Task 12 - Huffman Coding

Casper Kristiansson

February 24, 2022

Introduction

Task twelve of the course Programming II consisted of implementing the Huffman algorithm for compressing texts. The task includes building a Huffman tree, encoding it, and then decoding it. The author will be disusing how each subtask was solved and explain how the algorithm works.

Result

The task consisted of four different subtasks, one for counting the frequency of the letters, one for implementing the Huffman tree, one for creating a table for all the characters and one for encoding and decoding the text. The author will be dividing the result section into a subsection for each subtask.

Character frequency

The first part of the task includes counting the frequency of all the different characters in the text. The author solved this using a map where the key is the character, and the value is the frequency. The function checks if the character exists in the map and if it does not it adds it to the map with the value one. Otherwise, it increases its value by one.

```
def freq(sample) do freq(sample, %{}) end
def freq([], freq) do freq end
def freq([char | rest], freq) do
  case Map.get(freq, char) do
    nil ->
      freq(rest, Map.put(freq, char, 1))
    value ->
      freq(rest, Map.put(freq, char, value + 1))
  end
```

Figure 1: Character frequency in text

Huffman tree

The Huffman tree builds on the principal where high frequency characters have less branches between the root node while less frequency characters have more. The tree basic structure is that each character is stored as a leaf node and the parent node has the value of frequency of the child's characters.

The algorithm starts of by receiving an array of tuple which is sorted by the frequency in ascending order. The function will receive the two tuples with the lowest frequency and merge them together by adding their value together and setting their character value as a child node. By doing this a parent node will have the following structure {{character, character}, frequency + frequency}. The function will continuously do this until there is only one node left in the array.

```
def huffman([{tree, _}]) do tree end
def huffman([{key, value}, {key2, value2} | rest]) do
  huffman(insert({key, value}, {key2, value2}, rest))
end

def insert({key1, value1}, {key2, value2}, []) do
  [{{key1, key2}, value1 + value2}]
end
def insert({key1, value1}, {key2, value2}, [{key3, value3} | rest]) do
  if value1 + value2 < value3 do
    [{{key1, key2}, value1 + value2}] ++ [{key3, value3} | rest]
  else
    [{key3, value3}] ++ insert({key1, value1}, {key2, value2}, rest)
  end
end
```

Figure 2: Huffman tree

Finding characters paths

The third subtask of the task was to implement an array of tuples which contains the path in the tree for each of the characters. When navigating the tree, a path towards a left child is 0 and a path towards the right child is 1. The function finds the path to all the different characters by firstly navigating to the left most node and then recursively navigating the entire tree and appending the path to a list. This table will be used to encode the text to receive a compressed text.

```
def encode_table(tree) do
  find_path(tree, [])
end

def find_path({tree_left, tree_right}, current_path) do
  paths_left = find_path(tree_left, current_path ++ [0])
  paths_right = find_path(tree_right, current_path ++ [1])
  paths_left ++ paths_right
end
def find_path(tree, current_path) do [{tree, current_path}] end
```

Figure 3: Function for building a table which consists of all of the characters paths

Encoding a text

After building the table the next task is to create a bit array consisting of all the characters bits. Doing this should result in a text with a lot of less bits than the original one. The function is pretty basic to implement where it will navigate the list of characters and append the correct path for each character to the result array.

```
def get_bits([], _) do [] end
def get_bits([char | rest], tree) do
  get_path(char, tree) ++ get_bits(rest, tree)
end

def get_path(char, [{tree_char, path} | rest]) do
  if char == tree_char do
    path
  else
    get_path(char, rest)
  end
end
```

Figure 4: Finding the new bits for each character

Decoding a Huffman text

The next task includes decoding of an Huffman compressed text. The task can easily be solved by converting bits into different characters with the use of the Huffman tree. The function receives an array of bits, which represents the text, and the Huffman tree which was used to encode the

text. The function recursively navigates the tree until it finds a character which is represented by the path and appends it to a list. By doing this the text can be decoded.

```
def decode([], _) do [] end
def decode(seq, tree) do
  {char, rest} = decode_char(seq, 1, tree)
  [char | decode(rest, tree)]
end

def decode_char(seq, n, table) do
  {code, rest} = Enum.split(seq, n)
  case List.keyfind(table, code, 1) do
    {char, _} ->
      {char, rest};
    nil ->
      decode_char(seq, n + 1, table)
  end
end
```

Figure 5: Converting a list of bits to a text

Discussion

The last part of the task includes creating a bench method for testing each of the functions for a larger text. Running each of the functions with the text from "Kallocain" gets the following result in figure six. The fastest function of the algorithm is the function for creating the Huffman tree and the function for encoding texts. What took the most time for the algorithm was the decoding the text. This is because the algorithm requires the most recursions calls. This is because the algorithm does not know the bit length of each character and therefore, it needs to constantly check one bit at a time until it finds a path to a character. The algorithm was able to compress the text "Kallocain" to around 52% of its original size.

Function	**Time (ms)**
Tree Build	37
Encoding Table	0
Encoding Text	85
Decoding Text	1400

Figure 6: Table of the runtime for the different functions in milliseconds (ms)