CPE 593 Final Project – Milestone 2

**David Krauthamer**

I initially chose to do Run Length Encoding (RLE) as my algorithm of choice. I quickly found that on the sample text we were using, the results were terrible, which makes a lot of sense. RLE takes advantage of groups of the same thing coming one after another, which happens a lot in things like images (lots of similarly colored pixels together), but happens very rarely with plain text. I've now pivoted to trying to optimize Huffman coding, and seeing if RLE can be used to help there.

The first thing I tried to optimize was how big of a difference the size of each of your alphabet members is. Traditionally Huffman coding on plain text is done on a per-letter basis, but I wanted to see what would happen if I worked on different groups of letters. I've found generally what I expected, that the larger each alphabet member is the more you can compress the original text, but the larger the code mapping required for decoding the text is. Looking at it purely from a size standpoint (ignoring computational speed or anything else), there's a point where increasing the alphabet member size leads to the overall size (code + compressed data) to stop decreasing and start incresing back towards the original uncompressed size. For our sample text I found that to be an alphabet member size of 5.

The next thing I tried was changing the alphabet size, but a little bit more intelligently. The first test I've been able to run was grouping on words and punctuation characters instead of arbitrary groups, and that has produced the best compression results so far. I'm looking to try incorporating RLE post huffman coding, but so far it's had no benefit whatsoever. The next thing I'm thinking of trying is RLE on the sample data set, but instead of on a per character (byte) basis do it on a per bit basis).

**Joshua Canlas**

I started my search for a compression algorithm by looking at a comparative study between various algorithms Mohmmed and Emary ([Comparative Study between Various Algorithms of Data Compression Techniques](https://www.researchgate.net/publication/44261387_Comparative_Study_between_Various_Algorithms_of_Data_Compression_Techniques)). This paper introduced five different algorithms and showcased their performance and their use cases: run length encoding (RLE), Huffman coding, arithmetic coding, LZ-77 encoding, and LZW coding. Of these five algorithms, the one that piqued my interest the most was arithmetic coding.

Arithmetic coding is a slight improvement on Huffman coding. Like Huffman coding, arithmetic coding is a lossless algorithm and works by taking a character and assigning it a frequency to a table. However, instead of using frequency, arithmetic coding turns the frequency table into a probability table. Ultimately, arithmetic coding encodes the entire message using a single number between 0.0 and 1.0. Each character in the message takes a sub-interval in the number line, corresponding to its probability (see [Lossless Data Compression Using Arithmetic Encoding in Python and Its Applications in Deep Learning](https://neptune.ai/blog/lossless-data-compression-using-arithmetic-encoding-in-python-and-its-applications-in-deep-learning) for more details on how the encoding and decoding process works).

As part of this milestone, I’ve also set up the architecture (skeleton functions) for the arithmetic encoder and decoder in Python. I’ve updated our final report with my research and algorithm API specifications. I’m still working on writing up the pseudocode for each function.

Lastly, I read this paper ([Data Compression With Arithmetic Coding](https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html)), which introduces finite-precision arithmetic coding to solve the floating point representation problem with this algorithm. I am planning on using the “decimal” module in Python to represent more characters with a floating point. However, if this doesn’t work out, I may need to implement finite-precision arithmetic coding to solve that problem.

**John Theising**

Initially, I was investigating compression using neural nets. There have been a lot of recent papers with both lossy and lossless compression algorithms using neural networks on all sorts of data. Simply setting up the environments with gpu for these algorithms proved very difficult as most were intended to run on a linux machine. After attempting a couple of their implementations, I ultimately could not get any of the code from the papers to completely run (sometimes I could get a partial run, but a bug/package issues/environment issue would stop it from finishing) and pivoted to Bzip2.

Pivot to Bzip2.

Bzip compression has been around since the original paper in 1994 and is still a standard lossless compression today. Many linux systems come pre loaded with bzip2 and there are still papers being published improving or using bzip2 compression. In the other papers I looked at, bzip2 was used as a metric to compare other compression algorithms against.

Bzip initially uses Run-Length encoding on the data before the Burrows-Wheeler Transform which blocks and rearranges the characters within each block to help with compression down the line. Next, the Move-To-Front and algorithm and run-length encoding are applied to each block. After that, Huffman Coding is used at the block level.

Project Results

Compare all at the end on set text files with an experiment or testing script. Also should take a text file as input and compare scores. For each of the bzip and arithmetic encoding, we should run a few files through ours and then the library modules for comparison as well.