Trading storage space with processing time

Praanto Samadder

March 8, 2024

1 Frequency counter

We start by counting the frequency of each character. Implementation becomes trivial once we use a map. We can split our charlist into cons cells, then check if the character exists in our map. If it does, we increment it's corresponding counter, otherwise add the character to our list.

```
defmodule Frequency do
  def freq(sample) do
    freq(String.to_charlist(sample), Map.new())
  end
  def freq([], freq) do
    freq
  end
  def freq([char | rest], freq) do
    char_freq = Map.get(freq, <<char::utf8>>, 0)
    freq = Map.put(freq, <<char::utf8>>, char_freq + 1)
    freq(rest, freq)
  end
end
```

Frequency

We also make use of the `«char::utf8»` notation to ensure our keys in the map are characters and not binary. This allowed for easier debugging.

2 The `Mapper` module

The `Mapper` is a specialised module with two functions: (a) `max_pop/1` which returns the character with the most occurrences and `min_pop/1` which does the opposite and returns the least occurring character.

Both functions performs a pop operation meaning the item is removed from the map. Returned is a tuple, with a tuple containing the key-value, and the remaining map.

```elixir
defmodule Mapper do
  def max_pop(a) when a == %{} do
    {nil, %{}}
  end
  def max_pop(a) do
    {key, value} = Enum.max_by(a, fn {_, x} -> x end)
    {_, returning_map} = Map.pop(a, key)
    {{key, value}, returning_map}
  end
  def min_pop(a) when a == %{} do
    {nil, %{}}
  end
  def min_pop(a) do
    {key, value} = Enum.min_by(a, fn {_, x} -> x end)
    {_, returning_map} = Map.pop(a, key)
    {{key, value}, returning_map}
  end
end
```

Listing 1: Mapper module

3 Generating the code table

Let us first discuss how the coding table is generated before we discuss the tree.

At this point in our code, we already have a frequency map. We can make use of our `Mapper.min_pop/1` (or `Mapper.max_pop/1` as well) and use the key to traverse through the tree to find the code. Sounds like a lot of gibberish? Allow me to explain.

- We start with the `encode_table/3` method. This function calls the `traverse/3` method.
- Now `traverse/3` can either return a `:code` tuple or a `:no_match` tuple. But when do each of them occur? Let's find out.
- The `encode_table/3` calls `traverse/3` by passing an empty string as the `code` and the entire Huffman tree (how that tree is generated is to be discussed)
- First thing `traverse/3` does is it takes the left branch and calls `traverse/3` on it (RECURSION!). This recursive call continues the traversal down the left path.
- As the traversal down the left path continues, there comes a time when we encounter a `:node`. If this node matches the key that we are looking for, then we returns the `:code` tuple. Otherwise we return a `:no_match tuple`.
- BUT WEIGHT! Recursive traversal creates an implicit stack. When we return a `:no_match` tuple, this is caught by the `traverse/3` method (which was higher in the stack) which then calls `traverse/3` again on the right branch.

By design, a `:node` is placed on the left side of a `:leaf`. On the right side of the `:leaf` can either be another `:node` or another `:leaf`.

```
defmodule Huffman do
  def encode_table(_, freq, encoding_map) when freq == %{} do
    encoding_map
  end
  def encode_table tree, freq, encoding_map do
    {{k, _}, map} = Mapper.min_pop(freq)
    case traverse("", k, tree) do
      {:code, code} ->
        encoding_map = Map.put(encoding_map, k, code)
        encode_table(tree, map, encoding_map)
      {:no_match} -> {:error, "what the fuck?"}
    end
  end
  def traverse(code, to_find, {:node, to_find}) do
    {:code, code}
  end
  def traverse(_, _, {:node, _}) do
    {:no_match}
  end
  def traverse(code, to_find, {:leaf, _, left, right}) do
    case traverse(code <> "0", to_find, left) do
      {:code, code} -> {:code, code}
      {:no_match} -> traverse(code <> "1", to_find, right)
    end
  end
  ...
end
```

Huffman coding

4 Encoding and decoding our text

Now comes the part to encode and decode our plaintext.

4.1 Encoding

Encoding involves taking each character and replacing it with the code from our coding table. That's it.

4.2 Decoding

Decoding requires traversal of our tree. We take the first bit of our encoded text and traverse appropriately (left for 0, right for 1). We continue this until we reach a `:node` at which point we take the next bit but this time we restart from the top of the tree. We pass a `root` argument which is an unmodified copy of our tree that allows us to begin from the start.

Notice how we pattern match for 48 and 49 as they are ASCII for "0" and "1".

```elixir
defmodule Huffman do
    def decode [], _, _, decoded_text do
    decoded_text
  end

  def decode [48 | tail], {:leaf, v, l, _}, root, decoded_text do
    case l do
      {:node, k} -> decode(tail, root, root, decoded_text <> k)
      leaf -> decode(tail, leaf, root, decoded_text)
    end
  end

  def decode [49 | tail], {:leaf, v, _, r}, root, decoded_text do
    case r do
      {:node, k} -> decode(tail, root, root, decoded_text <> k)
      leaf -> decode(tail, leaf, root, decoded_text)
    end
  end
  ...
end
```

Decoding

5 Last but not least

And finally we tackle generating our tree. I could only think of two ways of traversing the tree: (a) the lazy way and (b) the smart way.

5.1 The lazy way

The lazy way involves taking the two smallest (i.e. lowest occurring) characters from our map. We can then create a tuple with a value that is the sum of their corresponding value. This leaves us with a tuple that looks like

```
{:leaf, v1 + v2, {:node, k1}, {:node, k2}}
```

And then we continue by taking the next smallest character from our map and add its value to the value of the tuple we just created.

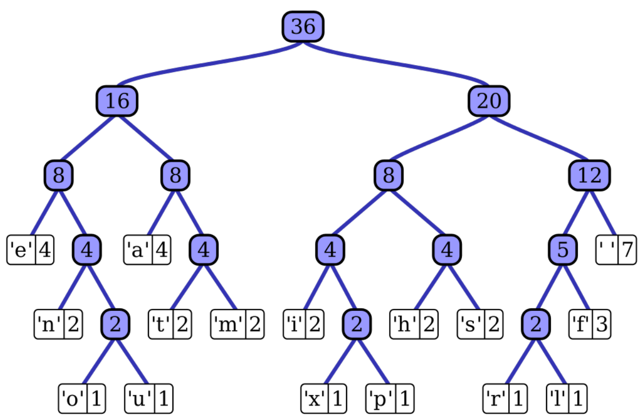
```
{:leaf, v3 + v1 + v2, {:node, k3}, {:leaf, v1 + v2, {:node, k1}, {:node, k2}} }
```

We can continue this process until we have accounted for all the characters in our list.

```elixir
defmodule Lazy do
  def tree_start(map) do
    {kv1, map} = Mapper.min_pop(map)
    tree(kv1, map)
  end
  def tree({k1, v1}, map) do
    {{k2, v2}, map} = Mapper.min_pop(map)
    new_leaf = {:leaf, v1 + v2, {:node, k1}, {:node, k2}}
    tree(new_leaf, map)
  end
  def tree(tree, map) when map == %{} do
    tree
  end
  def tree({:leaf, v, l, r}, map) do
    {{k2, v2}, map} = Mapper.min_pop(map)
    new_leaf = {:leaf, v + v2, {:node, k2}, {:leaf, v, l, r}}
    tree(new_leaf, map)
  end
end
```

The lazy way

5.2 The smart way



In our alternative (and better way) we implement a queue with FIFO properties. The diagram above, taken from Wikipedia, shows what I call a 'balanced Huffman tree'. Our objective for our 'smart' tree is to generate a balanced tree which will allow us to have smaller codes for each character.

We first take our two smallest (i.e. lowest occurring) characters and merge them. We then put this to the queue. We continue by taking the next smallest character and check if the value (i.e. frequency value) for this character is larger or equal to merged tuple we have stored in our queue. If such is the case, then we pop from the queue, merge the two and put the result back in the queue. If not, we allow the queue to remain unchanged while we draw another character and continue to merge the two characters we have at hand. We then put this back into the queue. We can continue this process until we have ran out of elements in our map but we ain't over yet.

We then use the `clean_queue/1` method to merge elements without our queue. The smallest-first nature of our `min_pop/1` method ensures that our queue is sorted from smallest to lowest 'merged' value.

We pop two elements from our queue and merge them, putting the result back into the queue. We then take two more elements from our queue, merge and put back into the queue. We continue with this process until either the queue is empty or there is only element left in the queue. This final value is our Huffman tree.

```elixir
defmodule Praanto do
  def tree_start(map) do
    first([], map)
  end
  def first(fifo, map) do
    {kv1, map} = Mapper.min_pop(map)
    {kv2, map} = Mapper.min_pop(map)
    fifo = FIFO.push(fifo, merge(kv1, kv2))
    next(fifo, map)
  end
  def next(fifo, map) when map == %{} do
    clean_fifo(fifo)
  end
  def next(fifo, map) do
    {{k1, v1}, map} = Mapper.min_pop(map)
    {{:leaf, vx, lx, rx}, fifo} = FIFO.pop(fifo)
    case v1 >= vx do
      false ->
        fifo = FIFO.push_last(fifo, {:leaf, vx, lx, rx})
        map = Map.put(map, k1, v1)
        first(fifo, map)
      true ->
        fifo = FIFO.push(fifo, {:leaf, v1 + vx, {:node, k1}, {:leaf, vx, lx, rx}})
        next(fifo, map)
    end
  end
  def merge({k1, v1}, {k2, v2}) do
    {:leaf, v1 + v2, {:node, k1}, {:node, k2}}
  end
  def merge({k1, v1}, {:leaf, vx, lx, rx}) do
    {:leaf, v1 + vx, {:node, k1}, {:leaf, vx, lx, rx}}
  end
  def clean_fifo(fifo) do
    case FIFO.pop(fifo) do
      {last, []} ->
        last
      {{:leaf, v, l, r}, fifo} ->
        case FIFO.pop(fifo) do
          {{:leaf, v_last, l_l, r_l}, []} ->
            {:leaf, v + v_last, {:leaf, v, l, r}, {:leaf, v_last, l_l, r_l}}
          {{:leaf, v_last, l_l, r_l}, fifo} ->
            fifo =
              FIFO.push(fifo, {:leaf, v + v_last, {:leaf, v, l, r}, {:leaf, v_last, l_l, r_l}})
            clean_fifo(fifo)
        end
    end
  end
end
```

Smart way to generate the tree

6 Benchmark? No no. Performance

The following table shows the number of bits that it takes to represent the entire provided text. Somehow the lazy method takes more bits to represent the text than the original (?) This could indicate that my implementation of the 'lazy' method is incorrect. But the table also makes it apparent that the 'smart' way of generating the Huffman tree allows us to reduce the file size by almost half.

Format	**Size (bits)**
ASCII	2 670 520
UTF-8	2 670,528
Lazy	2,766,721
Smart	1,483,422