IMPLEMENTATION OF LEMPEL-ZIV COMPRESSION
ALGORITHM USING MATLAB

PROJECT
at
ELECTRONICS & COMMUNICATION ENGINEERING DEPARTMENT,
NIT-DURGA PUR

Developed by: -
RAJESH KUMAR SINGH (05/EC E/16)
DEEPAK VERMA (05/EC E/50)
ASHUTOSH KUMAR GREWAL (05/EC E/52)
SANDEEP KO SHY (05/EC E/58)
ANSHU SHARMA (05/EC E/65)

Mentor: -
Mr. Aniruddha Chandra
Lecturer
ACKNOWLEDGEMENT

We are highly indebted to Mr. Aniruddha Chandra, Lecturer, Electronics & Communication Engineering, NIT Durgapur, who found us worthy enough to undertake this project. We owe to him as he endured us during his working hours and was always there for us whenever we got curious and wanted to enquire about something. He made us aware of many technical aspects which we could never even dream of.

What is the prospect of my field??... Had always been an issue which used to bewilder us. This project has aroused and nurtured in us the interest for this field. Here we could see the live, wide and much needed applications of our field.

We inculcated many essential feelings like dedication and precision to do the work. It was praiseworthy to see the desperation in him to take his organization to the peaks of success. Hard work and precision marked his excellence and motivated us to grasp his working abilities from him.

We would like to comprehend by saying that it was an awesome experience to learn from a highly qualified professional of our field.
# Table of Contents

- **LEMPEL-ZIV COMPRESSION ALGORITHM**  
  - Chapter 1: - Introduction  4  
  - Chapter 2: - History  5  
  - Chapter 3: - Sliding Window Compression (LZ77)  6  
    - Compression & Decompression  
    - Coding  
  - Chapter 4: - LZW Compression Algorithm  
    - Example 1: Encoding using LZW  11  
    - Example 2: Decoding using LZW  19  
  - Chapter 5: - Problems with LZ77  25  
  - Chapter 6: - Improvements  26  

- References  27


Chapter 1   Introduction

Data Compression seeks to reduce the number of bits used to store or transmit information. It encompasses a wide variety of software and hardware compression techniques. Data compression consists of taking a stream of symbols and transforming them into codes. For effective compression, the resultant stream of codes will be smaller than than the original symbol. For e.g, Huffman coding is a type of coding where the actual output of encoder is determined by a set of probabilities

Here the problem is that it uses an integral number of bits & also, one must have the prior information of probabilities. The problem of statistical model is solved by using adaptive dictionary. Lempel-Ziv-Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is designed to be fast to implement but is not usually optimal because it performs only limited analysis of the data. The first compression algorithm to be described was LZ77. The two different types of compression lossy & lossless differs in one respect: Lossy compression accepts a slight compression of data to achieve compression. Lossy compression is done on analog data stored digitally, with primary applications being graphics and sound files.
Chapter 2 History

Until 1980, most general compression schemes statistical modeling. But in 1977 and 1978, Jacob Ziv and Abraham Lempel described a pair of compression methods using an adaptive dictionary. These two algorithms sparked a flood of new techniques that used dictionary-based methods to achieve impressive new compression ratios.

The first compression algorithm described by Ziv and Lempel is commonly referred to as LZ77. It is relatively simple. The dictionary consists of all the strings in a window into the previously read input stream. A file-compression program, for example, could use a 4K-byte window as a dictionary.

While new groups of symbols are being read in, the algorithm looks for matches with strings found in the previous 4K bytes of data already read in. Any matches are encoded as pointers sent to the output stream.

LZ77 and its variants make attractive compression algorithms. Maintaining the model is simple; encoding the output is simple; and programs that work very quickly can be written using LZ77. Popular programs such as PKZIP and LHarc use variants of the LZ77 algorithm, and they have proven very popular.

The LZ78 program takes a different approach to building and maintaining the dictionary. Instead of having a limited-size window into the preceding text, LZ78 builds its dictionary out of all of the previously seen symbols in the input text. But instead of having carte blanche access to all the symbol strings in the preceding text, a dictionary of strings is built a single character at a time. The first time the string “Mark” is seen, for example, the string “Ma” is added to the dictionary. The next time, “Mar” is added. If “Mark” is seen again, it is added to the dictionary.

This incremental procedure works very well at isolating frequently used strings and adding them to the table. Unlike LZ77 methods, strings in LZ78 can be extremely long, which allows for high-compression ratios. LZ78 was the first of the two Ziv-Lempel algorithms to achieve popular success, due to the LZW adaptation by Terry Welch, which forms the core of the UNIX compress program.
Chapter 3  **SLIDING WINDOW COMPRESSION (LZ77)**

LZ77 compression uses previously seen text as a dictionary. It replaces variable-length phrases in the input text with fixed-size pointers into the dictionary to achieve compression. The amount of compression depends on how long the dictionary phrases are, how large the window into previously seen text is, and the entropy of the source text with respect to the LZ77 model.

The main data structure in LZ77 is a text window, divided into two parts. The first consists of a large block of recently decoded text. The second, normally much smaller, is a look-ahead buffer. The look-ahead buffer has characters read in from the input stream but not yet encoded.

The normal size of the text window is several thousand characters. The look-ahead buffer is generally much smaller, maybe ten to one hundred characters. The algorithm tries to match the contents of the look-ahead buffer to a string in the dictionary. A simplistic example of a text window is shown in Figure 3.1.

![Figure 3.1 A text window in use.](image)

Figure 3.1 shows a snippet of C code being compressed. The text window has a total width of 64 characters, with 16 of those characters used by the look-ahead buffer. The LZ77 algorithm, as originally conceived, issued sequences of tokens. Each token consists of three different data items which defined a phrase of variable length in the current look-ahead buffer. The three items in the token are: (1) an offset to a phrase in the text window; (2) the length of the phrase; and (3) the first symbol in the look-ahead buffer that follows the phrase.

In the above example, the look-ahead buffer contains the phrase “<MAX;j++):r.” By searching through the buffer, we find that “<MAX” is located at position 14 in the text
window. It matches the look-ahead buffer for the first four symbols. The first symbol not present in the look-ahead buffer is the space character. So this token is encoded as: 14, 4, ‘ ‘.

The compression program that implements the LZ77 algorithm first emits the token, then shifts the text window over by 5 characters, which is the width of phrase just encoded. First new symbols are then read into the look-ahead buffer, and the process repeats.

The next token issued by the compression algorithm would encode the phrase “;j+” as “40, 2, ‘+’.” The syntax of this token allows for phrases that have no match of any length in the window. If the look-ahead buffer shown above had no match, for example, it could be encoded a single character at a time using a phrase length of zero: “0, 0, ‘;’.” This method is not efficient, but it ensures that the algorithm can encode any message.

The code to implement this compression algorithm should be fairly simple. It merely has to look through the entire text window for the longest match, encode it, then shift. A brute force application of this algorithm is described.

The decompression algorithm for LZ77 is even simpler, since it doesn’t have to do comparisons. It reads in a token, output the indicated phrase, outputs the following character, shifts, and repeats. It maintains the window, but it does not work with string comparisons. A decompression program that been encoded to encode existing phrases. In a file that had one hundred consecutive ‘A’ characters, for example, we would encode the first A as (0, 0, ‘A’). Our window would then look like that shown in Figure 3.3.

We could then encode the next nine A characters as (38, 9, ‘A’). It may seem odd to use a nine-character phrase here. Though we can see the eight characters in the phrase presently
in the look-ahead buffer, the decoder won’t be able to. When the decoder receives the (38, 9, ‘A’) token, its buffer will look like Figure 3.4.

![Figure 3.4 The buffer for the decoder when it receives the (38, 9, ‘A’) token.](image)

But by examining the decompression algorithm, you can see how the decompression routine manages this trick. It sits in a loop, copying from the match position to the look-ahead buffer. After the first character has been copied, the buffer looks like Figure 8.5.

![Figure 3.5 What the buffer looks like after it copies the first character.](image)

The next time through the loop the second A character will be available to be copied though it was not in the window when the decoding started. After the entire copy is complete, along with the single character store, the buffer is ready to shift, as shown in Figure 3.6.
This illustrates a powerful feature of LZ77 compression: rapid adaptation to the character of the input stream. In this example, it encoded a sequence of ten characters when its “dictionary” had only been loaded with a single character.

**Coding**

```c
{
int window_cmp( char *w, int i, int j, int length )
int count = 0;
while ( length-- )
{
    if ( w[ i++ ] == w[ j++ ] )
        count++;
    else
        return( count );
}
return( count );
}

match_position = 0;
match_length = 0;
for ( i = 0 ; i < ( WINDOW_SIZE - LOOK_AHEAD_SIZE ); i++ )
{
    length = window_cmp( window, i, LOOK_AHEAD, LOOK_AHEAD_SIZE );
    if ( length > match_length )
    {
        match_position = i;
        match_length = length;
    }
}
encode( match_position, match_length,
window[ LOOK_AHEAD+match_length ] );
memmove( window, window+match_length+1, WINDOW_SIZE - match_length );
```
for ( i = 0 ; i < match_length + 1 ; i++ )
    window[ WINDOW_SIZE - match_length + i ] = getc(input);

decode( &match_position, &match_length, &character );
fwrite( window+match_position, 1, match_length, output );
putc( character, output );
for ( i = 0 ; i < match_length ; i++ )
    window[ LOOK_AHEAD+i ] = window[ match_position+i ];
    window[ LOOK_AHEAD+i ] = character;
memmove( window, window+match_length+1, WINDOW_SIZE - match_length );

..............
Chapter 4  LZW Compression Algorithm

4.1 Encoding using LZW

1. Initial table with initial character strings
2. P=first input character
3. WHILE not end of input stream
4. C=next input character
5. IF P+C is in the string table
6. P=P+C
7. ELSE
8. output the code for P
9. add P+C to the string table
10. P=C
11. END WHILE
12. output code for P
Example 1: Compression using Algorithm

Example 1: Use the LZW algorithm to compress the string

BABAABAAA
**Example 1: LZW Compression Step 1**

BABAABAAA

↑

C = empty

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output code</td>
<td>representing</td>
<td>codeword</td>
<td>string</td>
</tr>
</tbody>
</table>
### Example 1: LZW Compression Step 2

BABAABAAA

↑

C = empty

\( P = B \)

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output code</td>
<td>representing</td>
<td>codeword</td>
<td>string</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>256</td>
<td>BA</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>257</td>
<td>AB</td>
</tr>
</tbody>
</table>
### Example 1: LZW Compression Step 3

```
BABAABAAA
↑
P=A
C=empty
```

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output code</td>
<td>representing</td>
<td>codeword</td>
<td>string</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>256</td>
<td>BA</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>257</td>
<td>AB</td>
</tr>
<tr>
<td>256</td>
<td>BA</td>
<td>258</td>
<td>BAA</td>
</tr>
</tbody>
</table>
Example 1: LZW Compression Step 4

BABAABAAAA

↑

P=A

C=empty

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output code</td>
<td>representing</td>
<td>codeword</td>
<td>string</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>256</td>
<td>BA</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>257</td>
<td>AB</td>
</tr>
<tr>
<td>256</td>
<td>BA</td>
<td>258</td>
<td>BAA</td>
</tr>
<tr>
<td>257</td>
<td>AB</td>
<td>259</td>
<td>AB</td>
</tr>
</tbody>
</table>
Example 1: LZW Compression Step 5

BABAABAAA
↑
P = A
C = A

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output code</td>
<td>representing</td>
<td>codeword</td>
<td>string</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>256</td>
<td>BA</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>257</td>
<td>AB</td>
</tr>
<tr>
<td>256</td>
<td>BA</td>
<td>258</td>
<td>BAA</td>
</tr>
<tr>
<td>257</td>
<td>AB</td>
<td>259</td>
<td>AB</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>260</td>
<td>AA</td>
</tr>
</tbody>
</table>
Example 1: LZW Compression Step 6

BABAABAAA  
↑  
P=AA  
C=empty

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output code</td>
<td>representing</td>
<td>codeword</td>
<td>string</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>256</td>
<td>BA</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>257</td>
<td>AB</td>
</tr>
<tr>
<td>256</td>
<td>BA</td>
<td>258</td>
<td>BAA</td>
</tr>
<tr>
<td>257</td>
<td>AB</td>
<td>259</td>
<td>AB</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>260</td>
<td>AA</td>
</tr>
<tr>
<td>260</td>
<td>AA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.2 **Decoding using LZW**

1. Initialize table with single character strings
2. OLD = first input code
3. output translation of OLD
4. WHILE not end of input stream
5. NEW = next input code
6. IF NEW is not in the string table
7. S = translation of OLD
8. S = S+C
9. ELSE
10. S = translation of NEW
11. output S
12. C = first character of S
13. OLD + C to the string table
14. OLD = NEW
15. END WHILE
**Example 2: LZW Decompression Step 1**

<66><65><256><257><65><260>  OLD=65   S=A  
↑  NEW=66   C=A

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Codeword</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>256</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>256</td>
<td>BA</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: LZW Decompression Step 2

<66><65><256><257><65><260>       OLD=256   S=BA
       ↑               NEW=256   C=B

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td></td>
<td>Codeword</td>
<td>string</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>256</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>257</td>
<td>AB</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: LZW Decompression Step 3

<66><65><256><257><65><260>  OLD=257  S=AB
↑  NEW=257  C=A

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Codeword</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>256</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>257</td>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>258</td>
<td>BAA</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: LZW Decompression Step 4

\(<66>\langle 65>\langle 256>\langle 257>\langle 65>\langle 260>\rangle \uparrow \)

OLD=65  S=A  
NEW=65  C=A

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td></td>
<td>Codeword</td>
<td>string</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>256</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>257</td>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>258</td>
<td>BAA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>259</td>
<td>ABA</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: LZW Decompression Step 5

<66><65><256><257><65><260>        OLD=260   S=AA
↑                                NEW=260   C=A

<table>
<thead>
<tr>
<th>ENCODER</th>
<th>OUTPUT</th>
<th>STRING</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Codeword</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>256</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>257</td>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>258</td>
<td>BAA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>259</td>
<td>ABA</td>
<td></td>
</tr>
<tr>
<td>AA</td>
<td>260</td>
<td>AA</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5  Problems with LZ77

The implementation of LZ77 shown here is deliberately crude. It also has to be considered a somewhat liberal interpretation of the algorithm. The authors presented little discussion of implementation details when they presented their method.

There is clearly a major performance bottleneck in the LZ77 approach. When encoding, it has to perform string comparisons against the look-ahead buffer for every position in the text window. As it tries to improve compression performance by increasing the size of the window, and thus the dictionary, this performance bottleneck only gets worse. On the bright side, however, the decompression portion of this algorithm does not have to suffer through this bottleneck. Since it only copies the phrases, it can operate at a much higher rate. Even better, the LZ77 decompressor will be severely affected by increases in either the size of the text window or the look-ahead buffer.

A second performance problem occurs with the way the sliding window is managed. For conceptual convenience, the discussion here treated the sliding window as though it were truly sliding “across” the text, progressing from the end of the buffer to the front as the encoding process was executed.

This may be conceptually superior, but it is certainly not the best way to code an LZ77 program. In fact, it is much better to have a sliding index or pointer into a fixed buffer. Instead of moving the phrases toward the front of the window, a sliding pointer would keep the next in the same place in the window and move the start and end pointers along the buffer as text is encoded.

Besides the CPU cost problems, the LZ77 algorithm has a major efficiency problem. When encoding phrases, LZ77 achieves good compression rapidly. Even if the phrases being substituted for input text are short, they will still generally cause very effective compression to take place.
Chapter 6  Improvements

LZSS compression seeks to avoid some of the bottlenecks and performance problems in the original LZ77 algorithm. It makes two major changes to the way the algorithm works. The first is in the way the text window is maintained. Under LZ77, the phrases in the text window were stored as a single contiguous block of text, with no other organization on top of it. LZSS still stores text in contiguous windows, but it creates an additional data structure that improves on the organization of the phrases. LZSS uses a special code to indicate when the end of the compressed data is reached. LZSS instead uses a single bit as a prefix to every output token to indicate whether it is an offset/length pair or a single symbol for output. When outputting several consecutive single characters, this method reduces the overhead from possibly several bytes per character down to a single byte per character.

Another technique which speeds up both compression and expansion is to create a “ghost buffer” at the end of the text window. In the case of this program the ghost buffer would hold seventeen characters, which would be identical to the characters in the first seventeen locations of the window.

To effectively sidestep the problems of LZ77, Ziv and Lempel developed a different form of dictionary based compression. This algorithm, popularly referred to as LZ78, was published in “Compression of Individual Sequences via Variable-Rate Coding”.

LZ78 abandons the concept of a text window. Unlike LZ77, LZ78 does not have a ready-made full of text to use a dictionary.
REFERENCES


