Chapter 9. File Formats and Image Compression

9.1 Data Compression

Compression and Decompression. Consider a *stream* of data in the form of a large string of bytes

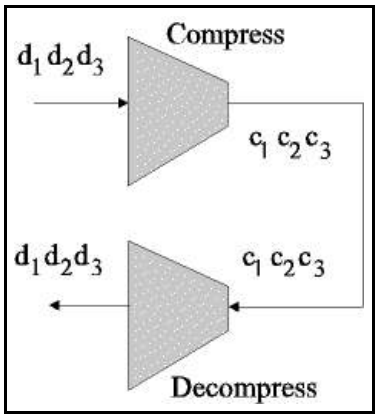
$$D = d_1 d_2 \ldots\ldots\ldots d_N \tag{9.1}$$

where N may be a few K (thousands) or may be hundreds of M (mega, or million). A *data compression* process accepts the bytes from the stream D and converts them to a shorter string

$$C = c_1 c_2 \ldots\ldots\ldots c_M \tag{9.2}$$

of bytes for which there is an inverse process for converting C back into D. We say that D is the data or dataset and that C is the *compressed* data. Figure 9.1 shows the processes of compressing and decompressing.

Figure 9.1. Data Compression



Compression always requires the input stream and yields the output stream. Often it also requires a *codebook*, i.e., a set of keys for doing the decompression. A contiguous block of bytes is represented by a smaller block of bytes in the compressed image. An algorithm or set of rules performs this conversion. The decompression requires that the smaller blocks of contiguous bytes be replaced with a larger set of blocks that it represents.

A *file* is a stream of data that has other information included, such as a *header*, or block of bytes at the front of the file to provide certain information about the file. The data compression and decompression process is one of converting files into smaller files that can be converted back into the original file, essentially. As we will see, we do not always get back the identical original file, but the degradation must be acceptable.

Compression Ratio and Loss. The relative size of the N and M in Equations (9.1) and (9.2) is called the *compression ratio*. More precisely, it is

$$r = N/(M + o) \tag{9.3}$$

where *o* is the overhead consisting of the number of bytes in a codebook, if one is used. For small or no codebooks, we take o to be zero and just use r = N/M. For example, if a 500KByte file is compressed with a compression ratio of 2, then the compressed file would contain approximately 250KBytes.

If the compression/decompression process restores a compressed image to the identical original file, then the process is called *lossless* because there is no loss in information. However, lossless compression schemes have low compression ratios, which may be from 1.1 to 2.5, depending on the data and the process. A compression scheme that restores data that is not the identical original data is said to be *lossy*. There are processes that are lossless with low compression ratio on the one hand and processes with high loss but high compression ratios on the other hand. Lossy compression ratios may be 9 or higher.

The tradeoff between lossless and lossy compression schemes is: we want to obtain the highest compression ratio possible with the smallest amount of loss. For *Internet* images that are for decoration and color rather than necessity, we need to send small image files for speed, which means that we choose a lossy method with high compression ratio. For displaying such images on a *World Wide Web* homepage we should keep the compressed file size from 9K to 25K bytes. But for medical images, where the details are crucial, we want lossless compression, so we must settle for a low compression ratio (and the resulting larger files). The tradeoffs are in the range between these extreme cases.

9.2 Some Common File Formats

Several file formats have arisen since the microcomputer age came into existence. One of the early ones was PCX that was developed by ZSoft Corporation of Marietta, Georgia for the purpose of desktop publishing. The more common file formats and their parent organizations are listed below.

PCX: ZSoft Corp., 450 Franklin Road, Suite 90, Marietta, GA 30067; 404-428-0008. The file format was developed for desktop publishing and image processing with *PC Paintbrush*, and it has been upgraded over the years. It uses from 1-bit to 24-bit color, is raster based with resolutions up to 64K by 64K. The data compression ratio varies with the data from about 1.1 to 1.5 using run length encoding (RLE), where a byte gives the number of bytes that have the same value and the next byte gives the value of the byte. The compression is lossless.

TIFF: Aldus Corp., 411 First Avenue South, Seattle, WA 9894; Microsoft Corp., 16011 NE 36th Way, Box 97017, Redmond, WA 98073. This was developed by Aldus and Microsoft and uses the *tagged image file format* that gives offsets into the file and the number of bytes to read for specific types of information about the image. The pixel data is written in raster scan order. The 1987 version is the most compatible in that many image display programs cannot correctly decompress the 1989 version. The compression is a type of RLE (run length encoding) and is lossless with low compression ratios.

TGA: Truevision Inc., 7340 Shadeland Station, Indianapolis, IN 46256. This was the first format to be able to use high color resolution. It uses 24-bit pixels and has the *true color* range of 16,777,216 colors. This requires 3 bytes per pixel in raster scan order and so the files are very large. Compression implements a different RLE from TIFF and PCX.

JPEG: Joint Photographic Experts Group, or JPEG, Littleton, MA. This uses multiple formats from 8-bit gray scale to 24-bit color. The compression uses the discrete cosine transform (DCT), or the real part of the DFT. Blocks of rxr pixel values are replaced with a set of cosine transform coefficients, from which the inverse cosine transform can be made. The compression is lossy.

GIF: CompuServe Inc., 500 Arlington Center Blvd., Columbus, Ohio 43220; 614-457-8650. This format is used by internet viewers such as Netscape and Internet Explorer. This was used originally in desktop publishing with 1 to 8 bit color pixels for raster scan order pixels. The compression builds a code table (codebood) as it goes for compression but does not store a codebook. Instead, the decompression builds its own code table during processing for successful decompression without loss. The compression scheme is based on the Limpel-Ziv-Welsh (LZW) algorithm and is patented by Unisys of Minneapolis.

BMP: Microsoft Corp., 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717. This uses raster scan pixel values of 1 to 8 bit color that are compressed by RLE. There is also a 24 bit true color mode where no compression is used.

9.3 A Simple Example: PCX

The Header. This early standard for the program *Paintbrush* has been heavily used by other "paintbrush" types of programs. Many images exist in this format. A **.pcx* file consists of

i) a header of fixed length, ii) the image data, iii) an extended color palette structure

The file header is of exactly 128 bytes, which have file addresses (offsets) of 0, 1,..., 127. It provides the following information. Taable 9.1 presents the interpretation of the bytes in the header.

Table 9.1. The PCX File Header Structure

Byte No.	Data	Meaning of Information
0	10 (decimal)	Must be 10 (0A hex)to denote PCX format.
1	0 => Version 2.5	
	2 => Version 2.8 w/palette information	
	3 => Version 2.8 w/o palette information	
	5 => Version 3.0	
	2	Version 2.8 with palette.
	3	Version 2.8 or 3.0 without palette.
	5	Version 3.0 with palette.
2	1	Simple run length encoding (RLE).
3	1	EGA/VGA 16 color modes.
	2	CGA 4 color mode.
	8	VGA 256 color mode.
4-11	four short integers	8 bytes define top-left and bottom-right (2 bytes each) corners, e.g., (0,0) and (799,599) for 800x600 are in the order 0 0 799 599 (we designate these as Xmin, Ymin and Xmax, Ymax).
12-13	N	2 bytes for horizontal resolution of creating device, such as 640.
14-15	M	2 bytes for vertical resolution of creating device, such as 480.
16-63	RGB	Color map of 48 bytes that form 16 triplets of 3-byte color pixel values with a pixel for each of R, G and B, so there are only 16 colors (16x3 = 48). For 256 colors (triplets), there are 256x3 = 768 bytes in a color table at the end of the file, 769 bytes from the end where 0C hex designates that the next byte starts the color table
64	0	Reserved.
65	b	Number of bit planes, NPlanes: 4 for 16 color, 1 otherwise
66-67	B	Bytes per scan line for each color (plane). It must be an even number compatible with above window size given.
68-69	1	Interpret palette as color or B&W.
	2	Interpret palette as gray scale.
70-127		Filler blanks to fill out the 128 byte header.

When the 256 color mode is to be used (Byte 3 has the value *8*), the 768 byte triplets of R, G and B byte values are stored starting from 769 bytes from the end of the file. To access these, the file must be advanced to the end of the file and then backed up exactly 769 bytes to read a tag. If the tag is 0Ch (12 decimal) then the 256 color byte triplets follow.

To decode a PCX file, first find the pixel dimensions of the image by calculating [XSIZE = Xmax - Xmin + 1] and [YSIZE = Ymax - Ymin + 1]. Then calculate how many bytes are required to hold one complete uncompressed scan line: TotalBytes = NPlanes * BytesPerLine. Note that since there are always an integral number of bytes, there will probably be unused data at the end of each scan line. TotalBytes shows how much storage must be available to decode each scan line, including any blank area on the right side of the image. You can now begin decoding the first scan line - read the first byte of data from the file.

RLE Encoding. We assume here that we are working with 8 bits per pixel value (256 colors, where each 8-bit value is the address one of the 256 color registers that contain 3 6-bit values for R, G and B, according to the discussion in Chapter 1). If two or more consecutive pixel values are the same, these identical pixel values are called a *run*. In writing the pixel byte-values in sequence according to the raster scan order, only consecutive bytes that are different from those adjacent to it are written to the output as pixel bytes. For any run, two bytes are written. The first is a byte that has the two most significant bits set to 1. The next 6 bits give the number of bytes in the run. The second byte gives the pixel value to be repeated. For example, if 8 bytes have the same value of 127, then we designate it by writing the following to the output PCX file.

11001000	01111111
Byte 1	Byte 2

Byte 1 contains 11 in the two highest order bits, which is 0C hex (12 decimal) to designate that this is a *run indicator* byte. The 6 lower order bits have the value 8 decimal to indicate that there are 8 pixel values of the value the is to appear next. Byte 2 is the value that is to appear 8 times in the displayed image. Because these 6 lower order bytes in a run indicator byte range from 0 to 63, the run lengths can be 0 to 63. Longer runs must be broken at 63 and a new run started.

A problem arises here. If a pixel has a value of 192 or greater, then it would be interpreted as a run indicator byte. Thus every pixel value of 192 or more must have a run created for it of length one with a run indicator byte followed by the pixel value of 192 or greater. This means that about 25% of the upper pixel values are more costly to encode if not in a run or length 2 or more. For this reason, the RLE encoding has a low compression ratio. Another reason that RLE has low compression ratios is that an image of great variation has few and short runs. The encoding is, however, lossless. For example, a one-time pixel value of 192 (11000000) would require the encoding of 11000001 11000000, while a one-time pixel value of 191 (10111111) would require only the one byte 10111111.

9.4 The TIFF File Format Scheme

The Tagged Image File Format Blocks. Aldus and Microsoft jointly supported the TIFF specification and made it very flexible (perhaps too flexible). A TIFF file starts at Byte 0 and continues in a stream to the end, but is grouped into blocks of bytes according to the following parts.

Offset Address in Bytes
0 1 2 ...7 8 9 ....EoF

Header	Image-File-Directory	Directory-Entry	Data

The Header. The Header bytes are listed in order below according to the way they are grouped.

Byte Number	Meaning
0 - 1	2 ASCII bytes to indicate the order of significance of the bytes, *II* for Intel or *MM* for Motorola (Intel format: 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 — Address N Least Byte, Address N+1 Most Byte (significant)) (Motorola: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 — Address N Most Byte, Address N+1 Least Byte (significant))
2 - 3	42 ASCII, the original version number (not the actual version number)
4,5,6,7	Address in number of file offset bytes to the first Image File Directory (usually there is only one image per file, so this would be the only Image File Directory

The Image File Directories. The first Image File Directory, which is the only one in the usual case where there is a single image stored in the file, consists of 12 bytes that are described as follows.

Byte Number	Meaning
0 - 1 (2 bytes)	Number of Directories in the Image File Directory
2 -13 (12 bytes)	First Directory entry (12 bytes)
14-25 (12 bytes)	Second Directory entry (12 bytes)
..	.. ..
N to N+11 (12 bytes)	Last Directory entry (12 bytes), N = 2 + 12*k for k-th Directory entry
N+12 to N+15 (4bytes)	Address of next Image File Directory (in case of more than one image in file)

The Directories for an Image File Directory. In the Image File Directory above (for the first and likely the only image), there is a sequence of Directories. Each such Directory has provides a tag number that prescribes a property of the image. Several of these Directories with their tags prescribe precisely the image properties. A Directory has the following bytes and meanings.

Byte Number	Meaning
0 - 1	Tag number that is a 2-byte integer
2 - 3	Field type: BYTE = unsigned 8-bit integer, ASCII = character, SHORT = 16-bit unsigned integer, LONG = 32-bit unsigned integer (pixel data)
4 - 7	Field length: the number of data items (of one or more bytes) in the field data
8 -11	Address in file offset bytes of the data

Tags. We list some commonly used tags below that can be used for the usual grayscale or color images.

Tag Number	Meaning	Type
256	Image Width	SHORT/LONG
257	Image Length (Height)	SHORT/LONG
258	Bits/Sample	SHORT
259	Compression (none, RLE, LZW or JPEG) 1=>none, bytes are packed; 2=>CCITT Huffman 3=>CCITT Group 3; 4=>CCITT Group 4 5=>LZW; 32773=> PackBits (Macintosh)	SHORT
262	Photometric Interpretation (0 => black)	SHORT
270	Image Description (text)	ASCII
273	Strip Offset (offset address to strip [multiple rows] of pixel data)	
277	Samples per Pixel	SHORT
278	Rows per Strip	SHORT/LONG
279	Strip Byte Count	SHORT/LONG
282	X-resolution	RATIONAL
283	Y-resolution	RATIONAL
296	Resolution Unit (inches, cm or none)	SHORT
320	Color Map (reds first, the blues, then greens)	

The RATIONAL type of data consists of two LONG types, of which the first ... and the second ... . TIFF images are classified as Class G (grayscale), Class P (color mapped) or Class R (full RGB color). The classes and their required tag numbers are given below.

Class	Tag Numbers
G	256, 257, 258, 259, 262, 273, 278, 279, 282, 283, 296
P	256, 257, 258, 259, 262, 273, 278, 279, 282, 283, 296, 320
R	256, 257, 258, 259, 262, 273, 277, 278, 279, 282, 283, 296, 320

An Actual TIFF Image. Here we take a look at the front part of the file *san_fran.tif* to see what can be seen by displaying all bytes as hexadecimal data.

Bytes (each byte is 2 hexadecimal characters)	Meaning
49 49 2A 00 18 00 00 00	"II" for Intel, ver. 42, Offset to Dir.=>00000018hex => 24decimal (Intel: rightmost byte is least significant)
2C 01 00 00 01 00 00 00	non-data
2C 01 00 00 01 00 00 00	non-data
0D 00 FE 00 04 00 01 00	"OD" is 24-th byte offset from file beginning 000Dh = 13 =>no. entries: first directory starts at 00 FE = 254 (tag number), 0004 = field type (long integer)
00 00 01 00 00 00 00 01	0000001 = length of field item.
03 00 01 00 00 00 80 02	
00 00 11 01 04 00 04 00	01 11 hex => 273 for tag number
00 00 C0 B0 04 00 15 01	B0C0 00 04 is offset to tag value where pixel strip data starts
03 00 01 00 00 00 01 00	
00 00 16 01 04 00 01 00	
00 00 7A 00 00 00 17 01	
04 00 04 00 00 00 D0 B0	
04 00 1A 01 05 00 01 00	
00 00 08 00 00 00 1B 01	
05 00 01 00 00 00 10 00	
00 00 2B 01 03 00 01 00	

9.5 The GIF File Format Scheme

LZW Compression. This scheme is based on the Limpel-Ziv-Welsh algorithm. In 1977, Abraham Limpel and Jacob Ziv published an algorithm for compression that was refined by Terry Welsh in 1984 so that the compression and decompression could generate the same codebook. The algorithm was considered public domain but after it was used by CompuServe over a few years to compress images for transmission on the *Internet* and other networks, Unisys of Minneapolis announced that it had a patent on the algorithm and that it required royalties for its use (this is strange, because the US Patent Office does not patent algorithms unless they are a part of a system that has real physical parts, some of which function according to the algorithm).

The most common case is where the pixels have byte values for 8-bit grayscale or 256 colors and n = 12 is the size of the codewords. First, a partial codebook is initialized that gives the binary integer values for the first 256 codewords, 0 through 255, that represent 8-bit values, or *characters*. The process reads the image data starting with the top-left pixel and proceeds reading one character at a time in the raster scan order. The single characters are considered to be strings of length 1 and are called *roots*. The algorithm builds up a code table as it goes by adding more and longer strings that are represented by 12-bit codewords.

The algorithm first sets the current prefix, designated here by @ to NULL. Then it reads the first current character C from the input stream and concatenates it on the right with @ to form the current string # = @ + C = NULL + C = C, where "+" represents the concatenation procss. NULL is not written to the output stream as it is not represented as a 12-bit codeword. C becomes the current prefix @. The algorithm reads the next character into C and concatenates it with the current prefix @ to obtain the new current string # = @ + C. This time it writes the 12-bit codeword for the character C to the output stream and puts a new entry in the table: codeword = next available 12-bit integer, string = #. The new current prefix becomes the character C via @ = C (what was in @ was output in the form of a codeword).

The process continues with all new strings being written to the codebook with new 12-bit codewords, and all prefixes are looked up in the codebook and their codewords are written to the output stream. The criterion for a new current string # is: is it already in the table? If it is not, then the codeword for the prefix is written to the output stream and C becomes the new current prefix. If the new current string is in the codebook, then nothing is written to the output stream and the new current prefix @ becomes the current string #. The next current character C is then read and concatenated on the right with @ to form a new extended string #. This is repeated until there are no more characters, that is, *End-of-File* (EOF) symbol is read and then the *End-of-Information* (EOI = *257*) symbol is written to the output stream. The 12-bit codewords are written to a GIF file (with a name of the form *filename*.gif).

In the usual case the value n = 12 is used, so that 4096 different strings can be encoded, including the single character strings from 0 to 255. When all 4096 codewords are used up, an *End-of-Table* (EOT, which is codeword *256*) symbol is written to the table and a new code table is initialized for the image data from that point on.

The Algorithm for LZW Compression

```
Step 1: initialize table of codewords from 0 to 257;
            @ ← NULL;                                   /Initialize prefix/

Step 2: read next character C from input stream;        /Read from input stream/
            if C = EOF then                                 /If end-of-file, stop/
               write EOI to output stream;
               stop;
            else
               # ← @ + C;                                 /Else concatenate char with prefix/

Step 3: if # in code table then                         /If string # is in code table/
               @ ← #;                                       /then build larger string/
               go to Step 2;                                /Continue with next character/
            else
               if  table is full then
                     write @ to output stream;
                     write EOT to output stream;
                     initialize new code table from 0 to 257;
                     @ ← C;
                     go to Step 2;
               else
                     put # in table paried with next consecutive codeword;
                     write codeword for C to output stream;
```

Figure 9.2 displays the algorithm flowchart. As each byte character (pixel value) is read, a *current string* denoted by # is constructed as follows: the current character read is concatenated on the right of the current prefix @ to form a new current string #. A search of the code table (codebook) is made for this current string. If # is found in the table, then the current prefix is set equal to the current string and another input pixel value is read and put through the same procedure. If # were not found in the table, then the current string is added to the table and the codeword for this string is output to the encoded GIF file. In this case, the current prefix

@ is now set to the current character. This process is repeated on each input character until all of them have been processed. At this point, the code for the current prefix is written to the output file as is an *End-of-Information* (EOI symbol).

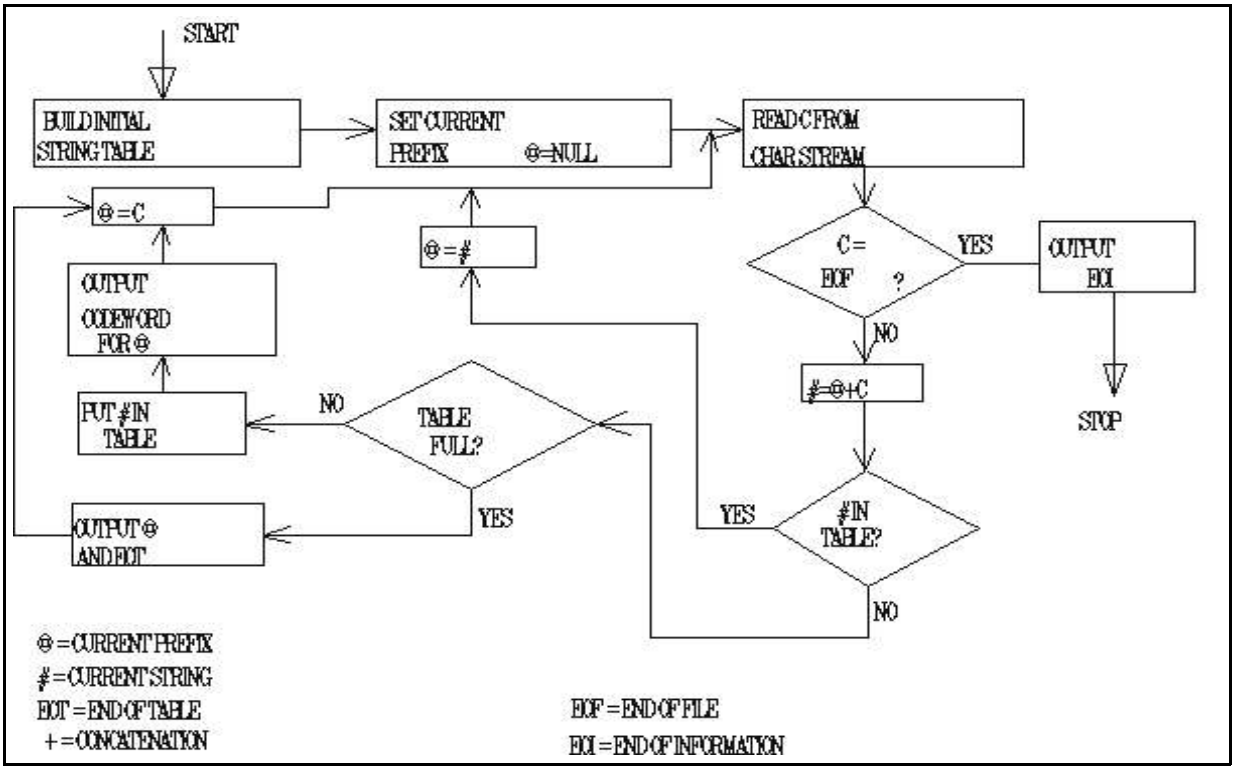


Figure 9.2. The LZW Compression Algorithm Flowchart.

A Simple Example of LZW Compression. Consider the following stream of data for which the compression process is done in Table 9.2. Assume that 0 - 255 have been used for characters and that 256 and 257 have been used respectively for EOT and EOI. In the table, the angle brackets <> enclosing a string denote the 12-bit codeword for the string inside of the brackets.

Input Data Stream: /$10.00/$22.00/$12.10/$222.00<EOF>

This is a very short stream of data that is to be encoded by the same process as the LZW algorithm for illustrative purposes. We assume here that the 12-bit codewords 0 - 255 have already been used in the codebook. The codewords *256* and *257* designate respectively the EOT and the EOI. The output stream of codewords from the output column is

</><$><1><0><.><0><0>258<2><2> 262 264 259 267 260 265 266 268<EOI>

Table 9.2. LZW Compression of the Example Data Stream

Next Input Char. C	Current Prefix @	String # = @+C	Output Codeword	New Current Prefix @	New Table Entry
/	NULL	/	--	/	--
$	/	/$	</>	$	258=</$>
1	$	$1	<**$**>	1	259=<$1>
0	1	10	<**1**>	0	260=<10>
.	0	0.	<**0**>	.	261=<0.>
0	.	.0	<**.**>	0	262=<.0>
0	0	00	<**0**>	0	263=<00>
/	0	0/	<**0**>	/	264=<0/>
$	/	/$	--	/$	[in table]
2	/$	/$2	**258**	2	265=</$2>
2	2	22	<**2**>	2	266=<22>
.	2	2.	<**2**>	.	267=<2.>
0	.	.0	--	.0	[in table]
0	.0	.00	**262**	0	268=<.00>
/	0	0/	--	0/	[in table] $
0/	0/$	**264**	$	269=<0/$>	
1	$	$1	--	$1	[in table]
2	$1	$12	**259**	2	270=<$12>
.	2	2.	--	2.	[in table]
1	2.	2.1	**267**	1	271=<2.1>
0	1	10	--	10	[in table]
/	10	10/	**260**	/	272=<10/>
$	/	/$	--	/$	[in table]
2	/$	/$2	--	/$2	[in table]
2	/$2	/$22	**265**	2	273=</$22>
2	2	22	--	22	[in table]
.	22	22.	**266**	.	274=<22.>
0	.	.0	--	.0	[in table]
0	.0	.00	--	.00	[in table]
<EOF>		**268**+<**EOI**>	--		

The LZW Decompression Algorithm. To decompress a GIF file we initialize the basic string table of roots as was done in the compression algorithm using 0 - 257 (256 and 257 are used for the EOT and EOI, respectively). Now we create a variable *old_code*, denoted by *%* here, to hold the previous input codeword. We initialize this via % = NULL. The current string to be written to the tale is #. Table 9.3 presents a flowchart of the decompression process.

The Algorithm for LZW Decompression

Step 1: initialize new code table 0 to 257;
% ← NULL; //Initialize old_code

Step 2: read next codeword C from input stream (of LZW codewords);
translate C to character;
if C = EOI then stop;
if C = EOT then reinitialize basic string;
else
write C to output stream;
if C decodes to a single character string then % ← C; # ← % + C;

```
    else
          c ← first_character(C);      # ← % + c;    % ← C;
    write # to code table;
go to Step 2;
```

A Simple Example of LZW Decompression. Table 9.3 below decompresses the compressed data from the above compression procedure.

Table 9.3. LZW Decompression of the Example Encoded Data Stream

Input Stream (Codewords)	Character String C	Old Variable %	String # = %+C	String C to Output Stream	# String Table
</>	/	NULL	/	**/**	[in table]
<$>	$	/	/$	**$**	258=</$>
<1>	1	$	$1	**1**	259=<$1>
<0>	0	1	10	**0**	260=<10>
<.>	.	0	0.	**.**	261=<0.>
<0>	0	.	.0	**0**	262=<.0>
<0>	0	0	00	**0**	263=<00>
258	/$	0	0/$	**/$**	264=<0/>
<2>	2	<258>	/$2	**2**	265=</$2>
<2>	2	2	22	**2**	266=<22>
262	.0	2	2.0	**.0**	267=<2.>
264	0/	<262>	.00/$	**0/**	268=<.00>
259	$1	<264>	0/$$1	**$1**	269=<0/$>
267	2.	<259>	$12.0	**2.**	270=<$12>
260	10	<267>	2.010	**10**	271=<2.1>
265	/$2	<260>	10/$2	**/$2**	272=<10/>
266	22	<265>	/$222	**22**	273=</$22>
268	.00	<266>	22.00/$	**.00**	274=<22.>
<EOI>				**<EOF>**	

The input stream of codewords to the decompression process is

</><$><1><0><.><0><0>258<2><2> 262 264 259 267 260 265 266 268<EOI>

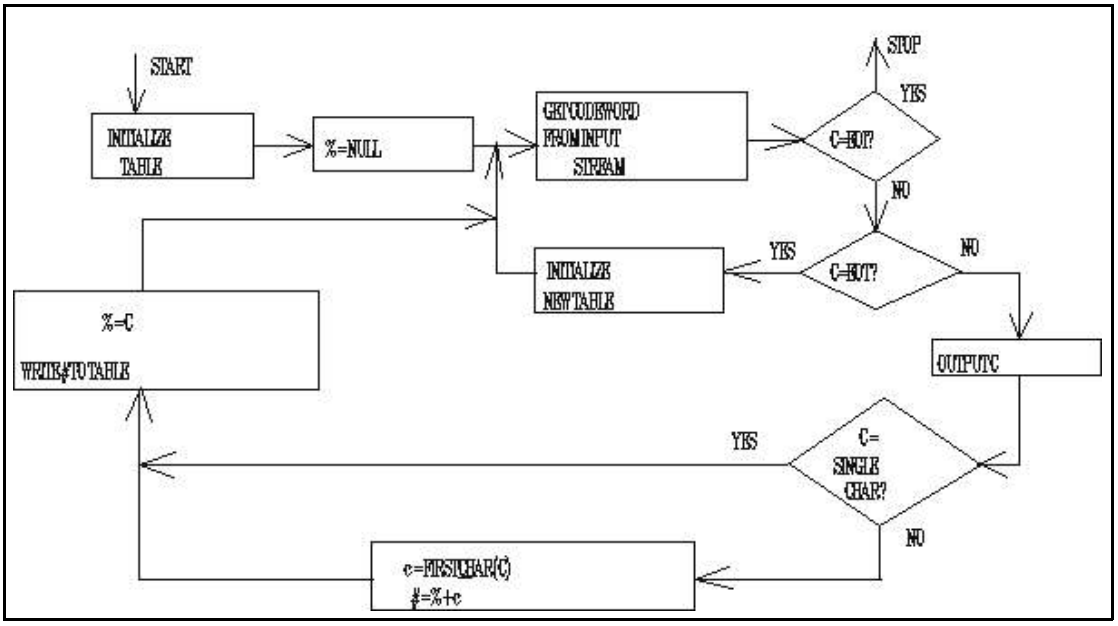
The output stream of decoded characters is

/$10.00/$22.00/$12.10/$222.00

This is identical to the original data stream. Thus there has been no loss of information.

Figure 9.3 presents a flowchart for the process of decompression. In a sense, it is the an analogous process to the compression process.

Figure 9.3. The LZW Decompression Flowchart.



GIF File Structure. GIF ("jif") files have a *file header*, a *logical screen descriptor*, a *global color palette* (optional), one or more *image data section*, and a *file trailer*. In the definitions given below, the byte number is the file offset (address).

File Header (6 bytes)

Bytes 0-2:	GIF	(the ASCII characters "GIF" appear here in that order)
Bytes 3-5:	Version	(this is either "87a" or "89a" for 1987 or 1989 versions)

Logical Screen Descriptor

Bytes 5-7	Screen Width	(this gives the screen width)
Bytes 8-9	Screen Ht.	(this gives the screen height)
Byte 10	n	(this packed fields byte provides the following information: Bits 0-2, $3*2^{n+1}$ bytes is size of global palette Bit 3, sort flag (0 → not sorted, 1 → sorted palette) Bits 4-6, number bits - 1 per primary color Bit 7, global palette flag (0 → not present, 1 → present)
Byte 11	BkClr	Background color index
Byte 12	Aspect	Pixel aspect ratio = (Aspect + 15)/64 if Aspect not zero

Global Color Paltette

If Byte 10 of the logical screen descriptor above has Bits 0-2 not all set to 0, then the global color palette follows immediately starting at Byte 13 in the file. It consists of triads for the R, G and B primary colors so that 16 colors would occupy 48 bytes and 256 colors would occupy 768 bytes.

Image Data Section

Byte 0 (of this section)	always 2Ch as an image separator
Bytes 1-2	image position at left in pixels
Bytes 3-4	image position at top in pixels
Bytes 5-6	image width in pixels
Bytes 7-8	image height in pixels
Byte 9	packed fields byte n is in bits 0-2, $3*2^{n+1}$ bytes = size of local palette bits 3-4 are reserved bit 5, sorted flag (0 → not sorted, 1 → sorted palette) bit 6, interlaced flag (0 → not interlaced, 1 → interlaced) bit 7, local pallete flag (0 → not present, 1 → present
Bytes 10-?	if local color palette is present, it follows here in the form of triplets of bytes for R, G and B for each color
Byte ?-?+1	the LZW minimum code size goes here, which is the number of bits in the codebook
Blocks	the first byte of each block of image data gives the size of the block, which can not exceed 256 bytes. Each data block contains LZW encoded pixel values in raster scan order from the top-left to the bottom-right in left-to-right, top-to-bottom fashion. The codes are packed from right to left: the 6-bit codes aaaaaa bbbbbb cccccc would appear as: Byte 1 = bbaaaaaa; Byte 2: ccccbbbb; etc.

The Trailer

This is always the single byte 3Bh that terminates the GIF data stream.

An Example of a GIF File. The following data is taken from the *san_fran.gif* file, which is a GIF image file with 256 colors. An examination of the data with a text editor shows the various parts of the file.

File Byte Offset	Hex Value	Notes
0	47	"G" ASCII code
1	49	"I"
2	46	"F"
3	38	"8"
4	37	"7"
5	61	"a"
6	80	lower byte hexadecimal value of raster width
7	02	upper byte of raster width: 02 80 = $2x16^2 + 8x16 = 640$
8	A6	lower byte hexadecimal value of raster heightf
9	01	upper byte of raster height: $1x16^2 + Ax16 + 6 = 422$
10	97	1001 0111: global color palatte, xxxx x111 = 8 bit color pixels
11	00	background color
12	00	aspect ratio, if given
13	DB	Red_1 (256 RGB byte triplets begin here for color palette)
14	7A	Green_1
15	5D	Blue_1
16	92	Red_2
17	55	Green_2
18	4D	Blue_2
19	7A	:
20	45	:

21	45	:
22	65	:
:	:	:
778	FF	Red_256
779	FF	Green_256
780	FF	Blue_256 (end of 256 colors, 768 bytes)
781	2C	"," (image separator)
782	00	LSB left edge of image [image descriptor]
783	00	MSB left edge of image
784	00	LSB top edge of image
785	00	MSB top edge of image
786	80	LSB image width in pixels
787	02	MSB image width
788	A6	LSB image height in pixels
789	01	MSB image height
790	07	Flag: 0xxx xxxx → no local color table; x0xx xxxx → no interlacing
791	08	8-bit colors are encoded in the following image data blocks
792	FE	254 = byte count of first image data block
793	00	code byte 1 (not a complete codeword): 00991C has 24 bits, split into 3 12-bit
794	99	code byte 2 (not a complete codeword) codewords, decompress, and when
795	1C	code byte 3 (not a complete codeword) block has been decoded, display

9.6 Targa Image Files

...

9.7 BMP Images Files

The Headers and the File Header. The BMP image file format was specified by Microsoft for ease of use and it has a structure somewhat similar to PCX. The file sections are

---

File Header, Image Header, Color Table, Pixel Data

We consider the File Header first. Its fields are described below.

1. 2-bytes: BM designate the file type
2. 4-bytes: file size in bytes
3. 2-bytes: reserved
4. 2-bytes: reserved
5. 4-bytes: offset to start of pixel data

The Image Header. The Image Header is larger than the File Header and provides information about the image. Its fields are listed below. There is a separate type of Image Header for the OS/2 operating system, but we give only the MS Windows version here.

Field 1	4-bytes:	header size (at least 40 bytes)
Field 2	4-bytes:	image width
Field 3	4-bytes:	image height
Field 4	2-bytes:	value must be 1
Field 5	2-bytes:	bits per pixel (1, 4, 8, 16, 24, 32)
Field 6	4-bytes:	compression type (may be none)
Field 7	4-bytes:	image size (0 if not compressed)

Field 8	4-bytes:	pixels/meter resolution in X-direction
Field 9	4-bytes:	pixels/meter resolution in Y-direction
Field 10	4-bytes:	number of entries in color map actually used
Field 11	4 -bytes:	number of significant colors

The Color Header. The Color Table (also called *color palette*) immediately follows the Image Header. It can be in one of three different formats. The first two map pixel data to RGB color values when the number of bits used per pixel is 1, 4, or 8. For BMP files in the Windows format, the palette consists of an array of 2^n structures (the second format is for OS/2). The BMP Windows format is

1. 1- byte: blue color value
2. 1-byte: red color value
3. 1-byte: green color value
4. 1-byte: reserved

The third color palette format is not really a color mapping. If the number of bits used is 16 or 32 and the value in the Compression Type field of the Image Header is 3, then in place of the RGB structure shown in the Color Palette above, there is an array of three 4-byte integers. These are bit masks that specify the bits
used for red, green and blue, respectively. The nonzeros must be contiguous in each mask. Images that use 24 bits, and 16 or 32 bits without the Compression Type field set 3 do not have a color palette. As an example, in a 32 bit image the three 32-bit values are

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

These specify 10-bit nonzero values for each of the colors red, green and blue.

The Pixel Data. The pixel data follows the Color Table if it is present, but if it is not present, then it follows the prescription given in the Image Header. The pixel data usually follows the previous headers immediately, but sometimes there may be filler before it and it is always a good idea to use the offset given in the Image Header. This offset is the address in bytes from the beginning of the file, where the first byte is offset 0.

The pixel rows are order from bottom row to top row. The number of rows is given in the Image Header. The number of pixels in a row is given from image width and bits-per-pixel numbers. The number of bytes per row is rounded up to a multiple of 4 and is found from

bytes-per-row = {[widthxbit-count + 7]/8 + 3 } / 4

The format of thepixel data depends upon the number of bits per pixel.

1 or 4 bits/pixel:	each data byte is either 8 or 2 fields that are indices into the color palette (the leftmost bit is the most significant)
8 bits/pixel:	each byte is an index into the color palette (register)
16 bits/pixel:	each 2-byte integer represents a pixel; if the compression type is 0 then the intensity of each color is represented by 5 bits with the most significant bit not being used (3x5 = 15 with 1 bit left); if the compression type is set to 3 (?), then three 4-byte bit masks used as described above (red, blue and green in order)
24 bits/pixel:	each pixel is represented by three consecutive bytes that specify blue, green and red in order

32 bits/pixel:	each pixel is represented by a 4-byte integer; if the compression type is set to 0 then the three low order bytes represent the 8-bit values for blue, green and red, in order (the high order byte is not used); if the compression type is 3, then the 3 4-byte masks given above specify the bits to be used for red, blue and green.

For example, the format for 16-bit pixel values is:

```
Bit 15 | 14  13  12  11  10 |  9   8   7   6   5 |  4   3   2   1   0
       |      Red            |       Blue         |       Green      |
```

Run Length Encoding. The BMP format supports simple run-length encoding (RLE). *RLE-8* is for 8-bit pixel values. It uses a pair of bytes for each pixel value, where the first byte gives the count of the number of pixels that have the pixel value, and the second byte gives the pixel value.

Count	Pixel Value
First Byte	Second Byte

For example, the encoded 2-byte pair 08h 00h expands to 8 bytes of 00h, i.e., to

00h 00h 00h 00h 00h 00h 00h 00h

The 2-byte pair of zeros is used as an escape code: 00h 00h, which means to jump to a new row in the image. Another special code is: 00h 01h, which means the end of the image. A third special code is: 00h 02h, which means changes the position in the image, where the next two bytes give the respective number of columns and rows to advance. This allows a large number of zeros to be skipped.

As another example, the encoded sequence 04h 15h 00h 00h 02h 11h 02h 03h 00h 01h decodes to the sequence

15h 15h 15h 15h
11h 11h 03h 03h
End of Image

RLE-4 is similar but uses 4 bits per pixel and is interpreted slightly in a special situation for color coding. The encoded pair

05h 56h

is decoded to

5 6 5 6 5

The count is 5, so the following byte is actually two 4-bit values, which are repeated for 5 pixel values.

9.8 JPEG Image Files

9.9 Exercises

9.10 References

Randy Crane, *A Simplified Approach to Image Processing*, Prentice-Hall, Upper Saddle River, NJ 1997.

Marv Luse, *Bitmapped Graphics Programming in C++*, Addison-Wesley, Reading, MA 1993.

Roger T. Stevens, *The C Graphics Handbook*, Academic Press, San Diego, CA 1992.

Control-Zed, *Bitmapped Graphics*, WROX Press, Ltd., Birmingham, UK 1994.

Gregory Baxes, *Digital Image Processing*, Wiley, NY, 1994.

John Miano, *Compressed Image File Formats*, ACM Press, SIGGRAPH Series, New York (Addison-Wesley), 1999.