A LZW Encoder

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Getting started

Lempel-Ziv-Welch (LZW) is a compression algorithm that takes advantage of frequent occurrence of sequences of characters. It will detect sequences on the fly while doing the compression and thus create individual codes for sequences as it goes along. The beauty of the algorithm is that the decoder must not be told what these new codes mean - it will learn as it does the decoding.

The encoder that we will implement will not use binary encoding i.e. codes are fixed size and are represented by an integer. A real implementation would start off by using, for example, a five-bit code and then increase the code length as needed. By implementing this simpler form you will understand the principles of the algorithm and you can easily extend it to use variable size codes.

Before you start to implement this encoder and decoder you should do some reading on the LZW algorithm so that you have a basic understanding of the process. The devil is as always in the detail and we will see how these are handled as we implement the encoder.

1 The table

The encoder and decoder have to agree on an initial alphabet (and in the general case, the code size). We will here use a very small alphabet that consists of the smaller cap letters and the space character. Given this we construct an initial encoder/decoder table that is represented as a list of character sequences and codes.

```
defmodule LZW do

  @alphabet 'abcdefghijklmnopqrstuvwxyz '

  def table do
    n = length(@alphabet)
    numbers = Enum.to_list(1..n)
    map = List.zip([@alphabet, numbers])
    {n + 1, map}
  end

end
```

The only sequences we know of in the beginning are the sequences consisting of single characters. We have 28 characters in total so our table will look like follows:

```
{28, [{97, 1}, {98, 2}, {99, 3}, ...]}
```

The number of sequences in the table is important to keep track of since we will add new codes as we encode our text. Have in mind that the encoder and decoder will both know the state of the initial table.

2 The encoder

So let's start the encoding of a sequence of characters. If the sequence is empty we're done but the common case is of course if we have at east one character. We use the first character to initiate the encoder. We pick up the encoding table, that of course holds a code for the single character word. We then call the `encode/4` function that is given: the text, the word, the code of this word and the coding table.

```
def encode([]), do: []

def encode([word | rest]) do
  table = table()
  {:found, code} = encode_word(word, table)
  encode(rest, word, code, table)
end
```

The function `encode/4` is where all the action takes place. The base case is simple, if there is not more characters in the text then we're done. If we have another character in the text we add this to the word we have read so far and check if this extended word can be found in the table. If we find a coding of the extended word we're happy but we might be even happier if we find an even longer world. This is where we continue with the extended word and its code.

```
def encode([], _sofar, code, _table), do: [code]

def encode([word | rest], sofar, code, table) do
  extended = [word | sofar]
  case encode_word(extended, table) do
    {:found, ext} ->
      encode(rest, extended, ext, table);
    {:notfound, updated} ->
      {:found, cd} = encode_word(word, table)
      [code | encode(rest, [word], cd, updated)]
  end
end
```

If a code is not found for our extended word we will return a list starting with the code of the word we had found so far. We will then continue the encoding.