An Optimized Compression Algorithm

Sami Serhan*

Received on Nov. 28, 2002  Accepted for publication on July 22, 2003

Abstract

In this paper we propose an optimized compression approach of Sequitur algorithm. The proposed approach is a compromise between the LZW and Sequitur compression algorithms.

The proposed enhancement of Sequitur improves the efficiency of the algorithm in terms of time complexity and number of operations. This enhancement requires to take the advantages of both LZW and Sequitur algorithms to produce a more optimized version.

Keywords: Compression, Decompression, Lossless, Lossy, LZW, Sequitur.

1. Introduction

Data compression is an interesting and challenging issue in the design and implementation of many computer applications and computerized systems.

Compression relies on the fact that data is redundant, and at some extent it is generated according to some rules[12]. LZW has its own approach and implementation structure which classifies redundant patterns on a file in a dictionary during the compression and decompression phases[11]. But the basic structure and operation of it have drawbacks due to the compression ratio using standard implementation. The Sequitur approach and implementation structure classify redundant patterns according to grammatical rules productions using two constraints to provide a good compression ratio [1]. The major drawback of sequitur during the compression phase is the overhead and time complexity.

Observing the nature of compression/decompression of both LZW and Sequitur, the proposed approach takes the benefits of both algorithms, and avoids the major drawbacks taking into consideration both the compression and decompression phases. So the resulted algorithm achieves the maximum possible compression ratio and the most compression speed.
The classification of patterns in the proposed approach has a hybrid nature of both LZW and Sequitur. The output file has the Sequitur file format which means faster decompression compared to the LZW file output which needs to rebuild its dictionary again during the decompression process.

A gap exists between theory and practice. The theory holds when the block length $n$ approaches infinity. In real-time compression, the compression algorithm must wait for $n$ consecutive source samples before it can begin the compression. When $n$ is large, this wait (or delay) may be too long. For example, in real-time speech compression, the speech signal is sampled at 8000 samples/second. If $n$ is say 4000, then the compression delay is half a second. In a two-way conversation, this long delay may not be desirable.

The theory does not take into consideration the complexities associated with the compression and decompression operations. Typically, as the block length $n$ increases, the complexities also increase. Often, the rate of increase is exponential to $n$.

The theory assumes that the statistical properties of the source is known. In practice, this information may not be available [7].

2. Terms and Definitions

**Compression**: transforming the data in the source file into some representation which length is as small as possible.

**Decompression**: retrieving the data of the source file from its compressed representation.

**Lossless**: compression method where the restored data file is identical to the original.

**Lossy**: compression method which allows degradation of data restored.

**Bit Rate**: average number of bits per original data element after compression.

**Character**: the basic data element in the charstream.

**Charstream**: a sequence of data to be encoded.

**Dictionary**: a table of strings. Every string is assigned a code word according to its index number in the dictionary.
3. Previous Work

In this section we introduce a brief description about LZW and Sequitur compression algorithms.

3.1 LZW Algorithm

The basic idea is to parse the input sequence into non-overlapping blocks of different lengths while constructing a dictionary of blocks which have been scanned [8].

LZW compression is called so because of the names of its developers: A. Lempel and J. Ziv, with later modifications by Terry A. Welch. It is the foremost technique for general purpose data compression due to its simplicity and versatility. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size. LZW also behaves well when presented with extremely redundant data files, such as tabulated numbers, computer source code, and acquired signals. Compression ratios of 5:1 are common for these cases. LZW is the basis of several personal computer utilities that claim to “double the capacity of your hard drive.”

LZW compression uses a code table. A common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists entirely of 12-bit codes, each referring to one of the entries in the code table. Decompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. For example, if only these first 256 codes were used, each byte in the original file would be converted into 12 bits in the LZW encoded file, resulting in a 50% larger file size. During decompression, each 12-bit code would be translated via the code table back into a single byte. Of course, this would not be the useful choice [4].

3.2 Sequitur Algorithm.

Sequitur is an algorithm that infers a hierarchical structure from a sequence of discrete symbols by replacing repeated phrases with a grammatical rule that generates the phrase, and continuing this process recursively. The result is a hierarchical representation of the original sequence, which offers insights into its lexical structure. The algorithm is driven by two constraints that reduce the size of the grammar, and produce structure as a by-product.

Based on the idea of subtracting subsequences that occur more than once into rules and continuing this operation recursively, the algorithm works by maintaining
two constraints: every diagram in the grammar must be unique, and every rule must be used more than once. Sequitur operates incrementally [1].

4. Suggested Optimization

This section explains the main idea behind the proposed optimization on the Sequitur algorithm. It also contains the proposed algorithm.

4.1 Significance of the proposed optimization

Grammar hierarchy based compression technique (Sequitur) works by creating grammars from the text incrementally. The major drawback of the Sequitur algorithm is complexity. It has an asymptotic quadratic complexity of the compression process, and an incremental process in generating rules. This critical feature of this algorithm can rise in the on-line, or real-time applications, and it can not be avoided due to grammar induction.

Dictionary based (LZW) compression technique tends to deal with the file to be compressed as a repeated sequence of letters or sentences. So it scan the text in a certain way and referencing each repeated code with a special index in a dictionary and finally replacing the original text by the reference code that exists in the dictionary.

In practice there is no way of telling that a certain file contents will behave in a certain manner. Otherwise it will be better to use some other specific compression algorithm that tends to compress the file according to a well studied method to achieve an efficient compression results, thus servicing a specific file format. But in general in Lossless file compression there is no way to predict how the file would look like or what would it contain [10].

Hashing tables are a very good technique in terms of constant time for index retrieval. Still using hashing tables has its own problems. As entries grow in a hashed table it becomes more difficult to handle, and two entries having the same hash formula will leads to collision. Then a special implementation is needed to overcome the collisions problem which implies a more difficult code to write and more overhead [5].

One problem occurs when using LZW that's when a file is rich of patterns and its size is large, the indices of the hash table will be soon over written. This implies that if an older pattern is occurred, the program will have the following side effects: building a new pattern, losing the ability to compress using this pattern, lowering the chance of a faster performance, more complicated code to deal with the situation.

Resizable arrays or linear linked lists showed a solution. The best approach is the Binary Search Trees (BST) as it has a very low complexity in its best case $O(n\log n)$.
and a very easy approach to implement. A BST has no visible upper boundary because it is a dynamic structure [4]. It has a very fast and simple way to access the required items upon search. So a BST can cover all the patterns of the file and provides a simple way to access them during storage or retrieval operations or other checks during any required operations[5].

As the size of the file grows the BST grows. The BST can grow in a fashion that will result in a linear complexity for some or most of the patterns. As the BST grows the older and possibly ‘no more’ needed patterns can be accessed faster because they are located in the higher portion of the BST while the newer and ‘mostly needed’ patterns are leaves or located at the deepest portion of the BST. This makes the hashing table technique faster and more handy approach to implement both LZW and Sequitur compression algorithms [2].

4.2 The Proposed Algorithm

The algorithm will scan the whole file to be compressed and meanwhile will put it in a buffer. Unlike Sequitur which consumes one character at a time, now each buffer is treated sequentially. A pattern of pair of characters will be scanned through the buffer while building the patterns in a hash table. If a pattern is occurred again the next character will be consumed forming a longer pattern as the LZW algorithm works. But any pattern with length greater than two will be inserted twice, first in the general hash table, or let us say the pattern collector, and the second time in another hash table which will be called the rule hash table that only contains patterns of length greater than two characters. Two hash tables holding patterns of variable length are created during the first scanning pass.

During each pass a counter of maximum pattern length is initialized and incremented to give the maximum pattern size available.

The second pass will compare the patterns in the buffer. The size of the patterns is selected according the maximum patterns size with the rules in the rules hash table. And if the pattern is not found in the rules, the length of it will be decremented until it reaches the size of two characters, if so it will be written as it is. This process transforms the original buffer into a new format.

Again the buffer will be scanned creating new patterns and the whole process is repeated until no more rules of length greater than two characters can be observed.

One important thing about this algorithm is that it does not compress better than the Sequitur algorithm, but actually its speed challenges the LZW algorithm some times. Finally why only rules consists of more than two characters are collected? The answer is that there are a lot of famous compression algorithms for calculating single characters individually like Huffman Coding or Arithmetic Coding [6]. In addition to

437
that, starting to find patterns of length over than two characters usually needs a better mechanism to predict the file flow of characters. So starting with a large pattern usually has a better compression chance if repeated, meanwhile patterns of pairs by practice has a very high probability to occur and through the expansion process of these patterns a larger probability of compression can be achieved [3,7].

The pseudo code of the proposed algorithm is shown below.

Begin
repeat
while not end of buffer do
begin
scan pairs of characters incrementally and check
if the new pattern of characters does not exist then
insert it into the table
else
repeat
consume another character if possible and adjust the
maximum pattern size and insert the pattern
if inserted then Done
if end of buffer is reached then Done
until Done
end
while not end of buffer do
begin
scan pairs of characters taking patterns according to the
maximum pattern length and check for the availability of the
patterns in the rule hash table
if the pattern exists then write its index instead of the pattern
else
repeat
decrement the pattern size and search again in the rules
hash table
if it exists then write its index instead of the pattern
until the pattern size becomes two characters
if the pattern size is two characters then write them
end
until maximum pattern size is < 3
then scan the buffer for the last time and tag the existing rules
the rule that does not exist anymore on the buffer can be substituted in the tagged rules
write the rules
write the buffer
End.
4.3 General Example

Here we introduce a simple example to show the difference in handling a given pattern by the different considered algorithms in this paper. The assumed pattern is: abcdefabcde. By tracing the handling of the given pattern using each of the considered three algorithms: LZW, Sequitur, and the proposed algorithm we can see that the proposed algorithm takes the advantages of both LZW and Sequitur. It has less complexity than Sequitur and better compression ratio than LZW.

4.3.1 LZW Example

Table1 shows how the given pattern is handled by LZW algorithm.

Table1: Handling of abcdefabcde pattern using LZW.

<table>
<thead>
<tr>
<th>Current Input</th>
<th>Operations</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>ab</td>
<td>add [ab]</td>
<td>ab</td>
</tr>
<tr>
<td>bc</td>
<td>add [bc]</td>
<td>abc</td>
</tr>
<tr>
<td>cd</td>
<td>add [cd]</td>
<td>abcd</td>
</tr>
<tr>
<td>de</td>
<td>add [de]</td>
<td>abcd</td>
</tr>
<tr>
<td>fa</td>
<td>add [a]</td>
<td>abcde</td>
</tr>
<tr>
<td>ab</td>
<td>[ab] found write its index and add [abc]</td>
<td>abcdefA</td>
</tr>
<tr>
<td>cd</td>
<td>[bc] found write its index and add [cde]</td>
<td>abcdefA</td>
</tr>
<tr>
<td>e</td>
<td>add [c] as end of pattern reached</td>
<td>abcdefABe</td>
</tr>
</tbody>
</table>

4.3.2 Sequitur Example

Table2 shows the handling of the considered pattern using Sequitur algorithm.

Table2: Handling of abcdefabcde pattern using Sequitur.

<table>
<thead>
<tr>
<th>Current Input</th>
<th>Grammar</th>
<th>Diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S → a</td>
<td>{a}</td>
</tr>
<tr>
<td>B</td>
<td>S → ab</td>
<td>{ab}</td>
</tr>
<tr>
<td>C</td>
<td>S → abc</td>
<td>{ab, bc}</td>
</tr>
<tr>
<td>D</td>
<td>S → abcd</td>
<td>{ab, bc, cd}</td>
</tr>
<tr>
<td>E</td>
<td>S → abcde</td>
<td>{ab, bc, cd, de}</td>
</tr>
<tr>
<td>F</td>
<td>S → abcd</td>
<td>{ab, bc, cd, de, ef}</td>
</tr>
<tr>
<td>A</td>
<td>S → abcd + defa</td>
<td>{ab, bc, cd, de, ef, fa}</td>
</tr>
<tr>
<td>B</td>
<td>S → abcd + defa</td>
<td>{ab, bc, cd, de, ef, fa, ab} match rearrange digrams {Ac, cd, de, ef, fA}</td>
</tr>
<tr>
<td>C</td>
<td>S → AcdefAc</td>
<td>{Ac, cd, de, ef, fA, Ac} match rearrange digrams {Bd, de, ef, fB}</td>
</tr>
</tbody>
</table>
4.3.3 The proposed Algorithm Example

Table 3 shows the handling of the assumed pattern using the proposed algorithm.

Table 3: Handling of the abcdefabcde pattern using the proposed algorithm.

<table>
<thead>
<tr>
<th>Current buffer</th>
<th>Diagrams</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcdefabcde</td>
<td>{abc, bc, cd, de, ef, fa, abc, bcd}</td>
<td>A → abc</td>
</tr>
<tr>
<td></td>
<td>Rescan input due to the largest pattern and write to new buffer</td>
<td></td>
</tr>
<tr>
<td>AdefAde</td>
<td>{Ad, de, ef, fA, Adef}</td>
<td>B → Adef</td>
</tr>
<tr>
<td></td>
<td>Rescan input due to the largest pattern and write to new buffer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B → Adef is found</td>
<td></td>
</tr>
<tr>
<td>BfB</td>
<td>{Bf, fB}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>End of pattern is reached, the Grammar is S → BfB and one rule B → abcde</td>
<td></td>
</tr>
</tbody>
</table>

5. Experiments and Results

The driven example in Table 2 shows clearly the overhead consumed in the Sequitur case.

Now we will assume that all of the three algorithms will store their data using 16-bit representations. So as shown in Table 1, the LZW needs 18 bytes to store its output data, while on the other hand only 16-bytes are needed to store the output data by both Sequitur and the proposed algorithm. Now in the decompression phase the LZW needs to rebuild its dictionary while only direct substitution of the rules by the grammar is required for both Sequitur and the proposed algorithm.
5.1 Experimental Results

A set of experiments were done on files of different sizes to see the difference between the three considered algorithms in this paper. The experiments were applied on files of standard types.

Table 4 shows the obtained results.

Table 4: Comparison between LZW, Sequitur, and the proposed algorithm.

<table>
<thead>
<tr>
<th>File Size</th>
<th>LZW</th>
<th>SEQUITUR</th>
<th>Proposed Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Compression Ratio</td>
<td>Time</td>
</tr>
<tr>
<td>0.5 MB</td>
<td>14 s</td>
<td>80%</td>
<td>26 s</td>
</tr>
<tr>
<td>1.7 MB</td>
<td>55 s</td>
<td>60%</td>
<td>1.2 m</td>
</tr>
<tr>
<td>2.5 MB</td>
<td>1.5 m</td>
<td>62%</td>
<td>3.2 m</td>
</tr>
<tr>
<td>6 MB</td>
<td>3.25 m</td>
<td>70%</td>
<td>6.2 m</td>
</tr>
<tr>
<td>15 MB</td>
<td>7.05 m</td>
<td>69%</td>
<td>12.3 m</td>
</tr>
<tr>
<td>33 MB</td>
<td>15.9 m</td>
<td>62%</td>
<td>28.1 m</td>
</tr>
<tr>
<td>60 MB</td>
<td>27.4 m</td>
<td>75%</td>
<td>50.5 m</td>
</tr>
<tr>
<td>110 MB</td>
<td>47.2 m</td>
<td>66%</td>
<td>1.3 h</td>
</tr>
</tbody>
</table>

5.2 Analysis of Results

The compression ratio, speed, and complexity depend on the file size. In general, the proposed algorithm and Sequitur algorithm provide the same compression ratio which is better than compression ratio obtained by LZW algorithm. The gap between the compression ratio by LZW and the proposed algorithm is various depending on the file size, see figure1.

The time required for compression depends on the file size and on the algorithm used for compression. In general the required compression time for LZW and the proposed algorithm is far less than the time required for Sequitur. The compression time required for LZW algorithm is almost the same or little less than the time required for the proposed algorithm when the size of the file is relatively small. But when the file size becomes larger the proposed algorithm needs less time for compression than LZW algorithm, see figure2.

So, we can say that the proposed algorithm is recommended to be used when the critical factor is the compression ratio because it provides better compression ratio than LZW and better speed than Sequitur. But if the critical factor is the speed, then the proposed algorithm is also recommended to be used for large files. For small files LZW may provide little better speed but the proposed algorithm will provide far better compression ratio.
Figure 1: compression ratio against file size

Figure 2: compression time against file size

Conclusion

The proposed compression algorithm is a compromise between the Sequitur algorithm [1] and LZW algorithm [4,11]. It is fast as LZW and at the same time it has better compression ratio than LZW. The proposed algorithm has less complexity than Sequitur but it can achieve as the same compression ratio as Sequitur.
An Optimized Compression Algorithm

The complexity of the proposed algorithm is linear O(n), but a lot of overhead is avoided compared to Sequitur. Hashing tables are selected for implementation, but always there is an option to use binary search trees especially if the data size is huge.

The proposed algorithm has its own drawbacks when the implementation structures are not chosen wisely, but when implementation is considered as precise as possible it will provide the best results.

خوارزمية الضغط المتلي
سامي سرحان
ملخص
النحو في هذا البحث طريقة محسنة لخوارزمية Sequitur الخاصة بضغط الملفات. الطريقة LWZ و Sequitur المطروحة هي حل وسط لطريقتي Sequitur و LWZ. الخطوة المطروحة من كفاءة خوارزمية Sequitur حيث أن هذه الطريقة تأخذ بين الاعتبار حسنت الخوارزميتين LWZ و Sequitur وعدد العمليات من حيث درجة التنفيذ وعدد العمليات.

References.


