LZW and GIF explained----Steve Blackstock

I hope this little document will help enlighten those of you out there who want to know more about the Lempel-Ziv Welch compression algorithm, and, specifically, the implementation that GIF uses.

Before we start, here's a little terminology, for the purposes of this document:

"character": a fundamental data element. In normal text files, this is just a single byte. In raster images, which is what we're interested in, it's an index that specifies the color of a given pixel. I'll refer to an arbitray character as "K".

"charstream": a stream of characters, as in a data file.

"string": a number of continuous characters, anywhere from one to very many characters in length. I can specify an arbitrary string as "[...]K".

"prefix": almost the same as a string, but with the implication that a prefix immediately precedes a character, and a prefix can have a length of zero. So, a prefix and a character make up a string. I will refer to an arbitrary prefix as "[...]".

"root": a single-character string. For most purposes, this is a character, but we may occasionally make a distinction. It is [...]K, where [...] is empty.

"code": a number, specified by a known number of bits, which maps to a string.

"codestream": the output stream of codes, as in the "raster data"

"entry": a code and its string.

"string table": a list of entries; usually, but not necessarily, unique.

That should be enough of that.

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.

For the moment, lets consider normal LZW encoding and decoding. GIF's variation on the concept is just an extension from there.

LZW manipulates three objects in both compression and decompression: the charstream, the codestream, and the string table. In compression, the charstream is the input and the codestream is the output. In decompression, the codestream is the input and the charstream is the output. The string table is a product of both compression and decompression, but is never passed from one to the other.

The first thing we do in LZW compression is initialize our string table. To do this, we need to choose a code size (how many bits) and know how many values our characters can possibly take. Let's say our code size is 12 bits, meaning we can store 0->FFF, or 4096 entries in our string table. Lets also say that we have 32 possible different characters. (This corresponds to, say, a picture in which there are 32 different colors possible for each pixel.) To initialize the table, we set code#0 to character#0, code #1 to character#1, and so on, until code#31 to character#31. Actually, we are specifying that each code from 0 to 31 maps to a root. There will be no more entries in the table that have this property.

Now we start compressing data. Let's first define something called the "current prefix". It's just a prefix 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 charstream. I will refer to the current string as "[.c.]K", where K is some character. OK, look at the first character in the charstream. Call it P. Make [.c.]P the current string. (At this point, of course, it's just the root P.) Now search through the string table to see if [.c.]P appears in it. Of course, it does now, because our string table is initialized to have all roots. So we don't do anything. Now make [.c.]P the current prefix. Look at the next character in the charstream. 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. Aha! Now we get to do something. Add [.c.]Q (which is PQ in this case) to the string table for code#32, and output the code for [.c.] to the codestream. Now start over again with the current prefix being just the root P. 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] K <- next character in charstream;
[4] Is [.c.]K in string table?
 (yes: [.c.] <- [.c.]K;
       go to [3];
 )
 (no: add [.c.]K to the string table;
      output the code for [.c.] to the codestream;
      [.c.] <- K;
      go to [3];
```

)

It's as simple as that! Of course, when you get to step [3] and there aren't any more characters left, you just output the code for [.c.] and throw the table away. You're done.

Wanna do an example? Let's pretend we have a four-character alphabet: A,B,C,D. The charstream looks like ABACABA. Let's compress it. First, we initialize our string table to: #0=A, #1=B, #2=C, #3=D. The first character is A, which is in the string table, so [.c.] becomes A. Next we get AB, which 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. So, the codestream is #0#1#0#2#4#0.

A few words (four) should be said here about efficiency: use a hashing strategy. The search through the string table can be computationally intensive, and some hashing is well worth the effort. 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.

An important thing to notice is that, at any point during the compression, if [...]K is in the string table, [...] is there also. 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.

Ok, that takes care of compression. Decompression is perhaps more difficult conceptually, but it is really easier to program.

Here's how it goes: We again have to start with an initialized string table. This table comes from what knowledge we have about the charstream that we will eventually get, like what possible values the characters can take. In GIF files, this information is in the header as the number of possible pixel values. The beauty of LZW, though, is that this is all we need to know. We will build the rest of the string table as we decompress the codestream. The compression is done in such a way that we will never encounter a code in the codestream that we can't translate into a string.

We need to define something called a "current code", which I will refer to as "<code>", and an "old-code", which I will refer to as "<old>". To start things off, look at the first code. This is now <code>. This code will be in the intialized string table as the code for a root. Output the root to the charstream. 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 [...]K to the string table, and set

the old-code <old> to the current code <code>. Repeat from where I typed the asterisk, and you're all set. Read this paragraph again if you just skimmed it!!! Now let's consider the possibility that <code> is not in the string table. Think back to compression, and try to understand what happens when you have a string like P[...]P[...]PQ appear in the charstream. Suppose P[...] is already in the string table, but P[...]P is not. The compressor will parse out P[...], and find that P[...]P is not in the string table. It will output the code for P[...], and add P[...]P to the string table. Then it will get up to P[...]P for the next string, and find that P[...]P is in the table, as the code just added. So it will output 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. If I am a decompressor, and I see code#124, and yet my string table has entries only up to code#123, I can figure out what code#124 must be, add it to my string table, and output the string. If code#123 generated the string, which I will refer to here as a prefix, [...], then code#124, in this special case, will be [...] plus the first character of [...]. So just add the first character of [...] to the end of itself. Not too bad. As an example (and a very common one) of this special case, let's assume we have a raster image in which the first three pixels have the same color value. That is, my charstream looks like: QQQ.... For the sake of argument, let's say we have 32 colors, and Q is the color#12. The

compressor will generate the code sequence 12,32,.... (if you don't know why, take a minute to understand it.) Remember that #32 is not in the initial table, which goes from #0 to #31. The decompressor will see #12 and translate it just fine as color Q. Then it will see #32 and not yet know what that means. But if it thinks about it long enough, it can figure out that QQ should be entry#32 in the table and QQ should be the next string output. So the decompression pseudo-code goes something like:

```
 [1] Initialize string table;
[2] get first code: <code>;
[3] output the string for <code> to the charstream;
[4] <old> = <code>;
[5] <code> <- next code in codestream;
[6] does <code> exist in the string table?
 (yes: output the string for <code> to the charstream;
       [...] <- translation for <old>;
       K <- first character of translation for <code>;
       add [...]K to the string table;        <old> <- <code>;  )
 (no: [...] <- translation for <old>;
      K <- first character of [...];
      output [...]K to charstream and add it to string table;
      <old> <- <code>
 )
[7] go to [5];
```

Again, when you get to step [5] and there are no more codes, you're finished. Outputting of strings, and finding of initial characters in strings are efficiency problems all to themselves, but I'm not going to suggest ways

to do them here. Half the fun of programming is figuring these things out!

Now for the GIF variations on the theme. In part of the header of a GIF file, there is a field, in the Raster Data stream, called "code size". This is a very misleading name for the field, but we have to live with it. What it is really is the "root size". 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 roots, as usual, but two special codes are added on top of those. Suppose you have a "code size", which is usually the number of bits per pixel in the image, of N. If the number of bits/pixel is one, then N must be 2: the roots take up slots #0 and #1 in the initial table, and the two special codes will take up slots #4 and #5. In any other case, N is the number of bits per pixel, and the roots take up slots #0 through #(2**N-1), and the special codes are (2**N) and (2**N + 1). The initial compression size will be N+1 bits per code. 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. 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>. The variable compression sizes are really no big deal. If you're encoding, you start with a

compression size of (N+1) bits, and, whenever you output the code (2**(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**(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**compression size) to the table. You'll have already missed a bit from the last code. The packaging of codes into a bitsream 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

So the differences between straight LZW and GIF LZW are: two additional special codes and variable compression sizes. If you understand LZW, and you understand those variations, you understand it all!

Just as sort of a 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.

Hope this helps out.----steve blackstock