Analysis of Lab 3

Sean Dickson
4/15/25

Analysis of Data Compression:

Huffman encoding can be used as data compression algorithm that reduces the size of data by encoding characters with shorter codes for more frequent characters and longer codes for less frequent ones. This method ensures that the overall length of the encoded data is minimized, as the frequent characters take up fewer bits, while the infrequent ones take up more.

For example, as shown in one of my test cases, the string “Hello World” gets compressed into the binary string “1101101000010001111100011111010000000101100”.

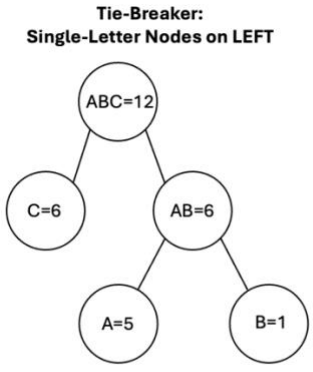
Through normal ASCII methodology, each character in the “Hello World” string requires 8 bits of space (for a total of 88 bits). When compressed into the above code, each character in the string can be represented by a single bit (for a total of 38), saving us 50 bits of space!

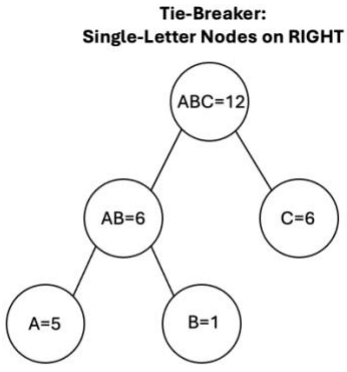
Comparison to Conventional Encoding and Other Tie-Breaking Schemes:

Unlike conventional encoding, Huffman Encoding is not as prone to breach by pattern or frequency analysis. The variable-length coding strategy helps prevents the identification of patterns or relationships between characters and prevents simple “substitution ciphers” from deciphering by exchanging each character for a letter of the alphabet.

Further, the encoding algorithm can be easily altered by changing the frequency table or tie-breaking strategy used. Changing the frequency table will change the tree more dramatically, but simply altering the tie-breaking strategy can alter the Huffman codes for characters as well.

For example, had we not used the tie-breaking strategy of placing the single-character nodes to the left, and instead placed them on the right, we could see the following changes take place, changing the codes for all characters involved:





Description of Data Structures Utilized:

In this package, I developed a Priority Queue class specifically for handling Huffman Encoding Tree nodes:

__init__()

This constructor simply initializes an empty array – thus the Priority Queue created here is an array-implemented queue. The left end of the array will be considered the front of the queue.

***** length()**

This function calculates the current length of the Priority Queue and is utilized by the "insertionSort()" function. (***) see "Discussion of Enhancements" section.

isEmpty()

This function determines if the priority queue is currently empty, returning True if so and False otherwise

enqueue()

This function adds a HET node to the array and sorts it using the "insertionSort()" function to give it correct priority.

dequeue()

This function removes an item from the front of the Priority Queue and returns it

insertionSort()

This function sorts the HET nodes within the array via the insert sort algorithm

Description and Justification of Module Design:

The Lab3 Python Package is broken into the following files and folders:

__init__.py

This file allows Python to recognize this folder as a Python package and exposes the process_files() function found in lab3.py.

__main__.py

This file is run when the package is called as a standalone program. The file uses the native "argparse" library to parse command line arguments from the user. The file then passes these arguments to the process_files() function, executing the lab3.py program.

lab3.py

This file contains the main functions for the Lab3 project. This is where the Huffman Encoding Tree is created via huffmanBuildtree(), the frequency table model is built via buildCharacterFrequencyTable(), the Huffman codes are obtained via huffmanGetCodes(), and the preorder traversal of the tree is output by preorderTraversal(). The process_files() function handles file I/O.

HETPriorityQueue.py

Contains the Priority Queue class and all its functions.

***InternalNode.py**

Contains the InternalNode class.

***LeafNode.py**

Contains the LeafNode class.

README.md

This file contains instructions on how to run the package as a standalone program and includes relevant details on language and IDE used to develop the module.

lab3/resources/

This directory contains a series of my own test cases (IO files), in addition to the required input and associated output files.

*I decided to make a distinction in classes between "internal" nodes and "leaf" nodes while building this package, even though their characteristics are very similar. This made it easier to determine which node was a "leaf" node when searching inside of the huffmanGetCodes() function. This way, I can simply use the "type()" internal Python function to identify leaf nodes.

I designed each file with readability and compartmentalization in mind. Each file is no longer than a page and has its own unique purpose, making readability and debugging relatively easy.

Efficiency with Respect to both Time and Space:

Huffman encoding is a greedy algorithm that constructs a binary tree (Huffman tree) to efficiently encode data based on character frequencies. The steps involved have the following time complexities:

1. **Building Frequency Table – O(n) time**, where n is the number of lines in the FreqTable.txt file
2. **Building Priority Queue (via Insertion Sort) – O(n^2) time**, where n is the number of leaf nodes
3. **Building the Huffman Encoding Tree – O(n log(n)) time**, where n is the number of total nodes
4. **Generating Huffman Encoding Messages – O(n) time,** where n is the number of characters in the original message

The space complexity for the Huffman Encoding Tree is O(k) where k is the number of total nodes.

What I Learned and What I Might do Differently Next Time:

I found that recursion played a major role in simplifying the construction of the tree, especially when it came to merging nodes and traversing the tree to generate codes. It allowed me to focus on the core logic without worrying about managing multiple iterations manually.

Next time, I would focus on optimizing the data structures used in the algorithm. For example, I could experiment with using more efficient sort methods for my priority queue/heap for faster node extraction. Additionally, I would work on making the tree-building process more visual, which could help in debugging and understanding the tree's structure more intuitively.

Discussion of Enhancements:

1. **Separation of Leaf and Internal Nodes**

   I created two separate node classes for this project – InternalNode and LeafNode. Both have similar characteristics, but having two distinct classes for these types of nodes made it easy to decipher between the two in the huffmanGetCodes() function. This way, I can simply use the "type()" internal Python function to identify leaf nodes.

2. **"length()" function in Priority Queue**

   As part of my HETPriorityQueue data structure, I included an additional function called "length()" that returns the current size of the queue. This was used in troubleshooting efforts, as well as the insertionSort() function.

3. **Added headers to output file for clarity**

   For the output files, I included headers for the different sections. At the top of each output file will be the "DECODED MESSAGES" or "ENCODED MESSAGES" header, designating where the encoded/decoded messages are located. Below that is the "HUFFMAN TREE IN PREORDER" header, designating the start of the tree traversal section.