The last assignment

Nariman Haidar

Spring Term 2023

Introduction

We're going to create a Morse signal encoder and decoder in this assignment. The task's primary purpose is to be completed with little time complexity.

Morse codes:

A technique known as Morse code is used in communications to encrypt text as defined sequences of two distinct signal durations, known as da and di. The principle is the same as the Huffman codes we created in the last assignment because it involves encoding frequent characters with few symbols, but there is a difference. We have a unique signal in Morse coding that lets us know when one character stops and the next begins. To decode a message, a pause between characters is required.

Encoding:

0.1 the encode table:

To do this assignment first we are going to implement the encode table. Function encode table generates a table of Morse code encoding from a binary tree data structure representing the mapping between characters and their Morse code equivalents.

This function initializes the process by creating the Morse code tree with `encoding_table` to return a list of pairs containing a character and its corresponding Morse code sequence, sorted alphabetically and then sorting the resulting table using the **Enum.sort()** function.

Then we implement the `table_case()` function is the main recursive function that walks through the binary tree, building up the Morse code table as it goes. The `table_case()` function takes cases like: `case {{:node, :na, a, b}}` that we used to traverse the tree and recursively build the Morse code table. It adds dots (.) and dashes (-) to the current code and descends the current node's left and right subtrees until it finds a leaf node, at which

point it adds the resultant character and its Morse code to the table. The `case {:node, char, nil, nil}` argument denotes a leaf node that contains a character but no additional left or right subtrees. In this case, the function adds the character and its Morse code to the table. The `case {:node, char, a, b}` argument denotes a leaf node having both left and right subtrees, as well as a character. In this instance, the function adds the character and associated Morse code to the table before continuing to iterative go along the left and right subtrees of the tree. Then the `encoding_table` function is a wrapper function that takes a Morse code tree and returns the table of Morse code encoding generated by `table_case()`.

```
def encode_table() do
    morse()
    |> encoding_table()
    |> Enum.sort(fn {a, _}, {b, _} -> a < b end)
  end
  def table_case({:node, :na, a, b}, l_list, list) do
    left = table_case(a, [?- | l_list], list)
    table_case(b, [?. | l_list], left)
  end
  def table_case({:node, char, nil, nil}, l_list, list) do
    [{char, Enum.reverse(l_list)} | list]
  end
  def table_case(nil, _, list) do
    list
  end
  def table_case({:node, char, a, b}, l_list, list) do
    list = [{char, Enum.reverse(l_list)} | list]
    left = table_case(a, [?- | l_list], list)
    table_case(b, [?. | l_list], left)
  end
  def table_case(tree) do
    table_case(tree,[],[])
  end
def encoding_table(tree) do
  table_case(tree)
end
```

0.2 the encoder:

Now after the `encode_table` method we can create the encode method, which accepts a text and returns an encoded Morse code sequence. The `encode()` method creates a list of Morse code sequences by accepting a

string of text and performing `lookup()` on each character in the string. On this list, the `unpack()` method is then invoked to concatenate the Morse code sequences into a single binary string.

```
def encode(text) do
   encode(text, [], encode_table())
 end

 def encode(list, acc, table) do
   case list do
     [] ->
       unpack(acc, [])

     [char | rest] ->
       encode(rest, [lookup(char, table) | acc], table)
   end
 end

 def unpack(list, acc) do
   case list do
     [] -> acc
     [head | tail] -> unpack(tail, head ++ [?\s | acc])
   end
 end
```

Now we can talking about the time complexity. The lookup method costs O(m) so in total the time complexity is O(n*m) for the method encode. That's because the encode method does a lookup before every encoding operation so in total the time complexity is O(n*m).

Decoding:

Now we will construct the decode method. The `decode()` method decodes the sequence one character at a time by traversing the Morse code tree. The`find_char()` method locates the next character in the Morse code sequence, then the `decode()` function is invoked repeatedly until the whole sequence is decoded. The decode approach was simply implemented by traversing a tree with an average search time of O for a character (logn).

```
 def decode(signal) do
   table = morse()
   Enum.reverse(decode(signal, table, []))
 end

 def decode(signal, table, acc) do
```

```
    case list do
      [] ->
        acc

      _ ->
        {char, rest} = find_char(signal, table)
        decode(rest, table, [char | acc])
    end
  end

  def find_char(list, {:node, char, left, rigth}) do
    case list do
      [] -> {char, []}
      [?- | signal] -> find_char(signal, left)
      [?. | signal] -> find_char(signal, right)
      [?\s | signal] -> {char, signal}
    end
  end
```

Result:

Here is the first sign decoded:
`iex(10)> Morse.test2()` 'all your base are belong to us'

Here is the second sign decoded:
`iex(11)> Morse.test3()` `'https://www.youtube.com/watch?v=d%51w4w9%57g%58c%51'`

Here is my name encoded:

```
iex(9)> Morse.encode('nariman haidar')
'-.  .- .-.  ..  -- .- -.  ..-- ....  .- ..  -..  .- .-.  '
```