Huffman Coding

Nariman Haidar

Spring Term 2023

Introduction:

In this assignment, we will implement the Huffman encoding and decoding functions algorithm using lists, trees, and tuples. Huffman coding is a data compression algorithm that is lossless. A variable-length code is assigned to different characters in this algorithm. The length of the code is proportional to how frequently characters are used. The most common characters have the shortest codes, while the least common characters have longer codes.

Huffman overview:

The algorithm is implemented in this module by first constructing a Huffman tree based on the frequency of characters in the sample text, and then using the tree to generate encoding and decoding tables. The text is encoded into a sequence of bits by the encoding table, and the sequence of bits is decoded back into the original text by the decoding table.

In order to implement Huffman code, we will first break the process into two parts: the creation of the coding tables and the much simpler encoding or decoding of text using the tables. First we are going to defines a module called Huffman that implements the Huffman coding algorithm.

The Huffman module contains the following functions:

1. sample: A function that returns a sample text to be used in the construction of the Huffman tree.
2. text: A function that returns a text to encode and decode.
3. test: A test function that encodes and decodes the text using the Huffman algorithm.
4. tree: A tree function that takes a sample text as input and returns the Huffman tree.
5. `encode_table`: A function that takes a Huffman tree as input and returns an encoding table containing the mapping from characters to codes.

6. `decode_table`: A function that takes a Huffman tree as input and returns a decoding table containing the mapping from codes to characters.

7. encode: A function that takes a text and an encoding table as input and returns a sequence of bits representing the encoded text.

8. decode: A function that takes a sequence of bits and a Huffman tree as input and returns the decoded text.

The Huffman tree:

In this section, we will implement the Huffman tree, beginning with the freq function, which accepts a string as input, counts the frequency of each character in the string, and returns a list of Leaf strut's, each of which represents a character and its corresponding frequency. The function employs Enum.reduce to iterate over the input string and count the frequency of each character, followed by Enum.sort by to sort the list of Leaf strut's in ascending order by frequency. Then to construct the Huffman tree, we must first determine the frequency distribution in our sample text. We can begin building the tree once we have the frequency distribution. It takes a list of Leaf strut's and uses the Huffman algorithm to build a binary tree. It starts by making a binary tree node for the two Leaf strut's with the lowest frequency, then builds the tree recursively until all Leaf strut's have been used. It returns the constructed tree's root.

Huffman encoding:

We must be able to maintain track of each character once the tree has been constructed. This is accomplished by outlining the route we follow to get to each character. Then we'll implement the function that will search for a single character. It traverses the tree to find the character's path, where each left turn is encoded as a 0 and each right turn is encoded as a 1. The path is returned as a list of 0s and 1s. Following that, we can create the encode function, which accepts a string as input, builds the Huffman tree and table, and then encodes the input text using the Huffman code for each character. The encoded string is returned as a collection of 0s and 1s by the function.

Huffman decoding:

We should be able to decode data after we've encoded it. The decode function accepts a list of 0s and 1s as input, as well as the Huffman table, and decodes the Huffman code by traversing the Huffman tree and returns the correct character for each path. The decoded string is returned as a list of characters by the function.

Performance:

In this part, we will run some benchmark to see how well it performs and estimate the time required to encode or decode a text given the length of the text and we can see the results in the table below.

Encode(ms)	**Decode(ms)**
233	4392

Table 1: The time measurements to encode kallocain and decode kallocain in (ms) in the Huffman implementation.