Morse Coding

Johan Montelius

January 12, 2018

Getting Started

Morse codes were used in the days of telegraphs and manual radio communication. It is similar in the idea to Huffman coding in that it uses fewer symbols for frequent occurring characters and more symbols for the infrequent ones. You task is to write a decoder for Morse signals and decode two secret messages.

1 Morse Codes

There are several standards for Morse codes and we will here use a slightly extended version since we also want to code some special character. The Morse code uses, as you probably know, long and short (often pronounced *di* and *da*) to encode characters. You might therefore think that is identical to Huffman codes but there is a difference. In Morse coding we have a special signal that tells us when one character ends and the next start. The pause between characters is necessary in order to decode a message.

The code for 'a' is *di-da*, 'i' is *di-di* and 'l' is *di-da-di-di*. If we just had the sequence *di-da-di-di* we would not know if this was "ai" or "j"; we need a third signal, the pause, to tell the difference. A sequence di-da-pause-di-di-pause is then decoded as "ai".

How does this change the structure of our decoding tree? In a Huffman tree we only have characters in the leafs and when we hit a leaf we know that we have a complete character and can start from the root again. In a Morse tree we can finish anywhere along the path to a leaf. We thus have characters in every node of the tree (apart from the root).

2 The Decoder

The Morse codes that we will use are given in the decoding table in the Appendix. As you see we have represented the table as a tree were each node is on the form:

```
{:node, character, left, right}
```

An empty tree is represented by the `nil` value and the root holds dummy value instead of a character; if everything works fine we will never have to see the nil nor the dummy value.

To decode a message you only have to choose the left branch when you hear a long signal and a right branch when you hear a short signal. When you hear the pause you have decoded a character and can start from the root again.

Implement a function `decode(signal, table)` that takes a *signal* and the decoding table and returns a decoded message. The signal is in the form of a string with dots and dashes ".- . .-.. .-.. —" (i.e. a list with ascii characters 45, 46 and 32 (dash, dot and space)).

Decode the secret messages below. If you cut and paste the code, make sure that you don't have carriage-return etc in the string. The string should only contain the dash, dot and space characters.

```
'.- .-.. .-.. ..-- -.-- --- ..- .-. ..--
 -... .- ... . ..-- .- .-. . ..--
 -... . .-.. --- -. --. ..-- - --- ..-- ..- ... '

'.... - - .--. ... ---... .----- .----- .-- .--
 .-- .-.-.- -.-- --- ..- - ..- -... . .-.-.- -.-.
 --- -- .----- .-- .- - -.-. .... ..--.. ...- .----.
 -.. .--.-- ..... .---- .-- ....- .-- ----. .--.--
 ..... --... --. .--.-- ..... ---.. -.-. .--.--
 ..... .---- '
```

If you by accident have two spaces between to characters this might be decoded as `na` which then will generate a string that is not a proper string. To solve this problem you might add a rule for the case where you're in the root of the tree (the character is `na`) and you hear a space. Then you simply do nothing and just start from the root again. The rule can be made general so that any unknown character will simply make the decoder start from the root.

3 The Encoder

To encode messages we simply need the table that gives us codes for each character. You could of course extract this from the decode tree but a more convenient form is given in the Appendix.

To encode a message you simply have to look up each character in the message and append all the codes together. This is of course a very simple exercise but we can of course make it more complicated that it has to be in order to learn something.

We first define two functions, one that will create a encoding table and one that will do look-up operations. We assume that `codes/1` returns a list of tuples {`char, code`}.

```
def encode_table, do: codes()

def lookup(char, table) do
  List.keyfind(table, char, 0)
  |> elem(1)
end
```

We can now implement the encoder and this can now be free from any knowledge of how the table is represented. Try to encode a message and the use the decoder to see that you got it right.

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

def encode([], _), do: []
def encode([char | message], table) do
  code = lookup(char, table),
  append(code, encode(message, table))
end
```

Every time we need the code of a character we will have to look it up in a list which of course is unnecessarily expensive. We could of course store the codes in an ordered tree and that would improve thing but we could do better.

We note that once we have the encoding table we will only do look-up operations. We could then choose a representation that will do look-up quickly even if adding or removing elements is dreadfully expensive.

We also note that the keys, i.e. the characters, are integers in the range 32 to 122. It could therefore be quite possible to store the codes in a tuple that was indexed by the keys. The tuple would not be fully populated (there are no characters with codes less than 32 nor any characters between 64 and 97, but this does not matter.) If we could create a tuple with 122 elements we would have direct access to any code given the character as key.

Instead of trying to write this tuple explicitly we will create it during run-time using the list of codes. There is a built-in function called `List.to_tuple/1` that will take a list and return a tuple of all the elements in the list.

If we could only create a list of 122 elements where the i'th element is the code for the character with code i. Implement a function `fill/1` that takes an ordered list of key-value pairs, where the key is an integer, and returns a list of values. The returned list of values should be such that if the key k is mapped to the value v then the value v is the *i'th* element in the lists. If

the *j'th* element in the list does not have a mapping, the element should be `:na`.

```
fill([{1, a}, {2, b}, {6, c}])
```

should return:

```
 [:na, a, b, :na, :na, :na, c]
```

If you do it right you can now create a tuple that will give you the right encoding.

```
def encode_tuple do
  codes()
  |> fill
  |> List.to_tuple
end
```

Doing the look-up operation in now simply doing a call to `elem/2`.

```
def lookup(char, table), do: elem(table, char)
```

If you do some benchmarks, you will see that you have managed to speed things up considerably. One thing to ponder is if you have changed the run-time complexity of the encoder.

The Morse Codes

```
def decode_table do
  {:node, :na,
    {:node, 116,
      {:node, 109,
        {:node, 111,
          {:node, :na, {:node, 48, nil, nil}, {:node, 57, nil, nil}},
          {:node, :na, nil, {:node, 56, nil, {:node, 58, nil, nil}}}},
        {:node, 103,
          {:node, 113, nil, nil},
          {:node, 122,
            {:node, :na, {:node, 44, nil, nil}, nil},
            {:node, 55, nil, nil}}}},
      {:node, 110,
        {:node, 107, {:node, 121, nil, nil}, {:node, 99, nil, nil}},
        {:node, 100,
          {:node, 120, nil, nil},
          {:node, 98, nil, {:node, 54, {:node, 45, nil, nil}, nil}}}}},
    {:node, 101,
      {:node, 97,
        {:node, 119,
          {:node, 106,
            {:node, 49, {:node, 47, nil, nil}, {:node, 61, nil, nil}},
            nil},
          {:node, 112,
            {:node, :na, {:node, 37, nil, nil}, {:node, 64, nil, nil}},
            nil}},
        {:node, 114,
          {:node, :na, nil, {:node, :na, {:node, 46, nil, nil}, nil}},
          {:node, 108, nil, nil}}},
      {:node, 105,
        {:node, 117,
          {:node, 32,
            {:node, 50, nil, nil},
            {:node, :na, nil, {:node, 63, nil, nil}}},
          {:node, 102, nil, nil}},
        {:node, 115,
          {:node, 118, {:node, 51, nil, nil}, nil},
          {:node, 104, {:node, 52, nil, nil}, {:node, 53, nil, nil}}}}}}
end
```

```
def codes do
  [{32, '..--'}, {37,'.--.--'}, {44,'--..--'},
   {45,'-....-'}, {46,'.-.-.-'}, {47,'.-----'},
   {48,'-----'}, {49,'.----'}, {50,'..---'},
   {51,'...--'}, {52,'....-'}, {53,'.....'},
   {54,'-....'}, {55,'--...'}, {56,'---..'},
   {57,'----.'}, {58,'---...'}, {61,'.----.'},
   {63,'..--..'}, {64,'.--.-.'}, {97,'.-'},
   {98,'-...'}, {99,'-.-.'}, {100,'-..'},
   {101,'.'}, {102,'..-.'}, {103,'--.'},
   {104,'....'}, {105,'..'}, {106,'.---'},
   {107,'-.-'}, {108,'.-..'}, {109,'--'},
   {110,'-.'}, {111,'---'}, {112,'.--.'},
   {113,'--.-'}, {114,'.-.'}, {115,'...'},
   {116,'-'}, {117,'..-'}, {118,'...-'},
   {119,'.--'}, {120,'-..-'}, {121,'-.--'},
   {122,'--..'}]
end
```