Lab 12 - Huffman

Weichen Chai

Spring 2023

Introduction

This paper seeks to answer the questions presented in"Huffman" for ID1019. In this assignment, we will explore how the functional language Elixir can be utilized to solve the Huffman encoding and decoding problem, working with processing of tuples, lists and trees.

Huffman

Huffman coding is an encoding method where binary codes are assigned to characters or symbols based on their frequency of occurrence in the input data. The Huffman coding algorithm works by constructing a binary tree of nodes, where each leaf node represents a character or symbol and the path from the root to each leaf node represents the binary code assigned to that character.

0.1 The table

Starting with the first module for Huffman, we begin by using the code given under this section and filling in the blanks.

```
//Creates Huffman tree given a sample text
  def tree(sample) do
    freq = freq(sample)
    huffman(freq)
  end
```

Furthermore, we would have to implement a frequency function, which returns the frequency of each character in string.

```
  def freq(sample) do
    freq(sample, [])
```

```
  end
  def freq([], freq) do
    freq
  end

  def freq([char | rest], freq) do
    freq(rest, add(char, freq))
  end
```

When running the sample given to us in the skeleton code with tree sample, it returns a large tuple.

0.2 The Huffman tree

The Huffman tree described here is a binary tree used in Huffman coding for data compression, where each character in the input data is assigned a length of binary code, with the length of the code determined by the frequency of the character in the input. The more frequently a character occurs, the shorter the binary code assigned to it.

To implement the Huffman tree functions, we need to consider a left frequency and

```
//base case
  def huffman_tree([{tree, _}]) do
    tree
  end
  def huffman_tree([{left, freqa}, {right, freqb} | rest]) do
// makes a new tree by merging left and right, adding their frequencies inserted to
    huffman_tree(insert({{left, right}, freqa + freqb}, rest))
  end
```

The function runs till base case conditions are met.

0.3 The encoding table

In order to begin, we have to first understand how the encoding table should work. The binary code is generated by traversing the Huffman tree from the root to the leaf node that corresponds to the symbol, using a left traversal is a 0 and a right traversal is a 1. When a leaf node is reached, the code for the corresponding symbol is complete and is added to the encoding table.

```
def huff_encode({left, right}, path) do
   toLeft = binary_encode(left, [0 | path])
```

```
    toRight = binary_encode(right, [1 | path])
    toLeft ++ toRight
  end
...
  //char for leaf and path for binary
// reverse path
  end
```

When running the encoder, the text file we are given :"this is something we should encode" is translated into numerical bits format: [1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, ...]

0.4 Huffman decoding

After completing the encoding section, we proceed to decoding, where we use Huffman table reading the binary sequences from left to right.We search the table for a character with a set pattern. If we find a character, we output it and move on If we don't find a character, we take the next binary number and add it to the previous one to form a two-bit, then continue the process stated above.

```
//Base function
def decode([], _) do [] end

//
  def decode(seq, table) do
    {char, rest} = decode_char(seq, 1, table)
    [char | decode(rest, table)]
  end

  def decode_char(seq, n, table) do
    {code, rest} = Enum.split(seq, n)
    case List.keyfind(table, code, 1) do
      {char, _} ->
      //return matching character and remaining bits
        {char, rest}
      nil ->
      // no match found, recursive call with n+1 for next bit.
        decode_char(seq, n + 1, table)

    end
  end
```

0.5 Performance

Finally, we are asked to run performance tests on the function, we will use the given sample test given in "kallocain.txt", we can insert this file into our codes using the codes given in the assignment. Running a benchmark on the tree table, encode and decode times, we realize decoding takes much longer than encoding shown below:

Func	Time(ms)
Tree	71
Encode	86
Decode	2024

This can be explained by the fact that decoding requires more recursion for a character than encoding. In this document, there are 318147 total characters. Ensuring we have all letters in the text, if changing the total number of characters, we can get the following:

Num of Char	Encode	Decode
105767	23	665
147550	38	959
203715	64	1336
249611	79	1639
318147	86	2024

Plotting it in Excel we get that both are quite linear with the decoding factor completely linear and encoding time leaning in between linear and logarithmic.

Testing the times for algorithms with different lengths