Alejandro Santacoloma

CDA3101

April 25, 2018

Huffman Algorithm Discussion

The purpose of this project was to understand the mechanics of the Huffman algorithm and how to implement it. This assignment illustrates the utility and efficiency of the Huffman algorithm and the Huffman tree it implements, the created Huffman encoder class is able to read a file and generate all the frequencies of each character in the read file, after which it can then encode the file into a compressed format that requires less data to store the file.

Data Structures Used

The data structures used in the implementation of the Huffman algorithm consisted of an array that stored the frequency values of characters found while parsing through the file, the ascii value of the character correlated to the index of the array. A HuffTree class was created to contain and use the HuffmanTreenode class, the HuffmanTreenode class consisted of a simple node that contained a variable named weight that was based on the frequencies of the elements contained in or related to the node, a Boolean to check whether or not the node was a leaf node or internal node of the Huffman tree, and pointers to the left and right children of the node if it had any children. In order to build the Huffman tree a priority queue was implemented to contain and sort the individual nodes and Huffman trees based on their weights. Encoded and decoded files are returned as strings by the HuffmanEncoder class.

Computational Complexities

All time complexities of methods used are in bold and inline with the method, the text following each method is to show my logic and work for how I reached the calculated time complexity of each method.

getFrequencies() : O(n)

The getFrequencies() method parses through all n characters once in the file it reads, it also stores all n characters and their respective frequency values once, therefore having a time complexity of O(n).

(n + n) -> O(2n)-> O(n)

buildTree() : O(nlogn)

buildTree() calls on the getfrequencies method once at the start of the method, it then implements two separate while loops that are not encapsulated in one another, the a priority queue is implemented inside one of the while loops however, therefore the time complexity of the bulidTree method is O(nlogn)

n + n*(logn) + n -> O(nlogn)

encodeFile() : O(n^2)

This method contains a for loop that goes through all the characters in the current line obtained from the file, this for loop s enclosed in a while loop that runs until all lines from the file have been parsed through. The time complexity of this method can be interpreted as O(n^2) since in the worst case scenario, the while loop runs n times and the for loop runs n times as well, resulting in O(n*n) or O(n^2).

decodeFile() : O(nlogm)

This method implements two methods, one of the while loops is nested in the other while loop, the outer while loop will run n times, n being the length of the string variable named code, the inner while loop will run at worst logm times each time it is called on, m is the number of nodes in the Huffman tree, the inner while loop traverses down a path of the Huffman tree and at worst the length of the longest path of a Huffman tree is logm, therefore the time complexity of the entire method is O(nlogm).

n characters * logm nodes travelled in tree -> bigO is n(log(m)), n characters and m nodes

traverseHuffmanTree() : O(n)

The traverseHuffmanTree method implements one while loop that runs n times, n is the size of the array has a time complexity of O(n).

maptree (n) + n -> O(n)

mapTree() : O(n)

This method implements a recursive algorithm that performs a breadth first search of the huffman tree, since each node is only visited once, if there are n nodes, then the time complexity of this method is O(n).