**Laboratory work**

**Topic: Coding information using the LZW algorithm**

**The purpose of the work** : to study the LZW coding method

**Theoretical part**

**I . Algorithm of the LZW method**

Statistical coding methods aimed at compressing the transmitted information - such as Huffman or Shannon- Fano prefix codes - require preliminary analysis of the entire coded document and the compilation of a code table. The latter must be known to the decoder or, in the general case, be attached to the coded document.

Is it possible to implement the process of encoding with compression without preliminary statistical analysis? In other words, we want to organize encoding of the source in a streaming mode, i.e. as the encoded data arrives, and form a code table (dictionary) simultaneously with encoding (or with a slight delay), dynamically replenishing it taking into account “past experience” — the initial piece of the document already encoded in this way. This is exactly what is proposed in the LZW algorithm.

The algorithm got its name from the first letters of the names of its developers - Lempel , Ziv and Welch . Unlike RLE, compression in it is performed by means of identical byte chains. The LZW algorithm is the most famous representative of the dictionary methods family.

This algorithm is a modification of another method by Abraham Lempel ( Abraham Lempel ) and Jacob Ziv ( Jacob Ziv ) - LZ78. Author of the modification - Terry Welch ( Terry Welch ).

LZW compression replaces character strings with some codes. This is done without any analysis of the input text. Instead, a table of strings is looked up each time a new character string is added. Compression occurs when a code replaces a character string. The codes generated by the LZW algorithm can be of any length, but they must contain more bits than a single character.

The compression process is as follows. The input stream characters are read sequentially and a check is made to see if such a string exists in the created string table. If such a string exists, the next character is read, and if the string does not exist, the code for the previously found string is entered into the stream, the string is entered into the table, and the search begins again.

For example, if byte data (text) is compressed, then the table will have 256 rows (from "0" to "255"). If a 10-bit code is used, then the values in the range from 256 to 1023 remain for the codes for the rows. New rows form the table sequentially, i.e., the row index can be considered its code.

The decoding algorithm requires only the encoded text as input, since it can reconstruct the corresponding conversion table directly from the encoded text. The algorithm generates a uniquely decodable code by adding a new string to the string table each time a new code is generated. LZW constantly checks whether the string is already known, and if so, outputs the existing code without generating a new one. Thus, each string will be stored in a single copy and have its own unique number. Therefore, when decrypting, a new string is generated when a new code is received, and when a known code is received, the string is retrieved from the dictionary.

Conventionally, the coding algorithm can be described as follows:

1. Initialize the dictionary with all possible single-character phrases. Initialize the input phrase ω with the first character of the message.
2. Read the next symbol K from the encoded message.
3. If END\_OF\_MESSAGE, then output the code for ω, otherwise:
4. If the phrase ω(K) is already in the dictionary, assign the input phrase to the value ω(K) and go to Step 2, otherwise output the code ω, add ω(K) to the dictionary, assign the input phrase to the value K and go to Step 2.
5. End

**Pseudocode of the LZW compression algorithm**

STRING = the next character from the input stream

WHILE input stream is not empty DO

SYMBOL = the next character from the input stream

IF STRING+CHAR in string table THEN

STRING = STRING+CHAR

ELSE

output the code for STRING to the output stream

add to table strings STRING+SYMBOL

STRING = CHAR

END OF IF

END OF WHILE

output the code for STRING to the output stream

Each time a new code is generated, a new string is added to the string table. LZW constantly checks whether the string is already known, and if so, outputs the existing code without generating a new one.

The peculiarity of LZW is that for decompression we do not need to save the table of lines in the file for unpacking. The algorithm is built in such a way that we are able to restore the table of lines using only the code stream.

Now imagine that we have received the encoded message above and we need to decode it. First of all, we need to know the initial dictionary, and we can reconstruct subsequent dictionary entries on the fly, since they are simply a concatenation of previous entries.

**Pseudocode of the LZW decompression algorithm:**

read OLD\_CODE

output OLD\_CODE

SYMBOL = OLD\_CODE

WHILE input stream is not empty DO

read NEW\_CODE

IF NOT in translation table NEW\_CODE THEN

STRING = translate OLD\_CODE

STRING = STRING+CHAR

ELSE

STRING = translate NEW\_CODE

END OF IF

output LINE

CHAR = first character of STRING

add to translation table OLD\_CODE+SYMBOL

OLD\_CODE = NEW\_CODE

END OF WHILE

**Example of coding**

Let us compress the sequence " **abacabadabacabae** ".

Step 1: Then, according to the algorithm described above, we will add “a” to the initially empty string and check if the string “a” is in the table. Since we entered all the strings of one character into the table during initialization, the string “a” is in the table.

Step 2: Next we read the next character “b” from the input stream and check if the string “ ab ” is in the table. There is no such string in the table yet.

Add to table <5> “ ab ”. To stream: <0>;

Step 3: “ ba ” — no. To table: <6> “ ba ”. To stream: <1>;

Step 4: “ ac ” — no. To table: <7> “ ac ”. To stream: <0>;

Step 5: “ ca ” — no. To table: <8> “ ca ”. To stream: <2>;

Step 6: “ ab ” — is in the table; “ aba ” — is not. To the table: <9> “ aba ”. To the stream: <5>;

Step 7: “ ad ” — no. To table: <10> “ ad ”. To stream: <0>;

Step 8: “ da ” — no. To table: <11> “ da ”. To stream: <3>;

Step 9: “ aba ” — is in the table; “ abac ” — is not. To the table: <12> “ abac ”. To the stream: <9>;

Step 10: “ ca ” — is in the table; “ cab ” — is not. To the table: <13> “ cab ”. To the stream: <8>;

Step 11: “ ba ” — is in the table; “ bae ” — is not. To table: <14> “ bae ”. To stream: <6>;

Step 12: And finally the last line is “e”, after that comes the end of the message, so we just output <4> to the stream.

So we get the encoded message " **0 1 0 2 5 0 3 9 8 6 4** ", which is 11 bits shorter.

Table 1. Dictionary construction during decoding

| **Data** | | **At the input** | **New record** | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Bits** | **Code** | **Full** | | **Partial** | |
| 000 | 0 | a | - | - | 5: | a? |
| 001 | 1 | b | 5: | ab | 6: | b? |
| 000 | 0 | a | 6: | ba | 7: | a? |
| 010 | 2 | c | 7: | ac | 8: | c? |
| 101 | 5 | ab | 8: | ca | 9: | ab? |
| 000 | 0 | a | 9: | aba | 10: | a? |
| 011 | 3 | d | 10: | ad | 11: | d? |
| 1001 | 9 | aba | 11: | da | 12: | aba? |
| 1000 | 8 | ca | 12: | abac | 13: | ca? |
| 0110 | 6 | ba | 13: | cab | 14: | ba? |
| 0100 | 3 | e | 14: | bae | - | - |

**Lab assignment:**

Write a program that implements encoding and decoding (\*) of information using the LZW algorithm.

The program must accept text either from the input window on the main form or read from a file as input information. Calculate and output the compression ratio.

**Application**

**Example of implementation of the LZW method**

public class Program

{

public static void Main(string[] args )

{

List<int> compressed = Compress("string to be compressed");

Console.WriteLine ( string.Join (", ", compressed));

string decompressed = Decompress(compressed);

Console.WriteLine (decompressed);

}

public static List<int> Compress(string uncompressed)

{

Dictionary<string, int> dictionary = new Dictionary<string, int>();

for (int i = 0; i < 256; i ++)

dictionary.Add (((char) i ). ToString (), i );

string w = string.Empty ;

List<int> compressed = new List<int>();

foreach (char c in uncompressed)

{

string wc = w + c;

if ( dictionary.ContainsKey ( wc ))

{

w = wc ;

}

else

{

compressed.Add (dictionary[w]);

dictionary.Add ( wc , dictionary.Count );

w = c.ToString ();

}

}

if (! string.IsNullOrEmpty (w))

compressed.Add (dictionary[w]);

return compressed;

}

public static string Decompress(List<int> compressed)

{

Dictionary<int, string> dictionary = new Dictionary<int, string>();

for (int i = 0; i < 256; i ++)

dictionary.Add ( i , ((char) i ). ToString ());

string w = dictionary[compressed[0]];

compressed.RemoveAt (0);

StringBuilder decompressed = new StringBuilder(w);

foreach (int k in compressed)

{

string entry = null;

if ( dictionary.ContainsKey (k))

entry = dictionary[k];

else if (k == dictionary.Count )

entry = w + w[0];

decompressed.Append (entry);

dictionary.Add ( dictionary.Count , w + entry[0]);

w = entry;

}

return decompressed.ToString ();

}

}