Data Compression of COVID-19 Genomes

Simon Lee[1] Mateo Etcheveste[1]
siaulee@ucsc.edu *metcheve@ucsc.edu*

[1]Jack Baskin School of Engineering, University of California, Santa Cruz, CA

Abstract

With the recent developments of COVID-19 pandemic, the *NCBI* has collected mass samples of the SARS-CoV-2 genome. As it currently stands there are 21,210 current available genomes that can be used in any sort of pandemic related research. However with every character holding a byte of memory, this raises two issues. The first issue is that it can be hard to run experiments on mass data without time efficient algorithms, and the second issue is that it can be hard to share this dataset easily across laboratories. Therefore we developed a program that tackles the second problem and runs the Huffman Compression Algorithm to compress the FASTA file containing the 21,210 Coronavirus genomes. The results from this paper show how much effective this algorithm is in compressing a file containing sensitive information and then making it ready for sharing. The methods developed in this paper can also be broadly used in any sort of program that needs to transfer and compress massive data files.

1 Introduction

In the world today, the COVID-19 pandemic has been prolonged due to the mass mutations that are developing from month to month. These variations of the COVID-19 occur in the genome directly and researchers are working hard to distinguish whether our current protocols (vaccines, mask mandates, quarantine) are required based on the severity of these mutations. Luckily for us, *The National Center for Biotechnology Information (NCBI)* is a public database that makes this multi-omics data (genome, proteome, transcriptome, epigenome) readily available for research use. In total the NCBI has collected 807,197 coronavirus genomes as a result.

While this has been a convenient public platform for downloading biological data sets, sharing this public data set has been a bit more challenging. With a character holding 1 byte of information, the amount of information carried within a data set containing 807,197 coronavirus genomes, is simply too much to handle over any platform of sharing. Therefore our goal is to use a lossless compression called the *Huffman compression algorithm* to reduce a smaller sample size of this data of 21,120 coronavirus genomes collected from this previous week (February 27th - March 4th) to demonstarte this algorithms power.

2 Huffman Compression

David A. Huffman, a professor in Computer Science at UCSC (1967) and MIT once wondered how to construct an optimal static encoding of information. What he would soon invent was a lossless compression algorithm that would change the world of how we conserved information. His algorithm was inspired by *entropy*, which is a measure of an amount of information based on a set of symbols. It can mathematically be denoted using the following:

$$H(\chi) = \sum_{i=1}^{n} Pr[x_i I(x_i)] = \sum_{i=1}^{n} Pr[x_i] \log_2 Pr[x_i] \tag{1}$$

In a very simplistic way, this equation means that the encoding of information would assign the least amount of bits to the most common symbols and the greatest number of bits to the least common symbols. Therefore, in the context of our problem, we could overly simplify our datafile because the most frequent keys are the nucleotide bases (A,C,G,T). Knowing this we take this algorithm and apply it to our problem going through the following steps.

2.1 Encoding

Our first step is to construct a histogram. A histogram in our case is a type of table that counts the frequency of unique characters that show up in our data file. By constructing a histogram we are able to construct a binary tree using the min heap data structure. For those unfamiliar, a *min heap* performs similarly to a priority queue where the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. From this min heap we are then able to use our binary tree to construct a list of codes that resemble the new representation of the character. This process is essential in the tree reconstruction and decoding the data file from its binarized compressed form.

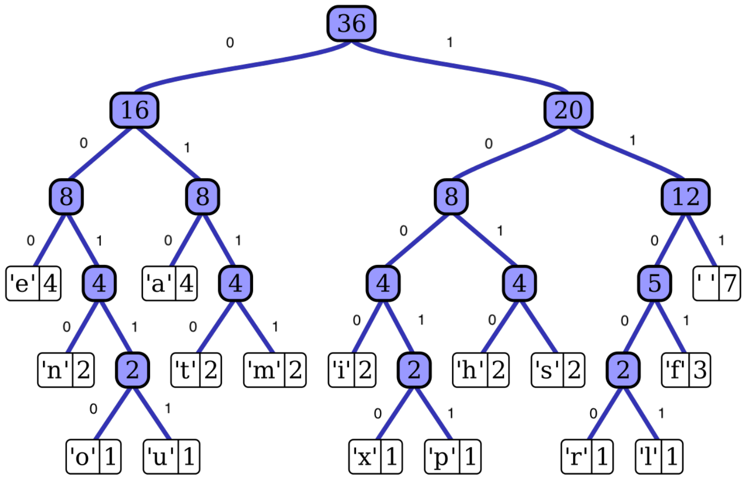


Figure 1: A binary tree and with its binary codes representation

An example of a code can be seen from our **Figure 1** where 000 would now resemble the ascii character *e* or 10011 will resemble the ascii character *p*. Since a binary tree is built on bits and not bytes, it makes intuitive sense as to why, a compression does indeed occur. After a codes chart is constructed within a python dictionary, *post order traversal* occurs. By traversing the tree in post order, the algorithm then takes each character one at a time and assigns its new code, usually being some combination of binary bits that are less than a byte (8 bits).

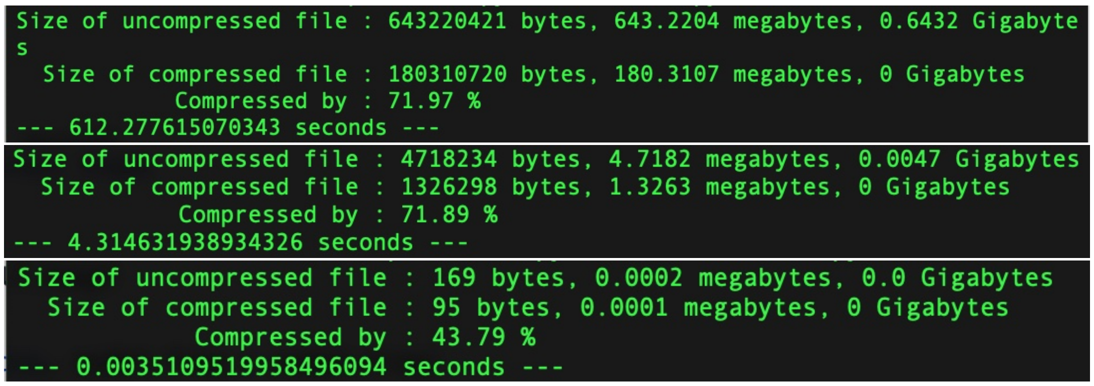
2.2 Decoding

After our file has been compressed (encoded) and shared, we can now begin to look to decompress our file and recover all the contents of our original file. After all this algorithm is lossless. Because we have a pre-established codes table, we are able to reconstruct the binary tree based off the codes table alone. After reconstructing the tree we take the binarized file and "walk the tree", meaning we read one bit at a time until we reach a character node. This process will then recover our original data file. So if our code was 10011 010 1011 1011, we can reconstruct the string literal 'pass' from these bit sequences based off our binary tree. And in the process of decompression, we can see that we were able to decompress this message from 2 bytes back to its original 4 bytes. On a much grander scale, this would obviously play a much bigger deal in our file transfer system.

3 Experiments Results

Now that the components of the algorithm are understood, we can now proceed to describe the set of experiments that we intend to run. Our main experiment is to compress the 21,210 coronavirus genomes that were uploaded in the week of February 27th - March 4th from the NCBI (https://www.ncbi.nlm.nih.gov/datasets/coronavirus/genomes/). The orginal FASTA file size is 0.642 gigabytes which can be converted into 643220421 bytes or 643220421 characters in a file. Our goal is to see how much we can compress this file size while taking into account of how long it takes to execute the program. We will also calculate compression percentage, and total compressed size as well.

3.1 Results Discussion



```
Size of uncompressed file : 643220421 bytes, 643.2204 megabytes, 0.6432 Gigabyte
s
  Size of compressed file : 180310720 bytes, 180.3107 megabytes, 0 Gigabytes
            Compressed by : 71.97 %
--- 612.277615070343 seconds ---
Size of uncompressed file : 4718234 bytes, 4.7182 megabytes, 0.0047 Gigabytes
  Size of compressed file : 1326298 bytes, 1.3263 megabytes, 0 Gigabytes
            Compressed by : 71.89 %
--- 4.314631938934326 seconds ---
Size of uncompressed file : 169 bytes, 0.0002 megabytes, 0.0 Gigabytes
  Size of compressed file : 95 bytes, 0.0001 megabytes, 0 Gigabytes
            Compressed by : 43.79 %
--- 0.0035109519958496094 seconds ---
```

Figure 2: The data from the three test files: covid.fa, delta_variant.fa, test.fa

Filename	Genome Count	Original Filesize	Compressed Filesize	Compression Percentage	Time
covid.fa	21,210	643.2204 mb	180.3107 mb	71.97	612.2776s
delta_variant.fa	230	4.7182 mb	1.3263 mb	71.89	4.3146s
test.fa	n/a	0.00016 mb	0.000095 mb	43.79	0.0035s

From these set of experiments we can see that the compression succeeded. Our Figure 2 and our table show off the same data and we can see that there was a significant effect with the two larger files. Having a compression percentage of over 70% is a success in our book and we were able to count this percentage by taking the mathematical equation of $1 - (\text{compressed\_bytes}/\text{total\_byte}) * 100$. Having iterated through 21,120 genomes with their genetic sequences, we can say that the 10 minutes it took to execute was well deserved knowing that this compression can make files more easily shared.

An interesting observation that can be made through our experiment is that some characters exceeded the 8 bits it takes to encode a message. Because in some cases we had so many characters to keep track of in our histogram, a few characters actually exceeded 8 bits taking up more space than what it would take for its normal space allocation for that single character. However as predicted, since the majority of the file consists of Nucleotide bases, our file becomes compressed, typically writing out 4 nucleotide bases with 1 byte vs just 1 nucleotide base. This algorithm can be slightly improved for optimization but are overall content with the results and our findings.

4 Conclusion

Overall the Huffman compression algorithm we think has practical use across many fields requiring big data. With bioinformatics continuing to reveal more about complex biological processes, there will be plenty of moments where lossless compression could help in sharing compressed data. Also by binarizing data, we also see applications in security by being able to securely hide sensitive patient data that cannot be easily brute forced. We are excited about our program and its capabilities and can't wait to see where more computer science algorithms can play a role in not just bioinformatics but the world.