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

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: ______________________

____

Table of Contents

List of Tables

List of Figures

Abstract

For 30 years the Lempel-Ziv factorization $\mathrm{LZ}_{\boldsymbol{x}}$ of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ has been a fundamental data structure of string processing, especially valuable for string compression and for computing all the repetitions (runs) in $\boldsymbol{x}$. 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 $\mathrm{LZ}_{\boldsymbol{x}}$ was based on $\Theta(n)$-time processing of the suffix tree $\mathrm{ST}_{\boldsymbol{x}}$ of $\boldsymbol{x}$. 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 $\mathrm{SA}_{\boldsymbol{x}}$ 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 pursuit 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 Gang Chen
July, 2007

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 $\text{LZ}_{\boldsymbol{x}}$ 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 $\text{LZ}_{\boldsymbol{x}}$ in $\boldsymbol{x}$ 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 $\boldsymbol{x}$ is a substring $\boldsymbol{w} = \boldsymbol{u}^e$ of $\boldsymbol{x}$, with maximum $e \geq 2$, where $\boldsymbol{u}$ is not itself a repetition in $\boldsymbol{w}$. See section 2.4 for a complete definition. A ***run*** in $\boldsymbol{x}$ is a substring $\boldsymbol{w} = \boldsymbol{u}^e\boldsymbol{u}^*$ of "maximal periodicity", where $\boldsymbol{u}^e$ is a repetition in $\boldsymbol{x}$ and $\boldsymbol{u}^*$ a maximum-length possibly empty proper prefix of $\boldsymbol{u}$. A run may encode as many as $|\boldsymbol{u}|$ repetitions. The maximum number of repetitions in any string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ 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 $\boldsymbol{x}$ 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 $\boldsymbol{x}$.

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 $\text{LCP}_{\boldsymbol{x}}$, computes $\text{LZ}_{\boldsymbol{x}}$ 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 $\text{SA}_{\boldsymbol{x}}$ 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** $\boldsymbol{A}$ is a set whose elements are called **letters**. Suppose $\boldsymbol{A}$=$\{l_1, l_2, ..., l_\alpha\}$, then for any $0 \leq i \leq \alpha$, we say that l_i is an element of $\boldsymbol{A}$ denoted by $l_i \in \boldsymbol{A}$ and $\alpha = |\boldsymbol{A}|$ is the **alphabet size**, that is the number of all elements contained in $\boldsymbol{A}$. If α is infinite, we say that $\boldsymbol{A}$ is an infinite alphabet; otherwise, we* *say that* $\boldsymbol{A}$ *is a finite alphabet.*

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

Definition 2.1.2 *A **string** $\boldsymbol{x}$ is sequence of zero or more elements drawn from an alphabet A. $|\boldsymbol{x}|$ denotes the **length** of string $\boldsymbol{x}$.*

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 $\boldsymbol{x} = baaabaabaababa$, then we can present this string as an array $\boldsymbol{x} = \boldsymbol{x}[1..14]$ such as:

$$\begin{array}{cccccccccccccc} \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 & \scriptstyle 12 & \scriptstyle 13 & \scriptstyle 14 \\ \boldsymbol{x} = b & a & a & a & b & a & a & b & a & a & b & a & b & a \end{array}$$

This string $\boldsymbol{x}$ is defined on an alphabet $\mathbf{A} = \{a, b\}$ with alphabet size $\alpha = 2$ and length 14. We say that $\boldsymbol{x}$ has 14 elements $\boldsymbol{x}[1] = b, \boldsymbol{x}[2] = a, \boldsymbol{x}[3] = a, ..., \boldsymbol{x}[14] = a$ and also we can say that $\boldsymbol{x}$ has 14 positions while position 1 is at the **leftmost** side of $\boldsymbol{x}$ and position 14 is the **rightmost** side of $\boldsymbol{x}$. Taking position 8 for example, we say that $\boldsymbol{x}[8] = b$, the **previous** element of $\boldsymbol{x}[8]$ is $\boldsymbol{x}[7] = a$ and the **next** element of $\boldsymbol{x}[8]$ is $\boldsymbol{x}[9] = a$.

Definition 2.1.3 *The **empty string** is a string containing zero elements and denoted by $\boldsymbol{\varepsilon}$. The length of the empty string is zero. The array model works also for the empty string $\boldsymbol{x} = \boldsymbol{\varepsilon}$ which corresponds to an empty array and has length zero.*

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

$$\boldsymbol{x}[i..j] = \boldsymbol{x}[i]\boldsymbol{x}[i+1]...\boldsymbol{x}[j]$$

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

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

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 $\boldsymbol{x} = x_1x_2...x_m$ and $\boldsymbol{y} = y_1y_2...y_n$, 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 $\boldsymbol{x} < \boldsymbol{y}$, if $x_j > y_j$, then $\boldsymbol{x} > \boldsymbol{y}$. If $x_i = y_i$ for all i, then if $m = n$, $\boldsymbol{x} = \boldsymbol{y}$, while if $m > n$, $\boldsymbol{x} > \boldsymbol{y}$, and if

$m < n$, $\boldsymbol{x} < \boldsymbol{y}$.

2.2 Suffix Tree

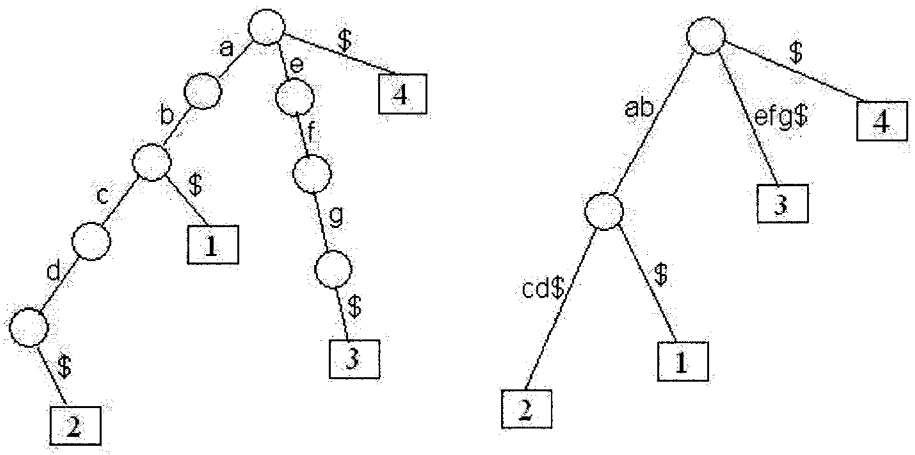
Before we discuss suffix trees, we should define tries and Patricia tries.

Definition 2.2.1 [50] *Given a set $W = \{x_1, x_2, ..., x_m\}$ of pairwise distinct strings, a **trie** on W is a tree containing exactly $m + 1$ terminal nodes, one for each x_i $(i = 1, 2, ..., m)$, plus one for ε. 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



(a) the trie on W={ab, abcd, efg} (b) the Patricia trie on W={ab, abcd, efg}

Figure 2.1: The trie and Patricia trie on W=$ab, abcd, efg$

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 ε) 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.

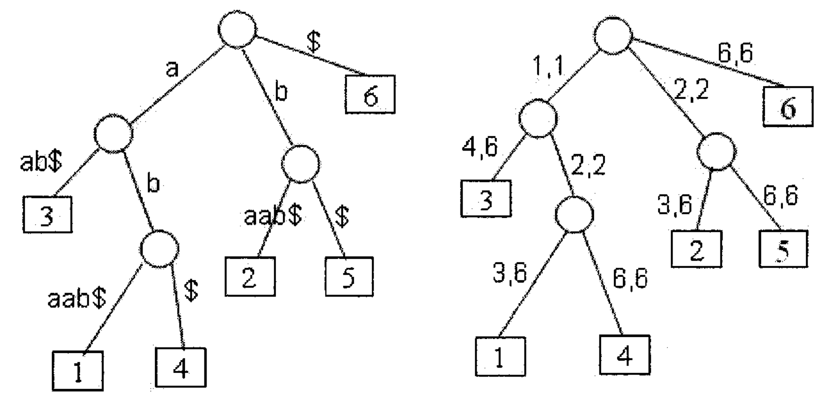
Both tries and Patricia tries can be used to search, not only for any string in W, but also for any prefix of any string in W. The difference is that, in a trie, any prefix of any string in W is a node of the trie, but in a Patricia trie, the prefix may not be a node; that means, it is possible that 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 $\boldsymbol{x}$ with length n, suppose a set W contains $n+1$ elements that are the suffixes of $\boldsymbol{x}$ including \$, then we say that the* ***suffix tree*** *T_x of $\boldsymbol{x}$ 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 $\boldsymbol{x} = abaab$ 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 $\boldsymbol{x}[i..6]$ of $\boldsymbol{x}$. And in T_x, each edge labels a substring of $\boldsymbol{x}$. We can replace the substring in each edge by two integers identifying the position of the substring in $\boldsymbol{x}$. Thus, in this way, we can bound the space required for each



(a)The suffix tree Tx of string x=abaab (b)The suffix tree T'x of string x=abaab

Figure 2.2: The suffix trees for string $\boldsymbol{x} = abaab\$$

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 $\boldsymbol{x}$ 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 $\boldsymbol{x}$, the suffix tree T_x can be used to determine whether a given pattern $\boldsymbol{u}$ is a substring of $\boldsymbol{x}$; the longest repeated substring of $\boldsymbol{x}$ can be represented by the deepest internal node of T_x and the repetitions as well as the number of distinct substrings of $\boldsymbol{x}$ also can be computed by using T_x. Furthermore, suffix trees are especially valuable in cases where $\boldsymbol{x}$ 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 $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet A of size α, we refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as* ***suffix*** *i. The* ***suffix array*** *of the string $\boldsymbol{x}$, or $SA_{\boldsymbol{x}}$ is defined to be the array on $1..n$ in which $SA_{\boldsymbol{x}}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the suffixes of $\boldsymbol{x}$.*

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

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

For example, for string $\boldsymbol{x} = \boldsymbol{x}[1..14]$,

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

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 $lcp_{\boldsymbol{x}}(i_1, i_2)$ denote the* ***longest common prefix*** *of suffixes i_1 and i_2 of $\boldsymbol{x}$. Then $LCP_{\boldsymbol{x}}$ is the array on $1..n{+}1$ in which $LCP_{\boldsymbol{x}}[1] = LCP_{\boldsymbol{x}}[n{+}1] = -1$, while for $j \in 2..n$,*

$$\mathrm{LCP}_{\boldsymbol{x}}[j] = \Big|\mathrm{lcp}_{\boldsymbol{x}}(\mathrm{SA}_{\boldsymbol{x}}[j-1], \mathrm{SA}_{\boldsymbol{x}}[j])\Big|.$$

When the context is clear, we write SA for $\mathrm{SA}_{\boldsymbol{x}}$, LCP for $\mathrm{LCP}_{\boldsymbol{x}}$. For example:

	1	2	3	4	5	6	7	8	9
$\boldsymbol{x} =$	a	b	a	a	b	a	b	a	
$\mathrm{SA}_{\boldsymbol{x}} =$	8	3	6	1	4	7	2	5	
$\mathrm{LCP}_{\boldsymbol{x}} =$	-1	1	1	3	3	0	2	2	-1

2.4 Repetition and Run

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

$$\boldsymbol{x}[i..i+e|\boldsymbol{u}|-1] = \boldsymbol{u}^e,$$

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

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

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

For example, for a string $\boldsymbol{x} = baaabaabaababa$,

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

Then $\boldsymbol{x}$ has a run $\boldsymbol{x}[3..12] = (aab)^3a$ of period $p = 3$ and exponent $e = 3$ with tail $\boldsymbol{t} = a$ of length $t = |\boldsymbol{t}| = 1$. 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 $\boldsymbol{x}[10..14] = (ab)^2a$ of period $p = 2$ and $e = 2$ with tail $\boldsymbol{t} = a$ of length $t = |\boldsymbol{t}| = 1$, 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 $\boldsymbol{x}$ specifies all the repetitions in $\boldsymbol{x}$.

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* $\boldsymbol{x} = \boldsymbol{w_1 w_2 ... w_k}$*, such that each* w_j*,* $j \in 1..k$*, is*

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

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

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

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

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

For example, given a string $\boldsymbol{x} = ababacba$,

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:

$\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = aba$, $\boldsymbol{w_4} = c$, $\boldsymbol{w_5} = ba$

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

LZ-factorization of $\boldsymbol{x}$ can be specified by

$$(1,0),\ (2,0),\ (1,3),\ (6,0),\ (2,2).$$

We observe that $\boldsymbol{w_5}$ has two left occurrences in $\boldsymbol{x} = \boldsymbol{w_1 w_2 ... w_j}$, 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 $x[1..i], i = 1, 2, ..., n$ 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 $\boldsymbol{w_j}$ in the s-factorization of $\boldsymbol{x}$ by a pair (i_L, ℓ), where i_L is the starting position of the leftmost occurrence of the repeating substring $\boldsymbol{w_j}$ in $\boldsymbol{x}$ and $\ell = |\boldsymbol{w_j}|$ is the substring's length. Let i_0 be the current position in $\boldsymbol{x}$ - that is, the first position following the prefix $\boldsymbol{w_1 w_2 ... w_{j-1}}$. For each i_0 we imagine computing the correct value of ℓ by searching T_x from the root for the suffix $\boldsymbol{x}[i_0..n]$ that will of course lead to the terminal node i_0. In order to search, we initialize $i \leftarrow i_0$, then match on $\boldsymbol{x}[i]$, 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 ℓ 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 — creat new leaf nodes if necessary
     — locate the suffix $w = uv = x[j_L + 1..i]$ 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
              creat 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 $\boldsymbol{x}$ as a sequence of pairs $(i_L, \ell)$
$i_0 \leftarrow 1$
while $i_0 \leq n$ do
    $(i_l, \ell) \leftarrow$ match$(i_0, T_x)$
    output $(i_l, \ell)$
    $i_0 \leftarrow i_0 + \ell$
```

Figure 3.2: Algorithm LZ: computing LZ$_{\boldsymbol{x}}$

For example, a string $\boldsymbol{x} = abaababa$, with s-factorization $\boldsymbol{w_1} = a, \boldsymbol{w_2} = b, \boldsymbol{w_3} = a, \boldsymbol{w_4} = aba, \boldsymbol{w_5} = ba$. That can also be represented as $a/b/a/aba/ba$. Its labelled suffix tree $T_{\boldsymbol{x}}$ is represented in Figure 3.3. The label of the root of $T_{\boldsymbol{x}}$ 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 $\boldsymbol{x}[1..8] = abaababa$ on $T_{\boldsymbol{x}}$ 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 $\boldsymbol{x}[2..8] = baababa$ on $T_{\boldsymbol{x}}$ 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 $\boldsymbol{x}[3..8] = aababa$ on $T_{\boldsymbol{x}}$ to find node 3. We follow the

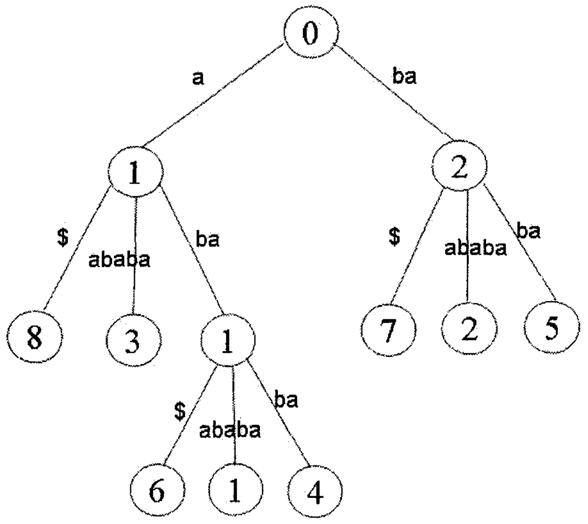


Figure 3.3: The labelled suffix tree for $\boldsymbol{x} = abaababa$

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 $\boldsymbol{x}[4..8] = ababa$ on $T_{\boldsymbol{x}}$ 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 $\boldsymbol{x}[7..8] = ba$ on $T_{\boldsymbol{x}}$ 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

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 $\boldsymbol{x} = \boldsymbol{x}[1..n]$. An interval $[i..j]$, $0 \le i < j \le n$, is an lcp-interval of lcp-value ℓ if*

(1) $LCP_{\boldsymbol{x}}[i] < \ell$,

(2) $LCP_{\boldsymbol{x}}[k] \ge \ell$ for all k with $i+1 \le k \le j$,,

(3) $LCP_{\boldsymbol{x}}[k] = \ell$ for at least one k with $i+1 \le k \le j$,,

(4) $LCP_{\boldsymbol{x}}[j+1] < \ell$.

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

We still use the example $\boldsymbol{x} = abaababa$. 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.

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

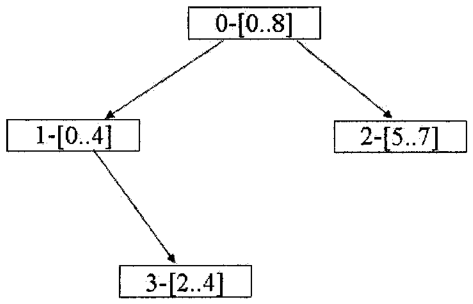


Figure 3.4: The lcp-interval tree of $\boldsymbol{x}$ = *abaababa*

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 $\langle lcp, lb, rb, childList, min\rangle$, 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 $\text{SA}_{\boldsymbol{x}}[lb..rb]$. The Lempel-Ziv factorization is represented by arrays $\text{POS}[0..n-1]$ and $\text{LEN}[0..n-1]$ (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]

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

procedure $process(lastInterval)$
  $i \leftarrow lastInterval.lb;$
  $j \leftarrow lastInterval.rb;$
  $M = \{\mathrm{SA}[q] \mid q \in [i..j]\}$
  $lastInterval.min \leftarrow min(M)$
  for all $p \in M$ with $q \neq min;$
      $POS[p] := min$
      $LEN[p] := lastInterval.lcp$
```

Figure 3.5: Algorithm AKO: computing $\mathrm{LZ}_{\boldsymbol{x}}$

Chapter 4

New Algorithms

4.1 Description of the Algorithms

Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet A of size α, its $\mathrm{SA}_{\boldsymbol{x}}$ and $\mathrm{LCP}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ time [18, 35]. For example:

	1	2	3	4	5	6	7	8	9
$\boldsymbol{x} =$	a	b	a	a	b	a	b	a	
$\mathrm{SA}_{\boldsymbol{x}} =$	8	3	6	1	4	7	2	5	
$\mathrm{LCP}_{\boldsymbol{x}} =$	-1	1	1	3	3	0	2	2	-1

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

Algorithm CPS

```
— Using $SA_{\boldsymbol{x}}$ and $LCP_{\boldsymbol{x}}$, compute $POS[1..n]$ and $LEN[1..n]$.
$i_1 \leftarrow 1$; $i_2 \leftarrow 2$; $i_3 \leftarrow 3$
while $i_3 \le n+1$ do
— Identify the next position $i_2 < i_3$ with $LCP[i_2] > LCP[i_3]$.
   while $\mathrm{LCP}[i_2] \le \mathrm{LCP}[i_3]$ 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 $POS[i] \leftarrow i$.
    $p_2 \leftarrow \mathrm{SA}[i_2]$; $\ell_2 \leftarrow \mathrm{LCP}[i_2]$
    assign(POS, LEN, $p_1$, $p_2$)
   while $\mathrm{LCP}[i_1] = \ell_2$ do
         $i_1 \leftarrow$ pop$(S)$
         assign(POS, LEN, $p_1$, $p_2$)
    $\mathrm{SA}[i_1] \leftarrow p_2$
— Reset pointers for the next stage.
    if $i_1 > 1$ then
         $i_2 \leftarrow i_1$; $i_1 \leftarrow$ pop$(S)$
    else
         $i_2 \leftarrow i_3$; $i_3 \leftarrow i_3+1$

procedure assign(POS, LEN, $p_1$, $p_2$)
$p_1 \leftarrow \mathrm{SA}[i_1]$
if $p_1 < p_2$ then
    $\mathrm{POS}[p_2] \leftarrow p_1$; $\mathrm{LEN}[p_2] \leftarrow \ell_2$; $p_2 \leftarrow p_1$
else
    $\mathrm{POS}[p_1] \leftarrow p_2$; $\mathrm{LEN}[p_1] \leftarrow \ell_2$
```

Figure 4.1: Algorithm CPS: computing $\mathrm{LZ}_{\boldsymbol{x}}$

The basic strategy of CPS is first to locate, in a left-to-right traversal of SA, a next position i_2 such that $\mathrm{LCP}[i_2] > \mathrm{LCP}[i_3]$ for some least $i_3 > i_2$; then second to backtrack (using the stack S) from i_2, setting $\mathrm{POS}[p_2] \leftarrow p1$ or $\mathrm{POS}[p_1] \leftarrow p2$

depending on whether $p_1 = \text{SA}[i_1] < p_2$ or not, until the LCP value for the position i_1 popped from S falls below $\text{LCP}[i_2]$. This processing does not guarantee that, for equal LCP (LEN), each corresponding position in POS necessarily points to the *leftmost* occurrence in $\boldsymbol{x}$, 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 $\text{POS}\big[\text{SA}[i]\big]$ 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 **while** loop, then replacing

$$\text{POS}[p_2] \leftarrow p_1 \ \text{ by } \ \text{SA}[i_2] \leftarrow p_2;\ \text{LCP}[i_2] \leftarrow p_1$$

$$\text{POS}[p_1] \leftarrow p_2 \ \text{ by } \ \text{SA}[i_2] \leftarrow p_1;\ \text{LCP}[i_2] \leftarrow p_2$$

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 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($\boldsymbol{x}$, POS, $i$)
$j \leftarrow$ POS$[i]$
if $j = i$ then
    LEN $\leftarrow 0$
else
    $\ell \leftarrow 1$
    while $\boldsymbol{x}[i+\ell] = \boldsymbol{x}[j+\ell]$ do
        $\ell \leftarrow \ell+1$
    LEN $\leftarrow \ell$
```

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 $\boldsymbol{x}$, SA, LCP, POS & LEN) plus $4s$ bytes for a stack of maximum size s — at most the maximum depth of $\text{ST}_{\boldsymbol{x}}$. For $\boldsymbol{x} = a^n$, $s = n$, but in the expected case, $s \in O(\log_\alpha n)$ [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 $\boldsymbol{x} = abaababa$ 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 $\boldsymbol{x}$ (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

$i = 1$, output (POS[1], LEN[1])=**(1,0)**, then $i = i + 1 = 2$;

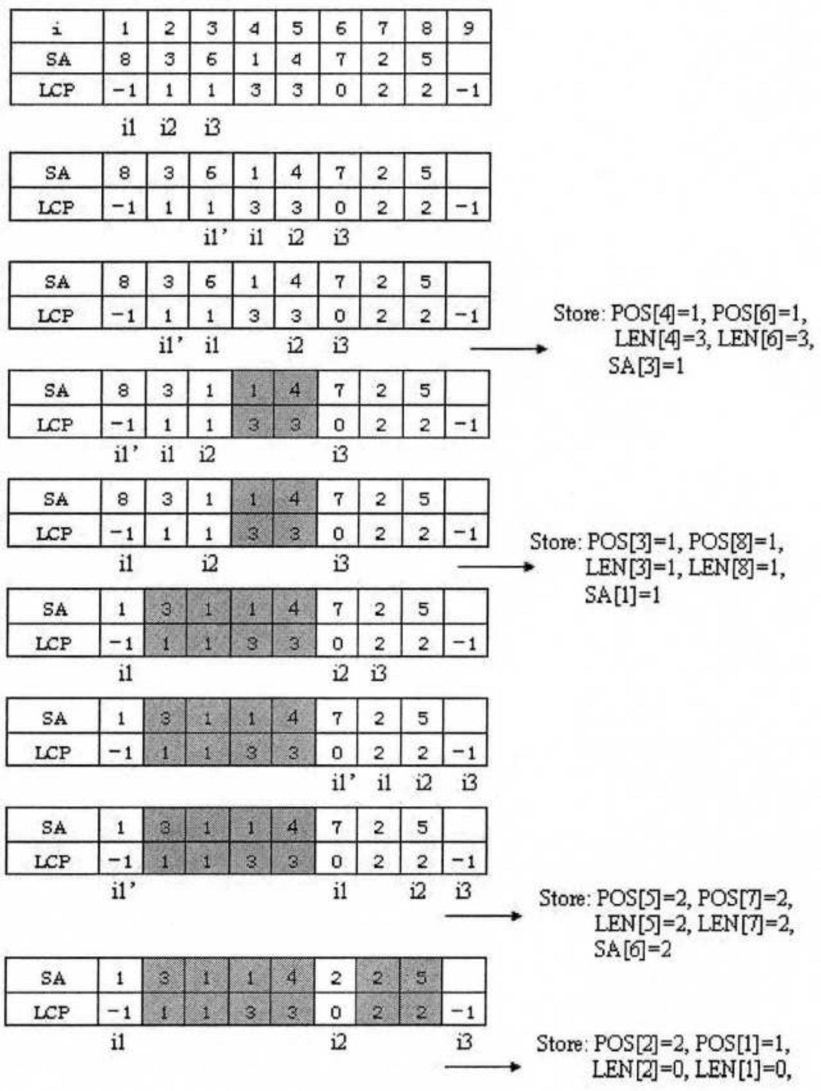
$i = 2$, output (POS[2], LEN[2])=**(2,0)**, then $i = i + 1 = 3$;

$i = 3$, output (POS[3], LEN[3])=**(1,1)**, then $i = i + 1 = 4$;

$i = 4$, output (POS[4], LEN[4])=**(1,3)**, then $i = i + 3 = 7$;

$i = 7$, output (POS[7], LEN[7])=**(2,2)**, then $i = i + 2 = 9 = n + 1$; stop

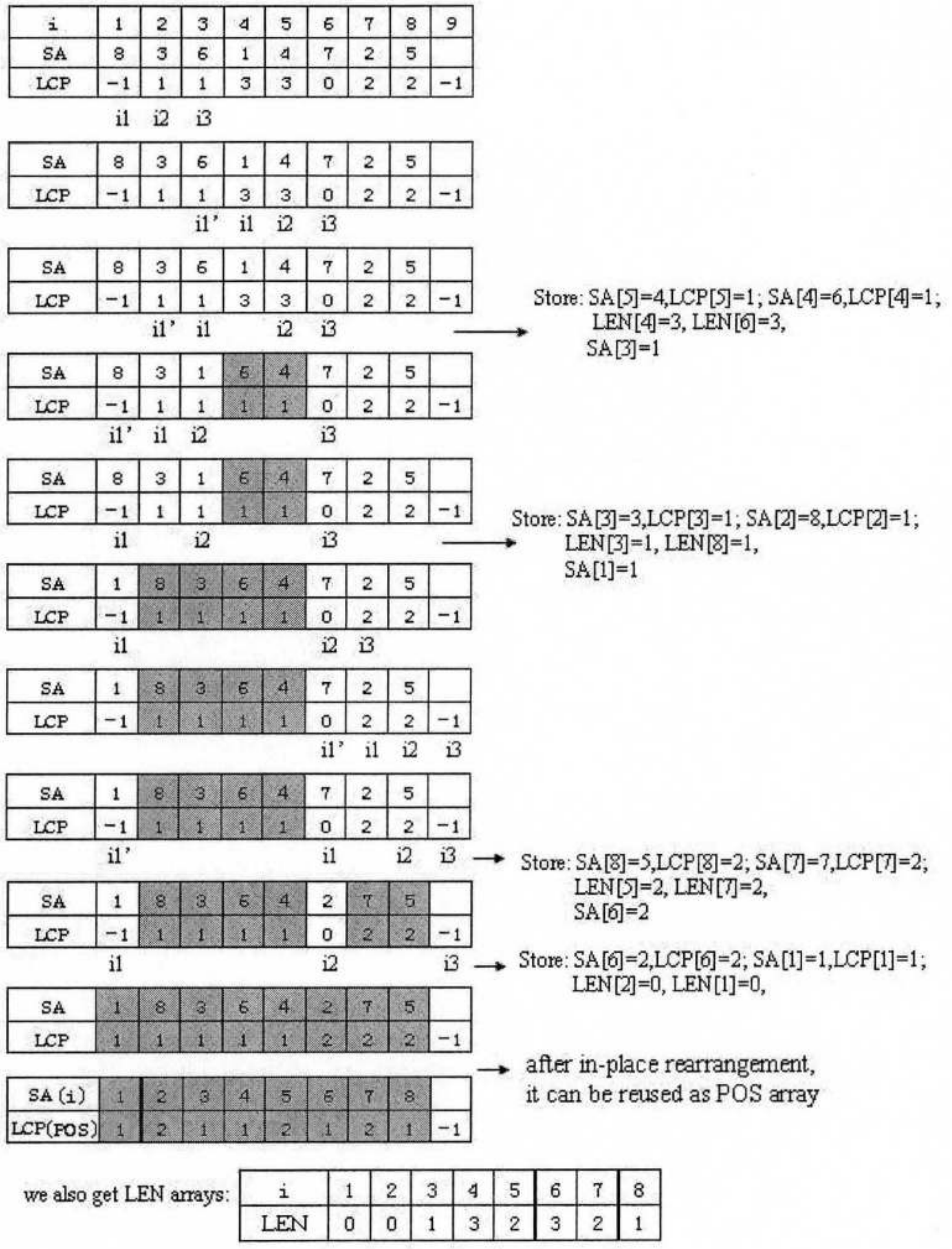
Finally we get LZ-factorization **(1,0)**, **(2,0)**, **(1,1)**, **(1,3)**, **(2,2)**.


i
SA
LCP
i1 i2 i3
i1' i1 i2 i3
Store: POS[4]=1, POS[6]=1, LEN[4]=3, LEN[6]=3, SA[3]=1
Store: POS[3]=1, POS[8]=1, LEN[3]=1, LEN[8]=1, SA[1]=1
Store: POS[5]=2, POS[7]=2, LEN[5]=2, LEN[7]=2, SA[6]=2
Store: POS[2]=2, POS[1]=1, LEN[2]=0, LEN[1]=0,

Finally we get LEN and POS arrays:

i	1	2	3	4	5	6	7	8
POS	1	2	1	1	2	1	2	1
LEN	0	0	1	3	2	3	2	1

Figure 4.3: Algorithm CPSa



i	1	2	3	4	5	6	7	8	9
SA	8	3	6	1	4	7	2	5	
LCP	−1	1	1	3	3	0	2	2	−1

i1 i2 i3

SA	8	3	6	1	4	7	2	5	
LCP	−1	1	1	3	3	0	2	2	−1

i1' i1 i2 i3

SA	8	3	6	1	4	7	2	5	
LCP	−1	1	1	3	3	0	2	2	−1

i1' i1 i2 i3 → Store: SA[5]=4,LCP[5]=1; SA[4]=6,LCP[4]=1; LEN[4]=3, LEN[6]=3, SA[3]=1

SA	8	3	1	6	4	7	2	5	
LCP	−1	1	1	1	1	0	2	2	−1

i1' i1 i2 i3

SA	8	3	1	6	4	7	2	5	
LCP	−1	1	1	1	1	0	2	2	−1

i1 i2 i3 → Store: SA[3]=3,LCP[3]=1; SA[2]=8,LCP[2]=1; LEN[3]=1, LEN[8]=1, SA[1]=1

SA	1	8	3	6	4	7	2	5	
LCP	−1	1	1	1	1	0	2	2	−1

i1 i2 i3

SA	1	8	3	6	4	7	2	5	
LCP	−1	1	1	1	1	0	2	2	−1

i1' i1 i2 i3

SA	1	8	3	6	4	7	2	5	
LCP	−1	1	1	1	1	0	2	2	−1

i1' i1 i2 i3 → Store: SA[8]=5,LCP[8]=2; SA[7]=7,LCP[7]=2; LEN[5]=2, LEN[7]=2, SA[6]=2

SA	1	8	3	6	4	2	7	5	
LCP	−1	1	1	1	1	0	2	2	−1

i1 i2 i3 → Store: SA[6]=2,LCP[6]=2; SA[1]=1,LCP[1]=1; LEN[2]=0, LEN[1]=0,

SA	1	8	3	6	4	2	7	5	
LCP	1	1	1	1	1	2	2	2	−1

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

SA (i)	1	2	3	4	5	6	7	8	
LCP(POS)	1	2	1	1	2	1	2	1	−1

we also get LEN arrays:

i	1	2	3	4	5	6	7	8
LEN	0	0	1	3	2	3	2	1

Figure 4.4: Algorithm CPSb

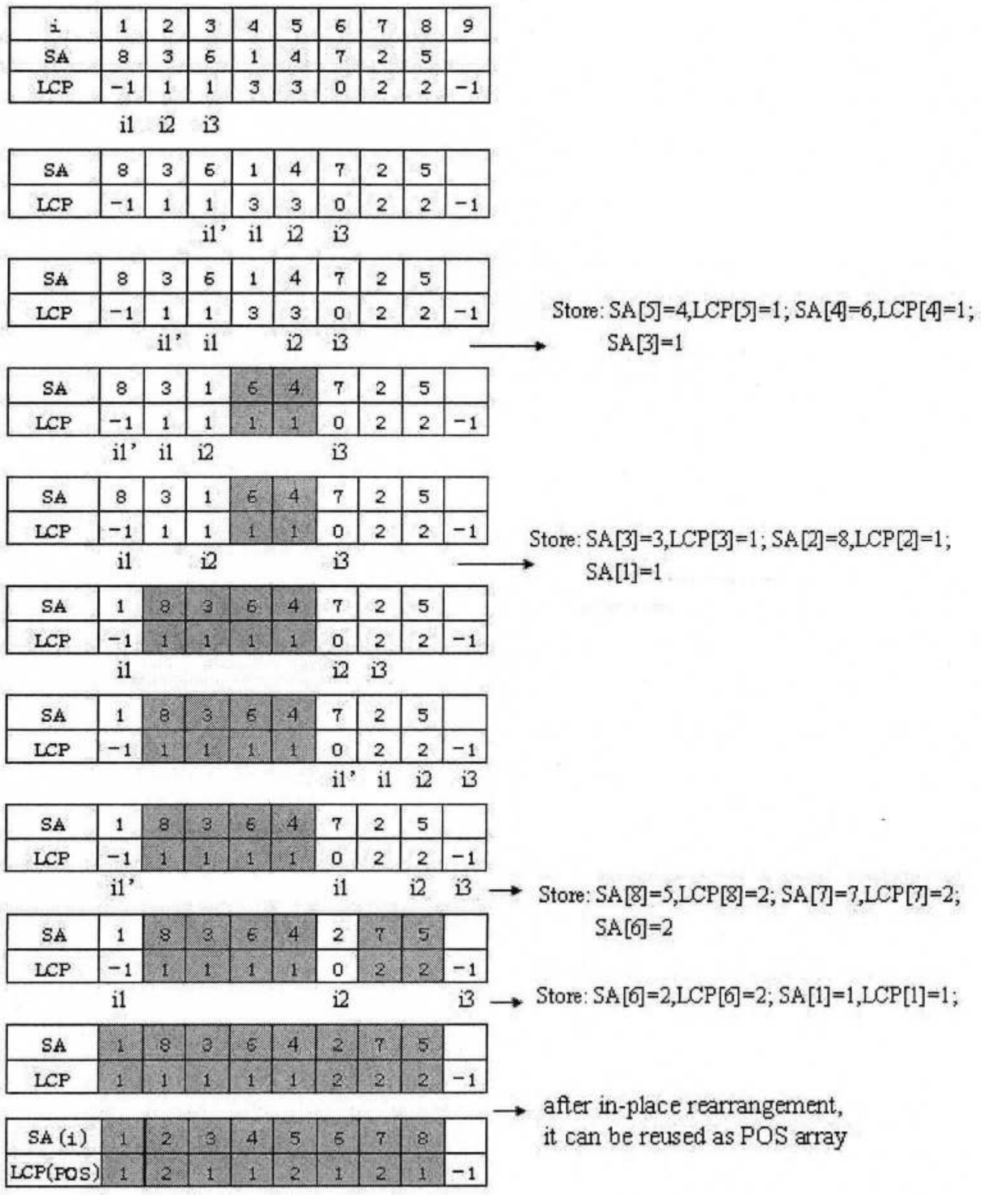

Store: SA[5]=4,LCP[5]=1; SA[4]=6,LCP[4]=1; SA[3]=1
Store: SA[3]=3,LCP[3]=1; SA[2]=8,LCP[2]=1; SA[1]=1
Store: SA[8]=5,LCP[8]=2; SA[7]=7,LCP[7]=2; SA[6]=2
Store: SA[6]=2,LCP[6]=2; SA[1]=1,LCP[1]=1;
after in-place rearrangement, it can be reused as POS array
SA(i)
LCP(POS)

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 AKO 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 AKO 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, AKO 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.

String	Size (bytes)	Σ	# factors	max	Description
fibo35	9227465	2	34	3524578	35th Fibonacci string (see [SMY03])
fibo36	14930352	2	35	5702887	36th Fibonacci string
fss9	2851443	2	40	1217712	9th run rich string of [FSS03]
fss10	12078908	2	44	5158310	10th run rich string of [FSS03]
random2	8388608	2	385232	42	Random string, small alphabet
random21	8388608	21	1835235	9	Random string, larger alphabet
ecoli	4638690	4	432791	2805	E.Coli Genome
chr22	34553758	4	2554184	1768	Human Chromosome 22
chr19	63811651	4	4411679	3397	Human Chromosome 19
chr1819	139928804	4	9560771	3397	Human Chromosomes 18 & 19
prot-a	16777216	23	2751022	6699	Small Protein dataset
prot-b	33554432	24	5040051	16190	Medium Protein dataset
prot-c	67108864	24	8391184	16190	Large Protein dataset
bible	4047392	62	337558	549	King James Bible
howto	39422105	197	3063929	70718	Linux Howto files
mozilla	51220480	256			Mozilla binaries
rfc	116421901	120	5656068	3317	IETF Request for comments

(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.

String	`saca`	`lcp13n`	`lcp9n`
fibo35	10852	4347	5810
fibo36	19253	7310	10166
fss9	2921	1267	1534
fss10	15346	5891	7047
rand2	5542	3347	5465
rand21	6110	5369	6734
ecoli	3871	3136	3563
chr22	29245	22543	26132
chr19	65379	58430	65137
chr1819	173452	152060	199294
prot-a	14218	12576	15733
prot-b	36725	32118	37632
prot-c	49326	45321	59596
bible	2225	2004	2386
howto	23187	22573	29697
mozilla	28213	29572	37439
rfc	84497	82268	131404

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).

String	`CPSa`	`CPSb`	`CPSc`	`CPSd`	`AKO`	`KK-LZ`
fibo35	17347	16321	17160	18623	23839	19033
fibo36	30273	26017	29176	32032	44146	30125
fss9	4651	4256	4478	4745	5922	2310
fss10	25412	23835	25090	26246	31041	15455
rand2	14688	13424	14165	16283	20335	19713
rand21	16134	14235	14870	16235	20176	-
ecoli	9452	9147	9336	9763	13245	3935
chr22	83560	79265	82418	86007	120239	31254
chr19	158613	152520	163653	170362	-	87842
chr1819	483954	461245	461418	508652	-	263135
prot-a	30836	30544	33368	36525	38233	-
prot-b	73478	71105	74214	79731	85790	-
prot-c	158712	143825	167036	181311	-	-
bible	6656	5867	6749	7131	7832	-
howto	66922	65579	67577	72701	65165	-
mozilla	81745	80625	82058	89925	-	-
rfc	218405	201305	220196	269332	-	-

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 $\bar{x} = \frac{1}{N}\sum_{i=1}^{N}(x_i)$, $x_1, ..., x_N$ are the values of the random variable X, N is the number of samples.)

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

Table 5.5: Peak memory usage in bytes per input symbol for the LZ factorization algorithms.

String	CPSa	CPSb	CPSc	CPSd	AKO	KK-LZ
fibo35	19.5	15.5	13.0	11.5	26.5	20.0
fibo36	19.5	15.5	13.0	11.5	26.5	20.8
fss9	19.1	15.1	13.0	11.1	25.1	21.4
fss10	19.1	15.1	13.0	11.1	25.1	22.5
rand2	17.0	13.0	13.0	9.0	17.1	11.8
rand21	17.0	13.0	13.0	9.0	17.1	-
ecoli	17.0	13.0	13.0	9.0	17.1	11.1
chr22	17.0	13.0	13.0	9.0	17.1	11.1
chr19	17.0	13.0	13.0	9.0	-	11.1
chr1819	17.0	13.0	13.0	9.0	-	10.7
prot-a	17.2	13.2	13.0	9.2	39.0	
prot-b	17.1	13.1	13.0	9.1	40.1	
prot-c	17.0	13.0	13.0	9.0		
bible	17.0	13.0	13.0	9.0	17.0	-
howto	17.0	13.0	13.0	9.0	17.0	-
mozilla	17.7	13.7	13.0	9.7		
rfc	17.0	13.0	13.0	9.0		

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 $\boldsymbol{x} = \boldsymbol{x}[1..n]$ by

(1) computing $\mathrm{ST}_{\boldsymbol{x}}$, the suffix tree of $\boldsymbol{x}$ [49];

(2) using $\mathrm{ST}_{\boldsymbol{x}}$ to compute $\mathrm{LZ}_{\boldsymbol{x}}$, the Lempel-Ziv decomposition of $\boldsymbol{x}$ [25];

(3) using $\mathrm{LZ}_{\boldsymbol{x}}$ 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 $\boldsymbol{x}$ 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 $\mathrm{LZ}_{\boldsymbol{x}}$ from a suffix array $\mathrm{SA}_{\boldsymbol{x}}$, together with other linear structures, rather than from $\mathrm{ST}_{\boldsymbol{x}}$. Since there now exist practical linear-time suffix array construction algorithms (SACAs), it thus becomes feasible to compute all the runs in $\boldsymbol{x}$ 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 $\boldsymbol{x}$;

(2) calculation of the Lempel-Ziv decomposition;

(3) calculation of the leftmost runs in $\boldsymbol{x}$ [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

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz & Enno Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete. Algs. 2* (2004) 53–86.

[2] Alberto Apostolico & Franco P. Preparata, **Optimal off-line detection of repetitions in a string**, *Theoret. Comput. Sci. 22* (1983) 297–315.

[3] Michael A. Bender & Martin Farach-Colton, **The LCA problem revisited**, *Latin American Theoretical Informatics* (2000) 88–94.

[4] Rainer Bauer & Joachim Hagenauer, **Symbol-by-Symbol MAP Decoding of Variable Length Codes**, *3rd ITG Conference Source and Channel Coding, Munich, Germany* (2000) 111–116.

[5] David Salomon, **Data Compression**, *Springer* (1997) 147–150.

[6] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *Inform. Process. Lett. 12-5* (1981) 244–250.

[7] Jean-Pierre Duval, Roman Kolpakov, Gregory Kucherov, Thierry Lecroq & Arnaud Lefebvre, **Linear-time computation of local periods**, *Theoret. Comput. Sci. 326–1-3* (2004) 229–240.

[8] Kangmin Fan, Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A new periodicity lemma**, *SIAM J. Discrete Math. 20–3* (2006) 656–668.

[9] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc.* 38th *IEEE Symp. Found. Computer Science* (1997) 137–143.

[10] Paolo Ferragina & Giovanni Manzini, **Opportunistic data structures with applications**, *Proc.* 41st *IEEE Symp. Found. Computer Science* (2000) 390–398.

[11] Johannes Fischer & Volker Heun, **Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE**, *Proc.* 17th *Annual Symp. Combinatorial Pattern Matching*, M. Lewenstein & G. Valiente (eds.) (2006) 36–48.

[12] Frantisek Franek, Jan Holub, W. F. Smyth & Xiangdong Xiao, **Computing quasi suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 593–606.

[13] Frantisek Franek, R. J. Simpson & W. F. Smyth, **The maximum number of runs in a string**, *Proc.* 14th *Australasian Workshop on Combinatorial Algs*, M. Miller & K. Park (eds.) (2003) 26–35.

[14] G. A. Hedlund, **Remarks on the work of Axel Thue on sequences**, *Nordisl. Mat. Tidskr. 16* (1967) 148–150.

[15] Dov Harel & Robert E. Tarjan, **Fast algorithms for finding nearest common ancestors**, *SIAM J. Computing 13-2* (1984) 338–355.

[16] Juha Kärkkäinen & Peter Sanders, **Simple linear work suffix array construction**, *Proc.* 30th *Internat. Colloq. Automata, Language & Programming* (2003) 943–955.

[17] S. Karlin, G. Ghandour, F. Ost, S. Tavare & L. J. Korn, **New approaches for computer analysis of nucleic acid sequences**, *Proc. Natl. Acad. Sci. USA 80* (1983) 5660–5664.

[18] 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*, LNCS 2089, Springer-Verlag (2001) 181–192.

[19] 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.

[20] Roman Kolpakov & Gregory Kucherov, **On maximal repetitions in words**, *J. Discrete Algs. 1* (2000) 159–186.

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

[22] Roman Kolpakov & Gregory Kucherov, `http://bioinfo.lifl.fr/mreps/`.

[23] Roman Kolpakov & Gregory Kucherov, **Finding approximate repetitions under Hamming distance**, *Theoret. Comput. Sci. 303-1* (2003) 135–156.

[24] Stefan Kurtz, **Reducing the space requirement of suffix trees**, *Software, Practice & Experience 29-13* (1999) 1149–1171.

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

[26] Abraham Lempel & Jacob Ziv, **A universal algorithm for sequential data compression**, *IEEE Trans. Information Theory 23* (1977) 337–342.

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

[28] André Lentin & Marcel P. Schützenberger, **A combinatorial problem in the theory of free monoids**, *Combinatorial Mathematics & Its Applications*, R. C. Bose & T. A. Dowling (eds.), University of North Carolina Press (1969) 128–144.

[29] Michael G. Main, **Detecting leftmost maximal periodicities**, *Discrete Applied Maths. 25* (1989) 145–153.

[30] Michael G. Main & Richard J. Lorentz, *An $O(n \log n)$ Algorithm for Recognizing Repetition*, Tech. Rep. CS-79-056, Computer Science Department, Washington State University (1979).

[31] Michael G. Main & Richard J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algs. 5* (1984) 422–432.

[32] Udi Manber & Gene Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM J. Computing 22-5* (1993) 935–948.

[33] Veli Mäkinen & Gonzalo Navarro, **Compressed full-text indices**, *ACM Computing Surveys* (2006) to appear.

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

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

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

[37] Edward M. McCreight, **A space-economical suffix tree construction algorithm**, *J. Assoc. Compul. Mach. 32–2* (1976) 262–272.

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

[39] Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys* (2006) to appear.

[40] Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **Inverted files versus suffix arrays for in-memory pattern matching**, *Proc.* 13th *Symposium on String Processing & Information Retrieval* (2006) 122–133.

[41] Wojciech Rytter, **Grammar compression, LZ-encodings, and string algorithms with implicit input**, *Proc.* 31st *Internat. Colloq. Automata, Languages & programming* (2004) 15–27.

[42] Wojciech Rytter, **The number of runs in a string: improved analysis of the linear upper bound**, *Proc. 23rd Symp. Theoretical Aspects of Computer Science*, B. Durand & W. Thomas (eds.), LNCS 2884, Springer-Verlag (2006) 184–195.

[43] J.S. Sim, D.K. Kim, H. Park & K. Park, **Linear-time search in suffix arrays,** *Proc. 14th Australasian Workshop on Combinatorial* (2003) 139–146.

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

[45] A. Slisenko, **Detection of periodicities and string matching in real time** , *Journal of Soviet mathematics 22* (1983) 1316–1386.

[46] Axel Thue, **Über unendliche zeichenreihen**, *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7* (1906) 1–22.

[47] Axel Thue, **Über die genenseitige Lage gleicher Teile gewisser Zeichenreihen** , *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 1* (1912) 1–67.

[48] Esko Ukkonen, **On-line construction of suffix trees**, *Algorithmica 14* (1995) 249–260.

[49] Peter Weiner, **Linear pattern matching algorithms**, *Proc. 14th Annual IEEE Symp. Switching & Automata Theory* (1973) 1–11.

[50] Xiangdong Xiao, **Computing Quasi suffix arrays**, *M.Sc. thesis. Department of computing and Software, McMaster University* (2003).