Repeats in Strings and Application in Bioinformatics

REPEATS IN STRINGS AND APPLICATION IN BIOINFORMATICS

BY

A S M SOHIDULL ISLAM, M.Sc., (Computer Science and Engineering)

BUET, Dhaka,Bangladesh

A THESIS

SUBMITTED TO THE SCHOOL OF COMPUTATIONAL SCIENCE & ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

PHD

© Copyright by A S M Sohidull Islam, July 2017

All Rights Reserved

PhD (2017) McMaster University
(School of Computational Science and Engineering) Hamilton, Ontario, Canada

TITLE: Repeats in Strings and Application in Bioinformatics

AUTHOR: A S M Sohidull Islam
M.Sc., (Computer Science and Engineering)
BUET, Dhaka, Bangladesh

SUPERVISOR: Dr. William F. Smyth and Dr. Brian Golding

NUMBER OF PAGES: xv, 135

To my family and friends

Abstract

A ***string*** is a sequence of symbols, usually called ***letters***, drawn from some ***alphabet***. It is one of the most fundamental and important structures in computing, bioinformatics and mathematics. Computer files, contents of a computer memory, network and satellite signals are all instances of strings. The genome of every living thing can be represented by a string drawn from the alphabet $\{a, c, g, t\}$. The algorithms processing strings have a wide range of applications such as information retrieval, search engines, data compression, cryptography and bioinformatics.

In a DNA sequence the ***indeterminate*** symbol {a, c} is used when it is unclear whether a given nucleotide is *a* or *c*, We could then say that {a, c} matches another symbol {c, g} which in turn matches {g, t}, but {a, c} certainly does not match {g, t}. The processing of indeterminate strings is much more difficult because of this nontransitivity of matching. Thus a combinatorial understanding of indeterminate strings becomes essential to the development of efficient methods for their processing. With indeterminate strings, as with ordinary ones, the main task is the recognition/computation of patterns called ***regularities***. We are particularly interested in regularities called ***repeats***, whether ***tandem*** such as *acgacg* or ***nontandem*** (*acgtacg*).

In this thesis we focus on newly-discovered regularities in strings, especially the

enhanced cover array and the Lyndon array, with attention paid to extending the computations to indeterminate strings. Much of this work is necessarily abstract in nature, because the intention is to produce results that are applicable over a wide range of application areas. We will focus on finding algorithms to construct different data structures to represent strings such as cover arrays and Lyndon arrays. The idea of cover comes from strings which are not truly periodic but "almost" periodic in nature. For example *abaababa* is covered by *aba* but is not periodic. Similarly the Lyndon array describes the string in another unique way and is used in many fields of string algorithms. These data structures will help us in the field of string processing. As one application of these data structures we will work on "Reverse Engineering"; that is, given data structures derived from of a string, how can we get the string back.

Since DNA, RNA and peptide sequences are effectively "strings" with unique properties, we will adapt our algorithms for regular or indeterminate strings to these sequences. Sequence analysis can be used to assign function to genes and proteins by observing the similarities between the compared sequences. Identifying unusual repetitive patterns will aid in the identification of intrinsic features of the sequence such as active sites, gene-structures and regulatory elements. As an application of periodic strings we investigate ***microsatellites*** which are short repetitive DNA patterns where repeated substrings are of length 2 to 5. Microsatellites are used in a wide range of studies due to their small size and repetitive nature, and they have played an important role in the identification of numerous important genetic loci. A deeper understanding of the evolutionary and mutational properties of microsatellites is needed, not only to understand how the genome is organized, but also to correctly interpret and use microsatellite data in population genetics studies.

Acknowledgements

I would first like to express my sincere gratitude to my supervisor Dr. William F. Smyth. Your great expertise and understanding of algorithms, constant encouragement and enormous support of my research has been the key for the publications and majority of my current thesis. You have consistently offered brilliant insights into my research or writing and encouraged me to think and solve research problems independently. Your priceless advice on graduate life helped me enjoy a fulfilling and wonderful time on the path of learning and discovery.

I would like to take the opportunity to thank my co-supervisor Dr. Brian Golding. Your constant support helped me understand different topics of Bioinformatics. Thank you for always be there to give me the research directions.

I would also like to thank the rest of my thesis committee: Dr. Frantisek Franek and Dr. Jonathon Stone. Thank you Dr. Franek for your time with me for the frequent discussions on my research problems. I would like to thank Dr. Stone as well for his help with my comprehensive exam report.

Thanks also go to my dearest friends and my family. Thank you for your endless support, encouragement and love. I am extremely lucky to have you all in my life.

Declaration

I declare that my thesis contains the following published materials that I was one of the co-authors. My contributions have also been stated. This work was done during my Ph.D. research and constitutes an integral components of this thesis. The copyright holder has agreed to grant an irrevocable, non-exclusive licence to McMaster University and the National Library of Canada to reproduce the material as part of the thesis.

Ali Alatabbi, A. S. M. Sohidull Islam, M. Sohel Rahman, Jamie Simpson, and W. F. Smyth. Enhanced covers of regular and indeterminate strings using prefix tables. Journal of Automata, Languages and Combinatorics, 21(3):131-147, 2016.

Contribution by me: I was involved in the design of a series of new algorithms to compute Enhanced covers and its variants using prefix tables. I was also involved in the implementations of the algorithms.

Frantisek Franek, A. S. M. Sohidull Islam, M. Sohel Rahman, and W. F. Smyth. Algorithms to compute the lyndon array. In Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2015, pages 172-184, 2016.

Contribution by me: I was involved in the design of three new algorithms to compute Lyndon array without using suffix array. I was also responsible for implementation and testing of the algorithms.

Daykin, J. W., Frantisek Franek, Jan Holub, A. S. M. Sohidull Islam, W. F. Smyth. Reconstructing a string from its Lyndon arrays. Theoretical Computer Science (2017).

Contribution by me: I was involved in the design of a new algorithm to Reconstruct a string from its Lyndon arrays. I was also involved in the analysis and the proof of the algorithm.

Contents

Abstract **iv**

Acknowledgements **vi**

Declaration **vii**

1 Introduction **1**

1.1 Major Contributions 4

1.2 Thesis Outline 7

2 Preliminaries and Definitions **8**

2.1 Basic Definitions 8

2.2 Definitions of Various Regularities 10

2.3 Data Structures Used in this Thesis 12

3 Enhanced Covers of Regular & Indeterminate Strings using Prefix Tables **19**

3.1 Introduction 19

3.2 Methodology 21

3.3 Correctness & Complexity of Compute_MEC 23

3.3.1 Combinatorics on the border length 26
3.3.2 Average case analysis 29
3.4 Enhanced Left Covers and Left Seeds 30
3.4.1 Minimum Enhanced Left Cover Array (MELC) 31
3.4.2 Minimum Enhanced Left Seed Array (MELS) 32
3.5 Comparing Border-Based and Prefix-Based Algorithms 35
3.6 Indeterminate Strings 36
3.7 Future Research 38

4 Construction of Lyndon Array 39

4.1 Introduction 39
4.2 Preliminaries 40
4.3 Basic Algorithms 43
4.3.1 Folklore — Iterated MaxLyn 43
4.3.2 Recursive Duval Factorization: Algorithm RDuval 45
4.3.3 NSV Applied to the Inverse Suffix Array 47
4.4 Elementary Computation of $\boldsymbol{\lambda_x}$ by Comparing Ranges 49
4.5 Computation of $\boldsymbol{\lambda_x}$ Using Ranges and NSV Idea 55
4.6 Experimental Results 60
4.7 Future Work 61

5 Reconstructing a String from its Lyndon Arrays 62

5.1 Introduction 62
5.2 When is $\mathcal{L}^*$ a Valid Lyndon Array of Some String? 65
5.3 Reconstructing a String from its Lyndon Arrays 69

5.4 Future Research 76

6 Microsatellite Evolution **79**

6.1 Introduction 79

6.2 Models and Parameters 82

6.3 Inter Species Comparison for Microsatellite with Repeat Length 2 . . 86

6.3.1 Analyzing the parameters for Model PL2 90

6.4 Inter Species Comparison for Microsatellite with Repeat Length 2 . . 97

6.4.1 Data Contain 10 Population (Single alleles) 97

6.4.2 Data Contain 10 Population (2 alleles) 99

6.4.3 Data Contain 5 Population (Single alleles) 100

6.4.4 Data Contain 8 Population 103

6.4.5 Bootstrap Analysis 106

6.5 Microsatellites with Repeat Length 3 108

6.6 Conclusion 110

7 Summary and Future Work **112**

7.1 Future Work 114

A Additional Results for Microsatellite Evolution **116**

List of Figures

2.1 An example string 9

2.2 Border Array β, Prefix Table π and Cover Array γ 13

2.3 $\boldsymbol{x'} = \{a, g\}\{a, t\}$ (or $\{c, g\}\{c, t\}$) is a border of $\boldsymbol{x}$; but neither $\boldsymbol{x''} = \{a, g\}$ nor $\{a, t\}$ is a border of $\boldsymbol{x}[3..4]$, and neither $\boldsymbol{x''} = \{c, g\}$ nor $\{c, t\}$ is a border of $\boldsymbol{x}[1..2]$. 15

2.4 Suffix Array and Inverse Suffix Array 18

2.5 For $i = 3$, we have two Lyndon words starting at the position: aab and $aabab$. Since $aabab$ is longest, therefore $\boldsymbol{\lambda}[3] = 5$ 18

3.1 Computing `MNC` from the prefix array $\boldsymbol{\pi}[1..n]$ and the cover array $\boldsymbol{\gamma}[1..B]$. 22

3.2 All the arrays required to compute `MEC` and `CMEC` arrays 23

3.3 Computing `MEC` amd `CMEC` from the prefix array $\boldsymbol{\pi}$. 24

3.4 Computing `MELS` amd `CMELS` from the prefix array $\boldsymbol{\pi}$. 33

3.5 The maximum number of operations performed by the Border-Based (ECB) [45] and Prefix-Based (ECP) algorithm (i.e., Compute_MEC) to compute the Minimum Enhanced Cover array, for all strings on the binary alphabet. 36

3.6 Ratio of the ***total*** number of operations performed by the Border-Based (ECB) [45] and Prefix-Based (ECP) algorithms to the length n of the string, for all strings on the binary alphabet. Note the linear behaviour of ECP compared to the slightly supralinear behaviour of ECB. 37
4.1 Algorithm MaxLyn 45
4.2 Apply NSV to $\texttt{ISA}_{\boldsymbol{x}}$ 47
4.3 Algorithm RangeLyndon computes $\mathcal{L}_{\boldsymbol{x}}$ of $\boldsymbol{x} = \boldsymbol{x_1 x_2 \cdots x_m}$. Only at critical points $j_L = c(r, r')$ does it need to do pattern-matching; otherwise all $\mathcal{L}_{r,j}$ are computed in constant time. 50
4.4 Compute $\mathcal{L}$ for all positions j such that $\boldsymbol{x_r}[j] > \boldsymbol{x_{r+1}}[1]$. 50
4.5 Find the next critical point $j_L = c(r, r')$ to the left of the current position j_R in $\boldsymbol{x_r}$, updating r' as required. 51
4.6 Compute $\mathcal{L}_{r,j}$ for all positions $j \in 1..c(r, r')$, taking account of the Monge property that arcs $(\boldsymbol{x_r}[j], \mathcal{L}_{r,j})$ cannot intersect (Observation 1). 52
4.7 Computing $\boldsymbol{\lambda}_x$ using modified NSV 57
4.8 Five algorithms compared on all binary strings of lengths $n \in 11..22$: the average processing time for each n is given in 10^{-4} seconds. For all the algorithms except RDuval, the pre-processing time is independent of length n. So the increase of time is very small which results very small amount of slope for the corresponding lines. 61
5.1 Given a valid Lyndon array $\mathcal{L}^*$ and an ordered alphabet $\Sigma = \{1, 2, \ldots, n\}$, in $\mathcal{O}(n)$ time construct a string $\boldsymbol{x}$ on Σ whose Lyndon array is $\mathcal{L}^*$. . . 65

5.2 Construct a string $\boldsymbol{x}$ on a subset of $\Sigma = \{1, 2, \ldots, n\}$ with Lyndon array L^*. 67
5.3 Determine whether (`TRUE`) or not (`FALSE`) a given integer array $\mathcal{L}^*$ is a Lyndon array of some string. 69
5.4 Lyndon arrays based on rotated orders for $\sigma = 3$. 72
5.5 Constructing a string from rotated Lyndon arrays. 75
5.6 $\boldsymbol{x} = adbc$ and its Lyndon arrays: consideration of fewer than four rotations of the alphabet may not allow $\boldsymbol{x}$ to be reconstructed. 77
6.1 Model Description. K denotes the number of free parameters. The fixed parameter(s) for each set of models is/are shown above branch leading to it 85
6.2 Comparision of parameter m for different datasets of Table 6.5 . . . 92
6.3 Comparision of parameter s for different datasets of Table 6.5 92
6.4 Comparision of parameter u for different datasets of Table 6.5 93
6.5 Comparision of parameter m for data with outliers and data without outliers of different datasets of Table 6.5 94
6.6 Comparision of parameter s for data with outliers and data without outliers of different datasets of Table 6.5 94
6.7 Comparision of parameter m for original data and bootstrap data of different datasets of Table 6.5 96
6.8 Comparision of parameter s for original data and bootstrap data of different datasets of Table 6.5 96
6.9 Tree for the Pop10 data 97
6.10 Tree for the Pop5 data 101

6.11 Tree for the Pop8 data 104

Chapter 1

Introduction

Thousands of research papers have been written by mathematicians and (over the last half century) also computer scientists that relate in some way to periodicity, or its variants, in strings. A word that has recently been brought into service to describe these variants is "regularities" which was first introduced by Iliopoulos and Mouchard in [58]. Various forms of regularity are central to the recognition of important patterns in performing retrieval from massive data sets. Perhaps the most conspicuous regularities in strings are those that manifest themselves in the form of repeated subpatterns; that is, repeats, multirepeats, tandem repeats, runs and others. Algorithms for computing regularities have myriad applications. For example, one of the earliest and still widely used compression algorithms is gzip. Regularities in the form of repeating substrings were the basis of gzip, which still remains central to other compression approaches. Repeats and repetitions of lengthy substrings in DNA and protein sequences are important markers in biological research. We will discuss this further later on. Email spam, also known as junk email, is a type of electronic spam where unsolicited messages are sent by email. The proportion of spam email

was approximately 80% of all email messages sent in the first half of 2010. Some methods for detecting spam are mainly based on similarity calculations on strings. In addition various forms of regularity are central to the recognition of important patterns in retrieval from massive data sets.

In this thesis we investigate mathematical and algorithmic aspects of regularities in strings. In particular, we have developed novel algorithms for computing regularities which are both time and space efficient. We begin by introducing the basics of strings. A string is a sequence of symbols drawn from an alphabet which is a finite set of distinct symbols. There are numerous examples of alphabets and strings in our world. One of the most common example is human language. An English word is a string drawn from the English alphabet which has 26 letters (or 52 including upper case letters). Strings also exist in a computer system, where essentially all the information is represented in the form of strings. Because an electronic computer is built on electrical circuits that usually have only two states, namely low voltage and high voltage, all the strings in computer systems are in essence drawn from a simple and natural "binary" alphabet $\{0, 1\}$. In biology, it is well known that DNA is made up of nucleotides, which can be modeled by strings. The DNA sequence can be represented by a string drawn from an alphabet of four basic letters a, c, t, g. DNA directs the activity of cells and is responsible for creating a particular organism with its own unique traits.

The notion of periodicity in strings and its many variants have been well-studied in many fields like combinatorics on words, pattern matching, data compression, automata theory, formal language theory, and molecular biology. But the notion of periodicity is too restrictive to provide a description of a string such as $\boldsymbol{x} = abaababa$,

which is covered by copies of *aba*, yet not exactly periodic. To fill this gap, the idea of quasiperiodicity was introduced [11]. Quasiperiodicity enables the detection of repetitive structures that would be ignored by the classical characterisation of periods. The most well-known formalisation of quasiperiodicity is the ***cover*** of a string; in the above example *aba* is a cover of *abaababa*. While covers capture very well the repetitive nature of extremely repetitive strings, nevertheless most strings, and particularly those encountered in practice, will have no cover, and so these measures of repetitiveness break down. Therefore new and more natural and applicable forms of quasiperiodicity need to be invented. The promising idea of an ***enhanced cover*** was introduced recently in [45] which is a form of quasiperiod. Further, on the analogy of the cover array, the authors proposed a ***minimum enhanced cover array*** and presented an algorithm to compute it by using a variant of the ***border array***.

An interesting aspect of DNA is the frequency of microsatellites. They are composed of short DNA sequences of length $2-5$ characters ("base pairs") that are repeated in tandem; for example *cgacgacga*. In genomes, perfect or near-perfect tandem iterations of short sequence motifs of this kind are extremely common. Between closely related species, their distribution and density in genomes can vary greatly. In the case of the human genome, they are found at hundreds of thousands of places along chromosomes [64]. Every possible motif of mono-, di, tri- and tetranucleotide repeat is found frequently in the genome. Also referred to as short tandem repeats (STRs) or simple sequence repeats (SSRs), the ubiquitous occurrence of microsatellites was first reported in the 1980s [65]. Supported by direct observations it is assumed that mutations must occur frequently among microsatellites. However, despite the extensive study of microsatellites over the past 25 years, it is clear that many theoretical

models fail to accurately explain allele frequency distributions in natural populations. Importantly, it seems that microsatellite evolution is a far more complex process than was previously thought.

1.1 Major Contributions

In this thesis we explore theoretical and algorithmic aspects of the computations of regularities in strings. We describe the following results in subsequent chapters.

In the first part of this thesis we introduce a new algorithm and data structures to compute the minimum enhanced cover array from the prefix table, and illustrate the ideas with examples. Computing the minimum enhanced cover array from the prefix table rather than from a variant of the border array allows us to extend the computation to indeterminate strings. We provide a proof of the algorithm's correctness, as well as an analysis of its complexity, both worst and average case. We extend the basic algorithm to enhanced left covers and enhanced left seeds. We discuss the practical application of our algorithms, in terms of time and space requirements, and compare our prefix-based implementation with the border-based implementation of [45]. Our algorithms, in addition to being faster in practice and more space-efficient than those of [45], allow us to easily extend the computation of enhanced covers to indeterminate strings. Both for regular and indeterminate strings, our algorithms execute in expected linear time. Along the way we establish an important theoretical result: that the expected maximum length of any border of any prefix of a regular string $\boldsymbol{x}$ is approximately 1.64 for binary alphabets, less for larger ones. We show how to extend the various enhanced cover array algorithms to indeteterminate strings and give a summary of our results and future research directions.

Computing all the periods of a string is one of the most challenging problems in the field of stringology. Bannai et al. [15] described a linear-time algorithm which does not rely on the Lempel-Ziv parsing of the string for computing all runs in a string. Their result thus may help pave the way to the algorithms for computing all runs in a string which are faster in practice. In the second part of our thesis, at first we outline three algorithms to compute the Lyndon array for which no clear exposition is available in the literature. Two of them require $O(n^2)$ time in the worst case, of which one is very fast and apparently linear in practice, the other supralinear in practice and $O(n \log n)$ in the average case on binary strings. The third algorithm is simple and worst-case linear-time, but requires suffix array construction and so is a little slower.

Next we formulate two new approaches to find the Lyndon array of a string. Both approaches use only elementary data structures (no suffix arrays). The first approach has two variants: one variant requires $O(n^2)$ time in the worst case, the other guarantees $O(n \log n)$ time, but with no clear advantage in processing time. In this approach we will process the string from left to right and use a "stack" data structure. In the second approach we will process the string from right to left to compute the Lyndon array. Finally after implementing these algorithms, we will show that all of them run in $\Theta(n)$ time in practice.

An interesting topic is to establish a "reverse engineering" result for Lyndon arrays: that is, to reconstruct the string from its Lyndon array. Since the Lyndon array is an array of positive integers, it is natural to ask under what conditions a given integer array is a Lyndon array. In the third part of the thesis, we present necessary and sufficient condition that a given integer array is a Lyndon array of some string

on some alphabet. We then describe a linear-time algorithm to evaluate these conditions for a given array. We establish a "reverse engineering" result for Lyndon arrays; that is, given certain Lyndon arrays, based on orderings of a given alphabet of size σ, what can be said about the corresponding string? This kind of problem was first introduced in [46, 41] for the border array, then later considered for various common string data structures; for example, prefix tables [25, 24], KMP arrays [42, 50, 51], cover arrays [29], and many others. We present an $O(\sigma n)$-time algorithm to compute the unique string determined by the Lyndon arrays computed for σ rotations of the alphabet, where σ is the size of the alphabet. We also briefly discuss the possibility of using fewer than σ rotations to determine $\boldsymbol{x}$.

In [78], the authors presented a group of models based on three sets of contrasting features in the existing models of microsatellite evolution. We work on those models with different datasets acquired from Human and Chimp genomes. We will describe the models and parameters. We will try to find the dynamics of the evolution of microsatellite with repeat length 2 by using Human and Chimp data. We present our work on intra-species analysis where we consider different Human populations. We will also try to find the differences between mutation based on the position (inside or outside exon regions) of microsatellites in the genome. We implement the models proposed by Sainudiin et. al [78] to analyze the dynamics of microsatellite evolution. We work on Human and Chimp DNA for inter species comparison and different human populations for intra species comparison. We compare the models using statistical methods and try to find the model which fits the data best.

1.2 Thesis Outline

The remainder of this thesis consists of the following chapters.

Chapter 2 introduces the related background and basic definitions and data structures used in the proposed algorithms.

Chapter 3 consists of our work on quasiperiodicity. We introduce a new algorithm and data structures to compute the minimum enhanced cover array from the prefix table. We will show how to extend the algorithms for indeterminate strings.

Chapter 4 focuses on algorithms to compute Lyndon arrays. We describe the existing algorithms as well as new ones. We also show the comparison of running time between these algorithms.

In Chapter 5, we describe the "reverse engineering" method to find the unique string implied by a collection of Lyndon arrays.

We study microsatellite evolution in Chapter 6, as an application of periodic strings.

Chapter 7 gives some concluding remarks and suggestions for future work.

Chapter 2

Preliminaries and Definitions

In this chapter, we give the notation and terminology used in this thesis. Basic string terminology in this thesis follows [80].

2.1 Basic Definitions

We identify a finite set Σ called an ***alphabet***, whose elements are ***letters***. The cardinality of an alphabet denoted by $\sigma = |\Sigma|$ is the number of distinct letters in the alphabet. A ***string*** $\boldsymbol{x}$ is a sequence of elements drawn from Σ. If every entry in $\boldsymbol{x}$ consists of exactly one element in Σ, then it is called a ***regular string***. Here in this thesis, in general when we talk about string we mean regular string. We represent $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$ of $n \geq 0$ letters where $n = |\boldsymbol{x}|$ is called the ***length*** of the string. We refer to $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$ where $\boldsymbol{x}$ has n ***elements*** $\boldsymbol{x}[1], \boldsymbol{x}[2], ..., \boldsymbol{x}[n]$. The ***empty string*** of length zero is denoted by ϵ. For a given Σ, let Σ^+ be the set of all possible nonempty finite letters of Σ. For example if $\Sigma = \{0, 1\}$, then Σ^+ consists of all distinct nonempty finite sequence of bits (0 or 1). We define Σ^* as $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

$$\begin{array}{cccccccccc} & & \mathit{1} & \mathit{2} & \mathit{3} & \mathit{4} & \mathit{5} & \mathit{6} & \mathit{7} & \mathit{8} \\ \boldsymbol{x} & = & a & b & a & a & b & a & b & a \end{array}$$

Figure 2.1: An example string

An example string is given in Example 1 which will be used throughout this section.

Example 1 *Let* $\Sigma = \{a, b\}$*, so* $\sigma = 2$*. A string* $\boldsymbol{x}$ *drawn from* $\Sigma = \{a, b\}$ *is shown in Figure 2.1*

For $i = 4$*,* $\boldsymbol{x}[i] = \boldsymbol{x}[4] = a$*.*

Corresponding to any pair of integers i and j that satisfy $1 \leq i \leq j \leq n$ we define a ***substring*** $\boldsymbol{u} = \boldsymbol{x}[i..j]$ as $\boldsymbol{u} = \boldsymbol{x}[i]\boldsymbol{x}[i+1]...\boldsymbol{x}[j-1]\boldsymbol{x}[j]$. If $i = j$ then $\boldsymbol{x}[i..j] = x[i]$; if $i > j$, then by convention $\boldsymbol{u} = \epsilon$. If $\boldsymbol{u}$ is substring of $\boldsymbol{x}$ then $\boldsymbol{x}$ is a ***superstring*** of $\boldsymbol{u}$. We say that $\boldsymbol{u}$ ***occurs*** at ***position*** i of $\boldsymbol{x}$ with length $j - i + 1$. If $j - i + 1 < n$, then $\boldsymbol{u}$ is called a ***proper substring***. For the string $\boldsymbol{x}$ in Example 1, a substring $\boldsymbol{u} = \boldsymbol{x}[4...6] = aba$.

Given two strings $\boldsymbol{u}[1..m]$ and $\boldsymbol{v}[1..n]$, the ***concatenation*** of $\boldsymbol{u}$ and $\boldsymbol{v}$ produces a new string $\boldsymbol{x}$ by appending $\boldsymbol{v}$ to $\boldsymbol{u}$. In other words, $|\boldsymbol{x}| = m+n$, $\boldsymbol{x}[1..m] = \boldsymbol{u}$ and $\boldsymbol{x}[m+1...m+n] = \boldsymbol{v}$. The concatenation of $\boldsymbol{u}$ and $\boldsymbol{v}$ is denoted by $\boldsymbol{uv}$.

Two special kinds of substring $\boldsymbol{u}$ have particular importance. If a string $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{v}$ is a substring of $\boldsymbol{x}$ and any of $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ may be the empty string ϵ. If $\boldsymbol{u} = \epsilon$, then $\boldsymbol{v}$ is also a ***prefix*** of $\boldsymbol{x}$. If $\boldsymbol{w} = \epsilon$, then $\boldsymbol{v}$ is also a ***suffix*** of $\boldsymbol{x}$. A string $\boldsymbol{x}$ is a substring, prefix and suffix of itself. If $\boldsymbol{v}$ is a substring, prefix or suffix of $\boldsymbol{x}$ and $|\boldsymbol{v}| < |\boldsymbol{x}|$, then we call $\boldsymbol{v}$ is a ***proper*** substring, prefix or suffix of $\boldsymbol{x}$ as appropriate. So in Example 1, $\boldsymbol{x}[1..3] = aba$ is a prefix of $\boldsymbol{x}$ and $\boldsymbol{x}[5..8] = baba$ is a suffix of $\boldsymbol{x}$. Here the prefix is also a proper prefix and the suffix is a proper suffix of $\boldsymbol{x}$.

If $\boldsymbol{x} = \boldsymbol{x}[1..n]$ has a proper (though possibly empty) prefix $\boldsymbol{u}$ that is also a suffix of $\boldsymbol{x}$, then $\boldsymbol{u}$ is said to be a ***border*** of $\boldsymbol{x}$. In Example 1, $\boldsymbol{u} = aba$ is a border or $\boldsymbol{x}$, since it is both suffix and prefix of $\boldsymbol{x}$. If for some $p \in 1..n$, $\boldsymbol{x}[i] = \boldsymbol{x}[p+i]$ for every $i \in 1..n-p$, then $\boldsymbol{x}$ is said to have ***period*** p. Thus $\boldsymbol{x}$ always has the empty border ϵ and trivial period n. We have the following observation between border and period.

Observation 2 *$\boldsymbol{x}$ has period p if and only if it has a border of length $n-p$.*

Since $\boldsymbol{x}$ has border with length k, then $\boldsymbol{x}[1..k] = \boldsymbol{x}[n-k+1..n]$, according to definition of border, which indicates $\boldsymbol{x}$ has period of length $p = n-k$. In Example 1 we have a period $p = 5 = 8-3$ for $\boldsymbol{x}$, since it has border $\boldsymbol{x}[1..3] = \boldsymbol{x}[6..8]$ of length 3.

2.2 Definitions of Various Regularities

The term ***regularity*** was introduced by Iliopoulos and Mouchard in [58] indicating substrings that occur in some regular pattern inside strings. In this section we define various regularities in strings which we borrowed from [81]. A ***repeat*** in $\boldsymbol{x}$ is a maximum cardinality collection of identical substrings of $\boldsymbol{x}$ that are not necessarily adjacent and occur more than once. A ***repeating substring*** in $\boldsymbol{x}$ is a proper nonempty substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that occurs more than once. In Example 1, $\boldsymbol{u} = aba$ is a repeating substring of string $\boldsymbol{x}$. A repeat in $\boldsymbol{x}$ is a tuple

$$M_{\boldsymbol{x},\boldsymbol{u},r} = \{\boldsymbol{u}; i_1, i_2, ..., i_r\} \tag{2.1}$$

where $\boldsymbol{u}$ is the repeating substring that occurs at positions $i_1, i_2, ..., i_r$, with $1 \leq i_1 < i_2 < ... < i_r \leq n-|\boldsymbol{u}|+1$. In Example 1, $M_{\boldsymbol{x},ab,3} = \{ab; 1, 4, 6\}$ and $M_{\boldsymbol{x},aba,3} = \{aba; 1, 4, 6\}$ are both repeats in $\boldsymbol{x}$.

A ***repetition*** is a maximum length sequence of adjacent repeating substrings; that is, a repetition in $\boldsymbol{x}$ is a repeat (Equation 2.1) which meets the following two constraints on the occurrences of the repeating substring $\boldsymbol{u}$,

1. adjacent, thus $i_{j+1}-i_j = |\boldsymbol{u}|$ for every $j \in 1..r-1$
2. maximal, thus $\boldsymbol{x}[i-|\boldsymbol{u}|..i-1] \neq u$ and $\boldsymbol{x}[i+r|\boldsymbol{u}|..i+(r+1)|\boldsymbol{u}|-1] \neq \boldsymbol{u}$

Of course $\boldsymbol{x}$ itself may be a repetition. If a string or substring is *not* a repetition, we say that it is ***primitive***. A repetition is fully specified by a triple (i, p, r), where

- $\boldsymbol{u} = \boldsymbol{x}[i..i+p-1]$ is the repeating substring and primitive
- $p = |\boldsymbol{u}|$ is the period of the repetition $\boldsymbol{u}^r$
- and r is the number of occurrences of $\boldsymbol{u}$ or ***exponent*** of the repetition

If $r = 2$, the repetition is a ***square*** and if $r = 3$, then it is ***cube***. In Example 1 we have a repetition $\boldsymbol{z} = (i, p, r) = (1, 3, 2)$, since aba is repeated twice and primitive. Also it is a square and we can write $\boldsymbol{z} = (aba)^2$.

A ***run*** [80] (or ***maximal periodicity*** [67]) in $\boldsymbol{x}$ is a 4-tuple (i, p, r, t) which meets the following three conditions:

1. (i, p, r) is a repetition
2. $(i-1, p, r)$ is not a repetition
3. $t \in 0..p-1$ is the maximum integer such that $\boldsymbol{x}[i+rp..i+rp+t-1] = \boldsymbol{u}[1..t]$

We call t the ***tail*** of the run. In Example 1, both (3, 1, 2) and (1, 3, 2) are repetitions (also runs with t = 0), while (4, 2, 2, 1) is a run that implies two repetitions (4, 2, 2)

and (5, 2, 2). Note that (5, 2, 2, 0) cannot be a run since it does not meet the second criterion ((4,2,2) is a repetition). So in a string, two repetitions can overlap but two runs of the same period cannot overlap. We have the following observation based on runs and repetitions.

Observation 3 *Computing all the runs determines all the repetitions.*

Every repetition is either a prefix, suffix or substring of a run. So computing all the runs will compute all the repetitions *implicitly*.

2.3 Data Structures Used in this Thesis

A ***border array*** is an integer array $\boldsymbol{\beta}[1..n]$ in which for every $i \in 1..n$, $\boldsymbol{\beta}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. An $O(n)$ time algorithm was given to compute $\boldsymbol{\beta}$ in [4], which was called the "failure function" algorithm. The following observations of border array are taken from [80].

- $\boldsymbol{\beta}[1] = 0$ (since ϵ is the longest border of $\boldsymbol{x}[1..1]$)
- for $2 \leq i \leq n$, if $\boldsymbol{x}[1..i]$ has a border of length $k > 0$, then $\boldsymbol{x}[1..i-1]$ has a border of length $k-1$; thus, in particular if $1 \leq i \leq n-1$, $\boldsymbol{\beta}[i+1] \leq \boldsymbol{\beta}[i]+1$;
- for $1 \leq i \leq n-1$, $\boldsymbol{\beta}[i+1] = \boldsymbol{\beta}[i]+1$ if and only if $\boldsymbol{x}[1+i] = \boldsymbol{x}[\boldsymbol{\beta}[i]+1]$ (since $\boldsymbol{\beta}[i]+1$ is the position of $\boldsymbol{x}$ immediately to the right of the prefix $\boldsymbol{x}[1..\boldsymbol{\beta}[i]]$ which is the longest border of $\boldsymbol{x}[1..i]$)
- given that $\boldsymbol{b}$ is a border of $\boldsymbol{x}$, then $\boldsymbol{b'}$ is a border of $\boldsymbol{b}$ if and only if $\boldsymbol{b'}$ is a border of $\boldsymbol{x}$

		1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x}$	=	a	b	a	b	a	a	b	a	b	a
$\boldsymbol{\beta}$	=	0	0	1	2	3	1	2	3	2	5
$\boldsymbol{\pi}$	=	10	0	3	0	1	5	0	3	0	1
$\boldsymbol{\gamma}$	=	0	0	0	2	3	0	0	3	0	5

Figure 2.2: Border Array β, Prefix Table π and Cover Array γ

For example, in Figure 2.2, for string $\boldsymbol{x} = ababaababa$, $ababa$ is a border of $\boldsymbol{x}$ and a is a border of $ababa$, so a is a border of $\boldsymbol{x}$.

A ***prefix table*** is an integer array $\boldsymbol{\pi}[1..n]$ in which for every $i \in 1..n$, $\boldsymbol{\pi}[i]$ is the length of the longest substring at position i of $\boldsymbol{x}$ that equals a prefix of $\boldsymbol{x}$. Figure 2.2 shows prefix table π for string $\boldsymbol{x} = ababaababa$. An $O(n)$ time construction algorithm for the prefix table was first described some 30 years ago in the Main & Lorentz all-repetitions algorithm [68]. The same algorithm is given in [27], but then a modified construction algorithm was proposed in [28]. Two other distinct algorithms on a "compressed prefix" table are described in [82]. The prefix table gives rise to an easy and efficient pattern-matching algorithm: given pattern $\boldsymbol{u}$ and text $\boldsymbol{v}$, form $\boldsymbol{x} = \boldsymbol{uv}$ and compute $\boldsymbol{\pi}$; then for $i \in |\boldsymbol{u}|+1..|\boldsymbol{x}|, x[i..i+|\boldsymbol{u}|-1] = \boldsymbol{v}[i-|\boldsymbol{u}|..i-1]$ is an occurrence of $\boldsymbol{u}$ in $\boldsymbol{v}$ if and only if $\boldsymbol{\pi}[i] > |\boldsymbol{u}|$.

Since for every $i \in 2..n$, $\boldsymbol{\pi}[i] = 0$ if and only if $\boldsymbol{x}[i] \neq \boldsymbol{x}[1]$, the number of nonzero elements in $\boldsymbol{\pi}$ is exactly m, where m is the number of occurrences of $\boldsymbol{x}[1]$ in $\boldsymbol{x}$ other than at position 1. The expected value of m is $m/\sigma-1$, a quantity less than $n/2$ for $\sigma \geq 2$. This led to the idea of ***compressed prefix array*** using integer arrays $POS[1..m]$ and $LEN[1..m]$, defined as follows

$$\boldsymbol{\pi}[POS[j]] = LEN[j]$$

iff the jth nonzero entry in $\boldsymbol{\pi}$ occurs in position $POS[j]$ and takes the value $LEN[j]$. Sometimes the compressed prefix array is called the POS/LEN version of the prefix array. For larger alphabets, the compressed form of the prefix array saves considerable storage space compared to the border array or the uncompressed prefix array. In [82], the authors introduced the idea of compressed prefix array and give an algorithm to compute $\boldsymbol{\pi}$ in compressed form POS/LEN.

It has long been folklore that $\boldsymbol{\beta}$ and $\boldsymbol{\pi}$ are "equivalent", but it has only recently been made explicit [19] that each can be computed from the other in linear time. However as we discuss below, this equivalence holds only for regular strings $\boldsymbol{x}$ in which each entry $\boldsymbol{x}[i]$ is constrained to be a single element of the underlying alphabet Σ.

A letter δ drawn from alphabet $\Sigma = \{\ell_1, \ell_2, \ldots, \ell_\sigma\}$ is said to be ***regular*** if $\delta = \ell_j$ for some $j \in 1..\sigma$; otherwise, if $\delta = \Sigma_k$, a subset of Σ of size $k > 1$, then δ is said to be ***indeterminate***. We say that two letters δ_1 and δ_2 ***match***, written $\delta_1 \approx \delta_2$, if and only if $\delta_1 \cap \delta_2 \neq \emptyset$; thus regular letters match if and only if they are equal, while for indeterminate letters this is not true. Two strings $\boldsymbol{x}[1..n]$ and $\boldsymbol{y}[1..m]$ are ***equal*** to each other if and only if $m = n$ and for all $1 \leq i \leq n$, $\boldsymbol{x}[i] = \boldsymbol{y}[i]$. Two strings $\boldsymbol{x}[1..n]$ and $\boldsymbol{y}[1..m]$ ***match*** each other if and only if $m = n$ and for all $1 \leq i \leq n$, $\boldsymbol{x}[i] \approx \boldsymbol{y}[i]$. In fact, "match" is in general nontransitive, as the following example shows:

$$\delta_1 = \ell_1, \delta_2 = \ell_2, \delta_3 = \{\ell_1, \ell_2\} \Longrightarrow \delta_1 \approx \delta_3 \approx \delta_2 \text{ but } \delta_1 \not\approx \delta_2.$$

Similarly, a string $\boldsymbol{x}$ on Σ is said to be ***indeterminate*** if at least one of the letter $\boldsymbol{x}[i]$, $1 \leq i \leq n$, is indeterminate. But note that a string — for example, $\boldsymbol{x} = \{c, g\}, a, t, \{c, g\}$ — may be indeterminate but at the same time give rise only to

transitive matches; such strings are called ***essentially regular***. Strings with letters restricted to either single elements ℓ_j of Σ or else Σ itself (called a ***hole*** or ***don't care*** letter) were introduced in [44] and have been intensively studied as ***partial words*** since 2003 by Blanchet-Sadri (see [18]). In the 1980s Abrahamson [3] dealt with the general case defined above (which he called "generalized strings"); in this century, beginning with [55], there has been continued interest in indeterminate strings — for example, [56, 83].

An important consequence of the nontransitivity of match is that the border array no longer correctly describes *all* of the borders of $\boldsymbol{x}$: it is no longer necessarily true, as for regular strings, that if $\boldsymbol{u}$ is the longest border of $\boldsymbol{v}$, in turn the longest border of $\boldsymbol{x}$, then $\boldsymbol{u}$ is a border of $\boldsymbol{x}$. An example is given in Figure 2.3. On the other hand, the prefix array retains all its properties for indeterminate strings $\boldsymbol{x}$ and, in particular, correctly identifies all the borders of every prefix of $\boldsymbol{x}$ [19]. [82] describes algorithms to compute the prefix table of an indeterminate string; conversely, [24] proves that there exists an indeterminate string corresponding to every feasible prefix table, while [2] describes an algorithm to compute the lexicographically least indeterminate string determined by any given feasible prefix table. There is thus a many-many relationship between the set of all strings, whether indeterminate or regular, and the set of all prefix tables.

$$\begin{array}{rcccc} & 1 & 2 & 3 & 4 \\ \boldsymbol{x} = & \{a,g\} & \{a,t\} & \{c,g\} & \{c,t\} \\ \boldsymbol{\beta x} = & 0 & 1 & 1 & 2 \end{array}$$

Figure 2.3: $\boldsymbol{x'} = \{a,g\}\{a,t\}$ (or $\{c,g\}\{c,t\}$) is a border of $\boldsymbol{x}$; but neither $\boldsymbol{x''} = \{a,g\}$ nor $\{a,t\}$ is a border of $\boldsymbol{x}[3..4]$, and neither $\boldsymbol{x''} = \{c,g\}$ nor $\{c,t\}$ is a border of $\boldsymbol{x}[1..2]$.

In 1990 Apostolico & Ehrenfeucht [11] introduced the idea of quasiperiodicity: a ***cover*** of a string $\boldsymbol{x}$ is a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position in $\boldsymbol{x}$ is contained in an occurrence of $\boldsymbol{u}$; $\boldsymbol{u}$ is then said to ***cover*** $\boldsymbol{x}$, which is said to be ***quasiperiodic*** with ***quasiperiod*** $|\boldsymbol{u}|$. Thus, for example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaaba$. The ***cover array*** is denoted by $\boldsymbol{\gamma}$, where $\boldsymbol{\gamma}[i]$ gives the length j of the longest cover of $\boldsymbol{x}[1..i]$. Several linear-time algorithms were proposed for the computation of covers [12, 20, 71, 72] culminating in an algorithm [66] to compute the ***cover array*** $\boldsymbol{\gamma}$. Since the longest cover of $\boldsymbol{x}[1..j]$ is also a cover of $\boldsymbol{x}[1..i]$, $\boldsymbol{\gamma}$ implicitly specifies all the covers of every prefix of $\boldsymbol{x}$. A recent paper [8] extends the computation of $\boldsymbol{\gamma}$ to indeterminate strings. Figure 2.2 shows cover array γ for string $\boldsymbol{x} = ababaababa$. This array tells us that $\boldsymbol{x}$ has cover $\boldsymbol{u} = ababa$ of length 5, but also, since $\gamma[\gamma[10]] = \gamma[5] = 3$, cover $\boldsymbol{v} = aba$ of length 3.

For indeterminate strings, there are two natural analogues of cover. A string $\boldsymbol{x}$ is said to have a ***sliding cover*** of length κ if and only if

1. $\boldsymbol{x}$ has a suffix $\boldsymbol{v}$ of length κ; and
2. $\boldsymbol{x}$ has a proper prefix $\boldsymbol{u}$, $|\boldsymbol{u}| \geq |\boldsymbol{x}|-\kappa$, with suffix $\boldsymbol{v'} \approx \boldsymbol{v}$; and
3. either $\boldsymbol{u} = \boldsymbol{v'}$ or else $\boldsymbol{u}$ has a cover of length κ

A sliding cover requires the adjacent or overlapping substrings of $\boldsymbol{x}$ to match. However it leaves open the possibility that nonadjacent elements of cover do not match because of the nontransitivity of matching. For example,

$$\boldsymbol{x} = \{a, b\}c\{a, c\}\{a, c\}ca$$

has a sliding cover of length $\kappa = 2$ because $\{a,b\}c \approx \{a,c\}\{a,c\} \approx ca$, but $\{a,b\}c \not\approx ca$. An indeterminate string $\boldsymbol{x}$ may have a sliding cover $\boldsymbol{v}$ which is not a substring of $\boldsymbol{x}$. This motivated the idea of ***rooted cover*** of length κ, where every covering substring is required to match, not the preceding entry in the cover, but rather the prefix $\boldsymbol{x}$ of length κ. A rooted cover is defined simply by changing "suffix" to "prefix" in the second condition of the above definition of sliding cover. The string

$$\boldsymbol{x} = \{a,b\}c\{a,c\}\{a,c\}ac$$

has both a sliding cover and rooted cover of length 2.

If Σ is ordered, then ***lexicographic order*** (dictionary order) is the corresponding induced order on the elements of Σ^*. The formal definition is given in Definition 4.

Definition 4 *Suppose we are given two strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..m]$, where $n \geq 0, m \geq 0$. We say $\boldsymbol{x} < \boldsymbol{y}$ ($\boldsymbol{x}$ is lexicographically less than $\boldsymbol{y}$) if and only if one of the following (mutually exclusive) conditions holds:*

- $n < m$ *and* $\boldsymbol{x}[1..n] = \boldsymbol{y}[1..m]$
- $\boldsymbol{x}[1..i-1] = \boldsymbol{y}[1..i-1]$ *and* $\boldsymbol{x}[i] < \boldsymbol{y}[i]$ *for some integer* $i \in 1..min(n,m)$

For a given alphabet $\Sigma = \{a,b\}$ with $a < b$, then for strings aba and abb, according to lexicographic order, $aab < abb$.

The ***suffix array*** SA of $\boldsymbol{x}$ is defined by $SA[i] = j, 1 \leq i \leq n$, where $x[j..n]$ is the ith smallest suffix of $\boldsymbol{x}$ in lexicographic order. It is one of the prominent data structures in string algorithms since its introduction in [69, 70]. Given a suffix array, SA, and the corresponding ***inverse suffix array***, ISA, is defined by $ISA[i] = j$ iff

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

Figure 2.4: Suffix Array and Inverse Suffix Array

		1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x}$	=	a	b	a	a	b	a	b	a	a	b
$\boldsymbol{\lambda}$	=	2	1	5	2	1	2	1	3	2	1
$\boldsymbol{\mathcal{L}}$	=	2	2	7	5	5	7	7	10	10	10

Figure 2.5: For $i = 3$, we have two Lyndon words starting at the position: aab and $aabab$. Since $aabab$ is longest, therefore $\boldsymbol{\lambda}[3] = 5$

$SA[j] = i$. So for a position i, $ISA[i]$ gives the position in the list of sorted suffix of the suffix $\boldsymbol{x}[i..n]$. In Figure 2.4, $ISA[6] = 1$, which means, $\boldsymbol{x}[6..8] = aab$ is least among all the suffices of $\boldsymbol{x}$.

If $\boldsymbol{x} = \boldsymbol{uv}$ for some $\boldsymbol{u}$ and nonempty $\boldsymbol{v}$, then $\boldsymbol{vu}$ is said to be the $|\boldsymbol{u}|^{\text{th}}$ ***rotation*** of $\boldsymbol{x}$, written $\boldsymbol{vu} = R_{|\boldsymbol{u}|}(x)$. A primitive string $\boldsymbol{x}$ that is lexicographically least among all its rotations $R_k(\boldsymbol{x}), k = 0, 1, \ldots, |\boldsymbol{x}| - 1$, is said to be a ***Lyndon word***. For example, if $\boldsymbol{x} = aab$, then it has two other rotations namely aba and baa. Since aab is least among all of them and primitive, therefore aab is a Lyndon word. Note that if $\boldsymbol{x}$ is not primitive (for example, $\boldsymbol{x} = abab$), then there cannot be a single rotation that is lexicographically least.

The ***Lyndon array*** $\boldsymbol{\lambda} = \boldsymbol{\lambda_x}[1..n]$ (equivalently, $\boldsymbol{\mathcal{L}} = \boldsymbol{\mathcal{L}_x}[1..n]$) of a given nonempty string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ gives at each position i the length (equivalently, the end position) of the longest Lyndon word starting at i. Figure 2.5 depicts both $\boldsymbol{\lambda_x}$ and $\boldsymbol{\mathcal{L}_x}$ for string $\boldsymbol{x} = abaababaab$.

Chapter 3

Enhanced Covers of Regular & Indeterminate Strings using Prefix Tables

The contents of this chapter have been published in [38].

3.1 Introduction

The concept of *periodicity* is fundamental to combinatorics on words and related algorithms: it is difficult to imagine a research contribution that does not somehow involve periods of strings. Nevertheless, only few strings have a period other than the trivial period. And even though the cover of a string can provide useful information, quasiperiodic strings are on the other hand infrequent among strings in general. Another approach to string covering was therefore proposed in [59]: a set $U_k = U_k(\boldsymbol{x})$ of strings, each of length k, is said to be a ***minimum*** k***-cover*** of $\boldsymbol{x}$ if every position in

$\boldsymbol{x}$ lies within some occurrence of an element of U_k, and no smaller set of k-strings has this property. Thus $U_2(abaababab) = U_2(ababaaba) = \{ab, ba\}$. In [26] the computation of U_k was shown to be NP-complete, though an approximate polynomial-time algorithm was presented in [57].

Recall that a border of $\boldsymbol{x}$ is a possibly empty proper prefix of $\boldsymbol{x}$ that is also a suffix: every nonempty string has a border of length zero. Recently the promising idea of an ***enhanced cover*** was introduced [45]; that is, a border $\boldsymbol{u}$ of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ that covers a maximum number $m \leq n$ of positions in $\boldsymbol{x}$. Then the ***minimum enhanced cover*** $\texttt{mec}(\boldsymbol{x})$ is the *shortest* border $\boldsymbol{u}$ that covers m positions, and [45] presented an algorithm to compute $\texttt{mec}(\boldsymbol{x})$ in $\Theta(n)$ time. Thus for $\boldsymbol{x} = abaababab$, $\texttt{mec}(\boldsymbol{x}) =$ $\texttt{ab}$. Further, on the analogy of the cover array defined above, the authors proposed the ***minimum enhanced cover array*** $\texttt{MEC}_{\boldsymbol{x}}$ — for every $i \in 1..n$, $\texttt{MEC}_{\boldsymbol{x}}[\texttt{i}] =$ $|\texttt{mec}(\boldsymbol{x}[\texttt{1..i}])|$, the length of the minimum enhanced cover of $\boldsymbol{x}[1..i]$ — and showed how to compute it in $\mathcal{O}(n \log n)$ time. In this chapter we introduce in addition the $\texttt{CMEC}$ array, where $\texttt{CMEC}[i]$ specifies the number of positions in $\boldsymbol{x}$ covered by the border of length $\texttt{MEC}[i]$. Thus, for example, $\texttt{MEC}_{\texttt{abaababab}} = \texttt{001123232}$ and $\texttt{CMEC}_{abaababab} =$ 002346688.

In order to compute $\texttt{MEC}_{\boldsymbol{x}}$, the authors of [45] made use of a variant of the border array. In this chapter we adopt a different approach to the computation of $\texttt{MEC}_{\boldsymbol{x}}$, using, instead of a border array, the prefix table $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$.

In Section 3.2 we outline the basic methodology and data structures used to compute the minimum enhanced cover array from the prefix table, while illustrating the ideas with an example. Then Section 3.3 provides a proof of the algorithm's correctness, as well as an analysis of its complexity, both worst and average case.

Section 3.4 defines enhanced left covers and enhanced left seeds and describes extensions of the basic MEC algorithm to compute these. In Section 3.5 we discuss the practical application of our algorithms, in terms of time and space requirements, and compare our prefix-based implementation with the border-based implementation of [45]. Section 3.6 shows how to extend the various enhanced cover array algorithms to indeterminate strings, while Section 3.7 summarizes our results and suggests future research directions.

3.2 Methodology

In this section we describe the computation of $\mathtt{MEC}_{\boldsymbol{x}}$, the enhanced cover array of $\boldsymbol{x}$, based on the prefix array $\boldsymbol{\pi}$. Since every minimum enhanced cover of $\boldsymbol{x}$ is also a border of $\boldsymbol{x}$, we are initially interested in the covers of prefixes of $\boldsymbol{x}$. For this purpose we need arrays whose size is $\mathtt{B}$, the maximum length of any border of any prefix of $\boldsymbol{x}$. Noting that $\mathtt{B}$ must be the maximum entry in the prefix array $\boldsymbol{\pi}$, we write $\mathtt{B} = \max_{2\leq \mathtt{i} \leq \mathtt{n}} \boldsymbol{\pi}[\mathtt{i}]$.

Definition 1 *In the* ***maximum no cover*** *array* $\mathtt{MNC} = \mathtt{MNC}[1..\mathtt{B}]$*, for every* $q \in 1..\mathtt{B}$*,* $\mathtt{MNC}[q] = q'$*, where* q' *is the maximum integer in* $1..q$ *such that the prefix* $\boldsymbol{x}[1..q']$ *has no cover — that is, such that* $\boldsymbol{\gamma}[q'] = 0$*.*

As shown in Figure 3.1, once $\mathtt{B}$ is computed in $\Theta(n)$ time from the prefix array $\boldsymbol{\pi}$, $\mathtt{MNC}$ can be easily computed in $\Theta(\mathtt{B})$ time using the cover array $\boldsymbol{\gamma}[1..\mathtt{B}]$ of $\boldsymbol{x}[1..\mathtt{B}]$. Note that the entries in $\mathtt{MNC}$ are monotone nondecreasing with $1 \leq \mathtt{MNC}[q] \leq q$ for every $q \in 1..\mathtt{B}$. The following is fundamental to the execution of our main algorithm:

Observation 2 *If a prefix* $\boldsymbol{v} = \boldsymbol{x}[1..q]$ *of* $\boldsymbol{x}$ *has a cover* $\boldsymbol{u}$*, then* $\boldsymbol{v} \neq \texttt{mec}(\boldsymbol{x})$ *(since* $|\boldsymbol{u}| < q$ *and* $\boldsymbol{u}$ *covers every position covered by* $\boldsymbol{v}$*).*

```
procedure Compute_MNC(n, π; B, γ, MNC)
B ← π[2]
for i ← 3 to n do
    B ← max(B, π[i])
▷ Compute γ[1..B] of x[1..B] using
▷ the algorithm Compute_CPR of [8].
Compute_CPR(B, π; γ)
▷ Note that MNC can overwrite γ.
for q ← 1 to B do
    if γ[q] = 0 then MNC[q] ← q
    else MNC[q] ← MNC[q−1]
```

Figure 3.1: Computing MNC from the prefix array $\boldsymbol{\pi}[1..n]$ and the cover array $\boldsymbol{\gamma}[1..B]$.

Thus MNC[q] specifies an upper bound $q' \in 1..q$ on the length of a minimum enhanced cover of $\boldsymbol{x}$. Two other arrays are required for the computation, both of length B:

Definition 3 *For every* $q \in 1..\texttt{B}$*:*

- PR[q] *is the rightmost position in* $\boldsymbol{x}$ *at which the prefix* $\boldsymbol{x}[1..q]$ *occurs;*
- CPR[q] *is the number of positions in* $\boldsymbol{x}$ *covered by occurrences of* $\boldsymbol{x}[1..q]$.

An example of the arrays introduced thus far is given in Figure 3.2. Note that for $\boldsymbol{x}[1..9]$ and $\boldsymbol{x}[1..10]$, there are actually *two* borders that cover a maximum number of positions; in each case the border of minimum length is identified in MEC.

The algorithm Compute_MEC is shown in Figure 3.3. In the first stage, B and MNC are computed and the arrays CMEC, PR and CPR are initialized. Then every position $i > 1$ such that $q = \boldsymbol{\gamma}[i] > 0$ is considered. Using MNC, the longest prefix $\texttt{Q}' = \boldsymbol{x}[1..q']$

		1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x}$	=	a	b	a	b	a	a	b	a	b	a
$\boldsymbol{\pi}$	=	10	0	3	0	1	5	0	3	0	1
$\boldsymbol{\gamma}$	=	0	0	0	2	3					
MNC	=	1	2	3	3	3					
PR	=	10	8	8	6	6					
CPR	=	6	8	10	8	10					
MEC	=	0	0	1	2	3	1	2	3	2	3
CMEC	=	0	0	2	4	5	4	6	8	8	10

Figure 3.2: All the arrays required to compute MEC and CMEC arrays

of $\boldsymbol{x}[1..q]$ that does *not* have a cover is identified; for prefixes of $\boldsymbol{x}[1..q]$ that do have a cover, the appropriate PR and CPR values have already been updated. There are two main steps in the processing of $\mathtt{Q}'$:

- Since i has now become the rightmost occurrence of $\mathtt{Q}'$ in $\boldsymbol{x}[1..i]$, we must set $\mathtt{PR}[q'] \leftarrow i$ and increment the corresponding number $\mathtt{CPR}[q']$ of positions covered.

- If the number $\mathtt{CPR}[q']$ of positions covered by occurrences of $\mathtt{Q}'$ exceeds $\mathtt{CMEC}[i+q-1]$, then CMEC and MEC must be updated accordingly.

These steps are repeated recursively for the longest proper prefix of $\mathtt{Q}'$ that does not have a cover.

3.3 Correctness & Complexity of Compute_MEC

We begin by proving the correctness of Compute_MEC, which depends on the prior computation of $\boldsymbol{\pi} = \boldsymbol{\pi_x}$ [19]. Consider first procedure Compute_MNC, where B is computed, followed by the cover array $\boldsymbol{\gamma}[1..\mathtt{B}]$. Then for every $q \in 1..\mathtt{B}$, $\mathtt{MNC}[q] \leftarrow q$ whenever there is no cover of $\boldsymbol{x}[1..q]$, with $\mathtt{MNC}[q] \leftarrow \mathtt{MNC}[q-1]$ otherwise, an easy and straightforward calculation.

```
procedure Compute_MEC(π; MEC, CMEC)
n ← |π|
Compute_MNC(n, π; B, γ, MNC)
MEC ← 0^n; CMEC ← 0^n; PR ← 1^B
for q ← 1 to B do CPR[q] ← q
for i ← 2 to n do
    q ← π[i]
▷ x[i..i+q−1] = x[1..q].
    while q > 0 do
▷ x[1..q'] is the longest prefix of x[1..q] without a cover.
        q' ← MNC[q]
▷ x[1..q'] also occurs at i: update CPR[q'] & PR[q'].
        if i−PR[q'] < q' then
            CPR[q'] ← CPR[q']+i−PR[q']
        else
            CPR[q'] ← CPR[q']+q'
        PR[q'] ← i
▷ Update MEC & CMEC accordingly for interval i..i+q'−1.
        if CPR[q'] ≥ CMEC[i+q'−1] then
            MEC[i+q'−1] ← q'
            if CPR[q'] > CMEC[i+q'−1] then
                CMEC[i+q'−1] ← CPR[q']
        q ← q'−1
```

Figure 3.3: Computing MEC amd CMEC from the prefix array $\boldsymbol{\pi}$.

Compute_MEC then independently considers positions $i = 2, 3, \ldots, n$ for which $\boldsymbol{\pi}[\boldsymbol{i}] > 0$; that is, such that a border of $\boldsymbol{x}$ of length $q = \boldsymbol{\pi}[i]$ begins at i. The internal **while** loop then processes in decreasing order of length the prefixes $\mathtt{Q'} = \boldsymbol{x}[1..q']$ of $\boldsymbol{x}[1..q]$ that have no cover — and that therefore, by Observation 2, can possibly be minimum enhanced covers of $\boldsymbol{x}[1..i{+}q'{-}1]$. Thus, for every $i \in 2..n$, all such borders $\boldsymbol{x}[1..q] = \boldsymbol{x}[i..i{+}q{-}1]$ are considered and, for each one, all such prefixes $\mathtt{Q'}$. For each q':

- the number $\mathtt{CPR[q']}$ of positions covered by $\mathtt{Q'}$ is updated, as well as the position $\mathtt{PR[q']} = \mathtt{i}$ of rightmost occurrence of $\mathtt{Q'}$;

- $\mathtt{MEC}[\mathtt{i}+\mathtt{q}'-\mathtt{1}]$ and $\mathtt{CMEC}[\mathtt{i}+\mathtt{q}'-\mathtt{1}]$ are updated accordingly for sufficiently large $\mathtt{CPR}[\mathtt{q}']$.

We claim therefore that

Theorem 4 *For a given string $\boldsymbol{x}$, Compute_MEC correctly computes the minimum enhanced cover array $\mathtt{MEC}_{\boldsymbol{x}}$ and the number $\mathtt{CMEC}_{\boldsymbol{x}}$ of positions covered by it, based solely on the prefix array $\boldsymbol{\pi}_{\boldsymbol{x}}$.*

We have seen that in aggregate Compute_MEC processes a subset of the nonempty borders of every prefix $\boldsymbol{x}[1..i]$, devoting $\mathcal{O}(1)$ time to each one. As we have seen, each border $\mathtt{Q}'$ in each such subset is constrained to have no cover. We say that a string $\boldsymbol{v}$ is ***strongly periodic*** if it has a border $\boldsymbol{u}$ such that $|\boldsymbol{u}| \geq |\boldsymbol{v}|/2$; otherwise $\boldsymbol{v}$ is said to be ***weakly periodic***. Observe that the borders $\mathtt{Q}'$ must all be weakly periodic; if not, then they would have a cover $\boldsymbol{u}$ with $|\boldsymbol{u}| \geq |\boldsymbol{v}|/2$. In [45] the following result is proved:

Lemma 5 *[45] There are at most $\log_2 n$ weakly periodic borders of a string of length n.*

It follows then that for each $i \in 2..n$, there are at most $\log_2 i$ borders considered, thus overall requiring $\mathcal{O}(n \log n)$ time.

The space requirement of Compute_MEC, apart from the $\boldsymbol{\pi}$, $\mathtt{MEC}$ and $\mathtt{CMEC}$ arrays, each of length n, consists of three integer arrays ($\mathtt{MNC}$ (overwriting $\boldsymbol{\gamma}$), $\mathtt{PR}$, $\mathtt{CPR}$), each of length $\mathtt{B} < \mathtt{n}$. Thus

Theorem 6 *In the worst case, Compute_MEC computes $\mathtt{MEC}$ and $\mathtt{CMEC}$ from $\boldsymbol{\pi}$ using*

(*a*) $\mathcal{O}(n \log n)$ *time;*

(b) *three additional arrays* $1..\mathtt{B}$ *of integers* $1..n$, *(*`MNC`*,* `PR`*,* `CPR`*) thus* $\Theta(\mathtt{B} \log \mathtt{n})$ *bits of space.*

Now we would like to analyze the expected (average) case behaviour of Compute_MEC. To do this, we prove and make use of a combinatorial result of independent interest. We first discuss this new interesting result in Section 3.3.1 and then we use it to complete the average case analysis in Section 3.3.2.

3.3.1 Combinatorics on the border length

Here we show that the expected length of the longest border of a string $\boldsymbol{x}$ approaches a limit as $|\boldsymbol{x}|$ tends to infinity, the limit depending on the alphabet size. For a binary alphabet it is approximately 1.64. We use the following notation: $\sigma = |\Sigma|$ is the alphabet size, $B(w)$ is the length of the longest border of string w, and $B_k(w)$ is the length of the longest border of string w which has length at most k (so ignoring any borders longer than k). Thus if $\boldsymbol{x} = babaababab babaabab$, then $B(\boldsymbol{x}) = 8$, since $\boldsymbol{x}$ has longest border $babaabab$, and $B_4(\boldsymbol{x}) = 3$, since the longest border of $\boldsymbol{x}$ which has length at most 4 is aba. Let W_n be the set of all strings of length n on an alphabet of size σ. Since W_0 contains only the empty string, we have $|W_0| = 1$. We start with describing the nested intervals theorem.

Theorem 7 *If* $a_1, a_2, ...a_n$ *and* $b_1, b_2, ..., b_n$ *are real numbers and*

$$[a_1,\, b_1] \;\supseteq\; [a_2,\, b_2] \;\supseteq\; [a_3,\, b_3] \;\supseteq \dots$$

is a sequence of nested closed intervals, then

$$\bigcap_{n=1}^{\infty}[a_n,\, b_n] \neq \varnothing.$$

If also $\lim_{n\to\infty}(b_n - a_n) = 0$, *then the infinite intersection consists of a unique real number.*

We need the following lemma.

Lemma 8 *(Jamie Simpson)The number of strings of length* n *on an alphabet of size* σ *which have a border of length exactly* k *(not necessarily the longest border) is* σ^{n-k}.

Proof. *A string with border of length* k *is periodic with period* $n-k$ *and so is determined by its length* $n-k$ *prefix. This prefix can be chosen in* σ^{n-k} *ways.* □

We also need the following formula (obtainable using a computer algebra system):

Lemma 9 $\sum_{i=a}^{b} m\sigma^m = \frac{\sigma^{b+1}(\sigma\,(b+1)-\sigma-b-1)}{(\sigma-1)^2} - \frac{\sigma^a(a\sigma-a-\sigma)}{(\sigma-1)^2}$.

Clearly $|W_n| = \sigma^n$. The expected size of the longest border of a string of length n on an alphabet of size σ is therefore

$$\overline{B}(n) = \frac{1}{\sigma^n}\sum_{w\in W_n} B(w). \tag{3.1}$$

Similarly, the expected size of the longest border not exceeding k is

$$\overline{B}_k(n) = \frac{1}{\sigma^n}\sum_{w\in W_n} B_k(w). \tag{3.2}$$

Clearly $B(w) \geq B_k(w)$ so

$$\overline{B}(n) \geq \overline{B}_k(n). \tag{3.3}$$

Note that if $n \geq 2k$ then $W_n = \{uvw : u \in W_k, x \in W_{n-2k}, v \in W_k\}$ and so

$$\overline{B}_k(n) = \frac{1}{\sigma^n} \sum_{u \in W_k} \sum_{x \in W_{n-2k}} \sum_{v \in W_k} B_k(uxv). \tag{3.4}$$

Now $B_k(uxv) = B_k(uv)$ so if $n \geq 2k$,

$$\begin{aligned} \overline{B}_k(n) &= \frac{1}{\sigma^n} \sum_{u \in W_k} \sum_{v \in W_k} B_k(uv) \sum_{x \in W_{n-2k}} 1 \qquad (3.5) \\ &= \frac{\sigma^{n-2k}}{\sigma^n} \sum_{u \in W_k} \sum_{v \in W_k} B_k(uv) \\ &= \frac{1}{\sigma^{2k}} \sum_{w \in W_{2k}} B_k(w) \\ &= \overline{B}_k(2k). \end{aligned}$$

With (3.3) we then have, for $n \geq 2k$,

$$\overline{B}(n) \geq \overline{B}_k(2k). \tag{3.6}$$

Now any border that is counted in the right hand side of (3.1) but not counted on the right hand side of (3.2) has length at least $k+1$. The sum of the lengths of such borders is, by Lemma 8,

$$\sum_{m=k+1}^{n} m\sigma^{n-m}.$$

So, by Lemma 9 and (3.5),

$$\begin{aligned}\overline{B}(n) &\leq \frac{1}{\sigma^n}(\sum_{w\in W_n} B_k(w) + \sum_{m=k+1}^{n} m\sigma^{n-m}) \\ &= \overline{B}_k(n) + \frac{1}{\sigma^n}(\frac{\sigma^{n-k+1}k + \sigma^{n-k+1} - \sigma^{n-k}k - \sigma\, n - \sigma + n}{(\sigma-1)^2}) \\ &< \overline{B}_k(n) + \frac{\sigma^{-k+1}k + \sigma^{-k+1} - \sigma^{-k}k - \sigma}{(\sigma-1)^2} \\ &= \overline{B}_k(2k) + O(k\sigma^{-k}).\end{aligned} \tag{3.7}$$

Thus for all $n \geq 2k$,

$$\overline{B}_k(2k) \leq \overline{B}(n) \leq \overline{B}_k(2k) + O(\sigma^{-k}),$$

so $\overline{B}(n)$ is contained in a sequence of nested intervals whose lengths have limit 0. By Theorem 7, this means the limit of $\overline{B}_n$ exists. □

Using (3.3) and (3.7) with $k = 11$ we find that $lim_{n\to\infty}\overline{B}(n)$ lies in the interval $(1.6356, 1.6420)$ for binary alphabets. For ternary alphabets using $k = 6$ the limit lies in $(0.6811, 0.6864)$.

3.3.2 Average case analysis

With the combinatorics of Section 3.3.1 at our disposal, we can easily complete the average case analysis of Compute_MEC. Clearly, this depends critically on the expected length of the maximum border of $\boldsymbol{x}[1..n]$; that is, the expected value of `B`. Now, from Section 3.3.1 we know that for a given alphabet size, `B` approaches a limit as n goes to infinity. The limit is approximately 1.64 for binary alphabets, 0.69 for ternary alphabets, and monotone decreasing in alphabet size. Thus

Theorem 10 *In the average case, Compute_MEC requires $\mathcal{O}(n)$ time and $\Theta(\log n)$ additional bits of space.*

3.4 Enhanced Left Covers and Left Seeds

In [45] the authors extended the concept of the enhanced cover to the notion of enhanced left covers and enhanced left seeds as follows. A proper prefix $\boldsymbol{u}$ of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an ***enhanced left cover*** (respectively, ***enhanced left seed***) of $\boldsymbol{x}$ if $\boldsymbol{u}$ has at least two occurrences in $\boldsymbol{x}$ and the number of positions in $\boldsymbol{x}$ that lie within occurrences of $\boldsymbol{u}$ in $\boldsymbol{x}$ (respectively, a superstring of $\boldsymbol{x}$) is the maximum over all such prefixes. For example, $\boldsymbol{x} = abaabab$ has enhanced left covers $\boldsymbol{u} = ab$ and aba, both covering six positions in $\boldsymbol{x}$, with both occurring twice in $\boldsymbol{x}$; but its only enhanced left seed is $\boldsymbol{u} = aba$, which lies *three* times in the superstring $\boldsymbol{x}a$ and so covers all seven positions of $\boldsymbol{x}$.

Thus, like the minimum enhanced cover array, we can analogously consider the minimum enhanced left cover (MELC) array and the minimum enhanced left seed (MELS) array. In fact, [45] provides an $O(n \log n)$ algorithm for computing the MELC array and an $O(n^2)$ algorithm for computing the MELS array. Both of these algorithms are extended from the border-based algorithm of [45] that computes `MEC`. In this section, we discuss how we can extend our (prefix-based) Compute_MEC algorithm to compute the MELC and MELS arrays.

3.4.1 Minimum Enhanced Left Cover Array (MELC)

We first consider the computation of the MELC array. As before, we also compute an associated array CMELC, analogous to the CMEC array; that is, CMELC[i] counts the number of positions in $\boldsymbol{x}$ covered by the prefix of length MELC[i]. Here, for clarity of presentation, we describe the algorithm assuming that the MEC and CMEC arrays are already computed. However, in practice we will compute MELC and CMELC on the fly as part of the computation of MEC and CMEC in Compute_MEC. The essential thing to note is that for MELC we only need to consider a prefix rather than a border of $\boldsymbol{x}$. The central argument for the computation of the MELC array from the MEC and CMEC arrays is presented in Proposition 11 below. For this purpose, we need to define two functions, MaxCount and CorLen, that work in tandem on a prefix of CMEC, with MaxCount returning the maximum value in the prefix and CorLen returning the corresponding value in the MEC array. Importantly, in case of a tie for the maximum value, CorLen will return the value from the MEC array that is lower. For example, consider Figure 3.2 and suppose that we are considering CMEC[1..6]. Then MaxCount will return CMEC[3] = 5 and CorLen will return MEC[3] = 3. Notice that we have CMEC[8] = CMEC[9] = 8. So, if we consider CMEC[1..9], then MaxCount will return 8 and CorLen will return 2, because MEC[9] = 2 is lower than MEC[8] = 3.

Proposition 11 *Suppose* MEC *and* CMEC *have been computed for* $\boldsymbol{x}$. *Then* MELC *and* CMELC *are computed according to the following equations:*

$$\texttt{CMELC}[i] = \texttt{MaxCount}(\texttt{CMEC[1..i]}), \tag{3.8}$$

$$\texttt{MELC}[i] = \texttt{CorLen}(\texttt{CMEC[1..i]}). \tag{3.9}$$

To see that Proposition 11 is correct, we only need to recall that now the cover need not be a border and hence its last occurrence may end before the (prefix of the) string under consideration. Clearly this extra work does not change the asymptotic behaviour of the algorithm: Theorems 6 and 10 hold also for the computation of `MELC`.

3.4.2 Minimum Enhanced Left Seed Array (MELS)

The computation of the `MELS` array is more complicated. The complication arises because now not only are we looking at the prefix (as opposed to a border) but also considering a superstring to cover rather than the original string. To comprehend the new setting let us recall how Compute_MEC actually works (see Figure 3.3). The heart of the computation is the **while** loop where we fix on a prefix and continue to work on the shorter prefixes that can cover that prefix. Note that during this **while** loop we remain on a particular index i and we only update index $i+q'-1$ of the `MEC` and `CMEC` arrays, where q' is the length of the prefix we are considering. This works fine when the prefix under consideration also has to be a border, because then we know that it must occur at the end aligning with the end of the prefix of the string under consideration. But for a left seed, more work is required. Now we are interested in the interval $i..i+q'-1$. Clearly, for the index $i+q'-1$ — that is, for the string $\boldsymbol{x}[1..i+q'-1]$ — the occurrence of the prefix under consideration — that is, the prefix of length q' — is also a border. But for $\boldsymbol{x}[1..i+\ell-1]$ with $q' < \ell \leq B$, $\boldsymbol{x}[1..i+l-1]$ is a superstring and the prefix of length ℓ is a left seed. Therefore, we need to update `MELS` and `CMELS` for $i+q'-1$ based on which prefix covers most, whereas in Compute_MEC we only need to update the index $i+q'-1$ with only one

```
procedure Compute_MELS(π; MELS, CMELS)
n ← |π|
Compute_MNC(n, π; B, γ, MNC)
MEC ← 0^n; CMEC ← 0^n; PR ← 1^B
for q ← 1 to B do CPR[q] ← q
for i ← 2 to n do
    q ← π[i]
▷ x[i..i+q−1] = x[1..q].
    while q > 0 do
▷ x[1..q'] is the longest prefix of x[1..q] without a cover.
        q' ← MNC[q]
▷ x[1..q'] also occurs at i: update CPR[q'] & PR[q'].
        if q' = q then
            if i − PR[q'] < q' then
                CPR[q'] ← CPR[q'] + i − PR[q']
            else
                CPR[q'] ← CPR[q'] + q'
        else
            q' ← q
        mpf ← B
▷ New inner while loop starts here
        while mpf > q' do
            mp ← MNC[mpf]
            if mp = q' then
                break
            if PRS[mp] > 1 && i + q' > PRS[mp] + mp then
                if i − PR[mp] < mp then
                    S_1 ← CPR[mp] − mp + i − PR[mp] + q'
                else
                    S_1 ← CPR[mp]+q'
            if S_1 ≥ S_p then
                maxp ← mp
                S_p ← S_1
            mpf ← mp − 1
▷ Update S_2 and q'' depending on Maximum of S_p and CPR[q']
        S_2 ← S_p or CPR[q']
        q'' ← maxp or q'
▷ Update CMELS & MELS accordingly.
        if S_2 ≥ CMELS[i+q'−1] then
            MELS[i+q'−1] ← q''
            if S_2 > CMELS[i+q'−1] then
                CMELS[i+q'−1] ← S_2
        if PR[q'] = 1 then
            PRS[q'] ← i
        PR[q'] ← i
        q ← q−1
```

Figure 3.4: Computing MELS amd CMELS from the prefix array $\boldsymbol{\pi}$.

prefix.

The correctness of Compute_MELS readily follows from the above discussion. However, the running time increases due to the newly introduced inner **while** loop at line 17 within the outer **while** loop. Now if we recall the time complexity analysis of Compute_MEC, we realize that the only change between the two algorithms is that in the former for each prefix considered we need to do the update on only one index (that is, index $i + q' + 1$) with one prefix q', whereas in the latter, we need to update one index while considering all the prefixes larger than q'. Notice that each prefix with length ℓ for a particular substing $\boldsymbol{x}[1..i + l - 1]$ is considered ℓ times since the substring can be a superstring of $\ell - 1$ substrings of $\boldsymbol{x}$. There are $O(\log n)$ of weakly periodic prefixes of a string each of which can at most be equal to B. Therefore, a straightforward analysis gives us a running time of $O(nB \log n)$ for Compute_MELS. Recall from Section 3.3.1, that for a given alphabet size, B approaches a limit as n goes to infinity and monotone decreasing in alphabet size. This ensures that the running time remains linear in the average case:

Theorem 12 *In the average case, Compute_MELS requires $\mathcal{O}(n)$ time and $\Theta(\log n)$ additional bits of space.*

However, a more careful analysis of the worst case running time of Compute_MELS can be performed as follows. We have already used the fact from [45] that there are at most $\log_2 n$ weakly periodic borders of a string of length n (Lemma 5). However, this is a consequence of the fact that a weakly periodic border of a string of length n can be of size at most $n/2$. Now recall that we are only handling weakly periodic prefixes in Compute_MELS in the inner **while** loop. And the number of superstrings involved with each weekly periodic prefix is the length of the current prefix under

consideration. Summing the lengths of these prefixes yields a geometric series up to $\log n$ that adds up to $O(n)$. This implies that the total work done by the **for** loop within the **while** loop during the complete execution of the algorithm remains $O(n)$. Hence:

Theorem 13 *In the worst case, Compute_MELS computes* MELS *and* CMELS *from* $\boldsymbol{\pi}$ *using* $\mathcal{O}(n^2)$ *time and* $\Theta(\mathtt{B}\log\mathtt{n})$ *bits of space.*

3.5 Comparing Border-Based and Prefix-Based Algorithms

As mentioned above, in order to compute $\mathtt{MEC}_{\boldsymbol{x}}$, the authors of [45] made use of the ***border array***. On the other hand Compute_MEC is based on the ***prefix table***. As we have seen, Compute_MEC requires only three additional arrays $1..B$ of integers, compared to $4n$ for the algorithm of [45]. In the next section we will see how use of the prefix table enables Compute_MEC to be extended to indeterminate strings, not a possibility for a border-based algorithm. Here we compare the time requirements of the two algorthms, referring to our algorithm as ECP and to the border-based algorithm as ECB.

We implemented ECP in C# using Visual Studio 2010. We got the implementation of ECB from the authors of [45]. However, ECB was implemented in C. To ensure a level playing ground, we re-implemented ECB in C# following their implementation. Then we ran both the algorithms on all binary strings of lengths 2 to 30. The experiments were carried out on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed

memory (RAM) of 8.00 GB. The results are illustrated in Figures 3.5 and 3.6.

Figure 3.5 shows the maximum number of operations (assignment, comparison, etc.) carried out by each algorithm. Figure 3.6 shows the ratio of the total number of operations performed by ECB and ECP to the length n of the string, over all strings on the binary alphabet. As is evident from the figures, ECP outperforms ECB and in fact it shows linear behaviour, verifying the claim in Theorem 10 above.

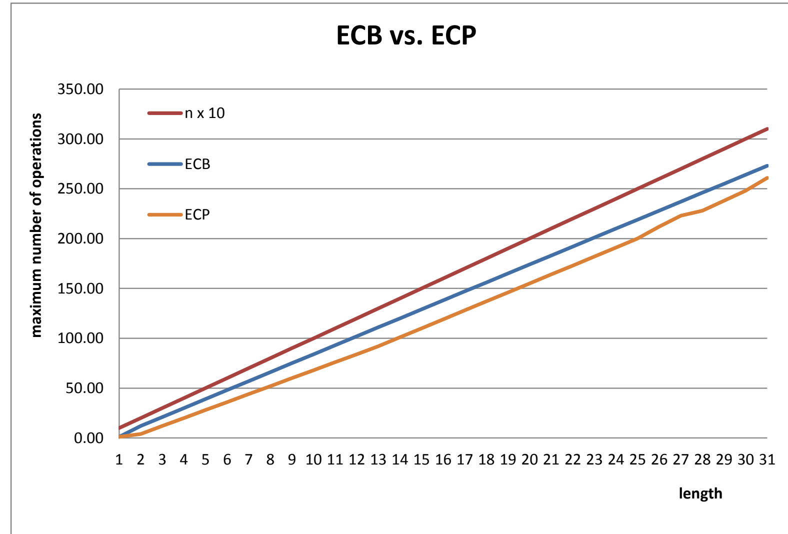

Figure 3.5: The maximum number of operations performed by the Border-Based (ECB) [45] and Prefix-Based (ECP) algorithm (i.e., Compute_MEC) to compute the Minimum Enhanced Cover array, for all strings on the binary alphabet.

3.6 Indeterminate Strings

In Sections 3.2 and 3.3 we describe an algorithm to compute the minimum enhanced cover array $\texttt{MEC}_{\boldsymbol{x}}$ of a given string $\boldsymbol{x}$, based only on the prefix array $\boldsymbol{\pi}_{\boldsymbol{x}}$. Then we have extended this algorithm for minimum enhanced left cover array and minimum enhanced left seed array in Section 3.4. As noted in the Introduction, since the prefix

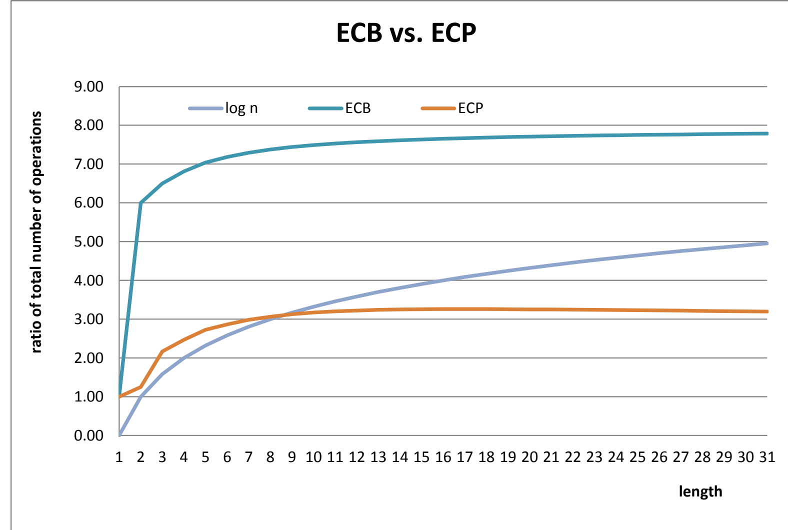

Figure 3.6: Ratio of the ***total*** number of operations performed by the Border-Based (ECB) [45] and Prefix-Based (ECP) algorithms to the length n of the string, for all strings on the binary alphabet. Note the linear behaviour of ECP compared to the slightly supralinear behaviour of ECB.

array can be computed also for indeterminate strings [82], this immediately raises the possibility of extending the MEC calculation to indeterminate strings.

The nontransitivity of matching inhibits implementation of a sliding cover, but [8] shows how to compute all the rooted covers of indeterminate $\boldsymbol{x}$ from its prefix array in $\mathcal{O}(n^2)$ worst case time, $\Theta(n)$ in the average case. Thus it becomes possible to execute Compute_MNC for rooted covers, simply by replacing the function call to Compute_CPR by a function call to PCInd of [8]; that is, to compute the rooted cover array $\boldsymbol{\gamma_R}[1..\mathtt{B}]$, hence $\mathtt{MNC}[1..\mathtt{B}]$ and thus $\mathtt{MEC}_{\boldsymbol{x}}$, all for indeterminate strings. Let us call this new algorithm Compute_MEC_Ind. We recall now a lemma from [16] stating that the expected number of borders in an indeterminate string is bounded above by a constant, approximately 29. Therefore, also for indeterminate strings, B can be treated as a constant, and we have the following remarkable result:

Theorem 14 *In the average case, Compute_MEC_Ind requires $\mathcal{O}(n)$ time and $\Theta(\log n)$ additional bits of space.*

Clearly, these results can be similarly extended for the `MELC` and `MELS` arrays to indeterminate strings.

3.7 Future Research

In this chapter we have described prefix array based algorithms to compute minimum enhanced cover arrays, minimum enhanced left cover arrays and minimum enhanced left seed arrays. The advantages of our prefix array based algorithms are threefold. Firstly, our prefix-array based algorithms exhibit the same worst case running time as the border-based algorithms of [45] but are shown to be faster in practice. Secondly, our algorithms exhibit superior space efficiency. And finally, because of the robustness of the prefix array, our algorithms, in addition to being faster in practice and more space-efficient than those of [45], allow us to easily extend the computation of enhanced covers to indeterminate strings. Additionally, both for regular and indeterminate strings, our algorithms execute in expected linear time. We have also established an important theoretical result which we believe is of independent interest: that the expected maximum length of any border of any prefix of a regular string $\boldsymbol{x}$ is approximately 1.64 for binary alphabets, less for larger ones.

As a future work it is worthwhile to design POS/LEN (compressed prefix array) version of Compute_MEC. Another natural question of course is to investigate whether the MEC array can be computed in linear time.

Chapter 4

Construction of Lyndon Array

Most of the contents of this chapter come from [47].

4.1 Introduction

Recall that, from Chapter 2 that the Lyndon array $\boldsymbol{\lambda} = \boldsymbol{\lambda}_{\boldsymbol{x}}[1..n]$ (equivalently, $\mathcal{L} = \mathcal{L}_{\boldsymbol{x}}[1..n]$) of a given nonempty string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ gives at each position i the length (equivalently, the end position) of the longest Lyndon word starting at i. It has recently become of interest since Bannai *et al.* [15] showed that it could be used to efficiently compute all the maximal periodicities ("runs") in a string. In this chapter we describe five algorithms to compute $\boldsymbol{\lambda}_{\boldsymbol{x}}$, three of them shown experimentally to be running in $\Theta(n)$ time in practice. Section 4.2 makes various observations that apply generally to the Lyndon array and its computation. In Section 4.3 we describe three algorithms, two that require $\mathcal{O}(n^2)$ time in the worst case, of which one is very fast and apparently linear in practice, the other supralinear in practice and $\mathcal{O}(n \log n)$ in the average case on binary strings. The third algorithm is simple and worst-case

linear-time, but requires suffix array construction and so is a little slower. Section 4.5 describes two variants of a new algorithm that uses only elementary data structures (no suffix arrays). One variant is $\mathcal{O}(n^2)$ in the worst case, the other guarantees $\mathcal{O}(n \log n)$ time, but with no clear advantage in processing time. Section 4.6 describes the results of preliminary experiments on the algorithms; Section 4.7 outlines future work.

4.2 Preliminaries

We begin the section with a "Monge property". The vectors $(i, \mathcal{L}[i])$ satisfy a Monge property that is exploited by Algorithm `NSV*` (Section 4.5):

Observation 1 ***[Monge Property]:*** *Suppose positions i, j in $\boldsymbol{x}[1..n]$ satisfy $1 \leq i < j \leq n$. Then either $\mathcal{L}[i] \leq j$ or $\mathcal{L}[i] \geq \mathcal{L}[j]$: the vectors $(i, \mathcal{L}[i])$ and $(j, \mathcal{L}[j])$ are nonintersecting.*

Proof. Suppose two such vectors do intersect. Then the maximum-length Lyndon words $\boldsymbol{w_1} = \boldsymbol{x}[i..\mathcal{L}[i]]$ and $\boldsymbol{w_2} = \boldsymbol{x}[j..\mathcal{L}[j]]$ have a nonempty overlap, so that we can write $\boldsymbol{w_1} = \boldsymbol{uv},\ \boldsymbol{w_2} = \boldsymbol{vv'}$ for some nonempty $\boldsymbol{v}$. But then, by well-known properties of Lyndon words, $\boldsymbol{w_1} < \boldsymbol{v} < \boldsymbol{w_2} < \boldsymbol{v'}$, implying that $\boldsymbol{w_1 v'}$ is a Lyndon word, contradicting the assumption that $\boldsymbol{w_1}$ is maximum-length. □

We have the following observation which follows from the Lemma 1 that apply to the algorithms described below.

Observation 2 *Let $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ be the Lyndon decompostion [23, 40] of $\boldsymbol{x}$, with Lyndon words $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$. Then every Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]]$ of*

length $\boldsymbol{\lambda}[i]$ *is a substring of some* $\boldsymbol{w_h},\ h \in 1..k$.

Proof. For some $h \in 1..k-1$, consider $\boldsymbol{w_h}$ with nonempty proper suffix $\boldsymbol{v_h}$, and for some $t \in 1..k-h$, consider $\boldsymbol{w_{h+t}}$ with nonempty prefix $\boldsymbol{u_{h+t}}$. Since $\boldsymbol{w_h}$ is a Lyndon word, $\boldsymbol{w_h} < \boldsymbol{v_h}$, and by lexicographic order (see Chapter 2), $\boldsymbol{u_{h+t}} \leq \boldsymbol{w_{h+t}}$. Thus $\boldsymbol{v_h} > \boldsymbol{w_h} \geq \boldsymbol{w_{h+t}} \geq \boldsymbol{u_{h+t}}$, and so $\boldsymbol{v_h}\boldsymbol{w_{h+1}}\cdots\boldsymbol{w_{h+t-1}}\boldsymbol{u_{h+t}}$ cannot be a Lyndon word for any choice of h or t. □

Therefore to compute $\mathcal{L}_{\boldsymbol{x}}$ it suffices to consider separately each distinct element $\boldsymbol{w_h}$ in the Lyndon decomposition of $\boldsymbol{x}$. Hence, without loss of generality suppose $\boldsymbol{x}$ is a Lyndon word and write it in the form $\boldsymbol{x_1}\boldsymbol{x_2}\cdots\boldsymbol{x_m}$, where for each $r \in 1..m$, $|\boldsymbol{x_r}| = \ell_r$ and

$$\boldsymbol{x_r}[1] \leq \boldsymbol{x_r}[2] \leq \cdots \leq \boldsymbol{x_r}[\ell_r], \tag{4.1}$$

while for $1 \leq r < m$,

$$\boldsymbol{x_r}[\ell_r] > \boldsymbol{x_{r+1}}[1]. \tag{4.2}$$

We call $\boldsymbol{x_r}$ a ***range*** in $\boldsymbol{x}$ and the boundary between $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$ a ***drop***. We identify a position j in range $\boldsymbol{x_r}$, $1 \leq j \leq \ell_r$, with its equivalent position i in $\boldsymbol{x}$ by writing $i = S_{r,j} = \sum_{r'=1}^{r-1} \ell_{r'} + j$.

Observation 3 *Let* $i = S_{r,j}$ *be a position in* $\boldsymbol{x}$ *that corresponds to position* j *in range* $\boldsymbol{x_r}$.

(*a*) *If* $\boldsymbol{x_r}[j] = \boldsymbol{x_r}[\ell_r]$, *then* $\mathcal{L}[i] = i$.

(*b*) *Otherwise,* $\mathcal{L}[i] = i'$, *where* i' *is the final position in some range* $\boldsymbol{x_{r'}},\ r' \geq r$; *that is,* $i' = \sum_{s=1}^{r'} \ell_s$.

Proof. (a) is an immediate consequence of (4.1) and (4.2). To prove (b), suppose that $\boldsymbol{x}[i..\mathcal{L}[i]]$ is a maximum-length Lyndon word, where $\mathcal{L}[i]$ falls within range r' but $\mathcal{L}[i] < i'$. Since by (4.1) $\boldsymbol{x}[\mathcal{L}(i)] \leq \boldsymbol{x}[\mathcal{L}[i]{+}1]$, there are two consecutive Lyndon words $\boldsymbol{x}[i..\mathcal{L}[i]], \boldsymbol{x}[\mathcal{L}[i]+1]$ that by the Lyndon decomposition theorem [23] can be merged into a single Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]+1]$. Thus $\boldsymbol{x}[i..\mathcal{L}[i]]$ is not maximum-length, a contradiction. □

We see then that if $\boldsymbol{x_r}[j] < \boldsymbol{x_r}[\ell_r]$, then $\boldsymbol{x_r}[j..\ell_r]$ is a (not necessarily maximum-length) Lyndon word, and for $i = S_{r,j}$, $\mathcal{L}[i] \geq S_{r,\ell_r}$:

$$\begin{array}{rccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \boldsymbol{x} = & a & a & a & b| & a & a & b| & a & b| & a & a & b & b \\ \mathcal{L} = & 13 & 13 & 4 & 4 & 9 & 7 & 7 & 9 & 9 & 13 & 13 & 12 & 13 \end{array} \tag{4.3}$$

Expressing a string in terms of its ranges has the same useful lexicographic order property that writing it in terms of its letters does:

Observation 4 *Suppose strings* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *are expressed in terms of their ranges:* $\boldsymbol{x} = \boldsymbol{x_1 x_2} \cdots \boldsymbol{x_m}$, $\boldsymbol{y} = \boldsymbol{y_1 y_2} \cdots \boldsymbol{y_n}$. *Suppose further that for some least integer* $r \in 1..\min(m,n)$, $\boldsymbol{x_r} \neq \boldsymbol{y_r}$. *Then* $\boldsymbol{x} < \boldsymbol{y}$ *(respectively,* $\boldsymbol{x} > \boldsymbol{y}$*) according as* $\boldsymbol{x_r} < \boldsymbol{y_r}$ *(respectively,* $\boldsymbol{x_r} > \boldsymbol{y_r}$*).*

Proof. If $\boldsymbol{x_r} < \boldsymbol{y_r}$, then either

(a) $\boldsymbol{x_r}$ is a nonempty proper prefix of $\boldsymbol{y_r}$; or

(b) there is some least position j such that $\boldsymbol{x_r}[j] < \boldsymbol{y_r}[j]$.

In case (a), if $r = m$, then $\boldsymbol{x}$ is actually a prefix of $\boldsymbol{y}$, so that $\boldsymbol{x} < \boldsymbol{y}$, while if $r < m$, then by (4.2), $\boldsymbol{x_{r+1}}[1] < \boldsymbol{y_r}[|\boldsymbol{x_r}|+1]$, and again $\boldsymbol{x} < \boldsymbol{y}$. In case (b) the result is immediate. The proof for $\boldsymbol{x_r} > \boldsymbol{y_r}$ is similar. □

4.3 Basic Algorithms

Here we outline three algorithms for which no clear exposition was available prior to [47]. We remark that the Lyndon array computation is equivalent to "Lyndon bracketing", for which an $\mathcal{O}(n^2)$ algorithm has been described [79].

4.3.1 Folklore — Iterated MaxLyn

For a string $\boldsymbol{x}$ of length n, recall that the ***prefix table*** $\pi[1..n]$ is an integer array in which for every $i \in 1\,..\,n$, $\pi[i]$ is the length of the longest substring beginning at position i of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. Given a nonempty string $\boldsymbol{x}$ on alphabet Σ, let us define $\boldsymbol{x'} = \boldsymbol{x}\$$, where the sentinel $\$ < \mu$ for every letter $\mu \in \Sigma$.

Observation 5 *$\boldsymbol{x}$ is a Lyndon word if and only if for every $i \in 2\,..\,n$, $\boldsymbol{x'}[1+k] < \boldsymbol{x'}[i+k]$, where $k = \pi[i]$.*

This result forms the basis of the algorithm given in Figure 4.1 that computes the length max $\in 1\,..\,n-j+1$ of the longest Lyndon factor at a given position j in $\boldsymbol{x}[1..n]$. The algorithm has long been existed in the literature but never properly used the word "Lyndon" (hence the term "Folklore" is used). Its efficiency is a consequence of the instruction $i \leftarrow i+k+1$ that skips over positions in the range $i+1\,..\,i+k-1$,

effectively assuming that for every position i^* in that range, $i^* + \pi[i^*] \leq i+k$. The following result justifies the strategy employed in Algorithm MaxLyn (Figure 4.1):

Lemma 6 *Suppose that for some position i in a Lyndon word $\boldsymbol{x}[1..n]$, $k = \pi[i] \geq 2$. Then for every $j \in i+1\,..\,i+k-1$, $\pi[j] \leq i+k-j$.*

Proof. The result certainly holds for $i+k = n+1$, so we consider $i+k \leq n$. Assume that for some $j \in i+1\,..\,i+k-1$, $\pi[j] > i+k-j$. It follows that

$$\boldsymbol{x}[1\,..\,i+k-j+1] = \boldsymbol{x}[j\,..\,i+k], \tag{4.4}$$

while $\boldsymbol{x}[j-i+1\,..\,k] = \boldsymbol{x}[j\,..\,i+k-1]$. Since $\boldsymbol{x}$ is Lyndon, therefore $\boldsymbol{x}[1+k] < \boldsymbol{x}[i+k]$, and so we find that

$$\boldsymbol{x}[j-i+1\,..\,1+k] < \boldsymbol{x}[j\,..\,i+k]. \tag{4.5}$$

From (4.4) and (4.5) we see that $\boldsymbol{x}[1..k+1]$ has suffix $\boldsymbol{x}[j-i+1..k+1]$ satisfying $\boldsymbol{x}[j-i+1..k+1] < \boldsymbol{x}[1..i+k-j+1]$, contradicting the assumption that $\boldsymbol{x}$ is Lyndon.

□

Simply repeating MaxLyn at every position j of $\boldsymbol{x}$ gives a simple, fast $\mathcal{O}(n^2)$ time and $\mathcal{O}(1)$ additional space algorithm to compute $\boldsymbol{\lambda_x}$.

Recent work on the prefix table [19, 24] has confirmed its importance as a data structure for string algorithms. In this context it is interesting to find that Lyndon words $\boldsymbol{x}$ can be characterized in terms of $\pi_{\boldsymbol{x}}$:

Observation 7 *Suppose $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ is a string on alphabet Σ such that $\boldsymbol{x}[1]$ is the least letter in $\boldsymbol{x}$. Then $\boldsymbol{x}$ is a Lyndon word over Σ if and only if for every $i \in 2\,..\,n$,*

```
procedure MaxLyn(x[1..n], j, Σ) : integer
i ← j + 1; max ← 1
while i ≤ n do
   k ← 0
   while x′[j + k] = x′[i + k] do
      k ← k + 1
   if x′[j + k] < x′[i + k] then
      i ← i + k + 1; max ← i − 1
   else
      return max
```

Figure 4.1: Algorithm MaxLyn

(*a*) $i + \pi_{\boldsymbol{x}}[i] < n + 1$; *and*

(*b*) *for every* $j \in i+1 \,..\, i+\pi_{\boldsymbol{x}}[i]-1$, $j + \pi_{\boldsymbol{x}}[j] \leq i + \pi_{\boldsymbol{x}}[i]$.

4.3.2 Recursive Duval Factorization: Algorithm RDuval

Rather than independently computing the maximum-length Lyndon factor at each position i, as MaxLyn does, Algorithm RDuval recursively computes the Lyndon decomposition into maximum factors, at each step taking advantage of the fact that $\mathcal{L}[i]$ is known for the first position i in each factor, then recomputing with the first letters removed. By Observation 2, whenever $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a Lyndon word, we know that $\mathcal{L}[1] = n$. Thus computing the Lyndon decomposition $\boldsymbol{x} = \boldsymbol{w_1 w_2} \cdots \boldsymbol{w_k}$, $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$, allows us to assign $\boldsymbol{\lambda}[i_j] = |\boldsymbol{w_j}|$, where i_j is the first position of $\boldsymbol{w_j}$, $j = 1, 2, \ldots, k$.

Algorithm RDuval applies this strategy recursively, by assigning $\boldsymbol{\lambda}[i_j] \leftarrow |\boldsymbol{w_j}|$, then removing the first letter i_j from each $\boldsymbol{w_j}$ to form $\boldsymbol{w'_j}$, to which the Lyndon decomposition is applied in the next recursive step. This process continues until each Lyndon word is reduced to a single letter.

The asymptotic time required for RDuval is bounded above by n times the maximum depth of the recursion, thus $O(n^2)$ in the worst case — consider, for example, the string $\boldsymbol{x} = a^{n-1}b$. However, to estimate expected behaviour, we can make use of a result of Bassino *et al.* [17]. Given a Lyndon word $\boldsymbol{w}$, they call $\boldsymbol{w} = \boldsymbol{uv}$ the ***standard factorization*** of $\boldsymbol{w}$ if $\boldsymbol{u}$ and $\boldsymbol{v}$ are both Lyndon words and $\boldsymbol{v}$ is of maximum size. They then show that if $\boldsymbol{w}$ is a binary string ($\Sigma = \{a, b\}$), the average length of $\boldsymbol{v}$ is asymptotically $3|\boldsymbol{w}|/4$. Thus each recursive application of RDuval yields a left Lyndon factor of expected length $|\boldsymbol{w}|/4$ and a remainder of length $3|\boldsymbol{w}|/4$ to be factored further. It follows that the expected number of recursive calls of RDuval is $\mathcal{O}(\log_{4/3} n)$. Hence

Lemma 8 *On binary strings RDuval executes in $O(n \log_{4/3} n)$ time on average.*

Example 9 *For*

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

the factors considered are first 1–12, then

- *2–3 and 4–12 in the first level of recursion;*
- *3, 5–7, 8–10 and 11–12 in the second level;*
- $6, 7, 9, 10, 12$ *in the third level.*

Positions are assigned as follows: $\boldsymbol{\lambda}[1] \leftarrow 12; \boldsymbol{\lambda}[2] \leftarrow 2, \boldsymbol{\lambda}[4] \leftarrow 9; \boldsymbol{\lambda}[3] \leftarrow 1, \boldsymbol{\lambda}[5] \leftarrow 3, \boldsymbol{\lambda}[8] \leftarrow 3, \boldsymbol{\lambda}[11] \leftarrow 2; \boldsymbol{\lambda}[6] \leftarrow 1, \boldsymbol{\lambda}[7] \leftarrow 1, \boldsymbol{\lambda}[9] \leftarrow 1, \boldsymbol{\lambda}[10] \leftarrow 1, \boldsymbol{\lambda}[12] \leftarrow 1.$

4.3.3 NSV Applied to the Inverse Suffix Array

The idea of the "next smaller value" (NSV) array for a given array (string) $\boldsymbol{x}$ has been proposed in various forms and under various names [9, 43, 74, 52].

Definition 10 (Next Smaller Value) *Given an array* $\boldsymbol{x}[1..n]$ *of ordered values,* $\texttt{NSV} = \texttt{NSV}_{\boldsymbol{x}}[1..n]$ *is the **next smaller value array** of* $\boldsymbol{x}$ *if and only if for every* $i \in 1..n$, $\texttt{NSV}[i] = j$, *where*

(*a*) *for every* $h \in 1..j-1$, $\boldsymbol{x}[i] \leq \boldsymbol{x}[i+h]$*; and*

(*b*) *either* $i+j = n+1$ *or* $\boldsymbol{x}[i] > \boldsymbol{x}[i+j]$.

Example 11

	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{x} =$	3	8	7	10	2	1	4	9	6	5
$\texttt{NSV}_{\boldsymbol{x}} =$	4	1	2	1	1	5	4	1	1	1

As shown in various contexts in [52], $\texttt{NSV}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ time using a stack. Our main observation here, touched upon in [54], is that $\boldsymbol{\lambda}_{\boldsymbol{x}}$ can be computed merely by applying NSV to the inverse suffix array $\texttt{ISA}_{\boldsymbol{x}}$. Here we present the very simple $\Theta(n)$-time, $\Theta(n)$-space algorithm for this calculation:

procedure NSVISA($\boldsymbol{x}[1..n]$) : $\boldsymbol{\lambda}_{\boldsymbol{x}}[1..n]$
Compute $SA_{\boldsymbol{x}}$ (see [73, 77])
Compute $\texttt{ISA}_{\boldsymbol{x}}$ from $\texttt{SA}_{\boldsymbol{x}}$ in place (see [77])
$\boldsymbol{\lambda}_{\boldsymbol{x}} \leftarrow$ NSV($\texttt{ISA}_{\boldsymbol{x}}$) (in place)

Figure 4.2: Apply NSV to $\texttt{ISA}_{\boldsymbol{x}}$

The following theorem as well as the proof is due to Jamie Simpson based on discussion with W. F. Smyth, that justifies Algorithm 4.2:

Theorem 12 *For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on totally ordered alphabet Σ, $\lambda_{\boldsymbol{x}} = \mathtt{NSV}_{\mathtt{ISA}_{\boldsymbol{x}}}$.*

Proof. Consider the longest Lyndon word beginning at $\boldsymbol{x}[i]$ which is $\boldsymbol{x}[i..i+\lambda[i]-1]$. Since it is Lyndon we have

$$\boldsymbol{x}[i..i+\lambda[i]-1] < \boldsymbol{x}[j..i+\lambda[i]-1] \tag{4.6}$$

for each $j \in [i+1..i+\lambda[i]-1]$ and therefore

$$\boldsymbol{x}[i..n] < \boldsymbol{x}[j..n]$$

for $j \in [i+1..i+\lambda[i]-1]$. This means that for $j \in [i+1..i+\lambda[i]-1]$, $\mathtt{ISA}[\mathtt{j}] > \mathtt{ISA}[\mathtt{i}]$, hence if there exists a value of $\mathtt{ISA}[\mathtt{j}]$ smaller than $\mathtt{ISA}[\mathtt{i}]$ with $j > i$ then it must occur with $j \geq i+\lambda[i]$, so $\mathtt{NSV}[\mathtt{i}] \geq \lambda[\mathtt{i}]$. Suppose we have strict inequality here which means $\mathtt{ISA}[\mathtt{i}+\lambda[\mathtt{i}]] > \mathtt{ISA}[\mathtt{i}]$. Which leads to the following

$$\boldsymbol{x}[i+\lambda[i]..n] > \boldsymbol{x}[i..n]$$

and

$$\boldsymbol{x}[i+\lambda[i]..i+\lambda[i]] > \boldsymbol{x}[i..i+\lambda[i]]$$

that means Equation 4.6 holds for $j \in [i+1..i+\lambda[i]]$. This means that $\boldsymbol{x}[i..i+\lambda[i]]$ is a Lyndon word, which is a contradiction. We conclude that $\mathtt{NSV}[\mathtt{i}] \geq \lambda[\mathtt{i}]$ for $i = 1, ..., n$ and we are done.

If there is no value $\mathtt{ISA}[\mathtt{j}]$ smaller than $\mathtt{ISA}[\mathtt{i}]$ with $j > i$ then $\mathtt{NSV}[\mathtt{i}] = \mathtt{n}+\mathtt{1}-\mathtt{i}$. Therefore $\boldsymbol{x}[j..n] > \boldsymbol{x}[i..n]$ for all $j \in [i+1..n]$, hence $\boldsymbol{x}[i..n]$ is Lyndon and $\lambda[i] =$

$n+1-i = \texttt{NSV[i]}$. □

4.4 Elementary Computation of $\boldsymbol{\lambda_x}$ by Comparing Ranges

Consider ranges $\boldsymbol{x_r}$ and $\boldsymbol{x_{r'}}$ in $\boldsymbol{x}$, $1 \le r < r' \le m$, and let $f_{r'}$ be the frequency of $\boldsymbol{x_{r'}}[1]$; that is, the largest integer such that $\boldsymbol{x_{r'}}[f_{r'}] = \boldsymbol{x_{r'}}[1]$. Let c be the largest integer such that $\boldsymbol{x_r}[c..c+f_{r'}-1] = \boldsymbol{x_{r'}}[1..f_{r'}]$, or $c = 0$ if there is no such integer. We call $c = c(r,r')$ the ***critical point*** of range $\boldsymbol{x_r}$ with respect to $\boldsymbol{x_{r'}}$ — that is, of ***range match*** $(r : r')$.

Observation 13 *For every* $j \in 1..c-1$, $\boldsymbol{x_r}[j..\ell_r] < \boldsymbol{x_{r'}}$; *for every* $j \in c+1..\ell_r$, $\boldsymbol{x_r}[j..\ell_r] > \boldsymbol{x_{r'}}$.

Thus only at a critical point $j = c(r,r')$ is it necessary to do pattern-matching in order to determine whether $\boldsymbol{x_r}[j..\ell_r] >,=,< \boldsymbol{x_{r'}}$, $r' > r$. This processing requires at most $\min(\ell_r, \ell_{r'}) \le \ell_r$ letter comparisons. Since for each range $\boldsymbol{x_{r'}}$, $f_{r'}$ can be determined by linear-time preprocessing, we have

Observation 14 *For every choice of* r *and* r' *satisfying* $1 \le r < r' \le m$, *and over all* $j \in 1..\ell_r$, *the* ℓ_r *comparisons*

$$\boldsymbol{x_r}[j..\ell_r] \; : \; \boldsymbol{x_{r'}}$$

can be performed in $\mathcal{O}(\ell_r)$ *total time.*

We now describe a simple algorithm that makes use of the ranges to compute the Lyndon array $\mathcal{L}_{\boldsymbol{x}}$ of a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. The outline is given in Figure 4.3.

```
procedure RangeLyndon(x, n, m)
for r ← m downto 1 do
▷ RHS computes L[i] for the righthand positions j in x_r
▷ such that x_r[j] > x_{r+1}[1]; it returns the rightmost position j_R,
▷ if any, such that x_r[j_R] ≤ x_{r+1}[1]; otherwise, zero.
  j_L ← j_R ← RHS(r); r' ← r+1
▷ The main loop for x_r computes the critical points j_L = c(r, r') for
▷ successive values of r' until L_{r,j} is computed for every j. FIND
▷ and COMP identify subranges x_r[j_L..j_R] of equal letters that
▷ match a maximum-length prefix of x_{r'}; only for substring pairs
▷ beginning at x_r[j_R+1] and x_{r'}[f_{r'}+1] is pattern-matching required.
  while j_L > 0 do
    (j_L, j_R, r') ← FIND(r, j_L, j_R, r')
    if j_L > 0 then (j_L, j_R, r') ← COMP(r, j_L, j_R, r')
```

Figure 4.3: Algorithm RangeLyndon computes $\mathcal{L}_{\boldsymbol{x}}$ of $\boldsymbol{x} = \boldsymbol{x_1 x_2 \cdots x_m}$. Only at critical points $j_L = c(r, r')$ does it need to do pattern-matching; otherwise all $\mathcal{L}_{r,j}$ are computed in constant time.

```
procedure RHS(r)
j ← ℓ_r
while j > 0 and x_r[j] = x_r[ℓ_r] do
  L_{r,j} ← S_{r,j}; j ← j−1
while j > 0 and x_r[j] > x_{r+1}[1] do
▷ Recall x_{m+1} = $.
  L_{r,j} ← S_{r,ℓ_r}; j ← j−1
return j
```

Figure 4.4: Compute $\mathcal{L}$ for all positions j such that $\boldsymbol{x_r}[j] > \boldsymbol{x_{r+1}}[1]$.

To simplify description of the algorithm, we suppose that the given string $\boldsymbol{x}$ is bracketed by the sentinel \$, a letter less than any other letter in the alphabet. Thus, for *abbacd*, we suppose $\boldsymbol{x} = \$abbacd\$$. Further, for accesses $\boldsymbol{x_r}[0]$, we suppose that for $r > 1$, $\boldsymbol{x_r}[0] = \boldsymbol{x_{r-1}}[\ell_{r-1}]$, while $\boldsymbol{x_1}[0] = \$$.

```
procedure FIND(r, j_L, j_R, r')
 EXIT ← FALSE
while j_L > 0 and not EXIT do
▷ Count matches with x_{r'}[1]; initially x_r[j] = x_{r'}[1],  j_L ≤ j ≤ j_R.
  count ← j_R − j_L + 1
  ▷ Skip segments starting at r' greater than segments starting at r.
  while (count > f_{r'} and x_r[j_L] = x_{r'}[1]) or x_r[j_L] < x_{r'}[1] do
    i' ← L_{r',1}+1;  r' ← S_{i'}^{-1}
  if r' = m+1 then
    ▷ Special case: end of input string.
    while j_L > 0 do L_{r,j_L} ← S_{m,ℓ_m};  j_L ← j_L−1
    return 0, 0, r'
  ▷ Identify the "critical range" j_L..j_R.
  while (j_L > 1 and count < f_{r'} and x_r[j_L−1] = x_{r'}[1])
  or x_r[j_L] > x_{r'}[1] do
    ▷ Count the number of letters which represent x_r[j_L]:
    ▷ if the letter changes, adjust j_R and set count ← 1.
    count ← count+1;  L_{r,j_L} ← S_{r',1}−1;  j_L ← j_L−1
    if x_r[j_L] < x_r[j_L+1] then j_R ← j_L;  count ← 1
  if count = f_{r'} then EXIT ← TRUE          x_r[j_L..j_R] = x_{r'}[1..f_{r'}].
  elsif x_r[j_L] = x_{r'}[1] then
    ▷ x_r[j_L..j_R] > x_{r'}[1..f_{r'}]: position j_L not critical for this r'.
    L_{r,j_L} ← S_{r',1}−1;  j_L ← j_L − 1;  j_R ← j_L
return j_L, j_R, r'
```

Figure 4.5: Find the next critical point $j_L = c(r, r')$ to the left of the current position j_R in $\boldsymbol{x_r}$, updating r' as required.

Algorithm FIND computes $\mathcal{L}_{r,j}$ in constant time per position j that is greater than the current critical point $j_L = c(r, r')$ — initially $r' = r+1$. When position $j = j_L$ is critical, COMP performs pattern-matching to determine the relationship between $\boldsymbol{x_r}[j_L..\ell_r]$ and $\boldsymbol{x_{r'}}$ — or, more precisely, between substrings beginning $\boldsymbol{x_r}[j_R+1..\ell_r]$ and $\boldsymbol{x_{r'}}[f_{r'}+1..\ell_{r'}]$, respectively. Then, for j such that $\boldsymbol{x_r}[j..\ell_r] < \boldsymbol{x_{r'}}$, as must be true for every $j \in 1..c(r, r')-1$, FIND and COMP may be called again to continue checking against the appropriate $\boldsymbol{x_{r'}}$, $r' > r+1$.

```
procedure COMP(r, j_L, j_R, r')
EXIT ← FALSE
repeat
  ▷ MATCH returns δ = −1, 0, +1 according as
  ▷ (for r' = r + 1) x_r[j_R + 1] ... x_{r'−1}[ℓ_{r'−1}] <, =, > x_{r'}[f_{r'} + 1] ...;
  ▷ (else) x_r[j_L] ... x_{r'−1}[ℓ_{r'−1}] <, =, > x_{r'}[1] ... — taking account
  ▷ of range boundaries and end condition x_{m+1} = $.
  δ ← MATCH(r, j_L, j_R, r')
  if δ = 1 then
    if r' = r+1 then L_{r,j_L} ← S_{r,ℓ_r} else L_{r,j_L} ← L_{r'−1,1}
    j_L ← j_L−1
    if x_r[j_L] < x_r[j_L + 1] then j_R ← j_L
    return j_L, j_R, r'
  elsif (δ = 0 and (L_{r',1} − S_{r',1} + 1) = (S_{r',1} − S_{r,j_L})) then
    L_{r,j_L} ← S_{r',1} − 1; EXIT ← TRUE
  else
    L_{r,j_L} ← L_{r',1}
  ▷ Recompute r' for next match.
  i' ← L_{r',1}+1; r' ← S^{−1}_{i'}
until EXIT
j_L ← j_L−1
if x_r[j_L] < x_r[j_L + 1] then j_R ← j_L
return j_L, j_R, r'
```

Figure 4.6: Compute $\mathcal{L}_{r,j}$ for all positions $j \in 1..c(r, r')$, taking account of the Monge property that arcs $(\boldsymbol{x_r}[j], \mathcal{L}_{r,j})$ cannot intersect (Observation 1).

Thus RangeLyndon performs pattern-matching between ranges $\boldsymbol{x_r}$ and $\boldsymbol{x_{r'}}$ only from the critical point $j_L = c(r, r')$ (if it exists). Note that if the substring starting at $\boldsymbol{x_r}[j]$ is less than or equal to a substring starting at $\boldsymbol{x_{r'}}$, then so *a fortiori* is the one starting at $\boldsymbol{x_r}[j-1]$ — this condition is the basis of the Monge property that arcs $(\boldsymbol{x_r}[j], \mathcal{L}_{r,j})$ are nonintersecting (Observation 1).

Observation 15 *For range matches $(r : r+1)$, $1 \leq r \leq m-1$, the maximum total number of letter matches performed by RangeLyndon is $n-\ell_m$, based on $m-1$ range matches.*

Thus if no matches $(r : r')$, $r' > r{+}1$, are performed, RangeLyndon executes in $\Theta(n)$ time. Indeed, if only range matches $(r : r{+}1)$ and $(r : r{+}2)$ occur, then RangeLyndon will have a maximum $2(n-\ell_m)-\ell_{m-1}$ letter matches, and so will again execute in linear time. We see that if there exists a positive integer t, independent of n, such that no matches $(r : r')$, $r' > r{+}t$, are performed, then in this case also RangeLyndon is linear. For range $\boldsymbol{x_r}$ in $\boldsymbol{x}$, let RM_r denote the number of right range matches $(r : r'), r' > r$, performed by RangeLyndon.

Lemma 16 *For $r \in 1..m$, the maximum value of RM_r is $\lceil \log_2(m-r+1) \rceil$.*

Proof. The maximum will be achieved if ranges are initially compared in pairs, then in pairs of pairs, and so on. Thus range $\boldsymbol{x_r}$ will be matched against ranges $\boldsymbol{x_{r+h}}$, $h = 2^j$, for every $j \geq 0$ such that $r{+}h \leq m$. Accordingly, for ranges $\boldsymbol{x_m}, \boldsymbol{x_{m-1}}, \ldots$, the number of right matches will be $0, 1, 2, 2, 3, 3, 3, 3, \ldots$; in general $\lceil \log_2(m-r+1) \rceil$, as required. □

Lemma 17 *Let $RM^*(k)$ denote the maximum number of range matches needed by Algorithm RangeLyndon to compute $\boldsymbol{\lambda_x}$, where $\boldsymbol{x}$ has length n with $m = 2^k$ ranges, $k = 0, 1, \ldots$. Then $RM^*(k) = m(\log_2 m-1)+1 \in \Theta(n \log n)$.*

Proof. From Lemma 16, we may write

$$
\begin{aligned}
RM^*(k) &= \sum_{j=1}^{k} j2^{j-1} \\
&= \sum_{j=1}^{k-1} j2^{j} + \sum_{j=0}^{k-1} 2^{j} \\
&= \big((k-1)(2^{k+1}-2^k)-2^k+2\big) + (2^k-1) \quad [61, \text{p. } 33] \\
&= (k-1)2^k+1,
\end{aligned}
$$

and so $RM^*(k) = m(\log_2 m - 1)+1$. Since every range $\boldsymbol{x_r}$ is compared with range $\boldsymbol{x_{r+2}}$, we may assume without loss of generality that range length is a constant ℓ. Thus $n = m\ell$ and $RM^*(k) \in \Theta(n \log n)$, as required. □

We may construct strings $\boldsymbol{y_k}$ whose range matches achieve the maximum of Lemma 17. For some integer $q > 0$, let $\boldsymbol{y_0} = a^q b_0$, and then for $k \geq 1$, let $\boldsymbol{y_k} = \boldsymbol{y_{k-1}} \boldsymbol{y'_{k-1}}$, where for $0 \leq h \leq k-1$, $\boldsymbol{y'_h}$ is identical to $\boldsymbol{y_h}$ except that b_{h+1} replaces b_h. We suppose $b_0 < b_1 < \cdots < b_k$. Then for $\boldsymbol{y_1} = a^q b_0 a^q b_1$ with 2^1 ranges and matches $\{1, 0\}$, and for $\boldsymbol{y_2} = a^q b_0 a^q b_1 a^q b_0 a^q b_2$ with 2^2 ranges and matches $\{2, 2, 1, 0\}$; and in general for $\boldsymbol{y_k}$ with 2^k ranges, it is easy to see that the range matches are greatest possible. Note that $|\boldsymbol{y_k}| = (q+1)2^k$. Thus:

Lemma 18 *Algorithm RangeLyndon requires $\Omega(n \log n)$ time in the worst case.*

As stated, Lemma 17 applies only to strings formed from ranges whose number is an exact power of 2. Nevertheless, in view of Lemma 16, the general result of Lemma 17, that $RM^* \in \mathcal{O}(n \log n)$, must hold also for strings formed from any

intermediate number of ranges: the detailed calcuation will merely be more complicated. Furthermore, without loss of generality, we can make two other simplifying assumptions:

- Every critical point occurs at the beginning of its range. This assumption ensures that in any range match $(r : r')$, every position in range $\boldsymbol{x_r}$ will require a letter match, thus maximizing letter matches with respect to string length.
- Every range match goes to the end of the range, again with the effect of maximizing letter matches.

In view of the fact that all adjacent ranges are compared, these two assumptions imply that the number of letter matches in each individual range match equals (is bounded by) constant range length ℓ.

4.5 Computation of $\boldsymbol{\lambda_x}$ Using Ranges and NSV Idea

In this section we describe an approach to the computation of $\boldsymbol{\lambda_x}$ that applies a variant of the NSV idea to the ranges of $\boldsymbol{x}$. Figure 4.7 gives pseudocode for Algorithm `NSV`* that uses the NSV stack ACTIVE to compute $\boldsymbol{\lambda}$. The processing identifies ranges in a single left-to-right scan of $\boldsymbol{x}$, making use of two range comparison routines, COMP and MATCH. COMP compares adjacent individual ranges $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$, returning $\delta_1 = -1, 0, +1$ according as $\boldsymbol{x_r} < \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} = \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} > \boldsymbol{x_{r+1}}$. MATCH similarly

returns δ_2 for adjacent *sequences* of ranges; that is,

$$\begin{aligned} \boldsymbol{X_r} &= \boldsymbol{x_r x_{r+1} \cdots x_{r+s}}, \text{ for some } s \geq 1; \\ \boldsymbol{X_{r+s+1}} &= \boldsymbol{x_{r+s+1} x_{r+s+2} \cdots x_{r+s+t}}, \text{ for some } t \geq 1. \end{aligned}$$

Algorithm `NSV*` is based on the idea encapsulated in the proof of Theorem 12, the main basis of the correctness of Algorithm NSVISA. We process $\boldsymbol{x}$ from left to right, using a stack ACTIVE initialized with index 1. At each iteration, the top of the stack (say, j) is compared with the current index (say, i). In particular, we need to compare $\boldsymbol{s_x}(i)$ with $\boldsymbol{s_x}(j)$, where $\boldsymbol{s_x}(i) \equiv \boldsymbol{x}[i..n]$. As long as $\boldsymbol{s_x}(i) \succeq \boldsymbol{s_x}(j)$, `NSV*` pushes the current index and continues to the next. When $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(j)$, it pops the stack and puts appropriate values in the corresponding indices of $\boldsymbol{\lambda_x}$. As noted above, especially Observations 2–1, ranges are employed to expedite these suffix comparisons.

Two auxiliary arrays, **nextequal** and **period**, are required to handle situations in which MATCH finds that a suffix of a previous range at position j equals the current range at position i. Thus, when $\delta_2 = 0$, the algorithm assigns nextequal$[j] \leftarrow i$ before i is pushed onto ACTIVE. Then when a later MATCH yields $\delta_2 = 0$, the value of **period** — that is, the extent of the following periodicity — may need to be set or adjusted, as shown in the following example:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\boldsymbol{x}$	=	a	a	a	b	a	a	b	a	a	b	a	a	b	a	b
nextequal	=	0	5	0	0	8	0	0	11	0	0	0	14	0	0	0
period	=	0	12	0	0	9	0	0	6	0	0	0	4	0	0	0

```
procedure NSV* (x, λ)
nextequal ← 0^n; period ← 0^n
push(ACTIVE) ← 1
▷ x[n+1] = $, a letter smaller than any in Σ.
for i ← 2 to n+1 do
   prev ← 0; j ← peek(ACTIVE)
▷ COMP compares suffixes specified by i, j of two ranges.
   δ1 ← COMP(x[j], x[i]); δ2 ← 1
      while (δ1 ≥ 0 and δ2 > 0) do
         if δ1 = 0 then δ2 ← MATCH(x[j], x[i])
         if δ2 > 0 then
            if prev = 0 or nextequal[j] ≠ prev then λ[j] ← i−j
            else
               λ[j] ← offset ← prev−j
               if period[prev] = 0 then
                  if λ[prev] > offset then
                     λ[j] ← λ[j]+λ[prev]
               else
                  if nextequal[j] = prev and offset ≠ λ[prev] then
                     λ[j] ← λ[j]+period[prev]
               if λ[prev] = offset then
            ▷ Current position is a part of periodic substring
                  if period[prev] = 0 then
                     period[j] ← period[prev] + 2 × offset
                  else
                     period[j] ← period[prev]+offset
            pop(ACTIVE)
            prev ← j; j ← peek(ACTIVE)
            ▷ Empty stack implies termination.
            if j = 0 then EXIT
            δ1 ← COMP(x[j], x[i])
▷ Finished processing i — it goes to stack.
   if δ2 = 0 then nextequal[j] ← i
   push(ACTIVE) ← i
```

Figure 4.7: Computing $\boldsymbol{\lambda}_x$ using modified NSV

A straightforward implementation of COMP and MATCH could require a number of letter comparisons equal to the length of the shorter of the two sequences of ranges being matched. However, by performing $\Theta(n)$-time preprocessing, we can compare two ranges in $\mathcal{O}(\sigma)$ time, where $\sigma = |\Sigma|$ is the alphabet size. Given $\Sigma = \{\mu_1, \mu_2, \ldots, \mu_\sigma\}$, we define Parikh vectors $P_r[1..\sigma]$, where $P_r[j]$ is the number of occurrences of μ_j in range $\boldsymbol{x_r}$. Since ranges are monotone nondecreasing in the letters of the alphabet, it is easy to compute all the $P_r, r = 1, 2, \ldots, m$, in linear time in a single scan of $\boldsymbol{x}$. Similarly, during the processing of each range $\boldsymbol{x_r}$, any value $P_{r,j}$, the Parikh vector of the suffix $\boldsymbol{x_r}[j..\ell_r]$, can be computed in constant time for each position considered. Thus we can determine the lexicographical order of any two ranges (or part ranges) $\boldsymbol{x_r}$ and $\boldsymbol{x_{r'}}$ in $\mathcal{O}(\sigma)$ time rather than time $\mathcal{O}(\max(\ell_r, \ell_{r'}))$. The variant of `NSV`* that uses Parikh vectors is called P`NSV`*; otherwise NP`NSV`* for Not Parikh.

Here we describe a simple data structure that yields an alternative approach to Algorithm `NSV`*, based on the comparison of longest Lyndon factors as described in Proof of Theorem 12. The ***dictionary of basic factors*** [28, 30] of string $\boldsymbol{x}[1..n]$ consists of a sequence of arrays $\mathcal{D}_t, 0 \leq t \leq \log n$. The array $\mathcal{D}_t$ records information about factors of $\boldsymbol{x}$ of length 2^t — that is, the basic factors. In particular, $\mathcal{D}_t[i]$ stores the rank of $\boldsymbol{x}[i..i + 2^t - 1]$, so that

$$\boldsymbol{x}[i..i + 2^t - 1] \preceq \boldsymbol{x}[i..i + 2^t - 1] \Leftrightarrow \mathcal{D}_t[i] \leq \mathcal{D}_t[i].$$

This dictionary requires $O(n \log n)$ space and can be constructed in $O(n \log n)$ time as follows. $\mathcal{D}_0$ contains information about consecutive symbols of $\boldsymbol{x}$ and hence can be computed in $O(n \log n)$ time by sorting all the symbols appearing in $\boldsymbol{x}$ and mapping

them to numbers from 1 and onward. Once $\mathcal{D}_t$ is computed, we can easily compute $\mathcal{D}_{t+1}$ by spending $O(n)$ time on a radix sort, because $u[i..i+2^{t+1}-1]$ is in fact a concatenation of the factors $u[i..i+2^t-1]$ and $u[i+2^t..i+2^{t+1}-1]$.

Once this dictionary is computed, we can compare any two factors by comparing two appropriate overlapping basic factors (i.e., factors having length power of two), which is done by checking the corresponding $\mathcal{D}$ array from the dictionary. This will require constant time and hence each suffix-suffix comparison can be done in constant time.

Now consider the worst case behaviour of Algorithm `NSV`*. Given the initial string $\boldsymbol{x_0} = a^hba^hc_0,\ h \geq 1,\ c_0 > b > a$, let $\boldsymbol{x_k^{(h)}} = \boldsymbol{x_k} = \boldsymbol{x_{k-1}}\boldsymbol{x^*_{k-1}},\ k = 1, 2, \ldots$, with $\boldsymbol{x^*_{k-1}}$ identical to $\boldsymbol{x_{k-1}}$ except in the last position, where the letter $c_k > c_{k-1}$ replaces c_{k-1}. Then $\boldsymbol{x_k}$ has length $n = (h{+}1)m$, where $m = 2^{k+1}$ is the number of ranges in $\boldsymbol{x_k}$.

Consider the vectors formed in the proof of Lemma 16 that count range matches. Each position in the righthand vector $(1, 0)$ is clearly largest possible over all selections of ranges, as are the preceding positions $(2, 2)$. Similarly, none of the values in $(3, 3, 3, 3)$ can possibly be greater than 3: in each case the three matches result from inequalities in the last positions of the ranges being matched. We see that in fact the vector corresponding to $\boldsymbol{x_k}$ must be maximal, and so, when each range match requires constant time (proportional to σ):

Lemma 19 *Algorithm* `PNSV`* *computes* $\boldsymbol{\lambda_x}$ *in* $\mathcal{O}(n \log n)$ *time for all* $\boldsymbol{x}$.

Consider now the execution of `NPNSV`* on the strings $\boldsymbol{x_k}$. Instead of one comparison per range match by `PNSV`*, now $h{+}1$ letter comparisons are required. For $h = 1$, the number of comparisons per range match is therefore 2, a multiple by a constant factor, thus still linear time per match. For arbitrary $h > 2$, the number of comparisons

increases by a factor of h, but at the same time range length (and therefore string length) increases by a factor of $(h+1)/2$, so that still $O(n \log n)$ ranges are processed in $O(n \log n)$ time. Thus

Lemma 20 *Algorithm* `NPNSV`* *computes* $\boldsymbol{\lambda_x}$ *in* $\mathcal{O}(n \log n)$ *time for all* $\boldsymbol{x}$.

4.6 Experimental Results

We have done preliminary tests on the algorithms described above, including the two variants of `NSV`*. The equipment used was an Intel(R) Core i3 at 1.8GHz and 4GB main memory under a 64-bit Windows 7 operating system. Figure 4.8 shows the results of exhaustive tests of the algorithms on all binary strings of lengths 11–22, with all but RDuval displaying linear-time behaviour. MaxLyn and NP`NSV`* are roughly equivalent in time requirement, with NSVISA several times slower, P`NSV`* perhaps 10 times slower.

We have also tested the linear average-case algorithms on much longer binary strings, several megabytes in length, both random and highly periodic [48]. On random strings, P`NSV`* and NP`NSV`* are comparable in speed and fastest by a factor of 2 or 3, while on the periodic strings, MaxLyn has an advantage by approximately the same margin. More testing needs to be done, especially on strings defined on larger alphabets, but of the current collection, it appears that the two new $\mathcal{O}(n \log n)$-time algorithms are the algorithms of choice.

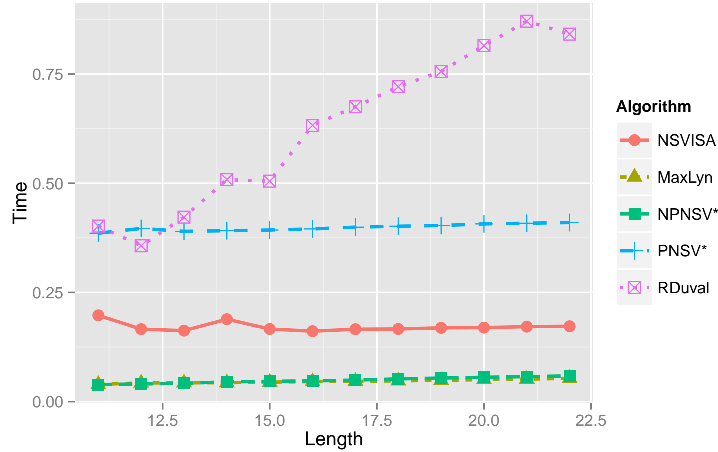

Figure 4.8: Five algorithms compared on all binary strings of lengths $n \in 11..22$: the average processing time for each n is given in 10^{-4} seconds. For all the algorithms except RDuval, the pre-processing time is independent of length n. So the increase of time is very small which results very small amount of slope for the corresponding lines.

4.7 Future Work

There is reason to believe [62] that the Lyndon array computation is less hard than suffix array construction. Recently an algorithm is proposed in [13, 14], that computes the sorted Lyndon Array in linear time using elementary methods. However the algorithm requires a deal of space. Thus a space-efficient linear-time algorithm is a high priority and left open as a possible future research direction.

Chapter 5

Reconstructing a String from its Lyndon Arrays

The contents of this chapter have been published in [7].

5.1 Introduction

Recall from Chpater 2, a primitive string $\boldsymbol{x}$ that is lexicographically least among all its rotations is said to be a Lyndon word. As a consequence of their interesting properties, Lyndon words have been much studied: the existence of a unique factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_s}$ of a string into Lyndon words $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_s}$ was demonstrated some 60 years ago [23] and a simple linear-time algorithm to compute the ***Lyndon factorization*** was proposed a quarter-century later [40].

In fact, Lyndon words are a special case of Unique Maximal Factorization Families (UMFFs), that over the last 15 years have also been studied extensively [32, 33, 35, 34]. When every factor $\boldsymbol{w_j}$, $1 \leq j \leq s$, of a (not necessarily Lyndon) factorization

of $\boldsymbol{x}$ belongs to a specified set $\mathcal{W}$, we say that it is a ***factorization of $\boldsymbol{x}$ over $\mathcal{W}$***, denoted by $F_{\mathcal{W}}(\boldsymbol{x})$. Then a subset $\mathcal{W} \subseteq \Sigma^+$ is a ***factorization family*** (FF) if and only if for every nonempty string $\boldsymbol{x}$ on Σ there exists a factorization $F_{\mathcal{W}}(\boldsymbol{x})$. If $\mathcal{W}$ is an FF on an alphabet Σ, then $\mathcal{W}$ is said to be a ***unique maximal factorization family*** (UMFF) if and only if there exists a unique factorization $F_{\mathcal{W}}(\boldsymbol{x})$ for every string $\boldsymbol{x} \in \Sigma^+$. We expect that the results given here for Lyndon arrays can be generalized to UMFFs.

The Lyndon array $\boldsymbol{\lambda} = \boldsymbol{\lambda_x}[1..n]$ (equivalently, $\boldsymbol{\mathcal{L}} = \boldsymbol{\mathcal{L}_x}[1..n]$) of a given $\boldsymbol{x} = \boldsymbol{x}[1..n]$ gives at each position i the length (equivalently, the end position) of the longest (or ***maximal***) Lyndon word starting at i. Thus $\boldsymbol{\mathcal{L}_x}[i] = \lambda_{\boldsymbol{x}}[i]+i-1$. Apparently first introduced as "Lyndon bracketing" [79], the Lyndon array has recently become of interest because of the central role it plays in the surprising and simple proof [15] that the maximum number $\rho(n)$ of maximal periodicities (runs) in any string of length n satisfies $\rho(n) < n$. In the previous chapter we showed algorithms to compute $\lambda_{\boldsymbol{x}}$, exhibiting several that apparently execute in linear expected time, while conjecturing that there exists a worst-case linear-time algorithm to compute $\lambda_{\boldsymbol{x}}$ that is "elementary" — not a precise term, but we intend by it an algorithm that computes local features of a string while avoiding prior computation of global data structures such as the suffix array. Indeed, such an algorithm has recently been found [13, 14] as a first step in a two-step non-recursive linear-time suffix array construction algorithm.

Here is an example of a Lyndon array, taken from [47]:

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a & a & b \\ \boldsymbol{\lambda_x} = & 2 & 1 & 5 & 2 & 1 & 2 & 1 & 3 & 2 & 1 \\ \mathcal{L}_{\boldsymbol{x}} = & 2 & 2 & 7 & 5 & 5 & 7 & 7 & 10 & 10 & 10 \end{array} \tag{5.1}$$

Since λ and $\mathcal{L}$ are arrays of positive integers, it is natural to ask under what conditions a given integer array is a Lyndon array. In Section 5.2 we give necessary and sufficient conditions that a given integer array $\mathcal{L}^*$ is a Lyndon array $\mathcal{L}_{\boldsymbol{x}}$ of some string $\boldsymbol{x}$ on some alphabet Σ. We then describe linear-time algorithms that compute a string $\boldsymbol{x}$ corresponding to a given Lyndon array $\mathcal{L}^*$ — the problem of computing a lexicographically least such string on a minimum-size alphabet appears to be computationally difficult. Finally we describe a linear-time algorithm to determine whether or not a given integer array is a Lyndon array of some string.

In Section 5.3 we go on to establish a "reverse engineering" result for Lyndon arrays; that is, given certain Lyndon arrays $\mathcal{L}_{\boldsymbol{x}}$ based on orderings of a given alphabet Σ of size σ, what can be said about the corresponding string $\boldsymbol{x}$? This kind of problem was first introduced in [46, 41] for the border array, then later considered for various common string data structures; for example, prefix tables [25, 2, 24], KMP arrays [42, 50, 51], cover arrays [29], and many others. Section 5.3 also presents an $\mathcal{O}(\sigma n)$-time algorithm to compute the unique string $\boldsymbol{x}$ determined by the Lyndon arrays computed for σ rotations of the alphabet. In Section 5.4 we discuss a variety of open problems arising.

5.2 When is $\mathcal{L}^*$ a Valid Lyndon Array of Some String?

Observation 1 from Chapter 4 tells us that a nonintersecting, or Monge-like, property necessarily holds for the arcs $(i, \mathcal{L}[i])$ determined by the Lyndon array $\mathcal{L}_{\boldsymbol{x}}$ of every string $\boldsymbol{x}$. To see that this property is also sufficient, consider an integer array $\mathcal{L}^*[1..n]$ in which $i \leq \mathcal{L}^*[i] \leq n$ for every $i \in 1..n$, and where either $\mathcal{L}^*[i] < j$ or $\mathcal{L}^*[i] \geq \mathcal{L}^*[j]$ for every $1 \leq i < j \leq n$. Suppose an alphabet $\Sigma = \{\mu_1, \mu_2, \ldots, \mu_n\}$ is given, with $\mu_1 < \mu_2 < \cdots < \mu_n$. We now outline an algorithm (see Figure 5.1) that assigns the n letters of Σ to positions in $\boldsymbol{x}$ in such a way that $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}^*$.

procedure SimpleAssign $(\mathcal{L}^*, n, \Sigma, \sigma; \boldsymbol{x})$
Radix sort pairs $(\mathcal{L}^*[i], i)$, $1 \leq i \leq n$, in ascending
  order of i within descending order of $\mathcal{L}^*[i]$
  to form sorted positions $I = I[1..n]$.
for $j \leftarrow 1$ **to** n **do**
  $\boldsymbol{x}[I[j]] \leftarrow j$

Figure 5.1: Given a valid Lyndon array $\mathcal{L}^*$ and an ordered alphabet $\Sigma = \{1, 2, \ldots, n\}$, in $\mathcal{O}(n)$ time construct a string $\boldsymbol{x}$ on Σ whose Lyndon array is $\mathcal{L}^*$.

This algorithm ensures that arcs $(i, \mathcal{L}^*[i])$ are processed in descending order of $\mathcal{L}^*[i]$ — specifically, so that for all $i_1, i_2, \ldots, i_m$ such that

$$\mathcal{L}^*[i_1] = \mathcal{L}^*[i_2] = \cdots = \mathcal{L}^*[i_m],\ i_1 < i_2 < \cdots < i_m,$$

it follows that $\boldsymbol{x}[i_1] < \boldsymbol{x}[i_2] < \cdots < \boldsymbol{x}[i_m]$. Thus for each choice of $\mathcal{L}^*[i]$, we ensure that $\mathcal{L}^*[i] = \mathcal{L}_{\boldsymbol{x}}[i]$. The descending order, together with the nonintersecting property, then guarantees that this identity holds for all i, and so the nonintersecting property is sufficient to ensure that $\mathcal{L}^*$ is the Lyndon array of some string — in particular $\boldsymbol{x}$.

Thus:

Lemma 1 *Suppose that $\mathcal{L}^*[1..n]$ is an integer array such that $1 \leq \mathcal{L}^*[i] \leq n$ for all $i \in 1..n$. Then $\mathcal{L}^*$ is a Lyndon array $\mathcal{L}_{\boldsymbol{x}}$ of some string $\boldsymbol{x}$ if and only if for all i, j such that $1 \leq i < j \leq n$, either $\mathcal{L}^*[i] < j$ or $\mathcal{L}^*[i] \geq \mathcal{L}^*[j]$.*

For the example (5.1), Algorithm SimpleAssign yields the following:

$$
\begin{array}{rcccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} \\
\mathcal{L}^* = & 2 & 2 & 7 & 5 & 5 & 7 & 7 & 10 & 10 & 10 \\
I = & 8 & 9 & 10 & 3 & 6 & 7 & 4 & 5 & 1 & 2 \\
\boldsymbol{x} = & 9 & 10 & 4 & 7 & 8 & 5 & 6 & 1 & 2 & 3
\end{array}
\tag{5.2}
$$

In an effort to construct a string $\boldsymbol{x}$ on a smaller alphabet, we employ a strategy (see Figure 5.2) that for each range of increasing values in I (that is, for each maximal Lyndon word of length at least two):

- chooses an initial letter one greater than the initial letter in the immediately following maximal Lyndon word;
- assigns the same letter to consecutive positions at the beginning of the current maximal Lyndon word — but excluding the final position;
- thereafter increments the letter by one at each successive position.

For example, in (5.2), after selecting $\boldsymbol{x}[8..10] = 112$, we choose

$$\boldsymbol{x}[3] = 2,\ \boldsymbol{x}[6] = 3,\ \boldsymbol{x}[7] = 4,$$

corresponding to $I[4..6] = 367$ and ensuring that $\mathcal{L}_{\boldsymbol{x}}[5] \leq 5$. Then, for $I[7..8] = 45$, we choose

$$\boldsymbol{x}[4] = 4,\ \boldsymbol{x}[5] = 5,$$

finally yielding $\boldsymbol{x} = 3424534112$, on an alphabet of size 5 rather than 10, but still far from the minimum of 2 ($\boldsymbol{x} = 1211212112$). Clearly Algorithm BetterAssign also executes in $\mathcal{O}(n)$ time; it yields the same worst-case result as SimpleAssign (when $I = n, n-1, \ldots, 1$), but otherwise finds a string $\boldsymbol{x}$ on a smaller alphabet.

procedure BetterAssign $(\mathcal{L}^*, n, \Sigma, \sigma; \boldsymbol{x})$
Compute $I[1..n]$ as in SimpleAssign
$I[n+1] \leftarrow 0; h \leftarrow 1; i \leftarrow 1$
while $i \leq n$ **do**
▷ *Assign letters to range of increasing values from* $I[i]$.
  repeat
  ▷ *Consecutive positions at start of range are identical.*
    $\boldsymbol{x}[I[i]] \leftarrow h;\ i \leftarrow i+1$
  until $I[i+1] \neq I[i]+1$
  if $I[i+1] < I[i]$ **then**
  ▷ *End position in range must be incremented.*
    $\boldsymbol{x}[I[i]] \leftarrow h+1;\ i \leftarrow i+1$
  else
    repeat
    ▷ *Elsewhere in range every position is incremented.*
      $h \leftarrow h+1;\ \boldsymbol{x}[I[i]] \leftarrow h;\ i \leftarrow i+1$
    until $I[i] < I[i-1]$
▷ *Reset* h *depending on the next range in* $\boldsymbol{x}$.
  if $i \leq n$ **then**
    $j \leftarrow i$
    while $I[j+1] > I[j]$ **do** $j \leftarrow j+1$
      $h \leftarrow \boldsymbol{x}[I[j]+1]+1$

Figure 5.2: Construct a string $\boldsymbol{x}$ on a subset of $\Sigma = \{1, 2, \ldots, n\}$ with Lyndon array L^*.

We know of no approach other than brute force (trial and error) to the computation of $\boldsymbol{x}$ on a minimum alphabet consistent with $\mathcal{L}^*$. Hence

Problem 2 *Given a valid Lyndon array L^*, what is the complexity of the problem of constructing a string $\boldsymbol{x}$ on a minimum alphabet consistent with L^*?*

We turn now to the problem of determining whether or not a given integer array $\mathcal{L}^*$ is valid; that is, whether or not it is a Lyndon array of some string. To solve this problem we introduce Algorithm CheckLyndon (see Figure 5.3), based on Lemma 1. It processes the segments $(i, \mathcal{L}^*[i])$ in ascending order of position i and places the "*end*" of each nontrivial segment on the stack. Before doing so, it checks to see if any previous *end* lies within the current segment: if so, $\mathcal{L}^*$ cannot be a Lyndon array. If not, then either the previous entry ended before the current range and so can be deleted from the stack, or else it includes the current range and so must be kept in the stack to be tested against later segments. Note that entries in the stack have all been tested against *preceding* segments. We claim therefore that CheckLyndon is correct.

To see that the algorithm executes in linear time, observe that segment i is either wholly contained in a preceding segment, so that access to the stack is terminated, or else the current stack entry is deleted. Thus the total time requirement of the **while** loop is $\mathcal{O}(n)$. We have:

Lemma 3 *Algorithm CheckLyndon correctly determines in $\mathcal{O}(n)$ time whether or not a given integer array $\mathcal{L}^*[1..n]$ is a Lyndon array.*

```
function CheckLyndon (L*, n)
STACK ← ∅
for i ← 1 to n do
    if L*[i] < i or L*[i] > n return (FALSE)
    if L*[i] > i then
        end ← 0
        while STACK ≠ ∅ and end < L*[i] do
            end ← peek(STACK)
            if end < i then pop(STACK)
            elsif end < L*[i] then return (FALSE)
        push(STACK, L*[i])
return (TRUE)
```

Figure 5.3: Determine whether (TRUE) or not (FALSE) a given integer array $\mathcal{L}^*$ is a Lyndon array of some string.

5.3 Reconstructing a String from its Lyndon Arrays

Suppose an alphabet $\Sigma = \{\ell_1, \ell_2, \ldots, \ell_\sigma\}$, $\sigma \geq 2$, is given with initial global order R_1: $\ell_1 < \ell_2 < \cdots < \ell_\sigma$. For $j = 2, 3, \ldots, \sigma$, the jth rotation R_j of R_1 is the order $\ell_j < \ell_{j+1} < \cdots < \ell_\sigma < \ell_1 < \cdots < \ell_{j-1}$. Thus ℓ_j is the least letter, and for $j > 1$ ℓ_{j-1} is the largest, in the rotation R_j. The collection of σ rotations is denoted by R_Σ.

In this section we deal with the problem of identifying a unique string on alphabet Σ corresponding to R_Σ. We begin with an observation from [47], that we can write $\boldsymbol{x}$ in the form $\boldsymbol{x_1 x_2} \cdots \boldsymbol{x_m}$, where for each $r \in 1..m$, $|\boldsymbol{x_r}| = len_r$ and

$$\boldsymbol{x_r}[1] \leq \boldsymbol{x_r}[2] \leq \cdots \leq \boldsymbol{x_r}[len_r], \tag{5.3}$$

while for $1 \leq r < m$,

$$\boldsymbol{x_r}[len_r] > \boldsymbol{x_{r+1}}[1]. \tag{5.4}$$

We call $\boldsymbol{x_r}$ a ***range*** in $\boldsymbol{x}$, and we identify a position j in range $\boldsymbol{x_r}$, $1 \leq j \leq len_r$, with its equivalent position i in $\boldsymbol{x}$ by writing $i = S_{r,j} = \sum_{r'=1}^{r-1} len_{r'} + j$. Then, again from [47], we have the following:

Observation 4 *Let $i = S_{r,j}$ be a position in $\boldsymbol{x}$ that corresponds to position j in range $\boldsymbol{x_r}$.*

(a) If $\boldsymbol{x_r}[j] = \boldsymbol{x_r}[len_r]$, then $\mathcal{L}_{\boldsymbol{x}}[i] = i$.

(b) Otherwise, $\mathcal{L}_{\boldsymbol{x}}[i] = i'$, where i' is the final position in some range $\boldsymbol{x_{r'}}$, $r' \geq r$; that is, $i' = \sum_{s=1}^{r'} len_s$.

Based on these remarks, for the special case $\sigma = 2$, we can now prove:

Lemma 5 *Let $\mathcal{L}_{\boldsymbol{x}}$ be the Lyndon array of a string $\boldsymbol{x}[1..n]$ on $\Sigma = \{a, b\}$, $a < b$. Then, provided that $\lambda_{\boldsymbol{x}} \neq 1^n$, $\boldsymbol{x}$ is determined uniquely by $\mathcal{L}_{\boldsymbol{x}}$.*

Proof. First observe that $\lambda_{\boldsymbol{x}} = 1^n$ if and only if $\boldsymbol{x} = b^m a^{n-m}$ for some $m \in 0..n$ — thus in this case the corresponding $L_{\boldsymbol{x}} = 12 \cdots n$ corresponds to $n+1$ choices for $\boldsymbol{x}$. Otherwise, let n' be the smallest index such that for every $i \in n'..n$, there exists no $j < i$ such that $\mathcal{L}_{\boldsymbol{x}}[j] = i$. If there is no such n', then $L_{\boldsymbol{x}}[j] = n$ for some $j < n$; in this case, set $n' = n+1$. Then for every $i \in n'..n$, $L_{\boldsymbol{x}}[i] = i$, which, since $\boldsymbol{x} \neq b^n$ by hypothesis, implies that $\boldsymbol{x}[i] = a$; that is, $\boldsymbol{x}[n'..n] = a^{n-n'+1}$. Since, again by hypothesis, $\boldsymbol{x} \neq a^n$, it follows that $n' > 2$ and $\boldsymbol{x}[n'-1] = b$. More generally, by Observation 4, for every $i < n'$, $\boldsymbol{x}[i] = b$ if and only if $\mathcal{L}_{\boldsymbol{x}}[i] = i$. Therefore, for all other i, $\boldsymbol{x}[i] = a$. Thus $\boldsymbol{x}$ is uniquely determined by $\mathcal{L}_{\boldsymbol{x}}$, as required. □

Suppose then that $\sigma \geq 3$. In this case, corresponding to R_Σ, we assume that Lyndon arrays $\lambda_\Sigma = \{\lambda_1, \lambda_2, \ldots, \lambda_\sigma\}$ are given, where λ_j, $1 \leq j \leq \sigma$, is based on

rotation R_j of the alphabet and determined by some (unknown) string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Then for $j \in 1..\sigma,\ i \in 1..n$, $\lambda_j[i]$ is the length of the maximal Lyndon word at position i in $\boldsymbol{x}$ based on rotation R_j of the alphabet. We say that position i in $\boldsymbol{x}$, $1 \le i \le n$, is ***covered*** by λ_j if and only if there exists a position $i' < i$ such that $i' + \lambda_j[i'] > i$ (alternatively, $L_j[i'] \ge i$).

We now prove the following result, enabling us to uniquely determine $\boldsymbol{x}$ from R_Σ:

Theorem 6 *Suppose that $\boldsymbol{x}$ is a string on an alphabet Σ of size $\sigma \ge 3$, whose Lyndon arrays λ_Σ are given based on rotations R_Σ. Let*

$$\lambda^+[i] = max(\lambda_1[i], \lambda_2[i], \dots, \lambda_\sigma[i]),\ 1 \le i \le n,$$

and let $P[i]$ be the nonempty ascending sequence $\{j_1, j_2, \dots, j_k\}$ of indices j that specify all the rotations R_j for which $\lambda^+[i] = \lambda_j[i]$; that is, that yield the maximum Lyndon word at position i. Then

1. $|P[i]| = 1 \implies \boldsymbol{x}[i] = \ell_{j_1}$.

2. $1 < |P[i]| < \sigma \implies \boldsymbol{x}[i] = \ell_{j_h}$, *where j_h is the unique (leftmost) entry in $P[i]$ such that $j_{(h+1) \bmod \sigma} \notin P[i]$.*

3. $|P[i]| = \sigma \iff \lambda^+[i] = 1$, *and $\boldsymbol{x}[i..n] = \ell_{j_h}^{n-i+1}$, where i is the least integer such that $\lambda^+[i] = 1$ and j_h is the unique entry in $P[i]$ such that $\boldsymbol{x}[i]$ is* not *covered by λ_{j_h}.*

We remark that, since the assignments to positions in $\boldsymbol{x}$ made here under Cases 1–3 are unique, therefore $\boldsymbol{x}$ must be the only string on Σ that satisfies the constraints given by λ_Σ. In Figure 5.4 the various cases of Theorem 6 are illustrated:

$$
\begin{array}{rcccccccl}
i & = & 1 & 2 & 3 & 4 & 5 & \\
\lambda_1 & = & 1 & \underline{4} & 3 & 2 & 1 & (a < b < c) \\
\lambda_2 & = & \underline{2} & 1 & \underline{3} & \underline{2} & 1 & (b < c < a) \\
\lambda_3 & = & 1 & 3 & 1 & 1 & \underline{1} & (c < a < b) \\
\lambda^+ & = & 2 & 4 & 3 & 2 & 1 & \\
k & = & 1 & 1 & 2 & 2 & 3 & \\
\boldsymbol{x} & = & b & a & b & b & c &
\end{array}
$$

Figure 5.4: Lyndon arrays based on rotated orders for $\sigma = 3$.

- in columns $i = 1, 2$, the respective maximum values $\lambda_2[1] = 2,\ \lambda_1[2] = 4$ occur only once, so that Case 1 applies, and $\boldsymbol{x}[1] = b,\ \boldsymbol{x}[2] = a$;
- in column $i = 3$, we find $k = 2 < \sigma$, so Case 2 applies, and since $2 \in P[3],\ 3 \notin P[3]$, we choose $\boldsymbol{x}[3] = \ell_2 = b$;
- similarly in column $i = 4$, Case 2 appies and again we choose $\boldsymbol{x}[4] = b$;
- of course in column $i = n = 5$, Case 3 applies, and we set $\boldsymbol{x}[5] = c$ because, while position 5 is covered by preceding entries in λ_1 and λ_2, it is not covered by any preceding entry in λ_3.

In order to prove Theorem 6, we first need the following:

Lemma 7 *Let $i \in 1..n$ be a position in $\boldsymbol{x}$ such that $\boldsymbol{x}[i] = \ell_j \in \Sigma$ for some $j \in 1..\sigma$, $\sigma \geq 3$. Then $\lambda^+[i] = \lambda_j[i] \geq \lambda_{j'}[i]$ for every $\ell_{j'} \in \Sigma$.*

Proof. Assume the contrary. Then there exists $j' \neq j$ ($\ell_{j'} \neq \ell_j$) such that $\lambda_{j'}[i] > \lambda_j[i]$. Suppose now that for some position h satisfying $i < h < i+\lambda_j[i]$ (in the range of the Lyndon word corresponding to R_j that begins at i), $\boldsymbol{x}[h] = \ell_*$, where ℓ_* is a letter such that $\ell_* < \ell_j$ in $R_{j'}$. But this implies that $\lambda_{j'}[i] < \lambda_j[i]$, contradicting our original assumption. Thus every letter ℓ that occurs in the range $\boldsymbol{x}[i..i+\lambda_j-1]$ must

satisfy $\ell \geq \ell_j$ in both R_j and $R_{j'}$. Since the same condition holds for both rotations, therefore $\lambda_{j'}[i]$ can be at most equal to $\lambda_j[i]$. □

If now we suppose, as in Case 1 of Theorem 6, that there exists a single j_1 such that $\lambda_{j_1}[i]$ is maximum, then it follows immediately from Lemma 7 that, for every rotation R_t, $t \in 1..\sigma$, except $t = j_1$, $\boldsymbol{x}[i] \neq \ell_t$. Thus $\boldsymbol{x}[i] = \ell_{j_1}$, establishing Case 1.

The next result gives us a basis for establishing Case 2 by providing a simple characterization of the entries in $P[i]$ when $1 < |P[i]| < \sigma$:

Lemma 8 *Suppose that j_1 is the least value and $j_2 > j_1$ the greatest value (with $j_2 - j_1 < \sigma - 1$) such that for some $i \in 1..n$, $\lambda^+[i] = \lambda_{j_1}[i] = \lambda_{j_2}[i]$. Then $P[i]$ contains exactly one of*

$$\begin{aligned} P_1 &= j_1, j_1{+}1, \ldots, j_2{-}1, \\ P_2 &= j_2, j_2{+}1, \ldots, \sigma, 1, 2, \ldots, j_1{-}1, \end{aligned}$$

where we auppose that the sequence $1, 2, \ldots, j_1{-}1$ is empty if $j_1 = 1$.

Proof. By hypothesis $\boldsymbol{x}^* = \boldsymbol{x}[i..i{+}\lambda^+[i]{-}1]$ is a Lyndon word in both orders R_{j_1} and R_{j_2}. Therefore every letter in $\boldsymbol{x}^*$ must be greater than or equal to $\boldsymbol{x}[i]$ in *both* orders

$$\begin{aligned} J_1 = &\ \{j_1 < j_1{+}1 < \cdots < j_2{-}1\}, \\ J_2 = &\ \{j_2 < j_2{+}1 < \cdots < \sigma < 1 < 2 \cdots < j_1{-}1\}, \end{aligned} \tag{5.5}$$

where again we must take account of the special case $j_1 = 1$. We can write $R_{j_1} \equiv J_1\{j_2{-}1 < j_2\}J_2$ and $R_{j_2} \equiv J_2\{j_1{-}1 < j_1\}J_1$. Now observe that if $\boldsymbol{x}^*$ contains letters from *both* J_1 and J_2, there must be at least one letter in one of the two orderings

that is less than $\boldsymbol{x}[i]$, and so $\boldsymbol{x}^*$ cannot be a Lyndon word in one of R_{j_1}, R_{j_2}. Thus $\boldsymbol{x}^*$ contains letters from exactly one of P_1, P_2, as required. □

In the context of Lemma 8, consider a maximal sequence of entries

$$j', j'+1, \ldots, j'+t,\ t > 0, \tag{5.6}$$

in $\lambda^+[i]$, where $(j'+t+1) \bmod \sigma \notin \lambda^+[i]$ — as in Lemma 8, we suppose that the sequence is circular, so that 1 follows σ. Recall that $R_{j'+1}$ is the rotation of $R_{j'}$ that turns the least letter $\ell_{j'}$ of rotation $R_{j'}$ into the greatest letter of rotation $R_{j'+1}$. Thus the occurrence of $j'+1$ in the sequence $\lambda^+[i]$ ensures that the letter $\ell_{j'}$ cannot be the first letter of the Lyndon array at position i — if it were, then in $R_{j'+1}$, we could have only $\lambda_{j'}[i] = 1$, certainly not maximum. It follows that only the final letter $\ell_{j'+t}$ in the sequence (5.6) can be the first letter of the Lyndon array at i, because it is the only letter that is not rotated. Noting that in both of the two possible orders given in (5.5) $j'+t$ will be the *leftmost* occurrence in $P[i]$, we thus establish Case 2 of Theorem 6.

In order to deal with Case 3, we first need:

Lemma 9 $|P[i]| = \sigma \iff \lambda^+[i] = 1$.

Proof. Suppose $|P[i]| = \sigma$. Since every letter in Σ occurs on the right in some rotation R_j of the alphabet, and so is maximum, it follows that $\lambda_j[i] = 1$ for some j. But since every such j yields a maximum $\lambda_j[i]$, it follows that $\lambda^+[i] = 1$. Conversely, if $\lambda^+[i] = 1$, then every rotation R_j yields $\lambda_j[i] = 1$, so that $|P[i]| = \sigma$. □

```
procedure ConstructString (λ, σ, n; x)
i ← 1; cover ← σ^n
while i < n do
    maxlen ← 0        — Length of maximum λ[1..σ, i]
    freqmax ← 1
    for j ← 1 to σ do
▷ Treat λ as a two-dimensional array; find final
▷ letter jmax & frequency freqmax for maxlen.
        if λ[j, i] > maxlen then
            maxlen ← λ[j, i]; jmax ← j; freqmax ← 1
        elsif λ[j, i] = maxlen then
            jmax ← j; freqmax ← freqmax+1
▷ Recompute maximum range covered by letter j.
        cover[j] ← max(cover[j], j+λ[j, i])
    if freqmax < σ then
▷ Cases 1 and 2 — Lemmas 7 & 8.
        x[i] ← jmax; i ← i+1
    else
▷ Case 3 — Lemma 9.
        j′ ← 1
▷ Find the letter that yields no cover of position i.
        while j′ ≤ σ and i < cover[j′] do j′ ← j′+1
        x[i] ← j′; i ← i+1
▷ In accordance with Lemma 10, extend to position n.
        while i ≤ n do x[i] ← j′; i ← i+1
```

Figure 5.5: Constructing a string from rotated Lyndon arrays.

Now consider $i = n$, and note that for every rotation R_j, $\lambda_j[n] = 1$, so that $\lambda^+[n] = 1$. Note also that in this case $\boldsymbol{x}[n]$ will be covered by $\lambda_j[i]$ for every position $i < n$ that marks the rightmost occurrence of ℓ_j in $\boldsymbol{x}$ — except for j such that $\ell_j = \boldsymbol{x}[i] = \boldsymbol{x}[n]$. Thus the letter ℓ_j satisfies exactly one of two conditions:

- either ℓ_j does not previously occur in $\boldsymbol{x}$; or
- under rotation R_j, at any position $i < n$ such that $\boldsymbol{x}[i] = \ell_j$, $\lambda_j[i]$ does not cover $\boldsymbol{x}[n]$.

Hence Case 3 provides the basis for assigning $\boldsymbol{x}[n]$. To complete the proof of Theorem 6, we need one more result:

Lemma 10 *If $|P[i]| = \sigma$ for $i < n$, then $|P[i']| = \sigma$ for every $i' \in i..n$.*

Proof. We know $\boldsymbol{x}[i] = \ell_j$, a minimum letter under rotation R_j, and from Lemma 9 we know that $\lambda^+[i] = 1$. This is possible only if the same minimum letter occurs also at positions $i+1, i+2, \ldots, n$, as required. □

Therefore Case 3 identifies strings $\boldsymbol{x}$ with suffix ℓ^{n-i+1} for some i and some ℓ.

Theorem 6 justifies Algorithm ConstructString, shown in Figure 5.5, that in $\mathcal{O}(\sigma n)$ time constructs the unique string $\boldsymbol{x}$ on a given ordered alphabet Σ, based on σ rotations of a given Lyndon array λ.

5.4 Future Research

In this chapter we have started to analyze the relationship between a string and its Lyndon arrays corresponding to cyclic orderings of the underlying alphabet. Many

$$
\begin{array}{rcllll}
i & = & 1 & 2 & 3 & 4 \\
\boldsymbol{x} & = & \mathrm{a} & \mathrm{d} & \mathrm{b} & \mathrm{c} \\
\lambda_1 & = & 4 & 1 & 2 & 1 \quad (a < b < c < d) \text{ I} \\
\lambda_2 & = & 1 & 1 & 2 & 1 \quad (b < c < d < a) \text{ II} \\
\lambda_3 & = & 1 & 2 & 1 & 1 \quad (c < d < a < b) \text{ III} \\
\lambda_4 & = & 1 & 3 & 2 & 1 \quad (d < a < b < c) \text{ IV}
\end{array}
$$

Figure 5.6: $\boldsymbol{x} = adbc$ and its Lyndon arrays: consideration of fewer than four rotations of the alphabet may not allow $\boldsymbol{x}$ to be reconstructed.

problem areas remain:

1. As indicated by Problem 2, we know of no efficient algorithm to compute a string on a minimum alphabet consistent with a given valid Lyndon array $\mathcal{L}^*$.

2. Similarly, as we have seen, it appears to be difficult to reconstruct a string exactly from Lyndon arrays. In Section 5.3 we have presented an algorithm to reconstruct a string $\boldsymbol{x}$ on an alphabet of size σ given Lyndon arrays of $\boldsymbol{x}$ based on σ rotations of the alphabet. It appears that indeed in the worst case σ such rotations are required, as shown by the example in Figure 5.6.

   If in Figure 5.6 only rotations I–III are considered, then the selection $\boldsymbol{x}[2] = c$ rather than d would be made; if rotations II–IV were used, no determination could be made for $\boldsymbol{x}[1]$; and if rotations I, III, IV were used, column $i = 3$ would become an instance of Case 2(b), and $\boldsymbol{x}[2] = b$ could not be selected. Thus it appears that, on an alphabet of size $\sigma > 2$, the Lyndon array does not strongly determine the underlying string.

3. Thus it would be of interest to determine minimal criteria for the reconstruction of $\boldsymbol{x}$ — the least number of rotations or the size of the smallest alphabet. Is

there any hope of reconstructing a string based on fewer than σ permutations of the alphabet?

4. The study of UMFFs has led to the idea of V-words, analogous structures to Lyndon words, derived from a global non-lexicographic ordering of strings called V-order [31, 36, 37, 5]. Also, linear-time algorithms for computing Lyndon border and Lyndon suffix arrays have recently been proposed [6]. There is scope to extend the results of this chapter to the UMFF based on V-order [32], then further to UMFFs in their full generality [33, 35, 34].

Chapter 6

Microsatellite Evolution

6.1 Introduction

Microsatellites are composed of short DNA sequences between $1-6$ bp in length and repeated in tandem. In eukaryotic genomes, perfect or near-perfect tandem iterations of short sequence motifs of this kind are extremely common. Between closely related species, their distribution and density in genomes can vary greatly. In the case of the human genome, they are found at hundreds of thousands of places along chromosomes [64]. Every possible motif of mono-, di, tri- and tetranucleotide repeat is found frequently in the genome. Also referred to as short tandem repeats (STRs) or simple sequence repeats (SSRs), the ubiquitous occurrence of microsatellites was first reported in the 1980s [65].

The analysis of DNA sequence variation is of major importance in genetic studies. In this context, molecular markers are a useful tool for assaying genetic variation and have greatly enhanced the genetic analysis of organisms. Because of their reproducibility, multiallelic nature, codominant inheritance, relative abundance and good

genome coverage, STR markers are chosen over different classes of molecular markers for a variety of applications in genetics and breeding [75]. STR markers have been useful for integrating the genetic, physical and sequence-based maps in different species. They also have provided breeders and geneticists with an efficient tool to link phenotypic and genotypic variation [53]. Microsatellites are also suitable for the study of population structure and pedigree analysis. PCR (polymerase chain reaction) for microsatellites can be automated for identifying simple sequence repeat polymorphisms. Small amounts of blood or alcohol preserved tissue is adequate for conducting microsatellite analyses. Most microsatellites are noncoding, and therefore variations are independent of natural selection. These properties make microsatellites ideal genetic markers for conservation genetics.

Over 20 unstable microsatellite repeats have also been identified as the cause of neurological disease in humans [21]. Microsatellite instability (MSI) is the condition of genetic change that results from impaired DNA mismatch repair (MMR). Additionally, high-level microsatellite instability is responsible for a subset of colorectal cancers. Unstable repeats can be located in the coding or the non-coding region of a gene. Understanding the pathogenic mechanisms underlying these diseases may help to find remedy of these diseases. By expanding and contracting in length (increasing or decreasing the length), microsatellites can increase or decrease levels of gene expression when found near a gene's transcription start site, i.e., in the 'promoter'. Microsatellites in these regions can regulate gene expression by forming unusual DNA secondary structures [85].

A mutation model of microsatellite evolution is needed if allele frequency data from two groups of individuals (for example, populations or species) are to be used

for estimating the genetic distance between them. A wide range of models of the evolutionary dynamics of microsatellites has been presented, most of which derive from the stepwise mutation model (SMM) [60] in which, upon a mutation, 1 repeat unit is either gained, resulting in an expansion, or lost, resulting in a contraction. However, recent data on germline microsatellite mutation events confirm that single-step changes are the most common class of mutations, but that changes involving two to five repeat units do occur as well [86, 76]. The two-phase model (TPM) of DiRienzo et al. [39] addresses this by allowing infrequent large jumps in repeat number while keeping most mutations as single-step changes. Under these models microsatellites are expected to grow or contract unconstrained over time as there is no bias toward an expansion or a contraction.

An attractive model of microsatellite evolution holds that a genome-wide distribution of microsatellite repeat length that is at equilibrium results from a balance between length and point mutations [63, 22]. According to this model, two opposing mutational forces operate on microsatellite sequences. The rate of length mutations increases with increasing repeat count and help loci to attain arbitrarily high values. Point mutations break long repeat arrays into smaller units. A balance these two mutations results an equilibrium distribution. To explain the presence of a linear bias, Garza et al. [49] proposed a target or focal length; microsatellites below the focal length tend to expand, and those above it tend to contract. Since it can explain differences in microsatellite distribution among species and provides an elegant solution to the problem of why microsatellites do not expand into enormous arrays this model, or derivatives thereof, has been well received in recent years.

From the above discussion we can say that we have three sets of contrasting features in the existing models of microsatellite evolution. The first is one-phase vs two-phase mutations. Which indicate whether the microsatellites expand or contract by one unit or more. The second is mutation rate proportionality vs rate equality i.e. the repeat length of microsatellites affect the mutation rate or not. And the final one is the presence or absence of mutational bias. This encompasses the probability of expansion or contraction depending on the repeat length of the mutating microsatellites. In [78], the authors presented a group of models based on these these observations. They addressed only two different biases, constant bias, where the probability that a mutation results in an expansion is constant for all alleles, and linear bias, where this probability varies linearly with repeat length. We will work on those models with different dataset. In Section 6.2 we will describe the models and parameters. We will try to find the dynamics of the evolution of Microsatellite with repeat length 2 by using Human and Chimp data in Section 6.3. Section 6.4 contains the work on intra-species analysis where we consider different Human populations. In Section 6.5 we focus our attention for Microsatellites with repeat length 3. Finally we conclude in Section 6.6.

6.2 Models and Parameters

The data D for our study are a $d \times N$ matrix of microsatellite allele lengths from N loci homologous in d population. When $d = 2$, then we consider only humans and chimps (inter species comparison). When $d > 2$, then we consider different populations of humans (intra species comparison). We model the distribution of D by superimposing $d + 1$ Markov chains, on the ancestral and d populations branches, respectively, of the

d-taxa tree τ. The Markov chains have truncated state space $S = \{\kappa, \kappa+1, \ldots, \Omega\}$. Where κ is the smallest repeat number in the data and Ω is set to 40. The likelihood for given Data $D_i = (C_i, H_i)$ at locus i, is

$$L_i(\Theta, \lambda | D_i) := \sum\nolimits_{j \in S} \pi_j^{(a)} P_{j,C_i}^{(c)}(\lambda_c) P_{j,H_i}^{(h)}(\lambda_h) \tag{6.1}$$

For number of loci N assumed to be independent, the likelihood of total data D

$$L(\Theta, \lambda | D) := \prod_{i=1}^{N} L_i(\Theta, \lambda | D_i) \tag{6.2}$$

For an ergodic continuous-time Markov chain, its transition probability matrix $\mathbf{P}(\lambda) := (P_{i,j})_{i,j=\kappa}^{\Omega} = exp\{\mathbf{Q}\lambda\}$, where $\mathbf{Q} := (q_{i,j})_{i,j=\kappa}^{\Omega}$. $\mathbf{Q}$ is defined in Equation 6.3.

$$q_{ij} = \begin{cases} \beta(i,s)\alpha(u,v,i)(p + (1-p)\gamma(m,i,j)) & i = j-1 \\ \beta(i,s)\alpha(u,v,i)(1-p)\gamma(m,i,j) & i < j-1 \\ \beta(i,s)(1-\alpha(u,v,i))(p + (1-p)\gamma(m,i,j)) & i = j-1 \\ \beta(i,s)(1-\alpha(u,v,i))(1-p)\gamma(m,i,j) & i = j-1 \\ \sum_{i \neq j} q_{ij} & i = j \end{cases} \tag{6.3}$$

Where $\gamma(m, i, j)$ is defined by Equation 6.4

$$\gamma(m,i,j) = \begin{cases} \dfrac{m(1-m)^{|i-j|-1}}{1-(1-m)^{\Omega-1}} & \kappa \le i < j \le \Omega \\ \dfrac{m(1-m)^{|i-j|-1}}{1-(1-m)^{i-\kappa}} & \kappa \le j < i \le \Omega \end{cases} \tag{6.4}$$

Also, $\beta(i, s) = \mu(1 + (i - \kappa)s)$ is the mutation rate of allele i and $\alpha(u, v, i) = max(0, min(1, u - v(i - \kappa)))$ is the probability that a mutation results in an expansion.

The details of the parameters are given below.

- When $p = 1$, any microsatellite allele mutates, is set(i.e., expands or contracts) by only 1 unit of repeat at 0. But when $p < 1$, it mutates by 1 or more unit(s) of length with probability $1 - p$ and by 1 unit of length with probability p. Given that an allele i undergoes a multistep mutation, the probability of expanding or contracting by k units is given by $\gamma(m, i, j)$. It is possible to obtain the one-phase models from Equation 6.3 by setting $p = 0$ to allow mutations of length 1 and setting $m = 1$ to force the geometric distribution in Equation 6.4 to put all its mass on one-step mutations. When $m < 1$ we have multi-step mutation models. In our experiments we set p=0 for all the models.

- The proportional dependence of mutation rate is captured by the proportional rate parameter $s \in (-1/(\Omega - \kappa + 1), \infty$ in $\beta(i, s)$. When $s = 0$ alleles of all lengths have the same mutation rate $\mu = (0, \infty)$ of allele i. Thus, s represents the strength of length dependence of the mutation rate.

- In the function $\alpha(u, v, i)$, the constant bias parameter is $u \in [0, 1]$ and the linear bias parameter is $v \in (-\infty, +\infty)$. If $u = 0.5$ and $v = 0$, then the mutational process is symmetric and unbiased in which the probability that a mutation is an expansion or a contraction is equal. If $v = 0$ then $\alpha(u, v, i) = u \in [0, 1]$ for any allele i and the model will have constant mutational bias. Setting $v \neq 0$ will result in a linear mutational bias. If $0.5 < u < 1$ and $(u - 0.5)/(\Omega - \kappa) < v < \infty$ then the probability of contraction will be equal of expansion in the focal length

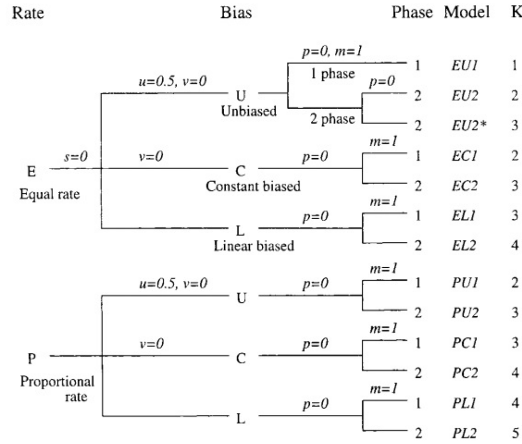

Figure 6.1: Model Description. K denotes the number of free parameters. The fixed parameter(s) for each set of models is/are shown above branch leading to it

$f = ((u - 0.5)/v) + \kappa$ towards which the mutational process is linearly biased. So when $i < f$ the mutational bias is upward toward f and when $i > f$ the mutational bias is downward toward f.

We will compare the Models through a second order Akaike information criterion (AIC_c) [84]. The best candidate model with a total of K parameters in (Θ, λ) is the one that minimizes the quantity

$$AIC_c = -2 \log L(\Theta, \lambda | D) + 2K + \frac{2K(K+1)}{N-K-1} \tag{6.5}$$

From Equation 6.5, it can be seen that for large N and small K, the value of AIC_c mainly depends on $\log L(\Theta, \lambda | D)$. For example if Model A has 1 more parameter than Model B and $L_A > L_B + 1$, then Model A is statistically better than Model B. We will use this technique to compare Models, throughout this chapter.

The Models and their associate parameters are shown in the Figure 6.1.

6.3 Inter Species Comparison for Microsatellite with Repeat Length 2

To find the homologous loci in the pair of human and chimp, at first we obtain the repeat data of human Chromosome 1 from UCSC table browser. Here we only consider repeats which have repeat length 2. We add 200 bp flanking sequence to each repeats. To do that we first locate the repeat in chromosome and then add 100 bp upstream and 100 bp downstream from the sequence of chromosome. Then we perform BLAST [10] against chimp genome. To filter the data we used very low E-value. After getting aligned chimp sequences (inside chromosome 1) we perform BLAST against human genome to get the sequences. After getting the sequences we find the repeats by using 'Tandem Repeat Finder(TRF)'. We discard the pair of microsatellites if it start before 40 base pairs and has more than 1 mutation interruption inside the repeats. Now we have the following data sets.

H65k15 We start with human repeats from UCSC table browser. Each repeat has more than or equal to **15** repeat units. To filter the alignment in BLAST we use E-value= 10^{-65}. After getting repeats from TRF if a sequence contain multiple repeats then we keep the repeats which started closer to 100bp. After meeting all the criteria we got 854 homologous loci of human and chimp.

H75k12 We start with human repeats from UCSC table browser. Each repeat has more than or equal to **12** repeat units. To filter the alignment in BLAST we use E-value= 10^{-75}. After getting repeats from TRF if a sequence contain multiple repeats then we keep the repeats which started closer to 100bp. After meeting all the criteria we got 1420 homologous loci of human and chimp. If we discard

the microsatellites having repeat length 12 from **H75k12**, then we get the dataset **H75k13**. It contains 1232 homologous loci.

H100k12 We start with human repeats from UCSC table browser. Each repeat has more than or equal to **12** repeat units. To filter the alignment in BLAST we use E-value= 10^{-100}. After getting repeats from TRF if a sequence contain multiple repeats then we discard the sequence. After meeting all the criteria we got 1010 homologous loci of human and chimp. If we discard the microsatellites having repeat length less than 15 from **H100k12**, then we get the dataset **H75k15**. It contains 672 homologous loci.

From the Table 6.1 it is clear that model EL2 is better than model EL1 (p-value is 0.0381969) and model PL2 is better than model PL1 (p-value is 0.02169683). However Model PL1 and PL2 is not significantly better than EL1 and EL2 repectively. Both EL2 and PL2 shows $m < 0.61$ which indicate multistep mutation has occurred. In all tables, the parameters that are fixed for a given sub-model are shown in italics.

From the Table 6.2 we can see that model EL2 and PL2 perform slightly better than model EL1 and PL2 respectively. However those are not statistically significant. Also Model PL1 and PL2 is not significantly better than EL1 and EL2 repectively. Both EL2 and PL2 shows m is very close to 1 which indicates very few multistep mutation has occurred.

Table 6.1: The parameters and maximum log-likelihood of the models for Data set **H65k15**

	MLEs of parameters					
Model	u	v	m	s	λ	$\log L$
EU1	*0.5*	*0*	*1*	*0*	9.477747	-4942.479
EU2	*0.5*	*0*	0.9662729	*0*	8.6001109	-4942.384
EC1	0.4554651	*0*	*1*	*0*	11.4673813	-4372.733
EC2	0.4551573	*0*	0.9999	*0*	11.4639361	-4372.733
EL1	0.54654453	0.01462743	*1*	*0*	12.82262596	**-4198.652**
EL2	0.5994591	0.0374068	0.5994591	*0*	3.608965	**-4197.082**
PU1	*0.5*	*0*	*1*	0.3148298	3.9154823	-4598.895
PU2	*0.5*	*0*	0.9999	0.3148122	3.9144643	-4598.904
PC1	0.454222123	*0*	*1*	-0.003928135	11.75537043	-4372.593
PC2	0.454214244	*0*	0.9999	-0.003927578	11.75178532	-4372.605
PL1	0.548346787	0.014650464	*1*	0.006972567	12.32528835	**-4198.628**
PL2	0.61604671	0.03746592	0.60649362	-0.02617842	4.3543008	**-4196.588**

Table 6.2: The parameters and maximum log-likelihood of the models for Data set **H75k12**

	MLEs of parameters					
Model	u	v	m	s	λ	$\log L$
EU1	*0.5*	*0*	*1*	*0*	10.24138	-8513.941
EU2	*0.5*	*0*	0.9999	*0*	10.23844	-8513.946
EC1	0.4634821	*0*	*1*	*0*	11.2357507	-7700.951
EC2	0.4634755	*0*	0.9999	*0*	11.2324682	-7700.971
EL1	0.54918232	0.01135839	*1*	*0*	12.37011564	**-7396.052**
EL2	0.5508148	0.01169873	0.98673212	*0*	11.903285	**-7396.042**
PU1	*0.5*	*0*	*1*	0.2333258	4.4847323	-8037.264
PU2	*0.5*	*0*	0.9999	0.2333138	4.4835513	-8037.28
PC1	0.462297161	*0*	*1*	-0.004801283	11.61683271	-7700.462
PC2	0.462290366	*0*	0.9999	-0.004801309	11.6133044	-7700.482
PL1	0.54631861	0.01135739	*1*	-0.01061397	13.29309011	**-7395.874**
PL2	0.54877562	0.01192654	0.97812494	-0.01111524	12.51892366	**-7395.848**

Table 6.3: The parameters and maximum log-likelihood of the models for Data set **H75k13**

	MLEs of parameters					
Model	u	v	m	s	λ	$\log L$
EU1	*0.5*	*0*	*1*	*0*	9.764473	-7292.017
EU2	*0.5*	*0*	0.947058	*0*	8.376018	-7291.713
EC1	0.4602567	*0*	*1*	*0*	10.9562861	-6532.692
EC2	0.4602496	*0*	0.9999	*0*	10.9530227	-6532.706
EL1	0.54364174	0.01177458	*1*	*0*	12.09168528	**-6295.519**
EL2	0.58467035	0.02144355	0.73253313	*0*	5.36193455	**-6291.607**
PU1	*0.5*	*0*	*1*	0.2813063	3.9891771	-6828.961
PU2	*0.5*	*0*	0.9999	0.2812887	3.9881584	-6828.972
PC1	0.459327823	*0*	*1*	-0.003821075	11.23854245	-6532.471
PC2	0.459320697	*0*	0.9999	-0.003820442	11.23518866	-6532.485
PL1	0.5829222	0.01404689	*1*	0.2075976	5.88534139	**-6295.181**
PL2	0.584803803	0.021425937	0.732908657	0.000486644	5.352619778	**-6291.607**

From the Table 6.3 we can see that model EL2 and PL2 perform significantly better than model EL1 and PL2 respectively (p-values are 0.0025 and 0.00375 respectively). Also Model PL1 is not significantly better than EL1. Both EL2 and PL2 shows $m < 0.74$ which indicates multistep mutations has occurred.

From the Table 6.4 we can see that model EL2 and PL2 perform slightly better than model EL1 and PL2 respectively. However those are not statistically significant. Also Model PL1 and PL2 is not significantly better than EL1 and EL2 repectively.

Table 6.4: The parameters and maximum log-likelihood of the models for Data set **H100k15**

	MLEs of parameters					
Model	u	v	m	s	λ	$\log L$
EU1	*0.5*	*0*	*1*	*0*	7.695878	-3856.103
EU2	*0.5*	*0*	0.9999	*0*	7.693698	-3856.104
EC1	0.455832	*0*	*1*	*0*	8.541308	-3426.291
EC2	0.4558243	*0*	0.9999	*0*	8.5388034	-3426.302
EL1	0.56120423	0.01692352	*1*	*0*	9.70166664	**-3268.856**
EL2	0.57542694	0.02048119	0.91344832	*0*	7.5523883	**-3268.642**
PU1	*0.5*	*0*	*1*	0.3035354	3.1098652	-3596.299
PU2	*0.5*	*0*	0.9999	0.3035196	3.1090622	-3596.307
PC1	0.455243502	*0*	*1*	-0.002470078	8.670865923	-3426.25
PC2	0.455235858	*0*	0.9999	-0.002469133	8.668248542	-3426.261
PL1	0.57840616	0.01765735	*1*	0.07143109	7.06699983	**-3268.504**
PL2	0.58670537	0.02022688	0.93067037	0.05061027	6.26142232	**-3268.384**

Both EL2 and PL2 shows m is quite close to 1 which indicates some multistep mutations has occurred. From these results we can see that the value of parameters m and s are different from [78]. So we focus our attention mainly on model PL2. This model contain parameters m, u, v, s and λ.

6.3.1 Analyzing the parameters for Model PL2

To work more on Model PL2 we are going to create some more dataset. We start with extracting tandem repeats with repeat number greater than 9 from Chromosome 1 of Human genome (found in [1]). Then we perform BLAST against chimp genome. To filter the data we used very low E-value. After getting aligned chimp sequences (inside chromosome 1) we perform BLAST against human genome to get the sequences. After getting the sequences we find the repeats by using ‘Tandem Repeat Finder(TRF)’. If a sequence contain more than one tandem repeats with repeat length greater than

9 then we discard the sequence. After getting the sequence which contain human and chimp repeats we find the pairs. We discard the pair of microsatellite if it start before 40 base pair and have more than 1 mutation interruption inside the repeats. The datasets are summarized in Table 6.5. The column name “MinLength” indicates the minimum repeat lengths of microsatellites in the data. Column “Evalue” contain the negative power of 10 of the e-value when performing BLAST operation. Column “N” represents the number of pairs of microsatellites of the dataset. We fit these data on Model PL2. The parameters are shown in Table A.1, A.2 and A.3.

Table 6.5: Summary of Datasets

DataSet	MinLength	Evalue	N
E65NN10	10	65	1820
E65NN11	11	65	1604
E65NN12	12	65	1390
E65NN13	13	65	1190
E65NN14	14	65	1013
E65NN15	15	65	860
E75NN10	10	75	1805
E75NN11	11	75	1591
E75NN12	12	75	1378
E75NN13	13	75	1180
E75NN14	14	75	1003
E75NN15	15	75	852
E100NN10	10	100	1240
E100NN11	11	100	1162
E100NN12	12	100	1066
E100NN13	13	100	968
E100NN14	14	100	855
E100NN15	15	100	739

Figure 6.2, 6.3 and 6.4 show the MLEs of parameters m, s and u for different dataset. It can be seen that u does not change much for datasets (0.55 to 0.62) where the reported value of u is 0.82 in [78]. The parameters are largely same for the

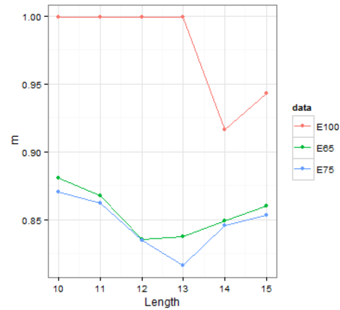

Figure 6.2: Comparision of parameter m for different datasets of Table 6.5

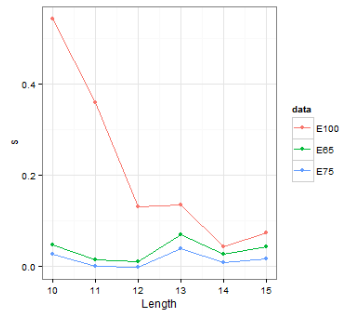

Figure 6.3: Comparision of parameter s for different datasets of Table 6.5

datasets with evalue 10^{-65} and 10^{-75}. In both case s is close to 0 and m is between 0.8 to 0.9. Where the reported value of m is 0.55 in [78]. However for the datasets with evalue 10^{-100}, there are some cases where m reaches to approximately 1 which in turn results higher value of s. But it is still lower than corresponding $s = 0.76$ reported in [78].

Outliers and Related Results

We perform the analysis on first 12 Datasets from Table 6.5. At first I find the absolute value of the difference of the length of homologous loci of human and chimp microsatellite. Then I find the mild upper threshold of the data by using 2nd and

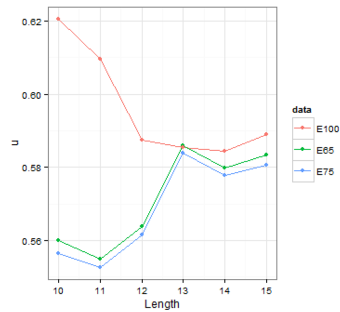

Figure 6.4: Comparision of parameter u for different datasets of Table 6.5

4th quantiles. And finally if the difference of a pair is greater than the mild upper threshold then I discard the pair. Table A.4 and A.5 contains the parameters for the data. From Figure 6.5 and 6.6 it can be seen that after discarding outliers, the MLE of parameters for datasets follow closely to each other.

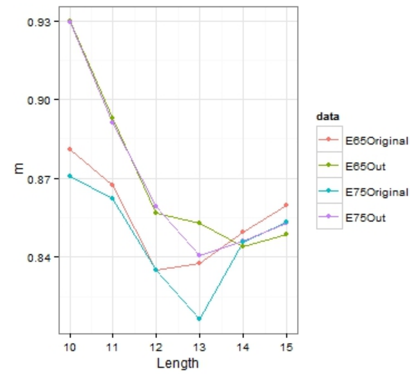

Figure 6.5: Comparision of parameter m for data with outliers and data without outliers of different datasets of Table 6.5

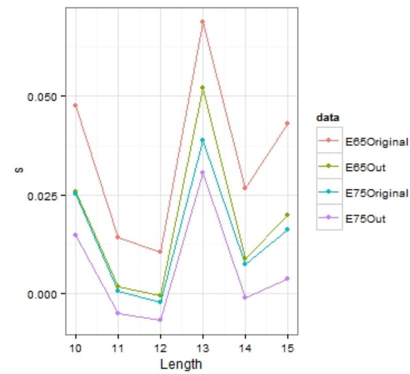

Figure 6.6: Comparision of parameter s for data with outliers and data without outliers of different datasets of Table 6.5

Bootstrap Results

For bootstrapping I re-sampled the data with replacement. For each dataset I have done the re-sampling 1000 times. The bootstrap average of the parameters and the MLEs of the parameters are shown in Table A.6 and A.7. The rows BootMean contains the mean of the parameters of 1000 iteration of the dataset indicated by previous rows. Table A.8 and A.9 contain the bootstrap variance of the parameters for various data. From Figure 6.7 and 6.8, it can be seen that the m and λ have higher variance compare to other parameters. But overall the variances of the parameters are quite low.

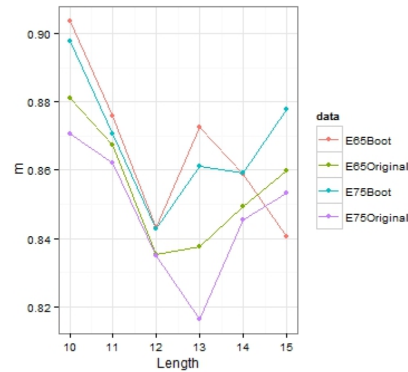

Figure 6.7: Comparision of parameter m for original data and bootstrap data of different datasets of Table 6.5

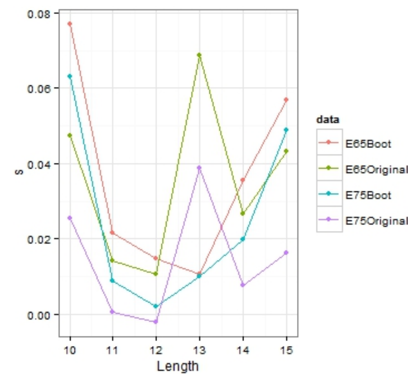

Figure 6.8: Comparision of parameter s for original data and bootstrap data of different datasets of Table 6.5

6.4 Inter Species Comparison for Microsatellite with Repeat Length 2

6.4.1 Data Contain 10 Population (Single alleles)

The data **Pop10** contains microsatellite data of 10 samples from different populations from East Asia (2 pop), AFrica (3 pop), Europe (3 pop) and America (2 pop). Table 6.6 shows the different parameters for different models for this data. For this data we used star tree with 10 tips, each represent a population. We assume that each branch has same branch length. The tree is depicted in Figure 6.9. In Figure 6.9 EA represents East asian populations, E represents European populations, AM represents American populations and A represents African populations.

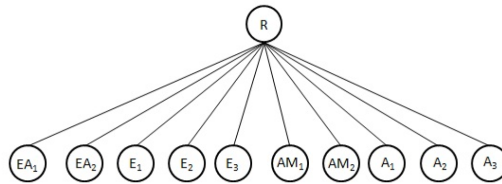

Figure 6.9: Tree for the Pop10 data

In Table 6.7, the bold number indicates the value of the parameter reached the lower limit. For parameter u it was set to 0.500001. For all microsatellites, only AC microsatellites and only AG microsatellites all the two-phase models perform significantly better than the corresponding one-phase models. Among the two phase models, PL2 is significantly better than PC2 and EL2, whereas EL2 is significantly better than EC2. The value of m for three different data is between 0.6 to 0.65 which indicates a fair number of multistep mutations has occurred. But it is still less than

Table 6.6: Parameters for Data Pop10

Model	Data	$u \times 10^{-1}$	$v \times 10^{-2}$	$m \times 10^{-1}$	$s \times 10^{-1}$	$\lambda \times 10^{-2}$	L
EC1	All	4.2451	0	10	0	24.8428	-43388.254
	AC	4.2802	0	10	0	28.7908	-37620.507
	AG	4.0876	0	10	0	10.0300	-4803.941
EC2	All	3.6795	0	6.5041	0	15.7006	-38464.893
	AC	3.7305	0	6.4882	0	17.9459	-33426.384
	AG	3.5474	0	6.9901	0	7.0907	-4350.493
EL1	All	**5.0000**	1.4269	10	0	24.4194	-43456.417
	AC	**5.0000**	1.3281	10	0	28.3670	-37629.709
	AG	**5.0000**	2.0372	10	0	9.7660	-4832.029
EL2	All	**5.0000**	3.0042	6.1901	0	15.1327	-38147.757
	AC	**5.0000**	2.8327	6.1975	0	17.3568	-33118.455
	AG	**5.0000**	4.0408	6.3171	0	6.7384	-4324.458
PC1	All	4.5593	0	10	3.1043	12.7278	-42620.217
	AC	4.5820	0	10	3.0134	14.7249	-36963.054
	AG	4.2791	0	10	1.1860	7.5807	-4786.059
PC2	All	4.0076	0	6.4907	1.2211	11.294	-38257.028
	AC	4.0468	0	6.4823	1.1720	12.9372	-33250.490
	AG	3.6552	0	6.9614	0.3297	6.4857	-4348.324
PL1	All	5.3831	1.3660	10	5.5143	9.2748	-42338.715
	AC	5.4489	1.4118	10	5.4372	10.6401	-36697.996
	AG	**5.0000**	1.3155	10	2.6023	5.8354	-4776.746
PL2	All	5.5877	3.1653	6.1825	2.2737	8.8787	-37816.045
	AC	5.7055	3.2493	6.1934	2.1726	10.1998	-32845.549
	AG	**5.0000**	3.3449	6.3326	0.9334	5.4245	-4313.573

the value of $m = 0.55$ for model PL2 reported in [78]. The value of parameters for all microsatellites and AC microsatellites is quite similar for all models. However for AG microsatellites λ is quite small compare to AC microsatellites and all microsatellites (for all models).

6.4.2 Data Contain 10 Population (2 alleles)

The data **Pop20** contains microsatellite data of 10 samples from 10 different populations from East Asia (2 populations), Africa (3 populations), Europe (3 populations) and America (2 populations). For this data we used same tree topology as in Pop10. However this tree contains 20 tips. Each sample's two allele is represented by two tips of the tree. Table 6.7 shows the different parameters for different models for this data.

In Table 6.7, the bold number indicates the value of the parameter reached the lower limit. For parameter u it was set to 0.500001. For all microsatellites, only AC microsatellites and only AG microsatellites all the two-phase models perform significantly better than the corresponding one-phase models. Among the two phase models, PL2 is significantly better than PC2 and EL2, whereas EL2 is significantly better than EC2. The value of m for three different data is between 0.57 to 0.61 which indicates a fair number of multistep mutations has occurred. But it is still less than the value of $m = 0.55$ for model PL2 reported in [78]. The value of parameters for all microsatellites and AC microsatellites is quite similar for all models. However for AG microsatellites λ is quite small compare to AC microsatellites and all microsatellites (for all models).

Table 6.7: Parameters for Data Pop20

Model	Data	$u \times 10^{-1}$	$v \times 10^{-2}$	$m \times 10^{-1}$	$s \times 10^{-1}$	$\lambda \times 10^{-2}$	L
EC1	All	4.2796	0	10	0	27.2918	-81824.127
	AC	4.3159	0	10	0	31.8481	-71751.320
	AG	4.1226	0	10	0	10.2496	-8087.904
EC2	All	3.6659	0	5.9776	0	15.9102	-68807.519
	AC	3.7211	0	5.9646	0	18.2657	-60602.042
	AG	3.5038	0	6.4414	0	6.7105	-6899.520
EL1	All	**5.0000**	1.1632	10	0	26.8606	-82137.439
	AC	**5.0000**	1.0719	10	0	31.4199	-71958.046
	AG	**5.0000**	1.8080	10	0	10.0163	-8133.889
EL2	All	**5.0000**	2.6716	5.8133	0	15.3836	-68467.206
	AC	**5.0000**	2.5082	5.8076	0	17.7274	-60270.911
	AG	**5.0000**	3.8605	6.0859	0	6.3874	-6850.787
PC1	All	4.6832	0	10	6.1577	9.4413	-78917.721
	AC	4.7049	0	10	6.1250	10.8065	-69216.413
	AG	4.3996	0	10	2.1744	6.4571	-8012.766
PC2	All	4.1439	0	5.9597	2.1401	9.3967	-68032.206
	AC	4.1817	0	5.9584	2.0589	10.8005	-59939.565
	AG	3.6974	0	6.3864	0.5982	5.7343	-6889.132
PL1	All	5.4332	1.1331	10	9.3499	7.0898	-78569.796
	AC	5.4810	1.1419	10	9.4033	8.0443	-68895.666
	AG	**5.0000**	1.0888	10	3.3626	5.3510	-7998.107
PL2	All	5.9049	3.3311	5.7800	3.4768	7.3474	-67422.572
	AC	5.9869	3.3488	5.7867	3.2983	8.5181	-59396.613
	AG	**5.0000**	3.1980	6.0476	0.9890	5.0683	-6830.233

6.4.3 Data Contain 5 Population (Single alleles)

The data **Pop5** contain 5 samples from 5 different populations from Africa (3 populations) and America (2 populations). Figure 6.10 depicted the tree topology used for this data. Table 6.8 and 6.9 shows the different parameters for different models for this data.

In Table 6.8, the bold number in column u indicates the value reached the lower

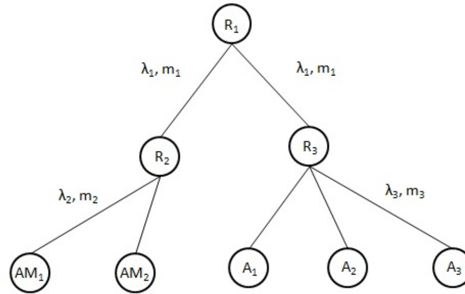

Figure 6.10: Tree for the Pop5 data

limit 0.500001. And in column m_1 the bold number indicates the value reached the upper limit 0.9999. The value of m_1 for all Microsatellites and AC microsatellites is quite high for all the models compare to m_2 and m_3. In model PL2 the values are more than 0.8. But for AG microsatellites the m_2 is higher than m_1 and m_3. This suggests for AC microsatellites and all Microsatellites before the divergence of African and American population from their respective common ancestors there were few multistep mutations occurred. From that point more multistep mutations occurred. If we consider only AG microsatellites then we can see that fewer multistep mutations occurred after the divergence of American populations compare to other branches.

Table 6.8: Parameters u, v and ms for Data Pop5

Model	Data	$u \times 10^{-1}$	$v \times 10^{-2}$	$m_1 \times 10^{-1}$	$m_2 \times 10^{-1}$	$m_3 \times 10^{-1}$
EC1	All	4.2350	0	10	10	10
	AC	4.2669	0	10	10	10
	AG	4.0952	0	10	10	10
EC2	All	4.2165	0	9.9050	6.0629	5.9202
	AC	4.2598	0	**9.9990**	5.9745	5.9007
	AG	4.0227	0	9.5249	8.3786	6.6283
EL1	All	**5.0000**	1.5045	10	10	10
	AC	**5.0000**	1.4078	10	10	10
	AG	**5.0000**	2.0593	10	10	10
EL2	All	**5.0000**	2.3639	7.7521	6.2311	5.9282
	AC	**5.0000**	2.0933	8.0803	6.1317	5.9068
	AG	**5.0000**	4.3083	6.1076	8.4059	6.7072
PC1	All	4.5009	0	10	10	10
	AC	4.5244	0	10	10	10
	AG	4.2192	0	10	10	10
PC2	All	4.3772	0	9.9237	6.0962	5.9680
	AC	4.4054	0	**9.9990**	6.0080	5.9509
	AG	4.0919	0	9.5345	8.3770	6.6311
PL1	All	5.4359	1.5172	10	10	10
	AC	5.5059	1.5710	10	10	10
	AG	**5.0000**	1.3212	10	10	10
PL2	All	5.3361	2.0261	8.2488	6.2782	6.0150
	AC	5.4032	2.0645	8.2898	6.1839	6.0012
	AG	**5.0000**	2.7784	7.0477	8.4004	6.7036

From Table 6.9 we can see that for all microsatellites, only AC microsatellites and only AG microsatellites all the two-phase models give much higher maximum likelihood corresponding one-phase models. For example the difference between log likelihood of PL1 and PL2 is more than 2000 for all Microsatellites. Among the two phase models, PL2 is significantly better than PC2 and EL2, whereas EL2 is significantly better than EC2. For all models and different kind of microsatellites λ_2 and λ_3 similiar to each other. However λ_1 is smaller than than other two branches.

Table 6.9: Parameters s and λs for Data Pop5

Model	Data	$s \times 10^{-2}$	$\lambda_1 \times 10^{-3}$	$\lambda_2 \times 10^{-2}$	$\lambda_3 \times 10^{-2}$	L
EC1	All	0	31.3235	17.8960	24.3754	-26475.780
	AC	0	34.4947	21.3051	28.5401	-22783.701
	AG	0	16.3487	5.6244	9.1726	-3112.932
EC2	All	0	21.5973	11.5299	15.0181	-24027.746
	AC	0	24.5881	13.3035	17.2246	-20627.996
	AG	0	9.5962	4.8619	6.6990	-2966.000
EL1	All	0	30.8617	17.5567	23.9505	-26469.276
	AC	0	33.9935	20.9491	28.1176	-22737.526
	AG	0	16.4500	5.4733	8.8959	-3128.999
EL2	All	0	22.2908	11.0494	14.6281	-23904.679
	AC	0	25.4732	12.7586	16.8319	-20497.214
	AG	0	11.4098	4.7515	6.2673	-2956.451
PC1	All	21.9373	18.7701	10.7206	14.5935	-26170.462
	AC	21.4002	20.6374	12.7240	17.0358	-22517.874
	AG	6.6214	13.8050	4.7675	7.7713	-3108.913
PC2	All	9.3956	16.7551	8.9271	11.6413	-23954.903
	AC	8.6566	19.3334	10.4022	13.4929	-20568.959
	AG	2.9169	8.8892	4.5010	6.2020	-2965.090
PL1	All	53.8163	11.8910	6.8157	9.3246	-25902.184
	AC	53.7357	12.8856	7.9834	10.7539	-22265.025
	AG	25.4533	9.6592	3.3085	5.3879	-3097.621
PL2	All	23.1887	13.1778	6.4897	8.6169	-23738.055
	AC	21.6281	15.2442	7.5843	10.0384	-20368.390
	AG	11.9339	8.1927	3.6100	4.8300	-2948.462

Also the branch lengths are smaller for AG microsatellites compare to others. This indicates smaller mutation rates for AG microsatellites compare to AC microsatellites.

6.4.4 Data Contain 8 Population

The data **Pop8** contains 8 samples from 8 different populations from East Asia (2 populations), Africa (3 populations) and Europe (3 populations). Table 6.10, 6.11 and 6.12 show the different parameters for different models for this data. Figure 6.11

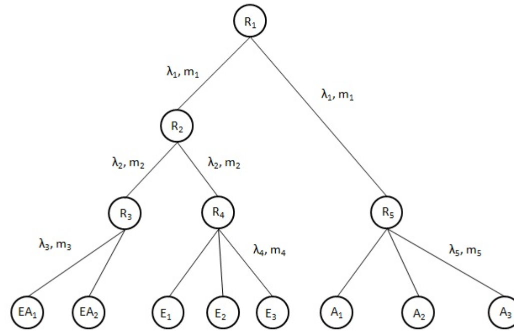

Figure 6.11: Tree for the Pop8 data

depicted the tree topology used for this data. The parameters m and λ adjacent to a branch indicates that parameters are working on that particular branch.

In Table 6.10, the value of m_1 for all Microsatellites and AC microsatellites is quite high for all the models compare to m_2, m_3, m_4 and m_5. In model PL2 the values are more than around 0.77. But for AG microsatellites the m_5 is higher than m_1 for EL2 and PL2. This suggests for AC microsatellites and all Microsatellites before the divergence of African, East Asian and European population from their respective common ancestors there were few multistep mutations occurred. From that point more multistep mutations occurred. For all Microsatellites and AC microsatellites the rate for multistep mutation is quite similar for all populations, but for AG microsatellites the rate is higher for African populations and lower in East Asian populations. Among the models, PL2 is significantly better than PC2 and EL2, whereas EL2 is significantly better than EC2.

From Table 6.11, it can be seen that for all models and different kind of microsatellites λ_1 and λ_2 are similar, whereas λ_3, λ_4 and λ_5 have similar values. However smaller values of λ_1 and λ_2 is indicates the populations diverged quickly from their common ancestors. Also all the branch lengths are smaller for AG microsatellites compare

Table 6.10: Parameter ms for all, AC and AG Microsatellites for Pop8

Model	Data	$m_1 \times 10^{-1}$	$m_2 \times 10^{-1}$	$m_3 \times 10^{-1}$	$m_4 \times 10^{-1}$	$m_5 \times 10^{-1}$	L
EC2	All	9.6119	5.3982	6.0685	6.1117	5.9345	-32449.483
EC2	AC	9.7442	5.3302	6.1695	6.1441	5.9090	-28046.264
EC2	AG	9.2200	6.0380	5.1562	6.2547	6.6626	-3826.452
EL2	All	7.4009	5.6531	6.0387	6.1165	5.9762	-32315.497
EL2	AC	7.6766	5.6099	6.1405	6.1475	5.9476	-27900.696
EL2	AG	6.1173	6.1323	5.1632	6.2828	6.7145	-3818.698
PC2	All	9.7360	5.3594	6.1133	6.1500	5.9810	-32317.000
PC2	AC	9.8188	5.3154	6.2140	6.1875	5.9596	-27937.353
PC2	AG	9.2317	6.0288	5.1680	6.2559	6.6699	-3824.592
PL2	All	7.7067	5.5905	6.1123	6.1754	6.0444	-32076.454
PL2	AC	7.7329	5.5629	6.2117	6.2120	6.0262	-27711.663
PL2	AG	6.5593	6.1011	5.1869	6.2685	6.7211	-3807.009

to AC microsatellites. This indicates smaller mutation rates for AG microsatellites compare to AC microsatellites. From Table 6.12 we can see that, for all microsatellites, only AC microsatellites and only AG microsatellites all the two-phase models give much higher maximum likelihood corresponding one-phase models. For example the difference between log likelihood of PL1 and PL2 is more than 3000 for all Microsatellites.

From above results we can see that two phase models always gives higher maximum likelihood values compare to one phase models for different kind of datasets. For more complex trees it can be seen that multistep mutations occurred more after the populations diverged from their respective common ancestors and have larger branch lengths. Whereas initially fewer multistep mutations occurs and branch lengths are also small. Also, AG microsatellites have less mutation rates compare to AC mutation rates (branch lengths are smaller).

Table 6.11: Parameter λs for all, AC and AG Microsatellites for Pop8

Model	Data	$\lambda_1 \times 10^{-2}$	$\lambda_2 \times 10^{-2}$	$\lambda_3 \times 10^{-2}$	$\lambda_4 \times 10^{-2}$	$\lambda_5 \times 10^{-2}$
EC1	All	3.4483	4.4065	17.8353	19.9125	24.6296
	AC	4.0584	5.1071	20.4956	22.6012	28.7325
	AG	1.1315	1.5490	7.8894	9.4407	9.5962
EC2	All	2.3497	2.8452	11.1064	12.4196	15.2367
	AC	2.7372	3.3112	12.7367	13.9558	17.4397
	AG	0.8433	1.0045	4.7082	6.4714	6.9800
EL1	All	3.4091	4.3724	17.5071	19.5419	24.1963
	AC	4.0073	5.0788	20.1673	22.2362	28.3085
	AG	1.1613	1.5225	7.6643	9.1908	9.2920
EL2	All	2.4110	2.6739	10.8896	12.1431	14.8500
	AC	2.8141	3.1051	12.5270	13.6777	17.0473
	AG	0.8800	0.9693	4.5600	6.3158	6.6951
PC1	All	1.8318	2.3625	9.6250	10.7245	13.3268
	AC	2.1498	2.7420	11.0546	12.1590	15.5536
	AG	0.8458	1.1593	5.9022	7.0692	7.1782
PC2	All	1.7138	2.0945	8.1606	9.1265	11.2298
	AC	2.0281	2.4471	9.4368	10.3412	12.9584
	AG	0.7662	0.9131	4.2828	5.8857	6.3479
PL1	All	1.2742	1.6575	6.7411	7.5089	9.3880
	AC	1.4760	1.9082	7.6706	8.4315	10.8730
	AG	0.6479	0.8705	4.4193	5.3062	5.3730
PL2	All	1.4201	1.5968	6.5100	7.2528	8.9075
	AC	1.6720	1.8641	7.5359	8.2176	10.2915
	AG	0.6641	0.7553	3.5586	4.9046	5.2204

6.4.5 Bootstrap Analysis

We have performed bootstrapping for all the different tree (except 5 pop simple tree). We performed bootstrapping on Models EC1, EC2, EL1, EL2, PC1, PC2, PL1 and PL2. Table A.10, A.11, A.12 and A.13 contains the result. The column "data" indicates the Microsatellies under consideration (ALL microsatellites, only AC, only AG). The results are summarized below.

Table 6.12: Rest of the Parameters for Pop8

Model	Data	$u \times 10^{-1}$	$v \times 10^{-2}$	$s \times 10^{-1}$	L
EC1	All	4.2387	0	0	-36581.518
	AC	4.2723	0	0	-31565.499
	AG	4.0889	0	0	-4239.735
EC2	All	4.1811	0	0	-32449.483
	AC	4.2332	0	0	-28046.264
	AG	3.9637	0	0	-3826.452
EL1	All	**5.0001**	1.4587	0	-36625.702
	AC	**5.0001**	1.3610	0	-31556.627
	AG	**5.0001**	2.0380	0	-4265.463
EL2	All	**5.0001**	2.4988	0	-32315.497
	AC	**5.0001**	2.2325	0	-27900.696
	AG	**5.0001**	4.2893	0	-3818.698
PC1	All	4.5392	0	2.7828	-36012.623
	AC	4.5614	0	2.6933	-31079.180
	AG	4.2869	0	1.2378	-4223.289
PC2	All	4.3842	0	1.1544	-32317.000
	AC	4.4154	0	1.0801	-27937.353
	AG	4.0516	0	0.3609	-3824.592
PL1	All	5.3859	1.3939	5.3936	-35741.971
	AC	5.4505	1.4403	5.3129	-30825.079
	AG	**5.0001**	1.2747	2.8458	-4211.087
PL2	All	5.3695	2.2682	2.2016	-32076.454
	AC	5.4599	2.3352	2.0803	-27711.663
	AG	**5.0001**	3.1464	1.0840	-3807.009

- **Pop10:** From Table A.10 we can see that, the bootstrap variance is quite low for all the parameters. The variance for u, v and λ is lower than m and s. Also the variance is almost always higher for AG microsatellites for all parameters in all models. However for model PL1 and PL2 u is lower for AG microsatellites.

- **Pop20:** From Table A.10 we can see that, the bootstrap variance is quite similar to the results for data **Pop10**. The variance is little bit higher than previous data.

- **Pop5:** Here in Table A.12, we can see that the variance for m_1 is higher than m_2 and m_3 for model EL2 and PL2 and lower for $EL1$ and $PL1$. And variance of λ_1 is lower than λ_2 and λ_3 (the value of λ_1 was lower than the other λs).

- **Pop8:** In Table A.13 we can see that, the variance for m_2 is highest among other ms, except for model PL2 where variance for m_1 is higher. In most of the cases the variance for AG microsatellites is higher than other two data.

So we can say that the variance for all the parameters are quite low for AC, AG and All microsatellites. The variances for AG microsatellites are higher than other two data in most of the cases.

6.5 Microsatellites with Repeat Length 3

To find the homologous loci in the pair of human and chimp, at first we obtain the repeat data of human Chromosome 1 from UCSC table browser. Here we only consider repeats which has repeat length 3 and at least repeated 3 times. We add 200 bp flanking sequence (100 bp upstream and 100 bp downstream) to each repeats. Then we perform BLAST against chimp genome. To filter the data we used very low E-value. After getting aligned chimp sequences (inside chromosome 1) we perform BLAST against human genome to get the sequences. After getting the sequences we find the repeats by using 'Tandem Repeat Finder(TRF)'. We discard the pair of microsatellite if it start before 40 base pair and have more than 1 mutation interruption inside the repeats. After finding the dataset we find the repeats which lied in the exon region of Human genome. So in the end we have two dataset, one containing all the repeats (**AllRepeat** henceforth) and repeats inside the exons (**ExonRepeat**

Table 6.13: Parameters for Data AllRepeat

Model	u	v	m	s	λ	$-\log L$
PU1	0.5	0	1	38.2236	0.003813	20476.2
PU2	0.5	0	0.6271	30.0097	0.002739	20144.37
PC1	0.47884	0	1	30.0113	0.004859	19985.59
PC2	0.47582	0	0.7754	30.0102	0.003099	19828.36
PL1	0.66999	0.03844	1	70.0099	0.002219	19212
PL2	0.99999	0.11844	0.5759	96.0099	0.000862	18785.54

henceforth). The data **AllRepeat** contains 29216 pairs of microsatellites and **Exon-Repeat** contains 1814 pairs of microsatellites.

With these data on hand we start testing on the models. We find that the test results in high value for parameter s and the models which have $s = 0$ results in very small likelihood. So we focus our attention on Models which involve parameter s (PU1, PU2, PC1, PC2, PL1 and PL2). From the Table we can see that model PL2 outperform rest of the models for both dataset. Also it is interesting to note that value of λ for data **AllRepeat** is about four times higher than value of λ for data **Exon-Repeat**. Which indicate that mutation rate of microsatellites inside exon is smaller than microsatellites outside the exons. Very high value of s for both dataset indicates the mutation rate strongly increases with the increase in length of microsatellites. The value of m is quite lower than 1 for both dataset which indicates multistep mutations. Also higher m for microsatellites inside the exons indicate lower multistep mutations than microsateliites outside the exons. The value of u and v assures that, microsatellites with repeat length more than 5 has higher probablity of contraction towards repeat length 4.

For the analysis of variance we only work on Model PL2. For **AllRepeat**, we randomly divided the data into 25 smaller groups. 24 of them has 1169 pairs each,

while the other one has 1160 pairs. The results are summarized in Table A.14. For **ExonRepeat**, we randomly divided the data into 20 smaller groups. 19 of them has 91 pairs each, while the other one has 85 pairs. The results are summarized in Table A.15. Table A.16 contains the Mean and Standard Deviations (SD) of data from Table A.14 and A.15. The Mean values for the parameters follow quite well to the parameters of original data for Model PL2. The standard deviation is also quite low for the data generated from **AllRepeat**. But since each group has considerably lower amount of data for **ExonRepeat** standard deviations are higher.

Table 6.14: Parameters for Data ExonRepeat

Model	u	v	m	s	λ	$-\log L$
PU1	0.5	0	1	64.6873	0.00126	957.0287
PU2	0.5	0	0.9199	60.1858	0.00119	952.1032
PC1	0.4886	0	1	65.0063	0.00126	949.5265
PC2	0.4856	0	0.9273	60.2740	0.00121	944.7132
PL1	0.6563	0.0416	1	66.0265	0.00128	883.865
PL2	0.9999	0.1280	0.7273	176.706	0.00029	863.9047

6.6 Conclusion

Microsatellites are used in a wide range of studies due to their small size and repetitive nature, and they have played an important role in the identification of numerous important genetic loci. For previously neglected species, novel technologies have enabled the development of markers through the generation of new sequences and a more refined search in databases. Microsatellites are also very suitable for analyzing forensic specimens. As microsatelllite repeat instability leads to various diseases, understanding the pathogenic mechanisms will give better chance to find treatments for those diseases. In multiple species, microsatellites also have been shown to modulate levels

of gene expression as they expand and contract. Perhaps for their ability to modulate gene expression, some microsatellites have been conserved in vertebrates over long evolutionary time periods, and many promoter microsatellites are conserved in mammals. Investigating other conserved promoter microsatellites will help determine whether microsatellites can be conserved as sources of variation in gene expression. From our Analysis we can make following remarks:

- For microsatellites with repeat length 2 (for Human and Chimp comparison), two-phase model performs better than one phase models, which indicates multistep mutations. The bootstrap variance is quite low for all the parameters.

- When comparing among human populations, we can see that AG microsatellites have slower mutation rate compare to AC microsatellites. The bootstrap variances for parameters for AG microsatellites also higher than that of AC microsatellites, although overall the variances are quite low. According to the best Model (PL2), probablity of multistep mutations is quite high, especially in the early stage of divergence.

- For microsatellites with repeat length 3, Microsatellites inside the exons have lower mutation rates than those outside exons. Longer microsatellites tend to mutates more often due to high value of s. The probability of having multistep mutations is quite high, especially for microsatellites outside the exons.

Chapter 7

Summary and Future Work

In this thesis we investigate mathematical and algorithmic aspects of regularities in strings, with emphasis on indeterminate strings, and study microsatellite evolution.

In the first part of this thesis we have introduced a new algorithm and data structures to compute the minimum enhanced cover array from the prefix table. Computing the minimum enhanced cover array from the prefix table rather than from a variant of the border array allows us to extend the computation to indeterminate strings. We have provided a proof of the algorithm's correctness. We also provided an analysis of its complexity, both worst and average case. We have extended the basic algorithm to enhanced left covers and enhanced left seeds. We have discussed the practical application of our algorithms, in terms of time and space requirements. After comparing our prefix-based implementation with the border-based implementation of [45] we found that our algorithm is faster in practice and more space-efficient than those of [45]. Our algorithms also allowed us to easily extend the computation of enhanced covers to indeterminate strings. Both for regular and indeterminate strings, our algorithms have executed in expected linear time. Along the way we have

included an important theoretical result: that the expected maximum length of any border of any prefix of a regular string $\boldsymbol{x}$ is approximately 1.64 for binary alphabets, less for larger ones. We also showed how to extend the various enhanced cover array algorithms to indeteterminate strings.

We have outlined three algorithms to compute the Lyndon array for which no clear exposition is available in the literature. Two of them require $O(n^2)$ time in the worst case. The first one is very fast and apparently linear in practice. The second one is superlinear in practice and runs in $O(n \log n)$ time in the average case on binary strings. The third algorithm is simple and worst-case linear-time, but requires suffix array construction and so is a little slower. Then we have two new approaches to find the Lyndon array of a string. We have used only elementary data structures (no suffix arrays) for both of the approaches. The first approach has two variants, one variant is $O(n^2)$ in the worst case, the other guarantees $O(n \log n)$ time, but with no clear advantage in processing time. We have processed the string from left to right and use a "stack" data structure in the first approach. In the second approach we have processed the string from right to left to compute the Lyndon array. Finally after implementing these algorithms, we have showed that all of them run in $\Theta(n)$ time in practice.

In the third part of the thesis, we have presented necessary and sufficient condition that a given integer array is a Lyndon array of some string on some alphabet. We then described a linear-time algorithm to evaluate these conditions for a given array. We presented an $O(\sigma n)$-time algorithm to compute the unique string determined by the Lyndon arrays computed for σ rotations of the alphabet, where σ is the size of the alphabet. We also briefly discussed the possibility of using fewer than σ rotations

to determine $\boldsymbol{x}$.

In the final part of the thesis we work on the group of models presented in [78] based on three sets of contrasting features in the existing models of microsatellite evolution. We implemented these models with and fit different datasets acquired from Human and Chimp genomes. We find the dynamics of the evolution of microsatellite with repeat length 2 by using Human and Chimp data. We present our work on intra-species analysis where we consider different Human populations. We also find the differences between mutation based on the position (inside or outside exon regions) of microsatellites in the genome. We work on Human and Chimp DNA for inter species comparison and different human populations for intra species comparison. We compared the models using statistical methods and find the model which fits the data best.

7.1 Future Work

There are several open problems and research directions worth considering by future researchers.

1. It is worthwhile to design POS/LEN (compressed prefix array) version of Compute_MEC. Another natural question of course is to investigate whether the MEC array can be computed in linear time.

2. We know of no efficient algorithm to compute a string on a minimum alphabet consistent with a given valid Lyndon array $\mathcal{L}^*$. Finding such algorithm certainly requires some attention.

3. It would be of interest to determine minimal criteria for the reconstruction

of $\boldsymbol{x}$ — the least number of rotations or the size of the smallest alphabet. Reconstructing a string based on fewer than σ permutations of the alphabet is still considered an open problem.

4. The study of UMFFs has led to the idea of V-words, analogous structures to Lyndon words, derived from a global non-lexicographic ordering of strings called V-order. Also, linear-time algorithms for computing Lyndon border and Lyndon suffix arrays have recently been proposed [6]. There is scope to extend the results of this chapter to the UMFF based on V-order [32], then further to UMFFs in their full generality.

5. There is still a lot to do in the field of microsatellite evolution. For inter species analysis we have only worked on the data of chromosome 1 (for both human and chimp). A natural extension of this work is to consider the data obtained from the whole genome of the species. Considering more than two species of mammal leads to another direction of research.

Appendix A

Additional Results for Microsatellite Evolution

Table A.1: Parameters of model PL2 for Human to Chimp data with e-value 10^{-65}

	MLEs of parameters					
Data	u	v	m	s	λ	$\log L$
E65NN10	0.56	0.01	0.88098	0.0475	6.26	-9931.4
E65NN11	0.555	0.011	0.86733	0.01427	7.507	-8641.133
E65NN12	0.5638	0.0134	0.83523	0.01056	7.1016	-7385.217
E65NN13	0.586	0.0157	0.8375	0.0688	5.2355	-6220.207
E65NN14	0.5799	0.0171	0.8494	0.0266	6.95	-5207.949
E65NN15	0.5833	0.0186	0.8599	0.04313	6.83	-4336.723

Table A.2: Parameters of model PL2 for Human to Chimp data with e-value 10^{-75}

	MLEs of parameters					
Data	u	v	m	s	λ	$\log L$
E75NN10	0.5564	0.01	0.8706	0.0253	6.704	-9822.613
E75NN11	0.5527	0.0114	0.8621	0.0006	7.9	-8544.562
E75NN12	0.5616	0.0138	0.8349	-0.0021	7.484	-7294.744
E75NN13	0.5838	0.0165	0.8164	0.0389	5.442	-6142.476
E75NN14	0.5778	0.0179	0.8455	0.0075	7.333	-5130.689
E75NN15	0.5807	0.0196	0.8532	0.0161	7.337	-4271.771

Table A.3: Parameters of model PL2 for Human to Chimp data with e-value 10^{-100}

	MLEs of parameters					
Data	u	v	m	s	λ	$\log L$
E100NN10	0.6205	0.013	0.999	0.5426	1.444	-6605.075
E100NN11	0.6097	0.013	0.999	0.3589	2.257	-6136.965
E100NN12	0.5874	0.0125	0.999	0.1307	4.4895	-5573.632
E100NN13	0.5855	0.0134	0.999	0.1344	4.8352	-4986.754
E100NN14	0.5844	0.0166	0.916	0.0419	5.773	-4331.986
E100NN15	0.589	0.0181	0.9428	0.0722	5.663	-3668.606

Table A.4: The parameters for the data without outliers

Data	u	v	m	s	λ	$\log L$	N
HE65k10	0.5492	0.0089	0.9298	0.0258	7.7651	-9843.563	1811
HE65k11	0.5492	0.0106	0.8929	0.0018	8.4841	-8580.957	1598
HE65k12	0.5589	0.0131	0.8567	-0.0004	7.8459	-7334.141	1385
HE65k13	0.581	0.0154	0.8528	0.0522	5.6974	-6178.11	1186
HE65k14	0.5774	0.0177	0.844	0.0087	7.4095	-5182.319	1011
HE65k15	0.5819	0.0196	0.8485	0.0198	7.2291	-4311.096	858

Table A.5: The parameters for the data without outliers

HE75k10	0.5468	0.0089	0.9294	0.0147	8.1465	-9747.141	1797
HE75k11	0.5481	0.0108	0.8911	-0.0049	8.686	-8497.448	1586
HE75k12	0.5576	0.0132	0.8593	-0.0067	8.0722	-7256.979	1374
HE75k13	0.5779	0.0158	0.8406	0.0305	5.9437	-6113.571	1177
HE75k14	0.5759	0.018	0.8461	-0.001	7.6665	-5118.914	1002
HE75k15	0.5779	0.0198	0.8531	0.0037	7.7665	-4260.042	851

Table A.6: Bootstrap Mean of parameters with original MLEs

Data	u	v	m	s	λ
E65NN10	0.56	0.01	0.88098	0.0475	6.26
BootMean	0.5614	0.099	0.9036	0.077	6.4934
E65NN11	0.555	0.011	0.86733	0.01427	7.507
BootMean	0.561	0.011	0.8759	0.0217	7.6496
E65NN12	0.5638	0.0134	0.83523	0.01056	7.1016
BootMean	0.5611	0.0134	0.8432	0.0147	7.276
E65NN13	0.586	0.0157	0.8375	0.0688	5.2355
BootMean	0.584	0.0152	0.8727	0.0105	5.5799
E65NN14	0.5799	0.0171	0.8494	0.0266	6.95
BootMean	0.5801	0.0172	0.859	0.0355	7.187
E65NN15	0.5833	0.0186	0.8599	0.04313	6.83
BootMean	0.5943	0.0211	0.8407	0.0568	6.717

Table A.7: Bootstrap Mean of parameters with original MLEs

Data	u	v	m	s	λ
E75NN10	0.5564	0.01	0.8706	0.0253	6.704
BootMean	0.5597	0.0101	0.8976	0.063	6.7961
E75NN11	0.5527	0.0114	0.8621	0.0006	7.9
BootMean	0.5541	0.0138	0.8706	0.0089	8.045
E75NN12	0.5616	0.0138	0.8349	-0.0021	7.484
BootMean	0.5622	0.0138	0.8429	0.0021	7.6423
E75NN13	0.5838	0.0165	0.8164	0.0389	5.442
BootMean	0.5849	0.0159	0.8612	0.01	5.5859
E75NN14	0.5778	0.0179	0.8455	0.0075	7.333
BootMean	0.5788	0.0178	0.8593	0.0199	7.5704
E75NN15	0.5807	0.0196	0.8532	0.0161	7.337
BootMean	0.5834	0.0195	0.8777	0.0488	7.6609

Table A.8: Bootstrap variance of parameters for data generated by e-value 10^{-65}

Data	u	v	m	s	λ
E65k10	0.00022	0.0000024	0.00548	0.00654	3.78689
E65k11	0.00011	0.0000028	0.00385	0.00112	2.81897
E65k12	0.00009	0.0000028	0.00329	0.00062	1.73653
E65k13	0.00022	0.0000042	0.00581	0.01104	1.97419
E65k14	0.0002	0.0000077	0.00507	0.00237	2.92911
E65k15	0.00152	0.0000682	0.01686	0.00485	5.56942

Table A.9: Bootstrap variance of parameters for data generated by e-value 10^{-75}

Data	u	v	m	s	λ
E75k10	0.00027	0.0000025	0.00532	0.00788	4.55715
E75k11	0.00013	0.0000023	0.00365	0.00156	3.33778
E75k12	0.00009	0.0000028	0.0031	0.00056	1.92929
E75k13	0.00026	0.0000042	0.00642	0.01716	2.35129
E75k14	0.00019	0.0000059	0.00416	0.00327	3.16054
E75k15	0.00063	0.0000174	0.00701	0.01005	5.88444

Table A.10: Bootstrap variance of parameters for tree 10pop

Model	M	$u \times 10^{-5}$	$v \times 10^{-5}$	$m \times 10^{-4}$	$s \times 10^{-3}$	$\lambda \times 10^{-5}$
1	All	0.083617	0	0	0	5.859859
	AC	0.090300	0	0	0	8.570779
	AG	0.726428	0	0	0	8.287492
4	All	0.543265	0	0.849113	0	1.400563
	AC	0.525031	0	0.969364	0	1.878841
	AG	3.858454	0	6.70091	0	2.820408
7	All	0	0.010316	0	0	5.689152
	AC	0	0.010359	0	0	8.331206
	AG	0	0.145220	0	0	7.855364
10	All	0	0.068828	0.942256	0	1.302372
	AC	0.048744	0.066638	1.054005	0	1.746395
	AG	0	1.076928	9.75446	0	2.514218
13	All	0.193306	0	0	0.731373	2.578319
	AC	0.222778	0	0	0.824936	4.238369
	AG	3.046808	0	0	1.976508	7.489087
16	All	0.652487	0	0.91231	0.124076	1.225253
	AC	0.686034	0	1.025494	0.142051	1.880466
	AG	5.730569	0	7.076667	0.348493	3.220305
19	All	2.940856	0.069072	0	3.179191	2.931216
	AC	3.367083	0.074061	0	3.809749	4.9716
	AG	0.120333	0.172743	0	3.874215	5.2662
22	All	8.522772	0.325311	1.061442	0.586543	1.712359
	AC	9.077616	0.319	1.156817	0.663833	2.688786
	AG	0.230172	1.668241	10.640485	1.269306	3.393445

Table A.11: Bootstrap variance of parameters for tree 20pop

Model	Data	$u \times 10^{-5}$	$v \times 10^{-6}$	$m \times 10^{-4}$	$s \times 10^{-3}$	$\lambda \times 10^{-5}$
EC1	All	0.10618	0	0	0	7.49109
	AC	0.11504	0	0	0	11.55062
	AG	0.79354	0	0	0	10.40822
EC2	All	0.65017	0	0.87168	0	1.42101
	AC	0.67525	0	1.01971	0	2.04282
	AG	4.1442	0	7.79766	0	2.75304
EL1	All	0	0.10046	0	0	7.27548
	AC	0	0.10175	0	0	11.61393
	AG	0	1.625	0	0	9.98254
EL2	All	0	0.68257	0.92894	0	1.33346
	AC	0.0483	0.68973	1.07079	0	1.91054
	AG	0.09488	8.08346	8.40386	0	2.45954
PC1	All	0.16157	0	0	2.49301	1.91326
	AC	0.17853	0	0	2.80129	3.14466
	AG	4.1945	0	0	6.19529	8.68535
PC2	All	0.78595	0	0.93974	0.33556	1.27542
	AC	0.8714	0	1.09683	0.35221	2.03207
	AG	8.20286	0	8.48603	0.80346	3.27171
PL1	All	3.36386	0.62582	0	10.44524	93.05365
	AC	3.32496	0.69655	0	9.3096	3.31223
	AG	1.71302	2.01889	0	10.72871	6.62691
PL2	All	13.68657	0.50637	1.03321	1.29865	1.65239
	AC	16.13145	5.722	1.18584	1.35653	2.67336
	AG	23.17498	20.10449	9.36832	2.30834	3.79356

Table A.12: Bootstrap variance of parameters for tree 5pop

Model	Data	$u \times 10^{-5}$	$v \times 10^{-5}$	$m_1 \times 10^{-3}$	$m_2 \times 10^{-3}$	$m_3 \times 10^{-3}$	$s \times 10^{-3}$	$\lambda_1 \times 10^{-5}$	$\lambda_2 \times 10^{-5}$	$\lambda_3 \times 10^{-5}$
EC1	All	0.0776	0	0	0	0	0	1.2027	7.1586	8.2727
	AC	0.0846	0	0	0	0	0	1.8481	10.926	12.061
	AG	0.7093	0	0	0	0	0	2.4543	5.4537	10.938
	AT	12.385	0	0	0	0	0	18.947	23.862	55.108
EC2	All	0.4453	0	0.1961	0.3392	0.2469	0	0.6363	1.8545	1.8289
	AC	0.2331	0	0.0801	0.3877	0.2317	0	0.8698	2.5409	2.6714
	AG	3.0041	0	0.9839	2.2350	2.2208	0	1.0112	3.5407	4.1949
	AT	40.759	0	7.0326	32.091	32.009	0	5.0359	9.0236	16.387
EL1	All	0	0.0111	0	0	0	0	1.1709	6.9129	8.0671
	AC	0	0.0107	0	0	0	0	1.8152	10.609	11.828
	AG	0	0.1438	0	0	0	0	2.4343	5.1582	10.311
	AT	0	4.2116	0	0	0	0	16.674	21.18	48.457
EL2	All	0	0.1648	0.7613	0.3329	0.2498	0	0.4408	1.6111	1.7227
	AC	0	0.1713	10.203	0.3975	0.2334	0	0.6866	2.3405	2.5632
	AG	0.9893	4.7169	3.9321	2.0919	2.2043	0	1.0509	3.4175	3.5587
	AT	2499	16306	21.04	31.063	31.541	0	6.8763	9.5433	16.821
PC1	All	0.2454	0	0	0	0	0.4787	0.5256	4.1966	4.1823
	AC	0.2059	0	0	0	0	0.438	0.7128	4.8786	5.4262
	AG	2.6582	0	0	0	0	1.1207	1.8445	4.9527	8.8517
	AT	44.967	0	0	0	0	31.154	10.383	20.639	47.92
PC2	All	0.3555	0	0.1452	0.3391	0.2488	0.0962	0.4026	1.3669	1.6014
	AC	0.2861	0	0.0785	0.3901	0.2336	0.1045	0.5621	2.0354	2.5138
	AG	3.6839	0	0.9693	2.2464	2.2251	0.4027	0.8768	3.3352	3.9878
	AT	68.057	0	7.1456	32.385	32.108	13.907	3.5715	7.6036	16.612
PL1	All	3.5249	0.0876	0	0	0	2.9067	0.2226	2.275	3.0912
	AC	3.605	0.0927	0	0	0	3.3899	0.3428	3.6383	4.9609
	AG	0.2926	0.1346	0	0	0	3.0728	0.921	2.7285	5.2119
	AT	25.042	4.2878	0	0	0	98.766	3.0242	6.4067	14.829
PL2	All	7.272	0.5027	1.3964	0.3397	0.2523	0.7195	0.1948	1.3852	2.0418
	AC	9.1239	0.6657	1.945	0.4060	0.2371	0.757	0.3066	2.1909	3.3543
	AG	18.217	10.396	13.732	2.0922	2.1959	2.5287	0.7426	3.2851	3.9224
	AT	1253.9	2772.6	41.374	34.886	32.711	104.52	1.2187	3.146	8.1527

Table A.13: Bootstrap variance of parameters for tree 8pop

Model	Data	$u10^{-5}$	$v10^{-6}$	$m_1 10^{-3}$	$m_2 10^{-3}$	$m_3 10^{-3}$	$m_4 10^{-3}$	$m_5 10^{-3}$	$s \times 10^{-3}$	$\lambda_1 10^{-5}$	$\lambda_2 10^{-5}$	$\lambda_3 10^{-5}$	$\lambda_4 10^{-5}$	$\lambda_5 10^{-5}$
EC1	All	0.0768	0	0	0	0	0	0	0	1.3433	2.1475	7.5073	6.2545	8.0251
	AC	0.0878	0	0	0	0	0	0	0	1.8581	3.2941	10.992	8.6166	12.547
	AG	0.6812	0	0	0	0	0	0	0	2.0945	2.9152	17.767	11.879	10.999
EC2	All	0.8004	0	0.3876	1.474	0.3848	0.2363	0.2258	0	0.6644	0.5701	1.6643	1.5148	1.8932
	AC	0.6734	0	0.3391	1.5993	0.3901	0.2779	0.2455	0	0.8797	0.828	2.3908	1.9352	2.6482
	AG	5.1908	0	1.5675	13.662	6.6044	2.194	2.4153	0	0.9664	0.9643	3.2283	3.7535	4.3772
EL1	All	0	0.0989	0	0	0	0	0	0	1.3438	2.1242	7.2638	6.0596	7.8209
	AC	0	0.1088	0	0	0	0	0	0	1.8605	3.2788	10.685	8.3963	12.211
	AG	0	1.3502	0	0	0	0	0	0	2.1112	2.8076	16.783	11.302	10.407
EL2	All	0	1.8875	0.8045	1.6013	0.3884	0.2362	0.2261	0	0.5116	0.5148	1.6075	1.4495	1.7993
	AC	0	1.9309	1.0558	1.7528	0.3954	0.2784	0.2417	0	0.6794	0.7311	2.3228	1.8492	2.4844
	AG	0.3097	39.449	4.0175	13.872	6.6018	2.1804	2.4254	0	0.8586	0.8776	3.0173	3.5926	4.0332
PC1	All	0.1731	0	0	0	0	0	0	0.5269	0.4123	0.6439	2.5761	2.372	3.2436
	AC	0.2097	0	0	0	0	0	0	0.6422	0.5774	1.0539	3.9651	3.3969	4.7749
	AG	3.0221	0	0	0	0	0	0	1.9831	1.2668	1.7178	9.5611	9.8041	9.2009
PC2	All	0.5097	0	0.2405	1.4477	0.3871	0.2368	0.2264	0.1493	0.38	0.3239	1.2833	1.2845	1.7708
	AC	0.4541	0	0.2034	1.5791	0.3898	0.2788	0.2475	0.1565	0.4994	0.4863	1.9408	1.7718	2.4394
	AG	6.5892	0	1.5972	13.653	6.6387	2.1977	2.4193	0.5516	0.8125	0.8531	3.0983	4.013	4.9347
PL1	All	2.8067	0.6611	0	0	0	0	0	2.7544	0.2337	0.3663	2.1015	2.2385	3.1584
	AC	3.4429	0.8033	0	0	0	0	0	3.5387	0.3409	0.6239	3.3474	3.3395	5.0678
	AG	0.5714	1.6102	0	0	0	0	0	4.0184	0.7416	0.9799	5.9535	6.0936	5.706
PL2	All	9.6962	8.4698	2.0388	1.5937	0.3911	0.2374	0.2287	0.6472	0.2248	0.2301	1.3933	1.528	2.1871
	AC	12.225	9.8229	2.4073	1.7719	0.3942	0.2792	0.2443	0.7231	0.3037	0.3578	2.1843	2.2395	3.2559
	AG	3.3595	68.798	8.1909	13.823	6.6645	2.1911	2.4122	1.9329	0.5544	0.6001	2.7168	3.9545	4.5649

Table A.14: Parameters for Divided Data of AllRepeat for Model PL2

u	v	m	s	$\lambda \times 10^{-3}$	$-\log L$
0.999989	0.108443	0.5758661	96.00984	1.1504	883.8795
0.99999	0.12844	0.5758647	96.0099	0.8291	695.3902
0.999988	0.128442	0.5758659	96.00983	1.1163	675.0056
0.99999	0.11844	0.5758647	96.0099	0.8309	793.7988
0.99999	0.10844	0.5758647	96.0089	0.9091	822.4733
0.99999	0.11844	0.5758646	96.0099	0.9226	747.6269
0.99999	0.11344	0.5758647	96.0099	0.8090	781.7875
0.999989	0.118442	0.5758665	96.00887	1.0288	727.5234
0.99999	0.118439	0.5758631	96.00989	1.1053	806.7188
0.999989	0.118441	0.5758644	96.0099	0.8781	684.2596
0.999989	0.12344	0.5758646	96.0099	0.8235	816.8429
0.999989	0.11844	0.5758646	96.0099	0.8309	711.3938
0.99999	0.11844	0.5058647	96.0089	0.9190	741.7655
0.99999	0.11844	0.5758649	96.0099	0.9462	757.9068
0.99999	0.11344	0.5758647	96.0099	0.8162	739.8546
0.99999	0.118441	0.575865	96.0099	0.8697	768.5873
0.99999	0.11844	0.5958647	96.0089	0.8165	751.9211
0.99999	0.11544	0.5958647	96.0099	0.8137	795.2097
0.99999	0.12344	0.5958647	96.0099	0.8101	731.8490
0.99999	0.12344	0.575865	96.00889	0.8802	719.2459
0.999989	0.12344	0.5758647	96.00889	0.8209	623.4584
0.999989	0.12344	0.5758647	96.00889	0.8211	715.8899
0.9999	0.11544	0.5758647	96.0089	0.8092	712.4332
0.99999	0.11844	0.5758646	96.0089	0.8161	844.0381
0.999989	0.118441	0.5758648	96.0089	0.8368	725.9261

Table A.15: Parameters for Divided Data of ExonRepeat for Model PL2

u	v	m	s	$\lambda \times 10^{-4}$	$-\log L$
0.98747	0.13607	0.76227	177.3814	1.21484	36.65764
0.99999	0.06475	0.40541	180.729	0.65025	56.61255
0.99999	0.12704	0.75917	177.1831	0.98702	45.01081
0.99999	0.12789	0.72723	176.7069	3.66850	43.79702
0.99999	0.12789	0.72723	176.7069	2.95931	29.19192
0.99999	0.13801	0.73844	176.8565	0.60128	52.6082
0.96946	0.12609	0.74899	176.7763	1.28097	40.8808
0.99999	0.12789	0.72723	176.7069	2.91102	31.25482
0.99999	0.12789	0.72723	176.7069	2.91208	37.50659
0.99997	0.13223	0.72559	176.7695	0.10000	33.066
0.99997	0.10392	0.74721	176.7069	1.09584	50.14624
0.99999	0.13894	0.74723	176.7069	2.91455	39.56728
0.99863	0.12924	0.74774	176.7057	1.41421	36.58562
0.99788	0.14295	0.74789	176.7078	14.2822	40.63495
0.99999	0.12789	0.75723	176.7069	3.00199	44.55519
0.99923	0.13749	0.80286	177.755	2.09211	58.86397
0.99794	0.13709	0.73749	176.8275	0.10000	36.37067
0.99612	0.13628	0.83784	178.7766	0.69854	48.32215
0.99996	0.13070	0.74724	176.7069	1.02964	49.23896
0.99999	0.13524	0.77339	176.4691	1.26452	35.80277

Table A.16: Mean and Standard Deviation of data from Table A.14 and A.15

	u	v	m	s	$\lambda \times 10^{-4}$
MeanAllRepeat	0.999986	0.118801	0.575465	96.00949	8.883972
SDAllrepeat	0.000018	0.004957	0.015937	0.000498	1.040247
MeanExonRepeat	0.997331	0.127776	0.734748	177.1296	2.258943
SDExonRepeat	0.007159	0.016879	0.082216	0.993910	3.028124

Bibliography

[1] http://hgdownload.cse.ucsc.edu/goldenpath/hg19/chromosomes/.

[2] Ali A., M. S. Rahman, and W. F. Smyth. Inferring an indeterminate string from a prefix graph. *Journal of Discrete Algorithms*, 32:6–13, 2015.

[3] K. Abrahamson. Generalized string matching. *SIAM Journal of Computing*, 16(6):1039–1051, 1987.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, 1974.

[5] A. Alatabbi, J. W. Daykin, J. Kärkkäinen, M. S. Rahman, and W. F. Smyth. V-Order: new combinatorial properties & a simple comparison algorithm. *Discrete Appl. Math.*, 215:41–46, 2016.

[6] A. Alatabbi, J. W. Daykin, and M. S. Rahman. Linear algorithms for computing the Lyndon border array and the Lyndon suffix array. 2015. *arXiv:1506.06983.*

[7] A. Alatabbi, A. S. M. Sohidull Islam, M. S. Rahman, J. Simpson, and W. F. Smyth. Enhanced covers of regular and indeterminate strings using prefix tables. *Journal of Automata, Languages and Combinatorics*, 21(3):131–147, 2016.

[8] A. Alatabbi, M. S. Rahman, and W. F. Smyth. Computing covers using prefix tables. *Discrete Applied Mathematics*, 212:2–9, 2016.

[9] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and new distributed algorithm. In *Proc. 1th Annual ACM Symp. on Parallel Algorithms & Architectures*, pages 258–264, 2002.

[10] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[11] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasi–periodicities in strings. Technical Report 90.5, The Leonardo Fibonacci Institute, Trento, Italy, 1990.

[12] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39:17–20, 1991.

[13] U. Baier. Linear-time suffix sorting — a new approach for suffix array construction. M.Sc. Thesis, University of Ulm, 2015.

[14] U. Baier. Linear-time suffix sorting — a new approach for suffix array construction. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The “runs” theorem. 2014. *arXiv:1406.0263v6*.

[16] M. F. Bari, M. S. Rahman, and R. Shahriyar. Finding all covers of an indeterminate string in o (n) time on average. In *Stringology*, pages 263–271, 2009.

[17] F. Bassino, J. Clément, and C. Nicaud. The standard factorization of Lyndon words: an average point of view. *Discrete Mathematics*, 290(1):1–25, 2005.

[18] F. Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words.* Chapman & Hall CRC, 2008.

[19] W. Bland, G. Kucherov, and W. F. Smyth. Prefix table construction and conversion. *Proc. 24th IWOCA*, pages 41–53, 2013.

[20] D. Breslauer. An on–line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992.

[21] J. R. Brouwer, R. Willemsen, and B. A. Oostra. Microsatellite repeat instability and neurological disease. *Bioessays*, 31(1):71–83, 2009.

[22] P. Calabrese and R. Durrett. Dinucleotide repeats in the drosophila and human genomes have complex, length-dependent mutation processes. *Molecular biology and evolution*, 20(5):715–725, 2003.

[23] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus. iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[24] M. Christodoulakis, P. J. Ryan, W. F. Smyth, and S. Wang. Indeterminate strings, prefix arrays and undirected graphs. *Theoretical Comput. Sci.*, 600:34–48, 2015.

[25] J. Clément, M. Crochemore, and G. Rindone. Reverse engineering prefix tables. In *Proc. 26th STACS*, pages 289–300, 2009.

[26] R. Cole, C. S. Ilopoulos, M. Mohamed, W. F. Smyth, and L. Yang. The complexity of the minimum k-cover problem. *Journal of Automata, Languages and Combinatorics*, 10(5-6):641–653, 2005.

[27] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert Informatique, 2001.

[28] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

[29] M. Crochemore, C. S. Iliopoulos, S. P. Pissis, and G. Tischler. Cover array string reconstruction. In *CPM*, volume 6129, pages 251–259. Springer, 2010.

[30] M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific, 2002.

[31] T.-N. Danh and D. E. Daykin. The structure of V–order for integer vectors. *Congressus Numerantium*, 113:43–53, 1996.

[32] D. E. Daykin and J. W. Daykin. Lyndon–like and V–order factorizations of strings. *J. Discrete Algorithms*, 1(3–4):357–365, 2003.

[33] D. E. Daykin and J. W. Daykin. Properties and construction of unique maximal factorization families for strings. *Internat. J. Found. Comput. Sci.*, 19(4):1073–1084, 2008.

[34] D. E. Daykin, J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Generic algorithms for factoring strings. In *Proc. Memorial Symp. for Rudolf Ahlswede,*

H. Aydinian, F. Cicalese & C. Deppe, (eds.), volume 7777 of *Lecture Notes in Comput. Sci.*, pages 402–418. Springer-Verlag, 2013.

[35] D. E. Daykin, J. W. Daykin, and W. F. Smyth. Combinatorics of unique maximal factorization families (UMFFs). *Fund. Inform. 97–3, Special Issue on Stringology, R. Janicki, S. J. Puglisi & M. S. Rahman (eds.)*, pages 295–309, 2009.

[36] D. E. Daykin, J. W. Daykin, and W. F. Smyth. String comparison and Lyndon–like factorization using V–order in linear time. In *22nd Annual Symp. on Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 65–76. Springer-Verlag, 2011.

[37] D. E. Daykin, J. W. Daykin, and W. F. Smyth. A linear partitioning algorithm for hybrid Lyndons using V–order. *Theoret. Comput. Sci.*, 483:149–161, 2013.

[38] J. W. Daykin, F. Franek, J. Holub, A. S. M. Sohidull Islam, and W. F. Smyth. Reconstructing a string from its lyndon arrays. *Theoretical Computer Science*, 2017.

[39] A. Di Rienzo, A. C. Peterson, J. C. Garza, A. M. Valdes, M. Slatkin, and N. B. Freimer. Mutational processes of simple-sequence repeat loci in human populations. *Proceedings of the National Academy of Sciences*, 91(8):3166–3170, 1994.

[40] J.-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4:363–381, 1983.

[41] J.-P. Duval, T. Lecroq, and A. Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005.

[42] J.-P. Duval, T. Lecroq, and A. Lefebvre. Efficient validation and construction of Knuth-Morris-Pratt arrays. In *Proc. Conference in Honour of Donald E. Knuth*, 2007.

[43] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-based compressed suffix tree. In *19th Annual Symp. on Combinatorial Pattern Matching*, volume 5029 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2008.

[44] M. J. Fischer and M. S. Paterson. String matching and other products. In R.M. Karp, editor, *Complexity of Computation,*, pages 113–125. American Mathematical Society, 1974.

[45] T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, and W. Tyczynski. Enhanced string covering. *Theoretical Computer Science*, 506:102 – 114, 2013.

[46] F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a border array in linear time. *J. Combinatorial Math. & Combinatorial Computing*, 42:223–236, 2002.

[47] F. Franek, A. S. M. Sohidull Islam, M. S. Rahman, and W. F. Smyth. Algorithms to compute the lyndon array. In *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2015*, pages 172–184, 2016.

[48] F. Franek, R. J. Simpson, and W. F. Smyth. The maximum number of runs in a string. *Proc. 14th Australasian Workshop on Combinatorial Algorithms*, pages 26–35, 2003.

[49] J. C. Garza, M. Slatkin, and N. B. Freimer. Microsatellite allele frequencies in humans and chimpanzees, with implications for constraints on allele size. *Molecular Biology and Evolution*, 12(4):594–603, 1995.

[50] P. Gawrychowski, A. Jeż, and L. Jeż. Validating the Knuth-Morris-Pratt failure function, fast and on-line. In *Proc. 5th Annual Computer Science Symposium in Russia*, volume 6072 of *Lecture Notes in Comput. Sci.*, pages 132–143. Springer-Verlag, 2010.

[51] P. Gawrychowski, A. Jeż, and L. Jeż. Validating the Knuth-Morris-Pratt failure function, fast and on-line. *Theory of Computing Systems*, 54:337–372, 2014.

[52] K. Goto and H. Bannai. Simpler and faster Lempel-Ziv factorization. In *Data Compression Conference*, pages 133–142, 2013.

[53] P. K. Gupta and R. K. Varshney. The development and use of microsatellite markers for genetic analysis and plant breeding with emphasis on bread wheat. *Euphytica*, 113(3):163–185, 2000.

[54] C. Hohlweg and C. Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003.

[55] J. Holub and W. F. Smyth. Algorithms on indeterminate strings. *Proc. 14th AWOCA*, pages 36–45, 2003.

[56] J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50, 2008.

[57] C. S. Iliopoulos, M. Mohamed, and W. F. Smyth. New complexity results for the k-covers problem. *Information Sciences*, 181(12):2571–2575, 2011.

[58] C. S. Iliopoulos and L. Mouchard. Quasiperiodicity and string covering. *Theoretical Computer Science*, 218(1):205–216, 1999.

[59] C. S. Iliopoulos and W. F. Smyth. On-line algorithms for k-covering. 1998.

[60] M. Kimura and T. Ohta. Stepwise mutation model and distribution of allelic frequencies in a finite population. *Proceedings of the National Academy of Sciences*, 75(6):2868–2872, 1978.

[61] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd edition*. Addison-Wesley, 1973.

[62] D. Kosolobov. Lempel-Ziv factorization may be harder than computing all runs. In *Proc. 32nd STACS*, 2015. *arXiv:1409.5641*.

[63] S. Kruglyak, R. T. Durrett, M. D. Schug, and C. F. Aquadro. Equilibrium distributions of microsatellite repeat length resulting from a balance between slippage events and point mutations. *Proceedings of the National Academy of Sciences*, 95(18):10774–10778, 1998.

[64] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[65] G. Levinson and G. A. Gutman. High frequencies of short frameshifts in poly-ca/tg tandem repeats borne by bacteriophage m13 in escherichia coli k-12. *Nucleic Acids Research*, 15(13):5323–5338, 1987.

[66] Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica, 32–1, 95–106*, 2002.

[67] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Maths.*, 25:145–153, 1989.

[68] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5:422–432, 1984.

[69] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *Siam Journal on Computing*, 22(5):935–948, 1993.

[70] U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. In *Proc. First Annual ACM-SIAM Symp. Discrete Algs.*, pages 319–327, 1990.

[71] D. Moore and W. F. Smyth. An optimal algorithm to compute all the covers of a string. *Inform. Process. Lett. 50, 239-246*, 1994.

[72] D. Moore and W. F. Smyth. Correction to: An optimal algorithm to compute all the covers of a string. *Inform. Process. Lett. 54 101-103*, 1995.

[73] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced–sorting. *Data Compression Conference*, 0:193–202, 2009.

[74] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *22nd Annual Symp. on Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2011.

[75] W. Powell, G. C. Machray, and J. Provan. Polymorphism revealed by simple sequence repeats. *Trends in plant science*, 1(7):215–222, 1996.

[76] C. R. Primmer, H. Ellegren, N. Saino, and A. P. Moller. Directional evolution in germline microsatellite mutations. *Nature genetics*, 13(4):391–393, 1996.

[77] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):1–31, July 2007.

[78] R. Sainudiin, R. T. Durrett, C. F. Aquadro, and R. Nielsen. Microsatellite mutation models: Insights from a comparison of humans and chimpanzees. *Genetics*, 168(1):383–395, 2004.

[79] J. Sawada and F. Ruskey. Generating Lyndon brackets: an addendum to "Fast algorithms to generate necklaces, unlabeled necklaces amd irreducible polynomials over GF(2)". *J. Algorithms*, 46:21–26, 2003.

[80] W. F. Smyth. *Computing Patterns in Strings*. Pearson/Addison–Wesley, 2003.

[81] W. F. Smyth. Computing regularities in strings: a survey. *European Journal of Combinatorics*, 34(1):3–14, 2013.

[82] W. F. Smyth and S. Wang. New perspectives on the prefix array. *Proc. 15th SPIRE, Lecture Notes in Computer Science, LNCS 5280, Springer Verlag*, 27:133–143, 2008.

[83] W. F. Smyth and S. Wang. A new approach to the periodicity lemma on strings with holes. *Theoretical Computer Science*, 410(43):4295 – 4302, 2009.

[84] N. Sugiura. Further analysts of the data by akaike's information criterion and the finite corrections: Further analysts of the data by akaike's. *Communications in Statistics-Theory and Methods*, 7(1):13–26, 1978.

[85] G. Wang and K. M. Vasquez. Z-dna, an active element in the genome. *Frontiers in bioscience: a journal and virtual library*, 12:4424–4438, 2006.

[86] J. L Weber and C. Wong. Mutation of human short tandem repeats. *Human molecular genetics*, 2(8):1123–1128, 1993.