A Software Tool for Data Compression Using the LZ77 ("Sliding Window") Algorithm

Student authors: Vladan R. Djokić[1], Miodrag G. Vidojković[1]
Mentors: Radomir S. Stanković[2], Dušan B. Gajić[2]

Abstract – **Data compression is a field of computer science that is always in need of fast algorithms and their efficient implementations. Lempel-Ziv algorithm is the first which used a dictionary method for data compression. In this paper, we present the software implementation of this so-called "sliding window" method for compression and decompression. We also include experimental results considering data compression rate and running time. This software tool includes a graphical user interface and is meant for use in educational purposes.**

Keywords – **Lempel-Ziv algorithm, C# programming solution, text compression.**

I. Introduction

While reading a book it is noticeable that some words are repeating very often. In computer world, textual files represent those books and same logic may be applied there. On the other hand, we can be reasonably sure that some words occur in a very small fraction of the text sources in existence.

Following an idea of repeating words, it is logical to keep a list, or dictionary, of frequently occurring patterns. When any of those patterns appear in the source output, they are encoded with appropriate reference to the dictionary. If the pattern does not occur in the dictionary, encoding should look for some other, less efficient, method for encoding. In effect, we are splitting the input into two classes - the frequently occurring patterns and the infrequently occurring patterns. In order to name this technique as successful, occurring patterns must appear often and also the size of the dictionary must be much smaller than the number of all possible patterns.

This paper describes an implementation of the LZ77 encoding which is one of the dictionary method techniques. In Sections II and III, we describe the theoretical bases of the LZ77 coding and some other types of dictionary coding, respectively. Next, in Section IV we present a software solution for data compression using the LZ77 algorithm realized in C# programming language, along with the experimental results for data compression ratio. This application is mainly developed for educational purposes. We close the paper with some conclusions in the final section.

Student authors:
[1]Vladan R. Djokić and Miodrag G. Vidojković are with the Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Nis, Serbia, E-mail: djolecorp@elfak.rs, miske87@gmail.com.

Mentors:
[2]Radomir S. Stanković and Dušan B. Gajić are with the University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia, E-mails: radomir.stankovic@gmail.com, dusan.gajic@elfak.ni.ac.rs.

II. Lempel-Ziv Algorithm

A. Theoretical basis

The Lempel-Ziv algorithm [1] is an algorithm for lossless data compression. It is actually a whole family of algorithms, (see Figure 1) stemming from the two original algorithms that were first proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977. [1] and 1978. [2]. LZ77 and LZ78 got their name by year of publishing.

The Lempel-Ziv algorithms belong to adaptive dictionary coders [1]. On start of encoding process, dictionary does not exist. The dictionary is created during encoding. There is no final state of dictionary and it does not have fixed size.

B. The field of use

Some of modern algorithms for compression and decompression are variations of LZ77. Known compression methods like *arj*, *lha*, *zip*, *zoo*, *stac*, *auto-doubler*, *7-zip*, *DEFLATE* have roots in this algorithm. Also GIF image compression [4] and the V.42 modem standard [4] are based on the LZ algorithm.

III. LZ77 Algorithm

A. Terms used in algorithm description

In order to explain how algorithm works some terms should be explained. Sequence of characters that needs to be compressed is an input stream. One element in input stream is a character. Coding position is position of the character that is currently coded and in same time it is beginning of look-ahead buffer which size is predefined and consists of another sequence of characters. A pointer has task to point to the match in the window. It also declares length of match.

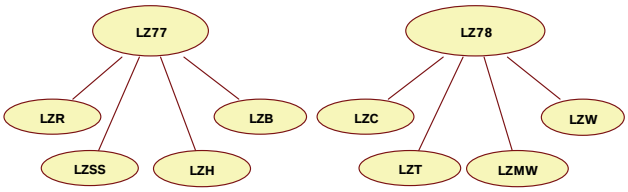

Figure 1. LZ family of algorithms.

Figure 2. LZ77 application architecture.

B. The encoding method

Idea of algorithm is to search the window for the longest match. If window reached end, it also should check beginning of the look-ahead buffer. Finally, it should output a pointer to appropriate match. Output may not contain only pointers since it may occur that there is not a single match. In that case, output is pair of zeros plus character which could not be found in dictionary. If there is a match, pair has number of repetitions, position where match is found and following character (see Table II, column Output).

C. The decoding method

Generally, decoder is much simpler than the encoder. It has to know some parameters which are used during compression and those are size of a dictionary and length of a look-ahead buffer. Then decoder should reconstruct dictionary and look-ahead buffer from original. Finally, it may start decoding process. First should get following token, if there exists any, then to find the match in its buffer. Output is complete match and character which comes after pair of numbers. Next action is to shift the matched string and the third field into the buffer. That process is repeated as long as there are tokens available.

D. Practical example

The encoding process is presented in Table II. The column Step is used only for numeric purposes so that can be seen when one turn of encoding is completed. The column Pos is current position that is coded. In first place, first character has position 1. The column Match represents the longest match found in the window. The column Char shows the first character in the look-ahead buffer after the match. Finally, the column Output, as already mentioned in section B, presents the output in the format (number, number) character. One example of encoding process may be seen in Table II. Notice how in step 3 match is longer than current dictionary and it goes through look-ahead buffer so output sequence is (3,1) S.

TABLE I
INPUT STREAM FOR ENCODING

Pos	1	2	3	4	5	6	7	8	9
Char	A	N	A	N	A	S	S	A	A

TABLE II
THE ENCODING PROCESS

Step	Pos	Match	Char	Output
1.	1	--	A	(0,0) A
2.	2	--	N	(0,0) N
3.	3	A N	S	(3,1) S
4.	7	--	S	(0,0) S
5.	8	A	A	(5,1) A

IV. SOFTWARE IMPLEMENTATION AND EXPERIMENTAL RESULTS

The software solution for LZ77 encoding and decoding that is implemented is named LZ77, and this application can encode and decode ASCII text. LZ77 is developed in C# programming language and .NET Framework 3.5. The architecture of the application is created using Visual Studio 2010 and is shown in Figure 2. Main window of application is shown in Figure 3.

The application consists of three buttons and two combo boxes. First one on the far left is “Load” which is used to load a file. It is possible to load files with any extension, textual files and special archives which are created with this programme using .lz77 extension.

After successful load, all text from input file is transferred into string and ready for compression or decompression. If we have loaded textual file, we can choose dictionary size in bits and look-ahead buffer size. Application offers several values as choice for dictionary size – from 8 to 16, 20, 24 and 28. Theoretically, dictionary size of 32 bits is possible but due to integer constraint it is not applicable. Also, length of look-ahead buffer can be changed with one of values 2, 4, 8, 16, 32, 64 and 128 characters or translated in bits, from 1 to 7 bits can be used for look-ahead buffer. Statistically is proven that optimal results are achieved with dictionary size between 10 and 12 bits and size of look-ahead buffer should be from 5 to 7 bits [7]. If user doesn’t choose any of these values, default values are used and they are 10 bits for dictionary and 5 bits for look-ahead buffer.

By clicking on “Compress” button, text is compressed using LZ77 algorithm and new window opens where output location of archive may be chosen with .lz77 extension. Besides regular LZ77 compression algorithm, already described in this paper, we have added file header which is useful for decoding. First byte is reserved for this file header. First five bits contain value of dictionary size and following three contain look-ahead buffer size. Even those bits have mini dictionary and for example, combination of 01010|101 means that dictionary size is 10 bits and look-ahead size is 5 bits. Binary value is decoded into decimal value. Now, dictionary size in characters is 2^d and size of look-ahead buffer is 2^l where d and l are values received from first bit.

If compression is successful to user is shown message box with speed of compression, measured in milliseconds. For small input files speed is usually around 0. Ratio and speed of compression may be seen in Table III.

To decompress file, first, file should be loaded using “Load” button and then by clicking on “Decompress”, it is decompressed. Note that only files created with this programme have extension .lz77 and therefore only those files may be successfully decompressed. If decompressing is successful to user is shown speed of decompression in message box, similar like compression. Speed of decompression may be seen in Table IV.

Decompress works as follows, first byte is read and values of dictionary and look-ahead are stored in variables, on way

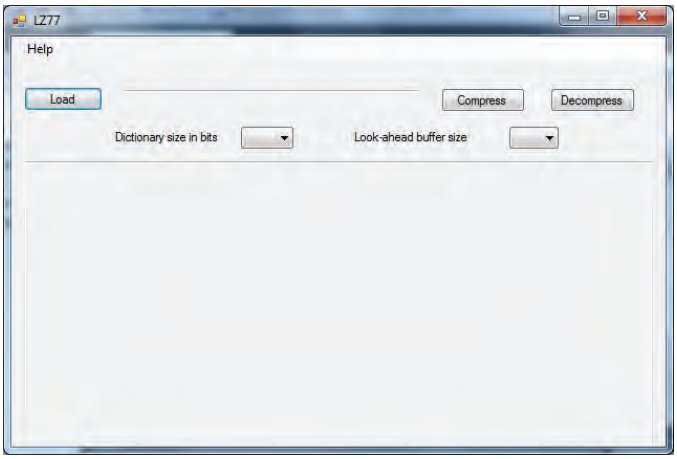

Figure 3. LZ77 application main window.

that is already explained in this paper. Then each pair is read, and returned their original value. Note that compression/decompression only works with ASCII characters since ASCII requires 7-bits for each character. Unicode coding/decoding is harder to achieve since Unicode coded character doesn’t have constant size in bits.

All tests are performed using optimal compression values for dictionary and look-ahead buffer. Also, all tests are performed on same machine HP Pavilion dv6 Notebook PC using Intel(R) Core(TM)2 Duo CPU T6600 @ 2.20GHz, 2.20GHz, 3GB Ram memory on 64-bit Operating System Win7. All values (unless there are written actual measurements) of files are measured in bytes, speed in milliseconds and ratio in percents.

TABLE III
RATIO AND SPEED OF ENCODING PROCESS

Input type	File size	Compressed	Speed	Ratio
Text- normal	5	16	0	320%
Text- repetitive	179	47	0	26%
Text- normal	640	709	0	111%
C++ code text	1528	384	16	25%
Book	12534	9853	63	79%
Book	69369	49518	281	71%
Book	544792	385953	2512	71%
Text- repetitive	760320	68361	172	9%
Text- repetitive	12MB	1MB	4200	8%
Text	100 MB	62.5MB	330 s	63%

TABLE IV
SPEED OF DECODING PROCESS

Input type	File size	Speed
Text- normal	16	0
Text- repetitive	47	0
Text- normal	709	0
C++ code text	384	0
Book	9853	56
Book	49518	405
Book	385953	2215
Text- repetitive	68361	180
Text- repetitive	1MB	7400
Text	62.5MB	350 s

V. CONCLUSION

In this paper, we have presented a software implementation of LZ77 algorithm. We have performed various tests on different file sizes and file types, and gathered experimental results which may be used in educational purposes. The application may be used in better understanding of LZ77 algorithm.

Implementation of LZ77 algorithm is giving the best results when the input file has repetitive text. Also, good compression is achieved on C++ source files and on book text. Compression is not satisfactory when the input file is too small. Normally, in real-world applications, the compression is performed on books and results in compression rates around 70-80% of the original file size.

REFERENCES

[1] Jacob Ziv, Abraham Lempel, “A universal algorithm for sequential data compression”, *IEEE Transactions On Information Theory*, Vol. It-23, No. 3, May 1977.

[2] Jacob Ziv, Abraham Lempel, "Compression of Individual Sequences via Variable Rate Coding", *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530-536, Sep. 1978.

[3] Khalid Sayood, *Introduction to Data Compression*, Morgan Kaufmann Publishers, Published 1996, Second Edition 2000.
[4] Christina Zeeh, *The Lempel Ziv Algorithm*, Seminar ”Famous Algorithms”, January 16, 2003.
[5] *http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/lz/lz77.html*, last visited on 8/12/2010-10:00.
[6] David Salomon, *Data Compression*, Fourth Edition, Springer 2007.
[7] Terry Welch, A Technique for High-Performance Data Compression, IEEE Computer, 17(6):8-19, June 1984.
[8] M. Salson, T. Lecroq, M. L´eonard, and L. Mouchard. Dynamic extended suffix arrays. Journal of Discrete Algorithms, In Press, Corrected Proof, 2009.
[9] M. Crochemore, L. Ilie, and W. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In DCC ’08: Proc. of the IEEE Conference on Data Compression, pages 482–488, 2008.