OPTIMIZATION OF LZ77 DATA COMPRESSION ALGORITHM

Hemraj Kumawat  
CSE, IIT Jodhpur, Jodhpur, India

Jitendra Chaudhary  
CSE, IIT Jodhpur, Jodhpur, India

ABSTRACT

Data compression refers to reducing the amount of space needed to store data or reducing the amount of time needed to transmit data. Many data compression techniques allow encoding the compressed form of data with different compression ratio. In particular, in the case of LZ77 technique, it reduces the data concurrency of an input file. In the output of this technique it conveys more information that is actually not needed in practical. Removing the extra information from the encoded file that makes this algorithm more optimal. Our task is to identify how much extra information it conveys and how can we minimize it so that there is no trouble at the time of decoding. Basically the encoded output of LZ77 is the sequence of triplets (a structure of encoded output) that is in binary and having fix size. For making the triplets of fix size, sometimes we are creating unnecessary information. We present the method of variable triplet size as a way to improve LZ77 compression and demonstrate it through many experiments. In our optimization algorithm we are getting more compression ratio compare to the conventional LZ77 data compression algorithm.

Keywords:  
**Look-ahead Buffer:** The look-ahead buffer contains characters yet to be encoded. This buffer starts where the Search buffer ends and during the algorithm the Search buffer extends into the look-ahead buffer.  
**Match Length:** The Match Length is the length of largest matching block in the look-ahead buffer. These pairs are called triplets, consisting of offset, matching length and code word of character. If the character is matching then next character code word is used, otherwise same character code word is used.  
**Offset:** The actual distance between the current position of the pointer and the look-ahead buffer is known as offset.  
Search Buffer: The Search Buffer represents the most recently encoded characters.  
**Sliding Window:** The Structure for Data manipulation, in which the Data is held The Sliding Window, is divided into two parts as Search buffer and look-ahead buffer.
1. INTRODUCTION

LZ77 algorithm achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the input (uncompressed) data stream. A match is encoded by a pair of numbers called a length-distance pair. Some common convention and definition of the words that we are using in this paper.

2. CONVENTIONAL LZ77 ALGORITHM

LZ77 compression algorithm exploits the fact that words and phrases within a text file are likely to be repeated. When there is repetition, they can be encoded as a pointer to an earlier occurrence, with the pointer accompanied by the number of characters to be matched. It is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source.

In the LZ77 approach the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window which consists of two parts: a search buffer that contains a portion of the recently encoded sequence and a look ahead buffer that contains the next portion of the sequence to be encoded. The algorithm searches the sliding window for the longest match with the beginning of the look-ahead buffer and outputs a reference (a pointer) to that match. It is possible that there is no match at all, so the output cannot contain just pointers. In LZ77 the reference is always represented as a triplet \(<o,l,c>\), where ‘o’ is an offset to the match, ‘l’ is length of the match and ‘c’ is the next symbol after the match. If there is no match, the algorithm outputs a null-pointer (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer. The values of an offset to a match and length must be limited to some maximum constant. For this algorithm we have to define the length of the look-ahead buffer, search buffer. The symbol is usually encoded in 8 bit. Moreover the compression performance of LZ77 mainly depends on these values. Generally the search buffer length is more than the look-ahead-buffer size. So the total triplet size:

\[ S_T = \left\lfloor \log_2(\text{search buffer length}) \right\rfloor + \left\lfloor \log_2(\text{look-ahead buffer length}) \right\rfloor + 8 \]

While (look-ahead Buffer not empty) {
    get a reference (position, length) to longest match;
    if (length > 0)
        {
            output (position, length, next symbol);
            shift the window length+1 positions along;
        } else {
            output (0, 0, first symbol in the look-ahead buffer);
            shift the window 1 character along;
        } }
We can have better understanding with an example- “aacaacabcabaac” For this example the size of look-ahead buffer is 6 and search buffer is 4.

<table>
<thead>
<tr>
<th>Triple</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- &lt;0, 0, a&gt;</td>
<td>00000011000001</td>
</tr>
<tr>
<td>-- &lt;1, 1, c&gt;</td>
<td>00100111000011</td>
</tr>
<tr>
<td>-- &lt;3, 4, b&gt;</td>
<td>01110011000010</td>
</tr>
<tr>
<td>-- &lt;3, 3, a&gt;</td>
<td>01101111000001</td>
</tr>
<tr>
<td>-- &lt;1, 2, c&gt;</td>
<td>00101011000011</td>
</tr>
</tbody>
</table>

The triplet length for this example is 14(3+3+8). So here the encoded binary string of this example.

00000011000001001001110000110111001100001001101101000011

The decoding is much faster than the encoding in this process because we have to move our pointer with fixed length (triple length-14 for this example) and it is one of the important features of this process. From this way we get the triplets .Now we can easily decode to original data by reversing the encoding process.

3. OPTIMIZATION OF LZ77

As we have described earlier the triplets of LZ77 algorithm have fix size. In the case, when offset is equal to the matching length we can modify the structure of triplets and represent it with the new structure that have only \(<l,c>\) where l is the matching length and ‘c’ is the next symbol after the match .We called this new structure as “doublet” in this paper .All the things are same as the conventional LZ77 algorithm except replacing the triplet with doublet in the case of matching length equal to offset length.

\[ S_B = \log_2(\text{look-ahead buffer length}) + 8 \]

By replacing the triplet with doublet we are saving \([\lceil \log_2(\text{search buffer length}) \rceil] + 8\) number of bits per matching case. The exact algorithms is described in the block.
While (look-ahead Buffer not empty) {
    get a reference (position, length) to longest match;
    if (length > 0)
        {
            If (length== position){
                output (length, next symbol);
                shift the window length+1 positions along;
            }
            else{
                output (position, length, next symbol);
                shift the window length+1 positions along;
            }
        }
    else {
        output (0, first symbol in the look-ahead buffer);
        shift the window 1 character along;
    }
}

We will have better understanding of this optimized algorithm with the previous section example- “aacaacabcabaaaac”

<table>
<thead>
<tr>
<th>Triplet/doublet</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&lt;0, a&gt;</td>
<td>00011000001</td>
</tr>
<tr>
<td>--&lt;1, c&gt;</td>
<td>00111000011</td>
</tr>
<tr>
<td>--&lt;3, 4, b&gt;</td>
<td>01110011000010</td>
</tr>
<tr>
<td>--&lt;3, a&gt;</td>
<td>01111000001</td>
</tr>
<tr>
<td>--&lt;1, 2, c&gt;</td>
<td>00101011000011</td>
</tr>
</tbody>
</table>

The length of triplet for this example is 14(3+3+8) and the doublet length is 11(3+8). So here the encoded binary string of this example
001100000100111000011011100110000100111000000010010101100011

doublet  doublet  triplet  doublet  triplet
----------------|----------------|----------------|----------------|----------------|

The decoding process is slower than the conventional LZ77 Decoding. In this algorithm we have to move our pointer with the variable size (doublet and triplet length) .But the problem in decoding is how we will identify which one is doublet and which one is triplet. So the decoding process is described in the next section. Once we decode encoded file to triplets and doublet then we can easily get to original data by reversing the encoding (data to triplet/doublet) process.
4. DECODING PROCESS

For decoding the binary string that contains doublets and triplets, first we have to identify which one is doublet’s binary string and which one is triplet binary string. To identify this we are using the concept of delimiter that will depend on the size of the sliding window length. We will put the delimiter before the doublet binary string.

**The effective doublet length** is: doublet size + delimiter size.

Now each binary substring is starting from the delimiter. Now for decoding we move the pointer from starting of the binary string and check whether the first consecutive k bits are equal to the delimiter or not where k is the length of delimiter. If it is equal then move the current pointer with the effective doublet length ahead and make this substring to the doublet substring otherwise move the pointer with the triplet length and make this substring to the triplet substring.

So the decoding algorithm is given below

```
Effective doublet length = doublet size + delimiter length
//delimiter is a binary string that depends on the offset length
Starting the moving pointer from 0
While(the moving pointer m<total binary length){
    if((m to m+ delimiter’s length substring) == delimiter)
        Move the pointer m with effective doublet length ahead.
        Get the doublet binary substring form (m + delimiter’s length) to (m + effective length)
    }
    else{
        Move the pointer m with triplet length ahead.
        Get the triplet binary substring form m to (m + triplet length)
    }
```

So here the above example’s binary string after using the delimiter before the doublets (here we are using the delimiter=”1”)

```
100011000001100110000110111001100001010111100000100101011000011
```

```
doublet doublet triplet doublet triplet
---------------------- ----------------------
```

So the substring of doublet length just after the blue one is the doublet binary substring and the rest substrings of triple size are the triplet binary string. We can easily identify the delimiter (blue one for this example) by moving the pointer with appropriate length according the doublet and triplets.

But what will happen when the offset’s first k bits are equal to the delimiter then this algorithm is not valid, where k is the length of delimiter. We cannot decompress the binary string from this delimiter.

Let’s see this from an example:
For maximum search buffer length=31 and look-ahead buffer=7 and delimiter =”111”

Binary string

```
1110001100000111100011000010111010111001101100011
```

```
|___|---doublet----||___|--doublet----||___|----triplet----------||___|--doublet---
```

From the starting we will see that first two strings are doublet string. The third substring is actually triplet substring but from our algorithm it reads out as the doublet substring because the first
3 bits are equal to the delimiter. This will affect the rest decoding process. Eventually, we will get the wrong decoding output.
So we are bounding our delimiter size by:
Delimiter size = no. of first consecutive 1’s in binary representation of sliding window length +1
All the bits in the delimiter are 1’s. From this Delimiter string we are getting the correct decoding output.

5. ALGORITHMS ANALYSIS

C: Total no of Matching (offset=Matching Length)
CR: Compression ratio
Op LZ77: Optimized LZ77 Algorithm
LZ77: Conventional LZ77 Algorithm
D: % Difference between conventional LZ77 and Optimized LZ77 = (Op LZ77 – LZ77) / LZ77
Max. Offset Value = maximum offset value = search buffer length

<table>
<thead>
<tr>
<th>Serial Number</th>
<th>File size (in bytes)</th>
<th>Comparison between conventional LZ77 and Optimized LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Max. Offset Value=111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max. Offset Value=223</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max. Offset Value=447</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>61,305</td>
<td>2208</td>
</tr>
<tr>
<td>2</td>
<td>4,21,144</td>
<td>11417</td>
</tr>
<tr>
<td>3</td>
<td>7,79,959</td>
<td>17044</td>
</tr>
<tr>
<td>4</td>
<td>73,246</td>
<td>2820</td>
</tr>
<tr>
<td>5</td>
<td>1,09,684</td>
<td>3755</td>
</tr>
<tr>
<td>6</td>
<td>6,04,919</td>
<td>19281</td>
</tr>
<tr>
<td>7</td>
<td>1,51,610</td>
<td>3489</td>
</tr>
<tr>
<td>8</td>
<td>1,30,725</td>
<td>3309</td>
</tr>
<tr>
<td>9</td>
<td>4,27,180</td>
<td>14369</td>
</tr>
<tr>
<td>10</td>
<td>6,78,036</td>
<td>23537</td>
</tr>
<tr>
<td>Average</td>
<td>10123</td>
<td>1.211</td>
</tr>
</tbody>
</table>

6. CONCLUSION

As we can see from the above analysis that for maximum offset value (111) the improved compression ratio is 1.896 and for maximum offset value (223) the improved compression ratio is 1.154. From this analysis result we can see that maximum offset value is increasing, the improved compression ratio is decreasing. When we increase the search buffer length then the total number of matching (offset=matching length) will be decreased. The improved compression is directly proportional to the matching. The improved compression will be more when the matching is more. So generally this optimal algorithm is more effective, where either conventional LZ77 algorithm is not so optimal or the input has less repetition.
REFERENCES

337A Universal Algorithm for Sequential Data Compression


- Links of the text files used in Analysis are given below (sorted by the index of the table)
  5. http://www.gutenberg.org/files/25731/25731-0.txt