LZW - Data Compression

Student CRISTIANA CREȚU
Student ALINA CRISTIAN
Student ANDREEA CUCULEANU
Student ALEXANDRA FESCI
Faculty of Economic Cybernetics, Statistics and Informatics,
Academy of Economic Studies

1.History

The name of this algorithm is Lempel Ziv Welch data compression algorithm, and has been taken from its three inventors. A. Lempel and J. Ziv published the first approach to this algorithm in 1977, in the May number of the IEEE Transactions on Information Theory magazine within an article called "A Universal Algorithm for Sequential Data Compression". In this algorithm were being used fix phrases dictionaries that were sliding on an already read text. The phrases were built starting from a symbol, and attaching a new symbol to an already existing phrase if it was found in the dictionary.

In 1978 a new article was published in the same magazine. These two articles were presenting the problem and the solution from a too technical and abstracted point of view.

Until 1984 no practical version of the algorithm existed on the market. In 1984 Terry Welch publishes in June an article called "A technique for High-Performance Data Compression" in the Computer magazine, which came with some refinements to the algorithm.

LZSS simplified the LZ77 algorithm by eliminating the restriction referring to the fact that every symbol was formed from a code and a character. LZW does the same for the LZ78 algorithm. Actually the LZW compressor never sends characters as such, only codes.

Because of this, the main refinement is considered to be the change of the start dictionary, which contains, from the very beginning, the standard character set (in the first 256 locations). Thus, all one character strings can immediately be coded, as they are already in the dictionary, even if they appeared for the first time in the data stream. The remaining codes are assigned to streams as the algorithm proceeds. If a dictionary uses a 12-bit code, the locations 0-255 refer to individual codes, while codes 256-4095 refer to sub-strings.

2.Fundamentals

The main idea of this algorithm is very simple. The LZW algorithm tries to always send codes for already known strings. Each time a new code is sent, a new string is attached to the dictionary. The algorithm doesn't do any analyses on the incoming text. It simply reads it and sees if the string has been read before. If this is the first time it reads that string it adds it to the dictionary. Text is being compressed when a code is being written instead of a string.

The start dictionary contains, as has already been said the standard character set (the ASCII codes). To this dictionary we attach two control codes 256 and 257. The 256 code is being used as an end of file character (the last character that appears in the text), and the 257 character is being used as an end of dictionary character (the last character in an dictionary used only when a dictionary is full and we need to use a new dictionary).

The advantage of this algorithm is that the LZW decompressor doesn't need to use the dictionary created when compressing the text. Instead it creates a new table starting from the compressed file and using the same criteria of adding to a dictionary as for compression.

3.Compression

The compression algorithm consists of three fazes.

The first step consists in the initialization of the start dictionary. As shown above, the standard character set is being put in the dictionary. It is also initialized the variable next_code with the next value in the dictionary. This value will be able to grow till it riches 4095 if the program uses a 12-bit dictionary.

The next step is the main loop. In this step a new character is read from the input stream. The new form string is looked for in the dictionary. If the string is found its code is transmitted as output. If the string is not found, a new code is generated and added to the dictionary. This loop ends when there are no more characters left in the input stream.

The final step is where the end of file code is generated and the compression process ends.

Here is how this compression procedure can be written. It has been published by Mark Nelson (see the references).

```
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 until 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");
```

To demonstrate how this algorithm works we will use a sample shown in Figure 1. This is a famous quote from Shakespeare's Hamlet: "That he is mad 'tis true 'tis true 'tis pity and pity 'tis 'tis true".

We can see from the very start that this is a highly redundant string. In it the word ‘tis can be found 5 times. The output stream can be seen in the same figure.

You can see that 29 characters and 16 codes form the output, while the input stream was formed by 66 characters. Therefore, if we were using a 9-bit dictionary, the compressed file would have had 45 bytes, while the original file would have had 66 bytes. However this was a carefully selected stream, highly redundant. Still we can acknowledge the fact that we obtained a 45% compression rate. Still the optimal alternative for this algorithm is obtained by using a 12-bit code dictionary. In this case figures would change.

We must also bear in mind that while this text as carefully chosen so that the code substitution can easily be seen in normal texts word substitution starts after a sizable table has been built. This usually happens after at least one hundred bytes have been read.

Figure1 Example stream: “That he is mad ‘tis true ‘tis true ‘tis pity and pity ‘tis ‘tis true”

Read code	Dictionary code	Last code	Transmitted code	Read code	Dictionary code	Last code	Transmitted code
‘t’	none	‘t	None	‘’’	None	‘ ‘’	None
‘h’	‘th’=258	‘h’	‘t’	‘t’	‘ ‘t’=285	‘t’	272
‘a’	‘ha’=259	‘a’	‘h’	‘I’	None	‘ti’	None
‘t’	‘at’=260	‘t’	‘a’	‘s’	‘tis’=286	‘s’	274
‘ ‘	‘t ‘=261	‘ ‘	‘t’	‘ ‘	None	‘s ‘	None
‘h’	‘ h’=262	‘h’	‘ ‘	‘p’	‘s p’=287	‘p’	267
‘e’	‘he’=263	‘e’	‘h’	‘I’	‘pi’=288	‘I’	‘p’
‘ ‘	‘e ’=264	‘ ‘	‘e’	‘t’	‘it’=289	‘t’	‘I’
‘I’	‘ I’=265	‘I’	‘ ‘	‘y’	‘ty’=290	‘y’	‘t’
‘s’	‘is’=266	‘s’	‘I’	‘ ‘	‘y ‘=291	‘ ‘	‘y’
‘ ‘	‘s ‘=267	‘ ‘	‘s’	‘a’	‘ a’=292	‘a’	‘ ‘
‘m’	‘ m’=268	‘m’	‘ ’	‘n’	‘an’=293	‘n’	‘a’
‘a’	‘ma’=269	‘a’	‘m‘	‘d’	‘nd’=294	‘d’	‘n’
‘d’	‘ad’=270	‘d’	‘a’	‘ ‘	None	‘d ‘	None
‘ ‘	‘d ‘=271	‘ ‘	‘d’	‘p’	‘dp’=295	‘p’	270
‘’’	‘ ‘’=272	‘’’	‘ ‘	‘I’	None	‘pi’	None
‘t’	‘’t’=273	‘t’	‘’’	‘t’	‘pit’=296	‘t’	288
‘I’	‘ti’=274	‘I’	‘t’	‘y’	None	‘ty’	None
‘s’	None	‘is’	None	‘ ‘	‘ty ‘=297	‘ ‘	290
‘ ‘	‘Is ‘=275	‘ ‘	266	‘’’	None	‘ ‘’	None
‘t’	‘ t’=276	‘t’	‘ ‘	‘t’	None	‘ ‘t’	None
‘r’	‘tr’=277	‘r’	‘t’	‘I’	‘ ‘ti’=298	‘I’	285
‘u’	‘ru’=278	‘u’	‘r’	‘s’	None	‘is’	None
‘e’	‘ue’=279	‘e’	‘u’	‘ ‘	None	‘is ‘	None
‘ ‘	None	‘e ‘	None	‘’’	‘is ‘=299	‘’’	275
‘’’	‘e ‘’=280	‘’’	264	‘t’	None	‘’t’	None
‘t’	None	‘’t’	None	‘I’	None	‘’ti’	None
‘I’	‘’ti’=281	‘I’	273	‘s’	‘’tis’=300	‘s’	281
‘s’	None	‘is’	None	‘ ‘	None	‘s ‘	None
‘ ‘	None	‘is ‘	None	‘t’	‘s t’=301	‘t’	267
‘t’	‘is t’=282	‘t’	275	‘r’	None	‘tr’	None
‘r’	None	‘tr’	None	‘u’	None	‘tru’	None
‘u’	‘tru’=283	‘u’	277	‘e’	‘true’= none	‘e’	283
‘e’	None	‘ue’	None	End of file	None	None	‘e’
‘ ‘	‘ue ‘=284	‘ ‘	279				

4.Decompression

The decompression sequence also has three phases.

The first step consists in the initialization of the variable old_code, which will be used to administrate the codes attached to the dictionary. The first character is read from the previously compressed file and sent to the decompressed file.

The second step is the main loop, which ends when the end of file character is read. The codes are read and looked for in the dictionary. When the stream is found is sent to the output file. A new string is formed from the old code and the first character from the string that has just been read.

The final step is when the end of file code is sent to the output file.

To see how this algorithm really works, we'll try to decompress the output obtained by compressing the text: "That he is mad 'tis true 'tis true 'tis pity and pity 'tis 'tis true". This example can be seen in Figure 2. We can clearly see now that there is no need to keep the dictionary formed at compression. The decompression algorithm forms its own dictionary, which is the same as the compressor defined while compressing the text. Therefore the decoding of the text encounters no problems.

Figure 2:decompressing the text code: "That he is mad 't266 tru264/273/275/277/279/272/274/267pity an271/288/290/285/275/281/267/283e"(the "/" character has been used to divide the distinct codes; it doesn't appear in the compressed file.)

Read code	Dictionary code	Last code	Transmitted code	Read code	Dictionary code	Last code	Trans-mitted
't'	None	None	't'	273	'e ''=280	'e '	''t'
'h'	'th'=258	't'	'h'	275	''ti'=281	''t'	'is '
'a'	'ha'=259	'h'	'a'	277	'is t'=282	'is '	'tr'
't'	'at'=260	'a'	't'	279	'tru'=283	'tr'	'ue'
' '	't '=261	't'	' '	272	'ue '=284	'ue'	' ''
'h'	' h'=262	' '	'h'	274	' 't'=285	' ''	'ti'
'e'	'he'=263	'h'	'e'	267	'tis'=286	'ti'	's '
' '	'e '=264	'e'	' '	'p'	's p'=287	's '	'p'
'I'	' I'=265	' '	'I'	'I'	'pi'=288	'p'	'I'
's'	'is'=266	'I'	's'	't'	'it'=289	'I'	't'
' '	's '=267	's'	' '	'y'	'ty'=290	't'	'y'
'm'	' m'=268	' '	'm'	' '	'y '=291	'y'	' '
'a'	'ma'=269	'm'	'a'	'a'	' a'=292	' '	'a'
'd'	'ad'=270	'a'	'd'	'n'	'an'=293	'a'	'n'
' '	'd '=271	'd'	' '	271	'nd'=294	'n'	'd '
'''	' ''=272	' '	'''	288	'd p'=295	'd '	'pi'
't'	''t'=273	'''	't'	290	'pit'=296	'pi'	'ty'
266	'ti'=274	't'	'is'	285	'ty '=297	'ty'	' 't'
' '	'Is '=275	'is'	' '	275	' 'ti'=298	' 't'	'is '
't'	' t'=276	' '	't'	281	'is ''=299	'is '	''ti'
'r'	'tr'=277	't'	'r'	267	''tis'=300	''ti'	's '
'u'	'ru'=278	'r'	'u'	283	's t'=301	's '	'tru'
264	'ue'=279	'u'	'e '	'e'	'true'=302	'tru'	'e'

The decompression algorithm seems a little too simple as it has been described above. Actually there is an exception to this algorithm. If there is a string consisting of a (string, character) pair, and the decoder finds a combination like this: (string, character, string, character, string), the compression algorithm will output a code before the decompression algorithm gets a chance to define it. To better understand what is happening lets suppose that at a certain point the compression algorithm defines the string "BEAUTY" with the code 400. Later on it finds the string "BEAUTYBEAUTYBEAUTY" which it defines with the code 500(see the table below).

Read character	Dictionary code	Output code
BEAUTY	BEAUTY=400	388(BEAUT)
BEAUTYB	BEAUTYB=500	400(BEAUTY)
BEAUTYBE	BEAUTYBE=501	500(???)

When the decompressor sees this input it first decodes the code 400, and outputs the BEAUTY string. After doing the output, it will add the definition 499 to the dictionary, which ever that might be, and tries to output the string for the code 500. But it hasn't defined yet this code. Here we have a problem. What should the decompressor output?

Since this is the only time the decompression algorithm will encounter a problem we can add an exception handler to the algorithm. This can look for undefined codes and handle the exception by translating the value of old_code and then adds the character value. In the sample old_code is 400, its string is BEAUTY and by adding the character B we obtain the correct string for the code 500, which is BEAUTYB.

The decompression procedure can be written as follows. It has been published by Mark Nelson (see the references). This procedure also includes the exception handler.

```
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 indicate 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");
```

5.Results

The level of compression achieved varies depending on several factors. LZW excels when applied to data streams that have any type of repeated streams. Therefore it has the best results when compressing English texts. In this case a compression level of 50% or more should be expected. The same results apply to saved screens and displays, which will generally show good results. From Figure 4 we can see that the compression level for an avi file is 75%.

Compressing the binary files is a little bit risky. There are data sets which can give a level of compression higher than texts, while there can be data set that offer a compression level equal to 0. From Figure 4 we can see that for jpg, z80, and gif files we get compression level below 0.

File Name	File dimension	The length of compressed file	Compression level
Test01.doc	37 KB	18 KB	51%
Test02.jpg	26 KB	35 KB	-34%
Test03.exe	46 KB	36 KB	22%
Test04.dat	3 KB	2 KB	33%
Test05.txt	3 KB	2 KB	33%
Test06.html	11 KB	6 KB	46%
Test07.ani	12 KB	3 KB	75%
Test08.obj	2.875 KB	2.686 KB	7%
Test09.gif	3 KB	4 KB	-33%
Test10.z80	45 KB	55 KB	-22%
Test11.pod	3 KB	1 KB	66%

6.References

[Ivan98] Ion Ivan , Daniel Vernis
Compresia de date
Editura Cison
Bucuresti 1998

[Nels92] Nelson, Mark
La compression de données. Texte. Images. Sons
Ed. Dunod,
Paris,1992

http://dogma.net/markn/articles/lzw/lzw.htm

http://www.csu.sfu.ca/cc/365/li/squeeze/lzw.html