Huffman Coding

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Getting started

In this seminar session we will look at different ways to represent data, using lists, trees and tuples to find the best representation. The *best representation* could of course mean many things, we might need a representation that gives us efficient code or we might want a representation that is easy to explain, implement and maintain. We will start by using quite simple representations and then refine them to gain better performance.

To have something to work with we will implement the *Huffman* encoding and decoding functions. You should do some reading on Huffman coding, this text will not explain the algorithm but how to implement it.

1 Huffman overview

Huffman coding can divided into two parts, one part is how to construct the coding tables and the other, much simpler, is how to encode or decode a text using the tables.

The idea behind Huffman coding is of course to encode frequent characters with few bits and infrequent characters with more bits. To keep things simple we will represent sequences of bits as lists of zeros and ones but this could of course be changes if we intend to do a real implementation that reads and writes to files. For our experiments it is sufficient.

The table should give a one to one maping from characters to codes but we might use one representation when we encode text and another when we decode text; the information it holds is the same but we might want to do this for efficiency.

Once we are done we will have a module that defines the following functions:

- `tree(sample)`: create a *Huffman tree* given a sample text.
- `encode_table(tree)`: create an *encoding table* containing the mapping from characters to codes given a Huffman tree.

- `decode_table(tree)`: create an *decoding table* containing the mapping from codes to characters given a Huffman tree.
- `encode(text, table)`: encode the text using the mapping in the table, return a sequence of bits.
- `decode(sequence, table)`: decode the bit sequence using the mapping in table, return a text.

Start by defining the module, some compile directives, things that are good to have and dummy code for the functions.

```
defmodule Huffman do

  def sample do
    'the quick brown fox jumps over the lazy dog
    this is a sample text that we will use when we build
    up a table we will only handle lower case letters and
    no punctuation symbols the frequency will of course not
    represent english but it is probably not that far off'
  end

  def text()  do
    'this is something that we should encode'
  end

  def test do
    sample = sample()
    tree = tree(sample)
    encode = encode_table(tree)
    decode = decode_table(tree)
    text = text()
    seq = encode(text, encode)
    decode(seq, decode)
  end

  def tree(sample) do
    # To implement...
  end

  def encode_table(tree) do
    # To implement...
  end

  def decode_table(tree) do
```

```
    # To implement...
  end

  def encode(text, table) do
    # To implement...
  end

  def decode(seq, tree) do
    # To implement...
  end
end
```

2 The table

In order to create the Huffman tree we need first to find out the frequency distribution in our sample text. Once we have the frequency distribution we can start building the tree.

```
def tree(sample) do
  freq = freq(sample)
  huffman(freq)
end
```

The sample is of course a list of characters (`[102,111,111]`), you should run through this list and collect the frequencies of the characters. If "foo" was the sample text we should have the frequencies *f/1*, *o/2*. How would you represent this information? Note that you need not know beforehand which characters that will occur in the sample.

You will probably end up with a structure that looks like this, but how you represent the frequencies is up to you.

```
def freq(sample) do
  freq(sample, ...)
end

def freq([], freq) do
  ...
end

def freq([char | rest], freq) do
  freq(rest, ...)
end
```

Now once we have the frequencies we will create a Huffman tree. This is simpler than you might think but before you read further you must understand why we create a tree and what properties it should have. If you started to read this with out understanding how Huffman coding works this is the time to stop reading.

2.1 The Huffman tree

OK, so a Huffman tree is a tree with the characters in the leafs but the low frequency characters have long branches and high frequency characters have short branches. Assume we represent a leaf with a single character and a node as a simple tuple with two branches: {`left, right`}.

If you turn your table into an ordered sequence of leafs where each leaf represents a character and its frequency. The "foo" example above would correspond to the sequence `[{'f', 1},{'o', 2}]`. You could also view this as a frequency table where each entry is a tuple {`tree, freq`}, after all, a single character is a leaf.

Now, assuming we have such a sequence, what would happen if we took the two smallest elements (lowest frequencies) and combined them into a new node {{`c1, c2`}, `f1 + f2`} and added the node to the remaining sequence while keeping the sequence sorted? Can we repeat this process, what will the final result be?

How do we represent the sequence so that it is easy to find the two smallest elements? How do we keep this representation? What is the final result when you only have one element in your sequence?

2.2 The encoding table

I assume now that you have a Huffman tree and it is time to extract the codes. The codes are of course hidden in the tree in the branches and the code of a character is the path to the leaf holding the character (*left*, *left*, *right*, *left* or *0, 0, 1, 0*).

Traverse the tree, and collect the characters in the leafs. Keep track of the path to the leaf and record this path as a sequence of zeros and ones. When you're done you should have something like `[{'f', [1, 1, 0]}, {'o', [1, 0, 1, 0]}, ...]`, or whatever the tree looks like.

Start by writing a function that only collects the characters, once this is mastered you can start to keep track of the path.

2.3 Half way

Half-way there might be an exaggeration but at lest you're now done with the first part, you have a mapping from characters to Huffman codes. It's represented by a list of tuples {`'f', [1, 1, 0]`}, one for each characters. Time to use this table in the encoding and decoding.

3 Huffman encoding

This is simple, we have a text represented as a list of characters and for each character we have a sequence of bits found in the table. You could probably create something very simple that works.

If you manage to implement the encoder you should be able to turn the text "this is something..." into a list of bits {`[1, 1, 0, 1, 0, 1, 0, ...]`.

Stop here an ponder what the time complexity is. You will of course have a linear factor, depending on the length of the text, since the text is encoded character by character but you might have other factors. What is the the time complexity of looking up a character in the table? What is the complexity of producing the final sequence of bits?

I'm quite sure that your original code is open for improvements but let's leave it for now.

4 Huffman decoding

Decoding is slightly more tricky since we do not know exactly how many bits are used to code each character. If we have a sequence `[1, 1, 0, 1, 0, 1, 0, ...]` it could be that the first four bits is a t and the following three is an i but we do not know; what we do know is that thanks to Huffman it is only possible to decode it in one way given a table with Huffman codes.

We will do a very simple implementation of the decoding and actually use the same table as wee used in the encoding phase; we will later see how to improve this.

Start by looking at the bit sequence, assume that one bit is used in the coding and then search the table for a character with this pattern (`[1]`). If a character is found, problem solved, if not, look for a character using two bits (`[1, 0]`). Let's implement this function first and see what we have, it will work and will probably not take that much time.

Your solution might look something like this:

```
def decode([], _)  do
  []
end

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

def decode_char(seq, n, table) do
  {code, rest} = Enum.split(seq, n)
```

```
  case List.keyfind(table, code, 1) do
    ... ->
      ...;
    nil ->
      decode_char(..., ..., table)
  end
end
```

I'm using some functions from the *Lists* and *Enum* libraries, you should look these up and understand how they work. If you fill in the blanks you're up and running, you have your first Huffman encoder/decoder.

Make sure that it works correctly, by testing the smaller functions and make sure that they behave correctly, then work your way up. Always test the corner cases i.e. when lists are empty etc before trying more complicated tasks.

5 Performance

Let's now run some performance tests. You need to find a large text that you can use to benchmark the program. You would also need to find a sample text of the given language that gives you the correct frequencies but we can cheat and use the text it self to do the frequency analysis.

One thing that you have to look out for is that you have to make sure that you sample text contains all the possible characters. You can ensure this by adding an alphabet to the text or pre-load the frequency table with all possible characters.

Do some benchmark of your system and determine how well it performs, try to estimate the time to encode or decode a text given the length of the text.

If you have time try to do some experiments where you change the size of the alphabet, for example using only eight characters, the regular alphabet, all ASCII characters. You would of course have to find suitable texts but you could of course work with a large text file and filter out a list with only the characters from your selected alphabet.

To read a file you can use the following code:

```
def read(file, n) do
  {:ok, file} = File.open(file, [:read])
  binary = IO.read(file, n)
  File.close(file)

  case :unicode.characters_to_list(binary, :utf8) do
    {:incomplete, list, _} ->
```

```
      list;
    list ->
      list
  end
end
```