

EN.605.202.81 Section 84

Alexander Shah

Lab 3: Huffman Encoding Analysis

November 13, 2023

Lab 3 Analysis

This program processes data to encode or decode using the Huffman Encoding scheme by taking in a frequency table, encoded data to decode, data to encode, and an output file to write to. The program builds a Huffman tree from the frequency table, develops codes from the tree, and then uses these codes to encode or decode data. The program uses error handling to report when characters could not be processed, and reports results and errors to the output file.

Huffman encoding is a lossless compression scheme and this program saves data by using shorter codes for more frequent characters. Compared to other fixed length encoding schemes, Huffman can save a good amount of data, though this depends on the character frequencies and thereby the message.

Breaking ties in a specific manner may influence the frequency table and how much data is compressed. This is a two sided problem, the method for building the tree can influence how optimal the codes are produced, but are also dependent on the message and frequency table. Working out a good tree building method in accordance with the message can effect the performance of the compression. For example in this program gives left precedence to single letters, then alphabetically where the tree is traversed in preorder traversal. If the program were to first alphabetize then consider groups then single letters, the codes would be much longer as short codes would be created for groupings alphabetically then longer codes for single letters.

It was necessary to use a priority queue in order to create a tree with frequency precedence and tie breaking mechanisms. The nodes themselves are defined as Huffman nodes in order for the queue to contain the relevant information a Huffman tree requires. This includes the character, frequency, and the nodes to the left and right if present.

Justification for Design Decisions

This program uses a priority queue, which is one data structure choice for building trees, another choice could have been a heap but I chose to further improve the previously used stack code from older labs by turning it into a priority queue. The intermediary data structures are arrays and strings which in Python are easy to manipulate and call other methods on. The program uses string methods like join and strip to remove whitespaces instead of processing them in the algorithm and discarding them. If necessary the error handling in the encoding and decoding methods could handle whitespace and any character which doesn't have a code. The error handling in the encoding method gathers any characters it did not convert and reports them to the output file. In the decoding method, if an incorrect bit is found, not a 0 or 1, the incorrect bits are gathered and an error is reported to the output file. This helps identify where things go wrong but also to try and keep going while keeping the user informed.

What I Learned/What I would Do Differently

The implementation of the Huffman tree and finding codes took a lot of trial and error to match the expected codes. Huffman trees and codes can be made from different rules that change the codes even though the correctness of encoding or decoding steps may be the same, the message will not be translated properly. I spent a lot of time trying to get the codes in my lab to match the codes I worked out by hand, and I had made a mistake in my calculations, which I could only notice by getting the same codes repeatedly from my programming which led me to recalculate the codes I had made. I would spend longer on the implementation than getting things to match and verifying that down the line the actual encoding and decoding process was going smoothly. If I had just tried to convert hello world, I would have noticed my progress much easier.