Task 8 Huffman

Adrian Jonsson Sjödin

Spring Term 2023

Introduction

The task is to implement Huffman encoding and decoding functions, and explore different ways one can represent data.

Huffman coding is a method of encoding data that assigns frequently occurring characters to shorter bit sequences, and less frequent characters to longer bit sequences.

In short terms it works as follow:

1. Count the frequency of each character in the message or data.
2. Build a binary tree from the characters, where each leaf node represents a character and its frequency, and each internal node represents the sum of the frequencies of its children.
3. Assign a "0" to the left branch and a "1" to the right branch of each internal node.
4. Traverse the binary tree from the root to each leaf, assign a binary code to each character based on the path taken to reach it.

The result will have the property that no code is a prefix of another code. This means that the encoded message can be uniquely decoded by using the HUffman tree. To decode a message one start at the root of the tree and follow the code until we reach a leaf node, which will be representing the character we want.

Method

A lot of the code was given in the task description as well as publicly available in the course GitHub repository. Thus I will focus mostly on explaining how the algorithm was implemented in code, and not show any code here. For those interested in the code please see the course GitHub.

To create our Huffman tree we must first compute the frequencies of all the characters for a given input text. These will be returned as a list of tuples containing the character and its frequency. We then construct our sorted tree by inserting one character at a time based on the frequencies we have. The nodes will represent the characters and the branches the bit sequence assigned to each character.

Next we need to encode it. We do this by creating an encoding table in which we traverses the tree in a depth-first manner and generate the binary codes for each character. We then sort it in ascending order. After this we take the input text and the encoding table and iterate over each character in the text. For each character we look up the binary code from the table and append it to a binary sequence that represents our compressed data, now containing our encoded text.

The decoding process is then the reverse of the encoding process.

For the benchmarking we need a large input text and a sample text that provides the correct frequency distribution. We also have to make sure the sample contains all the possible characters. The result from the benchmark will be discussed in the last section of this report.

Result

The benchmark war run on an input text of 318,997 characters and was able to build the Huffman tree in 45,82 ms and a Huffman table of size 77 in 0.032 ms. The text was encoded in 51.585 ms and decoded in 1,387.906 ms. The compression rate was 0.535.

When the input text was varied in size, the time taken to build the Huffman tree and the Huffman table increased linearly with the size of the input text. The time taken to encode the text increased slightly faster than linearly with the size of the input text, while the time taken to decode the text increased much faster than linearly with the size of the input text.

When the size of the Huffman table was increased, the time taken to build the Huffman table decreased linearly with the size of the table. However, the time taken to encode and decode the text remained relatively constant.

Discussion

It seems like the time taken to encode the text is dependant only on the size of the input text, while the decoding time is dependant on both the input text and the Huffman table.

All the functions except for the one building the tree and the decoding, should have a time complexity of $\mathcal{O}(n)$. This since we need to iterate through the length of the input text, or the length of the tree. For the tree however we have a time complexity of $\mathcal{O}(n \cdot log(n))$. This is also the reason why the decoding is the thing that takes the longest. It takes the encoded sequence and the tree as inputs and need to iteratively traverses the tree from root to nodes, consuming one bit at a time until a node is reached, and appending the corresponding character to the output string. Since we need to traverse the tree once for each bit in the input sequence, the time complexity of this function is $\mathcal{O}(n \cdot log(n))$.