Colin Harfst

cph5wr

Lab Section 103

26 November 2016

Post-lab 10

To analyze the time and space complexity of my Huffman encoding and decoding algorithms, I will examine the main methods that encode and decode. I will begin by analyzing the encoding.

The main method for Huffman encoding begins with a few constant time operations that prepare the command line parameter to be read in as a text file. A 1 byte char is created. A vector of integers of size 128 is created. That is an additional 4*128=512 bytes. Within a while loop, each character within the file is read in. This operation is dependent upon the size of the file, so we will say it is linear complexity or Θ(n) where n is the number of characters in the text file. A heap is then created. The heap is filled with up to 95 nodes. In terms of space, each node takes up 8+8+4+1=21 bytes (for two pointers, an int, and a char). 21*95=1995 bytes. At this point, we have used a maximum of 2508 bytes. This inserting operation is Θ(95*log(s)) where s is the size of the heap on average. This is because inserting a node into a heap uses vector.push_back a constant time operation and calls percolateUp, a method which runs in time log(s) on average where s is the size of the heap at a given point. At this point the time complexity is Θ(95*n*log(s)). There is then another while loop that runs in time Θ((s-1)*log(s)) since the loop iterates (s-1) times and performs an insert each time, an operation that is time log(s). Note that at this point the memory complexity hasn't changed, we have just rearranged where the nodes are being stored (as children). We then traverse by the newly created Huffman tree to read out the binary results. This method is recursive and will recurse an average of log(s) times as we are again operating through a tree. So at this point, our memory complexity hasn't changed, and the time complexity is Θ(95*n*(log(s)^3)*(s-1)). Two integers are then created to increase the total number of bytes to 2516 bytes. The file is then read through again and the results of the Huffman tree are used to print out the desired binary strings. This operation is linear with respect to n, so Θ(95*(n^2)*(log(s)^3)*(s-1)). We finally create two doubles to represent the ratio and cost of the tree and encoding. This is an additional 16 bytes, for a total of 2532 bytes needed. These are the final values of complexity in terms of time and space.

The main method for Huffman decoding also begins with a few constant time operations that prepare the command line parameter to be read in as a text file. A 21 byte node is created. A while loop reads through the first section of text and creates a Huffman tree. This loop iterates n times where n is the number of unique characters in the first section of text (including spaces and symbols). The function addToTree is then called. This function calls recursively an average of s times where s is the average number of binary values corresponding to each character. So we have Θ(n*s). Note that n and s are slightly different in this context then they were in the encoding process. Another while loop reads in the second section of the file and runs in time r where r is the number of characters being read in. Two strings are then created. The string sizes are dependent upon the size of the binary string read in, but we will declare these two string sizes to be of size p and q bytes, so we currently have Θ(n*s*r) and 21+p+q as the time and space complexities. Within a final while loop, the strings are modified and traverseTree is called recursively. This function also runs in time s. So finally, The total space complexity is 21+p+q and the overall time complexity is Θ(n*s^2*r) where n, s, and r are defined as above. (Note that it might be better to think about s as log(m) where m is related to the number of nodes, but this is taken care of whit how I defined s above).

Thus, the overall complexities of both main methods have been determined. Th