Landon Crowther

U0926601

04/21/2017

CS2420

Huffman Encoding

Partner information: When I first started this assignment, I had found a partner from the discussion page on Canvas. His name is Jacob Zenger. We met up early in the week and worked on the assignment for about an hour or so. During this time, all we could accomplish was 1 partially completed method and a few tests for the Node class; resulting in little-to-no progress. After this original meeting, I realized that there would be a couple problems with our partnership. The first was that our schedules did not seem to align at all. When I was available to work, he wasn't, and vice-versa. Secondly, after an hour of working with this partner, I realized that our level of understanding of the assignment was completely different. He hadn't been to class for a while, and had little-to-no understanding of the assignment and its requirements. After this initial meeting, I politely told him that I will no longer be partnering with him, as I had a very busy schedule and could not afford to spend the majority our time explaining the assignment to him, rather than coding. I then reached out to a past partner of mine, Brent Collins. Brent was willing to start the project as partners, and him and I could successfully complete the project.

The most important thing I learned from this assignment is how a complicated project can be broken down quite simply with proper class and method structure. There is a lot of work necessary for proper compression/decompression; however, if each method is taken one step at a time, and the flow of the methods is clear, each step in the process becomes much simpler.

This assignment gave me a better understanding of how data compression works. I've always known that compression exists, but never had a clue how it works. Although Huffman compression may be limited to specific types of data, it gave me a good understanding of one example of data compression. Looking back on the project, Huffman compression is a simple yet clever way of compressing data.

A Huffman tree works by placing all the symbols in a maximum priority queue, based on their frequency. This makes it so the most frequently appearing symbols are close to the root of the tree. Once this tree has been constructed, all the symbols are given a code, based on their position in the tree. The code is a sequence of "0" and "1", which can be written in byte notation once the compression is complete. In addition to constructing this tree, a header must be built. This header is used to represent a list of all the symbols and their frequencies in the file.

Huffman compression is a good algorithm for basic compression, however, I do not believe it is the most efficient compression algorithm.

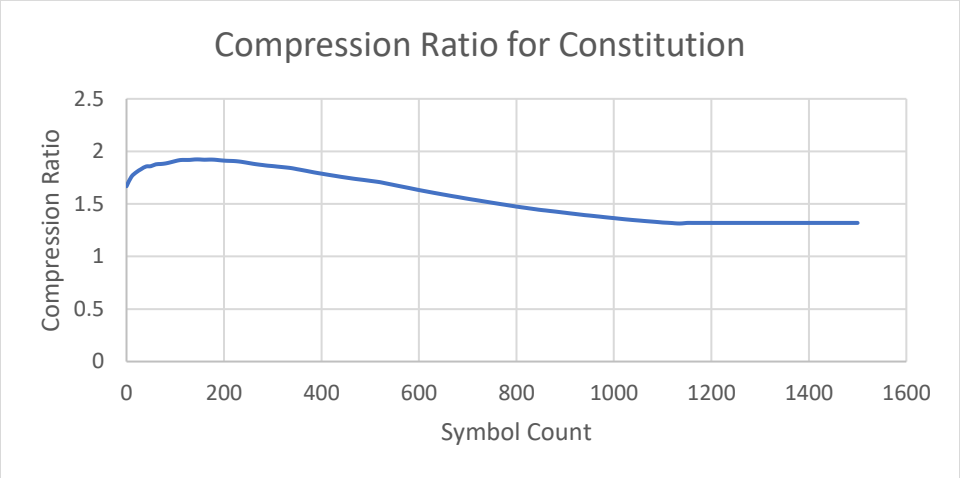
Analysis:

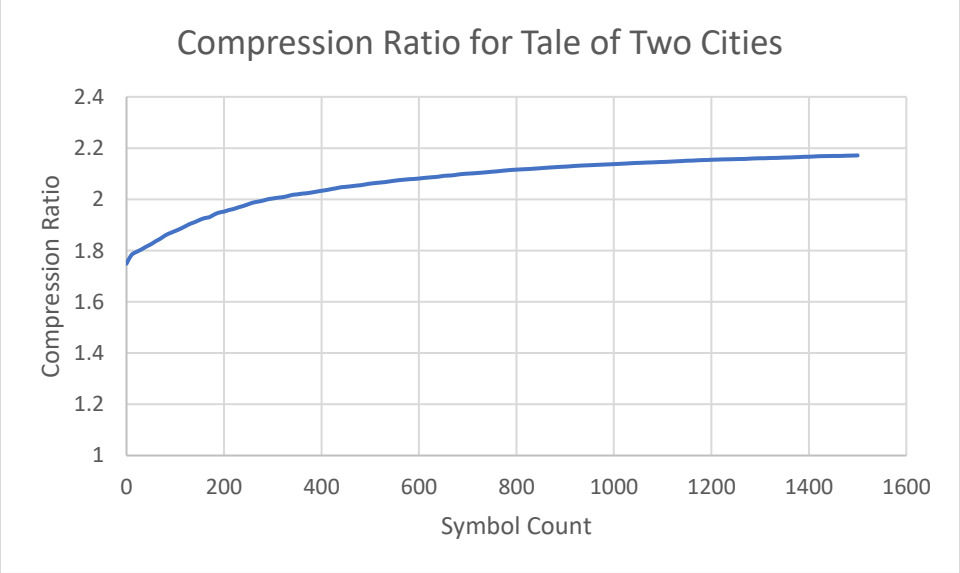
To test the efficiency of our Huffman compression algorithm, I studied the effects of symbol count on the compressed file size. To accomplish this study, I had to use a common metric for compression performance. I used compression ratio as this metric. In my case, compression ratio can be defined as follows:

$$compression\ ratio = \frac{compressed\ file\ size}{original\ file\ size}$$

Where file size refers to the number of characters in the file. In the case the case of the compression ratio I have defined above, the lower the compression ratio, the better.

As I was writing the tests, I realized that the actual file that would be used was very important. In some documents, there are words that are repeated multiple times throughout the text; whereas in other documents, there may be a lot of words that are only used once (i.e. a dictionary list). To illustrate these effects, I ran the compression ratio test on the constitution document, as well as Dickens' *Tale of Two Cities* document. My results are as follows:





In the case of the constitution, a larger symbol count resulted in a lower compression ratio. In fact, a symbol count of 1130 resulted in the lowest compression ratio. On the other hand, increasing the symbol count for *A Tale of Two Cities* seemed to decrease the effectiveness of the compression algorithm. I would hypothesize that *A Tale of Two Cities* had significantly less repeated words than the Constitution; therefore, increasing the symbol count had no beneficial effect on the compression ratio.

Software Engineering:

This assignment took me 18-20 hours. Although most of the code was straightforward, it took me a while to get used to the bit notation, as I have never worked at the byte level or worked with BitSets before. Additionally, this project seemed complex at first. It wasn't after it had been broken down into smaller pieces that everything finally "clicked" with me.

Data Structures:

Priority Queues:

The use of a priority queue turned out to be super helpful for this assignment. Priority queues work by storing all the data in a tree-like fashion, where the highest "priority" data is near the top, and the data that's furthest from ideal is towards the bottom. This allows the user to easily access the top elements of data, without having to sort the data. The data doesn't have to be sorted because it is arranged properly as the data is inserted to the priority queue. Priority queues have constant insertion time, which is helpful when many elements need to be inserted (the exact case for this assignment).

Hashtables

Hashtables work by storing a key-value pair. In our case, the key was the symbol, and the value was the associated node. One important feature of hashtables is that duplicate keys are not allowed. In our case, if a duplicate key was ever found, the frequency of the associated value (the node) was incremented. This functionality allowed us to add all the elements to a hashtable, and this hashtable acted as a histogram for us, keeping track of the frequencies of all the symbols.

BitSet

A BitSet was used in this assignment to work through the collection of bytes that represented the header and symbols for the compressed file. The BitSet was helpful, because the BitSet class had useful methods that made working with an array of bytes simple. In theory, this entire assignment could be done without BitSets (and therefore using byte[] commands instead); however, it would have proved to be much more difficult. For example, the BitSet class could keep track of the position within the array without having to keep track of an index, like one would with a for loop.