Lempel-Ziv Welch (LZW) compression

Lecture plan:

- Description of this refined compression scheme.
- Application in GIF's.
- Comparison with Huffman coding.

Part of this lecture is based on an article posted on a newsgroup a while ago (see e.g. `http://www.danbbs.dk/~dino/whirlgif/lzw.html`).

The original paper that describes the LZW algorithm is:

> Terry A. Welch. A Technique for High Performance Data Compression. IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.

The GIF format is described in more detail in

`http://ftp.funet.fi/pub/graphics/formats/gif.doc`

`http://ftp.funet.fi/pub/graphics/formats/gif89a.doc`

Generalia

LZW is a way of compressing data that takes advantage of repetition of strings in the data.

Since raster data usually contains a lot of this repetition, LZW is a good way of compressing and decompressing it.

LZW manipulates three objects in both compression and decompression: the input text, the (output) codestream, and a string table.

The string table is a product of both compression and decompression, but is never passed from one to the other.

Compression

The first thing we do in LZW compression is initialize a string table. If $\mathcal{A}$ is our alphabet we start by setting up a table in which a different code-word is assigned to each character of $\mathcal{A}$.

Example. If $\mathcal{A} = \{$ a, b, c, d $\}$ we may create the table

character	code word
a	000000
b	000001
c	000010
d	000011

Remarks

LZW compression algorithm works under the assumption that we can generate a large number of distinct codewords. In the example the codeword 000000 (six bits) for a may seem a waste, but binary codewords of length 6 allow us to encode 64 different strings.

Of course one could try to same more space by using a variable length code (e.g. Huffman code) instead.

These are implementation details that are left aside for now. From now on we assume that there are enough distinct codewords available. We will denote each of these by "#" followed by an integer order number. So the table on the previous page would be written as reported on the right

character	code word
a	#0
b	#1
c	#2
d	#3

Details of the compression process

Now we start compressing data. Let's first define something called the "current prefix". It's just a variable that we'll store things in and compare things to now and then. I will refer to it as `[.c.]`. Initially, the current prefix has nothing in it. Let's also define a "current string", which will be the current prefix plus the next character in the text. I will refer to the current string as `[.c.]K`, where `K` is some character.

Look at the first character in the text. Call it `P`. Make `[.c.]P` the current string. (At this point, of course, this is just `P`.) Now search through the string table to see if `[.c.]P` appears in it. Of course, it does, because our string table is initialized to have all single characters in the alphabet. So we don't do anything.

Now make `[.c.]P` the current prefix. Look at the next character in the text. Call it `Q`. Add it to the current prefix to form `[.c.]Q`, the current string. Now search through the string table to see if `[.c.]Q` appears in it. In this case, of course, it doesn't. Add `[.c.]Q` (which is `PQ` in this case) to the string table, and output the code for `[.c.]` to the codestream. Now start over again with the current prefix being just the root `Q`.

Keep adding characters to `[.c.]` to form `[.c.]K`, until you can't find `[.c.]K` in the string table. Then output the code for `[.c.]` and add `[.c.]K` to the string table.

In pseudo-code, the algorithm goes something like this:

```
[1] Initialize string table;
[2] [.c.] <- empty;
[3] while there is a character to read
[4]    K <- next character in text;
[5]    If [.c.]K is in string table
           [.c.] <- [.c.]K;
       else
           add [.c.]K to the string table;
           output the code for [.c.] to the codestream;
           [.c.] <- K;
[6] output the code for [.c.] to the codestream;
[7] delete the string table
```


Example

$\Sigma = \{$ A, B, C, D$\}$, $T =$ ABACABA.

We initialize our string table to: #0=A, #1=B, #2=C, #3=D.

The first character of T is A, which is in the string table, so `[.c.]` becomes A. Next we get B. `[.c.]`B (which is AB) is not in the table, so we output code #0 (for `[.c.]`), and add AB to the string table as code #4. [.c.] becomes B. Next we get `[.c.]`A = BA, which is not in the string table, so output code #1, and add BA to the string table as code #5. `[.c.]` becomes A. Next we get AC, which is not in the string table. Output code #0, and add AC to the string table as code #6. Now `[.c.]` becomes C. Next we get `[.c.]`A = CA, which is not in the table. Output #2 for C, and add CA to table as code#7. Now `[.c.]` becomes A. Next we get AB, which IS in the string table, so `[.c.]` gets AB, and we look at ABA, which is not in the string table, so output the code for AB, which is #4, and add ABA to the string table as code #8. `[.c.]` becomes A. We can't get any more characters, so we just output #0 for the code for A, and we're done.

The codestream is "#0#1#0#2#4#0".

Efficiency?

>>>> use a hashing strategy <<<<<

The search through the string table can be computationally intensive. Also, note that ”straight LZW” compression runs the risk of overflowing the string table - getting to a code which can’t be represented in the number of bits you’ve set aside for codes. There are several ways of dealing with this problem, and GIF implements a very clever one, but we’ll get to that.

Issue

An important thing to notice is that, at any point during the compression, if a string of the form `[...]K` is in the string table, then `[...]` is there too. This fact suggests an efficient method for storing strings in the table. Rather than store the entire string of K’s in the table, realize that any string can be expressed as a prefix plus a character: `[...]K`. If we’re about to store `[...]K` in the table, we know that `[...]` is already there, so we can just store the code for `[...]` plus the final character `K`.

Decompression

We again have to start with an initialized string table (this assumes we know the alphabet $\mathcal{A}$ and the codewords for each of its characters. The beauty of LZW, though, is that this is all we need to know at the beginning. We will build the rest of the string table as we decompress the codestream.

We need to define something called a *current code*, (referred to as `<code>`), and an *old-code*, which I will refer to as `<old>`.

Details

To start things off, look at the first code. This is now `<code>`. This code will be in the intialized string table. Output the corresponding character to the text. Make this code the old-code `<old>`.

Now look at the next code, and make it `<code>`. It is possible that this code will not be in the string table, but let’s assume for now that it is. Output the string corresponding to `<code>` to the codestream. Now find the first character in the string you just translated. Call this `K`. Add this to the prefix `[...]` generated by `<old>` to form a new string `[...]K`. Add this string to the string table, and set the `<old>` to `<code>`. Repeat from the beginning of the paragraph, and you’re all set (this is the most common case so you should understand this before going on).

What if <code> is not in the string table? (A case which can only occur for strings of the form P[...]P (for any character P).)

Think back to compression. What happens when P[...]P[...]PQ appears in the text? Suppose P[...] is already in the string table, but P[...]P is not. The compressor parses P[...], and find that P[...]P is not in the table. It outputs the code for P[...], and adds P[...]P to the string table. Then it gets up to P[...]P for the next string, and finds that P[...]P is in the table, as the code just added. So it outputs the code for P[...]P if it finds that P[...]PQ is not in the table.

The decompressor is always "one step behind" the compressor. When the decompressor sees the code for P[...]P, it will not have added that code to it's string table yet because it needed the beginning character of P[...]P to add to the string for the last code, P[...], to form the code for P[...]P. However, when a decompressor finds a code that it doesn't know yet, it will always be the very next one to be added to the string table. So it can guess at what the string for the code should be, and, in fact, it will always be correct.

So the decompression pseudo-code goes something like:

```
[1] Initialize string table;
[2] <code> <- first codeword;
[3] output the string for <code> to the text;
[4] <old> <- <code>;
[5] <code> <- next code in codestream;
[6] If <code> exists in the string table
       output the string for <code> to the text;
       [...] <- translation for <old>
       K <- first character of translation for <code>
    else
       [...] <- translation for <old>
       K <- first character of [...];
       output [...]K to text;
[7] add [...]K to the string table;
[8] <old> <- <code>
[9] go to [5] unless the codestream is empty;
```


Detailed simulation

$\mathcal{A} = \{$ A, B $\}$; #0 is the code for A, #1 is the code for B. Suppose we get the encode string

#0 #1 #2 #4 #0

instructions processed	<code>	<old>	[...]	K	output
[2] to [4]	#0				
''					A
''		#0			
[5]	#1				
[6] (then branch)					B
''			A		
''				B	
[7] and [8]					add "AB - #2" to string table
''		#1			
[5]	#2				
[6] (then branch)					AB
''			B		
''				A	
[7] and [8]					add "BA - #3" to string table
''		#2			
[5]	#4				
[6] (else branch)			AB		
''				A	
''					ABA
[7] and [8]					add "ABA - #4" to string table
''		#4			
[5]	#0				
[6] (then branch)					A
''			ABA		
''				A	
[7] and [8]					add "ABAA - #5" to string table
''		#0			

GIF variation

In part of the header of a GIF file, there is a field, in the Raster Data stream, called *code size*. This is the number of bits (currently) associated with the characters. The actual size, in bits, of the compression codes actually changes during compression/decompression, and I will refer to that size here as the *compression size*.

The initial table is just the codes for all the characters in the alphabet, as usual, but two special codes are added on top of those. The code size N is set to max(2,bits-per-pixel). In the table, the individual characters take up slots #0 through #($2^N - 1$), and the special codes are at #(2^N) and #($2^N + 1$). The initial compression size will be $N + 1$ bits per code.

Special codes

If you're encoding, you output the codes $N + 1$ bits at a time to start with, and if you're decoding, you grab $N + 1$ bits from the codestream at a time. As for the special codes: `CC` or the clear code, is 2^N, and `EOI`, or end-of-information, is $2^N + 1$. `CC` tells the compressor to re-initialize the string table, and to reset the compression size to $N + 1$. `EOI` means there's no more in the codestream.

How does it all work?

If you're encoding or decoding, you should start adding things to the string table at `CC` + 2. If you're encoding, you should output `CC` as the very first code, and then whenever after that you reach code #4095 (hex `FFF`), because GIF does not allow compression sizes to be greater than 12 bits. If you're decoding, you should reinitialize your string table when you observe `CC`.

Variable compression size

If you're encoding, you start with a compression size of $N + 1$ bits, and, whenever you output the code $2^{\text{compression size}} - 1$, you bump the compression size up one bit. So the next code you output will be one bit longer. Remember that the largest compression size is 12 bits, corresponding to a code of 4095. If you get that far, you must output `CC` as the next code, and start over. If you're decoding, you must increase your compression size AS SOON AS YOU write entry #($2^{\text{compression size}} - 1$) to the string table. The next code you read will be one bit longer. Don't make the mistake of waiting until you need to add the code $2^{\text{compression size}}$ to the table. You'll have already missed a bit from the last code.

The packaging of codes into a bitstream for the raster data is also a potential stumbling block for the novice encoder or decoder. The lowest order bit in the code should coincide with the lowest available bit in the first available byte in the codestream.

For example, if you're starting with 5-bit compression codes, and your first three codes are, say, `abcde`, `fghij`, `klmno`, where `e`, `j`, and `o` are bit #0, then your codestream will start off like:

```
byte \#0: hijabcde
byte \#1: .klmnofg
```


P.S.

You may have noticed that a compressor has a little bit of flexibility at compression time. I specified a "greedy" approach to the compression, grabbing as many characters as possible before outputting codes. This is, in fact, the standard LZW way of doing things, and it will yield the best compression ratio. But there's no rule saying you can't stop anywhere along the line and just output the code for the current prefix, whether it's already in the table or not, and add that string plus the next character to the string table. There are various reasons for wanting to do this, especially if the strings get extremely long and make hashing difficult. If you need to, do it.