LZW Compression using *CMS Pipelines*

Rob van der Heij
Origin IT Systems Management
P.O. Box 218, VA-5,
5600 MD Eindhoven, The Netherlands

c772503@nlevdpsb.snads.philips.nl
rob@e-mail.com
NLC2L52L at IBMMAIL

VM Workshop
June 1996

Introduction on compression algorithms

The general idea behind compression is to encode a string of characters such that it takes less bytes to store the compressed form than it would take to store the original characters. Apart from the simple algorithms like packing of bits in bytes there are basically two different “serious” compression algorithms.

Huffmann: Assign different codes to each of the characters such that the most common characters have the shortest code.

Lempel-Zev-Welch: Assign codes to represent each string of 2 or more characters and output these codes. The special thing about LZW algorithm is that the table is built implicitly by both encoder and decoder without actually transmitting that table in the coded data.

GIF specific issues The Graphics Interchange Format (GIF) uses a slight variation on the LZW algorithm by defining a string table of at most 4095 entries and adding two additional codes.**:** This paper shows an implementation of the LZW algorithm for GIF encoding using *CMS Pipelines* with bare hands (i.e. using only built-in stages, no REXX programming). The main reason for coding this in *CMS Pipelines* was the thrill of doing so as well as showing that it can be done. Though the code works, it probably not is the most cost effective way to execute the algorithm. Therefore I don't expect to be affected by the patent on the LZW algorithm that is held by Unisys (formerly Sperry) and IBM.

The Programs

The following sections show the *CMS Pipelines* code for both the encoder and the decoder, as well as two small test programs.

Though LZW is a general algorithm, I was really doing this for GIF encoding and decoding. Since GIF restricts the code size to 12 bits, several parts of the programs only use two bytes to represent a code. As long as there is a known upper bound on the code size, it should not be too difficult to change this.

Inspired by Donald E Knuth's “Literate Programming” I will present the program text interleaved with an explanation about what the code does (the actual program text is marked with a bar in the left margin). I also have a quick and dirty pipeline that extracts the program texts from my DCF input, this restricts me a bit in the order in which I present the program sections. I'm aware of that and I should be working on it, but it normally only gets a burden when I'm close to a deadline.

Note: The pipelines below do not yet handle the case where the string table overflows. It is properly defined in the GIF format how to handle that case, but it just isn't coded yet. It should not be too hard though.

ENGIFLZW REXX Encode a file

The LZW algorithm collects code values until it gets to a string that is new (not yet in the table). When that happens the code for the string without the last character (that was in the table) is output and a new string is started.

This “pixel depth” is the only value read in readable form, all other records contain the value of the pixel stored in two bytes. As soon as we know the code size we can output the ClearTable code (it is the convention to start with that). We also read in the first character to be encoded since that is treated differently (we don't have a prefix yet).

```
signal on error
'readto bits'                          /* Initial code length         */
ClearTable = 2**bits
EofCode = ClearTable + 1
'readto code'
```

We first need some things to stop the pipe when we're done. The gate will be closed at end-of-file on the primary input as well as on receiving a record on any of the other inputs of the faninany stage.

```
'callpipe (end ? name engif2) *: |',
    'f: fanout | hole | append literal |',
    'fg: faninany | g: gate',
  '? f: |',
```

The pipeline basically consists of a string table and a feedback that holds the current string. The current string is the concatenation of the previous string (the prefix) and the last code read. Since we output a code as soon as the current string is not in the table, this means that the prefix always is in the table. And because we only need to output the code for the prefix that is the only thing we have to remember. The entries in the table have the following form:

code	prefix	index

where

code the currently read value to be encoded

prefix the code for the prefix string (values read since last code was being output)

index the code for the prefix string followed by the code value

The following section combines the code just read with the code for the current prefix, matching the format of the records in the table. When the output of the spec is less than 4 byte we apparently got end of file on the primary input.

Note: We don't get end of file on the secondary input of the spec since that is in a feedback loop. Those short records are removed and the first one of them is used to do the closing of the encoder.

```
    's0: spec stop alleof 1.2 1',      /* Combine code                */
            'select 1 1.2 n |',        /* .. and prefix               */
    'end: totarget nlocate 3 |',
    'l: lookup 1.4 master |',
```

When we find a hit (the new string is already in the table) we just pick the index of that new string as the current prefix and continue the process of encoding. The faninany is used to collect the new value for the prefix when the string was not in the table.

```
    'copy |',
    'spec 5.2 1 |',                    /* Use the found code          */
    'f0: faninany |',
    'copy |',
    'preface var code |',
    's0:',                             /* .. as the prefix            */
```

When entries are added to the table we will now and then have to increase the code size being used. This is done by a juxtapose stage that gets the current length from a spec stage below (that counts the entries inserted in the table). To get things going we start the thing with a literal being the minimum code length used.

Note: The variable "bits" is the number of bits required for encoding the characters (i.e. the number of bits per pixel). Since we require two more characters defined (the ClearTable and the EoF) we start with a code size of one more. This means that for 3 bit per pixel we use the characters 0..7, have 8 and 9 for ClearTable and Eof, and start with 10 for the first string in the table. As soon as number 15 has been entered in the table we extend the code size to 5 bits. Both encoder and decoder will be aware of when this happens and also a character 7 will be output as the bit string "00111" after that has happened.

```
  '? literal' bits+1 '| spec 1-* d2c 1.2 ri |',
    'fj: fanin | j: juxtapose |',
```

The juxtapose now has prefixed the string of bits with the currently used code length. The c2v conversion will take the specified number of bytes from the start of the record so we have to assume the bits were presented to us in reverse order and undo that with another reverse before we output the data.

```
    'spec 1-* c2v 1 | reverse | *:',
```

The following fanin collects the codes that need to be written to the output. We start (by convention) with the ClearTable. The code to be output is converted to binary so we can use character based stages to cut the proper parts of the record. As explained above it must be passed through reverse to compensate for the other one.

```
  '? var ClearTable | spec 1-* d2c 3.2 ri |',
    'fx: fanin |',
    'spec 3.2 c2b 1 | reverse | j:',
```

Any records that are not yet in the table end up here. One copy of the record goes to the fanin to get written to the output.

```
  '? l: | copy |',
    'f2: fanout |',
    'fx:',
```

When we get to the end of the input stream we still have to write the code for the current string as well as the special Eof code.

Note: It is not trivial how we output the Eof code because it must be padded to the current code size. By feeding it into the pipeline this is handled automatically.

```
  '? end: | take 1 |',
    'spec 1.2 3 write x'd2x(EofCode,4)' 3 | fx:',
```

The following section takes a copy of the entries that were not yet in the table. The spec stage assigns sequence numbers to the records before they are inserted in the table. The g refers to the gate at the beginning of the pipeline and allows the feedback between the secondary output and tertiary input of the lookup to be broken.

```
   '? f2: |',
     'spec 1.4 1 number from' EofCode+1 'd2c n.2 ri |',
     'g: | copy |',
     'f3: fanout |',
     'l:',
```

A copy of the generated sequence numbers is also used to compute the current code size. The trick used here is to convert the number to binary and strip the leading "0" characters. The addrdw cms prefixes the record with a halfword length which is fed back to the juxtapose stage above.

```
   '? f3: |',
     'spec 5.2 c2b 1.16 | strip leading 0 |',
     'addrdw cms | spec 1.2 1 | fj:',
```

Another copy of the failed record is used to form the new prefix, so it is passed back to the top of the pipeline.

```
   '? f2: | spec 1.2 1 | f0:',

error: return rc * ( rc ¬= 12 )
```

The output of this pipeline are records with "0" and "1" that need to be converted to bits. This is can be done with the following pipeline stages (the reverse stages are needed because GIF format wants the codes packed from right to left).

```
reverse | fblock 8 | pad 8 0 | reverse | spec 1.8 b2c 1
```

For testing this readable form is rather convenient since it can be fed into the following "degiflzw" stage using

```
spec pad 0 1-* 1.16 ri | spec 1.16 b2c 1'
```

DEGIFLZW REXX Decode a LZW encoded file

This is the stage to decode a LZW compressed file. The input consists of the bit stream written in readable form (i.e. using the characters "0" and "1") where the bits are packed according to the GIF format).

```
'readto bits'
ClearTable = 2**bits
EofCode = ClearTable+1
```

The following is used to unpack the encoded bitstream. It needs to be integrated into the real decoding stage since they both run a counter for the number of codes received and this is going to cause problems when we overflow the table.

The parcel stage will read a length value on its secondary input and then copy that many bytes from its primary input to an output record. The following spec stages align it and convert it to a two-byte binary representation.

```
'addpipe (end ? name ungifx ) *: |',
     'reverse | p: parcel | reverse |',
     'spec pad 0 1-* 1.16 r | spec 1-* b2c 1.2 ri |',
     '*.input:',
```

Note: When each input record contains precisely one code (as output currently by the "engiflzw" stage) it also works since there are no bits that flow from one record to the other, so it does not matter on what side you start to pick them.

The code that produces the length values for the parcel above starts off with an infinite supply of records and numbers those. When that number is converted to binary the number of digits needed is equal to the number of bytes we need to parcel off the input. The addrdw cms will prefix the record with its legth, which is again converted to a readable form for parcel to read.

```
   '? literal  | dup * |',
     'spec number from' ClearTable '|',
     'spec 1-* d2b | strip leading 0 |',
     'addrdw cms | spec 2.1 c2d 1 | p:'
```

The gate below is set up similar to the code with the encoder shown before.

```
'readto'                              /* Clear Code                  */
'callpipe (end ? name degiflzw)',
     'fg: faninany |',
     'take 1 | g: gate',
```

The following spec combines the code just read with the sequence number and the current prefix. Since the encoder will add precisely one entry to its string table for each code being output, the decoder can do the same for each one it reads.

```
   '? *: |',
     's0: spec stop anyeof',
         '1.2 1',                     /* Code                        */
         'select 1 1.2 3',            /* Next                        */
         'select 2 1-* n |',          /* Prefix                      */
```

The record produced now is of the following form.

code	next	prefix

where

code The current compressed code just read from the input (16 bits)

next The sequence number of the next entry to be added to the string table.

prefix The output codes collected but not yet output (because we may have to add it to the string table)

When the code and the sequence number are the same we know it is a new entry, so it will not be in the table. The others are looked up in the table. The master record in the table has the entire string of codes associated with that entry (not just the prefix and the code as the encoder has) because we don't want to walk the tree for resolving it.

We start here by checking for the EoF code in the stream so we can stop the pipeline when that one is read.

```
     'end: totarget locate 1.2 x'd2x(EofCode,2) '|',
```

The following pick stage detects the case where the code refers to a code that cannot yet be in the table. This simplifies the rest of the pipeline a bit since we now know for sure there will alway be a match for the records fed into the lookup stage. The output, consisting of detail and master record, is used several times.

```
      'p: pick 1.2 ¬== 3.2 |',          /* Inside the table?          */
      'l: lookup 1.2 detail master |',
      'f2: fanout |',
```

The first time is to produce the output record (only using the master record).

The output of the decoder currently consists of two-byte records, the 16 bits being sufficient to keep the character (in the case of GIF 8 bits would even be sufficient). Maybe we should add a vchar to strip it down according to the specified maximum number of bits per character.

```
      'spec read 3-* 1 |',
      'fi0: faninany | *:',
```

Next is the pipeline that feeds the Eof code to the input of the gate to terminate the pipeline.

```
    '? end: | take 1 | fg:',
```

The following section is where the “roots” are entered in lookup (the codes from 0 to 2**bits-1).

```
    '? literal  | dup * |',
      'spec number from 0 d2c 1.2 ri |',
      't: take' ClearTable '|',
      'dup | join | l: |',
```

The secondary output of lookup has the codes that were not found in the table though we were sure they should be there. That indicates an error in the algorithm or in the data (may need a better way to handle that).

```
      'spec ,LZW error, 1 1-* c2x nw | cons',
```

After adding the roots to the table the other sequence numbers are passed to the spec stage that numbers the input codes.

Note: You may wonder why I only drop 1 sequence number though there are two codes (ClearTable and EoF) reserved by GIF. The reason for this is that the first code received is treated in a special way to initialize the algorithm so it does not hurt to use that as the initial prefix.

```
    '? t: | drop | s0:',
```

With the copy of the record that was found in the table we have to create the new entry that must be added to the table. That new entry is built by extending the prefix with the first byte of the code.

```
    '? f2: | spec 3-* 1 read 3.2 n |', /* pfx || w(tbl.code,1)       */
      'fi1: faninany |',
```

Since this is the pipeline that connects the secondary output of the lookup with its tertiary input, we need the g here to let gate break the loop.

```
      'g: | copy | l:',               /* Add to table               */
```

Another copy of the record found is used to create the new prefix.

```
    '? f2: | spec read 3-* 1 |',       /* pfx := tbl.code             */
      'fi2: faninany | elastic |',
      'strliteral // |',
      's0:',
```

The following deals with the code that was not yet in the table. When that happens we extend the prefix with a copy of its first byte. This new prefix is also output by the decoder. Since the entries in the table all need their index number as well, the section below will take that from the detail record too.

```
    '? p: | f1: fanout |',
      'spec 5-* 1 5.2 n | fi0:',       /* output pfx w(pfx,1)         */
    '? f1: | spec 3-* 1 5.2 n | fi1:', /* tbl <- nxt pfx w(pfx,1)     */
    '? f1: | spec 5-* 1 5.2 n | fi2:'  /* pfx := pfx w(pfx,1)         */
return rc
```

PIPEDLZW INPUT Sample input file

The following can be used as a simplified sample image of 20 pixels in 3 bit per pixel.

```
0  0  0  0  0
0  0  1  2  2
0  0  0  0  1
0  1  5  3  4
```

LZWTEST1 EXEC Test the encoder

This test encodes the input file shown above. When you look at the number of bits output by the encoder it shows that this does not really work well for very small input files (the output is 67 bits while the input would fit in 60 bits). If we duplicate the input file 100 times you get below 30% of the original size.

```
'PIPE < pipedlzw input | split | spec 1-* d2c 1.2 ri |',
     'literal 3 |',
     'engiflzw | > pipedlzw encoded a |',
     'spec 1-* 1.8 ri | join 7 | cons'
```

The output produced by the program is as follows:

```
 1000    0000    1010    1011    0000    0001    0010    0010
01100   00001   01101   00101   00011   00100   01001
```

When we use the tertiary output of the lookup stage in the “engiflzw” stage (and add the count keyword) the table is output when the stage terminates. The fifth line for example shows that code 11 represents a 0 prefixed by the string 10 (which again is a 0 prefixed by a 0) thus standing for the string of 3 consecutive zeroes.

```
code pfx  idx
   0    0   10
   0    1   18
   0    2   16
   0   10   11
   0   11   12
   1    0   13
   1   12   17
   2    1   14
   2    2   15
   3    5   20
   4    3   21
   5   13   19
```

LZWTEST2 EXEC Test the decoder

The file produced by “lzwtest1” can be decoded again using this test.

```
'PIPE (end ?) < pipedlzw encoded |',
     'reverse | fblock 8 | pad 8 0 | reverse |',
     'literal 3 |',
     'degiflzw |',
     'fblock 2 | spec 1.2 c2d 1.4 ri | join 4 , , | cons'
```

Conclusion

The code shown demonstrates that it actually is possible to code a fairly complex algorithm using *CMS Pipelines*. The “new input streams” of the lookup stage allow for a whole new world of applications to be piped.

The actual bare pipelines were written in an hour each. The fine tuning (like getting them to terminate properly, to produce the correct output) took another evening. Transforming the pipeline into something that could be presented in this paper was certainly the most time consuming.

Completely unrelated to this subject, I've again found that the Internet is an answer to many questions. Digging up the specifications for GIF89a was a matter of minutes. While looking for some information on LZW on the Internet I also ran into the site of Unisys that actually has a document called “LZW Patent Frequently Asked Questions” where I could find that they probably will not enforce their rights on the algorithm as long as my implementation is freeware.