Data Structures

EN.605.202.81.SP24

Lab 3: Huffman Encoding

Logan Griffin

4/16/2024

Analysis

Compression Effectiveness:

Huffman encoding is most effective when the frequencies come from the item that needs to be compressed. That way, every pixel, letter, code, whatever it may be, is accurately represented and receives an appropriate length Huffman code. For the item needing to be compressed to also serve as the key, the receiving entity/user would need to already have either the file to build the frequency table, or the frequency table itself in order to decode the Huffman encoded message. For this lab, a generic frequency table was used. Since the source of the frequency table is unknown, it is impossible to determine exactly how well it would compress the random messages. The messages from the clear text file were short, allowing a quick look to give a reasonable gauge on compression effectiveness. For example, the most frequent character from the table was the letter “e”. Looking at a sentence and seeing a lot of “e”’s would signal that the encoding may have useful compression.

Given that each letter is reduced to a one or zero, each character of the encoded message only requires one bit of storage. This is advantageous over other encoding methods that convert letters or other complex characters into other forms that still require a byte of data, instead of a bit.

If alphabetical ordering was given precedence before the number of letters in the key, the letters would be scattered around in the tree and not located at all of the leaf nodes. The leaf nodes, or collection of the highest priority items, would be any complex leaf containing the first letters of the alphabet. Logically, this seems like it could increase the effectiveness of the algorithm by mapping the singular characters higher up in the tree. If the single letters were closer to the root node, they would ultimately have shorter codes due to less traversing.

Structures:

This lab combined multiple data structures. An array was used to keep track of tree nodes representing a heap. The array based tree form is easy to use because any index can be reached without much work. If the array was not used, there would be added tree traversal and additional pointers, lengthening the run time. A parent node in the tree could easily be found by dividing the current index in half. The left and right child nodes could be found by multiplying the current index by 2 and multiplying by 2 and adding 1, respectively. One drawback of using the array is that the size is not easily changed. The heap array was created with a size of 50, but, if there were more than 50 characters in the input file, the program would fail to run. It would be beneficial to implement a dynamic array in order to avoid index out of bounds errors.

The heap array was specifically a minimum heap. While maximum heaps are more common, a minimum heap was more appropriate for this implementation. Max heaps maintain an order such that every parent node has a greater key than its children. Comparatively, a minimum heap maintains an order such that every parent node has a lesser key than its children. In this case, the lesser key corresponded to a lower frequency, which was ultimately a higher priority.

Using a heap array represented a priority queue. Each time a node was entered into the tree, it was percolated down to its appropriate priority position. This was extremely helpful when building the Huffman Tree because you could easily remove the highest priority items by removing the root node.

Implementation and Design:

The purpose of this project was to utilize a frequency table to build out a Huffman tree that could then be used to encode and decode messages containing the 26 letters in the alphabet.

The program works by first taking the file paths for 3 input files and 1 output file. The three input files must be referenced and read in a specific order. The first file should contain a frequency table with all of the characters that need to be assigned a code in the Huffman tree. The second file shall include clear/plain text that can be read and encoded, while the third file shall include encoded messages that can be read and decoded.

The frequency table is read in assuming the file contains a different character on each line in the form of letter, space, separator, space, frequency. Each frequency is read starting at the 5^{th} index in the string containing the entire line. This allows it to capture multiple digit frequencies, rather than expecting them all to be of the same power of 10. After each letter and frequency is read, a node is created and inserted into the next available index in the heap array. There, it is compared with its parent and swapped until it is the child of a node that has a higher priority.

Once the heap array has been populated with all of the individual character nodes, the program then begins assembling the Huffman Tree. This is achieved by removing the root node twice, combining the data, and combining the priorities. A new node is created with the combined attributes, with the first node that was removed as the left child and the second node that was removed as the right child. Finally, that node is inserted back into the tree, where it goes through the same comparisons to its parent. This process of removing two and adding one back is repeated until there is only one, root node left in the array. That node will store the root data of all the characters combined, and have a priority equal to the sum of all the priorities. The left and right child references can be employed to reach any singular character if you start at the one giant root node.

After the Huffman tree has been built, the program prints out the code for each letter by looking for leaves and adding the appropriate digit (0 for going to left child, 1 for going to right child). At this point, the program is ready to decode and encode.

To encode a message, the program reads a single letter from the file and searches for it within the Huffman tree the same way the codes were found to be printed. Once the character data matches, the current code is written to an output file. This is repeated until all of the characters on the line have been read and their codes written to the output file. One advantage of traversing the tree is that any characters that are not in the tree can easily be skipped over.

Decoding a message is slightly more tricky. The 1's and 0's are read from the encoded message one at a time, and, after every time a letter is added, the tree is traversed to see if that code matches a leaf, signaling a character match. This requires a lot of searching and could likely be replaced with a more efficient methodology. Once a character match is found, the string of bits being evaluated is rest to blank, and the whole process starts over. This continues until all the 1's and 0's in the line have been read.

What I Learned

Through completing this lab, I learned the proper steps to take to implement a priority queue, which I had never coded before. Checking for all the tie-breakers was quite intensive, and all of the comparisons had to be precise. I also learned that a buffered reader/writer could be passed around through method calls to different classes. With minimal programming experience, I didn't realize that was possible. However, it makes a ton of sense and seems extremely useful. At first, I thought I would need to create new readers and writers in each class.

Huffman Encoding was also a new topic for me, as it is the first form of encoding I have been exposed to. Using a tree to prioritize the more frequently used characters to have a shorter code is genius. As a result, I am now curious about other encoding methods and how they can be used to achieve different tasks such as compression.

What I Would Do Differently:

Looking back on my implementation, there are a numerous ways I would change it. In the methods to insert and remove nodes, I ended up using a lot of the same code to get the priority comparisons right. Since the inserted node only needed to be compared to its parent, it seemed that the comparisons would be much more simple than having to compare to two children. Even with the different set of comparisons, the code was largely the same, and I think the program would be more succinct with a specific priority method that could be passed a different number of nodes depending on whether the node was being inserted or percolated down.

Additionally, I think it would be a great enhancement for the program to be able to identify each type of input file rather than requiring them in a certain order. A method could be used to identify the file by reading the first line and checking whether there was a single letter on the line with a number, numerous letters and no numbers, or only 1's and 0's with no letters.

My methodology to find the letter for each code and the code for each letter required a lot of repeated searching through nodes. Each time the tree was traversed, the program had to pointlessly pass over all of the complex nodes. To eliminate the needless searching, a separate node array could have been created when the letters and codes were printed to the output file. That way, the program would only need to search through the array of 26 characters and codes instead of the entire Huffman tree. Such an array could be reused for both encoding and decoding.

To hone in on decoding an encoded message, a small change could be to not start looking for the appropriate leaf node for a code until a number of 1's and 0's equivalent to the shortest code was read. Since the "e" had the shortest code at 3 characters, searching for a code that was one or two characters was pointless. This small change would eliminate a large number of tree traversals, ultimately improving time complexity.

Efficiency:

For reading the frequency table and inserting each node into the tree, the time complexity is n * depth/2. Each line is looped through, and, for each line, the node will be compared to an average of depth/2 parents. The depth of a tree can be represented as log(n).

Merging the nodes requires (n-1) * (log(n)/2) * 2log(n)/2 * ((n^2)/8) operations. Removing two nodes and adding a single back takes n-1 operations to get to down to one node. For each merge, the insert method already referenced is used each time, counting for the second term. The remove method adds another 2log(n)/2 operations as the replaced root node has to percolated down the tree again. Lastly, the insertion sort for merged nodes takes n^2 time, with the average data size of a node being around n/8.

Printing the tree requires another (2*n) operations as every node in the tree is visited. For every two nodes that are removed, a single node is added back, but that node also has two children. Each child node is also a parent node, resulting in a balance of 2 nodes added for every single node removed.

Printing the codes for each letter also visits each node, adding another (2*n) operations.

Encoding files adds another variable into the complexity. For each letter, on average, half of the entire tree is traversed, resulting in a complexity of (2*n) / 2) * the # of characters in sentence. The average is likely higher than half of the tree due to spaces, punctuation, and other similar characters representing the worst case, causing a traversal of the entire tree.

Decoding the files is similar to encoding. For each coded number in the line on the file, the tree will be traversed an average of (n/2) times. The average will be slightly higher due to the worst case here being the evaluation of a code that is shorter than any of the letter codes, causing a traversal of the entire tree.

Ultimately, the big O time complexity should fall around n * ((log(n))^2) * (n^2). This time complexity is severe, and would definitely not hold up as an acceptable solution at scale.

For space complexity, the array is of size 2*n. All of the heap sorting is done in place, not requiring any additional memory. Merging creates 2*n new tree nodes for a big O space complexity slightly larger than n^2. The insertion sort adds additional memory requirements while it is being run, plus, the readers require additional memory by reading an entire line at a time.

These efficiencies are not exact. It would require data gathering and testing different n sizes to accurately describe the time and space complexities of the entire program as a whole. The operations are also very dependent on the variance between the overall number of characters with codes compared to the number of characters that need to be encoded or decoded.

Enhancements:

A small enhancement implemented into the program was sorting the data of nodes alphabetically when they are merged together. This keeps the printed tree easier to read and check based on the frequency table. The alphabetical sorting was achieved using an insertion sort, where every value is checked against the values prior to it.