LEMPEL-ZIV FACTORIZATION USING LESS TIME AND SPACE
LEMPEL-ZIV FACTORIZATION USING LESS TIME AND SPACE

BY
GANG CHEN, B.Eng.

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE AT MCMASTER UNIVERSITY HAMILTON, ONTARIO, CANADA AUGUST 2007

© Copyright by Gang Chen, 2007
MCMASTER UNIVERSITY
DEPARTMENT OF
COMPUTING AND SOFTWARE

The undersigned hereby certify that they have read and recommend to the Faculty of Engineering for acceptance a thesis entitled "Lempel-Ziv Factorization Using Less Time and Space" by Gang Chen in partial fulfillment of the requirements for the degree of Master of Science.

Dated: August 2007

Supervisor: Dr. William F. Smyth

Readers:
MCMASTER UNIVERSITY

Date: August 2007

Author: Gang Chen

Title: Lempel-Ziv Factorization Using Less Time and Space

Department: Computing and Software

Degree: M.Sc. Convocation: October Year: 2007

Permission is herewith granted to McMaster University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLEARLY ACKNOWLEDGED.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.3 The New Algorithms</td>
<td>5</td>
</tr>
<tr>
<td>2 Definitions and Notation</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Strings and Alphabet</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Suffix Tree</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Suffix Array</td>
<td>14</td>
</tr>
<tr>
<td>2.4 Repetition and Run</td>
<td>15</td>
</tr>
<tr>
<td>2.5 Lempel-Ziv LZ-factorization</td>
<td>16</td>
</tr>
<tr>
<td>3 Previous Related Algorithms for Lempel-Ziv Construction</td>
<td>19</td>
</tr>
<tr>
<td>3.1 The Algorithm KK-LZ [22]</td>
<td>20</td>
</tr>
<tr>
<td>3.2 The Algorithm AKO [1]</td>
<td>24</td>
</tr>
<tr>
<td>4 New Algorithms</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Description of the Algorithms</td>
<td>28</td>
</tr>
<tr>
<td>4.2 Demonstration of New algorithms</td>
<td>32</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Description of the data set used in experiments. .................................. 40
5.2 Runtime in milliseconds for suffix array construction and LCP computation. .......................................................... 41
5.3 Runtime in milliseconds for various LZ factorization algorithms. Times for CPSd-9n include times for suffix sorting and LCP array construction with lcp9n; times for CPSa, CPSb, CPSc-13n and AKO include times for suffix sorting and LCP construction with lcp13n (see Table 5.2). . . . . 42
5.4 Standard deviation for runtime in milliseconds for various LZ factorization algorithms. (The standard deviation of a random variable \( X \) is defined as: \( \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2}, \) where \( \bar{x} = \frac{1}{N} \sum_{i=1}^{N} (x_i), \ x_1, ..., x_N \) are the values of the random variable \( X \), \( N \) is the number of samples. ) 43
5.5 Peak memory usage in bytes per input symbol for the LZ factorization algorithms. .......................................................... 44
# List of Figures

2.1 The trie and Patricia trie on $W=ab,abcd,efg$ .......................... 11  
2.2 The suffix trees for string $x = abaab\$ ................................ 13  
3.1 Algorithm Ukkonen: constructing a suffix tree ......................... 21  
3.2 Algorithm LZ: computing $LZ_x$ ............................................. 22  
3.3 The labelled suffix tree for $x = abaababa$ ................................ 23  
3.4 The lcp-interval tree of $x = abaababa$ .................................... 25  
3.5 Algorithm AKO: computing $LZ_x$ .............................................. 27  
4.1 Algorithm CPS: computing $LZ_x$ ............................................... 29  
4.2 Computing LEN corresponding to POS[$i$] ................................. 31  
4.3 Algorithm CPSa ................................................................. 34  
4.4 Algorithm CPSb ................................................................. 35  
4.5 Algorithm CPSc and CPSd ..................................................... 36
Abstract

For 30 years the Lempel-Ziv factorization $LZ_x$ of a string $x = x[1..n]$ has been a fundamental data structure of string processing, especially valuable for string compression and for computing all the repetitions (runs) in $x$. When the Internet came in, a huge need for Lempel-Ziv factorization was created. Nowadays it has become a basic efficient data transmission format on the Internet.

Traditionally the standard method for computing $LZ_x$ was based on $\Theta(n)$-time processing of the suffix tree $ST_x$ of $x$. Ukkonen’s algorithm constructs suffix tree online and so permits LZ to be built from subtrees of ST; this gives it an advantage, at least in terms of space, over the fast and compact version of McCreight’s STCA [37] due to Kurtz [24]. In 2000 Abouelhoda, Kurtz & Ohlebusch proposed a $\Theta(n)$-time Lempel-Ziv factorization algorithm based on an “enhanced” suffix array — that is, a suffix array $SA_x$ together with other supporting data structures.

In this thesis we first examine some previous algorithms for computing Lempel-Ziv factorization. We then analyze the rationale of development and introduce a collection of new algorithms for computing LZ-factorization. By theoretical proof and experimental comparison based on running time and storage usage, we show that our new algorithms appear either in their theoretical behavior or in practice or both to be superior to those previously proposed. In the last chapter the conclusion of our new algorithms are given, and some open problems are pointed out for our future research.
Acknowledgements

First and foremost, I would like to express my deepest gratitude to Dr. William F. Smyth, my supervisor, for his enormous support throughout the research and this thesis write-up. This thesis would not be done without his foresighted guidance and careful corrections. Many thanks for his patience and help that kept me on the right track.

I should thank Simon J. Puglisi for his constant assistance. He shared many great ideas and source codes with me, and helped me for testing. I also should thank Manzini and Kucherov for responding to my inquires.

Of course, special thanks to my parents and my whole family, who always support me and encourage me to pursue graduate study and career path. Thanks for their endless love.

Thanks to (alphabetically) Feng Wang, Feng Xie, Hao Xia, Huan Wang, Jiaping Zhu, Jie Gui, Kedong Lin, Lei Hu, Munira Yusufu, Qian Yang, Shu Wang, Wei Li, Wen Yu, Xiang Ling, Zhe Li and many others in the Computing and Software department, for their friendship.

Last but not least, thanks to my roommates, we had a lot of fun these years. They were always curious about my research and listened to my explanations patiently, although they could understand little.

Hamilton, Ontario, Canada

July, 2007

Gang Chen
Chapter 1

Introduction

In this thesis we develop a collection of new algorithms for constructing Lempel-Ziv factorization. In this introductory chapter we first give a brief overview of the background of this problem. Then we introduce the motivation for our development. Finally we detail the new features of our new algorithms.

1.1 Background

In the field of data compression, Huffman coding is optimal for a symbol-by-symbol coding with a known input probability distribution [4]. In order to obtain the necessary frequencies of symbols in the data to be compressed, Huffman coding either had to rely on the ability to predict such occurrences or would require that the text be read in beforehand.

In 1977, two Israeli information theorists, Abraham Lempel and Jacob Ziv, introduced a radically different way of compressing data, which is called the Lempel-Ziv
algorithm. This algorithm is a dictionary-based data compression technique that does not require making predictions or pre-reading data. The Lempel-Ziv code is not designed for any particular source but for a large class of sources, such as GIF image formats [5] or TIFF files. Because of these advantages, the Lempel-Ziv code became a widely used technique for lossless file compression. For example, it is used for the gzip (Unix), winzip and pkzip compression algorithms.

Two original versions of the Lempel-Ziv algorithm are described by these two theorists in [26] and [27]. After Internet technology arrived, there was a huge need for the Lempel-Ziv algorithm to compress transmission data. Since this algorithm is so simple, others can change it slightly to make it optimal for a specific use. Then LZ77 and LZ78 gave rise to a series of variants of this algorithm that form the family of LZ algorithms. The variants are essentially identical to the method from which they originate (either LZ77 or LZ78).

In this thesis we mainly discuss the algorithms for constructing Lempel-Ziv factorization. The previous traditional algorithms use a suffix tree to construct LZ-factorization. There exist many algorithms for computing suffix trees. Farach’s suffix tree construction algorithm (STCA) [9] executes in linear time, but in practice it is not as fast as Ukkonen’s algorithm [48] and McCreight’s algorithm [37] as completed by Kurtz [24]. Ukkonen’s algorithm constructs the suffix tree on-line, and so the LZ-factorization can be built at the same time. Kolpakov & Kucherov [22] described
an implementation of the LZ algorithm based on Ukkonen's algorithm, called the KK-LZ algorithm. The KK-LZ algorithm is one of the most efficient suffix tree-based LZ algorithms.

In 2004, Abouelhoda, Kurtz & Ohlebusch [1] showed how to compute LZ\(x\) from a suffix array, together with other linear structures, rather than from a suffix tree. Since there now exist practical linear-time suffix array construction algorithms (SACAs) [16, 19], it thus becomes feasible to compute LZ\(x\) in \(x\) in \(\Theta(n)\) time for large values of \(n\). In this thesis, we will compare our new algorithm with the KK-LZ [22] algorithm and the AKO algorithm [1].

1.2 Motivation

Although LZ factorization has always been of great importance in data compression [38], our immediate motivation for developing new LZ construction algorithms is the central role of LZ in computing repetitions in strings, as we now explain.

A repetition in a string \(x\) is a substring \(w = u^e\) of \(x\), with maximum \(e \geq 2\), where \(u\) is not itself a repetition in \(w\). See section 2.4 for a complete definition. A run in \(x\) is a substring \(w = u^e u^*\) of "maximal periodicity", where \(u^e\) is a repetition in \(x\) and \(u^*\) a maximum-length possibly empty proper prefix of \(u\). A run may encode as many as \(|u|\) repetitions. The maximum number of repetitions in any string \(x = x[1..n]\) is well known [6] to be \(\Theta(n \log n)\).

Computing all the runs (maximal repetitions/periodicities) in a string is one way
to list all the repetitions the string contains. Repetitions and other forms of periodicity have long been considered important theoretical characteristics of strings, and today the detection of repetitions has become of practical interest, primarily in the field of bioinformatics, with algorithms for the task a standard part of any software for whole genome analysis.

In 2000 Kolpakov & Kucherov showed that the maximum number of runs in $x$ is $O(n)$; they also described a $\Theta(n)$-time algorithm, based on Farach's $\Theta(n)$-time suffix tree construction algorithm (STCA), $\Theta(n)$-time Lempel-Ziv factorization, and Main's $\Theta(n)$-time leftmost runs algorithm, to compute all the runs in $x$.

The original motivation of our research comes from the observation that the KK algorithm [20] uses a suffix tree to compute the Lempel-Ziv factorization. The fact that a suffix tree consumes huge memory space makes the KK algorithm difficult to perform for a large string. The main idea is to replace suffix trees with enhanced suffix arrays. We can replace Farach's algorithm [20] with the AKO algorithm [1] to construct a Lempel-Ziv factorization using suffix arrays instead of suffix trees. However the AKO algorithm depends on a structure called an lcp-interval tree, which makes the AKO algorithm slow and expensive in terms of memory space. Therefore the result of this improvement is not as notable as had been expected.
We considered the Lempel-Ziv factorization carefully and discovered a more efficient version with linear time. Replacing Farach's algorithm [20] with this new algorithm can greatly improve the KK algorithm either in running time or in memory space usage. We will detail the improvements for the KK algorithm later in chapter 6. A series of new algorithms named CPSa, CPSb, CPSc and CPSd are described; each of them runs in linear time and has its own features.

1.3 The New Algorithms

In this thesis we first describe our new linear-time algorithm (CPS) that, given the suffix array and the corresponding longest common prefix array $LCP_x$, computes $LZ_x$ in guaranteed $\Theta(n)$ time and, according to our experiments, does so faster than either of the algorithms AKO [1] or KK [20]. Note however [39] that the linear-time algorithms [16, 19] for computing $SA_x$ are not, in practice, as fast as other algorithms [36, 34] that have only supralinear worst-case time bounds. Thus in testing AKO and CPS we make use of the supralinear SACA [34] that is probably at present the fastest in practice. Similarly, for testing purposes, we use an implementation of KK that, instead of Farach's algorithm, uses a fast, compact, but still supralinear version of McCreight's STCA [37] due to Kurtz [24].

In Chapter 2 we will give definitions and notation for string, suffix tree, suffix array, and Lempel-Ziv factorization. In Chapter 3 we detail some previous relevant algorithms. In Chapter 4 we describe our new algorithms CPS and its variants CPSa,
CPSb, CPSc and CPSd. We will analyze the features and the performances of these algorithms. Chapter 5 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Chapter 6 discuss our work on one of the applications of Lempel-Ziv. Finally Chapter 7 outlines our conclusions and ideas for future work.
Chapter 2
Definitions and Notation

In this chapter, we give definitions of the terminology as well as of the notation which will be used in this thesis.

2.1 Strings and Alphabet

General speaking, a string is a sequence of symbols. For example, a string might be a word, a text file, a computer program or a DNA sequence. The important feature of any string is the nature of its elements. Every element in a string is a member of a set. This set is called an alphabet. In this thesis we use some definitions in [44] and [50].

Definition 2.1.1 An alphabet $A$ is a set whose elements are called letters. Suppose $A=\{l_1, l_2, ..., l_\alpha\}$, then for any $0 \leq i \leq \alpha$, we say that $l_i$ is an element of $A$ denoted by $l_i \in A$ and $\alpha = |A|$ is the alphabet size, that is the number of all elements contained in $A$. If $\alpha$ is infinite, we say that $A$ is an infinite alphabet; otherwise, we
say that \( A \) is a finite alphabet.

For example, if we have an alphabet \( A \) such as \( A = \{a, b, c, d\} \), then we say that the alphabet size of \( A \) is 4 and it contains four elements \( a, b, c, d \). Also we say that \( \alpha = |A| = 4 \).

**Definition 2.1.2** A string \( x \) is sequence of zero or more elements drawn from an alphabet \( A \). \( |x| \) denotes the length of string \( x \).

A string can be represented by different data structures, such as an array, a linked list or a suffix tree. In this thesis we use arrays to represent strings, because arrays are simple and natural, and cost less space than linked lists and suffix trees.

For example, given a string \( x = baaabaabaabaababa \), then we can present this string as an array \( x = x[1..14] \) such as:

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline \\
x &=& b & a & a & a & b & a & a & b & a & b & a & a & a \\
\end{array}
\]

This string \( x \) is defined on an alphabet \( A = \{a, b\} \) with alphabet size \( \alpha = 2 \) and length 14. We say that \( x \) has 14 elements \( x[1] = b, x[2] = a, x[3] = a, ..., x[14] = a \) and also we can say that \( x \) has 14 positions while position 1 is at the leftmost side of \( x \) and position 14 is the rightmost side of \( x \). Taking position 8 for example, we say that \( x[8] = b \), the previous element of \( x[8] \) is \( x[7] = a \) and the next element of \( x[8] \) is \( x[9] = a \).
Definition 2.1.3 The **empty string** is a string containing zero elements and denoted by \( e \). The length of the empty string is zero. The array model works also for the empty string \( x = e \) which corresponds to an empty array and has length zero.

Definition 2.1.4 For a given string \( x = x[1..n] \), and any integers \( i \) and \( j \) that satisfy \( 1 \leq i \leq j \leq n \), we define a **substring** \( x[i..j] \) of \( x \) as follows:

\[
x[i..j] = x[i]x[i+1]...x[j]
\]

We say that \( x[i..j] \) occurs at position \( i \) of \( x \) and that it has length \( j - i + 1 \). If \( i > j \), \( x[i..j] = e \). If \( j - i + 1 < n \), then \( x[i..j] \) is called a **proper substring** of \( x \).

Definition 2.1.5 For a given string \( x = x[1..n] \), we say that \( x[1..i] \) \((0 \leq i \leq n)\) is a **prefix** of \( x \), and \( x[1..i] \) \((0 \leq i < n)\) is a **proper prefix** of \( x \). We also define that \( x[j..n] \) \((1 \leq j \leq n + 1)\) is a **suffix** of \( x \), and \( x[j..n] \) \((1 < j \leq n + 1)\) is a **proper suffix** of \( x \).

Definition 2.1.6 *(Lexicographical Order Definition)*

In practice, the order of letters is defined by the ASCII code (American Standard Code for Information Interchange). For example, \( a < b \), if and only if the ASCII code of \( a \) is less than that of \( b \).

For two strings \( x = x_1x_2...x_m \) and \( y = y_1y_2...y_n \), if \( x_i \neq y_i \), for any position \( i \) \((1 \leq i \leq m, n)\), let \( j \) be the first such \( i \). If \( x_j < y_j \), then \( x < y \), if \( x_j > y_j \), then \( x > y \). If \( x_i = y_i \) for all \( i \), then if \( m = n \), \( x = y \), while if \( m > n \), \( x > y \), and if
$m < n, \ x < y.$

2.2 Suffix Tree

Before we discuss suffix trees, we should define tries and Patricia tries.

**Definition 2.2.1** [50] Given a set $W = \{x_1, x_2, \ldots, x_m\}$ of pairwise distinct strings, a **trie** on $W$ is a tree containing exactly $m + 1$ terminal nodes, one for each $x_i$ ($i = 1, 2, \ldots, m$), plus one for $\varepsilon$. The edges of a trie are labeled with the letters that occur in the strings of $W$ plus a special end-of-string marker conventionally denoted by $. Thus the edges from the root of the trie to the terminal node for a string $x_i$ spell out $x_i$ followed by $. Each edge of the trie is labeled with a single letter of one of the strings in $W$ and no two edges out of a node can be labeled with the same letter. The use of $\$ ensures that the $m + 1$ terminal nodes are in fact leaf nodes of the structure.

Actually a trie is a search tree that is very useful in string processing. That means, given a set of strings, all the strings of this set can be retrieved by traversing the trie along the edges from the root down to the leaf nodes.

For example, suppose we have a set $W = ab, abcd, efg$, then the trie on $W$ is shown in Figure 2.1(a).

In Figure 2.1(a), we say that the trie has 4 terminal nodes from nodes 1 to 4 standing for strings $ab, abcd, efg$ and $\$ respectively. And we see that all these 4 terminal nodes are leaf nodes of the trie. On the other hand, the internal nodes of
the trie are some prefixes of some strings $x_i$. For example, in figure 2.1(a), $ab$ is a prefix of $ab\$ while $efg$ is a prefix of $efg\$. and the edge leading to any node is different from every other; that means, common prefixes (such as $ab$ or $\varepsilon$) of elements of $W$ appear only once in the trie. In this figure, we see that traversing the trie along the edges from root down to the leaf nodes exactly gets all the strings in $W$ denoted by $x_i$.

**Definition 2.2.2** [50] A Patricia trie (or compacted trie) is constructed from a trie by eliminating all internal nodes of degree 2 (those with a parent and just a single child).

The main difference between a Patricia trie and a trie is that an edge of a Patricia
trie can spell out a substring rather than a single letter. For example, we give the Patricia trie on \( W \) in Figure 2.1(b).

In Figure 2.1(b), the leaf node \( i \) (\( 1 \leq i \leq 4 \)) identifies the element \( x_i \) represented at that node.

Both tries and Patricia tries can be used to search, not only for any string in \( W \), but also for any prefix of any string in \( W \). The difference is that, in a trie, any prefix of any string in \( W \) is a node of the trie, but in a Patricia trie, the prefix may not be a node; that means, it is possible that the prefix is just “on an edge”. Thus, searching a prefix of a string in \( W \) by traversing the Patricia trie needs to visit both edges and nodes, while by traversing the trie one only needs to visit nodes.

**Definition 2.2.3** [50] Given a string \( x \) with length \( n \), suppose a set \( W \) contains \( n + 1 \) elements that are the suffixes of \( x \) including \$, then we say that the suffix tree \( T_x \) of \( x \) is the Patricia trie on \( W \).

Note that \( T_x \) contains exactly \( n + 1 \) terminal nodes and at most \( n \) internal nodes. Thus, there are at most \( 2n + 1 \) nodes and at most \( 2n \) edges in \( T_x \).

Taking string \( x = abaab \) for example, we give the suffix tree \( T_x \) in figure 2.2(a).

In Figure 2.2(a), we see that there are 6 leaf nodes in \( T_x \) and each leaf node \( i \) (\( 1 \leq i \leq 6 \)) presents the suffix \( x[i..6] \) of \( x \). And in \( T_x \), each edge labels a substring of \( x \). We can replace the substring in each edge by two integers identifying the position of the substring in \( x \). Thus, in this way, we can bound the space required for each
edge by a constant. We give the suffix tree $T_x$ in Figure 2.2(b).

In figure 2.2(b), the labels of the terminal nodes identify the positions in $x$ at which each suffix begins, while 6 denotes the empty suffix.

As we discussed in the previous chapter, suffix trees are very important intrinsic patterns in strings. Once formed, they can be used to solve many string problems. For example, given a string $x$, the suffix tree $T_x$ can be used to determine whether a given pattern $u$ is a substring of $x$; the longest repeated substring of $x$ can be represented by the deepest internal node of $T_x$ and the repetitions as well as the number of distinct substrings of $x$ also can be computed by using $T_x$. Furthermore, suffix trees are especially valuable in cases where $x$ is not subject to change, searches are frequent, and the alphabet size is very small (for example, DNA sequences).
Note that although suffix trees are very useful in string processing, the space needed by a suffix tree is very large. Thus, in order to reduce the space requirement of suffix trees, some other suffix structures such as suffix arrays have been investigated by researchers.

### 2.3 Suffix Array

**Definition 2.3.1** Given a string $x = x[1..n]$ on an alphabet $A$ of size $\alpha$, we refer to the suffix $x[i..n]$, $i \in 1..n$, simply as suffix $i$. The **suffix array** of the string $x$, or $SA_x$ is defined to be the array on $1..n$ in which $SA_x[j] = i$ iff suffix $i$ is the $j^{th}$ in lexicographical order among all the suffixes of $x$.

**Definition 2.3.2** Given two strings $x_1$ and $x_2$, the **longest common prefix** ($lcp$) of these two strings is the longest prefix of both $x_1$ and $x_2$. We denote it by $lcp(x_1, x_2)$.

Given a string $x = x[1..n]$, for the suffixes $x[i..n]$ and $x[j..n]$ ($1 \leq i, j \leq n$) of $x$, we can denote $lcp(x[i..n], x[j..n])$ by $lcp_x(i, j)$.

For example, for string $x = x[1..14],

$$
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14
\n$$

$$
x = b \ a \ a \ a \ b \ a \ a \ b \ a \ a \ b \ a
\n$$

We have $lcp(aabaababa, aababa) = lcp_x(6, 9) = aaba$, while $lcp_x(3, 7) = a$ and $lcp_x(3, 8) = \varepsilon.$
Definition 2.3.3 Let $\text{lcp}_x(i_1, i_2)$ denote the longest common prefix of suffixes $i_1$ and $i_2$ of $x$. Then $LCP_x$ is the array on $1..n+1$ in which $LCP_x[1] = LCP_x[n+1] = -1$, while for $j \in 2..n$,

$$LCP_x[j] = |\text{lcp}_x(SA_x[j-1], SA_x[j])|.$$ 

When the context is clear, we write SA for $SA_x$, LCP for $LCP_x$. For example:

```
1 2 3 4 5 6 7 8 9
x = a b a a b a b a
SA_x = 8 3 6 1 4 7 2 5
LCP_x = -1 1 1 3 3 0 2 2 -1
```

### 2.4 Repetition and Run

**Definition 2.4.1** Given a nonempty string $u$ and an integer $e \geq 2$, we say that $u^e$ represents a repetition; if $u$ itself can not be represented by a repetition, then $u^e$ is said to represent a proper repetition. Given a string $x$, a repetition in $x$ is a substring

$$x[i..i+e|u|] = u^e,$$

where $u^e$ is a proper repetition and neither $x[i+e|u|..i+(e+1)|u|] = u$ nor $x[i-|u|..i-1] = u$. Following [44], we say the repetition has generator $u$, period $|u|$, and exponent $e$; it can be specified by the integer triple $(i, |u|, e)$. 

Definition 2.4.2 [44] Let $p^*$ be the minimum period of $x = x[1..n]$, and let $r^* = n/p^*$, $u = x[1..p^*]$. Then the decomposition $x = u^{r^*}$ is called the normal form of $x$. If $r^* = 1$, we say that $x$ is primitive; otherwise, $x$ is periodic.

Definition 2.4.3 A string $u$ is a run iff it is periodic of (minimum) period $p \leq |u|/2$. A substring $u = x[i..j]$ of $x$ is a run in $x$ iff it is a run of period $p$ and neither $x[i-1..j]$ nor $x[i..j+1]$ is a run of period $p$ (i.e., it is nonextendible). The run $u$ has exponent $e = \lfloor |u|/p \rfloor$ and possibly empty tail $t = x[i+ep..j]$ (proper prefix of $x[i..i+p-1]$).

For example, for a string $x = baaabaabaababa$,

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
 x = b & a & a & a & b & a & a & b & a & b & a & b & a \\
\end{array}
\]

Then $x$ has a run $x[3..12] = (aab)^3a$ of period $p = 3$ and exponent $e = 3$ with tail $t = a$ of length $t = |t| = 1$. It can also be specified by a triple $(i, j, p) = (3, 12, 3)$, and it includes the repetitions $(aab)^3$, $(aba)^3$ and $(baa)^2$ of period $p = 3$. For the run $x[10..14] = (ab)^2a$ of period $p = 2$ and $e = 2$ with tail $t = a$ of length $t = |t| = 1$, it includes two repetitions $(ab)^2$, $(ba)^2$. In general, for $e > 2$ a run encodes $p$ repetitions; for $e = 2$, $t+1$ repetitions. Clearly, computing all the runs in $x$ specifies all the repetitions in $x$.

2.5 Lempel-Ziv LZ-factorization

There are two (non-equivalent) common definitions for LZ-factorization.
Definition 2.5.1 (Weak Definition) An LZ-factorization is a decomposition \( x = w_1 w_2 \ldots w_k \), such that each \( w_j, j \in 1..k \), is

(a) a letter that does not occur in \( x = w_1 w_2 \ldots w_{j-1} \); or otherwise

(b) the longest substring that occurs at least once in \( x = w_1 w_2 \ldots w_{j-1} \).

Definition 2.5.2 (Strong Definition) An LZ-factorization is a decomposition \( x = w_1 w_2 \ldots w_k \), such that each \( w_j, j \in 1..k \), is

(a) a letter that does not occur in \( x = w_1 w_2 \ldots w_{j-1} \); or otherwise

(b) the longest substring that occurs at least twice in \( x = w_1 w_2 \ldots w_j \).

For example, given a string \( x = ababa \),

The weak LZ-factorization is: \( a / b / ab / a / c / ba / \)

The strong LZ-factorization is: \( a / b / aba / c / ba / \)

Many previous algorithms such as Mn [29] and KK [20] use the strong definition, because the strong LZ-factorization is more efficient and practical than the weak LZ-factorization. In this thesis we use the strong definition.

The strong LZ-factorization of the example can also be represented as:

\[ w_1 = a, w_2 = b, w_3 = aba, w_4 = c, w_5 = ba \]

We specify each \( w_j \) in the LZ-factorization by pairs \((\text{POS}, \text{LEN})\), where POS is the starting position of the left occurrence of the repeating substring \( w_j \) in \( x \) and \( \text{LEN} = |w_j| \) is the substring's length. If \( w_j \) is a letter that does not occur in \( x = w_1 w_2 \ldots w_{j-1} \), then POS is the position of \( w_j \), and LEN is 0. Thus, the strong
LZ-factorization of \( x \) can be specified by
\[(1,0), (2,0), (1,3), (6,0), (2,2).\]

We observe that \( w_5 \) has two left occurrences in \( x = w_1w_2...w_j \). Since POS is not necessarily the leftmost position of occurrence, the strong LZ-factorization can also be specified by
\[(1,0), (2,0), (1,3), (6,0), (4,2).\]
Chapter 3

Previous Related Algorithms for Lempel-Ziv Construction

In order to better understand the new algorithms that we were going to develop, let us discuss previous fundamental algorithms for computing the Lempel-Ziv factorization. Generally speaking, previous algorithms can be classified into two categories, the suffix tree-based algorithm, and the suffix array-based algorithm. KK-LZ [22] is one of the most efficient suffix tree-based algorithms for LZ-factorization, because it takes advantage of Ukkonen’s algorithm to compute LZ on-line. There is only one suffix array-based algorithm for computing LZ-factorization, that is the AKO algorithm [1], which was invented in 2004. We will detail the KK-LZ and AKO algorithms in this chapter to demonstrate how Lempel-Ziv factorization is traditionally computed, and what the features of KK-LZ and AKO are. Then we can see in next chapter what the improvement of our new algorithms is.
3.1 The Algorithm KK-LZ [22]

The algorithm KK-LZ for computing LZ-factorization, is a suffix tree-based implementation of Ukkonen’s algorithm [48] by Kolpakov and Kucherov specifically designed for alphabet size $\alpha \leq 4$. Ukkonen’s algorithm is an on-line algorithm for constructing the suffix tree; it inserts the prefixes $x[i..i], i = 1, 2, ..., n$ iteratively. Therefore algorithm LZ [26] can be performed to compute the LZ-factorization at the same time that the suffix tree is constructed.

KK-LZ uses Algorithm LZ [26] to compute LZ-factorization from a suffix tree. The process is described clearly in [44]: “We specify each factor $w_j$ in the s-factorization of $x$ by a pair $(i_L, \ell)$, where $i_L$ is the starting position of the leftmost occurrence of the repeating substring $w_j$ in $x$ and $\ell = |w_j|$ is the substring’s length. Let $i_0$ be the current position in $x$ - that is, the first position following the prefix $w_1w_2...w_{j-1}$. For each $i_0$ we imagine computing the correct value of $\ell$ by searching $T_x$ from the root for the suffix $x[i_0..n]$ that will of course lead to the terminal node $i_0$. In order to search, we initialize $i \leftarrow i_0$, then match on $x[i]$, incrementing $i$ until matching starts along a downward edge of $T_x$ whose lower end is a node labeled $i_0$. If at that point the last node traversed has a nonzero label, say $v$, then we set $i_L \leftarrow v$ and set $\ell$ equal to the length of the substring represented by $v$. If the last node traversed was the root node of label 0, we set $(i_L, \ell) \leftarrow (i_0, 0)$, indicating that a new letter has been identified.” The Pseudo-code of Algorithm LZ is presented in Figure 3.2.
Ukkonen's Algorithm [48, 44]

— Compute the linked suffix tree $T_x$ on-line

construct $T_1$ — with suffix link (root) = root

$j_L \leftarrow 1$; — leaf nodes added in order 1, 2, ...

$u \leftarrow \varepsilon$; — the initial prefix node is the root

for $i \leftarrow 1$ to $n$ do — transition $i$ from $T_i$ to $T_{i+1}$

— the last branch node formed in the repeat loop must have

— its suffix link updated upon exit from the loop

$w_{prev} \leftarrow \varepsilon$

exit $\leftarrow$ FALSE

repeat — create new leaf nodes if necessary

— locate the suffix $w = w = x[j_L + 1..i]$ in $T_i$

— and update the prefix node $u$ if necessary

$(u, w) \leftarrow$ smartscan($u, v$)

if there exists no downward path labelled $x[i + 1]$ from $w$ then

— a new leaf node must be formed

$j_L \leftarrow j_L + 1$; if $j_L > i$ then exit $\leftarrow$ TRUE

— insert node $w$ on its edge if necessary

if node $w$ does not exist in $T_i$ then

create node $w$;

label downward edge to $w$

label downward edge from $w$

add leaf node $j_L$ and the "infinity edge" from $w$ to $j_L$

— set suffix link from previous node processed

if $w_{prev} \neq \varepsilon$ then $s(w_{prev}) \leftarrow w$

— set prefix node for next execution

$u \leftarrow s(u)$

else

— the suffix $j_L + 1$ in $T_i$ extends to suffix $j_L + 1$ in $T_{i+1}$,

— and so do all subsequent suffixes

— set suffix link for last node processed

if $w_{prev} \neq \varepsilon$ then

$s(w_{prev}) \leftarrow w$ — $w$ must already exist as a node

exit $\leftarrow$ TRUE

— exit if $j_L > i$ or if no more leaf nodes can be added

until exit

Figure 3.1: Algorithm Ukkonen: constructing a suffix tree
Algorithm LZ [44]

— Using the labelled suffix tree $T_x$,
— compute the s-factorization of $x$ as a sequence of pairs $(i_L, \ell)$

$i_0 \leftarrow 1$

while $i_0 \leq n$ do

$(i_i, \ell) \leftarrow \text{match}(i_0, T_x)$

output $(i_i, \ell)$

$i_0 \leftarrow i_0 + \ell$

Figure 3.2: Algorithm LZ: computing $LZ_x$

For example, a string $x = abaababa$, with s-factorization $w_1 = a, w_2 = b, w_3 = a, w_4 = aba, w_5 = ba$. That can also be represented as $a/b/a/aba/ba$. Its labelled suffix tree $T_x$ is represented in Figure 3.3. The label of the root of $T_x$ is 0, and the label of a internal node is the minimum of its children's labels. We use this example to illustrate how to compute the Lempel-Ziv factorization from a labelled suffix tree.

To compute the LZ-factorization, for the first step $i_0 = 1$, we match the suffix $x[1..8] = abaababa$ on $T_x$ to find node 1. We search $abaababa$ from the root, until we find the first node labeled 1. In Figure 3.3 we follow along the left downward edge; we find node 1, then we set $i_L = 1, \ell = 0$, because the last visited node is the root. Output $(1,0)$, and set $i_0 = i_0 + 1 = 2$.

In the second step we match $x[2..8] = baababa$ on $T_x$ to find node 2. We follow the right downward edge from the root; we find node 2, then we set $i_L = 2, \ell = 0$, because the last visited node is still the root. Output $(2,0)$, and set $i_0 = i_0 + 1 = 3$.

In the third step we match $x[3..8] = aababa$ on $T_x$ to find node 3. We follow the
left downward edge from the root to match $a$, then we follow the middle downward edge to match $ababa$; we find node 3. We set $i_L = 1, \ell = 1$, because the last visited node is 1, the length of the substring is 1. Output (1,1), and set $i_0 = i_0 + 1 = 4$.

In the fourth step we match $x[4..8] = ababa$ on $T_x$ to find node 4. We follow the left downward edge from the root to match $a$, then we follow the right edge to match $ba$, next we still follow the right edge to match $ba$, then we find node 4. We set $i_L = 1, \ell = 3$, because the last visited node is 1, the length of the substring is 3. Output (1,3), and set $i_0 = i_0 + 3 = 7$.

In the fifth step we match $x[7..8] = ba$ on $T_x$ to find node 7. We follow the right downward edge from the root to match $ba$, then we follow the left edge; we find node

Figure 3.3: The labelled suffix tree for $x = abaababa$
7. We set $i_L = 2, \ell = 2$, because the last visited node is 2, the length of the substring is 2. Output $(2,2)$, and set $i_0 = i_0 + 2 = 9$. We finish here.

The output of the algorithm would be $(1,0), (2,0), (1,1), (1,3), (2,2)$. 

### 3.2 The Algorithm AKO [1]

The central concept of AKO algorithm is to use a tree structure, namely the lcp-interval tree. The lcp-interval tree can be generated from a LCP array in linear time, then from lcp-intervals it can easily compute the Lempel-Ziv factorization.

**Definition 3.2.1** [1] *Given a string $x = x[1..n]$. An interval $[i..j]$, $0 \leq i < j \leq n$, is an lcp-interval of lcp-value $\ell$ if*

1. $LCP_x[i] < \ell$, 
2. $LCP_x[k] \geq \ell$ for all $k$ with $i + 1 \leq k \leq j$,
3. $LCP_x[k] = \ell$ for at least one $k$ with $i + 1 \leq k \leq j$,
4. $LCP_x[j + 1] < \ell$.

We can use $\ell - [i..j]$ to represent an lcp-interval $[i..j]$ of lcp-value $\ell$.

We still use the example $x = abaababa$. Its suffix array, LCP array, and lcp-interval tree are shown in the following figures. Comparing Figure 3.3 with Figure
3.4, we can see that the lcp-interval tree is similar to its suffix tree.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\[
x = a \ b \ a \ a \ b \ a \ b \ a
\]

\[
SA_x = 8 \ 3 \ 6 \ 1 \ 4 \ 7 \ 2 \ 5
\]

\[
LCP_x = -1 \ 1 \ 3 \ 3 \ 0 \ 2 \ 2 \ -1
\]

![Diagram of the lcp-interval tree of x = abaababa]

Figure 3.4: The lcp-interval tree of \( x = abaababa \)

An lcp-interval is a node in the lcp-interval tree. All lcp-intervals are computed with the help of a stack, the elements of which are lcp-intervals represented by tuples \((lcp, lb, rb, childList, min)\), where \(lcp\) is the lcp-value of the interval, \(lb\) is its left boundary, \(rb\) is its right boundary, \(childList\) is a list of its child intervals, and \(min\) is the minimum value of suffix array \(SA_x[lb..rb]\). The Lempel-Ziv factorization is represented by arrays \(POS[0..n-1]\) and \(LEN[0..n-1]\) (Definition 2.5.2). Furthermore, \(\perp\) stands for an undefined value, \([\ ]\) stands for an empty list, \(add([c_1, ..., c_k], c)\) appends the element \(c\) to the list\([c_1, ..., c_k]\) and returns the result.
All lcp-intervals can be bottom-up generated. Every time an lcp-interval is generated, it calls a procedure to update POS and LEN arrays. Lempel-Ziv factorization can be computed on-line in linear time by a bottom-up construction of the lcp-interval tree. The pseudo-code of this process is represented in Figure 3.5.

Because of the use of the lcp-interval tree structure, the AKO algorithm is not a pure suffix array-based algorithm for LZ-factorization construction. The AKO algorithm computes the suffix array and LCP array from a string, and then constructs an lcp-interval tree. Since an lcp-interval tree costs nearly the same space as a suffix tree, the performance of AKO is not better than a suffix tree-based algorithm in terms of time and space. However the AKO algorithm illustrates that a suffix array of a string, together with its LCP array, has enough information for computing its LZ-factorization. Actually it proves that every algorithm using a suffix tree can systematically be replaced by an algorithm using an enhanced suffix array (i.e., a suffix array enhanced with the LCP array).
Algorithm AKO [1]

\[
\text{lastInterval} := \bot
\]
\[
\text{push}((0, 0, \bot, [], \bot))
\]
\[\text{for } i := 1 \text{ to } n \text{ do} \]
\[
\text{lb} := i - 1
\]
\[\text{while } \text{LCP}[i] < \text{top.lcp} \]
\[
\text{top.rb} := i - 1;
\]
\[
\text{lastInterval} := \text{pop}
\]
\[
\text{process(lastInterval)}
\]
\[
\text{lb} := \text{lastInterval.lb}
\]
\[\text{if } \text{LCP}[i] \leq \text{top.lcp} \text{ then} \]
\[
\text{top.childList} := \text{add(top.childList, lastInterval)}
\]
\[
\text{lastInterval} := \bot
\]
\[\text{if } \text{LCP}[i] > \text{top.lcp} \text{ then} \]
\[\text{if } \text{lastInterval} \neq \bot \text{ then} \]
\[
\text{push}((\text{LCP}[i], \text{lb}, \bot, [], \text{lastInterval}, \bot))
\]
\[
\text{lastInterval} := \bot
\]
\[\text{else push}((\text{LCP}[i], \text{lb}, \bot, [], \bot))
\]

\begin{algorithm}
\begin{algorithmic}
\Function{process}{lastInterval}
\State \( i \leftarrow \text{lastInterval.lb}; \)
\State \( j \leftarrow \text{lastInterval.rb}; \)
\State \( M = \{\text{SA}[q] \mid q \in [i..j]\} \)
\State \( \text{lastInterval.min} \leftarrow \min(M) \)
\State \For {all } \( p \in M \text{ with } q \neq \text{min}; \)
\State \( \text{POS}[p] \leftarrow \text{min} \)
\State \( \text{LEN}[p] \leftarrow \text{lastInterval.lcp} \)
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

Figure 3.5: Algorithm AKO: computing LZx
Chapter 4

New Algorithms

4.1 Description of the Algorithms

Given a string $x = x[1..n]$ on an alphabet $A$ of size $\alpha$, its $SA_x$ and $LCP_x$ can be computed in $\Theta(n)$ time [18, 35]. For example:

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
x = & a & b & a & a & b & a & b & a \\
SA_x = & 8 & 3 & 6 & 1 & 4 & 7 & 2 & 5 \\
LCP_x = & -1 & 1 & 3 & 3 & 0 & 2 & 2 & -1
\end{array}
\]

We use the strong LZ-factorization definition (Definition 2.5.2). For our example string, $x = w_1 w_2 \ldots w_k$, where $w_1 = a$, $w_2 = b$, $w_3 = a$, $w_4 = aba$, $w_5 = ba$. Typically, integer pairs (POS, LEN) specify the factorization, where POS gives a position in $x$ and LEN the corresponding length at that position (by convention zero if the position contains a “new” letter). The example thus yields (POS, LEN) = (1, 0), (2, 0), (3, 1), (4, 3), (7, 2). As mentioned above, $LZ_x$ can be quickly computed
from ST in $\Theta(n)$ time [26], also from SA [1]. Our new algorithm is displayed in Figure 4.1.

**Algorithm CPS**

--- Using SA and LCP, compute POS[1..n] and LEN[1..n].

\[ i_1 \leftarrow 1; \ i_2 \leftarrow 2; \ i_3 \leftarrow 3 \]

**while** $i_3 \leq n+1$

--- Identify the next position $i_2 < i_3$ with LCP[$i_2$] > LCP[$i_3$].

**while** LCP[$i_2$] $\leq$ LCP[$i_3$] **do**

- push($S$, $i_1$); $i_1 \leftarrow i_2$; $i_2 \leftarrow i_3$; $i_3 \leftarrow i_3 + 1$

--- Backtrack using the stack $S$ to locate the first $i_1 < i_2$ such that

--- LCP[$i_1$] < LCP[$i_2$], at each step setting the larger position in POS

--- corresponding to equal LCP to point leftwards to the smaller one,

--- if it exists; if not, then POS[$i$] $\leftarrow i$.

\[ p_2 \leftarrow SA[i_2]; \ \ell_2 \leftarrow LCP[i_2] \]

assign(POS, LEN, $p_1, p_2$)

**while** LCP[$i_1$] $=$ $\ell_2$ **do**

- $i_1 \leftarrow \text{pop}(S)$
- assign(POS, LEN, $p_1, p_2$)

SA[$i_1$] $\leftarrow p_2$

--- Reset pointers for the next stage.

**if** $i_1 > 1$ **then**

- $i_2 \leftarrow i_1$; $i_1 \leftarrow \text{pop}(S)$

**else**

- $i_2 \leftarrow i_3$; $i_3 \leftarrow i_3 + 1$

--- procedure **assign**(POS, LEN, $p_1, p_2$)

\[ p_1 \leftarrow SA[i_1] \]

**if** $p_1 < p_2$ **then**

- POS[$p_2$] $\leftarrow p_1$; LEN[$p_2$] $\leftarrow \ell_2$; $p_2 \leftarrow p_1$

**else**

- POS[$p_1$] $\leftarrow p_2$; LEN[$p_1$] $\leftarrow \ell_2$

--- Figure 4.1: Algorithm CPS: computing LZ

The basic strategy of CPS is first to locate, in a left-to-right traversal of SA, a next position $i_2$ such that LCP[$i_2$] > LCP[$i_3$] for some least $i_3 > i_2$; then second to backtrack (using the stack $S$) from $i_2$, setting POS[$p_2$] $\leftarrow p_1$ or POS[$p_1$] $\leftarrow p_2$
depending on whether \( p_1 = SA[i_1] < p_2 \) or not, until the LCP value for the position \( i_1 \) popped from \( S \) falls below \( LCP[i_2] \). This processing does not guarantee that, for equal LCP (LEN), each corresponding position in POS necessarily points to the leftmost occurrence in \( \mathbf{x} \), as normally required for LZ decomposition; however, the Main and KK algorithms do not require this property for their correct functioning, they require only that each position in POS should point left. In other terminology, what is in fact computed by CPS is a quasi suffix array (QSA) [12]. We call the algorithm of Figure 4.2 CPSa.

Now observe that none of the position pointers \( i_1, i_2, i_3 \) will ever point to any position \( i \) in \( SA \) such that \( POS[SA[i]] \) has been previously set. It follows that the storage for \( SA \) and LCP can be dynamically reused to specify the location and contents of the array POS, thus saving \( 4n \) bytes of storage — neither the Main nor the KK algorithm requires \( SA/LCP \). In Figure 4.2 this is easily accomplished by inserting \( i_2 \leftarrow i_1 \) at the beginning of the second inner \textbf{while} loop, then replacing

\[
\begin{align*}
\text{POS}[p_2] &\leftarrow p_1 \quad \text{by} \quad \text{SA}[i_2] \leftarrow p_2; \quad \text{LCP}[i_2] \leftarrow p_1 \\
\text{POS}[p_1] &\leftarrow p_2 \quad \text{by} \quad \text{SA}[i_2] \leftarrow p_1; \quad \text{LCP}[i_2] \leftarrow p_2
\end{align*}
\]

POS can then be computed by a straightforward in-place compactification of \( SA \) and LCP into LCP (now redefined as POS). We call this second algorithm CPSb.

But more storage can be saved. Remove all reference to LEN from CPSb, so that it computes only POS and in particular allocates no storage for LEN. Then, after
POS is computed, the space previously required for LCP becomes free and can be reallocated to LEN. Observe that only those positions in LEN that are required for the LZ-factorization need to be computed, so that the total computation time for LEN is $\Theta(n)$. In fact, without loss of efficiency, we can avoid computing LEN as an array and compute it only when required; given a sentinel value $\text{POS}[n+1] = $, the simple function of Figure 4.2 computes LEN corresponding to POS[i]. We call the third version CPSc.

Function LEN for CPSc

```
function LEN(x, POS, i)
  j <- POS[i]
  if j = i then
    LEN <- 0
  else
    $\ell$ <- 1
    while $x[i+\ell] = x[j+\ell]$ do
      $\ell$ <- $\ell$ + 1
    LEN <- $\ell$
```

Figure 4.2: Computing LEN corresponding to POS[i]

Since at least one position in POS is set at each stage of the main while loop, it follows that the execution time of CPS is linear in $n$. For CPSa space requirements total 17n bytes (for x, SA, LCP, POS & LEN) plus 4s bytes for a stack of maximum size $s$ — at most the maximum depth of STx. For $x = a^n$, $s = n$, but in the expected case, $s \in O(\log_a n)$ [17]. For CPSb the space required is 13n bytes. However, CPSc can be handled in two different ways, so that in fact two new variants, CPSc and
CPSd, are introduced. As we shall see, CPSc is faster than CPSd, but requires more space.

Observe that for CPSa and CPSb the original (and somewhat faster) method [18] for computing LCP can be used, since it requires $13n$ bytes of storage, not greater than the total space requirements of these two variants. However, to achieve $9n$ bytes of storage, the Manzini variant [35] for computing LCP must be used, that leads to the variant CPSd. In fact, thus CPSc requires $13n$ bytes including the stack, while CPSd requires $9n$ bytes plus stack. The difference between CPSc and CPSd is that CPSc uses the original LCP calculation [18] (and therefore requires no additional space for the stack), and CPSd uses the Manzini variant.

4.2 Demonstration of New algorithms

In this section we use an example to demonstrate our algorithms clearly.

Given a string $x = abababa$ of length 8, and its corresponding suffix and LCP arrays, the goal of algorithm CPS is to compute its LZ-factorization. In Figures 4.3, 4.4 and 4.5, we respectively show how algorithm CPSa, CPSb, and CPSc work.

In Figure 4.3 we can observe that the shaded positions in SA and LCP array will never be used. Therefore the difference between CPSb and CPSa is to reuse these positions. The SA and LCP arrays can be reused as POS array.

Algorithm CPSa and CPSb both compute POS and LEN arrays. Using POS and LEN arrays we can easily compute LZ-factorization. However during the process
for computing LZ-factorization, we can compute LEN array from POS and string \( x \) (Figure 4.2) when required. Therefore algorithm CPSc does not compute LEN array.

The process for computing LZ-factorization using POS and LEN arrays is

\[
i = 1, \text{ output } (\text{POS}[1], \text{LEN}[1])=(1,0), \text{ then } i = i + 1 = 2;
\]

\[
i = 2, \text{ output } (\text{POS}[2], \text{LEN}[2])=(2,0), \text{ then } i = i + 1 = 3;
\]

\[
i = 3, \text{ output } (\text{POS}[3], \text{LEN}[3])=(1,1), \text{ then } i = i + 1 = 4;
\]

\[
i = 4, \text{ output } (\text{POS}[4], \text{LEN}[4])=(1,3), \text{ then } i = i + 3 = 7;
\]

\[
i = 7, \text{ output } (\text{POS}[7], \text{LEN}[7])=(2,2), \text{ then } i = i + 2 = 9 = n + 1; \text{ stop}
\]

Finally we get LZ-factorization \((1,0), (2,0), (1,1), (1,3), (2,2)\).
Finally we get LEN and POS arrays:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>POS</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>LEN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.3: Algorithm CPSa


after in-place rearrangement, it can be reused as POS array

we also get LEN arrays:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.4: Algorithm CPSb
Figure 4.5: Algorithm CPSc and CPSd
Chapter 5

Experiments

As discussed in Chapter 3, LZ-factorization algorithms can be classified into two categories, according to whether they use suffix arrays and using suffix trees. Currently algorithm AK0 of [AKO04] is an efficient LZ-factorization algorithm using suffix arrays, and algorithm KK-LZ of [KK99] is an efficient LZ-factorization algorithm using suffix trees. In this chapter we compare our new algorithm CPS with the algorithms AK0 and KK-LZ.

We implemented the four versions of CPS described above; we call them CPSa, CPSb, CPSc (13n-byte LCP calculation), and CPSd (9n-byte LCP calculation). For CPSc, the original (and somewhat faster) method [18] for computing LCP is used. For CPSd, to achieve 9n bytes of storage, the Manzini variant [35] for computing LCP is used. We also implemented the other SA-based LZ-factorization algorithm, AK0 of [AKO04]. The implementation KK-LZ of Kolpakov and Kucherov’s algorithm was obtained from [KK99]. All programs were written in C or C++. We are confident
that all implementations tested are of high quality.

5.1 Testing Details

5.1.1 Environment

All experiments were conducted on an application server machine (Moore) with 4 AMD Opteron 2.6GHz CPUs and 8GB memory in total. When executing the testing program one CPU is used. The operating system was RedHat Linux. The compiler was g++ (gcc version 4.1.1) executed with the -O3 option.

5.1.2 Timing

Times were recorded with the standard C getrusage function. All running times given are the minimum of 10 runs, and do not include time spent reading input files. This is represented in Table 5.3. Since our program is executed on a multi-process server machine, it can be interfered with by other programs or a missed cache. Therefore the minimum running time reflects the actual running time better than the average running time. We also compute the standard deviation of the running times to show how widely spread the running times are, and the results are in Table 5.4.

5.1.3 Memory usage

Memory usage was recorded with the memusage command available with most Linux distributions. The peak of memory usage was recorded for each run (Table 5.5).
5.1.4 Test Data

We test the programs on various data files, which are described in Table 5.1. File chr22 and chr1819 was originally on an alphabet of five symbols A,C,G,T,N but was reduced by replacing occurrences of N with random selection of the other four symbols. The N's represent ambiguities in the sequencing process.

5.2 Test Results

Times for the CPS implementations and AKO include time required for SA and LCP array construction. The implementation of KK-LZ is only suitable for strings on small alphabets ($|\Sigma| \leq 4$) so times are only given for some of the files. Results are not given for AKO on other files because the memory required exceeded the capacity of the test machine.

5.3 Conclusions of Experiments

Time spent computing the suffix array hurts the CPSa-d and AKO algorithms, as which can be observed from Table 5.2. We conclude:

(1) KK remains the algorithm of choice for DNA strings of moderate size.

(2) For other strings encountered in practice, CPSb is consistently faster than AKO except for very large alphabets; it also uses substantially less space, especially on run-rich strings.
Table 5.1: Description of the data set used in experiments.

<table>
<thead>
<tr>
<th>String</th>
<th>Size (bytes)</th>
<th>( \Sigma )</th>
<th># factors</th>
<th>max</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibo35</td>
<td>9227465</td>
<td>2</td>
<td>34</td>
<td>3524578</td>
<td>35th Fibonacci string (see [SMY03])</td>
</tr>
<tr>
<td>fibo36</td>
<td>14930352</td>
<td>2</td>
<td>35</td>
<td>5702887</td>
<td>36th Fibonacci string</td>
</tr>
<tr>
<td>fss9</td>
<td>2851443</td>
<td>2</td>
<td>40</td>
<td>1217712</td>
<td>9th run rich string of [FSS03]</td>
</tr>
<tr>
<td>fss10</td>
<td>12078908</td>
<td>2</td>
<td>44</td>
<td>5158310</td>
<td>10th run rich string of [FSS03]</td>
</tr>
<tr>
<td>random2</td>
<td>8388608</td>
<td>2</td>
<td>385232</td>
<td>42</td>
<td>Random string, small alphabet</td>
</tr>
<tr>
<td>random21</td>
<td>8388608</td>
<td>21</td>
<td>1835235</td>
<td>9</td>
<td>Random string, larger alphabet</td>
</tr>
<tr>
<td>ecoli</td>
<td>4638690</td>
<td>4</td>
<td>432791</td>
<td>2805</td>
<td>E. Coli Genome</td>
</tr>
<tr>
<td>chr22</td>
<td>34553758</td>
<td>4</td>
<td>2554184</td>
<td>1768</td>
<td>Human Chromosome 22</td>
</tr>
<tr>
<td>chr19</td>
<td>63811651</td>
<td>4</td>
<td>4411679</td>
<td>3397</td>
<td>Human Chromosome 19</td>
</tr>
<tr>
<td>chr1819</td>
<td>139928804</td>
<td>4</td>
<td>9560771</td>
<td>3397</td>
<td>Human Chromosomes 18 &amp; 19</td>
</tr>
<tr>
<td>prot-a</td>
<td>16777216</td>
<td>23</td>
<td>2751022</td>
<td>6699</td>
<td>Small Protein dataset</td>
</tr>
<tr>
<td>prot-b</td>
<td>33554432</td>
<td>24</td>
<td>5040051</td>
<td>16190</td>
<td>Medium Protein dataset</td>
</tr>
<tr>
<td>prot-c</td>
<td>67108864</td>
<td>24</td>
<td>8391184</td>
<td>16190</td>
<td>Large Protein dataset</td>
</tr>
<tr>
<td>bible</td>
<td>4047392</td>
<td>62</td>
<td>337558</td>
<td>549</td>
<td>King James Bible</td>
</tr>
<tr>
<td>howto</td>
<td>39422105</td>
<td>197</td>
<td>3063929</td>
<td>70718</td>
<td>Linux Howto files</td>
</tr>
<tr>
<td>mozilla</td>
<td>51129180</td>
<td>256</td>
<td>5656068</td>
<td>3317</td>
<td>Mozilla binaries</td>
</tr>
<tr>
<td>rfc</td>
<td>116421901</td>
<td>120</td>
<td>5656068</td>
<td>3317</td>
<td>IETF Request for comments</td>
</tr>
</tbody>
</table>

(3) Overall, and especially for strings on alphabets of size greater than 4, CPSd(9n) is probably preferable since it will be more robust for main-memory use on very large strings: its storage requirement is consistently low (about half that of AKO, including on DNA strings) and it is only 25-30% slower than CPSb.

(4) The results in Table 5.4 demonstrate that the standard deviations of running times are small with respect to the average. Therefore we are confident in the validity of the timing results.
Table 5.2: Runtime in milliseconds for suffix array construction and LCP computation.

<table>
<thead>
<tr>
<th>String</th>
<th>saca</th>
<th>lcp13n</th>
<th>lcp9n</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibo35</td>
<td>10852</td>
<td>4347</td>
<td>5810</td>
</tr>
<tr>
<td>fibo36</td>
<td>19253</td>
<td>7310</td>
<td>10166</td>
</tr>
<tr>
<td>fss9</td>
<td>2921</td>
<td>1267</td>
<td>1534</td>
</tr>
<tr>
<td>fss10</td>
<td>15346</td>
<td>5891</td>
<td>7047</td>
</tr>
<tr>
<td>rand2</td>
<td>5542</td>
<td>3347</td>
<td>5465</td>
</tr>
<tr>
<td>rand21</td>
<td>6110</td>
<td>5369</td>
<td>6734</td>
</tr>
<tr>
<td>ecoli</td>
<td>3871</td>
<td>3136</td>
<td>3563</td>
</tr>
<tr>
<td>chr22</td>
<td>29245</td>
<td>22543</td>
<td>26132</td>
</tr>
<tr>
<td>chr19</td>
<td>65379</td>
<td>58430</td>
<td>65137</td>
</tr>
<tr>
<td>chr1819</td>
<td>173452</td>
<td>152060</td>
<td>199294</td>
</tr>
<tr>
<td>prot-a</td>
<td>14218</td>
<td>12576</td>
<td>15733</td>
</tr>
<tr>
<td>prot-b</td>
<td>36725</td>
<td>32118</td>
<td>37632</td>
</tr>
<tr>
<td>prot-c</td>
<td>49326</td>
<td>45321</td>
<td>59596</td>
</tr>
<tr>
<td>bible</td>
<td>2225</td>
<td>2004</td>
<td>2386</td>
</tr>
<tr>
<td>howto</td>
<td>23187</td>
<td>22573</td>
<td>29697</td>
</tr>
<tr>
<td>mozilla</td>
<td>28213</td>
<td>29572</td>
<td>37439</td>
</tr>
<tr>
<td>rfc</td>
<td>84497</td>
<td>82268</td>
<td>131404</td>
</tr>
</tbody>
</table>
Table 5.3: Runtime in milliseconds for various LZ factorization algorithms. Times for CPSd-9n include times for suffix sorting and LCP array construction with lcp9n; times for CPSa, CPSb, CPSc-13n and AKO include times for suffix sorting and LCP construction with lcp13n (see Table 5.2).

<table>
<thead>
<tr>
<th>String</th>
<th>CPSa</th>
<th>CPSb</th>
<th>CPSc</th>
<th>CPSd</th>
<th>AKO</th>
<th>KK-LZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibo35</td>
<td>17347</td>
<td>16321</td>
<td>17160</td>
<td>18623</td>
<td>23839</td>
<td>19033</td>
</tr>
<tr>
<td>fibo36</td>
<td>30273</td>
<td>26017</td>
<td>29176</td>
<td>32032</td>
<td>44146</td>
<td>30125</td>
</tr>
<tr>
<td>fss9</td>
<td>4651</td>
<td>4256</td>
<td>4478</td>
<td>4745</td>
<td>5922</td>
<td>2310</td>
</tr>
<tr>
<td>fss10</td>
<td>25412</td>
<td>23835</td>
<td>25090</td>
<td>26246</td>
<td>31041</td>
<td>15455</td>
</tr>
<tr>
<td>rand2</td>
<td>14688</td>
<td>13424</td>
<td>14165</td>
<td>16283</td>
<td>20335</td>
<td>19713</td>
</tr>
<tr>
<td>rand21</td>
<td>16134</td>
<td>14235</td>
<td>14870</td>
<td>16235</td>
<td>20176</td>
<td></td>
</tr>
<tr>
<td>ecoli</td>
<td>9452</td>
<td>9147</td>
<td>9336</td>
<td>9763</td>
<td>13245</td>
<td>3935</td>
</tr>
<tr>
<td>chr22</td>
<td>83560</td>
<td>79265</td>
<td>82418</td>
<td>86007</td>
<td>120239</td>
<td>31254</td>
</tr>
<tr>
<td>chr19</td>
<td>158613</td>
<td>152520</td>
<td>163653</td>
<td>170362</td>
<td></td>
<td>87842</td>
</tr>
<tr>
<td>chr1819</td>
<td>483954</td>
<td>461245</td>
<td>461418</td>
<td>508652</td>
<td></td>
<td>263135</td>
</tr>
<tr>
<td>prot-a</td>
<td>30836</td>
<td>30544</td>
<td>33368</td>
<td>36525</td>
<td>38233</td>
<td></td>
</tr>
<tr>
<td>prot-b</td>
<td>73478</td>
<td>71105</td>
<td>74214</td>
<td>79731</td>
<td>85790</td>
<td></td>
</tr>
<tr>
<td>prot-c</td>
<td>158712</td>
<td>143825</td>
<td>167036</td>
<td>181311</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bible</td>
<td>6656</td>
<td>5867</td>
<td>6749</td>
<td>7131</td>
<td>7832</td>
<td></td>
</tr>
<tr>
<td>howto</td>
<td>66922</td>
<td>65579</td>
<td>67577</td>
<td>72701</td>
<td>65165</td>
<td></td>
</tr>
<tr>
<td>mozilla</td>
<td>81745</td>
<td>80625</td>
<td>82058</td>
<td>89925</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rfc</td>
<td>218405</td>
<td>201305</td>
<td>220196</td>
<td>269332</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.4: Standard deviation for runtime in milliseconds for various LZ factorization algorithms. (The standard deviation of a random variable $X$ is defined as: $\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2}$, where $\bar{x} = \frac{1}{N} \sum_{i=1}^{N} (x_i)$, $x_1, \ldots, x_N$ are the values of the random variable $X$, $N$ is the number of samples.)

<table>
<thead>
<tr>
<th>String</th>
<th>CPSa</th>
<th>CPSb</th>
<th>CPSc</th>
<th>CPSd</th>
<th>AKO</th>
<th>KK-LZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibo35</td>
<td>50.6</td>
<td>50.5</td>
<td>50.8</td>
<td>51.9</td>
<td>64.5</td>
<td>52.4</td>
</tr>
<tr>
<td>fibo36</td>
<td>95.4</td>
<td>95.4</td>
<td>96.4</td>
<td>98.5</td>
<td>125.6</td>
<td>97.8</td>
</tr>
<tr>
<td>fss9</td>
<td>15.3</td>
<td>15.6</td>
<td>16.6</td>
<td>17.5</td>
<td>19.5</td>
<td>8.8</td>
</tr>
<tr>
<td>fss10</td>
<td>76.1</td>
<td>74.1</td>
<td>77.3</td>
<td>78.2</td>
<td>85.5</td>
<td>38.2</td>
</tr>
<tr>
<td>rand2</td>
<td>47.5</td>
<td>46.3</td>
<td>49.2</td>
<td>55.3</td>
<td>62.8</td>
<td>63.4</td>
</tr>
<tr>
<td>rand21</td>
<td>53.9</td>
<td>54.7</td>
<td>57.8</td>
<td>60.5</td>
<td>78.9</td>
<td>-</td>
</tr>
<tr>
<td>ecoli</td>
<td>35.3</td>
<td>35.7</td>
<td>35.2</td>
<td>35.0</td>
<td>38.3</td>
<td>13.4</td>
</tr>
<tr>
<td>chr22</td>
<td>250.1</td>
<td>245.9</td>
<td>255.6</td>
<td>268.3</td>
<td>319.6</td>
<td>134.4</td>
</tr>
<tr>
<td>chr19</td>
<td>513.8</td>
<td>506.3</td>
<td>546.4</td>
<td>554.2</td>
<td>-</td>
<td>286.9</td>
</tr>
<tr>
<td>chr1819</td>
<td>1546.2</td>
<td>1479.0</td>
<td>1587.1</td>
<td>1684.2</td>
<td>-</td>
<td>809.7</td>
</tr>
<tr>
<td>prot-a</td>
<td>104.6</td>
<td>103.3</td>
<td>132.6</td>
<td>165.8</td>
<td>187.4</td>
<td></td>
</tr>
<tr>
<td>prot-b</td>
<td>237.7</td>
<td>225.6</td>
<td>243.8</td>
<td>256.1</td>
<td>284.3</td>
<td></td>
</tr>
<tr>
<td>prot-c</td>
<td>579.6</td>
<td>568.5</td>
<td>590.6</td>
<td>639.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bible</td>
<td>23.5</td>
<td>24.4</td>
<td>25.5</td>
<td>31.5</td>
<td>34.7</td>
<td>-</td>
</tr>
<tr>
<td>howto</td>
<td>213.7</td>
<td>211.3</td>
<td>266.7</td>
<td>307.8</td>
<td>209.1</td>
<td>-</td>
</tr>
<tr>
<td>mozilla</td>
<td>243.7</td>
<td>241.7</td>
<td>257.0</td>
<td>275.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rfc</td>
<td>770.6</td>
<td>725.5</td>
<td>795.6</td>
<td>850.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.5: Peak memory usage in bytes per input symbol for the LZ factorization algorithms.

<table>
<thead>
<tr>
<th>String</th>
<th>CPSa</th>
<th>CPSb</th>
<th>CPSc</th>
<th>CPSd</th>
<th>AKO</th>
<th>KK-LZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibo35</td>
<td>19.5</td>
<td>15.5</td>
<td>13.0</td>
<td>11.5</td>
<td>26.5</td>
<td>20.0</td>
</tr>
<tr>
<td>fibo36</td>
<td>19.5</td>
<td>15.5</td>
<td>13.0</td>
<td>11.5</td>
<td>26.5</td>
<td>20.8</td>
</tr>
<tr>
<td>fse9</td>
<td>19.1</td>
<td>15.1</td>
<td>13.0</td>
<td>11.1</td>
<td>25.1</td>
<td>21.4</td>
</tr>
<tr>
<td>fss10</td>
<td>19.1</td>
<td>15.1</td>
<td>13.0</td>
<td>11.1</td>
<td>25.1</td>
<td>22.5</td>
</tr>
<tr>
<td>rand2</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>17.1</td>
<td>11.8</td>
</tr>
<tr>
<td>rand21</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>17.1</td>
<td>-</td>
</tr>
<tr>
<td>ecoli</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>17.1</td>
<td>11.1</td>
</tr>
<tr>
<td>chr22</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>17.1</td>
<td>11.1</td>
</tr>
<tr>
<td>chr19</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>-</td>
<td>11.1</td>
</tr>
<tr>
<td>chr1819</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>-</td>
<td>10.7</td>
</tr>
<tr>
<td>prot-a</td>
<td>17.2</td>
<td>13.2</td>
<td>13.0</td>
<td>9.2</td>
<td>39.0</td>
<td></td>
</tr>
<tr>
<td>prot-b</td>
<td>17.1</td>
<td>13.1</td>
<td>13.0</td>
<td>9.1</td>
<td>40.1</td>
<td></td>
</tr>
<tr>
<td>prot-c</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bible</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>17.0</td>
<td>-</td>
</tr>
<tr>
<td>howto</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td>17.0</td>
<td>-</td>
</tr>
<tr>
<td>mozilla</td>
<td>17.7</td>
<td>13.7</td>
<td>13.0</td>
<td>9.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rfc</td>
<td>17.0</td>
<td>13.0</td>
<td>13.0</td>
<td>9.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6

An Application of the New Algorithm

Lempel-Ziv factorization is an important data structure for information. Its original purpose is for data compression. However recently it was used in linear time algorithms for the computation of repetitions in a string [1, 29, 22]. This was also our initial motivation for improving the LZ-factorization construction algorithm. In this chapter we discuss the details of our work on the computation of repetitions.

6.1 Background of Algorithms For Repetitions

Periodicity (repetition) in infinite strings was the first topic of stringology [46]; counting and computing the maximum-length adjacent repeating substrings (repetitions) in a finite string was, along with pattern-matching, one of the earliest computational problems on strings to be studied [28, 30].

During the period 1906-1914, Axel Thue published four papers which represented the pioneering work in stringology. Two of these papers [46, 47] deal with repetitions
in finite and infinite words. However, [14] Thue's results were ignored for a long time and rediscovered over and over by other researchers. More recently, Thue's results have become well known because the study of repetition was widely applied in various subjects, such as string matching algorithms, molecular biology, or text compression.

In 1981 Crochemore [6] proved that a string with length $n$ can contain $O(n \log n)$ repetitions and several authors published algorithms to detect these structures in $O(n \log n)$ time [2, 6, 31]. Slisenko [45] published a difficult 100-page algorithm in linear time for finding all periodicities; after that other researchers looked for simple algorithms for detecting repetitions more efficiently.

In 1989 Main introduced the idea that a run or maximal repetition in a word describes several repetitions because its extension by one letter to the right or to the left yields a word with a bigger period. By computing all runs we are implicitly computing all repetitions. Main proposed a linear time algorithm which finds all leftmost maximal repetitions in a word. This algorithm is based on a special factorization of the word, called LZ-decomposition (Lempel-Ziv decomposition). It shows how to compute the leftmost occurrence of every run in a string $x = x[1..n]$ by

1. computing $\text{ST}_x$, the suffix tree of $x$ [49];
2. using $\text{ST}_x$ to compute $\text{LZ}_x$, the Lempel-Ziv decomposition of $x$ [25];
3. using $\text{LZ}_x$ to compute leftmost runs.
Since steps (2) and (3) require only $\Theta(n)$ (linear) time, the use of Farach’s linear-time STCA [9] enables the leftmost runs to be computed in linear time. In [20] Kolpakov & Kucherov proved that the maximum number of runs in any string of length $n$ is $\Theta(n)$, and then showed how to compute all the runs in $x$ from the leftmost ones in linear time. Thus in theory all runs, hence all repetitions, could be computed in linear time, though Farach’s algorithm is not practical for large $n$.

In [1] Abouelhoda, Kurtz & Ohlebusch show how to compute $\text{LZ}_x$ from a suffix array $SA_x$, together with other linear structures, rather than from $ST_x$. Since there now exist practical linear-time suffix array construction algorithms (SACAs), it thus becomes feasible to compute all the runs in $x$ in $\Theta(n)$ time for large values of $n$.

### 6.2 The improvements on KK algorithm [22]

We improve Kolpakov and Kucherov’s implementation [22] for computing all the runs in a string. The KK algorithm is composed of four stages:

1. calculation of the suffix tree of $x$;
2. calculation of the Lempel-Ziv decomposition;
3. calculation of the leftmost runs in $x$ [29];
4. calculation of the remaining runs.

We replace the first two stages with the following stages:

1. computing the suffix array using the algorithm in [34] and the LCP array using
Kasai et al.'s algorithm [18];

(2) computing the Lempel-Ziv decomposition using the suffix and LCP arrays.

These modifications significantly improve the KK algorithm's implementation in terms of time and space.
Chapter 7
Conclusions and Future Work

In this thesis we have discussed the background of Lempel-Ziv factorization and its applications. We analyzed the previous algorithms for the Lempel-Ziv construction, and chose two efficient algorithms to illustrate how the Lempel-Ziv factorization is traditionally computed. Then we presented our new algorithm, and compared it with previous algorithms in terms of time and space. By comparisons we can see the features of our new algorithms. We conducted comprehensive experiments on all sorts of data files. The conclusions can be drawn from the results of tests. We also detailed our work on one of Lempel-Ziv’s central applications, that is computing all the runs in a string.

Since our new algorithms have many advantages, we would like to apply them in other applications. There are 4 variants of our algorithm CPS with different features, and their performances are dependent on the types of data file. We want to analyze the reasons and improve the performances. On the other hand, because our algorithms
use suffix array construction algorithms, which are time-inefficient, we will try to modify the suffix array construction algorithms to increase their efficiency.

At last, we would like to extend our research to other approaches to computing Lempel-Ziv factorization, and more generally, on data compression and the computation of repetitions.
Bibliography


