Huffman Analysis

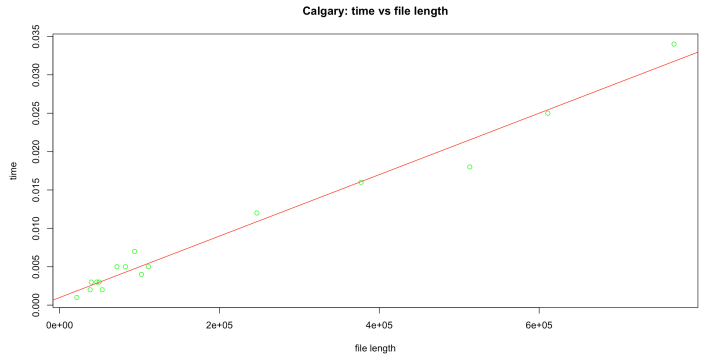
1. Benchmark your code on the given calgary and waterloo directories. Develop a hypothesis from your code and empirical data for how the compression rate and time depend on file length and alphabet size. Note that you will have to add a line or two of code to determine the size of the alphabet.

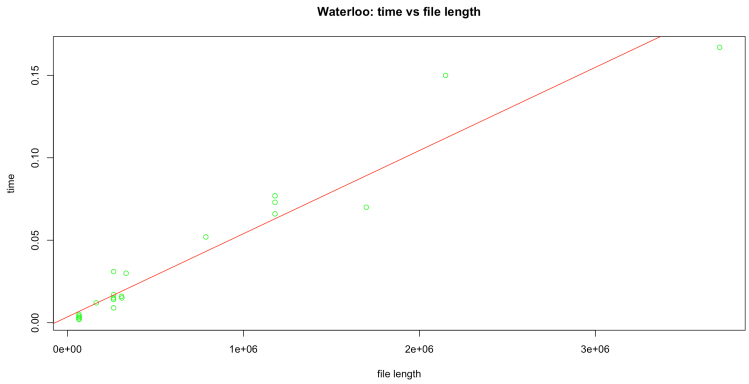
Answer:

File length vs time,

```
//count characters in file
int[] count = new int[ALPH_SIZE];
while(true){
	int character = in.readBits(BITS_PER_WORD);
	if(character == -1)
		break;
	count[character]++;
}
```

For this counting characters' part, it will loop over the whole file, so if the file length increases, the runtime will increase. The empirical data also support the linear relationship between the runtime and file length.



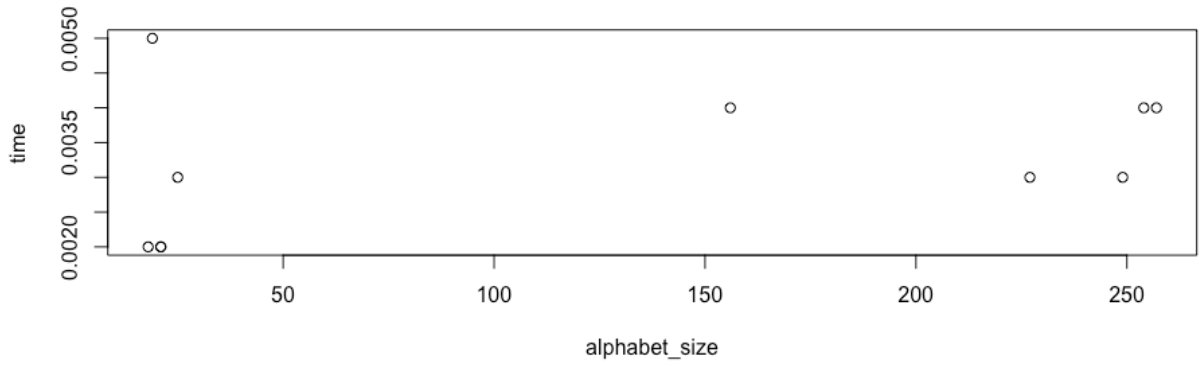


Alphabet size vs time,

```
while(HuffmanTree.size() > 1){
                HuffNode sub1 = HuffmanTree.poll();
                HuffNode sub2 = HuffmanTree.poll();
                HuffmanTree.add(new HuffNode(-1,
sub1.weight()+sub2.weight(), sub1, sub2));
      }
```

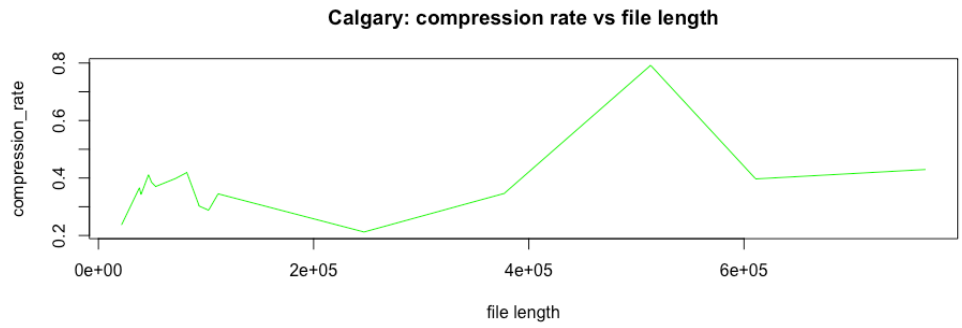
Since alphabet size is the size of the HuffmanTree, and it will loop over the HuffmanTree to add one new HuffNode to the tree and poll two HuffNodes for each iteration, the runtime will have linear relationship with the alphabet size.

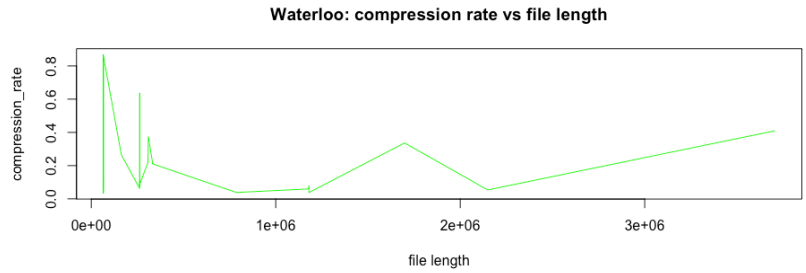
For waterloo data, when file length = 65666, the plot for time vs alphabet size is shown as follows. The linear relationship is not that clear, as alphabet size increases, the time increases.



File length vs compression rate,

The compression rate doesn't depend on the file length, since the new encoding is mainly based on the unique characters in the original file. And the empirical data also support my hypothesis.

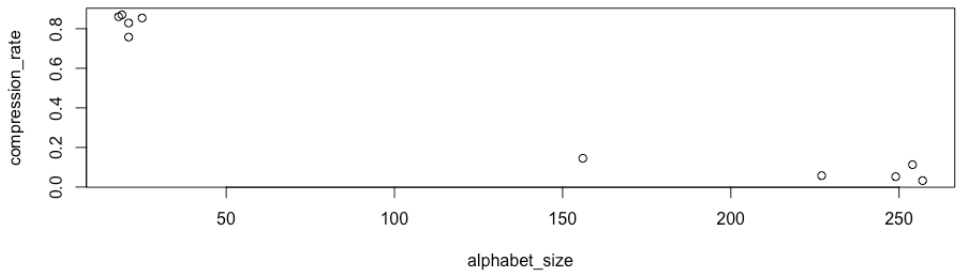




Alphabet size vs compression rate,

```
            while(HuffmanTree.size() > 1){
                  HuffNode sub1 = HuffmanTree.poll();
                  HuffNode sub2 = HuffmanTree.poll();
                  HuffmanTree.add(new HuffNode(-1,
sub1.weight()+sub2.weight(), sub1, sub2));
            }
            //traverse tree and extract codes
            String[] codes = new String[ALPH_SIZE+1];
            HuffNode root = HuffmanTree.poll();
            extractCodes(root, "", codes);
            //write the header
            out.writeBits(BITS_PER_INT, HUFF_NUMBER);
            writeHeader(root, out);
            //compress
            while(true){
                  int character = in.readBits(BITS_PER_WORD);
                  if(character == -1)
                        break;
                  String code = codes[character];
                  out.writeBits(code.length(), Integer.parseInt(code, 2));
      }
```

Since the compress process is based on the HuffmanTree, the less the alphabet size is (the lower the HuffmanTree is), the new encoding will be shorter. So I assume that as the alphabet size increases, the compression rate will decrease. And the data support my hypothesis.



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

Answer:

For files in calgary(text) folder, the percent space saved is 43.76%. For files in waterloo(image) folder, the percent space saved is 20.97%. The text files compress more because not all characters in the ASCII table appear in a text file and thus the alphabet size for a text file is generally less than that of an image file. Therefore, text files are more compressible than image files.

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.

Answer:

For Calgary data files, the average compression rate for compressed files is 2.59%. For waterloo data files, the average compression rate for compressed files is 1.80%. The Huffman coding is not effective because it extracts the unique characters in the files and give the most common character the shortest coding. After the first compression, the encoding has already had this feature, so it's hard to compress more using this strategy.

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).

Answer:

We can write a writeHeader helper method, every time we call it with the left HuffNode, we write a "0", and every time we call it with the right HuffNode, we write a "1". Then if we reach a leaf HuffNode, we write the value of the HuffNode.