Miguel Ricardo Anderson (netId: mra21)

CompSci 201 – Assignment Analysis for Huffman

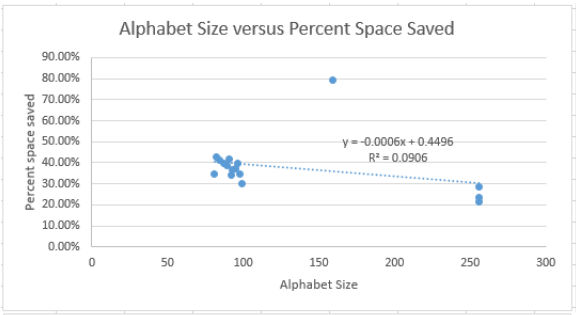
04/21/2016

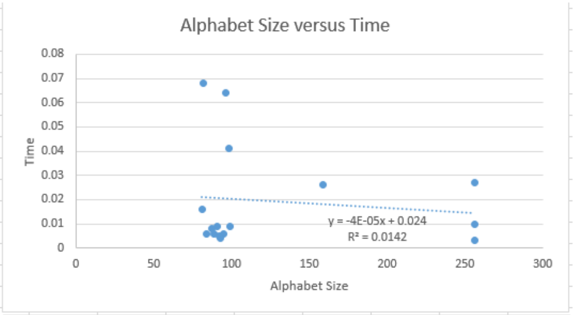
Part 1

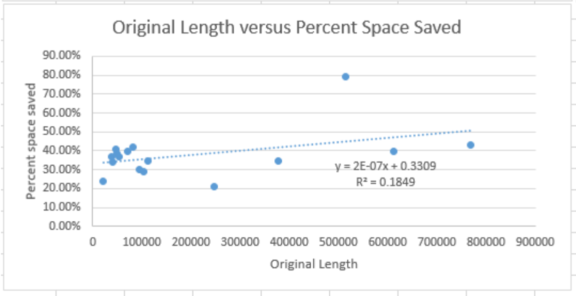
Benchmark your code on the given calgary and waterloo directories. Develop a hypothesis from your code and empirical data for how the compression rate and time depend on file length and alphabet size.

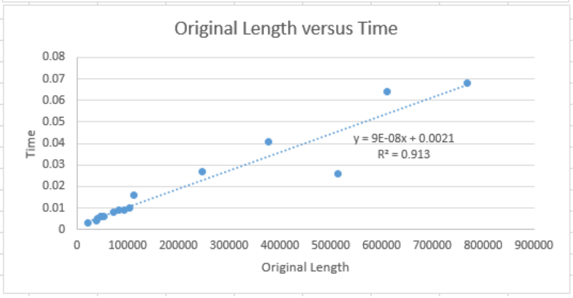
I hypothesize the larger the alphabet size, the less percent space saved. I also think time will increase as original length increases.

Calgary

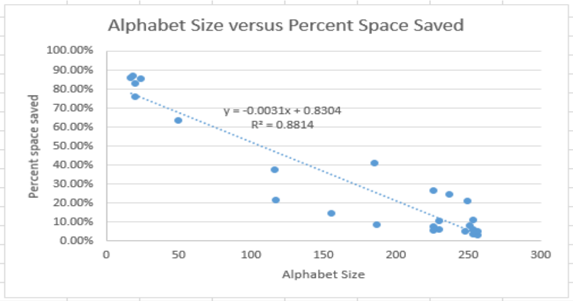


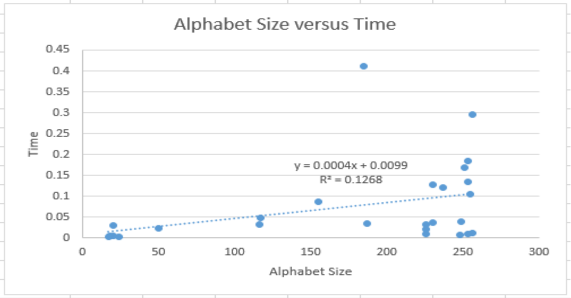


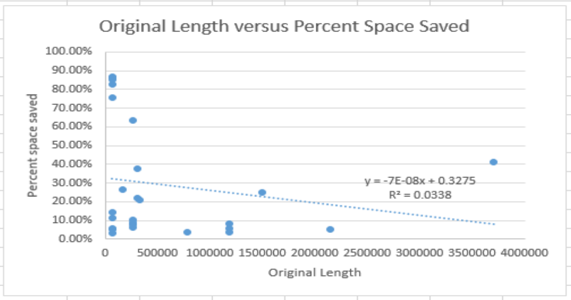


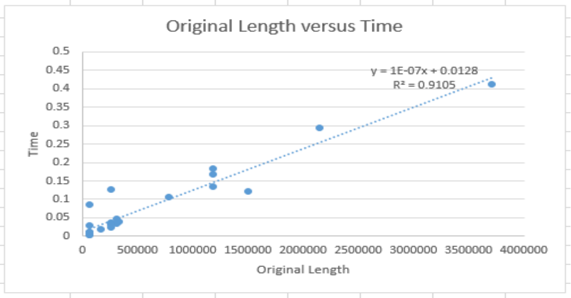


Waterloo









Part 2

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

The text files (calgary) compressed more than the image files (waterloo) because, on average the text files use less characters than the images. The more characters, and specifically the less instances of each unique character, the less comprised the file can be to the point where an even usage of all characters should lead to no compression.

Part 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.

Very little, if any, compression is achieved by compressing an already compressed file. Huffman coding is not effective after the first compression because the first compression maximizes compression by design. The second compression will only add the Huff number and the EOF number to the compression. The compression gained from those numbers is usually very small considering a new Huff number and EOF number will be added regardless. For some files the second compression did not actually save space.

Part 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 would be to write a representation of the character occurrence array (*occurArray* in my code). If this is done correctly, the same priority queue process can be done to recreate the tree. An easy way to do this would be to start a 1 and then a binary representation of the max number in the array. Then write the ASCII value of a character followed by its frequency. One must take care that the length of this frequency has leading zeros so that it is the length of the max number so that every character-frequency pair is the same length. This will construct the occurrence array which will create the priority queue and then the tree.