Tony Diep
4-24-17
CS 2420
Analysis Document

## ABSTRACT

Huffman encoding is one great example of how this class focuses on how data structures and algorithms correlate with the concept of “efficiency”.  Given a text file of a particular size in terms of bytes, we want to reduce the amount of bytes and still maintain the same data initially.  Many text files in the real world contain immense quantities of data and the amount of time and memory to process certainly can take a while.  Although I struggled to understand conceptually huffman encoding, it is nonetheless a proficient mechanism.  But the mechanism is not possible without the use of the Huffman Tree.  The Huffman Tree is a type of tree similar to a Binary Search Tree and a Heap in that it utilizes a priority queue.  The idea of the Huffman Tree is to order nodes based on how frequent a symbol occurs in a text file and place it on the top rows of the tree, whereas the low frequency symbols are placed in nodes that are on the bottom rows of the tree.

## TIMING AND GRAPHS

For our experiment, we are to test the optimal number of words to obtain the best compression.  To do this, we decide to test all number of top most frequent words starting from 0 to 100 inclusive and time how long the compressing and decompressing processes take in milliseconds.  For each different most frequent word count, we allocate a HuffmanTree object and pass in that exact word count.  We then put timing variables surrounding the compressing call while also turning the flag to “true”.  Once we complete that step, we change the flag to “false” and then put timing variables to time the decompression method. We then finally record the results. This experiment empirically tests to prove or disprove our following prediction:  As the number of top most frequent word counts increase, the amount of time for Huffman encoding to compress and decompress will be approximately similar to each other.

To conduct the experiment, we first keep track of start time and end time before beginning compressing a text file, then we do the same for after compressing a text file.  Similarly, we do this process for before and after decompressing a text file.  Then we do this process for each different frequent word count starting at 0 and going until the word count is 100.

The following graphs show the results for each of the 7 text files provided with the assignment, which starts on the next page:

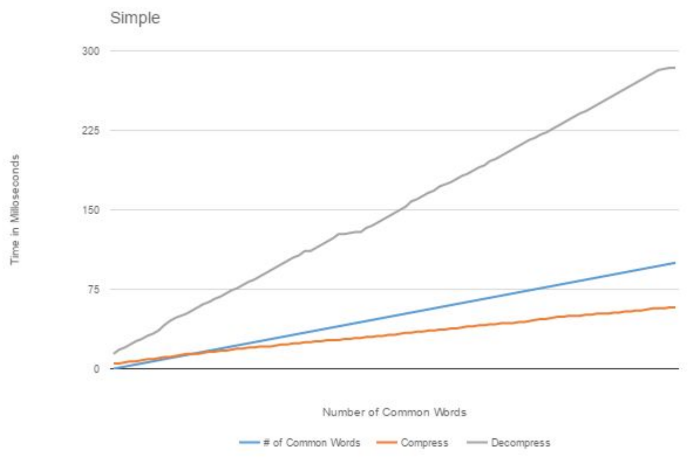

Figure 1: Simple

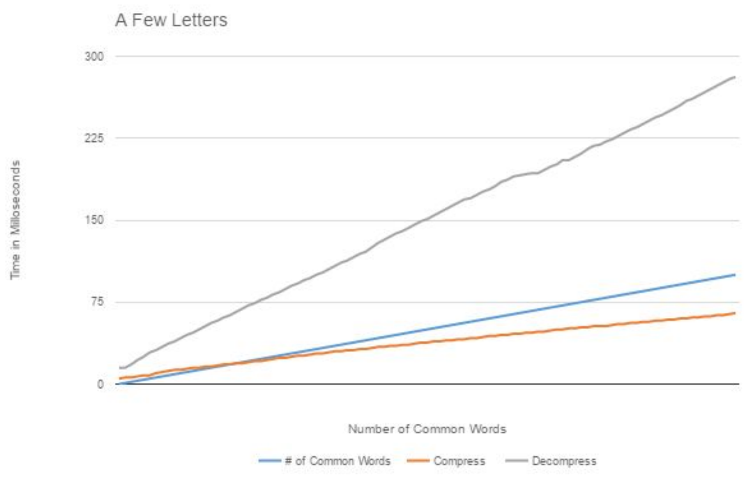

Figure 2: A Few Letters

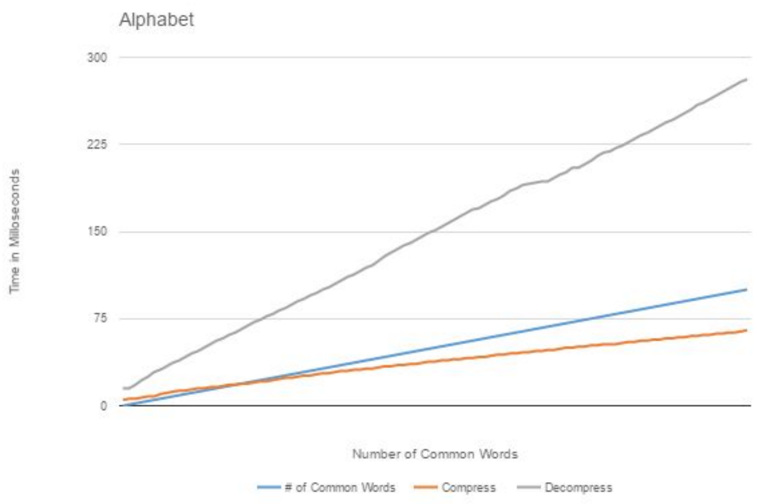

Figure 3: Alphabet

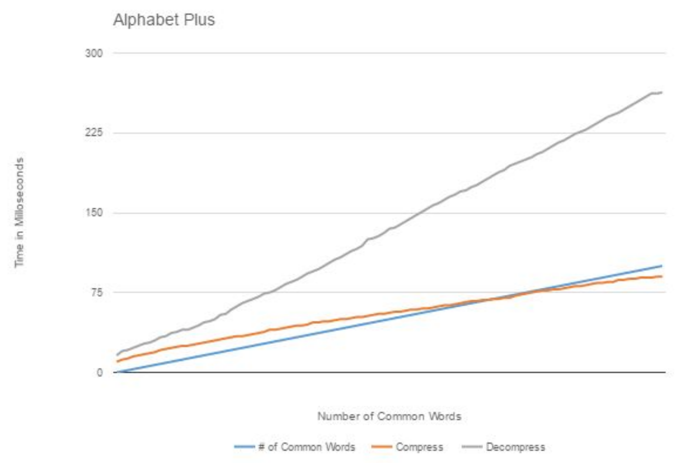

Figure 4: Alphabet Plus

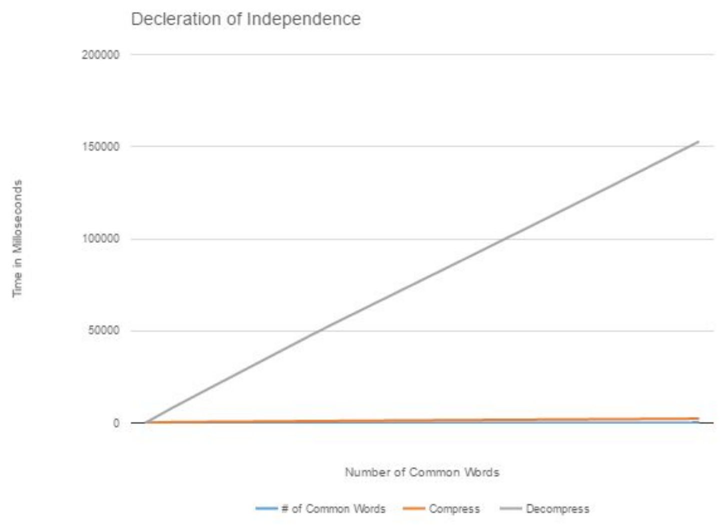

Figure 5: Declaration of Independence

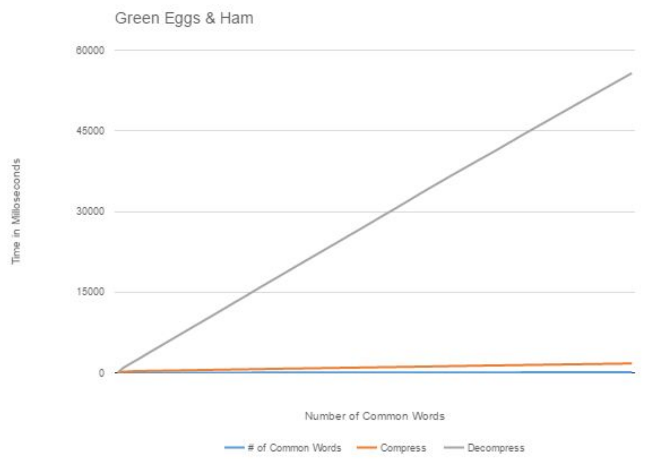

Figure 6: Green Eggs & Ham

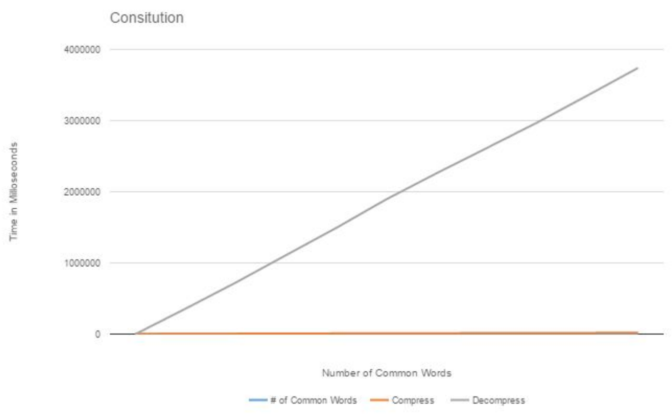

Figure 7: Constitution

From the results shown above, most of the text files took approximately close to linear (if not exactly linear) to compress and decompress after Huffman encoding. Comparing between the compression and decompression time for each text file, the decompressing process took significantly longer. This is because compression simply takes all of the symbols in the text file and essentially “summarizes” each representation of each symbol. In other words, the symbols in the text file are represented in a certain amount of bytes by ASCII code, but huffman encoding’s data compression assigns frequent symbols shorter bit patterns. On the other hand, infrequent symbols are all represented with longer bit patterns. As a result, when compressing a text file, we get a human unreadable text file that the computer can understand. However, the decompression requires a few more steps compared to compression. Thus, we can say that our prediction was not quite aligned to the actual results displayed on the graphs.

Because we can infer that decompression takes significantly longer than compression, we are unable to provide a graph for the “two_cities” file since that file is immense in terms of byte size. Although it is most likely unknown to predict the shape of the graph for “two_cities”, we can sort of hypothesize that the decompression and compression lines will be much steeper compared to the other text files. From this empirical analysis, we can say that the size of the text files and the compression time are independent from each other but has some correlation for the decompression time.

**SOFTWARE ENGINEERING**

This assignment took us between 18 and 22 hours to complete in which I anticipated.  This was the assignment I had the most fear out of because I did not fully understand how Huffman encoding works. There were certainly a lot of bugs in our code since we generally focused on the ballparks of the code by following provided pseudocode.  At first, we had the Huffman Tree set up correctly but the frequencies did not correlate with the actual frequencies in the text files.  Additionally, our compression and decompression methods worked incorrectly since the output huffman and uncompressed files turned out slightly different than expected.  I had one “non-heap” idea I learned after working on this assignment: use sorted lists.  Using sorted lists in essence act like heaps (specifically, a min-heap if choosing to sort every symbol in ascending order) without actually implementing a heap.  This will allow proficient building in terms of the Huffman Tree.

There were numerous advanced java data structures used in this assignment.  The top three data structures I will discuss are the priority queue, hash table, and bytebuffer.

The priority queue is a powerful data structure regardless of its simplistic structure.  It is quite simple to understand, utilize, and implement.  The essential idea about the priority queue is it is similar to a queue, where the queue is a FIFO data structure, which follows the “First In, First Out” principle.  However, the priority queue differs from a standard queue in that an element is firstly added based on a highly prioritized elements.  In other words, any elements that have higher priority are inserted out of the queue first, then any subsequent elements with lower priority are then inserted next.  Priority queues are the workhorse behind constructing all types of tree data structures, including the Huffman Tree and Heap.  In the Huffman Tree case, all nodes’ priorities are based on the frequencies of each symbol.  Thus, any nodes with higher frequency symbols are placed on the top rows of the tree and lower frequency symbols are placed on the bottom rows of the tree.  In the Heap Tree case, all nodes are inserted to build the tree based on exactly where they are positioned in the array.  This means any nodes that are at the beginning of the array are added first as opposed to nodes at the end of array, which are added last.

The hash table is also an efficient data structure with adding, searching, and finding complexities being on average O(C).  The hash table is good for inserting lots of elements which in our case was the different nodes representing each different symbol and their frequencies in a text file.  It served a great purpose for the Huffman Tree class to provide an efficient alternative to finding a particular node in the Huffman Tree; traversing down on a Huffman Tree in our assignment was certainly challenging without the hash table.

The byte buffer, although quite hard to understand initially, is an important data structure since it allows us to abstract the conceptual expectations for understanding bits and bytes.  This advanced data structure allows us to easily manipulate and work with bytes, which are the byproduct from converting each symbol in the text file.  Without using the data structure, we would have to provide our own methods for various bit and byte operations such as shifting, retrieving, and searching for a particular byte.