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.

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.

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 tree necessary to hold the encoder, I also employed a minimum queue and a hash table.

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 minqueue, 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 accomodates 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 untill 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 Limitations

2.2 Enhancements

As mentioned before, in addition to the requirement of a pre-order traversl, post and in order tree traversal methods were implemented.

3 Efficiency

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.