Post Lab 10 Report

Gabriel Groover

Implementation

In order to implement the Huffman encoding, I used a min heap data structure (later used to create the Huffman tree). The heap was implemented using a vector of "Huffman Nodes." Each Huffman Node contains a frequency value, character value and a left and right pointer. These are necessary in order to eventually build the Huffman tree and generate the prefixes. The choice of vector implementation was mainly for ease. Vectors tend to be less space efficient than arrays because they dynamically allocate memory, but this was a nonfactor because the vector would never get big enough for this to be a problem. There are only a limited number of characters, so the heap cannot have more than this number of nodes. This also means that we do not have to worry about vector amortized time, so we can take full advantage of the easy implementation. The heap itself also had a map field to keep track of the prefixes created. A map is not the most space efficient way to do this because it requires storing a second instance of each character in order to look of the prefixes. I chose it to increase the speed of future lookups and easily associate a prefix with each character. In order to generate these prefixes, I formed a Huffman Tree inside the heap.

The inlab portion required only a single data structure. When you are given the already encoded file, you are also given the prefixes. This means that you do not need to use the min heap to create a Huffman tree. I used only my Huffman Nodes and generated a Huffman tree from the given prefixes. This saves space and is easier to code.

Time/Space Complexity

The first thing we must do to encode a file is read in the input and operate on each value. This is unavoidably linear time. For each character we must call heap insert, which is constant, but percolate up is not. In the worst case we will have to percolate all the way to the root node each time creating a running time on nlog(n). Both the Huffman tree creation and the and the prefix generation are worst case linear, leading to a runtime of nlog(n) for encoding. Decoding, as stated above much simpler. To decode we must read in each character, generate the Huffman tree and then iterate through it. All of these are worst case linear. This makes our runtime for decode n. Neither of these runtimes are ideal.

To evaluate the space efficiency of encoding/decoding we must first consider the size of the Huffman Nodes that make up our structures. The nodes contain a 4 byte int, 1 byte char and two 4 byte pointers to left and right. The Huffman Nodes Have a total size of 13 bytes. The size of the actual vector in our heap is not known because it depends on the variety of characters in the input file. We can assume the worst case of all valid acii characters and a vector size of 95. This gives us a max vector size of (95*13) 1235 bytes. We must also account for the int vector size and the size of the map<>. Assuming 95 characters in the map, we have (1235+4+95) 1334 bytes. We cannot predict the size of the string that is associated with each character in the map. The string would be one byte for each character it contains.

Decoding requires much less space in memory due to avoiding the heap implementation. The only things taking up memory when decoding is the nodes in the Huffman tree. Each node is 13 bytes as before. The number of nodes in the tree will depend on its structure.