M.Sc. Engg. Thesis

String Regularities and Degenerate Strings

by
Md. Faizul Bari

Submitted to

Department of Computer Science and Engineering
in partial fulfilment of the requirments for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka 1000

December 2009

The thesis titled "**String Regularities and Degenerate Strings**," submitted by Md. Faizul Bari, Roll No. 100705050P, Session October 2007, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on December 19, 2009.

Board of Examiners

1. _____

Dr. M. Sohel Rahman — Chairman (Supervisor)
Assistant Professor
Department of Computer Science and Engineering
BUET, Dhaka 1000

2. _____

Dr. Md. Monirul Islam — Member (Ex-officio)
Professor & Head
Department of Computer Science and Engineering
BUET, Dhaka 1000

3. _____

Dr. M. Kaykobad — Member
Professor
Department of Computer Science and Engineering
BUET, Dhaka 1000

4. _____

Dr. Md. Mostofa Akbar — Member
Associate Professor
Department of Computer Science and Engineering
BUET, Dhaka 1000

5. _____

Dr. Mohammad Nurul Huda — Member (External)
Associate Professor
Department of Computer Science and Engineering
United International University, Dhaka 1209

Candidate's Declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Md. Faizul Bari
Candidate

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

All praises due to Allah, the most benevolent and merciful.

I express my heart-felt gratitude to my supervisor, Dr. M. Sohel Rahman for his constant supervision of this work. He helped me a lot in every aspect of this work and guided me with proper directions whenever I sought one. His patient hearing of my ideas, critical analysis of my observations and detecting flaws (and amending thereby) in my thinking and writing have made this thesis a success.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. Md. Monirul Islam, Dr. M. Kaykobad, Dr. Md. Mostofa Akbar and specially the external member Professor Dr. Mohammad Nurul Huda.

I also thank one of my friend Rifat Shahriyar, who was my partner in the early works of my research. We jointly wrote a paper on the very basic idea of this work and got the paper published in a conference, PSC'09.

In this regard, I remain ever grateful to my beloved father, mother, and wife who always exists as sources of inspiration behind every success of mine I have ever made.

Abstract

Finding patterns and characterizing regularities in strings are in the heart of research in the field of stringology. These problems are interesting as fundamental computer science problems and often turn out to be important and useful in many applications. In case of string regularities *periods*, *repeats*, and *covers* are the most common forms and there are several efficient algorithms for computing regularities in a string.

In this thesis, we focus on a generalization of the regular string commonly known as *degenerate strings*. A degenerate string can be represented as a sequence $T = T[1]T[2] \ldots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet. It is an extended notion of the regular string which is very much applicable especially in the context of computational biology. Pattern matching problem for degenerate string has also been investigated in the literature albeit with little success. The lack of any efficient algorithms for degenerate pattern matching may be attributed at least partially to the lack of fruitful research on regularities of degenerate strings. Very recently, the issue of regularities in degenerate string has received some attention. In this thesis, we study the problem of string regularities for degenerate strings. We present two novel and efficient algorithms for computing cover array for degenerate strings. Both of our algorithms run in $O(n^2)$ time and space. Using string combinatorics and probability analysis we have further proved that expected number of both borders and covers for a degenerate string is bounded by a constant. By using this probabilistic analysis, we have also shown that the average running time and space complexity of our algorithms are $O(n)$.

Chapter 1

Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology, repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory, regularities are important in the analysis of stochastic processes. In computer science, repetitive elements in strings are important in e.g. data compression, computational music analysis, coding, automata and formal language theory. As a result, in the last 20 years, string regularities have drawn a lot of attention from different disciplines of science.

1.1 Motivation

String regularities like repeats, borders, covers etc. are much studied problems for (regular) strings. Many important and well known efficient string algorithms depend on these regularities. The famous KMP [26] pattern matching algorithm depends on the *failure function*, which yields basically the same information as the *border array*. Another well known pattern matching algorithm commonly known as the Boyer-Moore [13] algorithm also depends on the information provided by this kind of regularities in strings. String regularities is very interesting from pure theoretical point of view as well. For (regular)

strings there are many algorithms for finding regularities which will be discussed in the following section. But for degenerate strings, a generalized and extended notion of regular string, there is still no efficient algorithm for finding such regularities. As we will see later, the computational complexity for finding the border and cover array of a degenerate string is much more than for traditional strings. And these and related topics have drawn much interest in recent years [4, 35, 3, 5, 23].

Degenerate strings are very much applicable especially in the context of computational biology [6]. Especially, because of the errors in experimentations, resulting DNA sequence may have positions where the nitrogen base is not properly known or only partially known: this basically gives us a degenerate string as opposed to a regular string. Furthermore, regularities like covers have applications for finding tandem repeats in the DNA sequence of a species. Tandem repeat describes a pattern that helps determine an individual's inherited traits e.g. behavior or habits inherited from one's parents. Tandem repeats are short nucleotide sequences that occur in adjacent or overlapping positions in DNA sequences. This type of repetition is exactly what is described by the cover array. So for finding tandem repeats in a DNA sequence we can apply the cover array algorithm.

Degenerate pattern matching has also been studied in recent years [24, 28]. But still there is no pattern matching algorithm for degenerate strings that is up to the mark. This absence of efficient algorithms for degenerate pattern matching may very well be attributed to the absence of deep and insightful results related to degenerate string regularities. Definitely, this, besides pure theoretical interest, has motivated researchers in stringology to study the regularities of degenerate strings with great interest in recent times. Clearly, the idea is that the insight about degenerate string regularities and efficient algorithms thereof would lead us to efficient algorithms for degenerate pattern matching.

1.2 Problem statement

The objective of this research is to devise novel algorithms for computing different kinds of regularities for degenerate strings. We mainly focus on computing the following data structures which contain information about different regularities in a string

- Prefix array: Describes the lengths of the longest common prefixes between x and each prefix of x.
- Border array: Represents the lengths of the longest border of each prefix of x.
- Cover array: Describes the lengths of the longest cover of each prefix of x..

In particular, in this thesis, we focus on the following problems:

Problem 1 Computing the prefix array of a degenerate string over a fixed alphabet Σ.
Input: *We are given a degenerate string x, of length n on a fixed alphabet Σ.*
Output: *We need to compute the* prefix array Π *of x.*

Problem 2 Computing the border array of a degenerate string over a fixed alphabet Σ.
Input: *We are given a degenerate string x, of length n on a fixed alphabet Σ.*
Output: *We need to compute the* border array β *of x.*

Problem 3 Computing the cover array of a degenerate string over a fixed alphabet Σ.
Input: *We are given a degenerate string x, of length n on a fixed alphabet Σ.*
Output: *We need to compute the* cover array γ *of x.*

1.3 Related works

The most common forms of string regularities are periods and repeats and there are several $O(n \log n)$ time algorithms for finding repetitions [9, 14], in a string x, where n is the length of x. Apostolico and Breslauer [7] gave an optimal $O(\log \log n)$-time parallel algorithm for finding all the repetitions of a string of length n. The preprocessing of the Knuth-Morris-Pratt algorithm [27] finds all periods of every prefix of x in linear time.

In many cases, it is desirable to relax the meaning of repetition. For instance, if we allow overlapping and concatenations of periods in a string we get the notion of *covers*. After periods and repeats, cover is the most popular form of regularities in strings. The idea of cover generalizes the idea of periods or repeats. A substring c of a string x is called a *cover* of x if and only if x can be constructed by concatenation and superposition of c. Another common string regularity is the *seed* of a string. A seed is an extended cover in the sense that it is a cover of a superstring of x.

Clearly, x is always a cover of itself. If a proper substring pc of x is also a cover of x, then pc is called a *proper cover* of x. For example, the string $x = abcababcabcabcab$ has covers x and $abcab$. Here, $abcab$ is a proper cover. A string that has a proper cover is called *coverable*; otherwise it is *superprimitive*. The notion of covers was introduced by Apostolico, Farach and Iliopoulos in [8], where a linear-time algorithm to test the superprimitivity of a string was given. Moore and Smyth in [32] gave linear time algorithms for finding all covers of a string.

In this research, we are interested in regularities of degenerate strings. A degenerate string is a sequence $T = T[1]T[2]\dots T[n]$, where $T[i] \subseteq \Sigma$ for $1 \leq i \leq n$, and Σ is a given alphabet. If $|T[i]| > 1$, then the position i of T is referred to as a *non-solid position*. The simplest form of degenerate string is one in which each non-solid position can only contain a don't care character, denoted by '*'; the don't care character matches any letter in the alphabet Σ. Effectively, $* = \{\sigma_i \in \Sigma \mid 1 \leq i \leq |\Sigma|\}$. The pattern matching problem with don't care characters has been solved by Fischer and Paterson [17] more than 30 years

ago. However, although the algorithm in [17] is efficient in theory, it is not very useful in practice. Pattern matching problem for degenerate strings has also been investigated in the literature, albeit with little success. For example, in [23], an algorithm was presented which works well only if the alphabet size is small. Pattern matching for degenerate strings has mainly been handled by bit mapping techniques (Shift-Or method) [10, 38]. These techniques have been used to find matches for a degenerate pattern p in a string x [21] and the agrep utility [37] has been virtually one of the few practical algorithms available for degenerate pattern-matching.

In [21] the authors extended the notion of degenerate string matching by distinguishing two distinct forms of degenerate match, namely, *quantum* and *deterministic*. Roughly speaking, a quantum match allows a non-solid letter to match two or more distinct letters during a single matching process; a determinate match restricts each non-solid letter to a single match [21].

Very recently, the issue of regularities in degenerate string has received some attention. For example, in [3], the authors investigated the regularities of *conservative* degenerate strings. In a conservative degenerate string the number non-solid positions is bounded by a constant. The authors in [3] presented efficient algorithms for finding the smallest *conservative cover* (number of non-solid position in the cover is bounded by a given constant), λ-conservative covers (conservative covers having length λ) and λ-conservative seeds. On the other hand, Antoniou et al. presented an $O(n \log n)$ algorithm to find the smallest cover of a degenerate string in [5] and showed that their algorithm can be easily extended to compute all the covers of x. The later algorithm runs in $O(n^2 \log n)$ time.

1.4 Main contributions

In this thesis we focus on efficiently solving Problems 1-3 defined above. Here we provide two algorithms called ***iCAb*** and ***iCAp*** for computing the cover array of a degenerate string x of length n. ***iCAb*** first computes the border array of x and then from the border

array computes the cover array of x. ***iCAb*** has a running time of $O(n^2)$ in the worst case. We further show that the running time of ***iCAb*** is $O(n)$ in the average case.

Our second algorithm, namely ***iCAp***, first computes the prefix array of x, and then using a novel approach computes the border array from the prefix array of a degenerate string. Finally, computes the cover array of x in worst case $O(n^2)$ time complexity.

In this research we have also shown that any algorithm for computing the border array and cover array of a degenerate string must have a lower bound of $O(n^2)$ in the worst case and provided two novel algorithm for doing this task in optimal time and space complexity.

1.5 Overview of the thesis

The rest of the chapters are organized as follows. Chapter 2 gives a preliminary description of some terminologies and concepts related to stringology that will be used throughout this thesis. Chapter 3 presents our first algorithm, ***iCAb***. In chapter 4 we describe our second algorithm, ***iCAp***. Chapter 5 draws the conclusion followed by some future research directions.

Chapter 2

Preliminaries

2.1 Basic definitions in stringology

A string is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The *length* of a string x is denoted by $|x|$. The *empty string*, the string of length zero, is denoted by ϵ. The i-th symbol of a string x is denoted by $x[i]$. A string $w \in \Sigma^*$, is a *substring* of x if $x = uwv$, where $u, v \in \Sigma^*$. We denote by $x[i \dots j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string $w \in \Sigma^*$ is a *prefix* (*suffix*) of x if $x = wy$ ($x = yw$), for $y \in \Sigma^*$. A string w is a *subsequence* of x (or x a *supersequence* of w) if w is obtained by deleting zero or more symbols at any positions from x. For example, *ace* is a subsequence of *abcabbcde*. For a given set S of strings, a string w is called a common subsequence of S if s is a subsequence of every string in S.

2.2 Repetitions in strings

The string xy is the *concatenation* of the strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1 \dots n]$ and $y = y[1 \dots m]$ such that $x[n -$

$i+1 \ldots n] = y[1 \ldots i]$ for some $i \geq 1$ (i.e., x has a suffix equal to a prefix of y), the string $x[1 \ldots n]y[i+1 \ldots m]$ is said to be a *superposition* of x and y. We also say that x *overlaps* with y. A substring y of x is called a *repetition* in x, if $x = uy^k v$, where u, y, v are substrings of x and $k \geq 2$, $|y| \neq 0$. For example, if $x = aababab$, then a (appearing in positions 1 and 2) and ab (appearing in positions 2, 4 and 6) are repetitions in x; in particular $a^2 = aa$ is called a *square* and $(ab)^3 = ababab$ is called a *cube*. A non-empty substring w is called a *period* of a string x, if x can be written as $x = w^k w'$ where $k \geq 1$ and w' is a prefix of w. The shortest period of x is called *the period* of x. For example, if $x = abcabcab$, then abc, $abcabc$ and the string x itself are periods of x, while abc is the period of x.

2.3 String borders, covers and seeds

A border u of x is a prefix of x that is also a suffix of x; thus $u = x[1 \ldots b] = x[n-b+1 \ldots n]$ for some $b \in \{0 \ldots n-1\}$. A substring w of x is called a *cover* of x, if x can be constructed by concatenating or overlapping copies of w. We also say that w covers x. For example, if $x = ababaaba$, then aba and x are covers of x. If x has a cover $w \neq x$, x is said to be *quasiperiodic*; otherwise, x is *superprimitive*. A substring w of x is called a *seed* of x, if w covers a superstring of x[1]. For example, aba and $ababa$ are some seeds of $x = ababaab$.

2.4 Prefix, border and cover array

Prefix, border and cover array are among the most important data structures in stringology. Many well known string algorithms like KMP [26], Boyer-Moore [13] are based on these data structures. Definitions of these data structures are given below:

[1]Note that, x is a superstring of itself. Therefore, every cover is also a seed but the reverse is not necessarily true.

2.4.1 Prefix array

The prefix array of x is an array Π such that for all $i \in \{1 \dots n\}$, $\Pi[i]$ = length of the longest common prefix between $x[1 \dots n]$ and $x[i \dots n]$. Since every prefix of any prefix of x is also a prefix of x, Π represents all the prefixes of every prefix of x. The prefix array of x can be defined as follows:

$$\forall\ i\ \in\ 1\ \dots\ n,\ \Pi[i]\ =\ lcp(x[1\ \dots\ n], x[i\ \dots\ n]) \tag{2.1}$$

Here *lcp* is the well known *longest common prefix* function. For any prefix $x[1 \dots i]$ of x, the following sequence

$$\Pi[i], \Pi^2[i], \dots, \Pi^m[i] \tag{2.2}$$

is well defined and monotonically decreasing to $\Pi^m = 0$ for some $m \geq 1$ and this sequence identifies every prefix of $x[1 \dots i]$.

2.4.2 Border array

The border array of x is an array β such that for all $i \in \{1 \dots n\}$, $\beta[i]$ = length of the longest border of $x[1 \dots i]$. Since every border of any border of x is also a border of x, β encodes all the borders of every prefix of x. The border array of x can be defined as follows:

$$\forall i \in 1 \dots n, \beta[i] =\ longest\ border\ of\ x[1 \dots i] \tag{2.3}$$

For every prefix $x[1 \dots i]$ of x, the following sequence

$$\beta[i], \beta^2[i], \ldots, \beta^m[i] \tag{2.4}$$

is well defined and monotonically decreasing to $\beta^m = 0$ for some $m \geq 1$ and this sequence identifies every border of $x[1 \ldots i]$.

2.4.3 Cover array

The *cover array* γ, is a data structure used to store the length of the longest proper cover of every prefix of x; that is for all $i \; \epsilon \; 1 \ldots \mathrm{n}$, $\gamma[i]$ = length of the longest proper cover of $x[1 \ldots i]$ or 0. In fact, since every cover of any cover of x is also a cover of x, it turns out that, the cover array γ describes all the covers of every prefix of x. The cover array of x can be defined as follows:

$$\forall i \in 1 \ldots n, \gamma[i] = \; longest \; cover \; of \; x[1 \ldots i] \tag{2.5}$$

For every prefix $x[1 \ldots i]$ of x, the following sequence

$$\gamma[i], \gamma^2[i], \ldots, \gamma^m[i] \tag{2.6}$$

is well defined and monotonically decreasing to $\gamma^m = 0$ for some $m \geq 1$ and this sequence identifies every cover of $x[1 \ldots i]$.

2.5 Examples of prefix, border and cover array for a regular string

The border array β and cover array γ are very important data structures in stringology. Many string algorithms are build upon these data structures. An example of border and

Table 2.1: Prefix, border and cover array of *abaababaabaababaababa*

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	b	a
$\Pi =$	0	0	1	3	0	6	0	1	8	0	1	3	0	8	0	1	3	0	3	0	3	0	1
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	2	3
$\gamma =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11	0	8	0	3

Table 2.2: Degenerate string *aa*[*abc*]*a*[*ac*]*bcaa*[*ac*]*bac*[*abc*]*a*[*bc*]

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$x =$	a	a	[abc]	a	[ac]	b	c	a	a	[ac]	b	a	c	[abc]	a	[bc]

cover array for the regular string *abaababaabaababaababa* is given in Table 2.1.

From Table 2.1 we can see that, prefix array of x, has the entries $\Pi[14] = 8$ and $\Pi[8] = 1$. Which means that $x[1 \ldots 14]$ has two prefixes; one of length 8 and the other of length 1. In the border array we have $\beta[19] = 11$, $\beta[11] = 6$, $\beta[6] = 3$ and $\beta[3] = 1$, so $x[1 \ldots 19]$ has four borders of length 11, 6, 3 and 1. Similarly, in the cover array the entries $\gamma[19] = 11$, $\gamma[11] = 6$ and $\gamma[6] = 3$ represents the three covers of length 11, 6 and 3 of x.

2.6 Degenerate strings

An degenerate string is a sequence $T = T[1]T[2] \ldots T[n]$, where $T[i] \subseteq \Sigma$ for all i, and Σ is a given alphabet of fixed size. When a position of the string is degenerate, and it can match more than one element from the alphabet Σ, we say that this position has *non-solid* symbol. If in a position we have only one element of the alphabet Σ present, then we refer to this symbol as *solid*. The definition of length for degenerate strings is the same, as for regular strings. We say that degenerate string x has a length of n, when x has n positions, where each position can be either solid or non-solid. The example in Table 2.2 identifies the solid and non-solid positions for a degenerate string.

Table 2.2 presents the degenerate string *aa*[*abc*]*a*[*ac*]*bcaa*[*ac*]*bac*[*abc*]*a*[*bc*], which has

non-solid symbols at positions 3, 5, 10, 14 and 16. The rest of the positions contains solid symbols.

2.7 Degenerate string matching

Let λ_i, $|\lambda_i| \geq 2, 1 \leq i \leq s$, be pairwise distinct subsets of Σ. We form a new alphabet $\Sigma^{'}$ $= \Sigma \cup \lambda_1, \lambda_2, \ldots, \lambda_s$ and define a new relation *match* $\approx$ on $\Sigma^{'}$ as follows:

1. for every $\mu 1, \mu 2 \in \Sigma, \mu 1 \approx \mu 2$ if and only if $\mu 1 = \mu 2$;

2. for every $\mu \in \Sigma$ and every $\lambda \in \Sigma^{'} - \Sigma, \mu \approx \lambda$ and $\lambda \approx \mu$ if and only if $\mu \in \lambda$;

3. for every $\lambda_i, \lambda_j \in \Sigma^{'} - \Sigma, \lambda_i \approx \lambda_j$ if and only if $\lambda_i \cap \lambda_j \neq \phi$

Observe that match is reflexive and symmetric but not necessarily transitive; for example, if $\lambda = a, b$, then $a \approx \lambda$ and $b \approx \lambda$ does not imply $a \approx b$.

From the example given in Table 2.2, we have a type 1 match corresponding to positions 2 and 4, as both contain the letter a. Positions 3 and 6 yield a match of type 2 as the letter b is contained in the symbol $[abc]$. A match of type 3 can be found at positions 3 and 5, as the symbols at these two positions have a and c common. Though positions 3, 6 and 3, 5 are matching positions but 5, 6 does not match, illustrating the non-transitivity of the matching operation for degenerate strings.

2.8 Quantum and deterministic match

Depending on the matching of letters, prefixes, borders and covers of degenerate strings can be of two types, namely, the *quantum match* and the *deterministic match*. Roughly speaking, a quantum match allows an non-solid symbol to match two or more distinct letters during a single matching process, whereas, a deterministic match restricts each non-

solid symbol to a single match. The notion of these two type of matching was introduced in [20].

2.9 Degenerate string matching and the bit vector technique

All string matching algorithms, needs to match the symbols at two positions. For a regular string this matching process is very simple and always requires a constant time. But matching a non-solid symbol is not so simple. In a degenerate string a non-solid position can contain up to $|\Sigma|$ symbols. So, to check whether the two positions match, we would basically need to perform a set intersection operation spending $O(|\Sigma|)$ time in the worst case. To perform this task more efficiently we use the following idea, originally borrowed from [28], and later used in [5, 3]. We use a bit vector of length $|\Sigma|$ as follows:

$$\forall T \subseteq \Sigma, \qquad \nu[T] = [\nu_{t_1}, \nu_{t_2}, \ldots, \nu_{t_{|\Sigma|}}],$$
$$\text{where } \forall t_i \in \Sigma \qquad \nu_{t_i} = \begin{cases} 1, \text{ if } t_i \in T \\ 0, \text{ otherwise} \end{cases} \tag{2.7}$$

During the string matching process instead of comparing $T[i]$ with $T[j]$, we test bit vectors $\nu[T[i]]$ and $\nu[T[j]]$ using bit operation ***AND***. Clearly, if the result of the AND operation is nonzero, only then there is a match. Thus we can compare any two symbols in constant time since the alphabet size is fixed.

2.10 Degenerate prefix and prefix array

The definition for prefix and prefix array are same for both regular and degenerate string. Depending on the matching process, a prefix of x can be defined in two ways; *quantum*

prefix and *deterministic prefix*. During a single match of two prefixes of x, in case of a quantum prefix, the non-solid positions in x can assume different letter assignments, but for a deterministic prefix the letter assignments for non-solid positions are restricted to a single assignment. As the matching operation on degenerate string is not transitive, prefix array algorithms for regular strings can not be readily extended for degenerate strings. Even with the bit matching technique the process of computing the prefix array for a degenerate string is computationally more complex than for a regular string.

2.11 Degenerate border and border array

The definition for border is same for both regular and degenerate strings. Depending on the matching process, a border of x can be defined in two ways; quantum border and deterministic border. During a single match of a border of x, in case of a quantum border, the non-solid positions in x can assume different letter assignments from Σ, but for a deterministic border the letter assignments for non-solid positions are restricted to a single assignment. But the border array of degenerate strings is totally different from the border array of a regular string. The Sequence 2.4 is not valid for degenerate strings, as is evident from the following discussion. In Table 2.3 the border array for the degenerate string $aba[ab]b$ is given. Because of the non-transitivity of the operation on degenerate letters, it is no longer true that $\beta[\beta[i]]$ would give the length of the second-longest border of $x[1 \ldots i]$. For example, if $x = aba[ab]b$, a border array $\beta = 00122$ would give correctly the longest borders of $x[1 \ldots i]$, $i \in 1 \ldots 5$, but because $\beta[\beta[4]] = 0$, it would not report the fact that actually $x[1 \ldots 4]$ has a border of length 1 in addition to its longest border of length 2. Thus the algorithms of [20, 22] need to compute a linked list at each position i of β in order to give specifically all the borders of the prefixes of the degenerate string x. In the worst case, this requires $O(n^2)$ storage. As a result any algorithm for computing the border array of a degenerate string must have a lower bound of $O(n^2)$ for both storage and running time. This fact is presented as Lemma 4.

Table 2.3: Border array of $aba[ab]b$

Index	1	2	3	4	5
$x =$	a	b	a	[ab]	b
$\beta =$	0	0	1	2	2
				1	

Lemma 4 *The lower bound for running time and space complexity of any algorithm for computing the border array of a degenerate string of length n is $\Omega(n^2)$.*

2.12 Degenerate cover and cover array

The definition for cover is same for both regular and degenerate strings. Depending on the matching process, a cover of x can be defined in two ways; *quantum cover* and *deterministic cover*. During a single match of a cover of x, in case of a quantum cover, the non-solid positions in x can assume different letter assignments from Σ, but for a deterministic cover, the letter assignments for non-solid positions are restricted to a single assignment. As the matching operation of degenerate strings is not transitive, cover array algorithms for regular strings can not be readily extended to degenerate strings. It follows from Lemma 4, that the cover array of a degenerate string does not follow Sequence 2.6 and the corresponding construction algorithm has a time and space complexity of $\Omega(n^2)$. We can state this as Lemma 5. In Table 2.4, an example is given where the Sequence 2.6 is violated and the space requirement for storing the cover array is $O(n^2)$.

Lemma 5 *The lower bound for running time and space complexity of any algorithm for computing the cover array of a degenerate string of length n is $O(n^2)$.*

From Table 2.4 and the Sequence 2.6, the degenerate string x has two covers of length 4 and 2, as $\gamma[6] = 4$ and $\gamma[4] = 2$. But from the table we can see that x has another cover of length 3 clearly violating the relation of Sequence 2.6. Note that, the space requirement for representing the cover array of x here is indeed $O(n^2)$.

Table 2.4: Border and cover array of $aba[ab][ab]a$

Index	1	2	3	4	5	6
$x =$	a	b	a	[ab]	[ab]	a
$\beta =$	0	0	1	2	3	4
				1	2	3
					1	1
$\gamma =$	0	0	0	2	3	4
					2	3

2.13 Conservative degenerate strings

A conservative degenerate string is a degenerate string where the number of non-solid symbols is bounded by a constant κ. As we will see later that in case of a conservative degenerate string, algorithms for computing prefix, border and cover arrays can achieve a much better running time and space complexity. In what follows, the corresponding data structures for conservative degenerate strings are referred to as conservative prefix (Π_c), conservative border (β_c) and conservative cover (γ_c) array respectively.

Two very important and well known algorithmic tools and data structures used by our algorithms in this thesis are the Aho-Corasick automaton and the range maxima query.

2.14 Aho-Corasick automaton

In our algorithms, we heavily use the Aho-Corasick Automaton (AC automaton) invented by Aho and Corasick in [1]. The Aho-Corasick Automaton for a given finite set P of patterns is a *Deterministic Finite Automaton* G accepting the sets of all words containing a word of P as a suffix. More formally, $G = (Q, \Sigma, g, f, q_0, F)$, where function Q is the set of states, Σ is the alphabet, g is the forward transition, f is the failure link i.e. $f(q_i) = q_j$, if and only if S_j is the longest suffix of S_i that is also a prefix of any pattern, q_0 is the initial state and F is the set of final (terminal) states [1]. The construction of the AC

automaton can be done in $O(d)$-time and space complexity, where d is the size of the dictionary, i.e. the sum of the lengths of the patterns which the AC automata will match. The pattern matching part of this algorithm has a time complexity of $O(n)$, where n is the length of the string parsed through the automata.

2.15 Range maxima (minima) query

Range Maxima (Minima) Query (Problem RMAX (RMIN)). We are given an array $A[1..n]$ of numbers. We need to preprocess A to answer the following form of queries: ***Query:*** Given an interval $[\ell..r], 1 \leq \ell \leq r \leq n$, the goal is to find the index k (or the value $A[k]$ itself) with maximum (minimum) value $A[k]$ for k in $[\ell..r]$. This query is called a *range maxima (minima) query.*

Range Maxima/Minima Query is one of the fundamental problems that could arise in different computer science problems. As a result, Problem RMAX/RMIN has received much attention in the literature not only because it is inherently beautiful from an algorithmic point of view, but also because efficient solutions to this problem often produces efficient solutions to various other algorithmic problems. It is clear that, with $O(n^2)$ preprocessing time, one can answer such queries in constant time. Interestingly, linear time preprocessing can still yield constant time query solutions and there exist several such results in the literature. To the best of our knowledge, it all started with Harel and Tarjan [19], who showed how to solve another interesting problem, namely the Lowest Common Ancestor (LCA) problem with linear time preprocessing and constant time queries. The LCA problem has as its input a tree. The query gives two nodes in the tree and requests the lowest common ancestor of the two nodes. It turns out that, solving the RMQ problem is equivalent to solving the LCA problem [18]. The Harel-Tarjan algorithm was simplified by Schieber and Vishkin [34] and then by Berkman et al. [12], who presented optimal work parallel algorithms for the LCA problem. Finally, Bender and Farach-Colton eliminated the parallelism mechanism and presented a simple serial

data structure [11]. Notably, the data structure in [11] requires $O(n \log n)$ bits of space. Recently, Sadakane [33] presented a succinct data structure, which achieves the same time complexity using $O(n)$ bits of space. In all of the above papers, there is an interplay between the LCA and the RMQ problems. Very recently, Fischer and Heun [25] presented the first algorithm for the RMQ problem with linear preprocessing time, optimal $2n+o(n)$ bits of additional space, and constant query time that makes no use of the LCA algorithm.

Chapter 3

The *iCAb* Algorithm

In this chapter, we describe our first algorithm called ***iCAb***. This algorithm computes all the covers and the cover array of a given degenerate string in a two-steps process. In the first step, the border array of the given string is computed and in the subsequent step, the Aho-Corasick pattern matching automaton is used to filter out the borders that are also covers. Using the different set of entries in the border array, ***iCAb*** can compute all the covers of a string, as well as, the corresponding cover array, both in $O(n^2)$ time in the worst case. The current best known algorithm [5, 3] for computing all the covers of a string of length n has a running time of $O(n^2 \log n)$, whereas ***iCAb*** requires $O(n^2)$ time in the worst case and $O(n)$ time in the average case to do the same task. Furthermore, ***iCAb*** can compute the cover array in $O(n^2)$ time. Notably, using the algorithms of [5, 3] one can devise an algorithm for computing the cover array. albeit, with $O(n^2 \log n)$ running time.

Before going into the details of ***iCAb***, we need to observe some properties and relationships between the border array and cover array. These properties and relationships constitute the main ideas behind ***iCAb*** and are mentioned in the following section.

Table 3.1: Border and cover array of *abaababaabaababaababa*

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	b	a
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	2	3
$\gamma =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11	0	8	0	3

3.1 Some facts

Fact 6 *Every cover of a string x is also a border of x.*

Explanation of Fact 6: From the definition of a border, we know that substring u of x is a border of x if and only if u is both a prefix and a suffix of x. Now if substring c of x is also a cover of x, then from the definition of cover, we can say that c must be a prefix and also a suffix of x. So c is also a border of x. So we can say that, if c is a cover of x, then c is also a border of x.

Suppose we are given a string $x = abaababaabaababaababa$. Here by simple observation we can see that, *aba* is both a cover and border of x, but x has another border a, which is not a cover of x. So we can say that, every cover of x is also a border but a border of x may not be a cover. For a better understanding of this concept we show the border and cover arrays of x in Table 3.1.

Correctness of Fact 6 can be easily checked from Table 3.1. From this table we can also find out some more properties about the cover array as described below.

Fact 7 *For all $i \in 1 \ldots n$, if $\beta[i] = 0$, then $\gamma[i] = 0$.*

Fact 8 *For all $i \in 1 \ldots n$, if $\beta[i] \geq \lfloor i/2 \rfloor$, then $\gamma[i] = \beta[i]$.*

Explanation of Facts 7 and 8: Fact 7 is self explanatory. If there is no border for a particular prefix $x[1 \ldots i]$ of x, then there will be no covers for that prefix. We can understand it from the example given in Table 3.1. To understand Fact 8, let us assume

that, a string y of length 15 has borders b of length 8. Now from the definition of a border, b is both a prefix and a suffix of y. So y is a superposition of b as $|y| < 2 * |b|$. For such cases there is no need to build the AC automata. We can simply set $\gamma[i] = \beta[i]$.

Two other very important relationship between the border and cover arrays were found by Y. Li and W. F. Smyth in [29]. These relationships are as follows:

Fact 9 *[29] For all $i \in 1 \ldots n$, if $\gamma[i] \neq 0$, then either $\gamma[i+1] \geq \gamma[i] + 1$ or $\gamma[i+1] = 0$.*

Fact 10 *[29] For all $i \in 1 \ldots n$, if $\beta[i+1] \leq \beta[i]$, then $\gamma[i] = 0$.*

From the proof of Facts 9 and 10 (see [26]) it is easy to realize that these two facts are also valid for degenerate strings.

For making the ***iCAb*** algorithm more efficient with respect to running time and memory usage we make use of the following fact. about the cover array is presented below:

Fact 11 *If u and c are covers of x and $|u| < |c|$ then u must be a cover of c.*

Explanation of Fact 11: If we look at our previous example, x has 3 covers of length 11, 6 and 3. They are $c_1 = abaababaaba$, $c_2 = abaaba$ and $c_3 = aba$ respectively. The length of the covers can be found easily from the cover array as was explained earlier in the preliminaries section. Now if we look at c_1, c_2 and c_3 then its easy to see that c_2 is a cover of c_1 and c_3 is a cover of c_2 and c_1. This fact is used by ***iCAb*** for efficiently computing all the covers of x. To identify a border as a cover we have to parse x through the AC automaton. This requires time proportional to the length of x. After finding the maximum length cover we can substitute x with the maximum length cover and parse that cover through the automata in place of x. As all covers must have a length less than x this approach provides substantial performance improvement.

The algorithm ***iCAb*** is built on top of the above facts and discussion. Using these properties and relationships we have devised a novel algorithm for computing the cover array of a degenerate string that outperforms all existing algorithms in the literature.

The running time analysis of our algorithm depends on the following lemma (Lemma 12), which is an extension of the results provided in [22] by Iliopoulos et al. In [22], the authors have shown that the number of borders of a regular string with don't care characters is bounded by 3.5 on average. Here, we have extended their result for degenerate strings and proved that, the expected number of borders of a degenerate string is also bounded by a constant.

Lemma 12 *The expected number of borders of a degenerate string is bounded by a constant.*

Proof. We suppose that the alphabet Σ consists of ordinary letters 1, 2, ..., α, $\alpha \geq 2$. First, we consider the probability of two symbols of a string being equal. Equality occurs in the following cases:

Case	Symbol	Match To	Number of cases
Case 1	$\sigma \in \{1, 2, \ldots, \alpha\}$	$\sigma \in \{1, 2, \ldots, \alpha\}$	α
Case 2	$\sigma = S$, $S \subseteq \Sigma$	$\sigma = S$, $S \subseteq \Sigma$, $\lvert S\rvert > 1$	$\sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$

Case 1: This case is trivial. Suppose we are trying to match symbol s_1 with symbol s_2 and both s_1 and s_2 are solid symbols. So s_1 and s_2 will match only when they are exactly equal. Given the set of all symbols in Σ there are α possible case as there are $|\Sigma| = \alpha$.

Case 2: In this case atleast one of the symbols is a non-solid symbol. Then we have to consider all possible symbol assignment for the following cases:

- *Case 1*: One of the symbols is non-solid and the other one is solid.

- *Case 2*: Both of the symbols are non-solid.

If we calculate the number of matches for this case for an alphabet of size 4 then we get the following formula:

$$\sum_{j=1}^{4} \binom{4}{j} \{2^{4-j}(2^j - 1) - \binom{j}{1}$$

Here first we have calculated all possible matches between two symbols without considering whether they are solid or non-solid. Next we have calculated the number of cases where both of the symbols are non-solid and subtracted that from the first sum. So the total number of possible matches for both Case 1 and Case 2 will be

$$4 + \sum_{j=1}^{4} \binom{4}{j} \{2^{4-j}(2^j - 1) - \binom{j}{1}$$

For an alphabet of size 4 there will be 2^{2*4} possible combinations of symbols, so the probability of matching two symbols will be

$$\frac{4 + \sum_{j=1}^{4} \binom{4}{j} \{2^{4-j}(2^j - 1) - \binom{j}{1}\}}{2^{2*4}}$$

If we now consider the same probability for an alphabet of size α, then total number of equality cases is $\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$ and the number of overall cases is $2^{2\alpha}$. Therefore the probability of two symbols of a string being equal is

$$\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}$$

Now let us consider the probability of string x having a border of length k. If we let $P[expression]$ denote the probability that the *expression* holds, then

$$P\Big[x[1\ldots k] = x[n-k+1\ldots n]\Big] = P\Big[x[1] = x[n-k+1]\Big] \times \ldots \times P\Big[x[k] = x[n]\Big]$$
$$= \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j}\{2^{\alpha-j}(2^j-1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k$$

From this it follows that the expected number of borders is

$$\sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j}\{2^{\alpha-j}(2^j-1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k$$

Let EX_b denotes the above expression. By using differentiation for finding the maximum value on the equation, $EX_b = 0$, we can find that, this summation assumes its maximum value when α is equal to 12 and the summation is bounded above by

$$\sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j}\{2^{\alpha-j}(2^j-1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k \leq 29.1746$$

So, the expected number of borders of a degenerate string is bounded by 29.1746. □

3.2 Our algorithm

Now by using the facts described above, we can compute all the covers of x from its border array. This can be done simply by checking each border and finding out whether it covers x or not. Our algorithm is based on this approach. Broadly speaking, our algorithm consists of two steps. In the first step, the deterministic border array β of x is computed. For this we have used the algorithm introduced by Holub and Smyth in [20], that can compute the deterministic border array of a degenerate string x in expected $O(n)$ time and space complexity, where n is the length of x. The deterministic border array β, stores the lengths borders for each prefix $x[1\ldots i]$ of x, as well as the assignment for the

non-solid positions for that border. In the second step, we consider each border to check whether it is a cover of x or not. Utilizing Fact 11, and the pattern matching technique of Aho-Corasick automaton, this step can be performed in $O(n)$ time and space complexity on average. In what follows, we explain the steps of the algorithm in more details.

3.2.1 First step: computing the border array

In the first step, we utilize the algorithm provided by Holub and Smyth [20] for computing the deterministic border of a degenerate string. The output of the algorithm is a array β. Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment and the this list is kept sorted in decreasing order of border lengths, b of $x[1 \dots i]$ at all time. So the first entry of β_i represents the largest border of $x[1 \dots i]$. The algorithm is given below just for completeness (Please see Algorithm 1).

If we assume that the maximum number of borders of any prefix of x is m, then the worst case running time of the algorithm is $O(nm)$. But from Lemma 12 we know that the expected number of borders of a degenerate string is bounded by a constant. As a result the expected running time of the above algorithm is $O(n)$.

3.2.2 Second step: checking the border for a cover

In the second step, we find out the covers of string x. Here we need only the last entry of the border array, $\beta[n]$, where $n = |x|$. If $\beta[n] = \{b_1, b_2, b_3\}$ then we can say that x has three borders, namely $x[1 \dots b_1]$, $x[1 \dots b_2]$ and $x[1 \dots b_3]$ of length b_1, b_2 and b_3 respectively and $b_1 > b_2 > b_3$. If the number of borders of x is m, then number of entry in $\beta[n]$ is m. We iterate over the entries of $\beta[n]$ and check each border in turn to find out whether it covers x or not. To identify a border as a cover of x we use the pattern matching technique of an Aho-Corasick automaton. Simply speaking, we build an Aho-Corasick automaton with the dictionary containing the border $x[1 \dots b]$ of x and parse x through the automaton

Algorithm 1 Computing deterministic border array of string x

1: $\beta_i[j] \leftarrow \phi, \qquad \forall\, i, j \in \{1 \ldots n\}$

2: $\nu_i \leftarrow \nu[x[i]], \qquad \forall\, i \in \{1 \ldots n\} \qquad$ {set bit vector for each $x[i]$}

3: **for** $i \leftarrow 1$ to $n - 1$ **do**

4: **for all** b, $\beta_i[b] \neq \phi$ **do**

5: **if** $2b - i + 1 < 0$ **then**

6: $p \leftarrow \nu_{b+1}$ ***AND*** ν_{i+1}

7: **else**

8: $p \leftarrow \beta_i[b + 1]$ ***AND*** ν_{i+1}

9: **end if**

10: **if** $p \neq \mathbf{0}^{|\Sigma|}$ **then**

11: $\beta_{i+1}[b + 1] \leftarrow p$

12: **end if**

13: **end for**

14: $p \leftarrow \nu_1$ ***AND*** ν_{i+1}

15: **if** $p \neq \mathbf{0}^{|\Sigma|}$ **then**

16: $\beta_{i+1}[1] \leftarrow p$

17: **end if**

18: **end for**

to find out whether x can be covered by it or not. Suppose in iteration i, we have the length of the i-th border of $\beta[n]$ equal to b. In this iteration, we build an Aho-Corasick automaton for the following dictionary:

$$D = \{x[1]x[2]\dots x[b]\}, \qquad \text{where } \forall j \in \text{ 1 to b , } x[j] \in \Sigma \tag{3.1}$$

Then we parse the input string x through this automaton to find out the positions where the pattern $c = x[1 \dots b]$ occurs in x. We store the starting index of the occurrences of c in x in a position array P of size $n = |x|$. We initialize P with all entries set to zero. If c occurs at index i of x then we set $P[i] = 1$. Now if the distance between any two consecutive 1's is greater than the length of the border b then the border fails to cover x, otherwise c is identified as a cover of x. We store the length of the covers in an array AC. At the end of the process, AC contains the length of all the covers of x. The definition of AC can be given as follows:

$$AC = \{c_1, c_2 \dots c_k\}, \qquad \text{where } \forall i \in 1 \text{ to k, } c_i \text{ is a cover of } x \tag{3.2}$$

Algorithm 2 formally presents the steps of a function $isCover()$, which is the heart of Step 2 described above.

In Step 4 of Algorithm 2, an Aho-Corasick automaton for the substring c is constructed, which is later used to find out whether c is a cover of x. For a better understanding of Algorithm 2 below we briefly describe an Aho-Corasick automaton [2].

An Aho-Corasick automaton for a given set P of patterns is a (deterministic) finite automaton G accepting the set of all words containing a word of P as a suffix.

G consists of the following components:

- finite set Q of states

Algorithm 2 Function isCover(x, c)

1: **if** $|c| \geq \lfloor |x|/2 \rfloor$ **then**
2: ***RETURN*** *TRUE*
3: **end if**
4: Construct the Aho-Corasick automaton for c
5: parse x and compute the positions where c occurs in x and put the positions in the array Pos
6: **for** $i = 2$ to $|Pos|$ **do**
7: **if** $Pos[i] - Pos[i-1] > |c|$ **then**
8: ***RETURN*** *FALSE*
9: **end if**
10: **end for**
11: ***RETURN*** *TRUE*

- finite alphabet Σ
- transition function $g: \ Q \ \ \Sigma -> \ Q \ + \ fail$
- failure function $h: \ Q -> \ Q \ + \ fail$
- initial state q_0 in Q
- a set F of final states

The construction of the Aho-Corasick automaton consists of two stages. In the first stage (Algorithm 3) a trie for the set of patterns is build and the transition function g is defined for each node of the trie. In this algorithm the following assumptions are made;

- $m[p]$ = length of $pat[p]$
- num_{pat} = number of patterns
- $g(Node, Character)$ is the transition function

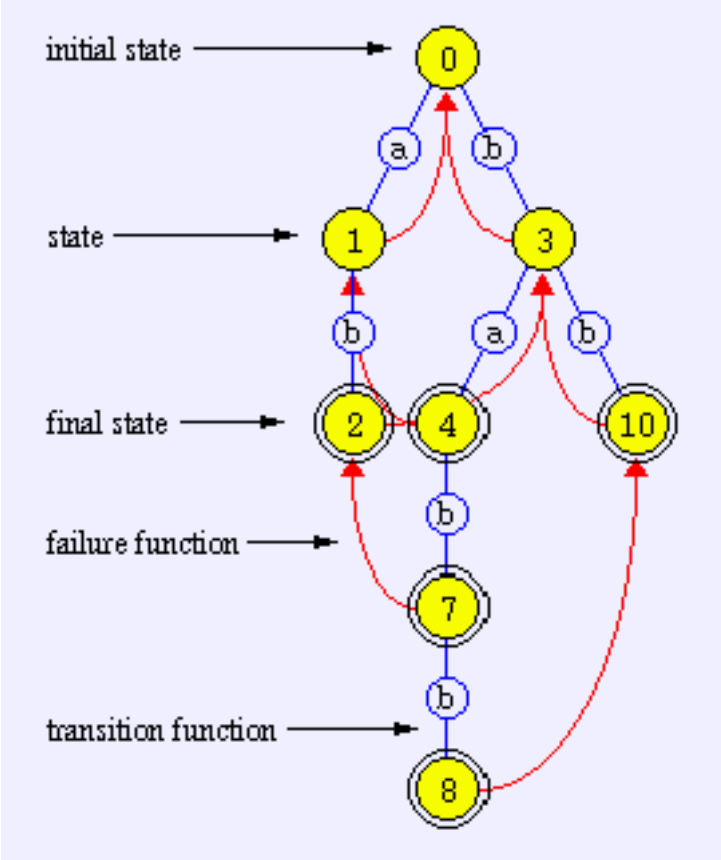

Figure 3.1: Aho-Corasick Automata with pattern set $P = \{ab, ba, babb, bb\}$

In the second stage the failure function is computed. These algorithm is outlines as algorithm 4.

The next step is the parsing od the given text through the finite automata. This step is described in Algorithm 5.

The construction and parsing of text, x through the Aho-Corasick automaton has both time and space complexity of $O(n)$, where $n = |x|$. So the time and space complexity of Algorithm 2 can be deduced as follows. Clearly, Steps 5 and 6 run in $O(n)$. Now, the complexity of Step 4 is linear in the size of the dictionary on which the automaton is build. Here the length of the string in the dictionary can be $n - 1$ in the worst case. So, the time and space complexity of this algorithm is $O(n)$.

A further improvement in running time is achieved as follows. According to Fact 11,

Algorithm 3 Computation of the trie

```
1: final ← φ
2: for p = 1 to num_pat do
3:    q ← root
4:    for j = 1 to m[p] do
5:       if g(q, pat[p][j]) = null then
6:          insert(q, pat[p][j])
7:       end if
8:       q ← g(q, pat[p][j])
9:    end for
10:   final ← final ∪ q)
11: end for
```

if u and c are covers of x and $|u| < |c|$ then u must be a cover of c. Now if $\beta[n] = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \ldots > b_n$. Now if in any iteration we find a b_i that is a cover of x then from Fact 11, we can say that for all $j \in i+1 \ldots m$, b_j is a cover of x if and only if b_j is a cover of b_i. So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| < |x|$ this policy improves the overall running time of the algorithm, albeit, not asymptotically. Algorithm 6 formally presents the overall process of Step 2 described above.

The running time of Algorithm 6 is $O(nm)$, where m is the number of borders of x or alternatively the number of entries in β_n. Again, from Lemma 12, we can say that the number of borders of a degenerate string is bounded by a constant on average. Hence, the expected running time of Algorithm 6 is $O(n)$.

It follows from above that our algorithm for finding all the covers of a degenerate string of length n runs in $O(n)$ time on the average. The worst case complexity of our algorithm is $O(nm)$, i.e., $O(n^2)$, which is also an improvement since the current best known algorithm [5] for finding all covers requires $O(n^2 \log n)$ in the worst case.

Algorithm 4 Computation of the failure function

1: $h(root) \leftarrow fail$
2: **for all** $q^{'} \mid parent(q^{'}) = root$ **do**
3: $h(q^{'}) \leftarrow root$
4: **end for**
5: **for all** $q^{'} \mid depth(q^{'}) > 1$ *in BFS order* **do**
6: $q \leftarrow parent(q^{'})$
7: $c \leftarrow$ *the symbol with* $g(q, c) = q^{'}$
8: $r \leftarrow h(q)$
9: **while** $r \neq fail$ ***AND*** $g(r, c) = fail$ **do**
10: $r \leftarrow h(r)$
11: **end while**
12: **if** $r = fail$ **then**
13: $h(q^{'}) = g(root, c)$
14: **else**
15: $h(q^{'}) = g(r, c)$
16: **if** $isElement(h(q^{'}), final)$ **then**
17: $final \leftarrow final \cup q^{'}$
18: **end if**
19: **end if**
20: **end for**

3.3 Computing cover array

As pointed out in Chapter 2, the cover array γ of a degenerate string is an array of lists whose worst case space complexity can be $O(n^2)$. Here γ is represented as a array of list where the lists containing all the cover lengths for every prefix of x, that is for all $i \in 1 \ldots n$, $\gamma[i]$ contains a list of integers representing the cover lengths of $x[1 \ldots i]$ if there is any, otherwise $\gamma[i]$ contains an empty list. Our algorithm can readily be extended to

Algorithm 5 Parsing text x through the automata

```
1: q ← root
2: for i = 1 to n do
3:    while q ≠ fail AND g(q, x[i]) = fail do
4:       q ← h(q)
5:    end while
6:    if q = fail then
7:       q ← root
8:    else
9:       q ← g(q, x[i])
10:   end if
11:   if isElement(q, final) then
12:      RETURN TRUE
13:   end if
14: end for
15: RETURN FALSE
```

compute the cover array of x. Algorithm 6 can be used here after some modification to compute the cover array of x.

Here we need to process all the borders of each prefix of x and as a result there will be a lot of repetitions in the border lengths. This can be seen from the example given in Table 3.2. There is a cover of length 2 at both positions 4 and 5 and cover of length 3 at positions 5 and 6. In Algorithm 6, separate Aho-Corasick automata were built for each entry in the list β_n as there was no possibility of repetitions. For avoiding building an Aho-Corasick automaton for the same border length twice, we change our approach of Algorithm 6, and build a single Aho-Corasick automaton for the maximum border length, $MaxBorder$ and mark the final stages for each border length separately and separate position arrays Pos_i's are kept for keeping track of the already covered prefix of x. Whenever a final stage is reached, the corresponding position array is updated

Algorithm 6 Computing all covers of x

1: $k \leftarrow n$
2: $AC \leftarrow \phi$ {AC is a list used to store the covers of x}
3: **for all** $b \in \beta_n$ **do**
4: **if** $isCover(x[1..k], x[1..b]) = true$ **then**
5: $m \leftarrow b$
6: $AC.Add(k)$
7: **end if**
8: **end for**

Table 3.2: Border and cover array of $aba[ab][ab]a$

Index	1	2	3	4	5	6
$x =$	a	b	a	[ab]	[ab]	a
$\beta =$	0	0	1	2	3	4
				1	2	3
					1	1
$\gamma =$	0	0	0	2	3	4
					2	3

accordingly. That is if the final stage reached by the current symbol of x is marked with a border length of k then position array Pos_k is updated in the same way as was done in Algorithm 6. An outline of the final algorithm is given in Algorithm 7.

Step 4 of Algorithm 7, constructs the extended Aho-Corasick automata discussed in the previous paragraph. This step requires $O(n^2)$ processing time in the worst case, as we need to mark final states for different border lengths. In Step 5 string x is parsed and the Pos arrays are updated accordingly. There are $MaxBorder$ number of Pos arrays and each array has n entries. So the running time of this step is $O(n.MaxBorder)$. Now the maximum value of $MaxBorder$ can be n, so the running time and space complexity of this step is $O(n^2)$.

In Algorithm 7, we iterate over the entries of the border array β and checks the borders

Algorithm 7 Computing cover array γ of x

1: $\gamma[i] \leftarrow 0 \quad \forall\, i \in \{1 \ldots n\}$
2: $\delta[i] \leftarrow 1 \quad \forall\, i \in \{1 \ldots MaxBorder\}$
3: $\alpha[i] \leftarrow n \quad \forall\, i \in \{1 \ldots MaxBorder\}$
4: Construct the Aho-Corasick automaton for $x[1 \ldots MaxBorder]$
5: parse x and compute the positions where borders occurs in x and put the positions in the array, Pos_i
6: **for** $i \leftarrow 1$ *to* n **do**
7: **for all** $b \in \beta_i$ **do**
8: **if** $\alpha[b] > i$ **then**
9: $lastIndex = 1$
10: **for** $j \leftarrow \delta[b]$ *to* i **do**
11: **if** $Pos_b[j] \neq 0$ **then**
12: **if** $Pos_b[j] - Pos_b[lastIndex] > b$ **then**
13: $\alpha[b] = i$
14: $Break$
15: **end if**
16: $lastIndex = j$
17: **end if**
18: **end for**
19: **if** $j \geq i$ **then**
20: $\delta[b] = j$
21: $\gamma[i].ADD(b)$
22: **end if**
23: **end if**
24: **end for**
25: **end for**

one by one. If the border $x[1 \ldots b]$ is identified as a cover then b is added to the list $\gamma[i]$, otherwise the algorithm goes on to processing the next border. Here two arrays, δ and α are maintained for avoiding re-computation of the same information. Both of these arrays have a length of $MaxBorder$. The array δ, keeps track of the length of the already covered portion of x. Whenever we find a cover in Step 20 the update the corresponding entry in the δ array. This value is later used in Step 10 to initialize the loop variable j, which ensures that each entry of the Pos array is accessed exactly once.

The array α, is used for marking the end positions for each cover length, e.i. if $\alpha[4] = 8$, then a cover of length 4 is possible only up to length 8 of x; any prefix of x having a length greater than 8 can not have a cover of length 4. Initially all the entries of α are initially to n. Whenever we find that a cover of a certain length can not exist beyond a particular length of x, we update the α array in Step 13. In Step 8 we find out whether a cover of length b, is possible for the current prefix length i, by checking $\alpha[b]$. If $\alpha[b]$ is greater or equal to i, only then we proceed to the next step of the algorithm; otherwise a cover of length b is not possible for $x[1 \ \ldots \ i]$ and we proceed to the next entry of the border array.

Using these two arrays the algorithm ensures that, each entry of the Pos arrays are accessed exactly once. There are $MaxBorder$ number of Pos arrays and each array has n entries. So the running time of Algorithm 7 is $O(n.MaxBorder)$. Now the maximum value of $MaxBorder$ can be n, so the overall running time and space complexity of this algorithm is $O(n^2)$.

3.4 Summary

To summarize, algorithm ***iCAb*** has a overall running time and space requirement of $O(n^2)$. Among all the previous algorithms for computing the cover array [5, 3] the best known algorithm [3] have a worst case running time of $O(n^2 \log n)$ and this algorithm uses a complex data structure called the *vEB tree* [36]. The algorithm in [5] can only

compute the conservative cover array γ_c in $O(n^3)$ time. Here we have provided an elegant and novel solution for computing the cover array γ in $O(n^2)$ time and space complexity. Furthermore, we have provided another (Algorithm 6) for computing all the covers of a string of length n with a worst case $O(n^2)$ time and space complexity. The algorithm provided in [5] can compute all covers of a string, but its running time is $O(n^2)$ even in the average case, whereas the average case running time of Algorithm 6 is just $O(n)$.

In the next chapter, we describe our second algorithm called ***iCAp***, which does the same task as ***iCAb*** but has a better space usage and avoids the use of an Aho-Corasick automata. The algorithm ***iCAp*** starts with computing the prefix array of x and then gradually computes the cover array of x from the prefix array.

Chapter 4

The *iCAp* Algorithm

In this chapter, we describe our second algorithm called ***iCAp***. This algorithm starts with the prefix array of x. The notion of the prefix array for a regular string was first introduced by Chochemore et al. in [15]. Here we first show that the prefix array algorithm for regular strings is not valid for degenerate strings and then provide an algorithm that can compute the prefix array for a degenerate string. After computing the prefix array Π, ***iCAp*** computes the border array β of x from the prefix array in a much more efficient manner than the technique shown in [20]. In the next step, ***iCAp*** computes the cover array γ using the prefix array and the border array. The algorithm ***iCAp*** has a worst case time and space complexity of $O(n^2)$.

4.1 Our algorithm

The *icap* algorithm consists of four steps. An overview of these steps is described as follows:

- *Step 1*: Compute the prefix array Π of the given degenerate string x.
- *Step 2*: Preprocess the computed prefix array Π, so that range maxima queries (RMQ) on this array can be answered in constant time.

- *Step 3*: Compute the border array β of x from the prefix array Π.
- *Step 4*: Using Π and β, compute the cover array γ of x.

In the subsequent sections, these steps are explained in greater details.

4.1.1 Step 1: computing the prefix array Π

In this step of ***iCAp***, the prefix array of the given degenerate string x is computed. Unlike the border and cover array, the prefix array is a linear space data structure for both regular and degenerate strings. First, as a starting ground, we present below the prefix array algorithms from [35] (we assume zero-based indexing of arrays for this algorithm).

Algorithm 8 Computing prefix array Π of a regular string x of length n

1: $\Pi[0] \leftarrow n$
2: $g \leftarrow 0$
3: **for** $i \leftarrow 1$ *to* $n - 1$ **do**
4: **if** $i < g$ ***AND*** $\Pi[i - f] \neq g - i$ **then**
5: $\Pi[i] \leftarrow min(\Pi[i - f],\ g - i)$
6: **else**
7: $(g, f) \leftarrow (max(g,\ i),\ i)$
8: **while** $g < n$ ***AND*** $x[g] = x[g - f]$ **do**
9: $g \leftarrow g + 1$
10: **end while**
11: $\Pi[i] \leftarrow g - f$
12: **end if**
13: **end for**

Here comparisons between characters is performed in Step 9. Every comparison between equal characters increments the variable g. As the value of g never decreases and it varies from 0 to n, there are at most n positive comparisons. Each negative comparison

leads to the next step of the loop; then, there are at most $n-1$ of them. Thus less than $2n$ comparisons. So the total running time of this algorithm is $\theta(n)$.

But we can't extend this algorithm for degenerate strings. If we look at Step 4, then there is a comparison between the current and already computed portion of the Π array. Now this step depends critically on the transitivity property of the matching function. But transitivity relation does not hold for degenerate strings. The situation can be better understood with an example.

Figure 4.1: Illustration of Algorithm 8

From Figure 4.1, we can see that the framed substrings $x[6 \ldots 12]$ and $x[14 \ldots 18]$, and the gray substrings $x[9 \ldots 10]$ and $x[17 \ldots 20]$ are prefixes of string $x = abbabaabbabaaaabbabbaa$. For $i = 9$ (i.e., the 9th iteration of the for loop at Line 3 of Algorithm 8), we have $f = 6$ and $g = 13$. The situation at this position is the same as at position $3 = 9 - 6$. We have $\Pi[9] = \Pi[3] = 2$ which means that ab, of length 2, is the longest factor at position 9 that is a prefix of x. For $i = 17$, we have $f = 14$ and $g = 19$. As $\Pi[17 - 14] = 2 = 19 - 17$, we deduce that the string $ab = x[i \ldots g - 1]$ is a prefix of x. In these kind of situations, at Line 4, the already computed values of Π is simply copied in the current index without performing any character comparison at all. It turns out that for these kind of situations there may exist positions $I > i$ such that $x[I] = x[1]$ and $I < g$; in such cases, there is a substring beginning at I that matches a substring beginning at position $I' < i$ (hence already computed) that in turn matches a prefix of x. For all such I and I' Algorithm 8 copies the already computed values corresponding to I' into the value corresponding to I. This operation depends critically on the transitivity of the matching ($a = b$ and $b = c$ implies $a = c$), a property always true for regular strings, but not valid for degenerate strings. For degenerate strings if we have $[abcd] \approx [ab]$ and $[abcd] \approx [cd]$, then we can't say

that $[ab] \approx [cd]$. So while Algorithm 8 yields a very efficient implementation of prefix array for regular strings we can't extend this algorithm for degenerate strings. In fact, we can safely state that any algorithm that uses pre-computed information of letter matching, can't be readily extended for degenerate string.

For computing the prefix array from a degenerate string we have to compare each pair of positions separately. We can't skip comparison using previously computed values. So the algorithm for computing the prefix array is the most trivial one which performs $O(n^2)$ comparisons and computes the longest common prefix length for each prefix of x. The algorithm is presented below.

Algorithm 9 Computing prefix array Π of a degenerate string x of length n

```
1: Π[1] ← 0
2: for i ← 2 to n do
3:    j = 0
4:    while j < n AND x[i + j] = x[j + 1] do
5:       j ← j + 1
6:    end while
7:    Π[i] ← j
8: end for
```

In the border array algorithm of the previous chapter, we saved the character assignments for non-solid positions along with the border lengths. But here we the character assignments for non-solid positions are not saved, as we are finding the longest common prefixes for all suffixes of x by matching each position individually. So character assignments for non-solid positions can be computed simply by taking the ***AND*** of the vector maps of two corresponding positions spending only $O(1)$ time, since the alphabet is fixed.

4.1.2 Step 2: preprocessing of the prefix array for RMAX

In this step, the prefix array is preprocessed so that the range maxima queries can be answered on this array in constant time per query. In the final step of ***iCAp*** when we will compute the cover array from the border array and prefix array we will need to find out the maximum value (and the corresponding index) within a range of the prefix array. To perform this operation in constant time we preprocess the prefix array in this step using one of the algorithms of [19, 34] spending O(n) time. So the overall running time of Step 2 is $O(n)$.

4.1.3 Step 3: computing the border array

In the previous chapter we computed the border array using the algorithm provided by Holub and Smyth in [20]. But the information provided by the border array can also be derived from the prefix array of any string. We first present the following fact that holds for any border array regardless of the underlying alphabet of x.

Fact 13 *If for some $i \in 2 \ldots n, \beta[i] > 0$, then $x[1 \ldots i-1]$ has a border of length $\beta[i]-1$.*

Fact 13 implies that all the borders of the prefixes of x occur in arithmetic sequence fully determined by the maximum border length k; more precisely, if the maximum border length of $x[1 \ldots i]$ is $k > 0$, then

$$x[1 \ldots i-1], x[1 \ldots i-2], \ldots x[1 \ldots i-k+1] \tag{4.1}$$

must have borders of (not necessarily maximum) lengths

$$k-1, k-2, \ldots, 1$$

Table 4.1: Computing border array from the prefix array of $aba[ab]b$

Index	1	2	3	4	5
$x =$	a	b	a	[ab]	b
$\Pi =$	0	0	2	2	0
$\beta =$	0	0	1	2	2
				1	

respectively. Thus to describe all the borders of every nonempty prefix of x, it suffices to specify only the maximum border k at each position $i > 1$. Since for each factor of the Sequences 4.1, we have $x[i-k+1] = x[1]$, it becomes clear that to describe all the borders of every prefix of x, it suffices to specify at every position $j \in 2 \ldots n$, the length $k > 0$ of the longest substring $x[j \ldots j+k-1] = x[1 \ldots k-1]$, i.e., the longest substring that matches a prefix of x. Based on the above discussion, we can claim the following

Lemma 14 *The prefix array* Π *describes all the borders of every prefix of* x*.*

Because of the non-transitivity of the match operation on degenerate letters, it is no longer true that $\beta[\beta[i]]$ gives the length of the second-longest border of $x[1 \ldots i]$. For example, if $x = aba[ab]b$, a border array $\beta = 00122$ would give correctly the longest borders of $x[1 \ldots i]$, $i \in 1 \ldots 5$, but because $\beta[\beta[4]] = 0$, it would not report the fact that actually $x[1 \ldots 4]$ has a border of length 1 in addition to its longest border of length 2. Thus the algorithms given in [20, 22] need to compute a linked list at each position i of β in order to specifically give all the borders of the prefixes of the degenerate string x. In the worst case, this requires $O(n^2)$ storage. However, due to the validity of Lemma 14 for both regular and degenerate strings, the computation of Π rather than β eliminates the requirement for the list for a degenerate string, the nonzero positions $i > 0$ in Π will be those for which $x[i] \approx x[1]$, and the values of these $\Pi[i]$ will be sufficient to specify all the borders of all the prefixes of x, just as in the degenerate case, with no additional storage required. For the example given in Table 4.1, it would suffice to return $\Pi = 00220$, as from this Π all the borders of each prefix of x can be specified.

From Table 4.1, an outline of the algorithm for finding the border array from the prefix array can be made immediately. As we can see in the table, the prefix of length 2 at position 4 contributes a border of length 2 at position 5 and a border of length 1 at position 4. In the same way the prefix of length 2 at position 3 contributes a border of length 2 at position 4 and a border of length 1 at position 3. So at position 4 we have 2 borders.

The algorithm for computing the border array from the prefix array in given as Algorithm 10. Here the border array is an array of lists. Each list contains the lengths of the borders for the corresponding prefix of x.

Algorithm 10 Computing border array β of a degenerate string x of length n

```
1: for i ← 1 to n do
2:     β[i] ← φ
3: end for
4: for i ← 2 to n do
5:     MaxBorderLength ← Π[i]
6:     j ← i + MaxBorderLength − 1
7:     while j ≥ i do
8:         β[j] ← β[j] ∪ MaxBorderLength
9:         MaxBorderLength ← MaxBorderLength − 1
10:        j ← j − 1
11:    end while
12: end for
```

For the example in Table 4.1, it is clear that there can be multiple entries in each list of the border array for a degenerate string. In the worst case there can be n entries in each of the lists of β. So the worst case running time of Algorithm 10 is $O(n^2)$ and it requires $O(n^2)$ space to store the entries of the border array β.

4.1.4 Step 4: computing the cover array

In the final step of ***iCAp***, the cover array of x is computed from the prefix and border arrays. The possible cover lengths are collected from the border array and the prefix array is used to find out which among them are actually covers. The algorithm also keeps another array δ, where for each possible cover length the already covered length of x is stored. The possible cover length are from 1 to the maximum prefix length ($MaxPrefix$) in the prefix array. The maximum value of $MaxPrefix$ can be n. The possible values of covers are actually the index of δ so any value can be updated in constant time. We present this algorithm as Algorithm 11.

In Algorithm 11, the cover array of x is computed. As we already know the cover array is two dimensional data structure and any algorithm for computing it has a lower bound of $O(n^2)$. Here at Step 14, the RMAX queries on Π is performed. The length of the already covered portion of x is stored in the array δ for each possible cover length. At Step 14, the algorithm tries to extend the already covered length as much as possible.

Now assume that we are considering the case for a cover c of length m and the worst case for this algorithm will appear when x will have a cover of length m. For finding whether c is a cover of x or not we will have to perform n/m number of RMAX queries on Π. Now if the maximum cover length is max_c then total number of RMAX queries can be represented by the following equation:

$$Total\ number\ of\ RMAX\ queries\ = n\ +\ n/2\ +\ n/3\ +\ \ldots\ +\ n/max_c \qquad (4.2)$$

The maximum value of max_c can be n in the worst case. So the worst case total number of RMAX queries will be

$$Total\ number\ of\ RMAX\ queries\ (worst\ case)\ = n\ +\ n/2\ +\ n/3\ +\ \ldots\ +\ 1 \qquad (4.3)$$

Algorithm 11 Computing cover array γ of a degenerate string x of length n

```
1: for i ← 1 to n do
2:    γ[i] ← 0
3: end for
4: for i ← 1 to MaxPrefix do
5:    δ[i] ← 0
6: end for
7: for i ← 2 to n do
8:    for all b ∈ β[i].LIST() do
9:       if b ≥ ⌊i/2⌋ then
10:         γ[i].ADD(b)
11:      else
12:         StartIndex ← δ[b]
13:         while StartIndex < i do
14:            (MaxValue, MaxIndex) = RMAX(StartIndex − b + 2, StartIndex +
               1)
15:            if MaxValue ≥ b then
16:               δ[b] = MaxIndex + b
17:               γ[i].ADD(b)
18:            end if
19:         end while
20:      end if
21:   end for
22: end for
```

Taking n common from Sequence 4.3, we get the sequence $1 + 1/2 + 1/3 + \ldots + 1/n$ which is known as the sequence of harmonic numbers and the result of the summation is $O(\log n) + a\ small\ constant$. So the summation of Sequence 4.3 has a upper bound of $O(n \log n)$. As we are avoiding recomputing the RMAX query on the same interval

Table 4.2: Worst case running time ***iCAp***

Step of algorithm	**Running Time**
Step 1	$O(n^2)$
Step 2	$O(n)$
Step 3	$O(n^2)$
Step 4	$O(n \log n)$

twice by storing the result in the array δ the overall running time of Algorithm 11 will be $O(n \log n)$ and the space requirement will be $O(n^2)$.

The worst case running time of each step of ***iCAp*** is described in Table 4.2.

4.2 Summary

In this chapter we have presented the algorithm ***iCAp***, which has an overall running time and space requirement of $O(n^2)$. This complexity is due to Step 1 and Step 3 where we compute the prefix and border array. Adapting the RMQ algorithm to perform the range lookups has provided an efficient and elegant way to compute the cover array from the prefix array.

Among all the previous algorithm for computing the cover array [5, 3] the best known algorithm [3] has a worst case running time of $O(n^2 \log n)$ and this algorithm uses a complex data structure called the *vEB tree* [36]. The algorithm in [5] can only compute the conservative cover array γ_c in $O(n^3)$ time. Here we have provided an elegant and novel solution for computing the cover array γ in $O(n^2)$ time and space complexity without applying any restriction on the type of covers. Our algorithm can be adjusted to compute both the quantum and deterministic cover array of x without sacrificing running time.

Chapter 5

Conclusion

In this last chapter, we draw the conclusion of our thesis by describing the major contributions made through this research followed by some directions for future research.

5.1 Major contributions

The contributions that have been made in this thesis can be enumerated as follows:

- Here we have proved that expected number of borders of a degenerate string in bounded by a constant. As each cover must also be a border so the expected number of covers of a degenerate string is also bounded by a constant.
- In case of regular strings a linear border array encodes all the borders of every prefix of string x. Here we have shown that the same is not true for degenerate strings. The border array of a degenerate sting is a two dimensional data structure where each entry of β is a sorted list containing the border lengths for its corresponding prefix of x.
- Similarly for a regular strings a linear cover array describes all the covers of every prefix of string x. Here we have shown that its not the case for degenerate strings.

Table 5.1: Performance comparison for computing all cover of x

Algorithm	**Running Time**	**Space Requirement**
Conservative String Covering (too restricted) [3]	$O(n^2)$	$O(n^2)$
Antonious [5]	$O(n^2 \log n)$	$O(n^2)$
iCAb	$O(n^2)$ Average case $O(n)$	$O(n^2)$ Average case $O(n)$
iCAp	$O(n^2)$	$O(n^2)$

The cover array of a degenerate sting is also a two dimensional data structure where each entry of γ is a sorted list containing the cover lengths for its corresponding prefix of x.

- We have provided an algorithm called ***iCAb*** which can compute the border and cover array of x having a overall running time and space requirement of $O(n^2)$. Among all the previous algorithm for computing the cover array [5, 3] the best known algorithm [3] have a worst case running time of $O(n^2 \log n)$ and this algorithm uses a complex data structure called the *vEB tree*. The algorithm in [5] can only compute the conservative cover array γ_c while having a worst case running time of $O(n^3)$. We have discussed an extension of ***iCAb*** for computing all the covers of a string of length n with expected $O(n)$ time and space complexity. The algorithm provided in [5] can also compute all covers of a string, but its running time is $O(n^2)$ even in the average case.

- Another algorithm is provided in this thesis which is called ***iCAp***. This algorithm computes the prefix array, border array and cover array for degenerate strings. The running time of this algorithm is also $O(n^2)$. But this algorithm avoids the use of an Aho-Corasick pattern matching automata and computed the cover array using only simple data structures like array and lists. Performance comparison of ***iCAb*** and ***iCAp*** with current best known algorithm is given in Table 5.1 and Table 5.2.

Table 5.2: Performance comparison for computing the cover array of x

Algorithm	**Running Time**	**Space Requirement**
Conservative String Covering (too restricted) [3]	$O(n^3)$	$O(n^2)$
Antonious [5]	$O(n^2 \log n)$	$O(n^2)$
iCAb	$O(n^2)$	$O(n^2)$
iCAp	$O(n^2)$	$O(n^2)$

5.2 Future directions for further research

A number of new questions and issues has popped out of our research. A few of them has been listed below.

1. As has been mentioned previously, one of the motivation of our work follows from lack of good pattern matching algorithms for degenerate strings. For regular strings, famous pattern matching algorithms like KMP [26], Boyer-Moore [13] are dependent on the information provided by regularities like the border array or prefix array. The algorithms provided in this research may work as building blocks for efficient and up to the mark pattern matching algorithm for degenerate strings.

2. The Aho-Corasick pattern matching automata can be modified for degenerate strings so that it can match non-solid positions of the pattern directly. This may provide a new research direction for devising efficient pattern matching algorithms.

3. As the border and cover array encoding scheme for regular strings is not suitable for degenerate strings, new encoding schemes may be thought of which can provide a linear space representation of these arrays; opening the scope of reducing the running time of algorithms for computing them to a great extent.

4. Duval's algorithm for Lyndon word decomposition [16] has a close relation to the border array. An extension of this algorithm for degenerate strings can be an interesting research topic.

5. Other famous repetition algorithms like Main and Lorentz's [30, 31] repetition algorithms can be considered as good candidates for future research for degenerate strings.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[3] Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, and Gad M. Landau. Conservative string covering of indeterminate strings. *Proceedings of the Prague Stringology Conference 2008*, pages 108–115, 2008.

[4] Pavlos Antoniou, Costas S. Iliopoulos, Inuka Jayasekera, and M. Sohel Rahman. Implementation of a swap matching algorithm using a graph theoretic model. In *BIRD*, pages 446–455, 2008.

[5] Pavlos Antoniou, Costas S. Iliopoulos, Inuka Jayasekera, and Wojciech Rytter. Computing repetitive structures in indeterminate strings. *Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008)*, 2008.

[6] Pavlos Antoniou, Costas S. Iliopoulos, Laurent Mouchard, and Solon P. Pissis. Practical and efficient algorithms for degenerate and weighted sequences derivedfrom high throughput sequencing technologies. In *Proceedings of the International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing (IJCBS 2009)*, 2009.

[7] Alberto Apostolico and Dany Breslauer. An optimal o(loglog n)-time parallel algorithm for detecting all squares in a string. *SIAM Journal of Computing*, 25(6):1318–1331, 1996.

[8] Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(11):17–20, 1991.

[9] Alberto Apostolico and Franco P. Preparata. Optimal off-line detection of repetitions in a string. *Theory of Computer Science*, 22:297–315, 1983.

[10] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[11] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Theoretical INformatics (LATIN)*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[12] Omer Berkman, Dany Breslauer, Zvi Galil, Baruch Schieber, and Uzi Vishkin. Highly parallelizable problems (extended abstract). In *STOC*, pages 309–319. ACM, 1989.

[13] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[14] Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[15] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, Cambridge, 2007.

[16] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[17] Michael J. Fischer and Michael S. Paterson. String-matching and other products. *Technical report, Cambridge, MA, USA*, 1974.

[18] H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Symposium on the Theory of Computing (STOC)*, pages 135–143, 1984.

[19] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[20] Jan Holub and William F. Smyth. Algorithms on indeterminate strings. *Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03*, pages 36–45, 2003.

[21] Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50, 2008.

[22] Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina G. Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. *Nordic Journal of Computing*, 10(1):40–51, 2003.

[23] Costas S. Iliopoulos, Laurent Mouchard, and Mohammad Sohel Rahman. A new approach to pattern matching in degenerate dna/rna sequences and distributed pattern matching. *Mathematics in Computer Science*, 1(4):557–569, 2008.

[24] Costas S. Iliopoulos and M. Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009.

[25] Volker Heun Johannes Fischer. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In Bo Chen and Guochuan Zhang, editors, *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.

[26] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[27] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[28] I. Lee, A. Apostolico, C. S. Iliopoulos, and K. Park. Finding approximate occurrences of a pattern that contains gaps. In *Proceedings 14-th Australasian Workshop on Combinatorial Algorithms*, pages 89–100. SNU Press, 2003.

[29] Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.

[30] Michael G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[31] Michael G. Main and Richard J. Lorentz. An o(n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

[32] Dennis Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994.

[33] Kunihiki Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[34] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[35] William F. Smyth and Shu Wang. New perspectives on the prefix array. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *SPIRE*, volume 5280 of *Lecture Notes in Computer Science*, pages 133–143. Springer, 2008.

[36] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[37] Sun Wu and Udi Manber. Agrep – a fast approximate pattern-matching tool. *In Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA*, pages 153–162, 1992.

[38] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.