Andrew Taylor

Atayl136

Data Structures Lab 3 Analysis – Huffman Encoding

Introduction

After a strong performance on Labs 1 & 2, I came into this project with a lot of confidence, in fact a little cocky. I was in for a humbling experience. As I began to flesh out my design, it seemed that a neverending cascade of functions was required, and the errors kept popping up. Online resources were unhelpful, as they seemed to be implementing Huffman Compression, not Huffman Encoding. Out of three pieces of example code, not one fit the project here, so I had no guide to work by. I had to think for myself completely.

I took it step by step, but honestly there were moments when I thought I would not complete a working program by the deadline; I reached out to both Vicky and Mr. Almes for help, but at a certain point even he couldn't figure out why it wasn't working. He did point me on the right path to debugging, however, and eventually I got it working. This is by far the hardest challenge yet of the course.

The Data Structure

The task of building a binary tree called for a heap, built by a priority queue. Many lists, dictionaries and strings went into the final min Heap, which was simple in design but very difficult to implement. In the end I had to rely upon both an insertion sort that I programmed, and the built-in .sort() function just to make sure the priorities in my queue were lining up right. For some reason I couldn't use the insertion sort for both the queue and the dictionary, and Mr. Almes told me that including in every pass of the priority queue was inefficient and would increase the complexity, so it is a problem that I did not solve in the best way, and could not think of any other approach on my own. Using sort() was the only function that may have been outside of the scope of the requirements, I believe.

Otherwise, the min Heap was built simply from one class of Nodes that I nested according to frequency first, then alphabetical order. An additional attribute of Node.code was populated after the tree was built. All this came together to create a Huffman Tree built from the frequency table in FreqTable.txt, and the tree worked for decoding, so I assume that my encodings are correct too. Mr. Almes confirmed that I had done the decoding correctly.

On Compression and Encoding

The Huffman scheme provided useful compression. Versus UTF-8, for instance, which is a four byte encoding scheme in the worst case, Huffman Compression is able to encode the most common bytes in as few as 1 or 3 bits, while uncommon bytes can take up more than 8 bits. This comparison is stark and drastic. I noticed the ClearText was a group of very short binary numbers indeed. I misunderstood the project at first, I thought the encoding frequency came

from the string, so I was able to get a clear look at some of the differences of using another frequency table from the final project. While the example “HelloWorld” came out in a similar format, it was shorter than the encoding that used a larger tree, which was built from the FreqTable.txt that assigned every letter. So, even further compression could be achieved in this fashion, however, it would not be practical for encryption purposes because you would have to transmit the table/key along with the message. However, I think this is standard practice in Huffman Compression for other means.

The scheme to break ties was critically important to the final encoding or decoding. Other examples online used other tie breaking schemes and the results of encoding or decoding using those programs was completely different. I think in that way it was useful to look at the examples. If I had given precedence to alphabetical ordering then to the number of letters the tree would have been completely different, with A, B , C….and so forth populating the leaves from right to left. Since groups of letters are always in the internal nodes, I don’t believe having them subsequent would change the result much, but putting them any higher in precedence would be radically different, and would no longer qualify as a Huffman Encoding.

Design

My design worked this way, I thought the most repeatable element in the data structure was the nodes, so I set that as my class. Incorporating pointers, and attributes for symbol, frequency, and code were easy with that choice. I think this was better than creating a heap class. I thought about opening up the heapq library for that approach, but I realized a list of nodes as a priority queue would get me the right results, in theory.

I came upon a problem. The leaves were being added correctly but the merging of nodes into letter groups was not happening properly. I used a list for a queue, and removed nodes as they were merged putting the merged nodes back on the end of the queue. However, this didn’t produce the right results for some reasons I can’t understand still. I spent a lot of time on this problem. Sorting the nodes with every action of the queue fixed it, but it presented a problem, this would burden the program in production or when using many more nodes. I used an insertion sort that I implemented to order the nodes by frequency and then alphabetically at first, but the need for this constant sorting still puzzles me. Somehow, my nodes were getting out of order. What was I doing wrong?

Other than that, things were more straightforward and my design worked: Reading in the inputs including table and data, parsing the data one character at a time. I did not take this approach for the frequency table because it didn’t seem to qualify as data in my ADT. I thought it would be overkill. In the case of the data I was screening out spaces, but not for the frequency table. I was unclear if this was also necessary.

Another choice I made was to include the capability for named files but comment it out just for the convenience of the grader. My program as it sits just works with the filenames

hardcoded, but I could comment back in the sys statements and it would be necessary to supply an argument when it runs.

Efficiency

This is where my program might fall short, in efficiency and time complexity. The standard implementation of Huffman compression has a complexity of O(n lg n) when encoding each unique character based on it's frequency. Extracting minimum frequency from the priority queue takes place 2*(n-1) times giving it this overall complexity. Presumably, reading a preset frequency table would make it faster, but my implementation does a sort every pass of the loop driving the queue, so I think the complexity of my algorithm is closer to O(n) or even O(n^2) depending on the sort selected. I could have done better on this but I spent maybe six hours on this problem so I settled on something that works. Hopefully in another setting a co-worker could catch this mistake because me and my rubber ducky couldn't really see what was happening.

Error Checking

My program doesn't do any error checking because it didn't seem to be an issue for this task. For encoding, any character is possible, and for decoding I simply checked for numbers. Presumably, I could have forced only o's and 1's, but I wanted to read in with only one function, and I didn't think of a way to easily distinguish a file for decoding from a file of only 0's and 1's that you want to encode. In fact, the user can present this case, so I didn't do any error checking of the data coming in. It didn't seem to be the point of this lab but the program could have been better with that.

Enhancements

The main enhancement of my implementation is something I commented out – the ability to run the program on one file and it would encode or decode depending on the input. I distinguished files for encoding as having alphabetical elements, and files for decoding as having numerical components. In practice I think it is best to have a separate script for encoding and one for decoding and instead employ some error checking, but seeing as how this is one project I took the other route of combining features.

Lessons Learned

In this lab I learned to be clear on:

1) …the client's requirements. It wasn't until I was many many hours into the code that I read on Dr. Cost's discussion forum that the FreqTable.txt is both for encoding and decoding. I had implement a frequency counter and was confused about my results for a long time before I noticed the comment there, as I searched for an answer.

2) …what the data structure is doing. Mr. Almes had me add print statements inside my loop and that led to resolving the error. I had not been checking the merging of the nodes and so I was puzzled by the output. I am still not sure how a list of nodes successfully insertion sorted would not merge without further sorting, but it solved the problem. Sometimes things just don't work as intended.

3) …the different options for my strategy and how different implementations have an effect on the outcome. It was eye-opening to see a smaller encoding for Hello World depending on the frequency table. It gave me more respect too, for Huffman's invention because many things other than alpha characters are encoded. I saw that a table based on the data works best for other applications.

Conclusion

This was by far the hardest part of the class so far. The fact that my program works and gets the right output still surprises me. I had to get help from the both the grader and the instructor, which was vital. I appreciate the help. The implementation still raises questions for me that I do not understand. Also, I couldn't split up my code into modules without breaking something. In the Programming Guidelines it says for some projects it won't be necessary, depending on the project, so I decided it wasn't a good idea to make modules, I hope this is OK.

However, I am much more confident building binary trees, and I can do it without a library. Using a library the effort must be even simpler so I'll have to learn that too someday. Thank you!