Huffman Encoding System

Joshua Hodges - joh55@aber.ac.uk

6th December 2016

Contents

1 Introduction

For this assignment, we are given a message of characters and symbols and are instructed to encode this using Huffman encoding for optimal results. We have to determine the frequency of all the characters and symbols in the message and then sort them by frequency. The two lowest frequencies are then combined to create a Huffman node with a new frequency. This should be repeated till there is one symbol left and this is delegated as the root. The root is then used to create the tree and assign several bits e.g. 01011, to all the nodes. This is then used to encode the letters and work out the statistics of the compression which includes sizes before and after the compression, the ratio of compression, the height of the tree, the number of nodes and the average depth of the tree.

In my implementation, the program covers all the required specifications and prints out all the required information to the terminal. This is achieved using arrays, PriorityQueues and objects that I created. The file handling aspect of the code was straight forward and didn't take up too much of my time, however finding a way to store the frequency of carriage returns was what lead me to use an array with the size of 256 to represent the ASCII representation of each character and symbol. The ASCII array did make it easier to store all the characters and frequencies for later use. The most challenging part of the assignment was iterating through the linked list to either draw the tree, assign bits or calculate the total number of nodes and the average depth of the tree. This was because the links went two ways rather than one, so I had to iterate through one side then the other using recursion. During the assignment, I had to research more into linked lists and PriorityQueues. In the end the program works like it is intended to and performs what it needs to.

2 System Design

The system I plan on implementing will take input from the user and use a file with that name. The file will then be read into a string. The string will be iterated through as a char array to count each character. An int array with a length of 256 for the size of the extended ASCII character set. This array allows the encoding of carriage returns. For each occurrence of a character in the char array the location in the array relevant to the ASCII representation of the character is increased. This array will then be used to create nodes which will be inserted into a priority queue. The nodes in the queue will be removed in pairs and a new node created with a link to the two nodes that were removed and their combined frequency. This will be done till there is one node left for the root of the tree. Bits will then be assigned accordingly to each node. The calculations will then take place and the results will then be displayed. This is how I plan to implement my system.

3 System Implementation

3.1 Overview

My program is split into 5 classes including the main class, the classes excluding the main class will be discussed in detail following the overview. These classes are the FileHandler class, the TreeHandler class, the HuffNode object class and the Compress class. One of the last classes I created was the Compress class and this is used to do all the calculations and store these values rather than the main class.

3.2 FileHandler Class

The Filehandler class has an object that stores the brute-force fixed length size of the file, all the characters and symbols of the file in a string and a int array called ASCII that stores the frequency of the characters and symbols relative to their ASCII value. To store the file as a string I used an InputStream, BufferedReader and a String Builder to read the file line by line and add it to a string. This string was then iterated through by casting it as a char array and taking the value of every char and increasing the relative position in the ASCII array by 1. The ASCII array would then end up with frequencies at the position of each characters ASCII number. To find the size of the file is used logs to find the minimum number of bits needed to encode using brute-force fixed bit length encoding. To find the power you log the amount of unique characters and divide it by log(2) and round it up to the nearest whole number and then multiply this by the length of the string. The rest of the code is made up of generated getters and setters.

3.3 HuffNode Class

The HuffNode class is the constructor for the nodes that form the basis of this program. The object itself contains 4 variables, the stored character, the characters frequency and if it has a left and right child. this forms the base for every node. Along with the getters and setters in this class there is a debug/test function that draws the Huffman tree in the console to check that it was assigning children correctly, a function to compare two nodes and a function to check if a node was a leaf. This function proved valuable when iterating through the tree/ linked list. The assignment of the Bits are dealt with in another class.

3.4 TreeHandler Class

The primary function of this class is to instantiate and populate the PriorityQueue. The buildTree function of this class takes in the ASCII frequency array from the FileHandler object and iterates through it. While iterating through it checks if any of the positions in the array is greater than 0 and creates a node. The character of the new node is determined by casting the position in the array into a char, the frequency of the node is the number stored in that position and the left and right child is set to null for now. Proceeding that in the same function, while the queue is greater than 1, it removes the two least frequent nodes combines their frequency and creates a parent node with no character, the combined frequency of the two nodes and sets its children node to the two that were removed. The new node is the added back into the PriorityQueue. This is then repeated. The last root node is then used to reference the linked list.

3.5 Compress Class

This is the final and largest class in terms of code. This class defines an object which contains and instantiates the FileHandler object, the root HuffNode and the TreeHandler object. This object also contains the variables which store: the uncompressed size, the compressed size, the compression ratio, the tree height, the number of nodes and the average depth of the whole tree. These are then used in the final print to the console. The printStats function is the one which takes those values and nicely fits them into sentences and prints it out. The avgHNode function checks the height of each leaf and adds them together so that it can be divided by the number of nodes. This is done by the countNodes function. The treeHeight function recursively checks the height of each leaf and returns the largest. The getSize function multiplies each leaf bitPattern size by the frequency, this returns the size of the compressed text. The buildCode function assigns 1's and 0's to the left and right child recursively, this is used to assign the bitPattern to each of the leaf nodes.

4 Time and Space Complexity

In the program, the data structures used are Priority Queues and Arrays. Regarding the array I think I could have used a better data structure to store my frequencies with regards to space. However arrays are quicker to access than stacks and queues however takes longer to insert and delete. They all however require the same amount of time for searching. Below are the average and worst cases for time and space complexity for each of the data structures that where used. This was found on http://bigocheatsheet.com .

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

5 Testing

most of the testing done on the program was done during the coding. using prints to display if frequencies were being tallied up correct and having it print out the built tree. They where used to brint out the bits that were assigned to each character and ASCII array was printed out because there where quite a few null pointer errors during development.

Below are screen captures of the results of the program being printed out in the console using the two test files that where used on the discussion board.

```
What is the name of the file you would like to process
5.txt
The file size is 48.0 Bits
The compressed file size  is 36.0 Bits
The compression ratio is :1.3333334
The hieght of the tree is :4.0
The tree has 13.0 nodes.
The average depth is :2.6153846
```

Below is the function I used to draw the tree to check that children where being assigned correctly.

```
What is the name of the file you would like to process
4.txt
The file size is 12288.0 Bits
The compressed file size  is 9216.0 Bits
The compression ratio is :1.3333334
The hieght of the tree is :4.0
The tree has 13.0 nodes.
The average depth is :2.6153846
```

```
public static void drawTree(HuffNode n, String dashes) {
    if ((int) n.getCh() > 0) {
        if (n.getCh() == ' ') {
            System.out.println(dashes + n.getFreq() + ":" + "Space");
        } else {
            if (n.getCh() == '\n') {
                System.out.println(dashes + n.getFreq() + ":" + "Cartridge Return");
            } else {
                System.out.println(dashes + n.getFreq() + ":" + n.getCh());
            }
        }
    } else {
        System.out.println(dashes + n.getFreq());
    }
    if (n.left != null) {
        drawTree(n.getLeft(), dashes + "-");
    }
    if (n.right != null) {
        drawTree(n.getRight(), dashes + "-");
    }
}
```

Below is the output of the program when compressing shakespeare.txt which considering how big the file was, compressed in a short amount of time.

```
What is the name of the file you would like to process
shakespeare.txt
The file size is 3.8207384E7 Bits
The compressed file size  is 2.5257996E7 Bits
The compression ratio is :1.5126847
The hieght of the tree is :23.0
The tree has 181.0 nodes.
The average depth is :9.447514
```

6 Conclusion

During this assignment I had learnt more about how to use priority queues and how they work. I had also learnt more about linked lists and how they can form a tree of nodes which was the basis of this assignment.Using the code to compress the given text files took quicker than expected, such as when reading in shakespeare.txt . Overall the program meets the functional requirements and prints out the correct statistics that is required. If i had to do it again i would most likely find a different way to store the frequencies rather than using the array. Some of the array is populated by 0, which still takes up the space of an int even though its not used. This could be a problem if the system it has to run on has space restrictions. I would also try and process the file line by line rather than storing the whole text file in one string and this could prove problematic when trying to compress larger text files or multiple text files at the same time. Comparing the two methods of encoding, fixed length and Huffman, it really depends on the constraints of the machine. Huffman is useful if the space on the machine is more valuable and the fixed length being for straight forward would take up less processing time.

7 Self Evaluation

Overall I think that all the functional requirements where met. However I think I could have gone beyond and improved the user interface having the program use a gui. This could have enhanced the user experience. If I had to implement this system again I would have done it in a simpler language such as c or python. I would have used a different data structure for the storage of the frequencies such as a binary search tree, that would have however required a lot more research on my part and would not have fit in the time constraint. I would have also better constructed my classes and packages. With all that said the program functions as needed and should atleast achieve 70