USING SUFFIX ARRAYS FOR LEMPEL-ZIV DATA COMPRESSION

USING SUFFIX ARRAYS FOR LEMPEL-ZIV DATA COMPRESSION

BY

ANISA AL-HAFIDH, B.Sc.

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
MCMASTER UNIVERSITY
HAMILTON, ONTARIO, CANADA
AUGUST, 2009

MCMASTER UNIVERSITY

Date: **August, 2009**

Author: **Anisa Al-Hafidh**

Title: **Using Suffix Arrays for Lempel-Ziv Data Compression**

Supervisor: **Dr. William F. Smyth**

Department: **Computing and Software**

Degree: **M.Sc.** Convocation: **Nov** Year: **2009**

Signature of Author

To Our Beloved Prophet
Muhammad
Peace Be Upon Him

Abstract

In the 1970s, Abraham Lempel and Jacob Ziv developed the first dictionary-based compression methods (**LZ77** and **LZ78**). Their ideas have been a wellspring of inspiration to many researchers, who generalized, improved and combined them with run-length encoding (**RLE**) and statistical methods to form many commonly used lossless compression methods for text, image and sounds.

The proposed methods factor a string $\boldsymbol{x}$ into substrings (factors) in such a way as to facilitate encoding the string into a compressed form (lossless text compression). This **LZ factorization**, as it is commonly called today, became a fundamental data structure in string processing, especially valuable for string compression. Recently, it found applications in computing various "regularities" in strings.

The main principle of LZ methods is to use a portion of the previously seen input string as the dictionary. **LZ77** and **LZ78** encoders differ in two aspects. The first aspect is that **LZ77** uses a sliding window unlike **LZ78** which uses the entire string for building the dictionary. The use of a sliding window in **LZ77** makes its decoder much simpler and faster than the **LZ78** decoder. This implies that **LZ77** is valuable in cases where a file is compressed once (or just a few times) and is decompressed often. A rarely used archive of compressed files is a superb example. The other aspect is the format of the codewords. **LZ77** codewords consist of three parts: position, length and first non-matching symbol, while **LZ78** removes the need for the length of the match in the codeword since it is implied in the dictionary.

A whole family of algorithms has stemmed out of the original LZ algorithms (**LZ77** and **LZ78**). This was a result of an effort to improve upon the LZ encoding algorithm

in terms of speed and compression ratio. Some of these variants involved the use of sophisticated data structures (e.g. suffix trees, binary search trees, etc) to hold the dictionary in order to boost the search time. The problem with such data structures is that the amount of memory required is variable and cannot be known in advance. Furthermore, some of these data structures require a substantial amount of memory. LZ is the basis of the `gzip` (Unix), `winzip` and `pkzip` compression techniques.

In the testing for [1], we scaled up an **LZSS** implementation due to Haruhiko Okumura [37] so as to be useful for regularities ($\boldsymbol{N} = \boldsymbol{n}$, the length of the whole input string, and $\boldsymbol{F}$ equal to the full length of the unfactored suffix). We found that the binary tree approach becomes uncompetitive with algorithms that use the suffix array (SA) approach for the LZ factorization of the whole string. This observation triggered us to scale down the SA approach.

The main contribution of this thesis is a novel **LZ77** variant (`LZAS`) that uses a suffix array (SA) to perform the searches. The SA is augmented with a very simple and efficient algorithm that alternates between searching left and right in SA to find the longest match. Suffix arrays have gained the attention of researchers in recent years due to their simplicity and low memory requirements. They solve the sub-string problem as efficiently as suffix trees, using less memory. One notable advantage of using SA in an LZ encoder is that the amount of memory is independent of the text to be searched and can be defined a priori. The low and predictable memory requirement of this approach makes it suitable for memory-critical applications such as embedded systems. Moreover, our experiments show that the processing time per letter is almost stable and hence we can predict the processing time for a file given its size. Our proposed algorithm can additionally be used for forward/backward sub-string search.

In this thesis we investigate three variants of the `LZAS` algorithm. The first two of these variants (i.e. `LZAS1` and `LZAS2`) use a dynamic suffix array `DSA`. `DSA` is a suffix array that can be updated whenever a letter or a factor is edited (i.e. deleted/inserted/substituted by another letter or factor) in the original string. The

suffix array of a sliding window changes whenever the window slides. Hence, we use the `DSA` to make sure that the suffix array is up to date. The DSA can be compressed using a sampling technique; therefore we decided to experiment with both sampled and non-sampled DSA. The third variant (i.e. `LZAS3`) re-computes the suffix array instead of updating it. We use an implementation of a suffix array construction algorithm (`SACA`) that requires supralinear time [28] but performs well in practice. We tested these variants against each other in terms of time and space. We further experimented with various window sizes and noticed that re-computing SA becomes better than updating it using DSA when the window size is small (i.e. hundreds of bytes compared to thousands of bytes).

Acknowledgements

He who does not thank people is not grateful to God.

Prophet Muhammad (pbuh)

Most of all, I would like to thank my supervisor Bill Smyth. Thanks for the opportunity to become your student and get the chance to join McMaster. Thanks for your patience, support and guidance. Thanks for your kind words when I needed them and your criticisms when I needed them. Thanks for your encouraging words and reminding me of what I am capable of. Without you this thesis won't be here today.

I would like also to thank my thesis committee Mark Lawford and Jeffery Zucker. Thanks for accepting to serve on the committee on such a short notice and for your helpful comments. You were such a great committee.

I wish to extend my gratitude to everyone who helped me with providing information and answering my questions. Thanks to Mikaël Salson and Laurent Mouchard for providing me with their papers and code even before the papers were published. Thanks for always taking the time to answer my questions and discuss your work. I would like also to thank Simon Puglisi for taking the time to answer my questions about suffix arrays and their construction.

I must acknowledge as well my colleagues at McMaster. Shu Wang and Munira Yusufu, thanks for answering my questions about your experiences in string algorithms and providing me with information when I need it. Evgenia Kopylov, thanks for answering my questions about your testing and experience with Okuruma's code for LZSS. Hadeel Al-Dauod and Tahani Al-Mabruk, thanks for your sisterhood that helped me through periods of homesickness, you have been such a great companion.

I would like also to thank my best friend Fatima Al-Raisi for sharing her past experience through her Masters study in US and replying my emails when I needed her support and advice through tough times. Fatima, I can not imagine how my life would be without you, you are such a great friend.

I will always be indebted to my family, without them I wouldn't be where I am today. Father, thanks for believing in me and supporting me emotionally and financially; I assure you, you will never regret it. Mother, thanks for being patient and providing me with strength while we are miles away. My brothers Ali and Omar, thanks for your company and for sacrificing your time to come with me to Canada to pursue my dream. I will always be indebted to you and won't be able to pay you back whatever I do.

Finally, all praise and gratitude is due to God the Exalted.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

LZ77 and LZ78 compression [47, 48] started a novel category of compression methods: dictionary-based compression. The ideas of LZ77 and LZ78 have been a wellspring of inspiration to many researchers. This led to whole a family of variants that stemmed out of the original LZ algorithms (LZ77 and LZ78).

The proposed methods factor a string $\boldsymbol{x}$ into substrings (factors) in such a way as to facilitate encoding the string into a compressed form (lossless text compression). This LZ factorization, as it is commonly called today, became a fundamental data structure in string processing, especially valuable for string compression.

The main principle of LZ methods is to use a portion of the previously seen input string as the dictionary. We will describe briefly this principle in the context of a sliding window which is used in LZ77 but not in LZ78. For more details on LZ77 and LZ78 see Chapter 3.

LZ77 operates not on the string as a whole, but only on a sliding window of length

N. The window is divided into two parts (see Fig. 1.1):

1. A search buffer which contains input that has already been encoded.

2. A lookahead buffer of length F, an as-yet-unencoded suffix.

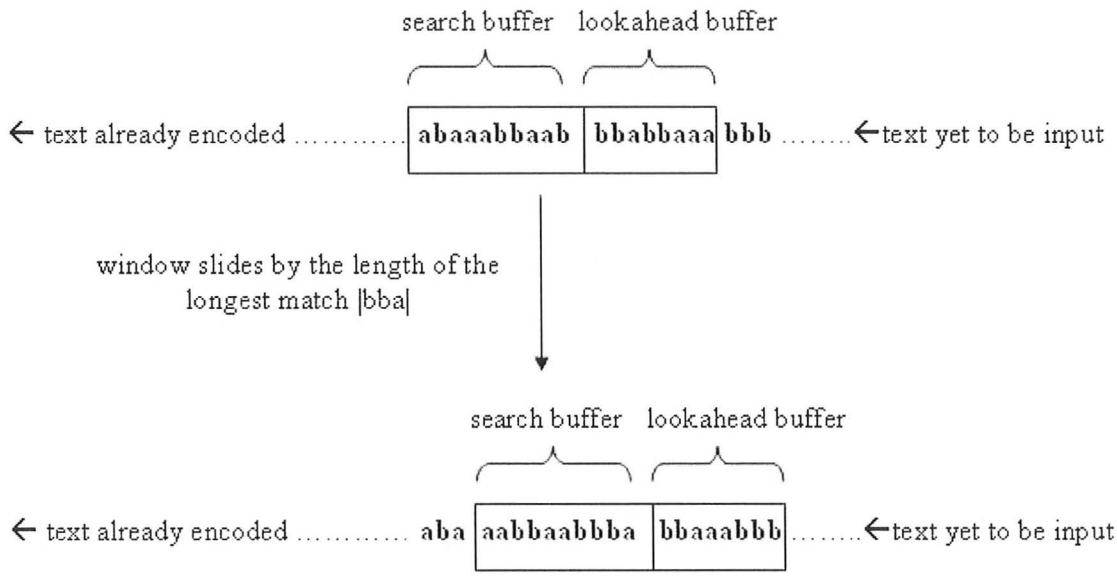

Figure 1.1: Example to illustrate the sliding window in **LZ77**

The encoder maintains the window and shifts it from left to right as strings of symbols are being encoded. It scans the search buffer looking for the longest match to a prefix of the lookahead buffer. Once a match is determined, the encoder outputs a codeword and shifts the window to the right by the length of the match. In practical implementations the search buffer is usually some thousands of bytes long, while the

lookahead buffer is only tens of bytes long. It has been found that in practice the use of the sliding window provides compression as good as using the entire string would yield, and of course processing time is substantially reduced.

LZ77 has many variants that improved upon the original version. LZSS is the first and most prominent amongst them. It was developed by Storer and Szymanski in 1982 [43]. Bell [3] improved LZSS by using a binary search tree to hold the dictionary. Another improvement comes from the observation that using a large tree with $\boldsymbol{N - F}$ strings in it would lead to an expected tree height of $\boldsymbol{log(N)}$, hence $\boldsymbol{log(N)}$ comparisons, assuming the tree was balanced. A direct way to reduce that maximum height is to use 256 different trees, one for every possible initial character. Some implementations use $\boldsymbol{N}$ trees, choosing them by hashing on the first three characters. Hence, one can decide which trees to search using the first three characters of the lookahead buffer and the hashing function.

In the testing for [1], we scaled up an **LZSS** implementation due to Haruhiko Okumura so as to be useful for regularities ($\boldsymbol{N = n}$, the length of the whole input string, and $\boldsymbol{F}$ equal to the full length of the unfactored suffix). We found that the binary tree approach becomes uncompetitive with algorithms that use the suffix array (SA) approach for the LZ factorization of the whole string. This observation triggered us to scale down the SA approach.

The key idea is to replace the binary search tree (or any other type of tree) by

the suffix array. The search is done using a simple elegant algorithm that performs a left/right search of the suffix array to obtain the longest match (for the lookahead buffer in the search buffer). Another advantage of such a method is that the memory is fixed and independent of the string length.

1.1 Background

Data Compression algorithms exploit characteristics such as repeating substrings (patterns) to make the compressed data smaller than the original data. Lossless compression algorithms — as opposed to "lossy" compression algorithms — ensure that the original information can be accurately reproduced from the compressed data.

Well-known lossless compression techniques include:

- Run-length encoding (RLE) which basically replaces n consecutive occurrences of item d by the single pair nd,
- statistical techniques such as Huffman coding and predication by partial matching (PPM), and
- dictionary coders.

The Lempel-Ziv algorithms belong to the last category.

Dictionary coding techniques rely upon the observation that there are correlations between parts of data (recurring patterns). The basic idea is to replace those

repetitions by (shorter) references to a "dictionary". The dictionary holds strings of symbols from the original data, and it can be static or dynamic (adaptive). The former is permanent, sometimes allowing the addition but no deletions, whereas the latter holds strings previously found in the input stream allowing for additions and deletions of strings as new input is being read. Thus, we can divide the dictionary coders further according to the nature of the dictionary in the following categories:

- static dictionary coders,
- semi-adaptive dictionary coders, and
- adaptive dictionary coders.

Most of the Lempel-Ziv algorithms belong to the third of the above categories. The dictionary is built in a single pass, while at the same time the data is encoded. It is not necessary to explicitly transmit/store the dictionary because the decoder can build up the dictionary in a way similar to the encoder.

In general, an adaptive dictionary-based method is preferable. It can start with an empty dictionary or with a small, default dictionary: add entries to it as they are found in the input stream, and delete old entries since a large dictionary means slow search. We can visualize an adaptive dictionary-based method as a loop where each iteration:

- starts by reading the input stream, and

- breaks it up (parses it) into words or phrases,
- then searches the dictionary for each word or phrase;
- if a match is found, writes a token in the output stream.
- Otherwise, the uncompressed word should be written and also added to the dictionary.
- Finally, it checks to see whether an old entry should be deleted from the dictionary.

This may seem complicated, but it has two important features that differentiate it from other compression techniques:

1. It involves string search and match operations, rather than numerical computations.
2. The decoder is simple. In statistical compression methods, the decoder is the exact opposite of the encoder (symmetric compression). In an adaptive dictionary-based method, however, the decoder has to read its input stream, determine whether the current item is a token or uncompressed data, use tokens to acquire data from the dictionary, and output the final, uncompressed data. It does not have to parse the input stream in a complex way, and it does not have to search the dictionary to find matches.

1.2 Applications

LZ factorization has recently found application in the computation of various "regularities" in strings: repetitions [6], runs (maximal periodicities) [20, 2, 10, 11, 7, 8], repeats with fixed gap [21], branching repeats [42], sequence alignments [9], and local periods [15]. For these applications, the LZ factorization of the entire string, not merely a window, is required. However, we are interested here in the original application of LZ in data compression.

We seem to be preprogrammed with the idea of sending as little data as we can to save time. We normally tend to accumulate data and hate to throw anything away. Over time, this can lead to an overflow no matter how large our storage device is. Time is also an issue for us. We hate to wait for a large file to download or a web page to be displayed on our screen; anything longer than a few seconds is usually a long time to wait. Hence, compression is useful as it reduces the consumption of expensive resources such as hard disk space or transmission time.

Data compression is the process of encoding data using fewer bits than the actual unencoded data. A compression scheme is either lossless or lossy. Lossless compression schemes— as opposed to "lossy" compression schemes— ensure that the original information can be accurately reproduced from the compressed data. LZ compression is a lossless compression scheme. Therefore, in what follows we will mention applications related to lossless compression.

Optimizing disk space or bandwidth of a network is usually done with lossless data compression. Also, in compressing files that contain symbols such as spreadsheets, texts, executable program, etc., losslessness is important as changing even a single bit cannot be tolerated.

Modern modems contain hardware that automatically compresses data as they send it. If the data is already compressed, there will not be any farther compression. As there might be expansion sometimes, the modem should be able to monitor the compression ratio "on the fly" and, if it is low, it should stop compressing and send the rest of the data uncompressed. An example of this technique is V.42bis protocol. V.42bis protocol uses the `LZW` variant (a famous `LZ78` variant) when operating in "compressed" mode.

Most graphics file formats use some kind of compression. GIF (the graphics interchange format) was developed by *Compuserve Information Services* in 1987 as an efficient, compressed graphics file format, which allows for images to be sent between different computers. It uses a variant of `LZW` to compress the graphics data.

1.3 The New Algorithms

As we mentioned before, the experiment of scaling up an implementation of `LZSS`, due to Haruhiko Okumura [37], triggered the idea of scaling down the SA-based LZ algorithms. Okumura implementation's uses 256 binary search trees to achieve fast searches. Most **LZ77** variants use some sort of trees (i.e. suffix trees, binary search

trees, tries, etc) to speed up the encoding process.

We designed a novel **LZ77** variant that makes use of the suffix array to perform the search in the dictionary. The main idea is to replace the binary search tree (or any other kind of tree) by the suffix array. The search is done using a simple elegant algorithm that performs a left/right search of the suffix array to obtain the longest match in the search buffer for a prefix of the look-ahead buffer. Another advantage of such a method is that the memory is fixed and independent of the string length.

Notice that the suffix array needs to be updated wherever the window slides. In this thesis we experiment with two solutions:

1. Use a dynamic suffix array **DSA** [27] which can be updated whenever characters are added, deleted or edited in the original string. **DSA** can be used in its compact form (i.e. sampled SA) or non-compact (i.e. non-sampled SA) form. We experiment with both forms.

2. Recompute the suffix array whenever the window slides. For testing purposes, we use a supralinear suffix array construction algorithm **SACA** but efficient according to [28].

These two solutions give rise to three variants of the `LZAS` algorithm. The first two of these variants (i.e. `LZAS1` and `LZAS2`) use a dynamic suffix array `DSA`. The suffix array of a sliding window changes whenever the window slides. Hence, we use the DSA to make sure that the suffix array is up to date. The DSA can be compressed

using a sampling technique. We decided to experiment with both sampled and non-sampled DSA. The third variant (i.e. `LZAS3`) re-computes the suffix array instead of updating it. We tested these variants against each other in terms of time and space. We further experimented with various window sizes and noticed that recomputing SA becomes better than updating it using DSA when the window size is small (i.e. hundreds of bytes compared to thousands of bytes).

We compared our results with Okumura's implementation of `LZSS`. It turns out that our approach becomes uncompetitive to Okumura implementation. This might be related to the fact that the binary search trees are more efficient for small strings and hence work well for a sliding window approach; while suffix arrays are more efficient for long strings and hence work efficiently for a whole string. When scaling up the window to accommodate the whole string, the binary search trees grow bigger and become inefficient. The other reasoning is that we need to update the suffix array whenever the window slides and if the update was not efficient enough then this will slow the whole algorithm as this is the most time consuming part of our algorithm. DSA was designed for a general update in mind i.e. adding/deleting /substituting a letter or a factor in the original string, regardless of the position. We think it is possible to develop an algorithm that updates the suffix array more efficiently for cases of deleting a factor (prefix) at the beginning of the string and adding a factor (suffix) at the end of the string.

1.4 Thesis Outline

The reminder of this thesis consists of the following chapters. Chapter 2 prepares the background required to make the reading process efficient. It provides the required definitions and notation that are used throughout the thesis. Furthermore, data structures that are used in our algorithms are described in this preliminary chapter. We then give a very short chapter, Chapter 3, on LZ77 and LZ78 to enable the reader to distinguish between the two schemes. Chapter 4 introduces our novel LZ77 variant algorithm for data compression. We describe the algorithm in detail and its variants. Moreover, we state an interesting lemma that we discovered while investigating the process of updating the SA for a particular case (deleting a prefix and adding a suffix to the original string). This observation was not considered in the algorithm of DSA. We think this lemma can be used to develop a more efficient algorithm for updating the suffix array for this particular case which would be useful for our context (i.e. sliding window). Chapter 5 presents the results of experiments that compare the algorithms against each other and against an existing LZ77 variant (LZSS). Finally, Chapter 6 provides some concluding remarks and suggestions for improving this work.

Chapter 2

Preliminaries

2.1 Basic Definitions

In this section we give definitions and notation required for the reader to comprehend the material presented in this thesis. Most of these definitions come from [41].

2.1.1 Alphabets and Strings

This thesis deals with linear strings. A **linear string** is a finite sequence of characters called **letters** which are elements of a nonempty finite set Σ called an **alphabet**. We use σ to denote the size of the alphabet $|\Sigma|$. A string of length n is denoted by $\boldsymbol{x} = \boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]\ldots\boldsymbol{x}[n]$, where $\boldsymbol{x}[i]$ represents a character at position i for $i \in 1..n$.

We use $\boldsymbol{x}[i..j]$ to denote a substring of $\boldsymbol{x}$ starting at position i of length $j-i+1$, where $1 \leq i \leq j \leq n$, i.e. $\boldsymbol{x}[i..j] = \boldsymbol{x}[i]\boldsymbol{x}[i+1]\ldots\boldsymbol{x}[j]$. If $j < i$, then $\boldsymbol{x}[i..j] = \boldsymbol{\epsilon}$, the empty string. Using this notation we can write a **prefix** of $\boldsymbol{x}$ starting at position $j \in 1..n$, as $\boldsymbol{x}[1..j]$. A prefix is said to be a **proper prefix** when $j < n$. Likewise,

$\boldsymbol{x}[i..n]$ denotes a **suffix** of $\boldsymbol{x}$ starting at position $i \in 1..n+1$ and when $i > 1$ it is called a **proper suffix**. For example, the prefixes and suffixes of *abaaabba* are shown in Figure 2.1. Proper prefixes and suffixes are produced by simply excluding the string itself from the previous lists.

When a proper prefix $\boldsymbol{x}[1..j]$ and a proper suffix $\boldsymbol{x}[i..n]$ of a string $\boldsymbol{x}$ are equal so that $i = n - j + 1$, we say that $\boldsymbol{x}$ has a **border** $\boldsymbol{b} = \boldsymbol{x}[1..j] = \boldsymbol{x}[i..n]$ of length j. From Figure 2.1, we see that string $\boldsymbol{x} = abaabaab$ has two nonempty borders: *ab* and *abaab*. Observe that the longest one, *abaab*, overlaps with itself.

prefixes	ϵ, *a*, *ab*, *aba*, *abaa*, *abaab*, *abaaba*, *abaabaa*, *abaabaab*
suffixes	ϵ, *b*, *ab*, *aab*, *baab*, *abaab*, *aabaab*, *baabaab*, *abaabaab*

Figure 2.1: Prefixes and suffixes of $\boldsymbol{x} = abaabaab$

We require Σ to be an ordered set; hence we can define a **lexicographic order** on strings of Σ. Suppose we are given two strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..m]$, where $n \geq 0$ and $m \geq 0$. We say that $\boldsymbol{x} < \boldsymbol{y}$ ($\boldsymbol{x}$ is lexicographically less than $\boldsymbol{y}$) if and only if one of the following (mutually exclusive) conditions hold:

- $n < m$ and $\boldsymbol{x}[1..n] = \boldsymbol{y}[1..n]$ (i.e. $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$);
- $\boldsymbol{x}[1..i-1] = \boldsymbol{y}[1..i-1]$ and $\boldsymbol{x}[i] < \boldsymbol{y}[i]$ for some integer $i \in 1..\min n, m$ (this is the case in which there is a first position i in which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ).

When needed, we use \$ as a sentinel letter in position $n + 1$, that is not equal to

any other letter in $\boldsymbol{x}$ and that is lexicograhically less than all letters in Σ.

A **cyclic shift** (or **rotation**) of $\boldsymbol{x}$ is defined to be the string $\boldsymbol{x}^{[j]} = \boldsymbol{x}[j+1..n]\boldsymbol{x}[1..j]$ for every integer $j \in 0..n-1$. Thus $\boldsymbol{x}^{[0]} = \boldsymbol{x}$ and the eight cyclic shifts of our example $\boldsymbol{x} = abaabaab$ are as follows:

$$\begin{aligned}
\boldsymbol{x}^{[0]} &= abaabaab \\
\boldsymbol{x}^{[1]} &= baabaaba \\
\boldsymbol{x}^{[2]} &= aabaabab \\
\boldsymbol{x}^{[3]} &= abaababa \\
\boldsymbol{x}^{[4]} &= baababaa \\
\boldsymbol{x}^{[5]} &= aababaab \\
\boldsymbol{x}^{[6]} &= ababaaba \\
\boldsymbol{x}^{[7]} &= babaabaa
\end{aligned} \tag{2.1.1}$$

Writing $\boldsymbol{x} = \boldsymbol{w_1 w_2 \ldots w_k}$ where the $\boldsymbol{w_i}$ are nonempty substrings, $i \in 1..k$, is called a **factorization** or **decomposition** of $\boldsymbol{x}$ into **factors** $\boldsymbol{w_i}$. Thus a factor is just a nonempty substring.

2.1.2 Lempel-Ziv Factorization LZ

For an **LZ** factorization of $\boldsymbol{x}$, we use the following definition:

Definition 2.1.1. A factorization of $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ is $\boldsymbol{LZ}$ if and only if each $\boldsymbol{w_j}$, $j \in 1..k$, is

(a) a letter that does *not* occur in $\boldsymbol{w_1 w_2 \cdots w_{j-1}}$; or otherwise

(b) the longest substring that occurs at least twice in $\boldsymbol{w_1 w_2 \cdots w_j}$.

We observe that $\boldsymbol{w_1} = \boldsymbol{x}[1]$, further that a factor $\boldsymbol{w_j}$ may overlap with its previous occurrence in $\boldsymbol{x}$. For the string

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x} = & a & b & a & a & b & a & a & b, \end{array} \tag{2.1.2}$$

$\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = a$, $\boldsymbol{w_4} = abaab$.

For most of the last 30 years, LZ factorization has been used primarily for text compression, and many LZ variants have been proposed and computed, including factorization of infinite words [4]. Useful surveys are available at [14, 36, 46]. In the context of compression, LZ algorithms generally operate not on the string as a whole, but only on a sliding window. Many sliding-window algorithms have been proposed, of which several are described in [43, 3, 38] and the surveys noted above. See Chapter 3 for more details on the original LZ compression algorithms: **LZ77** and **LZ78**.

2.2 Fundamental Data Structures

In this section we define the fundamental data structures that are the basis of our algorithm and are important for almost all string algorithms.

2.2.1 Suffix Array SA

Consider a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ of length n over an ordered alphabet $\boldsymbol{\Sigma}$. As mentioned previously, the suffix of $\boldsymbol{x}$ starting at position i is denoted by $\boldsymbol{x}[i..n]$, for $1 \leq i \leq n$ (here we are not interested in the suffix ϵ, hence $i \neq n+1$). To simplify the notation, let us use the expression suffix i to denote the suffix $\boldsymbol{x}[i..n]$. Then, the **suffix array** of $\boldsymbol{x}$, denoted **SA**, gives the suffixes of $\boldsymbol{x}$ sorted in ascending lexicographical order, that is:

$$SA[1] < SA[2] < ... < SA[n].$$

The suffix array of the string *abaabaab* is shown in the second column of Figure 2.2.

i	**SA**[i]	$\boldsymbol{x}$[SA[i]..n]	**ISA**[i]	**LCP**[i]
1	6	*aab*	5	-1
2	3	*aabaab*	8	3
3	7	*ab*	2	1
4	4	*abaab*	4	2
5	1	*abaabaab*	7	5
6	8	*b*	1	0
7	5	*baab*	3	1
8	2	*baabaab*	6	4

Figure 2.2: SA and LCP arrays of $\boldsymbol{x} = abaabaab$

SA can be computed in $\Theta(n)$ worst-case time [25, 22], though various supralinear methods [32, 33] are certainly much faster, as well as more space-efficient, in practice [40], in some cases requiring space only for $\boldsymbol{x}$ and SA itself.

In [2] an Enhanced Suffix Array (ESA) is introduced, consisting of the suffix array together with an "lcp-interval tree". Recently, a Dynamic ESA (DSA) was introduced in [27]. They presented an algorithm that updates ESA/SA when the text is edited (insertion, deletion or substitution of a letter or a factor). We discuss the DSA algorithm briefly in 2.3.

2.2.2 Inverse Suffix Array (ISA)

The **inverse suffix array** (**ISA**) gives for each suffix $\boldsymbol{x}[i..n]$, its lexicographical order among other suffixes. That is, $\text{ISA}[i] = j$ iff $\text{SA}[j] = i$. Many algorithms use inverse suffix array to build suffix arrays in linear time. Reversibly, an inverse suffix array can be turned into a suffix array in place in linear time, too. The fourth column in Figure 2.2 represents the ISA for our string example.

2.2.3 Longest Common Prefix (LCP) Array

Another important data structure that is usually used with suffix arrays is the Longest Common Prefix (LCP) array. Let us denote the length of the longest common prefix of suffixes $\text{SA}[i]$ and $\text{SA}[j]$ by $\text{lcp}(\text{SA}[i], \text{SA}[j])$. Then, the **LCP** array contains the lengths of the longest common prefixes between successive suffixes of SA. That is,

for $1 < i \leq n$, $\mathrm{LCP}[i] = \mathrm{lcp}(\mathrm{SA}[i-1], \mathrm{SA}[i])$ and $LCP[1] = -1$ since it is otherwise undefined. An important property of lcp [25] is that for any $1 \leq i < j \leq n$:

$$lcp(SA[i], SA[j]) = \min_{i<k\leq j} LCP[k].$$

Given $\boldsymbol{x}$ and SA, LCP can also be computed in $\Theta(n)$ time [23, 31, 39, 24]: the first algorithm described in [31] requires $9n$ words of storage and is almost as fast in practice as that of [23], which requires $13n$ words. However the algorithm described in [39] is generally faster and requires about $6n$ words of storage for its execution, since it overwrites the suffix array. The very recent LCP algorithm is the one proposed in [24], that first computes a "permuted" LCP array; it executes consistently faster than all other LCP algorithms, but uses $13n$ bytes.

The fifth column of Figure 2.2 gives the LCP array of the string *abaabaab*.

2.2.4 Burrows-Wheeler Transform BWT

The BWT and its calculations are important for the methods used in this thesis. We use a dynamic SA (DSA) [27] that can be updated whenever the window slides. DSA needs BWT in order to maintain the suffix array after an insertion or deletion.

Formally, we define the Burrows-Wheeler Transform **BWT** of $\boldsymbol{x}$ [5] as follows:

Definition 2.2.1. For a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\mathrm{BWT}[i] = \boldsymbol{x}[\mathrm{SA}[i]-1]$ for $\mathrm{SA}[i] > 1$; otherwise, $\mathrm{BWT}[i] = \boldsymbol{x}[n]$.

Usually, when using the BWT, it is convenient to suppose that the sentinel letter \$ has been appended to $\boldsymbol{x}$, yielding $\boldsymbol{x}[1..n+1] = \boldsymbol{x}\$$. Since \$ is the least letter, this

means that the least cyclic shift (the lexicographically least suffix of $\boldsymbol{x}$) is $\$\boldsymbol{x}$ and occurs as the first row in the conceptual matrix M (see Fig. 2.3).

The BWT is equal to the text corresponding to the last column L of the conceptual matrix M whose rows are the lexicographically sorted cyclic shifts of the string $\boldsymbol{x}\$$ (see Fig. 2.3). There is a strong apparent relation between the matrix M and the suffix array SA of the string $\boldsymbol{x}\$$. When sorting the rows of the matrix M we are essentially sorting the suffixes of $\boldsymbol{x}\$$. Consequently, $SA[i]$ points to the suffix of $\boldsymbol{x}\$$ occupying (a prefix of) the ith row of M. Hence, the cost of constructing the BWT is given by the cost of constructing the suffix array, and this requires $O(n)$ time.

$\boldsymbol{i}$	$\boldsymbol{F}$								$\boldsymbol{L}$
9	\$	a	b	a	a	b	a	a	b
6	a	a	b	\$	a	b	a	a	b
3	a	a	b	a	a	b	\$	a	b
7	a	b	\$	a	b	a	a	b	a
4	a	b	a	a	b	\$	a	b	a
1	a	b	a	a	b	a	a	b	\$
8	b	\$	a	b	a	a	b	a	a
5	b	a	a	b	\$	a	b	a	a
2	b	a	a	b	a	a	b	\$	a

Figure 2.3: The conceptual matrix M and BWT=$bbbaa\$aaa$ (column L)

The Burrows-Wheeler transform (BWT) [5] sorts the letters of a text $\boldsymbol{x}$ to facilitate its compression. It is used as a preprocessor by some famous lossless text compression tools (such as bzip) that incoperate it with Run-length Encoding or Prediction by Partial Matching (PPM) methods [12, 13]. Due to its structure and its similarity with the suffix array, it has been used a lot for advanced compressed index structures

[17, 18, 27] that compute approximate pattern matching, which make them useful for search engines.

Salson et. al.[26] studied the impact of edit operations (insertion/deletion/substitution of a letter or a factor) on BWT($\boldsymbol{x}$). Moreover, they presented a four-stage algorithm for updating BWT($\boldsymbol{x}$). We explain this algorithm briefly in section 2.3.2.

2.3 Algorithms

[27] presented an algorithm that modifies the SA and the LCP arrays based on changes to the text (i.e. insertion/deletion/substitution of a letter or a factor). This algorithm is based on a four-stage algorithm described in [26] that updates BWT.

We will briefly discuss in this section the updating process of BWT and how the authors extended it to update the SA. The reader who is interested in the details of updating the LCP can refer to [27]. To ease the reading process, we will denote the SA updating algorithm by **DSA** (an acronym for dynamic suffix array).

2.3.1 Preliminaries

Recall that F and L are respectively the first and last columns of the conceptual matrix M (see Fig. 2.3). Notice that F is sorted and hence can be deduced from L. If we want to add/delete a character from the original string $\boldsymbol{x}$, we will need to reconstruct $\boldsymbol{x}$ from L. That can be done by navigation through the characters of L according to their positions in $\boldsymbol{x}$. In order to navigate through L, we need a function

that tells us how the rows are ranked. This function is called $\boldsymbol{LF}$ and maps a letter in L to its equivalent in F; for example, the unique \$ in L is mapped to the unique \$ in F, the first a in L is mapped to the first a in F, etc (see Fig. 2.4). That is, we want to map corresponding letters in F and L.

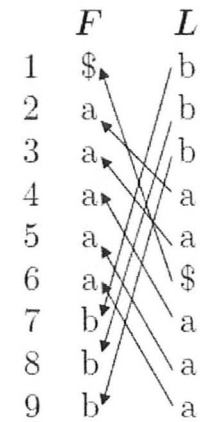

Figure 2.4: LF function

To understand how we can compute LF, we will examine the relationship between L and F closely. First, we consider the first row of the conceptual matrix M: it corresponds to $\boldsymbol{x}^{[n]}$. This row necessarily contains \$ in F and $\boldsymbol{x}[n]$ in L (since \$ is the smallest letter and F is sorted). Consequently, the row that contains the letter $\boldsymbol{x}[n]$ in L has to be mapped with the row corresponding to the cyclic shift starting with $\boldsymbol{x}[n]\$$, that is the row where $\boldsymbol{x}[n]$ appears for the first time in F. Hence, we need a simple mechanism for navigating from the row corresponding to $\boldsymbol{x}^{[i+1]}$ to $\boldsymbol{x}^{[i]}$. By definition, if the row $\boldsymbol{x}^{[i+1]}$ has a letter c in L, then the row $\boldsymbol{x}^{[i]}$ has also the same letter c in F.

Remark 2.3.1. $\boldsymbol{x}[i+1] = \boldsymbol{x}^{[i \bmod |\boldsymbol{x}|]}[n+1]$, *for* $0 \leq i \leq n$.

From the previous remark, we know that if p is the position of $\boldsymbol{x}^{[i]}$ in the sorted cyclic shifts, then $\boldsymbol{x}[i+1] = L[p]$. In order to map corresponding letters in L and F, we therefore need a function $rank_{\boldsymbol{x}}(c, i)$ that returns the number of c in $\boldsymbol{x}[1..i]$, for any string $\boldsymbol{x}$ over Σ. Now given two positions p and p' such that $F[p'] = L[p] = c$, we are connecting them if and only if $rank_F(c, p') = rank_L(c, p)$.

We also need a table C storing, for each letter c of the alphabet, the number $C[c]$ of characters smaller than $c = \boldsymbol{x}[i]$ in $\boldsymbol{x}[1..n]$. Since letters are lexicographically sorted in F, the number of letters smaller than c is one less than the position at which c appears for the first time in F.

Finally, using the rank function and the count table, the LF function which permits us to compute the position of a cyclic shift $\boldsymbol{x}^{[i]}$ from the position of the following cyclic shift $\boldsymbol{x}^{[i+1]}$, can be computed as follows:

$$LF(i) = rank_{\boldsymbol{x}}(L[p], i) + C[L[p]]. \tag{2.3.1}$$

2.3.2 Updating BWT

As we mentioned before, DSA is based on an algorithm that updates the BWT. This was due to the similarities between BWT and SA (see section 2.2.4). As defined previously, the BWT is the text of length $n+1$ equal to the last column L of the

conceptual matrix M whose rows are the lexicographically sorted cyclic shifts of $\boldsymbol{x}\$$ (see Fig.2.5(a)).

	$\boldsymbol{F}$								$\boldsymbol{L}$
$x^{[8]}$	\$	a	b	a	a	b	a	a	b
$x^{[5]}$	a	a	b	\$	a	b	a	a	b
$x^{[2]}$	a	a	b	a	a	b	\$	a	b
$x^{[6]}$	a	b	\$	a	b	a	a	b	a
$x^{[3]}$	a	b	a	a	b	\$	a	b	a
$x^{[0]}$	a	b	a	a	b	a	a	b	\$
$x^{[7]}$	b	\$	a	b	a	a	b	a	a
$x^{[4]}$	b	a	a	b	\$	a	b	a	a
$x^{[1]}$	b	a	a	b	a	a	b	\$	a

(a)The conceptual matrix M of $\boldsymbol{x}\$ = abaabaab\$$

$\longrightarrow$

	$\boldsymbol{F}$								$\boldsymbol{L}$
$x'^{[9]}$	\$	a	b	a	a	c	b	a	a	b
$x'^{[6]}$	a	a	b	\$	a	b	a	a	c	b
$x'^{[2]}$	a	a	c	b	a	a	b	\$	a	b
$x'^{[7]}$	a	b	\$	a	b	a	a	c	b	a
$x'^{[0]}$	a	b	a	a	c	b	a	a	b	\$
$x'^{[3]}$	a	c	b	a	a	b	\$	a	b	a
$x'^{[8]}$	b	\$	a	b	a	a	c	b	a	a
$x'^{[5]}$	b	a	a	b	\$	a	b	a	a	c
$x'^{[1]}$	b	a	a	c	b	a	a	b	\$	a
$x'^{[4]}$	c	b	a	a	b	\$	a	b	a	a

(b)The conceptual matrix M' of $\boldsymbol{x}'\$ = abaacbaab\$$

Figure 2.5: The impact of inserting a letter c at position 5 in $\boldsymbol{x}$ on the matrix M

We will give an overview of the algorithm developed by [26] to see how this algorithm can be extended to update the suffix array. Let us consider the following simple case: a letter c is inserted at position i in $\boldsymbol{x}\$$ (i.e. $\boldsymbol{x'} = \boldsymbol{x}[1..i-1]c\boldsymbol{x}[i..n]\$$). We will take as an example our string $\boldsymbol{x} = abaabaab$. Suppose we are inserting a letter c at position $i = 5$. The new string will become $\boldsymbol{x'} = abaacbaab$. Let $\boldsymbol{M'}$ be the the conceptual matrix of $\boldsymbol{x}'\$$ (see Fig.2.5(b)). Examining the cyclic shifts of $\boldsymbol{x}'\$$ (i.e. rows of M'), we can notice that the letter c appears in a cyclic shift $\boldsymbol{x'}^{[j]}$ at one of the following positions (see Fig. 2.6):

- c appears right of **\$** and before L (case Ia). The cyclic shifts $\boldsymbol{x'}^{[9]}$, $\boldsymbol{x'}^{[6]}$, $\boldsymbol{x'}^{[7]}$ and $\boldsymbol{x'}^{[8]}$ are examples for this case (see Fig. 2.5(b)).

- c appears in L (case Ib). For example, the cyclic shift $\boldsymbol{x}'^{[5]}$ has c in column L (see Fig. 2.5(b)).

- c appears left of \$ and after F (case IIa). The cyclic shifts $\boldsymbol{x}'^{[3]}$, $\boldsymbol{x}'^{[0]}$, $\boldsymbol{x}'^{[1]}$ and $\boldsymbol{x}'^{3}$ are examples for this case (see Fig. 2.5(b)).

- c appears in F (case IIb). For example, the cyclic shift $\boldsymbol{x}'^{[4]}$ has c in column F (see Fig. 2.5(b)).

$$\begin{array}{rll} \boldsymbol{x}[j..n]\$\boldsymbol{x}[1..i-1]c\boldsymbol{x}[i..j-1] & \text{if } i < j \leq n & \text{(Ia)} \\ \boldsymbol{x}[i..n]\$\boldsymbol{x}[1..i-1]c & \text{if } j = i & \text{(Ib)} \\ c\boldsymbol{x}[i..n]\$\boldsymbol{x}[1..i-1] & \text{if } j = i-1 & \text{(IIa)} \\ \boldsymbol{x}[j+1..i-1]c\boldsymbol{x}[i..n]\$\boldsymbol{x}[1..j] & \text{if } 0 \leq j < i-1 & \text{(IIb)} \end{array}$$

Figure 2.6: All possible locations of c in $\boldsymbol{x}'^{[j]}$ after its insertion at position i

The following lemma is proved in [26]:

Lemma 2.3.2. *Inserting a letter c at position i in $\boldsymbol{x}$ has no effect on the respective ranking of cyclic shifts whose orders are strictly greater than i. That is: for all $j \geq i$ and $j' \geq i$, we have $\boldsymbol{x}^{[j]} < \boldsymbol{x}^{[j']} \leftrightarrow \boldsymbol{x}'^{[j+1]} < \boldsymbol{x}'^{[j'+1]}$.*

Based on the above four situations and the previous Lemma, [26] presented the following four-stage algorithm for updating BWT (see Fig. 2.7 and Fig. 2.8):

(Ia) Ignore: no direct impact on either L or F.

(Ib) Modification: for row $ISA[i]$, the letter in L is stored (i.e. $b = L[ISA[i]]$) and replaced by c.

(IIa) Insertion: a new row is inserted at position $LF(ISA[i])$, F receives c and L receives b.

(IIb) Gently reorganize the rows that are affected by the insertion.

	F	***L***		***F***	***L***		***F***	***L***
1	\$	b		\$	b		\$	b
2	a	b		a	b		a	b
3	a	b		a	b		a	b
4	a	a		a	a		a	a
5	a	a	$\overset{(Ib)}{\rightarrow}$	a	a	$\overset{(IIa)}{\rightarrow}$	$\boldsymbol{c}$	$\boldsymbol{a}$
6	a	\$		a	\$		a	a
7	b	a		b	a		a	\$
8	b	a		b	$\boldsymbol{c}$		b	a
9	b	a		b	a		b	c
10							b	a
	(Ia)			(Ib)			(IIa)	

(Ia) No impact;

(Ib) For ISA[5] = 8, the letter a in L is stored and replaced with c,

(IIa) A new row is inserted at position $LF(\text{ISA}[5]) = LF(8) = 5$, F receives $\boldsymbol{c}$ and L receives the stored a.

Figure 2.7: Stages (Ia), (Ib) and (IIa) of updating BWT

	F	***L***		***F***	***L***
1	\$	b		\$	b
2	a	b		a	b
3	a	b		a	b
4	a	a		a	a
5	c	a	$\overset{(IIb)}{\rightarrow}$	a	\$
6	a	a		a	a
7	a	\$		b	a
8	b	a		b	c
9	b	c		b	a
10	b	a		c	a
	(IIa)			(IIb)	

The fourth stage is slightly more complicated: by inserting a new row in M during stage (IIa), we somehow disrupt the LF function and create inappropriate relations between letters in F and L. We therefore have to consider the local rearrangement that might occurs (they consist in rotations, a row k moves to row k' and all the rows between k and k' are shifted by one position accordingly).

Figure 2.8: Stage (IIb) of updating BWT

The rearrangement in stage (IIb) is performed as long as the "expected" LF value is different from the "actual" LF value. The "expected" LF value is computed by

summing $rank_{\boldsymbol{x}}(c, i)$ and the value $C(c)$ (see equation 2.3.1). Fig.2.9 shows the reordering algorithm used in stage (IIb).

2.3.3 Updating SA

The four-stage algorithm for updating BWT naturally extends to updating SA due to the closeness between the BWT and SA [27].

Recall that the LF function allows us to navigate in L from the ith to the $(i-1)$th cyclic shifts. Notice also that the ith cyclic shift corresponds to the suffix beginning at position $i+1$. Thus, $LF(i) = j$ iff $\mathrm{SA}[i] = \mathrm{SA}[j] + 1$ (see section 2.3.1).

function REORDER(L, i)
  — *gives the actual position of* $\boldsymbol{x}^{[i-1]}$ *in* M
$j \leftarrow index(\boldsymbol{x}^{[i-1]})$;
  — *gives the computed position of* $\boldsymbol{x'}^{[i-1]}$ *in* M'
$j' \leftarrow LF(index(\boldsymbol{x'}^{[i-1]}))$;
while ($j \neq j'$) **do**
  $new_j \leftarrow LF(j)$;
  — *moves a row of* L *from position* j *to* j'
  MOVEROW(L, j, j);
  $j \leftarrow new_j$;
  $j \leftarrow LF(j')$;

Figure 2.9: REORDER used in stage (IIb)

[27] considered the BWT updating algorithm and induced the required modification on both SA and ISA arrays caused by inserting a letter c at position i of $\boldsymbol{x}$. The following is the extension of the four-stage BWT algorithm to SA:

- Stage 1 (Ia) - suffixes $\boldsymbol{x}[j..n]$, $\forall j > i$: From Lemma 2.3.2, the respective ranking of the corresponding cyclic shifts is conserved. Hence, SA and ISA are not

modified during this stage.

- Stage 2 (Ib) - suffix $\boldsymbol{x}[i..n]$: The same condition applies here as in the previous stage. Notice that during this stage of BWT algorithm, for $k = \text{ISA}[i]$ the letter $b = L[k]$ is stored and replaced by c.

- Stage 3 (IIa) - suffix $c\boldsymbol{x}[j..n]$: At this stage in BWT algorithm, a new row is inserted at position $k' = LF(k)$ in l with the letter stored from the previous stage i.e. $L[k'] = b$. This would reflect to the following modifications in ISA and SA (for our example see Fig. 2.10):

  - SA: insertion of i at index k':

    1. all values in SA greater than or equal to i are incremented.
    2. value i is inserted at index k'.

  - ISA: insertion of k' at index i:

    1. all values in ISA greater than or equal to k' are incremented.
    2. value k' is inserted at index i.

- Stage 4 (IIb) - suffixes $\boldsymbol{x}'[j..n]$, $j < i$: This is the reordering stage which is done using the same REORDER algorithm that is used in BWT algorithm (see 2.9). If without loss of generality, we suppose $j < j'$, then the following applies (for our example see Fig. 2.11):

- L: the element at position j is moved to position j'.
- SA: the element at position j is moved to position j'.
- ISA: all values between j (excluded) and j' (included) are decremented by 1. Then, j is modified to j'.

	SA	***ISA***
1	9	6
2	6	9
3	3	3
4	7	5
5	4	8
6	1	2
7	8	4
8	5	7
9	2	1
10		

(Ia) & (Ib)

$\xrightarrow{(IIa)}$

	SA	***ISA***
1	10 ~~9~~	7 ~~6~~
2	7 ~~6~~	10 ~~9~~
3	3	3
4	8 ~~7~~	6 ~~5~~
5	$\boxed{5}$	$\boxed{5}$
6	4	9 ~~8~~
7	1	2
8	9 ~~8~~	4
9	6 ~~5~~	8 ~~7~~
10	2	1

(IIa)

(Ia) No impact on SA or ISA.

(Ib) For ISA[5] = 8, we store the original $LF(8) = 5$ in *pos* but there is no impact on SA or ISA at this stage.

(IIa) All values greater than or equal to $i = 5$ are incremented in SA and value $i = 5$ is inserted at position $k' = LF(8) = 5$. We update *pos* to 6. All values greater than or equal to $LF(8) = 5$ are incremented in ISA and value $k' = LF(8) = 5$ is inserted at position $i = 5$.

Figure 2.10: Stages (Ia), (Ib) and (IIa) of updating SA and ISA

	SA	***ISA***
1	10	7
2	7	10
3	3	3
4	8	6
5	5	5
6	4	9
7	1	2
8	9	4
9	6	8
10	2	1

(IIa)

$\xrightarrow{(IIb)}$

SA	***ISA***
10	5
7	9
3	3
8	6
1	10
4	8
9	2
6	4
2	7
5	1

(IIb)

Figure 2.11: Stage (IIb) of updating SA and ISA

DSA needs to store L (which provides *rank* queries) and C (which provides *count* function) which are essential for computing the LF function. **DSA** also requires the inverse suffix array ISA during the reordering stage.

The previous algorithm can be generalized to the insertion/deletion/substitution of a letter or a factor (see [27] for details).

2.3.4 Sampling

Since SA and ISA are space-consuming structures, the authors discussed the choice of compressing these arrays using sampling techniques. Without using the sampling technique, the space requirement would be $8n$ bytes plus the space required for storing L and C (see previous section).

Basic Idea

Compressed data structures [35] that support rank/select operations and require only $o(n)$ bits of storage provided a solution for reducing the space requirements. This solution is based on the idea of sampling; we store only a few values over the entire sequence.

We will illustrate the idea of sampling on the ISA. Let us consider the following ISA from Figure 2.2:

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{ISA} = & 5 & 8 & 2 & 4 & 7 & 1 & 3 & 6, \end{array} \tag{2.3.2}$$

A dynamic sample of ISA consists of two bit vectors and one integer array as

follows:

- The first bit vector, m_{ISA}, indicates the positions where ISA is sampled.
- The second bit vector,v_{ISA} , indicates the set of values that are sampled.
- The integer array, π_{ISA}, gives the respective order of the sampled values. In other words, it maps a sampled position to its corresponding sampled value.

Figure 2.12 shows the bit vectors and the integer array for our ISA example.

$$
\begin{array}{rcccccccccc}
\text{sampled i} & & & 1 & & 3 & & 5 & & 7 & \\
\boldsymbol{m_{ISA}} & = & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\text{sampled ISA} & = & 5 & & 2 & & 7 & & 3 & \\
\boldsymbol{v_{ISA}} & = & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
\boldsymbol{\pi_{ISA}} & = & 5 & 2 & 3 & 7 & & & & &
\end{array}
$$

Figure 2.12: Retrieving a value for a sampled position ISA[5]

Suppose we want to retrieve the value of ISA[5]. We need to follow the following steps:

1. $m_{ISA}[5] = 1$ is a sampled position. Hence, we apply $rank_{m_{ISA}}(1, 5) = 3$.
2. $\pi_{ISA}[3] = 3$.
3. $ISA[5] = select_{m_{ISA}}(1, 3) = 7$.

Since rank and select functions can be formed in $O(log\ n)$ worst-case time, retrieving a value at a sampled position costs at most $O(log\ n)$ plus the cost of accessing the π_{ISA} dynamic structure.

Retrieving a value at an unsampled position is a bit more complex process. It consists of applying a series of *rank* and *select* queries and one call to the LF function at the last step. The time for this process is bounded by $O(off \times log\ n(1 + \frac{log\ \sigma}{log\ log\ n}))$ plus the time for accessing π_{ISA}, where σ is the alphabet size and ***off*** is the offset between i and the sampled position to the right of i.

For more details on adding/removing a sample, updating the permutation , insertion/deletion of a value j at position i we refer the reader to [27].

Improving Retrieve and Update Time

In order to guarantee fast access to the ISA values, π_{ISA} needs to be stored and updated in an efficient way. [27] proposed using two balanced binary trees $\boldsymbol{A}$ and $\boldsymbol{B}$. Let $A[i]$ be the i^{th} node in A and $B[j]$ be the j^{th} node in B. Using A and B, a permutation π of n element is defined as follows: $\pi[i] = j$ if and only if there exists a link from $A[i]$ to $B[j]$.

Since ISA is the inverse of SA; SA can be computed in a similar fashion. Notice that for computing ISA, π_{ISA} is the mapping between m_{ISA} (positions in ISA) and v_{ISA} (values in ISA). Symmetrically, for computing SA: π^{-1}_{ISA} is the mapping between v_{ISA} (positions in SA) and m_{ISA} (values in SA). Using the previous presented structure of the two balanced tree, π^{-1}_{ISA} can be easily computed by making the links bidirectional.

2.3.5 Space and Time Complexity

	Sampling	Non-Sampling
Retrieve ISA/SA value	$O(log^{1+\epsilon}\ n\ (1+\frac{log\sigma}{log\ log\ n}))$	$O(log\ n)$
Update	$O(n\ log\ n(1+\frac{log\ \sigma}{log\ log\ n}))$	$O(log\ n)$
Space	$o(n)$ bits	$O(n\ log\ n)$ bits

Table 2.1: Sampling vs. Non-sampling: Time and space complexity

Let us consider the space and time complexity for both cases sampling and non-sampling.

For the sampling case, we need two bit vectors. [27] uses dynamic compressed bit vectors developed by Mankinen and Navarro [35] that need $nH_0+o(n)$ bits and handle all the needed operations (i.e. rank/select) in $O(log\ n)$ worst-case time. H_0 denotes the zero-th order entropy of the bit vector and since the bit vectors are sparse, their space consumption is $o(n)$ bits only. Furthermore, we need the permutation array π_{ISA} which require $O(n\ log\ n/log^{1+\epsilon}\ n) = o(n)$ bits using the structure of the two binary trees. Obtaining a value from SA or ISA requires $O(log^{2+\epsilon}\ n + log\ n\ log\sigma/log\ log\ n)$ worst-case time as the LF function requires $O(log\ n+log\ n\ log\ \sigma/log\ log\ n)$ worst-case time. Table 2.2 summarizes these bounds.

Using the whole SA and ISA can be seen as a special case of the sampled SA and ISA where all positions are sampled. The bit arrays become useless here since every position is essentially sampled. Only π_{ISA} is meaningful since it corresponds to SA and ISA arrays and it is represented by the two binary trees. According to [27], in

bit vectors m_{ISA} **and** v_{ISA}	space rank/select time	$nH_0 + o(n)$ bits $O(log\ n)$
permutation array π_{ISA}	space operations time (retrieving/inserting/deleting)	$O(\frac{n\ log\ n}{log^{1+\epsilon}\ n}) = o(n)$ bits $O(log\ n)$
LF function time		$O(log\ n + \frac{log\ n\ log\ \sigma}{log\ log\ n})$

Table 2.2: Time and space complexity for various operations and structures used for updating SA

this case any value of SA and ISA can be accessed in $O(log\ n)$ worst-case time using $O(n\ log\ n)$ bits. See Table 2.1 for a comparison between sampling and non-sampling space/time complexity.

Chapter 3

LZ Compression

In general, compression methods based on strings of symbols can be more efficient than methods that compress individual symbols. The probabilities of strings of symbols vary more than the probabilities of the individual symbols constituting the strings. Hence, dictionary-based compression methods select strings of symbols and encode each string as a *token* using a dictionary. The dictionary holds strings of symbols, and it may be static or dynamic (adaptive). As we mentioned previously, LZ compression methods are dictionary-based. In this brief chapter, we describe the original LZ methods: **LZ77** and **LZ78**. We attempt to differentiate between the two and illustrate each method using a simple example.

3.1 LZ77

In general terms, an **LZ** factorization of $\boldsymbol{x}$ is a decomposition of $\boldsymbol{x}$ into nonempty factors: $\boldsymbol{x} = \boldsymbol{w_1 w_2 \ldots w_k}$. The factorization of $\boldsymbol{x}$ can be reported in several ways. In its native form, **LZ77** factorization [47] reports each factor $\boldsymbol{w_j}$ as a triple (POS,LEN,λ),

where:

- POS is the location of a prior occurrence of $\boldsymbol{w_j}$ in $\boldsymbol{x}$ or the location of $\boldsymbol{w_j}$ if no previous occurrence exists;
- LEN is the length (possibly zero) of the matching previous occurrence;
- λ is the "letter of mismatch".

It is noteworthy that this (in general, compressed) encoding of $\boldsymbol{x}$ permits the original string to be reconstituted (decoded) with no need for an explicit dictionary. Essentially, **LZ78** factorization [48] removes LEN from the output, thus compressing the text further, but introducing the need for a dictionary in order to retrieve the original text.

LZ77 operates not on the string as a whole, but only on a *sliding window* of length $\boldsymbol{N}$ (usually $\boldsymbol{N} = 4096$ or 8192), with a long prefix that has already been factored and a short (typically 18 letters) as-yet-unfactored suffix $\boldsymbol{F}$. The next factor $\boldsymbol{w_j}$ is the longest prefix of $\boldsymbol{F}$ that matches a preceding substring within the window. Once $\boldsymbol{w_j}$ has been determined, the window is shifted right by $|\boldsymbol{w_j}|$ positions. It has been found that in practice the use of the sliding window provides compression as good as using the entire string would yield, and of course processing time is substantially reduced. We will illustrate the basic idea of **LZ77** using an example. We will use ␣ to denote a white space. The data shown in Fig. 3.1 is to be encoded. The string

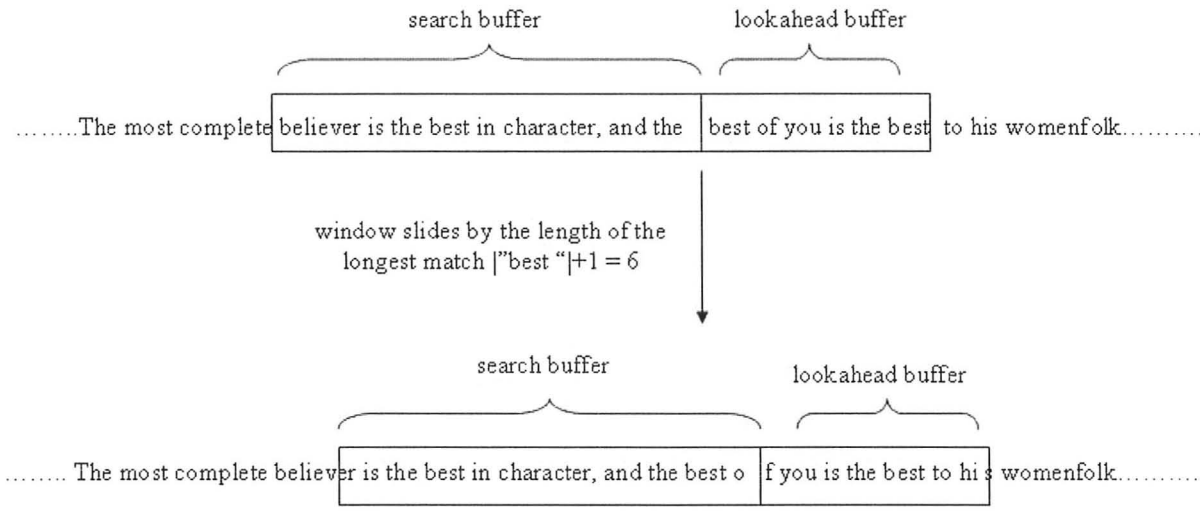

Figure 3.1: Example to illustrate **LZ77**

in the search buffer has already been encoded, while the string in the lookahead buffer is yet to be encoded. The algorithm works from left to right and has already encoded the string $\boldsymbol{S}$="believer␣is␣the␣best␣in␣character,␣and␣the␣". The string $\boldsymbol{F}$="best␣of␣you␣is␣the␣best" is the data yet to be encoded.

First, the algorithm searches for the longest match for the string in the encoded string $\boldsymbol{S}$ matching a prefix of $\boldsymbol{F}$. In this specific example, the longest match is the string "best␣" starting at the 17th position (counting from one). Therefore, it is possible to code the first five characters of $\boldsymbol{F}$ (i.e."best␣") as a reference to the substring that occurs at position 17 of the search buffer. As mentioned previously, references are encoded as a fixed-length codeword consisting of three elements: position, length and first non-matching symbol. In our case, the codeword would be (17, 5, 'o'). As we

can see, five characters have been coded with just one codeword. When the matches get longer, those coded references will consume significantly fewer space than, for example, coding every thing in *ASCII*.

Fig. 3.2 is pseudocode of **LZ77**.

```
while lookAheadBuffer not empty do

    get a reference (position, length) to longest match;
    if length > 0 then
        output(position, length, next symbol);
        shift the window length+1 positions along;
    else
        output( 0, 0, first symbol in the lookAheadBuffer);
        shift window 1 character along;
```

Figure 3.2: Pseudocode of **LZ77**

The algorithm starts out with the lookahead buffer filled with the first symbols of the data to be encoded and the search buffer filled with a predefined symbol of the input alphabet (zeros, for example). Some of the **LZ77** variants that improved upon the original version are: **LZSS** [43], **LZRW** [44], **LZB** [30], **LZH**(developed by Haruyasu Yoshizaki), and LZP [19]. **LZSS** is the most prominent amongst **LZ77** variants.

The **LZ77** encoder and the decoder exhibit high asymmetry. In particular, the decoder is much simpler than the encoder. It merely prepares a buffer with the same window size as the encoder. It starts each iteration by grabbing the next token (i.e. codeword) on its input stream and finds the match in its buffer. It then writes the

match and the third token field (i.e. the symbol) on the output stream. In the last step of every iteration, the decoder shifts the matched string and the third token to the buffer. Because the decoder is fast and simple, **LZ77** is particularly useful in cases when a file is compressed once and decompressed several times.

3.2 LZ78

Unlike **LZ77**, **LZ78** [48] is a dictionary-based compression algorithm that maintains an *explicit* dictionary. **LZ78** has a slightly different codewords from **LZ77**. **LZ78** codewords consist of two fields: the location of the longest matching entry in the dictionary and the first "letter of mismatch". The **LZ78** scheme removes the need for the length of the match since it is implied from the matching entry in the dictionary. This scheme thus further compresses the text. When outputting the codeword, the algorithm simultaneously adds the index and the symbol pair to the dictionary as a new entry. When a symbol, that is not yet in the dictionary, is encountered, the codeword is assigned the next available index value in the dictionary. Then, it is added to the dictionary as a new entry. With this approach, the algorithm gradually builds up a dictionary. Fig. 3.3 shows pseudocode of **LZ78**.

Table 3.1 shows the first 31 steps in encoding the string:

> "The most complete believer is the best in character, and the best of you is the best to his womenfolk" [1]

[1]Prophet Mohammed, Tirmidhi #1162 and verified

```
w ← NIL;

while there is input do
    k ← next symbol from input;
    if wk exists in the dictionary then
        w ← wk;
    else
        output( index(w), k);
        add wk to the dictionary;
        w ← NIL;
```

Figure 3.3: Pseudocode of LZ78

A good data structure for the dictionary is a tree, but not a binary one. The tree starts with the null string as the root. All the strings that start with the null string (strings for which the token pointer is *zero*) are added to the tree as children of the root. In the above example those are 'T', 'h', 'e', '␣', 'm', 'o', 's', 't', 'p', 'l', 'b', 'i', 'a' and 'r'. Each of them become a root of a subtree as shown in Fig. 3.4. For example, all the strings that start with 't' (the three strings "th", "t␣", and "te") constitute the subtree of node 't'.

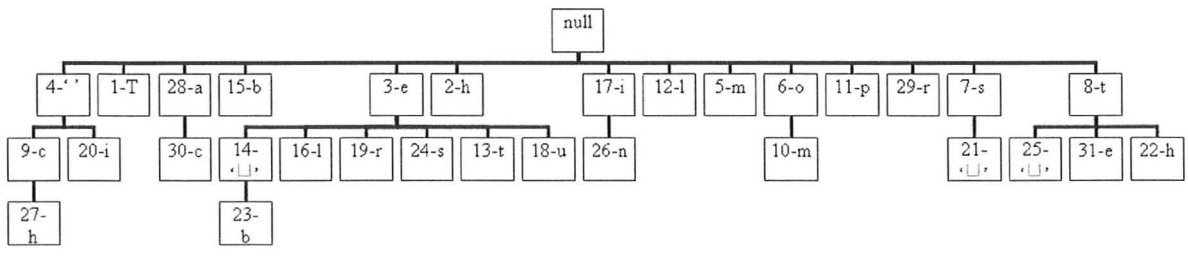

Figure 3.4: An **LZ78** Dictionary Tree

As you can see from the algorithm, the **LZ78** decoder is more elaborate than the **LZ77** decoder. It builds the dictionary in a similar manner as it is built through the

Dictionary		Token	Dictionary		Token
0	null		16	"el"	(3,'l')
1	"T"	(0, 'T')	17	"i"	(0,'i')
2	"h"	(0, 'h')	18	"eu"	(3,'u')
3	"e"	(0, 'e')	19	"er"	(3,'r')
4	" "	(0, '␣')	20	" i"	(4,'i')
5	"m"	(0, 'm')	21	"s "	(7,'␣')
6	"o"	(0, 'o')	22	"th"	(8,'h')
7	"s"	(0, 's')	23	"e b"	(18,'b')
8	"t"	(0, 't')	24	"es"	(3,'s')
9	" c"	(4, 'c')	25	"t "	(8,'␣')
10	"om"	(6, 'm')	26	"in"	(17,'n')
11	"p"	(0, 'p')	27	" ch"	(9,'h')
12	"l"	(0, 'l')	28	"a"	(0,'a')
13	"et"	(3, 't')	29	"r"	(0,'r')
14	"e "	(3, '␣')	30	"ac"	(28,'c')
15	"b"	(0, 'b')	31	"te"	(8,'e')

Table 3.1: First 32 steps in LZ78

encoding process. It grabs a codeword and uses the POS to locate the entry in the dictionary and copies that entry at the end. Next, it adds the copied entry contacted with lmda as a new entry to the dictionary if there is a position available. It continues applying this process until the entire string is recovered.

Note that this pseudocode is a simplified version of the algorithm and it does not prevent the dictionary from growing without bound. The simplest solution to limit the dictionary size is to stop adding entries and continue as a *static dictionary* encoder. Another solution is to throw the dictionary away and start from scratch after a specific number of entries has been reached. There are more sophisticated

approaches that give raise to the collection of **LZ78** family algorithms. Some of **LZ78** family are: LZW [45] , LZC (a software version of LZW with some additional features used in **compress** utility), LZMW [34], LZMS, LZJ, LZFG [16] (a hybrid of LZ77 and LZ78). LZW is the most popular variant of LZ78.

Chapter 4

New Algorithm

As we mentioned in Chapter 1, the experiment of scaling up an implementation of LZSS, due to Haruhiko Okumura [37], triggered the idea of scaling down the SA-based LZ algorithms. In this chapter we describe our new variant **LZAS** and discuss the variations that stemmed out of it.

4.1 The Core Idea

We designed a novel **LZ77** variant (LZAS) that makes use of the suffix array to perform the search in the dictionary. The main idea is to replace the binary search tree (or any other kind of tree) by the suffix array. Fig. 4.1 is the pseudocode of our algorithm.

The suffix array (SA) posses two nice structural properties which are usually exploited to support fast pattern search:

(i) all the suffixes of the string $\boldsymbol{x}[1..n]$ prefixed by a pattern $\boldsymbol{p}[1..m]$ occupy a contiguous portion (subarray) of SA.

Algorithm LZAS

```
— initialize the window by initializing its two parts: search buffer
— and lookahead buffer
Initialize searchBuffer with predefined symbol of the input alphabet;
Initialize lookAheadBuffer with the first symbols of the data to be encoded;
Compute SA;
while lookAheadBuffer not empty do
    Search for a match (position, length) using SA;
    get a reference to longest match;
    if length > 1 then
        output(flag = 1, position, length);
        shift the window length positions along;
    else
        output(flag = 0, first symbol in the lookAheadBuffer);
        shift window one character along;
    Update SA;
```

Figure 4.1: Pseudocode of **LZAS**

(ii) that subarray has a starting position *sp* and ending position *ep*, where *sp* is actually the lexicographic position of the suffix ***sp*** among the ordered sequence of text suffixes.

We use the above properties to design our search algorithm which we describe in the next section.

One notable advantage of using SA in an LZ encoder is that the amount of memory is independent of the text to be searched and can be defined a priori. The low and predictable memory requirement of this approach makes it suitable for memory-critical applications such as embedded systems. Our proposed algorithm can additionally be used for forward/backward sub-string search.

We follow LZSS codeword format which contains just a position and a length. If no match was found, the encoder emits the uncompressed code of the next symbol instead of the wasteful three-field token (0, 0, ...). To distinguish between tokens and uncompressed codes, each is preceded by a single bit (flag).

4.2 The Search Algorithm

The search in LZAS is done using a simple algorithm that performs a left/right search of the suffix array to obtain the longest match in the search buffer for a prefix of the look-ahead buffer. The pseudocode of the search algorithm is illustrated in Fig. 4.2.

Let N be the length of the window and F be the length of the look ahead buffer. Given a position $r = N - F + 1$ in x, at the right hand side of the window, we want to compute the position of the longest previous substring matching the substring $\boldsymbol{x}[r..N]$. This can be done by searching the neighbourhood of position $j = ISA[r]$ in the suffix array SA.

The search algorithm alternates between searching left and searching right, depending on which side has the maximum LCP value. This approach is very efficient since r will normally be large, because it is on the right hand side of the window, and so the probability will be high that any suitable i' that satisfies $i' < r$ is located immediately. Our experiments confirm this observation.

To see why this observation is true, recall that the window size N is typically a few thousands long (e.g. 4096), while the look ahead buffer F is just tens of bytes

The SA Left/Right Search Algorithm

function Longest_Match_Search(i, SA, ISA, LCP)

```
1   j_l ← ISA[i];
2   if (j_l ≤ 1)
3       i_l ← −1; len′, len_l ← 0
4   else
5       i_l ← SA[j_l − 1]; len′, len_l ← LCP[j_l]
6   j_r ← j_l + 1;
7   if (j_r > N)
8       i_r ← −1; len″, len_r ← 0
9   else
10      i_r ← SA[j_r]; len″, len_r ← LCP[j_r]
11  len ← max(len_l, len_r)
12  while (len > 0) do
13      if (len_l ≥ len_r) then  — Search left.
14          while ( len′ ≥ len_r and i_l > i ) do
15              j_L ← j_L − 1;
16              if (j_l ≤ 1)
17                  i_l ← −1; len′, len_l ← 0
18              else
19                  i_l ← SA[j_l − 1]; len′ ← LCP[j_l]
20                  len_l ← min(len′, len_l)
21          if len′ ≥ len_r then
22              return (i_l, len_r)
23          else
24              len ← len_r
26      else  — Search right.
27          while ( len″ ≥ len_l and i_r > i ) do
28              j_r ← j_r + 1;
29              if (j_r > N)
30                  i_r ← −1; len″, len_r ← 0
31              else
32                  i_r ← SA[j_r]; len″ ← LCP[j_r]
33                  len_r ← min(len″, len_r)
34          if len″ ≥ len_l then
35              return (i_r,len_r)
36          else
37              len ← len_r
38  return (i,0)
```

Figure 4.2: SA Search algorithm: Compute longest previous factor given SA, ISA and LCP

only (e.g. 16). This is what makes our search algorithm very efficient. Let us examine the algorithm in figure 4.2 more closely. We are searching for a position to the left of r that has the longest match with the prefix of the look ahead buffer. We have the following four cases for SA values and LCP values that we use to guide our left/right search:

1. $\text{SA}[j_l - 1]$ and $\text{SA}[j_r]$ are both to left of $i = r$.

2. $\text{SA}[j_l - 1]$ and $\text{SA}[j_r]$ are both to the right of $i = r$.

3. $\text{SA}[j_l - 1] > i$ and $\text{LCP}[j_l] > \text{LCP}[j_r]$.

4. $\text{SA}[j_r] > i$ and $\text{LCP}[j_r] > \text{LCP}[j_l]$.

In the first case we can immediately locate the position of the longest match by comparing $\text{LCP}[j_l]$ and $\text{LCP}[j_r]$ and take the position with greater LCP value. The other three cases require more work. The probability that the first case occurs is very high. To show that, let us try to approximate the probability of the other three cases. We can formulate our question as follows:

> What is the probability that one or both of $\text{SA}[j_l - 1]$ and $\text{SA}[j_r]$ occur to the right of $i = r$ in $\boldsymbol{x}$?

One would expect this probability to be approximately equal to $\frac{F}{N}$ which is very small since N is very large compared to F as we mentioned previously. Therefore,

the probability that that any suitable i' that satisfies $i' < r$ is located immediately is equal to $1 - \frac{F}{N}$ which is very big.

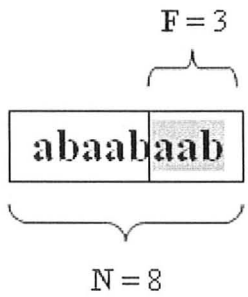

Figure 4.3: Example to illustrate the search algorithm

The approach requires ISA, SA and LCP to be available. In regard to LCP, we had two choices: either to compute the whole LCP array or compute LCP values on a demand basis. We experimented with both choices and found that the second one is more efficient as we expected. This confirms the fact that we need the LCP values rarely according to our previous observation. Hence, our final implementation computes LCP values on demand basis.

Let us illustrate the algorithm with a small example. Let $\boldsymbol{x}$ be our regular string example *abaabaab*. Let us assume that the window length N is equal to the string length $n = 8$ and that $F = 3$. Now we want to find the longest match for the lookahead buffer *aab*, in the search buffer *abaab* (see Fig. 4.3). Figure 4.4 shows the SA/ISA/LCP arrays for our example.

We need to call our search algorithm with position $i = r = N - F = 8 - 3 + 1 = 6$. The first line of our search algorithm (Fig. 4.3) would access ISA at position $i = 6$

$\boldsymbol{i}$		1	2	3	4	5	6	7	8
$\boldsymbol{x}$	=	a	b	a	a	b	a	a	b
$\boldsymbol{SA}$	=	6	3	7	4	1	8	5	2
$\boldsymbol{ISA}$	=	5	8	2	4	7	1	3	6
$\boldsymbol{LCP}$	=	-1	3	1	2	5	0	1	4

Figure 4.4: SA/ISA/LCP arrays for $\boldsymbol{x} = abaabaab$

and store the value in $j_l = 1$. Before accessing SA$[j_l - 1]$ and LCP$[j_l - 1]$, we need to make sure that these are valid positions by checking if $j_l = 1$. Since this is the case in our example, we initialize i_l to -1 and len', len_l to 0 and hence we will not do a left search.

Next, the algorithm computes the values needed to do the right search. In line 6, we compute $j_r = j_l + 1 = 2$ and then line 7 checks if it is in the valid boundary. Since $2 > N = 8$, the variables i_r and len''/len_r are initialized with SA$[j_r] = 3$ and LCP$[j_r] = 3$ respectively. In Figure 4.5, we highlight array values that we had accessed in SA/ISA/LCP.

Before we start the search, we need to store the maximum of len_l and len_r which is 3 in the variable len (i.e. the length of the longest match so far). The rest of the algorithm consists of a while loop that iterates as long as we still did not find our longest match and alternate between searching left and searching right. Once we find a match with the longest length so far that occurs previously (i.e. $i' = j_l$ or $i' = j_r$ and $i' < i$), we return these values (i.e. $POS = i'$, $LEN = l'$ where $l' = len_l$ or $l' = len_r$ which ever is larger).

In our example, we will start by searching right as $len_r > len_l$. But since $i_r = 3 < i = 6$, we will not to search at all since we already have our longest match. The algorithm will terminate at line 36 with $POS = 3$ and $LEN = 3$. That agrees with our conjecture that there is a high probability that any suitable i' that satisfies $i' < r$ will be located immediately. Figure 4.5 shows the variables through the various states of the algorithm.

lines	i	len	j_l	i_l	len'	len_l	j_r	i_r	len"	len_r
$1-5$	6		**1**	**−1**	**0**	**0**				
$6-10$	6		1	−1	0	0	**2**	**3**	**3**	**3**
11	6	**3**	1	−1	0	0	2	3	3	3
$12-35$	6	3	1	−1	0	0	2	3	3	3

Figure 4.5: Various states of the search algorithm executed for $\boldsymbol{x} = abaabaab$ with $i = 6$

4.3 Update or Recompute?

Another question was raised from the fact that SA changes whenever the window slides. Therefore, we need the new SA that corresponds to the new window. That led us to two choices: either updating the SA or re-computing it.

To update the SA, we needed to find an algorithm that does this efficiently. The dynamic suffix array ***DSA*** described in Chapter 2 (see section 2.3) allows us to do that. The DSA [27] is designed in such a way as to be updateable whenever characters are added, deleted or edited in the original string. DSA can be used in its compact form (i.e. sampled SA) or non-compact (i.e. non-sampled SA) form. We

decided to experiment with both forms. This give raise to two variants of the LZAS: **LZAS1** and **LZAS2**.

Recomputing SA is the straightforward choice and hence we decided to experiment with it as well. We use it as a measure for the performance of the updating approach. We call this variant **LZAS3**.

4.4 Is there a better way to update SA?

As we mentioned in section 1.3, it turns out that our approach becomes uncompetitive when compared to the Okumura [37] implementation. We gave two reasons for this result in Section 1.3. We will discuss in this section the second reason which relates to the mechanism of updating the suffix array.

Recall that we need to update the suffix array whenever the window slides. If the update was not efficient enough, then this will slow the whole algorithm as this is the most time consuming part of our algorithm. It seems that DSA might not be effecient enough after all. DSA was designed with a general update in mind i.e. adding/deleting /substituting a letter or a factor in the original string, regardless of the position. We think it is possible to develop an algorithm that updates the suffix array more efficiently for cases of deleting a factor (prefix) at the beginning of the string and adding a factor (suffix) at the end of the string.

Let us investigate this case closely. Suppose we have a string $\boldsymbol{x}$ and we have its suffix array $SA_{\boldsymbol{x}}$. Deleting a prefix $\boldsymbol{p}$ of length t from $\boldsymbol{x}$ will not affect $SA_{\boldsymbol{x}}$ too

much. Let $\boldsymbol{x'}$ be the string produced by this operation. This delete corresponds to deleting the longest t suffixes in $\boldsymbol{x}$. For $j > t$, suffix j in $\boldsymbol{x}$ becomes suffix $j - t$ in $\boldsymbol{x'}$. Then, we only need to subtract t from each entry in the $SA_{\boldsymbol{x}}$ and delete the nonpositive entries to get $\mathrm{SA}'_{\boldsymbol{x'}}$. We will see later on that for practical purposes we will not delete these entries as we can use their slots for the new suffixes that will result from adding a suffix of the same length as $\boldsymbol{p}$. Figure 4.6 illustrates this idea with the string $\boldsymbol{x} = abaabaab$ and $\boldsymbol{p} = aba$.

$\boldsymbol{i}$		1	2	3	4	5	6	7	8
$\boldsymbol{x}$	=	a	b	a	a	b	a	a	b
$\boldsymbol{SA_x}$	=	6	3	7	4	1	8	5	2

subtract $|p| = 3$

⇓

$\boldsymbol{x'}$	=				a	b	a	a	b
$\boldsymbol{SA'_x}$	=	3	0	4	1	-2	5	2	-1

delete nonpositive entries

⇓

$\boldsymbol{i}$		1	2	3	4	5
$\boldsymbol{x'}$	=	a	b	a	a	b
$\boldsymbol{SA_{x'}}$	=	3	4	1	5	2

Figure 4.6: The effect of deleting a prefix $\boldsymbol{p} = \boldsymbol{aba}$ from $\boldsymbol{x} = abaabaab$ on SA

Now let us investigate the effect of adding a suffix $\boldsymbol{s}$ of length t to $\boldsymbol{x'}$ on $SA_{\boldsymbol{x'}}$. Let us call this new string $\boldsymbol{x''}$, that is $\boldsymbol{x''} = \boldsymbol{x's}$. This situation requires more work in order to update the suffix array.

Let us examine the effect of this insert/append operation on the corresponding order of suffixes of $\boldsymbol{x'}$. Suppose $\boldsymbol{u}$ and $\boldsymbol{v}$ are suffixes of $\boldsymbol{x'}$ and assume that $|u| > |v|$ without loss of generality. One of the following two cases would arise:

1. $\boldsymbol{u} > \boldsymbol{v}$: From the definition of lexicographic order in Section 2.1.1, one of the following two cases will hold:

   1.1 $\boldsymbol{v}$ is a prefix of $\boldsymbol{u}$, that is $\boldsymbol{u} = \boldsymbol{vu'}$. Hence, the new suffixes $\boldsymbol{us} = \boldsymbol{vu's} < \boldsymbol{vs}$ if and only if $\boldsymbol{u's} < \boldsymbol{s}$.

   1.2 $\boldsymbol{u}$ and $\boldsymbol{v}$ differ at position i, that is $\boldsymbol{u}[1..i-1] = \boldsymbol{v}[1..i-1]$ and $\boldsymbol{u}[i] > \boldsymbol{v}[i]$. Therefore, the new suffixes will have the same order (i.e. $\boldsymbol{us} > \boldsymbol{vs}$).

2. $\boldsymbol{u} < \boldsymbol{v}$: Since $|\boldsymbol{u}| > |\boldsymbol{v}|$, from the definition of lexicographic order $\boldsymbol{u}$ and $\boldsymbol{v}$ necessarily differ at some position i. Hence, adding a suffix s will not change the corresponding order of the new suffixes (i.e. $\boldsymbol{us} < \boldsymbol{vs}$).

From the above analysis, we can see that the order of suffixes changes rarely. Specifically, the order changes if and only if $\boldsymbol{v}$ is a prefix of $\boldsymbol{u}$ and $\boldsymbol{u's} < \boldsymbol{s}$ (see case 1.1 above). This analysis leads us to the following lemma:

Lemma 4.4.1. *Suppose $\boldsymbol{u}$ and $\boldsymbol{v} > \boldsymbol{u}$ are suffixes of a given string $\boldsymbol{x}$. Then for any nonempty string $\boldsymbol{s}$, $\boldsymbol{vs} > \boldsymbol{us}$ if and only if one of the following conditions holds:*

(a) $\boldsymbol{u}$ is not a border of $\boldsymbol{v}$;

(b) $\boldsymbol{v}$ is a border of $\boldsymbol{u} = \boldsymbol{vu'}$ and $\boldsymbol{s} > \boldsymbol{u'}$.

The previous lemma indicates that $\boldsymbol{v}$ changes its relation to another suffix $\boldsymbol{u}$ only rarely, since a very large proportion of the time either (a) or (b) will hold. It seems to show that the technique of using SA in a sliding window context is viable.

Now given the suffix array $SA_{\boldsymbol{x'}}$ and the substring $\boldsymbol{s}$ added to the end of $\boldsymbol{x'}$, how can we construct (or update) the suffix array of the new string $\boldsymbol{x''} = \boldsymbol{x's}$? There are two things that we need to take care of in this setting:

1. The respective ranking of the suffixes that results from appending $\boldsymbol{s}$ to the suffixes of $\boldsymbol{x'}$.

2. The new suffixes that need to be inserted and that comes from the suffixes of $\boldsymbol{s}$.

A quick glance at the first problem leads us to a simple solution. The solution involves scanning the suffix array $SA_{\boldsymbol{x'}}$ from right to left and comparing the updated suffixes that are adjacent i.e. $\boldsymbol{us}$ and $\boldsymbol{vs}$. If they are not in a correct lexicographic order, we swap them. We stop this process as soon as we cross over suffixes with maintained respective ranking (see lemma 4.4.1). This is just an outline of a possible solution and needs more investigation. See Figure 4.7 which illustrates this process when adding $\boldsymbol{s} = baa$ to $\boldsymbol{x'} = abaab$. Notice that we have kept the empty entries when we have deleted the prefix $\boldsymbol{p} = \boldsymbol{aba}$ from $\boldsymbol{x} = abaabaab$ as we will need them for the final updated SA.

$\boldsymbol{i}$		1	2	3	4	5	6	7	8
$\boldsymbol{x''}$	=	a	b	a	a	b	b	a	a
$\boldsymbol{SA_{x'}}$	=	3		4	1		**5**	**2**	

compare suffix 2 and suffix 5: $baabbaa < bbaa \rightarrow$ swap suffix 5 and suffix 2

$$\Downarrow$$

$\boldsymbol{x''}$	=	a	b	a	a	b	b	a	a
$\boldsymbol{SA_{x'}}$	=	3		4	1		$\boxed{2}$	$\boxed{5}$	

compare suffix 2 and suffix 1: $baabbaa < abaabbaa \rightarrow$ no swap is required. Stop scanning.

Figure 4.7: Step 1 of updating the suffix array after the adding a suffix $\boldsymbol{s} = baa$ to $\boldsymbol{x'} = abaab$

Handling the insertion of the new suffixes that comes from appending $\boldsymbol{s}$ to $\boldsymbol{x'}$ needs more work. One solution would involve the following steps:

1. Construct the suffix array of $\boldsymbol{s}$, $SA_{\boldsymbol{s}}$. Since $\boldsymbol{s}$ is usually small (typically 18 characters long), this is not a costly computation.

2. Add $|\boldsymbol{x'}|$ to each entry of $SA_{\boldsymbol{s}}$. Let us call this resulting array A.

3. We then can compute the updated suffix array $SA_{\boldsymbol{x''}}$ by merging A and $\mathrm{SA}'_{\boldsymbol{w'}}$ using insertion sort. Simply, we take each suffix $A[i]$ and compare it to the suffixes in $SA'_{\boldsymbol{x'}}$ scanning it from right to left. Once we found a suffix j that is less than $A[i]$, we stop and insert $A[i]$ before that suffix. This would probably involve shifting the previously scanned suffixes if there is no empty slot before suffix j.

It is apparent from step 3 above why we decided to keep the empty slots (i.e. slots for the nonpositive entries) in the suffix array after deleting the prefix $\boldsymbol{p}$. Figure 4.8 shows the above processing for our previous example in which we add $\boldsymbol{s} = baa$ to $\boldsymbol{x'} = abaab$

1. Compute the suffix array of $\boldsymbol{s} = baa$

$$\begin{array}{lcccc} \boldsymbol{i} & & 1 & 2 & 3 \\ \boldsymbol{s} & = & b & a & a \\ \boldsymbol{SA_s} & = & 3 & 2 & 1 \end{array}$$

$$\Downarrow$$

2. Add $|x'| = 5$ to each entry of SA_s

$$\begin{array}{lcccc} \boldsymbol{i} & & 1 & 2 & 3 \\ \boldsymbol{s} & = & b & a & a \\ \boldsymbol{A} & = & 8 & 7 & 6 \end{array}$$

$$\Downarrow$$

3. Merge $SA_{\boldsymbol{x'}}$ and A to produce $SA_{\boldsymbol{x''}}$ using insertion sort

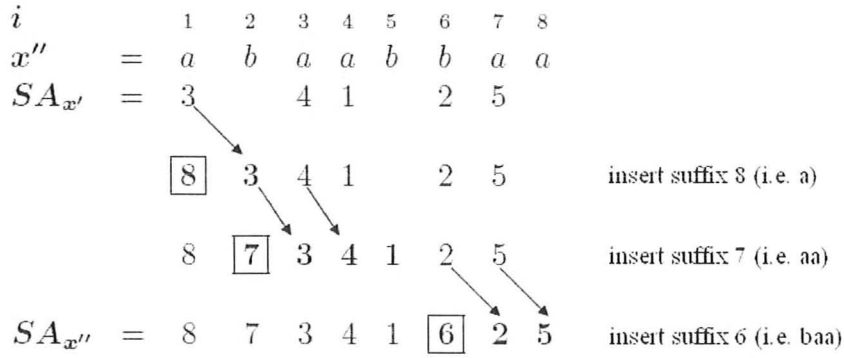

Figure 4.8: Step 2 of updating the suffix array after the adding a suffix $\boldsymbol{s} = baa$ to $\boldsymbol{x'} = abaab$

Chapter 5

Experiments

In this chapter we investigate the practical times of our algorithm variants (i.e. LZAS1, LZAS2 and LZAS3) using standard files from the well-known Calgary[1] and Canterbury[2] corpora.

5.1 Implementation

We have implemented the three variants of LZAS described in Chapter 4. As we mentioned in Chapter 4, the first two variants, LZAS1 and LZAS2, use the dynamic suffix array **DSA** desrcibed in Section 2.3. DSA can be used in its compact form (i.e. sampled SA) or non-compact (i.e. non-sampled SA) form. We experimented with both forms. The code for both forms were provided by Mikael Salson. The third vraiant, LZAS3, recomputes the suffix array whenever the window slides. For testing purposes, we use a supralinear suffix array construction algorithm **SACA** but

[1] `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus?`
[2] `http://corpus.canterbury.ac.nz?`

efficient according to [28]. The code for this SACA is due to N. Jesper Larsson[3]. We also tested an LZSS implementation in C due to Okumura [37] against LZAS variants.

All codes were optimized using various optimization techniques. They were also compiled with an optimization option of the highest level, -o3 (i.e. using GNU g++ compiler). This option turns on more expensive optimizations, such as function inlining, in addition to all the optimizations of the lower levels -o2 and -o1. The -o3 optimization level may increase the speed of the resulting executable, but can also increase its size.

5.2 Platform

Hardware All tests were conducted on a SUN X4600 M2 Server with four 2.6 GHz Dual-Core AMD Opteron(tm) 8218 Processors (total of eight processor cores), 32GB of RAM (64-bit word length), and four 146GB SAS disks.

Software The operating system is Redhat Linux 5.3 running kernel 2.6.18. All implementations were in C++, compiled using GNU g++ (gcc version 4.1.2) at the -O3 optimization level.

[3]Can be obtained at the bottom of `http://www.larsson.dogma.net/research.html?` from the source code section

5.3 Timing

We use a C++ library function "**gettimeofday**"[4] to measure the execution time of the algorithms. We run each algorithm 4 times against each test string. The minimum of these 4 tests was taken as the final result.

5.4 Test Data

The corpus of our test data consists of 20 files from the well-known Calgary and Canterbury corpora. The files in these corpora have being developed specifically for testing compression algorithms. They were selected based on their ability to provide representative performance results. The investigated data files are listed in Table 5.1.

5.5 Discussion of Test Results

Table 5.2 give the total runtime in microseconds/letter for LZAS variants and LZSS with window size $N = 4096$ and look ahead buffer size $F = 18$. Table 5.3 give the average runtime in microseconds/letter with the window size ranges from 256 up to 8192. Figures 5.1, 5.2 and 5.3 show the processing time verses various window sizes for LZAS1, LZAS2 and LZAS3 respectively for our test data strings. Figure 5.4 compares the behaviour of all three variants using the average processing time in milliseconds/letter. We are not measuring the memory usage since all algorithms have very small memory requirement (about 400-900 KB).

[4]include: $< sys/time.h >$

File name	Category	Size
Calgary Corpus		
bib	Bibliography (refer format)	111261
book2	Non-fiction book (troff format)	610856
news	USENET batch file	377109
paper1	Technical paper	53161
paper2	Technical paper	82199
paper3	Technical paper	46526
paper4	Technical paper	13286
paper5	Technical paper	11954
paper6	Technical paper	38105
progc	Source code in "C"	39611
progl	Source code in LISP	71646
progp	Source code in PASCAL	49379
Canterbury Corpus		
alice29.txt	English text	152089
asyoulik.txt	play Shakespeare	125179
cp.html	HTML source	24603
fields.c	C source	11150
grammar.lsp	LISP source	3721
lcet10.txt	Technical writing	426754
plrabn12.txt	Poetry	481861
xargs.1	GNU manual page	4227

Table 5.1: Description of test data

File name	LZAS1	LZAS2	LZAS3	LZSS
Calgary Corpus				
bib	54.63	**14.03**	091.23	**0.40**
book2	51.87	**13.44**	89.59	**0.40**
news	61.43	**14.40**	102.41	**0.37**
paper1	52.27	**13.33**	89.92	**0.41**
paper2	51.145	**13.65**	91.12	**0.41**
paper3	52.89	**13.61**	95.00	**0.40**
paper4	55.39	**13.93**	103.11	**0.41**
paper5	57.89	**13.90**	103.73	**0.41**
paper6	54.19	**13.44**	91.59	**0.42**
progc	54.24	**13.19**	88.86	**0.41**
progl	38.21	**11.34**	68.15	**0.45**
progp	39.05	**11.22**	69.67	**0.48**
Canterbury Corpus				
alice29.txt	50.94	**13.36**	91.99	**0.42**
asyoulik.txt	54.87	**13.97**	95.94	**0.39**
cp.html	55.98	**14.15**	91.45	**0.40**
fields.c	41.10	**11.37**	76.23	**0.45**
grammar.lsp	48.209	**11.69**	96.75	**0.43**
lcet10.txt	50.83	**13.23**	89.35	**0.41**
plrabn12.txt	55.08	**14.42**	100.43	**0.39**
xargs.1	53.14	**13.30**	111.19	**0.38**

Table 5.2: Runtime in microseconds/letter for LZAS1, LZAS2, LZAS3 and LZSS

Window Size	LZAS1	LZAS2	LZAS3
256	62.3	8.6	**8.2**
512	55.0	**9.3**	14.2
1024	46.6	**10.1**	25.8
2048	50.7	**11.3**	48.3
4096	51.7	**13.2**	91.9
8192	53.9	**16.5**	178.5

Table 5.3: Average runtime in microseconds/letter for LZAS1, LZAS2, LZAS3 over various window sizes

We make the following observations:

(1) LZSS is 30-40 times faster than LZAS2 (the fastest among LZAS variants). As we mentioned before, the experiment of scaling up an implementation of `LZSS` triggered the idea of scaling down the SA-based LZ algorithms. However, it turns out that our approach becomes uncompetitive to LZSS. Okumura implementation's [37] uses 256 binary search trees to achieve fast searches. It seems that the binary search trees are more efficient for small strings and hence work well for a sliding window approach; while suffix arrays are more efficient for long strings and hence work efficiently for a whole string.

(2) For $N = 4096$, among the LZAS variants, LZAS2 is the fastest one. This was expected as we are updating the suffix tree here rather than recomputing it (LZAS3) and we are maintaining the whole suffix array rather than just sampled values of SA/ISA (LZAS1).

(3) LZAS1 processing time per letter is relatively stable (see Figure 5.1). This probably refelcts the logarithimc factor involved in sampling.

(4) LZAS2 processing time per letter increases steadily as the window size increases (Figure 5.2) but it remains the generally the fastest. This increase is clearly logarithmic: as N varies from 2^8 to 2^{13} (a factor of 2^5), time per letter approximately doubles.

(5) LZAS3 processing time increases linearly as the window size increases. This is expected since we are recomputing SA whenever the window slides. Since the window size is increased by a factor of 2, we would expect an increase by a factor of 2 in the computation of SA as the update/recompute step is the most consuming part of our algorithm..

(6) When varying the window size, LZAS2 is still the fastest except for $N = 256$ where LZAS3 is a little bit faster. Also notice that for window sizes 256-1024 LZAS3 is faster than LZAS1 (see Figure 5.4). We can conclude that LZAS3 is better for small window sizes.

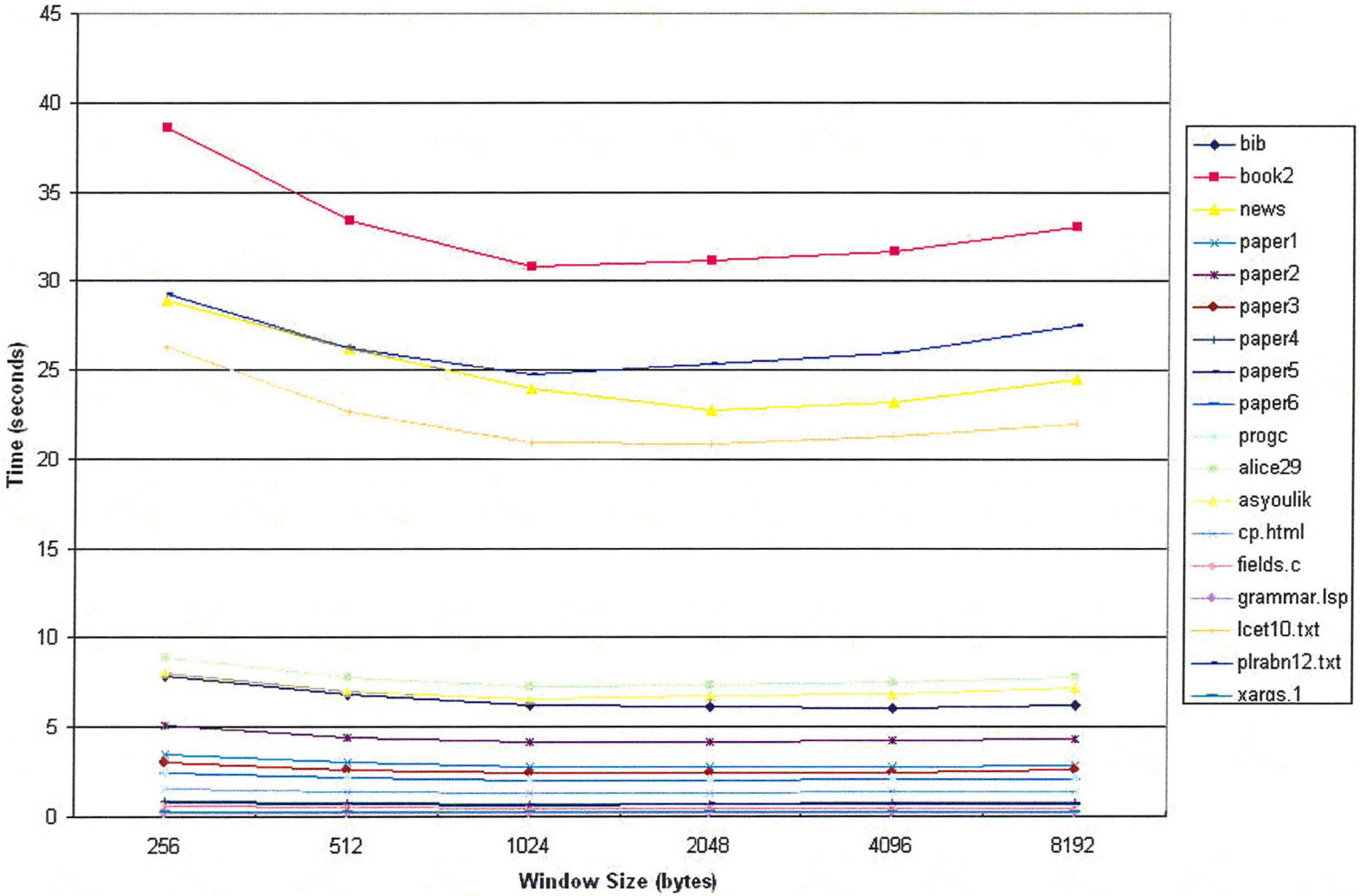

Figure 5.1: Time vs. Window Size For LZAS1

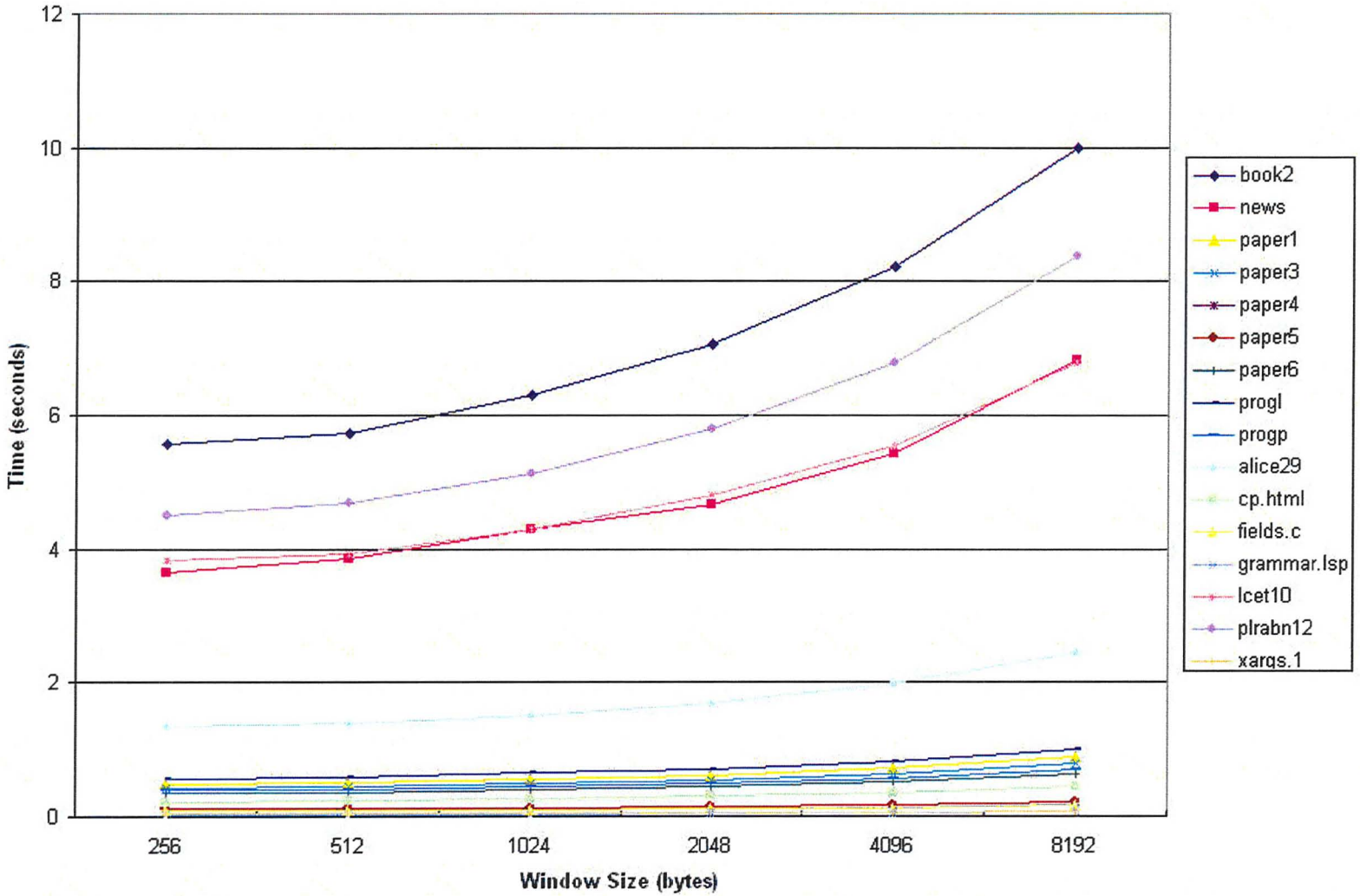

Figure 5.2: Time vs. Window Size For LZAS2

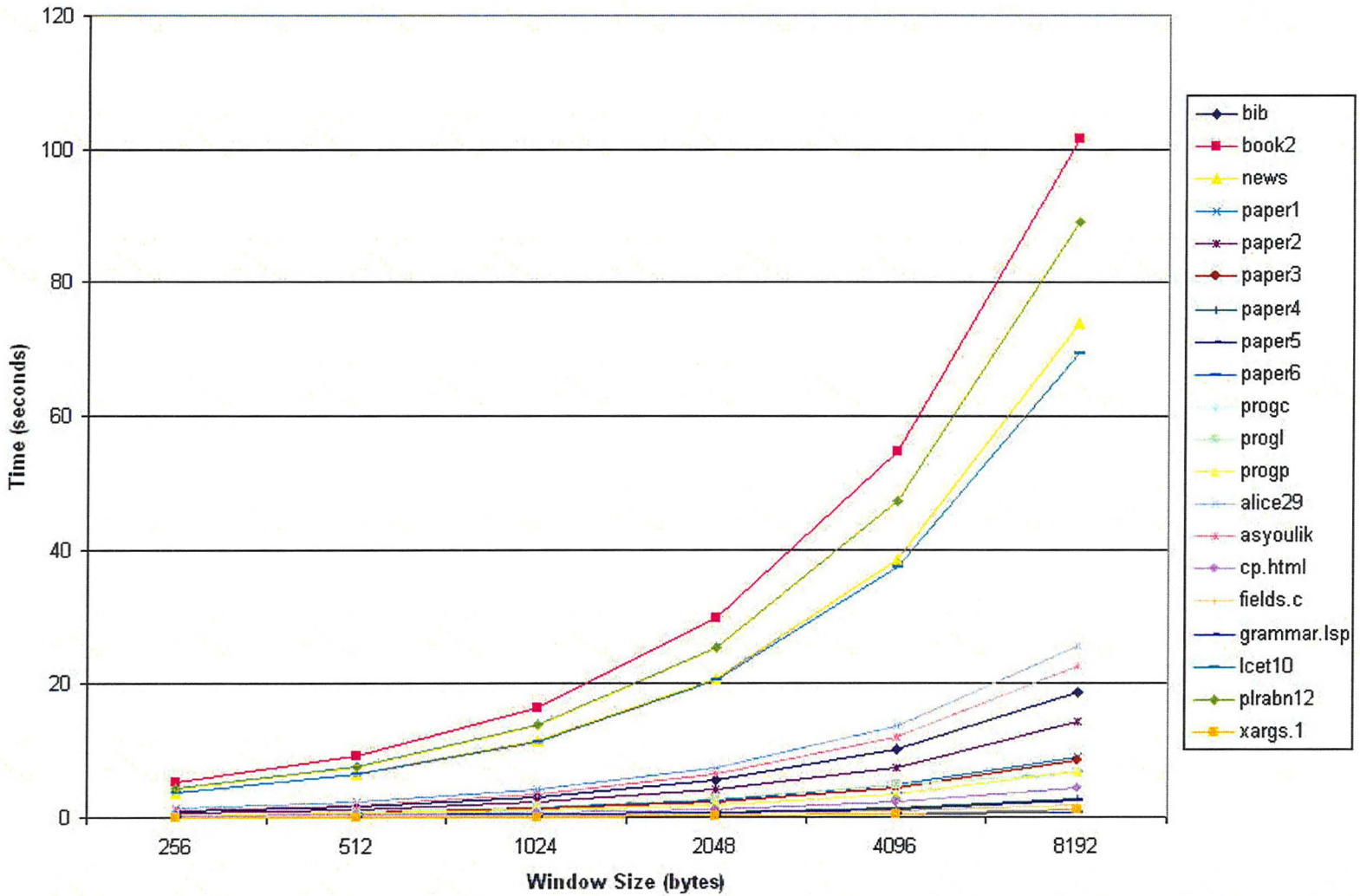

Figure 5.3: Time vs. Window Size For LZAS3

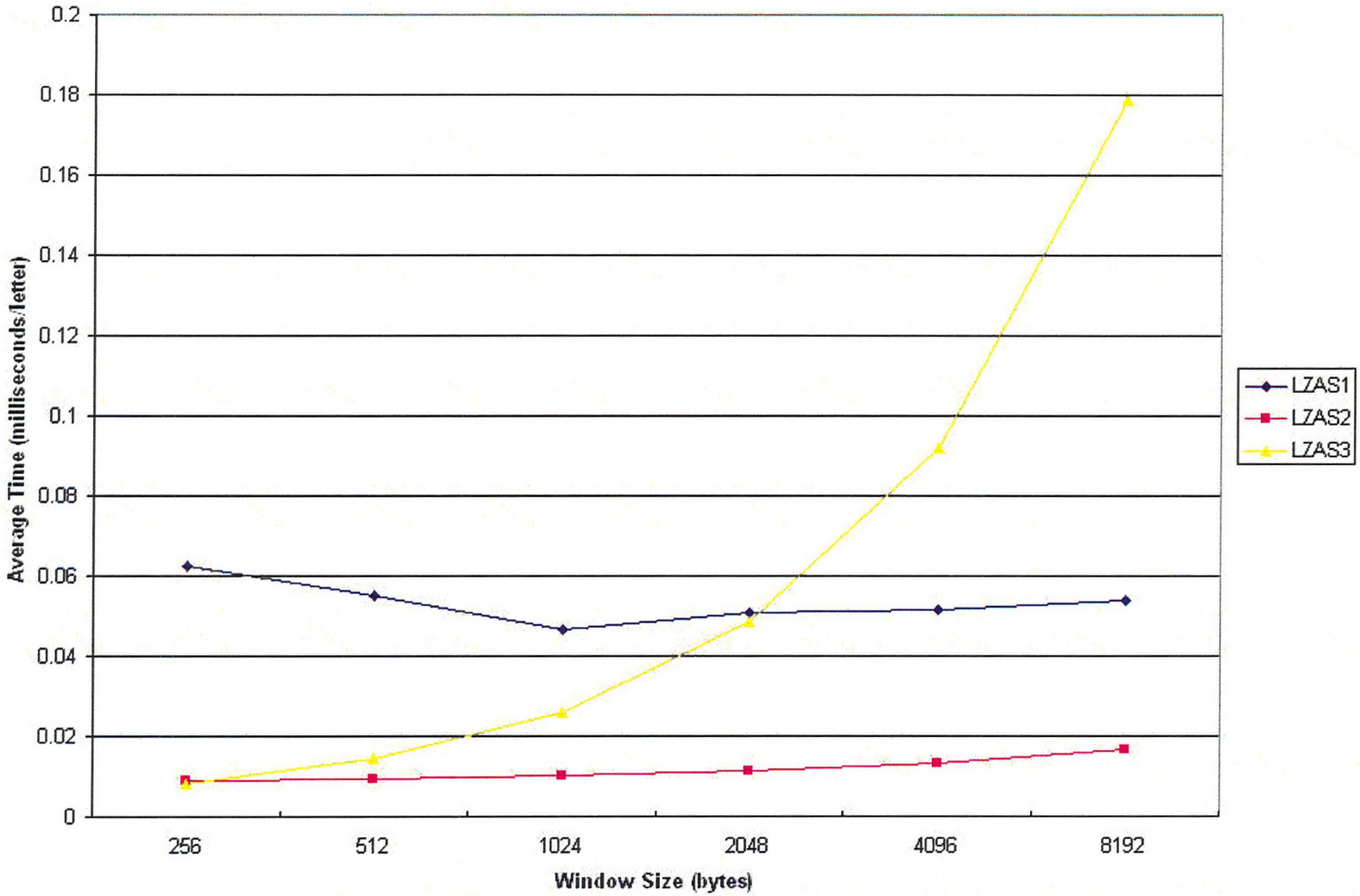

Figure 5.4: Average Time vs. Window Size For LZAS variants

Chapter 6

Conclusion and Future Work

In this thesis we have discussed the use of Lempel-Ziv factorization for dictionary-based data compression. We gave a brief explanation of the general processing in LZ77 and LZ78 to enable the reader to distinguish between the two schemes. We also discussed the process of updating the suffix array using DSA and the idea of sampling. Then we presented our new algorithm along with its variants and an interesting simple search algorithm that is used to find the longest previous match. We conducted comprehensive testing using well-known corpora that are designed for compression algorithms. We compared our algorithm variants against each other and against a previous algorithm (i.e. LZSS). Some observations were drawn from the test results.

We have discussed the updating process of the suffix array and how we can make it more efficient for the case of deleting a prefix and adding a suffix. The idea seems promising, although it might not be enough to improve our algorithm. It might be

useful for other applications.

Another improvement that can be done is related to the computation of the inverse suffix array ISA. In our search algorithm we use only one ISA value, specifically ISA$[r]$, where $r = N - F + 1$. So we do not really need to compute the whole ISA array. One solution would be to make the SACA algorithm pick up and return the value at position r that we need; alternatively, a binary search of SA would yield the same value.

Bibliography

[1] Anisa Al-Hafidh, Maxime Crochemore, Lucian Ilie, Jenya Kopylov, W. F. Smyth, German Tischler, & Munina Yusufu, **A Comparison of Lempel-Ziv LZ77 Factorization Algorithms** *in prepration.*

[2] M. I. Abouelhoda, S. Kurtz, & E. Ohlenbusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algs. 2* (2004) 53–86.

[3] Timothy C. Bell, **Better OPM/L text compression**, *IEEE Trans. Communications COM-34 (12)* (1986) 1176–1182.

[4] Jean Berstel & Alessandra Savelli, **Crochemore factorization of Sturmian and other infinite words**, *Proc. 31st Internat. Symp. Math. Foundations of Computer Sci.*, Ratislav Kralovic & Pawel Urzyczyn (eds.), LNCS 4162, Springer-Verlag (2006) 157–166.

[5] Michael Burrows & David J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Technical Report 124, Digital Equipment Corporation (1994).

[6] Maxime Crochemore, **Transducers and repetitions**, *Theoret. Comput. Sci. 45-1* (1986) 63–86.

[7] Maxime Crochemore & Lucian Ilie, **Computing longest previous factor in linear time and applications**, *Inform. Process. Lett. 106* (2008) 75–80.

[8] Maxime Crochemore, Lucian Ilie, & W. F. Smyth, **A simple algorithm for computing the Lempel-Ziv factorization**, *Proc. 18th Data Compression Conference (DCC'08)*, J. A. Storer & M. W. Marcellin (eds.) (2008) 482–488.

[9] Maxime Crochemore, Gad M. Landau & Michal Ziv-Ukelson, **A sub-quadratic sequence alignment algorithm for unrestricted cost matrices**, *Proc. 12th ACM-SIAM Symp. Discrete Algs.* (2002) 679–688.

[10] Gang Chen, Simon J. Puglisi, & W. F. Smyth, **Fast and practical algorithms for computing all runs in a string**, *Proc. 18th Annual Symp. Combinatorial*

Pattern Matching, Bin Ma & Kaizhong Zhang (eds.), LNCS 4580, Springer-Verlag (2007) 307–315.

[11] Gang Chen, Simon J. Puglisi & W. F. Smyth, **Lempel-Ziv factorization using less time & space**, *Mathematics in Computer Science 1-4*, Joseph Chan and Maxime Crochemore (eds.) (2008) 605–623.

[12] J. G. Cleary & I. H. Witten, **Data compression using adaptive coding & partial string matching**, *IEEE Transactions on Communications 32(4)* (1984) 396–4023.

[13] J. G. Cleary, W. J. Teahan & I. H. Witten, **Unbounded length contexts for PPM**, *The Computer Journal 40(2/3)* (1997) 67–75.

[14] Michael Dipperstein, *LZSS (LZ77) Discussion and Implementation*

`http://michael.dipperstein.com/lzss/`

[15] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, & A. Lefebvre, **Linear-time computation of local periods**, *Theoret. Comput. Sci. 326 (1-3)* (2004) 229–240.

[16] Fiala & Greene, **Data Compression with Finite Windows**, *CACM: Communications of the ACM 32* (1989).

[17] P. Ferragina & G. Manzini, **Opportunistic data structures with applications**, *Proc. ofthe 41st IEEE Symposium on Foundations of Computer Science* (2000) 390–398.

[18] P. Ferragina, G. Manzini, V. Makinen & G. Navarro **Compressed representation of sequences and full-text indexes**, *ACM Trans. Algo. 3* (2007) article 20.

[19] Yoko, Hidetoshi, **An improvement of dynamic huffman coding with a simple repetition finder**, *IEEE Transctions on Communications 39(1)* (1991) 8–10.

[20] Roman Kolpakov & Gregory Kucherov, **Finding maximal repetitions in a word in linear time**, *Proc. 40th Annual IEEE Symp. Found. Computer Science* (1999) 596–604.

[21] Roman Kolpakov & Gregory Kucherov, **Finding repeats with fixed gap**, *Proc. 7th International Symposium on String Processing & Information Retrieval* (2000) 162–168.

[22] Pang Ko & Srinivas Aluru, **Space efficient linear time construction of suffix arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003) 200–210.

[23] T. Kasai, G. Lee, H. Arimura, S. Arikawa & K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, Amihood Amir & Gad M. Landau (eds.), LNCS 2089, Springer-Verlag (2001) 181–192.

[24] Juha Kärkkäinen, Giovanni Manzini & Simon J. Puglisi, **Permuted longest-common-prefix array**, *Proc. 20th Annual Symp. Combinatorial Pattern Matching* (2009) to appear.

[25] Juha Kärkkäinen & Peter Sanders, **Simple linear work suffix array construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming*, LNCS 2719, Springer-Verlag (2003) 943–955.

[26] Mikaël Salson, Thierry Lecroq, Martine Léonard, & Laurent Mouchard **A four-stage algorithm for updating a Burrows-Wheeler Transform**, *Theory of Computer Science. To appear..*

[27] Mikaël Salson, Martine Léonard, Thierry Lecroq & Laurent Mouchard **Dynamic Extended Suffix Arrays**, *Journal of Discrete Algorithms. To appear..*

[28] J. N. Larsson & Kunihiko Sadakane, *Faster Suffix Sorting*, Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)], Department of Computer Science, Lund University, Sweden (1999) 20 pp.

[29] Abraham Lempel & Jacob Ziv, **On the complexity of finite sequences**, *IEEE Trans. Information Theory 22* (1976) 75–81.

[30] Mohammad Banikazemi, **LZB: Data Compression with Bounded References**, *Data Compression Conference* (2009) 436.

[31] G. Manzini, **Two space saving tricks for linear time LCP computation**, *Proc. 9th Scandinavian Workshop on Algorithm Theory*, T. Hagerup & J. Katajainen (eds.), LNCS 3111, Springer-Verlag (2004) 372–383.

[32] Giovanni Manzini & Paolo Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[33] Michael Maniscalco & Simon J. Puglisi, **Faster lightweight suffix array construction**, *Proc. 17th Australasian Workshop on Combinatorial Algs.*, Joe Ryan & Dafik (eds.) (2006) 16–29.

[34] V. S. Miller & M. N. Wegman, **Variations on a theme by Ziv and Lempel**, *Combinatorial Algorithms on Words*, A. Apostolico & Z. Galil (eds.), NATO ASI series Vol. F12, Springer-Verlag (1985) 131–140.

[35] V. Makinen & G. Navarro, **Dynamic entropy-compressed sequences and full-text indexes**, *ACM Trans. Alg. 4(3)* (2008) article 32.

[36] Mark Nelson & Jean loup Gailly, *The Data Compression Book*, M&T Books (1995) 541 pp.

[37] Haruhiko Okumura, *LZSS source code in C*

`http://www.programmersheaven.com/download/2260/4/ZipView.aspx`

[38] Pawel Pylak, **Efficient modification of LZSS compression algorithm**, *Annales UMCS Informatica AI 1* (2003) 61–72.

[39] Simon J. Puglisi & Andrew Turpin, **Space-time tradeoffs for longest-common-prefix array computation**, *Proc. 19th Internat. Symp. Algs. & Computation*, S.-H. Hong, H. Nagamochi & T. Fukunaga (eds) (2008) 124–135.

[40] Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys 39–2* (2007) Article 4, 1–31.

[41] Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.

[42] Jens Stoye & Dan Gusfield, **Simple and flexible detection of contiguous repeats using a suffix tree**, *Theoret. Comput. Sci. 279-1/2* (2002) 843–850.

[43] James A. Storer & Thomas G. Szymanski, **Data compression via textual substitution**, *J. Assoc. Comput. Mach. 29–4* (1982) 928–951.

[44] Ross N. Williams **An Extremely Fast Ziv-Lempel Data Compression Algorithm** *Data Compression Conference* (1991) 362–371.

[45] Terry A. Welch **A Technique for High Performance Data Compression** *IEEE Computer 17(6)* (1984) 8–19.

[46] Christina Zeeh, *The Lempel-Ziv Algorithm* (2003)

`http://tuxtina.de/files/seminar/LempelZiv.pdf`

[47] Jacob Ziv & Abraham Lempel, **A universal algorithm for sequential data compression**, *IEEE Trans. Information Theory 23* (1977) 337–343.

[48] Jacob Ziv & Abraham Lempel, **Compression of individual sequences via variable-rate coding**, *IEEE Trans. Information Theory 24* (1978) 530–536.