Daniel Rubinstein

NetID : dr148

Huffman Analysis

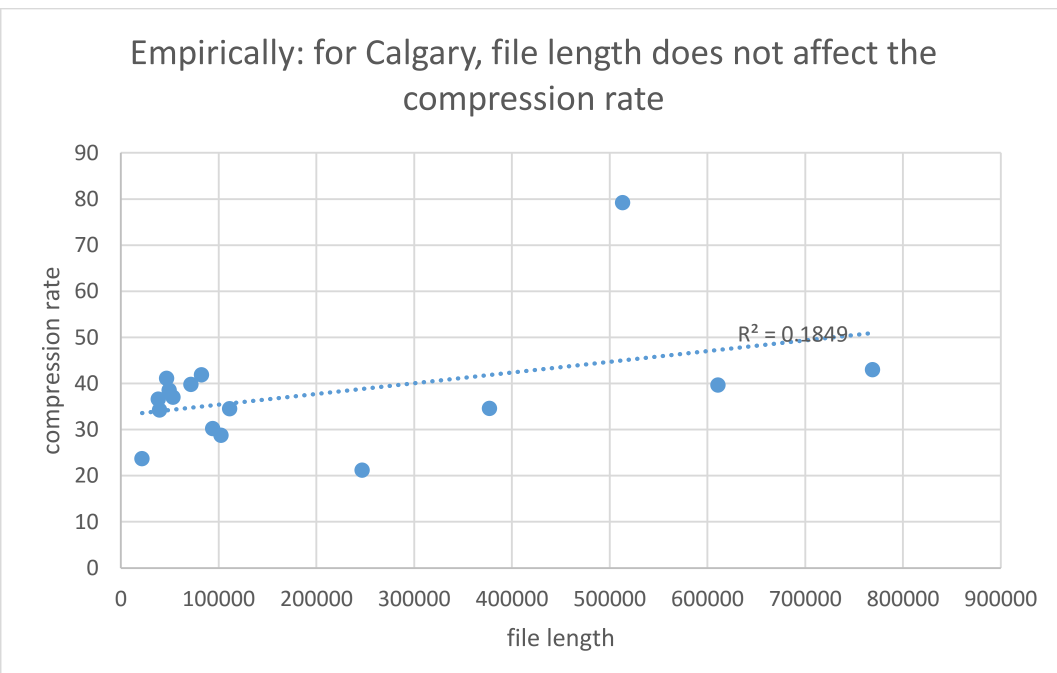
1. Benchmark your code on the given calgary and waterloo directories. Develop a hypothesis from your code and empirical data for how the compression rate and time depend on file length and alphabet size. Note that you will have to add a line or two of code to determine the size of the alphabet.

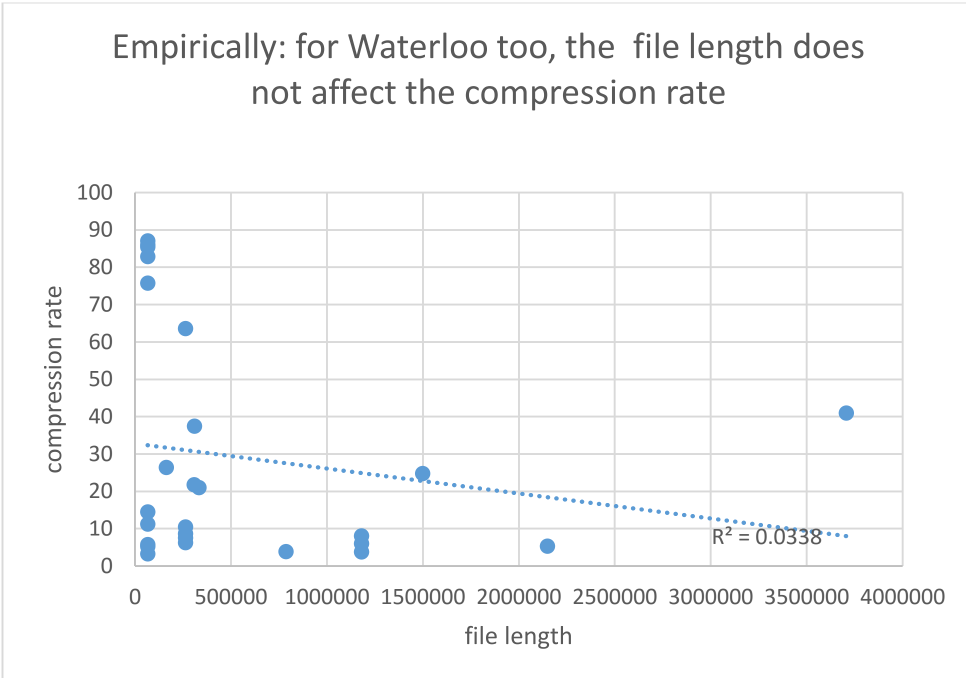
Data

Directory	File name	Compression time (seconds)	Compression rate (percentage)	Original file length (bytes)	Alphabet Size
calgary	bib -> bib.hf	0.049	34.4963644	111261	81
calgary	book1 -> book1.hf	0.101	42.96155812	768771	82
calgary	book2 -> book2.hf	0.049	39.68463926	610856	96
calgary	geo -> geo.hf	0.012	28.79199219	102400	256
calgary	news -> news.hf	0.031	34.62473714	377109	98
calgary	obj1 -> obj1.hf	0.002	23.68396577	21504	256
calgary	obj2 -> obj2.hf	0.029	21.21354542	246814	256
calgary	paper1 -> paper1.hf	0.006	37.03090612	53161	95
calgary	paper2 -> paper2.hf	0.008	41.91170209	82199	91
calgary	paper3 -> paper3.hf	0.006	41.11249624	46526	84
calgary	paper6 -> paper6.hf	0.005	36.60149587	38105	93
calgary	pic -> pic.hf	0.05	79.19433533	513216	159
calgary	progc -> progc.hf	0.004	34.24048875	39611	92
calgary	progl -> progl.hf	0.006	39.82916004	71646	87
calgary	progp -> progp.hf	0.004	38.5487758	49379	89
calgary	trans -> trans.hf	0.01	30.24067453	93695	99
waterloo	barb.tif -> barb.tif.hf	0.106	6.235844956	262274	230
waterloo	bird.tif -> bird.tif.hf	0.016	14.51436055	65666	155
waterloo	boat.tif -> boat.tif.hf	0.067	10.50085026	262274	230
waterloo	bridge.tif -> bridge.tif.hf	0.024	3.234550605	65666	256
waterloo	camera.tif -> camera.tif.hf	0.026	11.28285566	65666	253
waterloo	circles.tif -> circles.tif.hf	0.029	75.78351049	65666	20
waterloo	clegg.tif -> clegg.tif.hf	0.335	5.359276645	2149096	256
waterloo	crosses.tif -> crosses.tif.hf	0.003	87.0176347	65666	18
waterloo	france.tif -> france.tif.hf	0.027	21.01774821	333442	249
waterloo	frog.tif -> frog.tif.hf	0.037	37.42323555	309388	116

waterloo	frymire.tif -> frymire.tif.hf	0.442	40.97030844	3706306	185
waterloo	goldhill.tif -> goldhill.tif.hf	0.006	5.756403618	65666	226
waterloo	horiz.tif -> horiz.tif.hf	0.003	85.43538513	65666	24
waterloo	library.tif -> library.tif.hf	0.017	26.41657184	163458	226
waterloo	mandrill.tif -> mandrill.tif.hf	0.026	7.594729176	262274	226
waterloo	monarch.tif -> monarch.tif.hf	0.171	5.974737749	1179784	253
waterloo	mountain.tif -> mountain.tif.hf	0.029	21.80782872	307330	117
waterloo	peppers.tif -> peppers.tif.hf	0.068	3.84912684	786568	255
waterloo	sail.tif -> sail.tif.hf	0.206	8.049354797	1179784	251
waterloo	serrano.tif -> serrano.tif.hf	0.285	24.79087889	1498414	237
waterloo	slope.tif -> slope.tif.hf	0.006	5.244723297	65666	248
waterloo	squares.tif -> squares.tif.hf	0.003	82.88612067	65666	20
waterloo	text.tif -> text.tif.hf	0.005	86.06584838	65666	17
waterloo	tulips.tif -> tulips.tif.hf	0.119	3.780522536	1179784	253
waterloo	washsat.tif -> washsat.tif.hf	0.022	63.56901561	262274	50
waterloo	zelda.tif -> zelda.tif.hf	0.024	8.6954864	262274	187

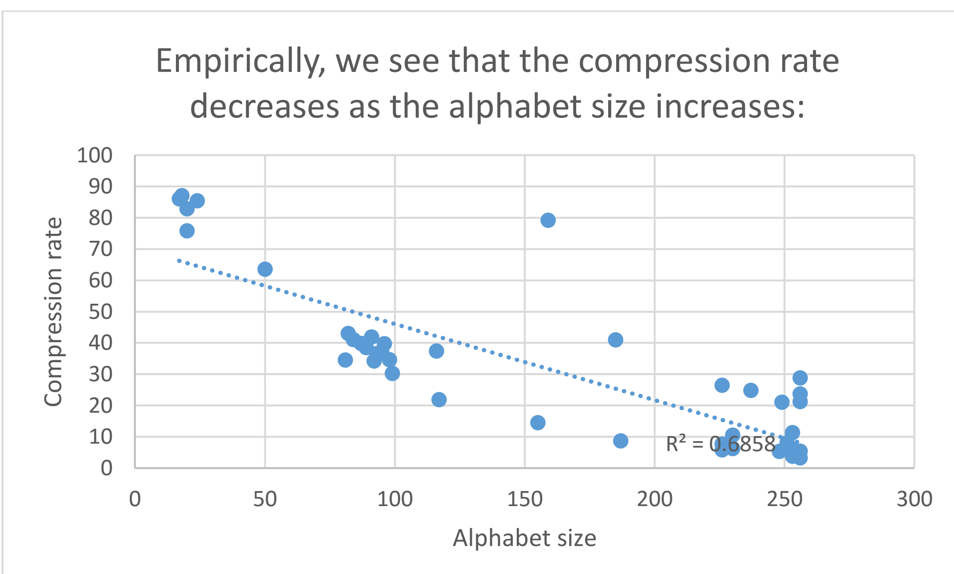
1. File length and compression rate:





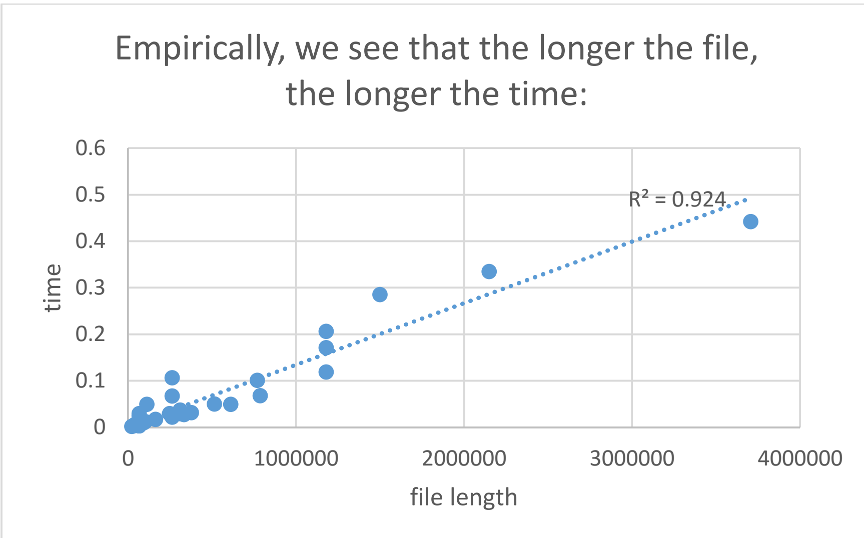
This hypothesis is supported by the code: file length does not affect the compression rate because the only thing that matters for the compression is the distribution of the characters, not the length of the file itself.

2. Compression rate and alphabet size:



This hypothesis is supported by the code: the compression rate decreases as the alphabet size increases because as the alphabet size increases, the paths for the characters will become longer. This causes compression to decrease.

3. Time and file size

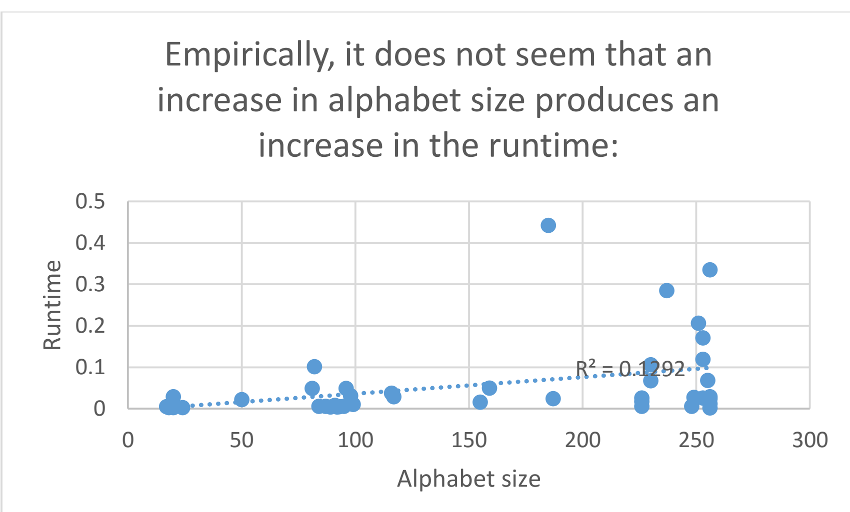


This hypothesis is supported by the code: the time increases as the file length increases because we need to read in the file. We have the following while loops which are O(n) (where n is the length of the file):

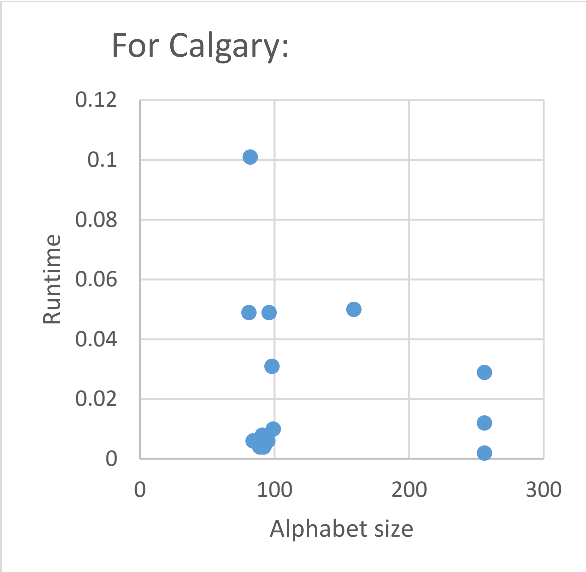
-while (in.readBits(BITS_PER_WORD)!= -1)

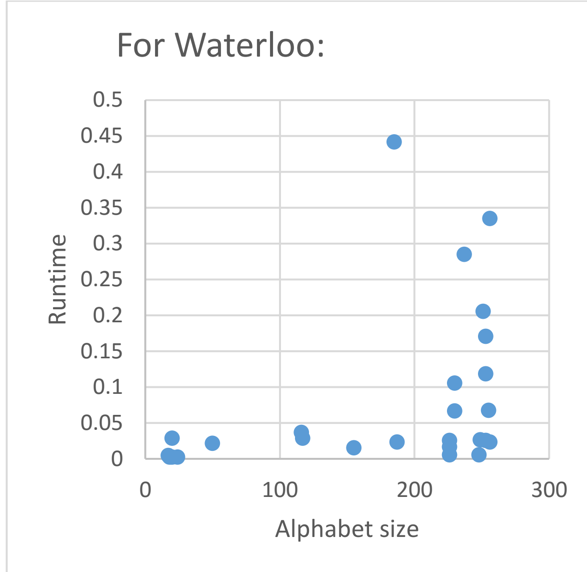
-while (in.readBits(BITS_PER_INT)!= -1)

4. Alphabet size and runtime



Empirically, this appears to be the case for both for Calgary and Waterloo:





However, from the code we can deduce that an increase in alphabet size **does** produce an increase in the runtime because it will take longer to build the Huffman tree (we will need to add more nodes to the tree, and we will need to recombine nodes more times).

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

20.97% of space is saved in the waterloo (image) folder, whereas 43.76% of space is saved in the calgary (text) file. We can see that text files can be compressed more than image files.

This is the case because Huffman is an entropy encoding algorithm. Entropy encoding algorithms work by utilizing variable length codes combined with character frequencies. Characters that occur a lot get shorter codes while lesser used characters get longer codes. Therefore, there will be a stronger compression for characters which appear frequently than for characters which appear less frequently.

In a text file, certain characters (such as vowels) appear very frequently. They will be associated to short codes and the compression will be good.

However, an image file is composed of pixels with RGB values. Each pixel can be represented as a sequence of characters depending on its RGB value. Since there are not certain pixels values which appear much more frequently than the other pixels, there will not be certain characters which appear much more frequently than others. Therefore, compression for an image is not very good.

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.

First, I tried re-compressing some already compressing files:

-compressing monarch.tif.hf.hf: -0.07% space saved
compressing monarch.tif.hf: -0.07% space saved
compressing monarch.tif: 5.97% space saved

-compressing melville.txt.hf.hf: -0.67% space saved
-compressing melville.txt.hf: 0.65% space saved
compressing melville.txt: 43.62% space saved

Huffman coding does not appear to be very effective after the first compression. Compression the following times is either very low or compression actually marginally increases the file size.

This is because with compression, characters that occur a lot get shorter codes while lesser used characters get longer codes. In a compressed file, characters that occur a lot are already represented with short codes whereas character that rarely occur are already represented with long codes. Re-compressing the file will therefore not be effective in further compressing the file.

Re-compressing can even increase the file size because we need to re-write a header when we compress again.

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).

We could use the frequency array to store the header. The frequency array has a length of 256 (because there are 256 ASCII values). Each character's index in the array is its ASCII value, and the corresponding value is the character's frequency. We could then just use the frequency array to create the tree.

However, this solution would be less space efficient.