Last Assignment

Azer Hojlas

March 2022

Introduction

The assignment is based on, given a conversion tree, encoding a text into Morse code and vice versa. Due to time constraints, some parts of the code are largely based on Johan Montelius github as well as his proposed improvements during the final seminar.

Code In Depth

The teachers constructor of encode_table() is used and will therefore not be submitted. For the sake of time complexity, a hashmap is used to allow for constant time access of values, where the keys are ASCII representation of characters and the values their corresponding Morse codes.

encode_table()

```
defp traverse(:nil, _), do: []

defp traverse({:node, :na, dash, dot}, tree) do
    traverse(dash, [?-|tree]) ++ traverse(dot, [?.|tree])
end

defp traverse({:node, char, dash, dot}, tree) do
    [{char,tree}] ++ traverse(dash, [?-|tree]) ++ traverse(dot, [?.|tree])
end
```

The morse code values were given in a similar fashion to how huffman coding was implemented. That is, depending on if one traverses left or right in the tree, a dot or dash is added. If a leaf or node is empty (base case), nothing is added. If a node with :na atom is traversed through, only the values of the left and right tree are added as :na represents no value at current node. The other recursive clause is only different in that it adds the value at the current node iteration.

Encode

```
def encode(text), do: encode(text, encode_table(), [])
def encode([], _, reversed), do: reversed

def encode([char|text], table, accumulator) when accumulator == [] do
    encode(text, table, Enum.reverse(Map.get(table,char)))
end

def encode([char|text], table, accumulator) do
    encode(text, table, accumulator ++ [32] ++ Enum.reverse(Map.get(table,char)))
end
```

The first clause receives a text to encode, where the given encoding table is used for the encoding. The second clause, the base case, returns the Morse code once all characters have been processed. A problem would earlier occur where the first character in the result would be a star as there would be a blank space at the end of the reversed list. This is remedied by having two recursive clauses where the first character is added to the list without an additional blank space. During this first operation, the accumulator is empty. In the second recursive clause, the head of the list, i.e a letter, is looked up in the table for its corresponding Morse code, after which a blank space as well as the accumulator are concatenated. The blank space is added as it is necessary later on for the decode function to discern different letters, where 32 acts as a delimiter. As the codes are received in reversed order, it is necessary to remedy this by invoking Enum.reverse().

Decode Inspired By Johan

```
def decode(signal, table) do
    case signal do
      [] -> []
      _ ->
        {char, rest} = decode_char(signal, table)
        [char|decode(rest, table)]
    end
end

def decode_char([], {:node, char, _, _}) do
    {char, []}
end
def decode_char([?-|signal], {:node, _, dash,_}) do
    decode_char(signal,dash)
end

def decode_char([?.|signal], {:node, _, _,dot}) do
    decode_char(signal, dot)
```

```
end

def decode_char([?\s|signal], {:node, :na, _,_}), do: {?*, signal}
def decode_char([?\s|signal], {:node, char, _,_}), do:  {char, signal}
```

In the base case, if no more code remains to be decoded, an empty list is returned. Otherwise, the signal is passed on to decode_char/2. This signal will traverse down the tree in different directions depending on what value is given in the head (either dot or dash). This way one will traverse to the correct character. Once the current iteration is a blank space aka 32 aka
s, the correct character will be returned. If no appropriate character exists, a star is returned.

The authors decode

```
def decode2([]), do: []
def decode2(text = [char | rest]) do
    map = Map.new(encode_table(), fn {key, val} -> {val, key} end)
    {translate, tail} = iterate(text, text, [])
    [Map.get(map, translate) | decode2(tail)]
end

def iterate([h | t], text, acc) when h != 32 do
    iterate(t, text, [h | acc])
end

def iterate(_, text, acc), do: {acc, text -- [32 | acc]}
```

What the code does is that it initially reverses the keys and values of the encode_table() map. The idea being that incoming codes will simply be looked up in this reversed map and the corresponding letter is returned. Each letters Morse code is found by invoking iterate(). Iterate accumulates all characters until a blank space is encountered. Upon this triggered base case event, the accumulated list represents Morse code for for a single ASCII character and this *acc* is returned along with the rest of the signals where the accumulated values are subtracted from the original inputted list of signals. The aforementioned accumulated list of signals is then, within decode2(), looked up in the reversed map where the corresponding character is returned. This is done along the length of the list. Note that this implementation is probably a lot more slower than Johans code, as it takes O(n) time to reverse the map, and then O(n) time to iterate throughout the list as well as additional time to perform the $--$ operation. Disregarding the time-complexity, this function was intuitive and easy to comprehend if the reader understands the encode() and encode_table() functions.

Conclusion

This section will provide proof that encoding and decoding work by encoding/decoding the authors name as well as decoding the hidden message provided in the assignment instructions. Additionally, time-complexity will be briefly discussed.

Name

The authors name, 'azer', translates in Morse code to '.- –.. . .-.'

Hidden Message

The hidden message translates to "all your base are belong to us" along with the link: https://www.youtube.com/watch?v=d%51w4w9%57g%58c%51 which leads to a well-known music video.

Time complexity

The first thing to note is that hashmaps are used as lookup tables for both the decode and encode_table which enables constant time operations when looking up character and Morse code values. The encode() function uses concatenate functions along with Enum.reverse() which adds to time complexity. The encode function is tail recursive and characters are added in $O(n)$ time, the author is however unsure, due to the previously stated reasons, if the time complexity is less than $O(n * m)$.