Lab 3

Justin Ely

615.202.81.FA15 Data Structures

13 November, 2015

---

1 Comments

This was the most difficult lab to date. The necessity of implementing both a tree and a minimum queue, along with the necessary encoding and decoding methods, was a significant challenge. However, seeing all the pieces fit together in the end was rewarding. In particular, this lab showed how the right data structure makes the coded solution much more elegant and robust. The combination of minimum queue and binary tree made creating a complex structure very simple.

1.1 Compression

Limits on the possible compression can be determined by examining the best and worst case encodings. For the frequency table given in this lab, the letter e is the most-used and will be encoded to the least number of bits. The letter q was the least used, and will be encoded to the largest number of bits. The Huffman tree produced with the rules specified will encode e as 010, and q as 10110010. Comparing to standard 8-bit ASCII, an e is compressed from 8 bits to just 3. The q, on the other hand, remains at 8 bits regardless. Therefore, an e achieves a 62.5% compression while a q achieves a 0% compression.

Real-world application will see a compression gain somewhere between these extremes. The alphabet, by taking each letter just a single time, achieves a 36.5% compression rate. Other input of simple words or sentences typically achieves a $\sim$ 30 - 50 % compression.

Overall, the rate of compression found from this technique is impressive - and easily demonstrates why the Huffman encoding technique has moved from encryption to compression.

1.2 Tie-Breaking

A very important part of building the Huffman encoding tree is deciding how to break ties of priority. Sorting alphabetically, of same length strings will put the alphabetically smaller element to the left, and the larger to the right. Choosing a reverse-alphabetic sort would of course do the opposite.

Since a 0 always means to follow the left path, and 1 always means follow the right path, the particular tie-breaking method employed will move nodes from left to right, and cause a different decoding to be performed.

1.3 Necessary Structures

Beyond the binary tree necessary to hold the encoder, I also employed a minimum queue and a hash table. The hash table was only used to store the letter:frequency pairs in a convenient structure, but the minimum queue was necessary to build the tree efficiently.

The queue structure is typically double ended, where elements are inserted at one end and deleted from the other. However, this application called for a variant of a priority queue called a minimum queue, where elements are inserted with a specific priority and the element with the lowest priority is always returned. By inserting individual letters and letter combinations, along with their priorities, into the minimum queue, it's trivial to always pull out the two minimum priority elements. This is the strategy intrinsic to building the encoding tree from the frequency table.

2 Design

The design of the tree structure is very straightforward, where every node is implemented as another tree and linked appropriately to parents and children. This type of design accommodates recursion easily, as evidenced by the traversal methods. In, pre, and post order traversals are implemented as recursive calls on the node and all children nodes until NULL elements are found. With very limited amounts of code, building and traversing the trees are simply accomplished.

The minimum queue created here is a very specialized structure for the application. It only keeps track of the minimum priority element, and relies on the linked nodes to track the rest. New elements, when inserted, are inserted at the correct order specifically for the Huffman tree. The method goesBefore() determines the order first by priority, then by length, then alphabetically. Additionally, the queue nodes hold the value of the element, the priority, as well as an associated tree. Keeping the tree inside the queue was another example of tailoring this structure for this application.

2.1 Enhancements

As mentioned before, in addition to the requirement of a pre-order traversal, post and in-order tree traversal methods were implemented. Though this lab required the pre-order, each of the various traversal methods are useful for different applications, so building them into the tree can have utility in the future.

To clearly show the effect of the encoding, the translator has a method to calculate the fractional decrease in size between the encoded and decoded messages. Th savings are printed along the messages when using the main driver.

Additionally, documentation and unittests were added to the package. However, due to possible non-standard libraries, these options can be turned off in the given makefile.

On the user interface side, a simple enhancement was done to improve usability. Since non-alpha characters cannot be encoded, a O or 1 in the input string is an indicator that the input is a coded message that needs to be decoded. Because of this, the parser simply checks if the first element of each input line is a 0 or a 1, and then chooses whether to encode or decode the line. This means the user doesn't have to specify for the program to perform one or the other. Rather, simply supplying their message will cause the appropriate action to be taken.

2.2 Limitations

The current frequency table intrinsically limits the scope of this program to alphabetic encoding and decoding. In many applications, numbers, special characters, and white space would be necessary to effectively communicate.

This limitation also extends past the frequency table. Since certain characters can't be encoded, functionality needed to be put in place to ignore such characters while parsing. This means that even with a new frequency table, the entire application still wouldn't be able to work with other non-alpha characters until the special handling functionality were removed. This limits the expansion capability of the code.

Though the main driver will choose encoding or decoding based on the input string, this does lead to a not insignificant ambiguity. An illustrative example would be an input like 0101110Hello. Does this represent an improper coded message or a plain text message with non-acceptable characters? As there is no unambiguous way to determine which it is, an assumption simply has to be made.

3 Efficiency

The minimum queue, implemented as a linked structure, cannot take advantage of random access and has O(n) insertion efficiency. However, as only the elements of lower priority need to be traversed, a real-world application will average out to O(n/2) as $\sim$half of the queue is traversed. Deletion from the minimum queue is O(1), as the lowest priority element is always at the top and nothing needs to be searched.

Encoding or decoding a single letter has a maximum cost of the height of the tree, but the true value of that cost will be different based on the frequency table used to build the encoding tree. Encoding or decoding an entire message is of O(n), though there will be variations as each letter will traverse a different height through the tree. In this case, the cost will likely not average out to the height/2, as the most frequently used letters will have shorter paths and will weight the average to lower values.

4 What I learned

My main take-away from this lab was the simplicity of the tree structure. While learning about the construction, traversal, and general use of trees and graphs they appear as very complicated objects. However, from an using them in an actual application, their built-in recursive properties make them much easier to build and use than I had anticipated.

5 Next time

The main improvement I would make next time would be to the minimum queue structure. The requirements of the structure evolved as my understanding of procedure evolved, which made it end up less clean than I'd prefer. In particular, the queue began as a rather generic structure with generic nodes containing a value and an associated priority. This allowed the priority insertion and deletion to take place. However, as it became clear that to build up the encoding tree each node would need to contain the entire tree, the tree aspect of the nodes was simply tacked on.

This resulted in a structure with a value, a priority, and a tree that contained redundant value and priority attributes. For a clean solution, the minimum queue should have been build around storing trees and simply knowing where to access the priority within the tree.

Additionally, I would like to implement special handling to only ignore characters which do not appear in the frequency table when encoding. This would allow new frequency tables to be loaded without needed to change any other aspects of the encoding process and increase the utility of the routines.