A LZW Encoder

Programming II - Elixir Version

Johan Montelius

Spring Term 2021

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 words and codes.

```
defmodule LZW do

  @alphabet 'abcdefghijklmnopqrstuvwxyz '

  def table(), do: table(@alphabet)
  def table(alphabet) do
    n = length(alphabet)
    words = Enum.map(alphabet, fn x -> [x] end)
    codes = Enum.to_list(1..n)
    map = List.zip([words, codes])
    next = n + 1
    {next, map}
  end

end
```

The only sequences we know of in the beginning are the sequences consisting of single character words. We have 27 characters in total so our table will look like follows:

```
  {28, [{'a', 1}, {'b', 2}, {'c', 3}, ...]}
```

The number of words 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.

We now define three functions that uses the table. One will lookup a code given a word, one will lookup a word given a code and the last one will add a new word to the table; this is where we use the last code number.

```
  def lookup_code({_, words}, word) do
    List.keyfind(words, word, 0)
  end

  def lookup_word({_, words}, code) do
    List.keyfind(words, code, 1)
  end

  def add({n, words}, word) do
    IO.puts("Adding #{word} as code #{n}")
    {n+1, [{word, n}|words]}
  end
```

There are of course better ways of implementing this table and there are of course reasons to use two different tables. The encoder will only lookup codes given words and the decoder will only lookup words given codes. Both will however add new words to the table.

2 The encoder

So let's start the encoding of a sequence of characters given a table. 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 encode the single character word using the table (that of course holds a code) and 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([char | rest], table) do
  word = [char]
  {: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 are no 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([char | rest], sofar, code, table) do
  extended = sofar ++ [char]
  case encode_word(extended, table) do
    {:found, ext} ->
      encode(rest, extended, ext, table)

    {:notfound, updated} ->
      sofar = [char]
      {_, cd} = encode_word(sofar, updated)
      [code | encode(rest, sofar, 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 but now with an updated table.

Then function `encode_word/2` does a lookup using the table and if not found. The next time this word occurs in the text it will have a unique code. This is how the encoder learns about and take advantage of frequent occurring words.

```
def encode_word(word, table) do
  case lookup_code(table, word) do
    {_, code} ->
      ## if found we return the code
      {:found, code}
    nil ->
      ## otherwise we update the table
      {:notfound, add(table, word)}
  end
end
```

3 The decoder

The decoder is only slightly more complicated. The magic here is that the decoder is give a sequence of codes where it only knows the meaning of the first codes of the alphabet. The trick is for the decoder to add the words to the table in the same order as the encoder and thus gradually build up the table as it progress through the text.

The two first clauses of the decoder are the base case. The first one will actually only be used if we try to decode an empty code sequence. The second one is the general base case. If we only have one code left in the sequence the only thing we can do is look it up in the table.

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

def decode([code], table) do
  ## this is the last code in the sequence
  {word, _code} = lookup_word(table, code)
  word
end

def decode([code | codes], table) do
  {word, _} = lookup_word(table, code)
  updated = decode_update(codes, word, table)
  word ++ decode(codes, updated)
end
```

The third clause is where the magic occurs. We have a code and we take for granted that this code has a meaning in the table. We could have continued with the same table but we would then sooner or later encounter a code that is not in the table. This is where the magic comes in, we update the table using the word that we have found and the next code in the sequence.

The function `decode_update/3` will take the next code in the sequence and do a lookup in the table. In most cases the word of the next code is `found` and we there fore know what the encoder saw when it encoded the sequence. If `word` is `'banana'` (the word found in `decode/2`) and `found` is `'rama'` then we know that

- The encoder read `'banana'` and had a code for it but then read the first `r` in `'rama'` but could not find a code for `'bananar'`,

- The encoder used the code for `'banana'` but then also added `'bananar'` to the table of endoded words.

```
def decode_update(next, word, table) do
  char = case lookup_word(table, next) do
           {found, _} ->
             hd(found)
           nil ->
             IO.puts("Could not find the code #{next}")
             hd(word)
         end
  add(table, word ++ [char])
end
```

If the decoder does not find a word for the `next` code one might think that all hope is lost but we can save the situation. The strangest thing in the world is that the next character that was in the text was the same character as we had in the word already found. Why this works is the magic of LZW. The

4 The magic

To see how the magic works we look at an example. We will encode and then decode the string `'abababa'`. We need a lot of repetition for the magic to show it self.

When we encode this string we start with a table that only knows about the codes of the characters 1..27. The encoding therefore starts using these.

The first two codes will be 1 and 2 (the codes for ’a’ and ’b’) but we will also add a unique code 28 for ’ab’ in case we will be able to use this later. As we read and encode the third character we will also add a unique code 29 for the sequence ’ba’.

We now have ’a’ and read the following ’b’. The sequence ’ab’ is found in the table so we could encode our ’ab’ as 28. We could be lucky so we take a look at the next character ’a’ but ’aba’ is not in the table so we use 28 for ’ab’ but also add ’aba’ with code 30.

We now have the code

$$1, 2, 28$$

representing 'a b ab' and continue. We now read an 'a' and continue in the hope of finding a longer sequence. We read 'b' and find that 'ab' is in the table. The next character is 'a' and 'aba' is found with the code 30. We could have continued but this is the end of the string.

Note that we added 'aba' to the table and then immediately found a new 'aba' sequence. This is the crux of the LWZ algorithm.

The final code is

$$1, 2, 28, 30$$

and it is now time to decode this sequence.

The decoder will decode 1 as 'a' and 2 as 'b' but also adds 'ab' with code 28 to the table since it knows that this is what the encoder would have done. It therefor is not surprised when it finds 28 in the sequence. Since it is holding 'b' in its hand and knows that the next two characters are 'ab' it adds 'ba' as code 29 since it knows that this is what the encoder would have done. This is the case in `decode_update/3` where it has found an entry for the code.

The decode now continues with 'ab' as the last word it has decoded (28). When it reads the next code, 30, it is a bit surprised since 30 is not in the table. This is the magic, since it is not in the table it must be a code that the encoder just added to the table. Hmm, the encoder has read 'ab' and then reads the next character '?'. The sequence 'ab?' is not in the table so it adds this with a unique code ... and this is the code that we are looking at. The code that we are looking at 30 must therefore represent 'ab?' but that means that the sequence should be encoded 'a', 'b', 'ab', 'ab?'. What is the character '?', this is the character that the encoder read after having read 'ab' i.e. an 'a'!

So the word that the unknown code represents must start with the same characters that we have as our current word. If this is not magic, I don't know what is.