A New Compression Method for Compressed Matching

Student names: Avichai Paniri, Omer Kissos
Supervisor: Prof. Dana Shapira

Table of contents

Abstract

This project deals with lossless data compression and suggests a new compression method, especially suited to allow easy searching in the compressed files, even at the price of reducing compression efficiency.
Based on the article *"A New Compression Method for Compressed Matching*[1]*"*, we developed, implemented and tested the new method.
The implementation is in C language with Visual Studio 2010 IDE.
We based our application on the LZSS implementation of Michael Dipperstein[2].

Introduction

Data compression is the art of reducing the number of bits needed to store or transmit data. Compression can be either lossless or lossy. Lossless compressed data is decompressed to its exact original version, as opposed to Lossy compression, which discards "unimportant" data (usually used for image, audio, etc.).

LZSS (Lempel Ziv Storer Szymanski)[3] is a lossless data compression algorithm. It replaces a substring of symbols with a pointer to a previous occurrence. The pointer is represented as an ordered pair (Offset, Length), where

■ Offset: the number of characters from the current location to a previous occurrence of the matched substring.

■ Length: the length of the matched substring.

Following is an example:

Let T = "I meant what I said and I said what I meant".

Thus E (T) = "I meant what I said and (11, 7) (22, 6) (35, 7)" is the LZSS encoded file.

For the pointer, (11, 7), we need to go 11 steps back from the current location and copy the following 7 characters.

This project concentrates on LZSS, because it is one of the most popular compression algorithm.

The general approach for looking for a pattern in a file that is stored in its compressed form, is first decompressing and then applying one of the known pattern matching algorithms on the decoded file. In many cases, however, in particular on the Internet, files are stored in their original form, for if they were compressed, the host computer would have to provide memory space for each user in order to store the decoded file. This requirement is not reasonable, as many users can simultaneously quest the same information reservoir which will demand an astronomical quantity of free memory. Therefore there is a need to develop methods for searching directly within a compressed file.

When searching directly on LZSS compressed text, the problem is that when a character string is processed, we have no knowledge of whether it will be referenced later by means of a back-pointer. Therefore, we need to develop a new compression method especially suited to allow searching in the compressed file.

A New Compression Method

Idea:

The idea of the new algorithm is based on LZSS but with some changes:

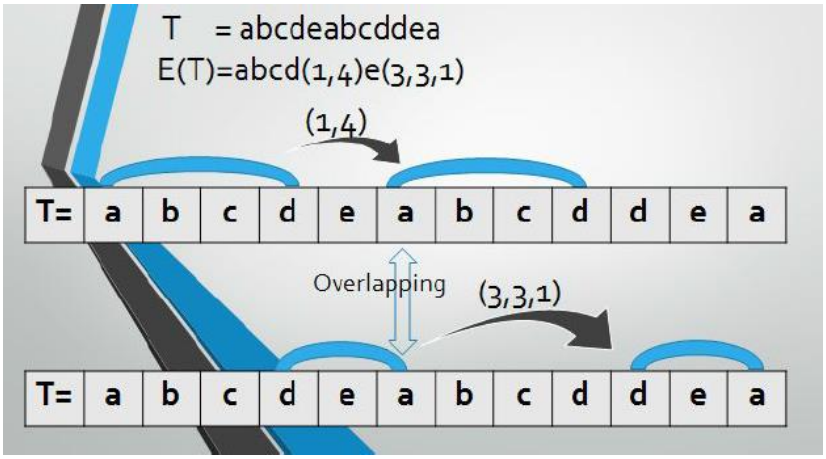
❑ the pointer points to the **next** occurrences of the matched substring instead of the previous occurrence.

❑ the pointer is composed out of 3 parameters instead of 2:

- Offset: the number of characters from the current location to the next occurrence of the matched substring.
- Length: the length of the matched substring.
- Slide: number of characters the pointer should be shifted forwards because of overlapping strings.

For example: Let T = "I meant what I said and I said what I meant", then the encoded form using the new method is E(T) = "I meant(28,7,0) what I(16,6,0) said(4,7,0) and". For the pointer, (28, 7, 0), means that the last 7 characters occur again 28 characters later.

Another example:



The new compression method ensures that whenever we meet a pointer item, we would immediately know if we must remember the characters of the string it refers to.

Project Components:

The project contains the next functions and parameters:

❏ **WINDOW_SIZE** – the size of the window, where the next occurrences are searched for.

❏ **MAX_CODED** - the maximum length of a substring the pointer can point to.

❏ **MAX_UNCODED** - the maximum length of a substring that will not be replace by a pointer.

❏ **EncodeLZSS** – This function reads an input file and writes an encoded output file according to the traditional LZSS algorithm using Brute force matching algorithm. The function doesn't accept "self-references".
*O (N * WINDOW_SIZE)*

❏**AddSlide** - the function gets a LZSS compressed file and adapts it to include a slide parameter in every pointer.
For every pointer, *it counts the sequence of overlapping characters.*
*O (N * WINDOW_SIZE)*

❏**EncodeProject** – the function gets a LZSS compressed file with a SLIDE parameter in all pointers, and adjusts it according to project format.
The function moves every pointer to the right place according to the OFFSET and SLIDE parameters.
*O (N * WINDOW_SIZE)*

❏ **EncodeProjectBack** – the function gets the encoded file according to the format of the project, and transforms it to the traditional LZSS encoded file.
The function moves every pointer to the right location according to the OFFSET and SLIDE parameters.
*O (N * WINDOW_SIZE)*

❏ **DecodeLZSS** - This function decodes a LZSS encoded file.
The function replaces every pointer with the substring the pointer points to.
*O (N * MAX_CODED)*

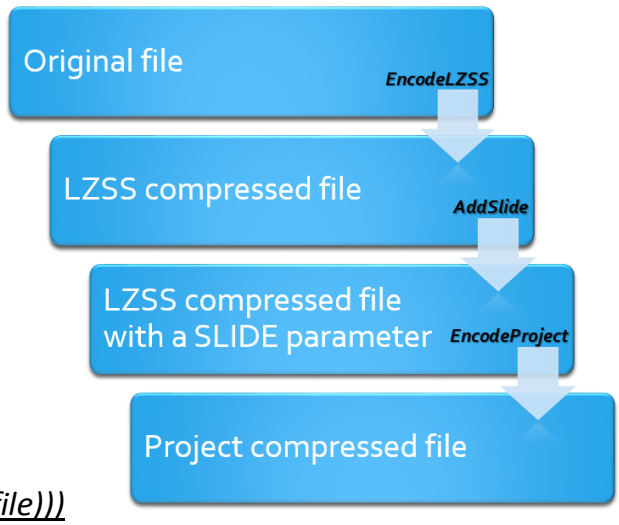
❏ **Diff** – The function gets two files, and checks whether the files are the same.
The function is used to verify whether the original file and the decompressed encoded file are the same (According to lossless compression requirements).
O (N)

Processes:

Compression:

In order to compress files according to the project's format we first run the function ***EncodeLZSS*** on the file.

EncodeLZSS (Original file)



Next, we run the function ***AddSlide*** on the result of the previous function.

AddSlide (EncodeLZSS (Original file))

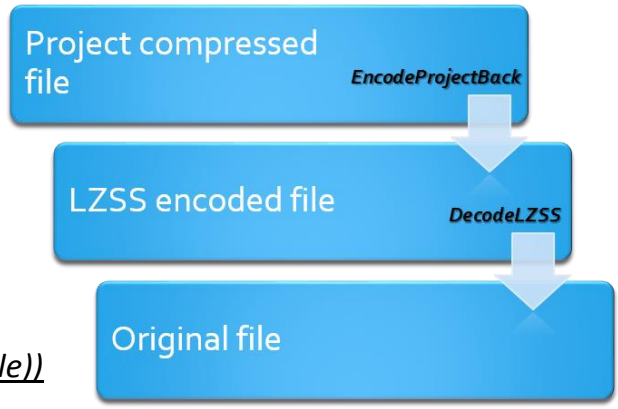
And finally, we run the function ***EncodeProject*** on the result of the previous function.

EncodeProject (AddSlide (EncodeLZSS (original file)))

Decompression:

In order to decompress the file encoded according to the project's format, we first run the function ***EncodeProjectBack*** on the compressed file.

EncodeProjectBack (compressed file)



We then run the function ***DecodeLZSS*** on the output of the previous function.

DecodeLZSS (EncodeProjectBack (compressed file))

It should be noted that for the application described above, compression is done only once, usually off-line, and if indeed we can search within the compressed text, decompression may never be necessary.

Comparison:

In order to verify that the algorithms work well we run the function **Diff** on a file in its original form and the file after decompressing its encoded format.

Diff (org' file, *DecodeLZSS (EncodeProjectBack (EncodeProject (AddSlide (EncodeLZSS (*org' file*)))))*

Improvement for Project Compression Method

Studies shows that the ideal OFFSET parameter is 12 bits and the LENGTH parameter is 4 bits.

The parameter WINDOW_SIZE is defined as 2^{OFFSET}. As bigger the OFFSET is, the bigger is the chance to find a better match, however, the cost of every pointer will grow too.

The MAX_CODED is defined by 2^{LENGTH}. The bigger the parameter LENGTH is, the longer the copied string becomes. However, shortening it will just partition the copy to several pointers.

The SLIDE parameter must be the same size as the OFFSET parameter because the slide is derived from the Buffer size, which is 2^{OFFSET}.

In our first version, the cost of every character is 1 bit (to sign if the symbol is a char or a pointer) + 8 bits, for a total of 9 bits.

The cost of each pointer is 1 bit (to sign if the symbol is a char or a pointer) + OFFSET + LENGTH + SLIDE), for a total of 29 bits.

The SLIDE parameter almost doubles the cost of the pointer and is not always needed.

Therefore, in our second version we added another type of pointer having only two coefficients:

■ Offset: the number of characters from the current location to the next occurrence of the matched substring.

■ Length: the length of the matched substring.

This pointer will be used when the SLIDE parameter is zero.

The cost of every character is 9 bits as before.

The cost of the pointer with 2 parameter is 1 bit (to sign if the symbol is a char or a pointer) + 1 (to sign if the symbol is a pointer with two or three parameters) + OFFSET + LENGTH), for a total of 18 bits.

The cost of the pointer with 3 parameter is 1 bit (to sign if the symbol is a char or a pointer) + 1 (to sign if the symbol is a pointer with two or three parameters) + OFFSET + LENGTH + SLIDE), for a total of 30 bits.

Experiment results

Name/size	Original (KB)	LZSS (KB)	Project V1 (KB)	Project V2 (KB)
Big	6337	3206	4690	4539
The bible	3027	1191	1803	1787
Lincoln Book	373	186	271	263
Pi- The first million digits of pi	977	616	941	874

Table 1: Comparative chart of compression performance

Parameter/ algorithm	LZSS (bit)	Project V1 (bit)	Project V2 (bit)
Offset	12	12	12
Length	4	4	4
Slide	---	12	12
MAX_UNCODED	2	3	3

Table 2: Parameters chart of the experiment tests

Table 1 gives the compression results. The second column gives the size (in KB) of the original (uncompressed) files. The next columns give the size of the compressed files (in KB).

Table 2 gives the compression parameters (in bits) in which the experiments were test with.

Conclusion

The new compression method is not meant to compete against others on the grounds of compression ratio or processing speed. It is an adaptation of the standard LZ method for compressed matching, and the use of forward pointers causes both encoding and decoding to be more involved and therefore much slower. We should however remember that for the application at hand here, compression is done only once, usually off-line, and if indeed we can search within the compressed text, decompression may never be necessary.

The findings indicates that there is a certain loss in compression efficiency when using the new algorithms V1/V2 instead of LZSS, which is mainly due to the encoding of the triples. Although the compression efficiency dropped while using the new compression method, the option for searching on the compress file is available.

Our experiments also indicates that there is an improvement in compression efficiency when using V2 as compared to V1, however it's not a major improvement as we thought at the beginning of the project.

References and related work

References:

1. S. Klein, D. Shapira, (2000) "*A New Compression Method for Compressed Matching*", Data Compression Conference – DCC-2000, 400-409
2. M. Dipperstein, "*LZSS (LZ77) Discussion and Implementation*", Version 0.7, http://www.michael.dipperstein.com/lzss/
3. J. A. Storer, T. G. Szymanski, (1982) "Data Compression via Textual Substitution"

Related work:

1. M. Takeda, A. Shinohara, (2015) "*Pattern Matching on Compressed Text*".
2. P. Gawrychowski,(2011) "Pattern Matching in Compressed Text"
3. P. Gawrychowski, (2011) optimal pattern matching in LZW compressed strings.
4. G. Navarro, M. Raffinot, (1999) "*A general practical approach to pattern matching over Ziv-Lempel compressed text*".
5. S. R. Kosaraju, (1995) "*Pattern matching in compressed texts*".
6. J. H. Morris, Jr. and V. R. Pratt, (1970) "*A linear pattern-matching algorithm*".