Task 14 - Morse Signals

Casper Kristiansson

March 18, 2022

Introduction

Task fourteen of the course Programming II consisted of implementing both a encoder and decoder for Morse signals. The task main goal of the task is to solve it with low time complexity. Therefore the author will discussing how each part of the program was implemented.

Result

The main parts of the program was implementing the encoder and decoder for the program. The author started off by implementing the encoder and the data structure which represent the different alphabetic translations to Morse code.

Encoding

The first part of the program was to translate the given character tree to a data structure which should hold key value pairs with letters and their Morse code. The tree that was given in the instructions to the tasks works where navigating towards the left in the tree represents the Morse symbol **"-"** and navigating right represents **"."**.

The author started off by simply recursively navigating the tree and if a character exists in the current node the path to that node is appended. The author choose to use a Map to represents all of the key value pairs. The lookup operations for the map will be on average O(1) and therefore one of the better options for representing the Morse translation table.

The function for transforming the given table to the Map consists of six different methods for covering all of the different scenarios of the tree. Figure one consists of three of these scenarios. The structure of each of those functions is either add the current character path to the map or continue searching traversing the tree.

```
def generate_map({:node, :na, tree_left, tree_right}, map, current_path) do
  map = generate_map(tree_left, map, current_path <> "-")
  generate_map(tree_right, map, current_path <> ".")
end
def generate_map({:node, character, :nil, :nil}, map, current_path) do
  Map.put(map, [character], current_path)
end
def generate_map({:node, character, tree_left, :nil}, map, current_path) do
  map = generate_map(tree_left, map, current_path <> "-")
  Map.put(map, [character], current_path)
end
```

Figure 1: Transforming character tree into a Map data structure

The next part of the encode is to match characters in a sequence with the characters in the Map. The function than appends the Morse representation of the character to a string. The author added a case statement for the **Map.get** function which simply ignores if a unknown character is entered. Otherwise the function will continue recursively with the next character. By using a Map the total time complexity will be O(m) where m is the message length. The author also made sure that the solution follows the guidelines for tail recursion so no unnecesary space is stored in the stack. The encoded author name (casper) ended up being ” -.-. .- ... .–. . .-.”.

```
def encode(str) do
  morse_tree = Morse.morse()
  map = generate_map(morse_tree, %{}, "")
  encoder(map, str, "")
end
def encoder(_, [], result) do result end
def encoder(map, [head | tail], result) do
  case Map.get(map, [head]) do
    nil ->
      encoder(map, tail, result)
    char ->
      encoder(map, tail, result <> " " <> char)
  end
end
```

Figure 2: Encoding a string

Decoding

The next part of the task was to create the decoding methods to translate a Morse string into characters. The author solved this by converting the input charlist to string. The string is than split into characters by dividing the string into a list using the **String.split** method (based on spaces).

The next part is to recursively navigate each character and translating the Morse character into a letter. The function does this by using the **String.at** and **String.slice** methods. Therefore the function will traverse the tree (either left or right depending on "." or "-"). When the current Morse character is empty the current tree node is the character.

The way the author choose to represent the decoder method is nearly the same way the author solved the previous task *Huffman*. The method builds on the principal that the most frequent characters have the shortest path. The general time complexity for traversing a tree is on average O(logn) but because the Morse tree is built on most frequency characters the average time complexity will be faster. The total time complexity for decoding a Morse code will than be O(logn * m) where m is the message length. The translations of the two given texts gives a quote and a YouTube video.

```
def decode([], _) do [] end
def decode([head | tail], tree) do
  [decode_char(head, tree) | decode(tail, tree)]
end
def decode_char(sequence, {:node, character, tree_left, tree_right}) do
  case String.at(sequence, 0) do
    "-" ->
      decode_char(String.slice(sequence, 1..-1), tree_left)
    "." ->
      decode_char(String.slice(sequence, 1..-1), tree_right)
    _ ->
      character
  end
end
```

Figure 3: Decoding a Morse code

Discussion

The goal of the task was to implement the methods for encoding and decoding a string to Morse code or the other way around. The most important part of the task was to really think about which data structure would be best to represent the different parts of the program and how the different functions affects the time complexity for both encoding and decoding. The method for translating the tree structure to a Map has a time complexity of **O(n)** (on average) where n is the nodes in the Morse code tree. The author was able to implement a Map to have on average O(n) lookup for translating the characters to Morse code.

The decode method was implemented by simply traversing a tree where on average the lookup time will be O(logn) for a character (note that this in reality will be faster due to the tree being a ”frequency character tree”). The author thought that these were the best way to solve the encode and decode methods.