Lab 03 Huffman Encoding Tree Analysis

Jung Woo J Jeong

When given a limited space for usage, a simple and basic rule applies. That is, being able to make data smaller allows you to fit more data and use the space better. This is the simple but powerful motivation of compression. Using an analogy of filling in a glass with different sizes of stones, the key is not leaving as much free space as possible. We need to find out how many of which size of stone are given so that we can fit them properly wasting as little free space as possible in optimal space usage. This is exactly the logic Huffman Encoding is using. Given frequency of elements, we assign proper size of data into elements so that there is little space wasted.

The key to implementation here is to assign the biggest form of data to the most frequent element and vice versa. Accordingly, sorting is essential. In this implementation, sorting has been constantly maintained using minimum heap tree that keeps the smallest or the least frequent element on the root node. Having achieved this far, Huffman Encoding Tree is constructed by repeatedly creating a binary tree with height of 1 in ascending order and inputting it again to the min-heap tree so that it will continuously get maintained to be sorted while making parent-child connection to each other. Continuous execution of such process eventually creates a binary tree with the least frequent element, as a leaf node, to be positioned to the farthest from the root and the most frequent element to the closest from the root. Personally speaking, while reading on it did not get me into it much, having implemented it really got me amazed on how elegant and clean the solution is.

The answer to whether any useful data compression has been achieved or not given frequency table correctly reflect each character's number of appearances is yes. Intuitively, as mentioned above, assigning smaller data to the more frequent element makes sense to expect a better compression than all equal data assignment. To check this for certain, I have experimented this by using a frequency table with equal frequency of 16 for all letters, which is the average frequency from given frequency table. Looking at the encoded strings and compare results from different frequency table, I have noticed that results using original frequency table were shorter. You can see this result in *outputs/comparison.txt*.

While it is clear that frequency-sensitive data distribution results in better compression, we can now argue with how it would be different when using a different scheme for a tie-breaking. Currently, tie-breaking priorities are, first, smaller number of letters getting priority, and second, is alphabetical order. This can be determined by looking at how this scheme would affect the data size. Looking at C and BD, original scheme would result in C to be on the left and BD to be on the right and further making B and D larger data size than C. On the other hand, when alphabetical order is priorized, the ordering would be BD and C and further making B and D bigger data size than C. Looking at data-length-only, two cases do not seem to make difference. However, what changes in this case are whether the changed point would have either 0 or 1. And depending on how 0 and 1 is differently sized, they do make difference. And in this case specifically, if we assume 1 is bigger than 0, BD from earlier is bigger than BD from later.

While working on this project, data structures that have been used are min-heap binary tree, linkedlist, recursion, iteration, and stack, which was taken out later. Min-heap binary tree was firstly used because we needed to maintain ascending order with the smallest node on top.

And then, in the process of building Huffman Encoding Tree while taking two smallest nodes and combine to create a binary tree, linkedlist with left and right children is used to maintain nodes connected. Recursion is used when preorder traversal is being done. Iteration is used multiple times whenever node comparison with either children or parent is necessary and swap with each other usually with while loop. Lastly, although this was taken out, I think it is worth mentioning the usage of stack because I was stuck. Initially, I attempted creating Huffman Tree using the Stack while maintaining pointers on the array manually. This required incredible amount of array size and unnecessary and unused space in the array. While attempting on this, my idea was to collect mini-binary trees in the stack and pop out one by one to create the Huffman tree. Although the architecture is largely inefficient in memory allocation, I did manage to complete it and obtained a correct result. However, inefficient memory allocation really bothered me and later changed the direction in using Linkedlist approach, which is much cleaner solution.

Speaking of memory allocation, we can calculate performance in Big-O when looking at each segment of implementation. Those segments consists of building min-heap and building Huffman tree out of two pops and merge process. Given the number of element as N, a single Heapify costs O(logN) on average while checking parent node for size comparison. And since this is conducted N times, overall complexity on building minheap is O(NlogN). On the other hand, looking into building Huffman Tree, one pop that requires restructuring, another pop with restructuring, and putting back a combined node that entails another restructuring costs 3O(logN) because each iteration cost O(logN). And because each cycle reduces the tree by one, reducing it to eventually 1 node means that there are N iteration of the cycle occurring. Multiplying these two, it would be 3O(NlogN). Now, combining both minheap complexity and Huffman tree segment complexity, it would be O(NlogN) + O(NlogN) = 2O(NlogN), which gives in conclusion a value of O(NlogN).

As enhancements, there are six things I have added in total:

- Unit Tests for Significant methods: They cover 4 different classes with total number of 20 Tests. When executing test command instructed in README.md, I have made it so that executing one would trigger all available tests.
- Made available 6 additional frequency tables and included 5 of them in test and 1 of them specifically for experiment in compression comparison.
- For encoding and decoding, I have made characters that are not available in frequency table to be included as it is without stopping the execution. This is also a error handling. I have contemplated how it would be simple and intuitive for the users to recognize it and concluded that including/concatenating it as it is the most straight-forward way. Of course, since this is not a bit-wise expressed and this would not be usable in machine context.
- For the output, I have included the processing time for encoding as well as decoding in nanoseconds. They are printed out as well as included in output files. I have initially attempted it in milisecond, but it ended up only showing 0 and 1 because the value was so small, so I went for nanoseconds to make the number more exciting.
- As briefly mentioned above, I have made an efficiency comparison as an experiment. This specifically uses the frequency table file of `outputs/FreqTable_unix_equal_distributed.txt` and the result is in file `outputs/comparison.txt`.

- I have improved a method of printing out `preorder traversal` and `letter table of encoded values` so that they not only echo out in terminal but gets generated as files. They are available in outputs directory.

In closing, this Lab was very challenging and exciting to work on. There was a major struggle when I was stuck in design choice of using Stack instead of Linkedlist but even this was very rewarding when I figured out things and made it work. In addition, because I have struggled and looked through the architecture while debugging very thoroughly, I was able to really grasp of the logic behind the architecture better than just reading. While working on it, major concern in the improvement was mainly focused on memory allocation and performance. In this sense, what I would do differently next time is to add a feature where a program can generate a frequency table out of a specific text so that we can be more dynamic on alphabet frequency matters. Besides, I would add a dynamic feature to be able to add characters to the frequency table when recognized that it is not present.