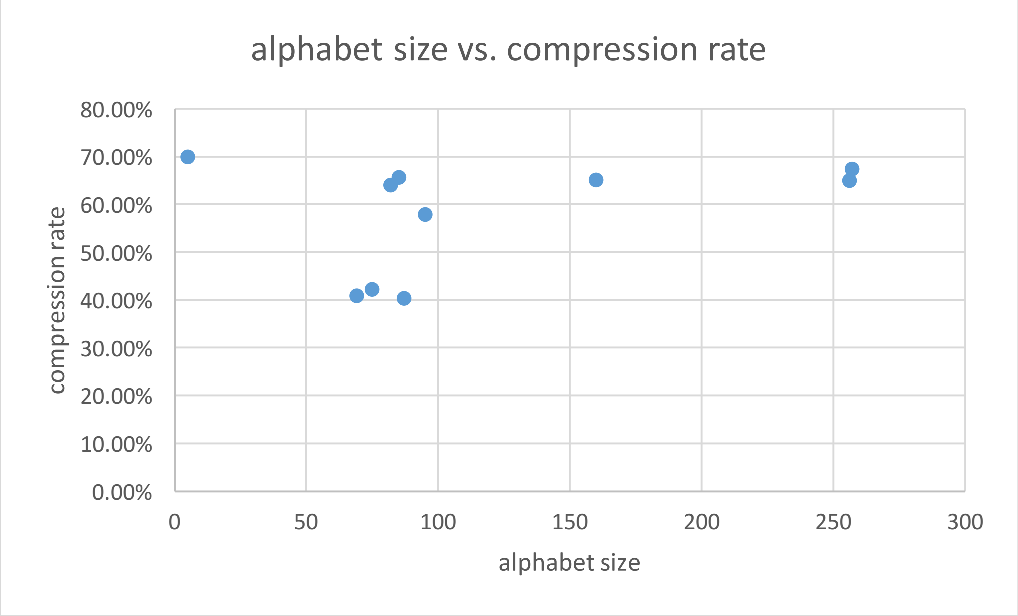
This linear relationship is supported by the data from Calgary, Canterbury and Waterloo, which have been graphed above.

Alphabet Size vs. Compression Rate

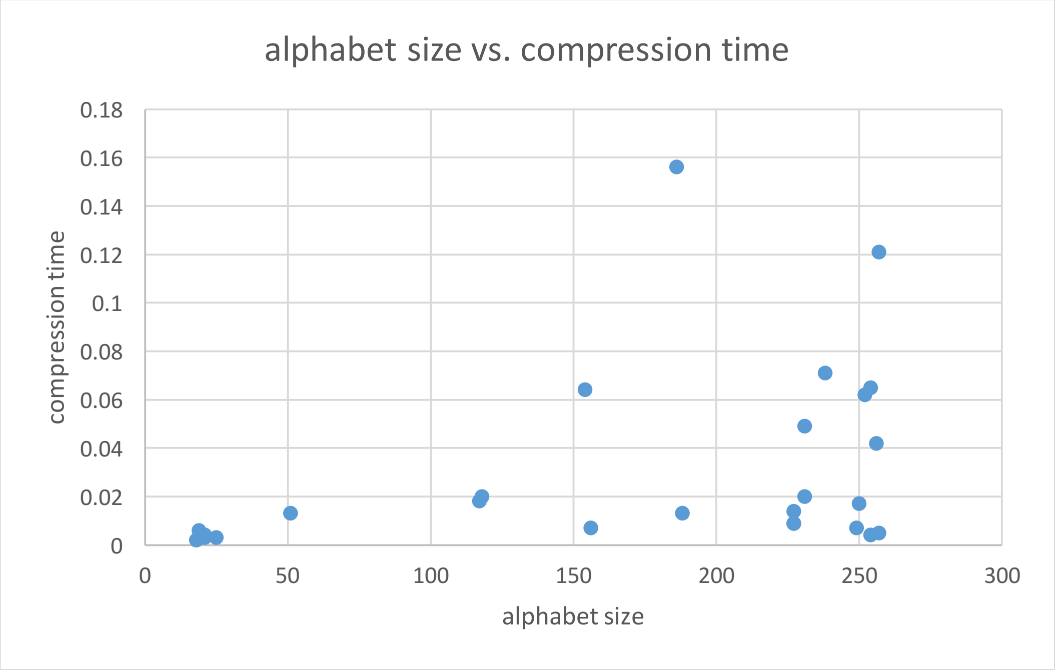
With Calgary:



With Canterbury:



With Waterloo:

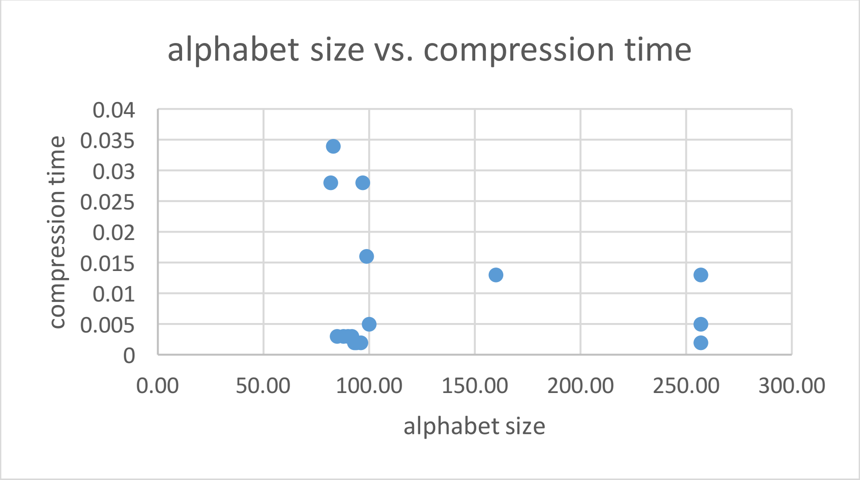


Analysis

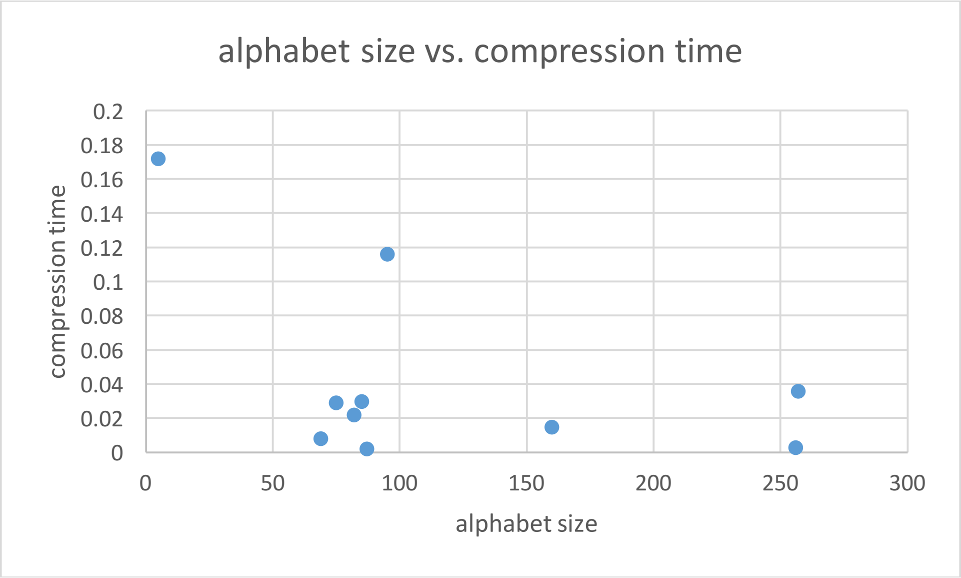
There's no significant relationship between alphabet size and compression rate. This is because alphabet size doesn't affect frequency distribution, and thus it doesn't affect how unbalanced the Huffman Tree is. This is supported by data from Calgary, Canterbury and Waterloo, which are graphed above. As shown in the graphs, there's no correlation between alphabet size and compression rate.

Alphabet Size vs. Compression Time

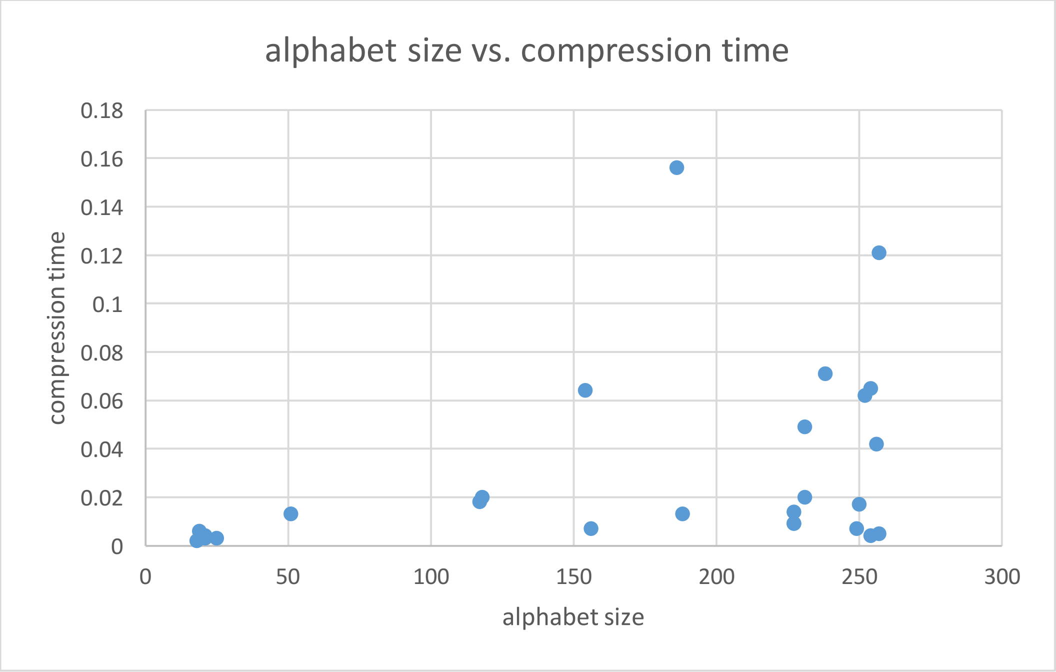
With Calgary:



With Canterbury:



With Waterloo:



Analysis

Compression time doesn't depend on alphabet size. This is because compression time only depends on file length, since with longer file there are more bits; more bits indicate that it will take a longer time to compress the file, which has nothing to do with alphabet size. This is supported by data from Calgary, Canterbury and Waterloo, which are graphed above. As shown in the graphs, there's no correlation between alphabet size and compression time.

2. Do text files or binary (image) files compress more (compare the calgary (text) and waterloo (image) folders)? Explain why.

Compression rate	Text file (Calgary)	Binary(image) file (Waterloo)
	34.50%	6.24%
	41.89%	7.89%
	40.99%	9.05%
	40.20%	8.47%
	39.14%	8.73%
	38.97%	14.32%
	37.01%	7.76%
	37.01%	9.50%
	37.18%	10.65%
	37.26%	12.92%
	37.25%	27.06%
	44.49%	26.87%
	44.36%	27.39%
	44.25%	27.37%
	44.16%	26.71%
	43.76%	24.02%
		23.95%
		22.39%
		20.90%
		21.36%
		21.28%
		21.59%
		21.91%
		20.41%
		21.19%
		20.97%
Average compression rate	40.15125%	18.11153846%

Text files compress more than binary (image) files. This is because text files consist of characters, which abide by the ASCll table that gives the numerical presentations of characters. Since the file is read in 8 bits, each character is represented by 8 numbers in the binary system, consisting of 0 or 1. Thus, when the text files are compressed, eight bits are read each time, which correspond to one character. Since in each text file some characters (such as "e" or "i") appear with more frequency than others (such as "x" or "z"), this creates an uneven distribution, making the Huffman tree unbalanced, enabling the text files to compress more. However, with binary (image) files the content of the file is more random, so there's a lesser chance to have the same eight numbers appear with high frequency like in text files with characters. As a result, binary (image) files tend to have less uneven distribution in the Huffman tree due to the fact that they don't use the ASCll system.

The data from Calgary (text file) and Waterloo (binary, image file) support this explanation, as the average compression rate of Calgary (40.15125%) is much higher than that of Waterloo (18.11153846%).

3. How much additional compression can be achieved by compressing an already compressed file? Explain why Huffman coding is or is not effective after the first compression.

Little compression can be achieved by compressing an already compressed file. This is proven when I compressed the kjv10.txt.hf, which has already been compressed before. The percent space saved from compressing kjv10.txt.hf is a small percentage, 1.51%.

Huffman coding is not effective after the first compression because after the files have been compressed once, the files after compression no longer consist of the same content as before. In fact, after kjv10.txt, a text file, has been compressed, the majority of kjv10.txt.hf consists of symbols that are non-characters. Thus, to compress files that are already compressed, each time eight bits are still read, but the file's content is more random than before. This means there is less of an uneven distribution in the Huffman tree, since we don't get as high of a frequency of reoccurring characters or symbols as before, with the more ordered original file. Consequently, Huffman coding is not effective after the first compression.

4. Devise another way to store the header so that the Huffman tree can be recreated (note: you do not have to store the tree directly, just whatever information you need to build the same tree again).

Another way to store the header is to create key-value pairs in a map and continuously store the frequencies of each character with their number representation 0-255, from 0 up to 255. This way it's similar to having an ASCll table because each number representation of character has its corresponding frequency, stored as the value in the map. We would store the map as the header, so that the Huffman tree can be recreated.