605.202: Data Structures

Lab 3 Analysis

Renee Ti Chou

Due Date: April 18, 2017

Lab 3 Analysis

Data Compression

- Encoding: 4.19, 4.41, 4.33, 4.48; Avg. = **4.35 (bits/character)**

- Decoding: 4.33, 5.36, 4.53, 4.30; Avg. = **4.63 (bits/character)**

(The above values for statistics is from Output.txt in Protocol#1 file)

On average, the Huffman Encoding algorithm compresses 1 character into 4.5 bits. Comparing to the extended ASCII table, typically 1 character is encoded into 8 bits. The present compression of Huffman Encoding is then (1 - 4.5 / 8) * 100 = 43.74%, which is very close to one half of the original file size, and thus could be an effective data compression.

Different Protocols of breaking ties

In Protocol#1 file, the resolving scheme is given by the assignment instruction; whereas in Protocol#2 file, the order is to examine the ties alphabetically, and then by the length of groups.

The output results show that overall the Huffman tree has similar topology in which the level of the tree is the same, but there are some minor differences near the leaves. This causes wrong decoding results of the strings provided. The possible reason for this is that basically the frequencies dominantly determine the shape of the tree, and there are few situations of resolving ties. In other words, there are few ties occurring during the process of building the tree.

In different schemes of resolving ties, the percent compression remain the same. The statistics for the encoding results in Ouput.txt in Protocol#2 file is: 4.19, 4.41, 4.33, 4.48; Avg. = **4.35 (bits/character)**. Thus, no matter which scheme is used, the compression effect is the same when frequency is always the first subject of comparison.

Data Structures Implementation

In this lab, the data structures I used is a binary tree for storing the Huffman tree, and a priority queue for building the tree. The binary tree is implemented by the linked structure because it is easier to trace the left child or right child and to traverse through the tree than an array implementation.

The priority queue is useful because it helps determine the two least frequently letters in the list. It is implemented by a binary heap, which is simple and more efficient to delete the item with the highest priority in contrast to using an array and applying a sorting algorithm. The binary heap is represented by an array, and the reason is that it is convenient to change the parent and the child arithmetically.

Justification for Design Decisions

There are five classes in this lab. Lab3 contains the main entry of the program, which simply called the methods of HuffmanEncoding and outline the output format.

The HuffmanEncoding class implements the main algorithms for Huffman encoding and decoding, which are modularized and separate from the main entry to make the code succinct.

The HuffmanTree builds the tree, along with the code table constructed in order to facilitate the process of encoding. The code table is without annotation for two reasons. One is it saves space by avoiding storing data repeatedly. The other is it makes the code simpler since no additional string array is created.

The PriorityQueue facilitates the process of building the tree by determining the highest priority, which uses a heap structure represented by an array. The root starts from index 1 for arithmetic convenience.

The node class contains the priority scheme that is written in the compareTo method. It also has the parent field in addition to the left and right children. This makes the process of making code table easier because it helps travel from leaves up to the root.

Issues of Efficiency

Priority queue (the main data structure in this lab):

Time complexity

insert() –

Since the priority queue is implemented by a heap, the time complexity of insert() is **O(ln n)** in the **worst case**, where the attached child node traveling up to the root. (ln n) is the level of the binary tree.

On average, when the values of the items to be inserted have a uniform distribution, the chance of the attached child having the priority higher than its parent is 0.5, and it would be 0.5 * 0.5 = 0.25 possibility that the attached child has priority higher than its grandparent, and so on. Thus, the sum of the possibilities is approximately 1, having the constant time cost of **O(1)**.

delete() –

The time complexity of delete() is **O(ln n)** on average because the last item which swaps with the root usually travels down to the lowest level of the tree.

Space complexity

The **space** complexity of the heap structure is linear to the number of stored items in it, which is **O(n)**, because no additional space is required to assist the algorithm.

What I learned

I learned how to apply different data structures to organize data to achieve the purposes of the resolution, which, in this case, is the one of Huffman Encoding.

I leaned that, to reduce the programmer time more effectively, and to make the programming process more efficient, I have to write down an ADT for each module first, so that I know what methods are needed in each class, and outline the structure of the whole program.

I also learned to debug each unit of module thoroughly before combining them together. In this way, I would readily realize that the problem is from the interface of modules if there is one arising.

What I might do differently next time

I might divide this program into two pieces to make it more user friendly. The first program reads in the message file, calculates the frequency of each letter in the text, and creates a file of encoded strings, along with a file containing the code table. The other program reads in the two files generated by the first program, decodes the file using the provided code table file, and prints the results to the output.