If you have a scan of the 10/89 DDJ cover, please mail it to me!

LZW Data Compression

by Mark Nelson

Dr. Dobb's Journal October, 1989

This page contains my original text and figures for the article that appeared in the October 1989 DDJ. I haven't broken it up into pages, so loading the entire thing might take some time.

LZW Data Compression

by Mark Nelson

Any programmer working on mini or microcomputers in this day and age should have at least some exposure to the concept of data compression. In MS-DOS world, programs like ARC, by System Enhancement Associates, and PKZIP, by Pkware are ubiquitous. ARC has also been ported to quite a few other machines, running UNIX, CP/M, and so on. CP/M users have long had SQ and USQ to squeeze and expand programs. Unix users have the COMPRESS and COMPACT utilities. Yet the data compression techniques used in these programs typically only show up in two places: file transfers over phone lines, and archival storage.

Data compression has an undeserved reputation for being difficult to master, hard to implement, and tough to maintain. In fact, the techniques used in the previously mentioned programs are relatively simple, and can be implemented with standard utilities taking only a few lines of code. This article discusses a good all-purpose data compression technique: Lempel-Ziv-Welch, or LZW compression. The routines shown here belong in any programmer's toolbox. For example, a program that has a few dozen help screens could easily chop 50K bytes off by compressing the screens. Or 500K bytes of software could be distributed to end users on a single 360K byte floppy disk. Highly redundant database files can be compressed down to 10% of their original size. Once the tools are available, the applications for compression will show up on a regular basis.

LZW Fundamentals

The original Lempel Ziv approach to data compression was first published in in1977. Terry Welch's refinements to the algorithm were published in 1984. The algorithm is surprisingly simple. In a nutshell, LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output instead of a string of characters.

The code that the LZW algorithm outputs can be of any arbitrary length, but it must have more bits in it than a single character. The first 256 codes (when using eight bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds. The

sample program runs as shown with 12 bit codes. This means codes 0-255 refer to individual bytes, while codes 256-4095 refer to substrings.

Compression

The LZW compression algorithm in its simplest form is shown in Figure 1. A quick examination of the algorithm shows that LZW is always trying to output codes for strings that are already known. And each time a new code is output, a new string is added to the string table.

Routine LZW_COMPRESS

```
STRING = get input character
WHILE there are still input characters DO
    CHARACTER = get input character
    IF STRING+CHARACTER is in the string table then
        STRING = STRING+character
    ELSE
        output the code for STRING
        add STRING+CHARACTER to the string table
        STRING = CHARACTER
    END of IF
END of WHILE
output the code for STRING
```

The Compression Algorithm

Figure 1

A sample string used to demonstrate the algorithm is shown in Figure 2. The input string is a short list of English words separated by the '/' character. Stepping through the start of the algorithm for this string, you can see that the first pass through the loop, a check is performed to see if the string "/W" is in the table. Since it isn't, the code for '/' is output, and the string "/W" is added to the table. Since we have 256 characters already defined for codes 0-255, the first string definition can be assigned to code 256. After the third letter, 'E', has been read in, the second string code, "WE" is added to the table, and the code for letter 'W' is output. This continues until in the second word, the characters '/' and 'W' are read in, matching string number 256. In this case, the code 256 is output, and a three character string is added to the string table. The process continues until the string is exhausted and all of the codes have been output.

Input String = /WED/WE/WEE/WEB/WET			
Character Input	**C**ode Output	**N**ew code value	**New String**
/W	/	256	/W
E	W	257	WE
D	E	258	ED
/	D	259	D/
WE	256	260	/WE
/	E	261	E/
WEE	260	262	/WEE
/W	261	263	E/W
EB	257	264	WEB
/	B	265	B/
WET	260	266	/WET
EOF	T		

The Compression Process

Figure 2

The sample output for the string is shown in Figure 2 along with the resulting string table. As can be seen, the string table fills up rapidly, since a new string is added to the table each time a code is output. In this highly redundant input, 5 code substitutions were output, along with 7 characters. If we were using 9 bit codes for output, the 19 character input string would be reduced to a 13.5 byte output string. Of course, this example was carefully chosen to demonstrate code substitution. In real world examples, compression usually doesn't begin until a sizable table has been built, usually after at least one hundred or so bytes have been read in.

Decompression

The companion algorithm for compression is the decompression algorithm. It needs to be able to take the stream of codes output from the compression algorithm, and use them to exactly recreate the input stream. One reason for the efficiency of the LZW algorithm is that it does not need to pass the string table to the decompression code. The table can be built exactly as it was during compression, using the input stream as data. This is possible because the compression algorithm always outputs the STRING and CHARACTER components of a code before it uses it in the output stream. This means that the compressed data is not burdened with carrying a large string translation table.

Routine LZW_DECOMPRESS

```
Read OLD_CODE
output OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    STRING = get translation of NEW_CODE
    output STRING
    CHARACTER = first character in STRING
```

```
add OLD_CODE + CHARACTER to the translation table
OLD_CODE = NEW_CODE
END of WHILE
```

The Decompression Algorithm

Figure 3

The algorithm is shown in Figure 3. Just like the compression algorithm, it adds a new string to the string table each time it reads in a new code. All it needs to do in addition to that is translate each incoming code into a string and send it to the output.

Figure 4 shows the output of the algorithm given the input created by the compression earlier in the article. The important thing to note is that the string table ends up looking exactly like the table built up during compression. The output string is identical to the input string from the compression algorithm. Note that the first 256 codes are already defined to translate to single character strings, just like in the compression code.

Input Codes: / W E D 256 E 260 261 257 B 260 T				
Input/ NEW_CODE	OLD_CODE	STRING/ Output	CHARACTER	New table entry
/	/	/		
W	/	W	W	256 = W
E	W	E	E	257 = WE
D	E	D	D	258 = ED
256	D	/W	/	259 = D
E	256	E	E	260 = /WE
260	E	/WE	/	261 = E/
261	260	E/	E	262 = /WEE
257	261	WE	W	263 = E/W
B	257	B	B	264 = WEB
260	B	/WE	/	265 = B
T	260	T	T	266 = /WET

The Decompression Process

Figure 4

The Catch

Unfortunately, the nice simple decompression algorithm shown in Figure 4 is just a little *too* simple. There is a single exception case in the LZWcompression algorithm that causes some trouble to the decompression side. If there is a string consisting of a (STRING,CHARACTER) pair already defined in the table, and the input stream then sees a sequence of STRING, CHARACTER, STRING,

CHARACTER, STRING, the compression algorithm will output a code before the decompressor gets a chance to define it.

A simple example will illustrate the point. Imagine the the string JOEYN is defined in the table as code 300. Later on, the sequence JOEYNJOEYNJOEY occurs in the table. The compression output will look like that shown in Figure 5.

Input String: ...JOEYNJOEYNJOEY		
Character Input	New Code/String	Code Output
JOEYN	300 = JOEYN	288 (JOEY)
A	301 = NA	N
.	.	.
.	.	.
.	.	.
JOEYNJ	400 = JOEYNJ	300 (JOEYN)
JOEYNJO	401 = JOEYNJO	400 (???)

A problem

Figure 5

When the decompression algorithm sees this input stream, it first decodes the code 300, and outputs the JOEYN string. After doing the output, it will add the definition for code 399 to the table, whatever that may be. It then reads the next input code, 400, and finds that it is not in the table. This is a problem, what do we do?

Fortunately, this is the only case where the decompression algorithm will encounter an undefined code. Since it is in fact the only case, we can add an exception handler to the algorithm. The modified algorithm just looks for the special case of an undefined code, and handles it. In the example in Figure 5, the decompression routine sees a code of 400, which is undefined. Since it is undefined, it translates the value of OLD_CODE, which is code 300. It then adds the CHARACTER value, which is 'J', to the string. This results in the correct translation of code 400 to string "JOEYNJ".

Routine LZW_DECOMPRESS

```
Read OLD_CODE
output OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    IF NEW_CODE is not in the translation table THEN
        STRING = get translation of OLD_CODE
        STRING = STRING+CHARACTER
    ELSE
        STRING = get translation of NEW_CODE
    END of IF
    output STRING
    CHARACTER = first character in STRING
    add OLD_CODE + CHARACTER to the translation table
    OLD_CODE = NEW_CODE
END of WHILE
```

The Modified Decompression Algorithm

Figure 6

The Implementation Blues

The concepts used in the compression algorithm are very simple -- so simple that the whole algorithm can be expressed in only a dozen lines. Implementation of this algorithm is somewhat more complicated, mainly due to management of the string table.

In the code accompanying this article, I have used code sizes of 12, 13, and 14 bits. In a 12 bit code program, there are potentially 4096 strings in the string table. Each and every time a new character is read in, the string table has to be searched for a match. If a match is not found, then a new string has to be added to the table. This causes two problems. First, the string table can get very large very fast. If string lengths average even as low as three or four characters each, the overhead of storing a variable length string and its code could easily reach seven or eight bytes per code.

In addition, the amount of storage needed is indeterminate, as it depends on the total length of all the strings.

The second problem involves searching for strings. Each time a new character is read in, the algorithm has to search for the new string formed by STRING+CHARACTER. This means keeping a sorted list of strings. Searching for each string will take on the order of log 2 string compares. Using 12 bit words means potentially doing 12 string compares for each code. The computational overhead caused by this would be prohibitive.

The first problem can be solved by storing the strings as code/character combinations. Since every string is actually a combination of an existing code and an appended character, we can store each string as single code plus a character. For example, in the compression example shown, the string "/WEE" is actually stored as code 260 with appended character E. This takes only three bytes of storage instead of 5 (counting the string terminator). By backtracking, we find that code 260 is stored as code 256 plus an appended character E. Finally, code 256 is stored as a '/' character plus a 'W'.

Doing the string comparisons is a little more difficult. Our new method of storage reduces the amount of time for a string comparison, but it doesn't cut into the the number of comparisons needed to find a match. This problem is solved by storing strings using a hashing algorithm. What this means is that we don't store code 256 in location 256 of an array. We store it in a location in the array based on an address formed by the string itself. When we are trying to locate a given string, we can use the test string to generate a hashed address, and with luck can find the target string in one search.

Since the code for a given string is no longer known merely by its position in the array, we need to store the code for a given string along with the string data. In the attached program, there are three array elements for each string. They are code_value[i], prefix_code[i], and append_character[i].

When we want to add a new code to the table, we use the hashing function in routine "find_match" to generate the correct i. Find_match generates an address, then checks to see if the location is already in use by a different string. If it is, it performs a secondary probe until an open location is found.

The hashing function in use in this program is a straightforward "xor" type hash function. The prefix code and appended character are combined to form an array address. If the contents of the prefix code and character in the array are a match, the correct address is returned. If that element in the array is in use, a fixed offset probe is used to search new locations. This continues until either an empty slot is found, or a match is found. By using a table about 25% larger than needed, the average number of searches in the table usually stays below 3. Performance can be improved by increasing the size of the table. Note that in order for the secondary probe to always work, the size of the table needs to be a prime number. This is because the probe can be any integer between 0 and the table size. If the probe and the table size were not mutually prime, a search for an open slot could fail even if there were still open slots available.

Implementing the decompression algorithm has its own set of problems. One of the problems from the compression code goes away. When we are compressing, we need to search the table for a given string. During decompression, we are looking for a particular code. This means that we can store the prefix codes and appended characters in the table indexed by their string code. This eliminates the need for a hashing function, and frees up the array used to store code values.

Unfortunately, the method we are using of storing string values causes the strings to be decoded in reverse order. This means that all the characters for a given string have to be decoded into a stack buffer, then output in reverse order. In the program here this is done in the "decode_string" function. Once this code is written, the rest of the algorithm turns into code very easily.

One problem encountered when reading in data streams is determining when you have reached the end of the input data stream. In this particular implementation, I have reserved the last defined code, MAX_VALUE, as a special end of data indicator. While this may not be necessary when reading in data files, it is very helpful when reading compressed buffers out of memory. The expense of losing one defined code is minimal in comparison to the convenience.

Results

It is somewhat difficult to characterize the results of any data compression technique. The level of compression achieved varies quite a bit depending on several factors. LZW compression excels when confronted with data streams that have any type of repeated strings. Because of this, it does extremely well when compressing English text. Compression levels of 50% or better should be expected. Likewise, compressing saved screens and displays will generally show very good results.

Trying to compress binary data files is a little more risky. Depending on the data, compression may or may not yield good results. In some cases, data files will compress even more than text. A little bit of experimentation will usually give you a feel for whether your data will compress well or not.

Your Implementation

The code accompanying this article works. However, it was written with the goal of being illuminating, not efficient. Some parts of the code are relatively inefficient. For example, the variable length input and output routines are short and easy to understand, but require a lot of overhead. An LZW program using fixed length 12 bit codes could experience real improvements in speed just by recoding these two routines.

One problem with the code listed here is that it does not adapt well to compressing files of differing

sizes. Using 14 or 15 bit codes gives better compression ratios on large files, (because they have a larger string table to work with), but actually has poorer performance on small files. Programs like ARC get around this problem by using variable length codes. For example, while the value of "next_code" is between 256 and 511, ARC inputs and outputs 9 bit codes. When the value of next_code increases to the point where 10 bit codes are needed, both the compression and decompression routines adjust the code size. This means that the 12 bit and 15 bit versions of the program will do equally well on small files.

Another problem on long files is that frequently the compression ratio begins to degrade as more of the file is read in. The reason for this is simple. Since the string table is of finite size, after a certain number of strings have been defined, no more can be added. But the string table is only good for the portion of the file that was read in while it was built. Later sections of the file may have different characteristics, and really need a different string table.

The conventional way to solve this problem is to monitor the compression ratio. After the string table is full, the compressor watches to see if the compression ratio degrades. After a certain amount of degradation, the entire table is flushed, and gets rebuilt from scratch. The expansion code is flagged when this happens by seeing a special code from the compression routine.

An alternative method would be to keep track of how frequently strings are used, and to periodically flush values that are rarely used. An adaptive technique like this may be too difficult to implement in a reasonably sized program.

One final technique for compressing the data is to take the LZW codes and run them through an adaptive Huffman coding filter. This will generally exploit a few more percentage points of compression, but at the cost of considerable more complexity in the code, as well as quite a bit more run time.

Portability

The code shown was written and tested on MS-DOS machines, and has successfully compiled and executed with several C compilers. It should be portable to any machine that supports 16 integers and 32 bit longs in C. MS-DOS C compilers typically have trouble dealing with arrays larger than 64 Kbytes, preventing an easy implementation of 15 or 16 bit codes in this program. On machines using different processors, such as a VAX, these restrictions are lifted, and using larger code sizes becomes much simpler.

In addition, porting this code to assembly language should be fairly simple on any machine that supports 16 and 32 bit math. Significant performance improvements could be seen this way. Implementations in other high level languages should be straightforward.

Bibliography:

1. Terry Welch, "A Technique for High-Performance Data Compression", Computer, June 1984
2. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, May 1977
3. Rudy Rucker, "Mind Tools", Houghton Mifflin Company, 1987

LZW.C

```
/*************************************************************************
**
** Copyright (c) 1989 Mark R. Nelson
**
** LZW data compression/expansion demonstration program.
**
** April 13, 1989
**
*************************************************************************/
#include <stdio.h>
#define BITS 12                   /* Setting the number of bits to 12, 13*/
#define HASHING_SHIFT BITS-8      /* or 14 affects several constants.    */
#define MAX_VALUE (1 << BITS) - 1 /* Note that MS-DOS machines need to   */
#define MAX_CODE MAX_VALUE - 1    /* compile their code in large model if*/
                                  /* 14 bits are selected.               */
#if BITS == 14
  #define TABLE_SIZE 18041        /* The string table size needs to be a */
#endif                            /* prime number that is somewhat larger*/
#if BITS == 13                    /* than 2**BITS.                       */
  #define TABLE_SIZE 9029
#endif
#if BITS <= 12
  #define TABLE_SIZE 5021
#endif

void *malloc();

int *code_value;                  /* This is the code value array        */
unsigned int *prefix_code;        /* This array holds the prefix codes   */
unsigned char *append_character;  /* This array holds the appended chars */
unsigned char decode_stack[4000]; /* This array holds the decoded string */

/*************************************************************************
**
** This program gets a file name from the command line.  It compresses the
** file, placing its output in a file named test.lzw.  It then expands
** test.lzw into test.out.  Test.out should then be an exact duplicate of
** the input file.
**
*************************************************************************/

main(int argc, char *argv[])
{
FILE *input_file;
FILE *output_file;
FILE *lzw_file;
char input_file_name[81];

/*
**  The three buffers are needed for the compression phase.
*/
  code_value=malloc(TABLE_SIZE*sizeof(unsigned int));
  prefix_code=malloc(TABLE_SIZE*sizeof(unsigned int));
  append_character=malloc(TABLE_SIZE*sizeof(unsigned char));
  if (code_value==NULL || prefix_code==NULL || append_character==NULL)
  {
    printf("Fatal error allocating table space!\n");
    exit();
  }
/*
** Get the file name, open it up, and open up the lzw output file.
*/
  if (argc>1)
```

```
    strcpy(input_file_name,argv[1]);
  else
  {
    printf("Input file name? ");
    scanf("%s",input_file_name);
  }
  input_file=fopen(input_file_name,"rb");
  lzw_file=fopen("test.lzw","wb");
  if (input_file==NULL || lzw_file==NULL)
  {
    printf("Fatal error opening files.\n");
    exit();
  };
/*
** Compress the file.
*/
  compress(input_file,lzw_file);
  fclose(input_file);
  fclose(lzw_file);
  free(code_value);
/*
** Now open the files for the expansion.
*/
  lzw_file=fopen("test.lzw","rb");
  output_file=fopen("test.out","wb");
  if (lzw_file==NULL || output_file==NULL)
  {
    printf("Fatal error opening files.\n");
    exit();
  };
/*
** Expand the file.
*/
  expand(lzw_file,output_file);
  fclose(lzw_file);
  fclose(output_file);

  free(prefix_code);
  free(append_character);
}

/*
** This is the compression routine.  The code should be a fairly close
** match to the algorithm accompanying the article.
**
*/

compress(FILE *input,FILE *output)
{
unsigned int next_code;
unsigned int character;
unsigned int string_code;
unsigned int index;
int i;

  next_code=256;              /* Next code is the next available string code*/
  for (i=0;i<TABLE_SIZE;i++)  /* Clear out the string table before starting */
    code_value[i]=-1;

  i=0;
  printf("Compressing...\n");
  string_code=getc(input);    /* Get the first code                         */
/*
** This is the main loop where it all happens.  This loop runs util all of
```

```
** the input has been exhausted.  Note that it stops adding codes to the
** table after all of the possible codes have been defined.
*/
  while ((character=getc(input)) != (unsigned)EOF)
  {
    if (++i==1000)                          /* Print a * every 1000    */
    {                                       /* input characters.  This */
      i=0;                                  /* is just a pacifier.     */
      printf("*");
    }
    index=find_match(string_code,character);/* See if the string is in */
    if (code_value[index] != -1)            /* the table.  If it is,   */
      string_code=code_value[index];        /* get the code value.  If */
    else                                    /* the string is not in the*/
    {                                       /* table, try to add it.   */
      if (next_code <= MAX_CODE)
      {
        code_value[index]=next_code++;
        prefix_code[index]=string_code;
        append_character[index]=character;
      }
      output_code(output,string_code);  /* When a string is found  */
      string_code=character;            /* that is not in the table*/
    }                                   /* I output the last string*/
  }                                     /* after adding the new one*/
/*
** End of the main loop.
*/
  output_code(output,string_code); /* Output the last code               */
  output_code(output,MAX_VALUE);   /* Output the end of buffer code      */
  output_code(output,0);           /* This code flushes the output buffer*/
  printf("\n");
}

/*
** This is the hashing routine.  It tries to find a match for the prefix+char
** string in the string table.  If it finds it, the index is returned.  If
** the string is not found, the first available index in the string table is
** returned instead.
*/

find_match(int hash_prefix,unsigned int hash_character)
{
int index;
int offset;

  index = (hash_character << HASHING_SHIFT) ^ hash_prefix;
  if (index == 0)
    offset = 1;
  else
    offset = TABLE_SIZE - index;
  while (1)
  {
    if (code_value[index] == -1)
      return(index);
    if (prefix_code[index] == hash_prefix &&
        append_character[index] == hash_character)
      return(index);
    index -= offset;
    if (index < 0)
      index += TABLE_SIZE;
  }
}
```

```
/*
**  This is the expansion routine.  It takes an LZW format file, and expands
**  it to an output file.  The code here should be a fairly close match to
**  the algorithm in the accompanying article.
*/

expand(FILE *input,FILE *output)
{
unsigned int next_code;
unsigned int new_code;
unsigned int old_code;
int character;
int counter;
unsigned char *string;
char *decode_string(unsigned char *buffer,unsigned int code);

  next_code=256;           /* This is the next available code to define */
  counter=0;               /* Counter is used as a pacifier.            */
  printf("Expanding...\n");

  old_code=input_code(input);  /* Read in the first code, initialize the */
  character=old_code;          /* character variable, and send the first */
  putc(old_code,output);       /* code to the output file                */
/*
**  This is the main expansion loop.  It reads in characters from the LZW file
**  until it sees the special code used to inidicate the end of the data.
*/
  while ((new_code=input_code(input)) != (MAX_VALUE))
  {
    if (++counter==1000)   /* This section of code prints out     */
    {                      /* an asterisk every 1000 characters   */
      counter=0;           /* It is just a pacifier.              */
      printf("*");
    }
/*
** This code checks for the special STRING+CHARACTER+STRING+CHARACTER+STRING
** case which generates an undefined code.  It handles it by decoding
** the last code, and adding a single character to the end of the decode string.
*/
    if (new_code>=next_code)
    {
      *decode_stack=character;
      string=decode_string(decode_stack+1,old_code);
    }
/*
** Otherwise we do a straight decode of the new code.
*/
    else
      string=decode_string(decode_stack,new_code);
/*
** Now we output the decoded string in reverse order.
*/
    character=*string;
    while (string >= decode_stack)
      putc(*string--,output);
/*
** Finally, if possible, add a new code to the string table.
*/
    if (next_code <= MAX_CODE)
    {
      prefix_code[next_code]=old_code;
      append_character[next_code]=character;
      next_code++;
    }
```

```
    old_code=new_code;
  }
  printf("\n");
}

/*
** This routine simply decodes a string from the string table, storing
** it in a buffer.  The buffer can then be output in reverse order by
** the expansion program.
*/

char *decode_string(unsigned char *buffer,unsigned int code)
{
int i;

  i=0;
  while (code > 255)
  {
    *buffer++ = append_character[code];
    code=prefix_code[code];
    if (i++>=4094)
    {
      printf("Fatal error during code expansion.\n");
      exit();
    }
  }
  *buffer=code;
  return(buffer);
}

/*
** The following two routines are used to output variable length
** codes.  They are written strictly for clarity, and are not
** particularyl efficient.
*/

input_code(FILE *input)
{
unsigned int return_value;
static int input_bit_count=0;
static unsigned long input_bit_buffer=0L;

  while (input_bit_count <= 24)
  {
    input_bit_buffer |=
        (unsigned long) getc(input) << (24-input_bit_count);
    input_bit_count += 8;
  }
  return_value=input_bit_buffer >> (32-BITS);
  input_bit_buffer <<= BITS;
  input_bit_count -= BITS;
  return(return_value);
}

output_code(FILE *output,unsigned int code)
{
static int output_bit_count=0;
static unsigned long output_bit_buffer=0L;

  output_bit_buffer |= (unsigned long) code << (32-BITS-output_bit_count);
  output_bit_count += BITS;
  while (output_bit_count >= 8)
  {
    putc(output_bit_buffer >> 24,output);
```

```
    output_bit_buffer <<= 8;
    output_bit_count -= 8;
  }
}
```

Return to Mark Nelson's Home Page