2011

Parameterized Strings: Algorithms and Data Structures

Richard A. Beal
West Virginia University

Parameterized Strings: Algorithms and Data Structures

by

Richard A. Beal

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Arun A. Ross, Ph.D.
Elaine M. Eschen, Ph.D.
Donald A. Adjeroh, Ph.D., Chair

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2011

Keywords: parameterized suffix array, parameterized suffix sorting, parameterized longest common prefix, structural match, p-string, s-string, p-match, s-match, arithmetic coding, fingerprints, LPF, LCP

Abstract

Parameterized Strings:
Algorithms and Data Structures

by

Richard A. Beal

A parameterized string (p-string) $T = T[1]T[2]...T[n]$ is a sophisticated string of length n composed of symbols from a constant alphabet Σ and a parameter alphabet Π. Given a pair of p-strings S and T, the parameterized pattern matching (p-match) problem is to verify whether the individual constant symbols match and whether there exists a bijection between the parameter symbols of S and T. If the two conditions are met, S is said to be a p-match of T. A significant breakthrough in the p-match area is the *prev* encoding, which is proven to identify a p-match between S and T if and only if $prev(S) == prev(T)$. In order to utilize suffix data structures in terms of p-matching, we must account for the dynamic nature of the parameterized suffixes (p-suffixes) of T, namely $prev(T[i...n])\ \forall\ i, 1 \leq i \leq n$.

In this work, we propose transformative approaches to the direct parameterized suffix sorting (p-suffix sorting) problem by generating and sorting lexicographically numeric fingerprints and arithmetic codes that correspond to individual p-suffixes. Our algorithm to p-suffix sort via fingerprints is the first theoretical linear time algorithm for p-suffix sorting for non-binary parameter alphabets, which assumes that each code is represented by a practical integer. We eliminate the key problems of fingerprints by introducing an algorithm that exploits the ordering of arithmetic codes to sort p-suffixes in linear time on average.

The longest previous factor (LPF) problem is defined for traditional strings exclusively from the constant alphabet Σ. We generalize the LPF problem to the parameterized longest previous factor (pLPF) problem defined for p-strings. Subsequently, we present a linear time solution to construct the $pLPF$ array. Given our pLPF algorithm, we show how to construct the $pLCP$ (parameterized longest common prefix) array in linear time. Our algorithm is further exploited to construct the standard LPF and LCP arrays all in linear time.

We then study the structural string (s-string), a variant of the p-string that extends the p-string alphabets to include complementary parameters that correspond to one another. The s-string problem involves the new encoding schemes *sencode* and *compl* in order to identify a structural match (s-match). Current s-match solutions use a structural suffix tree (s-suffix tree) to study structural matches in RNA sequences. We introduce the suffix array, LCP, and LPF data structures for the s-string encoding schemes. Using our new data structures, we identify the first suffix array solution to the s-match problem. Our algorithms and data structures are shown to apply to s-strings and also p-strings and traditional strings.

Acknowledgments

My committee chair and advisor Dr. Donald A. Adjeroh has always proposed the questions to fuel novel, cutting-edge research. Dr. Adjeroh introduced me to the world of strings and always provided the guidance to see an idea through to success. For the research and teaching opportunities, I am grateful. I would like to deeply thank my committee members Dr. Arun A. Ross and Dr. Elaine M. Eschen for their insights, suggestions, and support of my work. The knowledge obtained in the coursework taught by committee members was fundamental to the results achieved in this research. The wisdom accumulated in those courses will indeed extend beyond this work. Together, we have advanced string theory.

The individuals involved in my "road to graduate school" also deserve appreciation. My pursuit of an advanced degree is thanks to the persistence of Dr. Anthony Pyzdrowski, in addition to the support of Dr. Lisa Kovalchick and Dr. Weifeng Chen. My professional mentors – Dr. George Novak, Bjorn Moreau, Bill Cooey, and Thad Magyar – are responsible for shaping my knowledge base with internship opportunities. I am appreciative of Dr. Edward Chute and Professor Erin Mountz for fostering my creativity with honors research. To my overseeing committee – Chris Pollaro, Chris Janovich, Vince Baronti, Aric Ilko, Brian Krukowsky, and Ray Boyles – and colleague Jessie Salmon, I am thankful for your friendship.

I am infinitely appreciative of my father, Richard Sr., and mother, Pamela. My parents have been instrumental in my studies and very supportive in the roller coaster that is academia – spanning from pressures to praises of proposals, presentations, and papers. To sharpen my focus, my brother, Eric, has always been there to *first* break any monotony in my studies with in-depth discussions on football rivalries and situational comedies. I would like to deeply thank my cousin James V. Matthews for his continued confidence in my abilities.

This work was partly supported by a grant from the National Historical Publications & Records Commission (in collaboration with the DCAPE team at the University of North Carolina, Chapel Hill).

Contents

Listings

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and the Problem

Strings are everywhere; they construct the World Wide Web, represent the human genome, and even provide the transmission layout for our daily communications! A traditional string is a production of symbols from the constant alphabet Σ. An exact match exists between two traditional strings S and T when each symbol matches. Traditional strings are powerful data structures to determine whether two strings are exactly equivalent by simply comparing symbols. The limitation of the traditional string is that any further intricate study of the symbol composition and structure requires intelligent algorithms and bookkeeping. The source code in Figure 1.1 displays two programs with slightly different code and output to achieve the same function: to display all possible permutations of DNA sequences of length n. Exact matching will not detect this relationship between the source files.

The parameterized matching (p-match) problem is a sophisticated pattern matching scheme that utilizes a parameterized string (p-string), which is a production from the constant alphabet Σ and parameter alphabet Π, with $\Sigma \cap \Pi = \emptyset$. A p-match exists between two p-strings S and T when the constant symbols $\sigma \in \Sigma$ match and there exists a bijection of parameter symbols $\pi \in \Pi$ between S and T. If we disregard whitespace and let $\Sigma = \{class, public, static, ..., \{, \}, (,), ...\}$ represent the keywords and special tokens and let $\Pi = \{n, num, prog, Program, ...,A,C,G,T, ...\}$ represent the remaining tokens, namely the

variables and values, then we observe that a p-match exists between the two source files in Figure 1.1. The use of the p-string in the p-match problem permits a more natural pattern matching scheme to observe the composition of parameters in a string. The notion that the p-string can provide more involved pattern matching capabilities for applications provides the motivation to 1) redefine traditional string problems for p-strings, 2) construct p-string oriented data structures with algorithms that run in linear time with practical memory footprints, and 3) further advance the string theory of related sophisticated string definitions as the structural string (s-string) used in the structural match (s-match) problem.

1.2 General Approach

In this work, we advance the theory of p-strings [1, 2, 3] and s-strings [4, 5] with algorithms that construct traditional and newly proposed data structures by extending proven solutions for traditional strings. The resulting data structures are intended to be used for string applications, such as pattern matching. The time efficient pattern matching technique via the space practical suffix array (SA) and longest common prefix (LCP) array combination introduced by Manber and Myers [6] is the core incentive to computing the SA and LCP data structures. The longest previous factor (LPF) data structure introduced by Crochemore [7] is used in fundamental string applications dealing with compression and duplication. The purpose of this thesis is to construct the SA, LCP, and LPF data structures for p-strings and s-strings intended for applications analogous to the respective data structures for traditional strings. The major challenge that we conquer is the correct and efficient handling of the *dynamic* nature of p-string and s-string suffixes, which is an intricate process since the suffixes of a p-string and s-string are encoded. Thus, the *dynamic* suffix encodings require more sophisticated methods than their traditional counterparts. The general approach used to directly construct our proposed suffix arrays for p-strings and s-strings *without* the use of a suffix tree is centralized on generalizing the traditional direct suffix sorting approaches of [8] for the encoding schemes of the p-string and s-string. Our approach to construct the LPF data structure also revolves around generalizing the traditional LPF algorithm in

[7] to handle the p-string and s-string encodings. We further compute the respective LCP arrays by exploiting the respective LPF algorithms. Throughout the thesis, we make it clear that our algorithms and data structures are generalized to handle s-strings, p-strings, and traditional strings.

1.3 Thesis Contributions

We introduce a transformative approach to direct parameterized suffix sorting (p-suffix sorting), *without* the assistance of a suffix tree, by representing *m-block* prefixes of each individual parameterized suffix (p-suffix) of a p-string with a parameterized arithmetic code (pAC). It is shown that pAC codes can be generated in linear time by transitioning the codes between neighboring p-suffixes, conceptually similar to the approach used in [8]. The resulting codes are then sorted to, in turn, sort the p-suffixes and construct the parameterized suffix array (p-suffix array) in linear time on average. This same approach is used to construct the suffix arrays for the structural encodings (s-encodings) of an s-string.

In addition to the suffix array data structure, we introduce new flavors of conventional problems, originally defined for traditional strings, in terms of p-strings and s-strings. We define the parameterized longest previous factor (pLPF) problem analogous to the LPF problem for traditional strings. An algorithm is presented to construct the $pLPF$ data structure in linear time. We then use the pLPF algorithm to also construct the parameterized longest common prefix ($pLCP$) data structure. Even though in [7] the LPF array is constructed with the LCP array and it is acknowledged that LCP and LPF arrays are permutations of one another, we are the first to observe that a single LPF algorithm can be exploited to construct both the LPF and LCP data structures. We show how this construction is achieved. We also show similar results for the s-string encodings.

The current state of the art solutions for the s-match problem defined for s-strings revolves around the s-suffix tree data structure [4, 5]. By constructing the suffix array and LCP array for s-string encodings, we provide the first suffix array solutions to the s-match problem. Our data structures and algorithms are generalized to also apply to the s-string, the p-string, and the traditional string.

1.4 Thesis Outline

Prior to the intricate details of our contributions, we place our work in perspective by describing the current state of the art in the area of strings within Chapter 2. The chapter discusses the history of exact pattern matching with traditional strings from conventional matching, to suffix trees, and ultimately leading to the suffix array data structure. The suffix array is highlighted as a time and space practical data structure for efficient pattern matching. We further detail the suffix array pattern matching solutions introduced by Manber and Myers [6]. Next, a similar history is shown for the p-string and the p-match problem. It is here that we identify the modern challenges of p-suffix sorting. The chapter then moves to a discussion of the suffix array in terms of the LPF problem for traditional strings with a brief spotlight on the importance of LPF in fundamental string applications. We then venture into the world of s-strings and highlight the exclusive use of the structural suffix tree (s-suffix tree) to address the s-match problem, which serves as a motivation for additional s-matching data structures. The preliminaries that conclude the chapter present the foundation for p-strings used throughout the thesis.

The essence of Chapter 3 is the novel use of coding techniques to address the p-suffix sorting problem, namely, constructing the p-suffix array. We start the chapter by setting the stage for the p-suffix sorting problem, highlighting the demand for new approaches. Our task is to propose approaches to directly construct the p-suffix array *without* the use of a parameterized suffix tree (p-suffix tree). Fingerprinting is explored as a new method to address direct p-suffix sorting by which, sorting the integral fingerprints will sort the p-suffixes. We then derive a mapping function to map symbols to lexicographically sorted integers. Subsequently, we identify the challenges of constructing fingerprints that represent each individual p-suffix with traditional fingerprinting techniques. Then, we introduce the parameterized Karp Rabin (pKR) algorithm to generate the fingerprints. We then identify a more efficient way to *transition* fingerprints to construct neighboring fingerprints in constant time. It is then shown how to sort our fingerprints to, in turn, p-suffix sort in linear time. The promise of fingerprinting is accompanied with the practical limitations of conventional KR fingerprints. Addressing these limitations leads to our main contribution: a fundamentally

unique method to p-suffix sort in linear time on average by representing the *m-block* prefixes of the individual p-suffixes with parameterized arithmetic codes (pAC). We further discuss the detail of pAC codes using the same methodology as the pKR discussion.

In Chapter 4, we first introduce the LPF problem for traditional strings. We then define the pLPF problem for p-strings analogous to the traditional LPF. Motivated by the traditional LPF algorithm in [7], we show how to similarly *extend* pLPF computations for neighboring p-suffixes in a p-string. Using this, we present the algorithm to construct the $pLPF$ data structure in linear time. The chapter continues with the observation that the $pLPF$ and $pLCP$ data structures are related. We then identify how to construct the $pLCP$ data structure from the pLPF algorithm by simply altering the data structures in the function call. Our p-string contributions are supplemented by the proof that our algorithms are also capable of working with traditional strings. We conclude the chapter by briefly discussing practical applications of the LPF and $pLPF$ data structures.

Chapter 5 explores the s-string, a variant of the p-string. In this chapter, we begin with an introduction on the history and theory of s-strings and the exclusive use of the s-suffix tree to solve the s-match problem. We then present additional s-string preliminaries to provide the foundations for the chapter. Next, we introduce the suffix array data structures for the s-encodings, utilizing a similar approach to that of Chapter 3. Mirroring the methodology of Chapter 4, we define the LPF problem in terms of the s-encodings and present solutions to compute the corresponding LPF and LCP arrays. We use our proposed s-string data structures to introduce the first suffix array solutions to the s-match problem. Throughout the chapter, we show the generalized nature of our work by proving that s-string algorithms and data structures also address corresponding problems in both p-strings and traditional strings.

Chapter 6 concludes the thesis by summarizing our main contributions and identifying the future research areas emerging from this work.

```
1  public class Program {
2   private static char[] alphabet
3               = {'A','C','G','T'};
4   private int num;
5
6   public Program(int num)
7                   throws Exception
8   {
9     this.num = num;
10    if(this.num <= 0) //invalid
11      throw new Exception("!!!");
12    this.dnaPermutations("");
13  }
14
15  public void dnaPermutations(
16                           String str){
17    if(str.length() != this.num)
18    {
19      for(char q : this.alphabet)
20        this.dnaPermutations(str+q);
21    }
22    else
23      System.out.println(str);
24  }
25
26  public static void main(
27      String[] args) throws Exception
28  {
29    new Program(3);
30  }
31 }
```

```
public class prog {
 private static char[] alpha
              = {'A','T','G','C'};
 private int n;

 public prog(int n)
                  throws Exception{
   this.n = n;
   if(this.n <= 0) //invalid
     throw new Exception("!!!");
   this.dna_perm("");
 }


 public void dna_perm(String s){
   if(s.length() != this.n){
     for(char q : this.alpha)
       this.dna_perm(s+q);
   }else System.out.println(s);
 }




 public static void main(
  String[] args) throws Exception{
   new prog(3);
 }

}
```

Figure 1.1: Source files that p-match

Chapter 2

Background / Related Work

Baker [9] defines three types of pattern matching: 1) exact matching, 2) parameterized matching, and 3) matching with modifications. The exact and parameterized matching schemes are the relevant background materials required for this work. We further discuss the longest previous factor (LPF) problem in terms of exact matching, which is significant in many pattern matching applications. We conclude by observing a variant of the parameterized matching (p-match) problem known as the structural matching (s-match) problem defined especially for applications involving biological sequences that require more intricate pattern matching techniques.

2.1 Exact Matching

Exact pattern matching of a pattern P ($m = |P|$) on a string $W = W[1]W[2]...W[n]$ ($n = |W|$) from the alphabet Σ is the exact matching of symbols between P and W at some position i in W. Traditional mechanisms of exact matching with algorithms KMP and BM are the basis of many hybrid algorithms to match patterns in $O(n)$ time [10, 11]. Suffix trees and suffix arrays are suffix data structures that were introduced to improve pattern matching capabilities by exploiting the relationships between the individual suffixes $W[i...n]$ of a string. The i^{th} suffix of a string $W = AABABA$ with $n = 6$ and $i = 3$ is $W[3...6] = BABA$. An overview of suffix structures is included in [12]. The suffix tree is a tree structure that represents each suffix as a path from the root to a leaf, which may be constructed in $O(n)$

time with $O(n)$ space [10, 11, 12, 13]. The practical space required to represent the suffix tree was the bottleneck that led way to more space efficient data structures. Manber and Myers [6] introduced the suffix array with a construction algorithm originally requiring $O(n \log n)$ time, which escapes the practical space limitations of the suffix tree. They also showed how to use the LCP (longest common prefix) array to competitively search for a pattern in $O(m + \log n)$ time. Table 2.1 displays the suffix array SA and LCP for a traditional string. Any algorithm that constructs a suffix array and indirectly requires the use of a suffix tree suffers from the same suffix tree space limitations. Direct suffix sorting algorithms construct the suffix array *without* the suffix tree and thus, do not require the memory footprint of the suffix tree. Linear time direct suffix sorting algorithms were described by [14] along with lightweight suffix sorting algorithms developed by [15]. More recently, Adjeroh and Nan [8] proposed a transformative approach to linear time direct suffix sorting by representing *m-blocks*, short prefixes of the suffixes, with arithmetic codes and intelligently sorting the *m-blocks* to obtain the suffix array.

Table 2.1: String computations on W=$CABCABCC\$$, $\Sigma = \{A, B, C\}$, $\$ \notin \Sigma$

i	$W[SA[i]...n]$	$SA[i]$	$LCP[i]$
1	\$	9	0
2	ABCABCC\$	2	0
3	ABCC\$	5	3
4	BCABCC\$	3	0
5	BCC\$	6	2
6	C\$	8	0
7	CABCABCC\$	1	1
8	CABCC\$	4	4
9	CC\$	7	1

Consider finding the pattern $P = BCA$ with $m = 3$ in our example string $W = CABCABCC\$$ with $n = 9$ and $SA = \{9, 2, 5, 3, 6, 8, 1, 4, 7\}$ as the suffix array of W, displayed in Table 2.1. Manber and Myers [6] identify two main techniques that extend the traditional binary search to string suffixes by exploiting the lexicographical ordering of the SA, in order to efficiently determine the location of pattern P in string W. Their first approach compares each pattern P with the individual suffixes of W encountered in the binary search process, which requires $O(m \log n)$ time. Initially, we consider that P may

exist the range $[L, R] = [1, n]$: that is, $[1, 9]$ or the entire SA. As with the binary search, we identify the midpoint $M = \lfloor \frac{1+9}{2} \rfloor = 5$ and compare P with $W[SA[M]...SA[M] + m - 1]$ to identify whether: 1) P is found, 2) P exists in the range $[L, M)$, or 3) P exists in the range $(M, R]$. So, we compare $P == W[SA[M]...SA[M] + m - 1]$ or $BCA == BCC$ and identify that only the first two symbols BC match. Since $k = |BC| < m$, a match does not occur. Moreover, since $P[k + 1] < W[SA[M + k]]$ or $A < C$ is lexicographically the case, we can refine the possible location of P to the left half of the SA, namely $[L, M) = [1, 5)$. We continue this process next with $M = 3$ by comparing $P == W[SA[M]...SA[M] + m - 1]$ or $BCA = ABC$ and detecting that $k = 0$ symbols match. Since $P[1] > W[SA[3]]$ or $B > A$ is lexicographically true, we further refine the location of M to be in the right half of the previous range, namely $(M, R] = (3, 5]$. Upon the next trial, we identify that $M = 4$ and indeed $P == SA[SA[M] + m - 1]$. Thus, we can report that P exists in W at position $SA[4] = 3$.

The notion that the SA maintains a lexicographical ordering of the suffixes of a string permits the use of the binary search. The problem with the previously described algorithm with running time $O(m \log n)$ is the need to continually match all m symbols of P between each suffix of W. It was shown in [6] that a significant number of symbol comparisons can be saved by *extending* matches using the LCP values between each suffix at midpoint M with $1 < M < n$ and the two suffixes L and R such that $M = \lfloor \frac{L+R}{2} \rfloor$. They further identify how to use the $2 \times (n - 2) \in O(n)$ LCP values to search for P in the SA of W in $O(m + \log n)$ time. The improved algorithm uses the same conceptual idea of our previously discussed example and differs only in the integration of the LCP values.

2.2 Parameterized Matching

A parameterized string (p-string) is composed of symbols from a constant symbol alphabet Σ and a parameter alphabet Π. A pair of p-strings S and T of length n are said to p-match when the constant symbols $\sigma \in \Sigma$ match and there exists a bijection of parameter symbols $\pi \in \Pi$ between the pair of p-strings. Baker [3] offered the first p-match breakthroughs, namely, the *prev* encoding and the parameterized suffix tree (p-suffix tree).

The p-suffix tree is analogous to the suffix tree for traditional strings [10, 11, 12, 13]. Baker discovered that a p-match exists between the p-strings S and T when $prev(S) == prev(T)$. The p-suffix tree is built on the *prev* encodings of the individual suffixes of the p-string, which requires $O(n(|\Pi| + \log(|\Pi| + |\Sigma|)))$ construction time [3]. Improvements to the p-suffix tree construction were introduced in [16, 17, 18]. The physical space required for the p-suffix tree implementation was a limitation acknowledged as traditional pattern matching approaches were extended to offer time and space efficient p-match functionality. Amir et al. proposed the parameterized-KMP [19] and Baker introduced the parameterized-BM [20], in which each method detected p-matches optimally in time $O(n \log(min\{m, |\Pi|\}))$ with m as the length of the pattern. Idury et al. [21] studied the multiple p-match problem using the traditional Aho-Corasick automata [22]. The native time and space efficiency of the suffix array led to the origination of the parameterized suffix array (p-suffix array). Table 2.2 displays the p-suffix array pSA and parameterized longest common prefix $pLCP$ arrays for a p-string. The p-suffix array is analogous to the suffix array for traditional strings [6, 10, 11, 12]. Direct p-suffix array construction was first introduced by Deguchi et al. [23] for binary strings with $|\Pi| = 2$, which required $O(n)$ construction time through the assistance of a defined fw encoding. Deguchi and colleagues [24] later proposed the first approaches to p-suffix sorting with an arbitrary alphabet size requiring $O(n^2)$ time in the worst case. The parameterized longest common prefix ($pLCP$) array analogous to the traditional LCP array was also defined and constructed in [23, 24]. We introduce new algorithms to p-suffix sort in linear time on average using coding methods from information theory.

Table 2.2: p-string computations on T=$AwBzABwz$\$, Σ=$\{A, B\}$, Π=$\{w, z\}$, \$ $\notin \Sigma \cup \Pi$

i	$T[pSA[i]...n]$	$prev(T[pSA[i]...n])$	$pSA[i]$	$pLCP[i]$
1	\$	\$	9	0
2	z\$	0\$	8	0
3	wz\$	00\$	7	1
4	zABwz\$	0AB04\$	4	1
5	wBzABwz\$	0B0AB54\$	2	1
6	AwBzABwz\$	A0B0AB54\$	1	0
7	ABwz\$	AB00\$	5	1
8	Bwz\$	B00\$	6	0
9	BzABwz\$	B0AB04\$	3	2

2.3 Longest Previous Factor

In a novel application of the suffix array and the corresponding LCP array, Crochemore and Ilie [7] introduced the longest previous factor (LPF) problem for traditional strings. Table 2.3 shows an example LPF for a short sequence $W = AAABABAB\$$. For any suffix u beginning at index i in string W, the LPF problem is to identify the exact matching longest factor between u and another suffix v starting prior to index i in W. We note that this definition is similar to (though not the same as) the *Prior* array used in [10]. Crochemore and Ilie [7] exploited the notion that the nearby elements within a suffix array are closely related en route to proposing a linear time solution to the LPF problem. They also proposed another linear time algorithm to compute the LPF array by using the LCP structure. The significance of an efficient solution to the LPF is that the resulting data structure simplifies computations in various string analysis procedures. Typical examples include computing the Lempel-Ziv factorization [25, 26], which is fundamental in string compression algorithms such as the UNIX *gzip* utility [10, 11] and in algorithms for detecting repeats in a string [27]. Our motivation to study the LPF in terms of p-strings and s-strings is the power of generalizations and the relevance to various important applications.

Table 2.3: LPF calculation for string $W = AAABABAB\$$, $\Sigma = \{A, B\}$, $\$ \notin \Sigma$

i	$SA[i]$	$W[SA[i]...n]$	$LCP[i]$	$W[i...n]$	$LPF[i]$
1	9	$	0	AAABABAB$	0
2	1	AAABABAB$	0	AABABAB$	2
3	2	AABABAB$	2	ABABAB$	1
4	7	AB$	1	BABAB$	0
5	5	ABAB$	2	ABAB$	4
6	3	ABABAB$	4	BAB$	3
7	8	B$	0	AB$	2
8	6	BAB$	1	B$	1
9	4	BABAB$	3	$	0

2.4 Structural Matching

A structural string (s-string), introduced by Shibuya [4, 5], is a p-string with the added notion of *complementary* parameter symbols. Two parameter symbols $\pi_1, \pi_2 \in \Pi$ are comple-

ments if they correspond to each other in the following way: $\pi_1 == complement(\pi_2) \wedge \pi_2 == complement(\pi_1)$. A structural match (s-match) exists between two s-strings S and T when 1) the constant symbols match, 2) a bijection exists between the parameter symbols of S and T, and 3) the complementary symbols are consistently structured in both s-strings. The relationship between the s-match and p-match problem is evident since conditions 1) and 2) are fulfilled by identifying a p-match. The notion of complementary symbols adds a level of structure not achieved in the p-match problem. Shibuya [4] proposes a *compl* encoding scheme that shows the structure of the complementary parameters to enforce condition 3). It is shown in [4] that an s-match exists between the s-strings S and T when $prev(S) == prev(T) \wedge compl(S) == compl(T)$. The *sencode* scheme is then proposed by [4] to simplify s-matching so that two s-strings S and T s-match when $sencode(S) == sencode(T)$. The structural suffix tree (s-suffix tree) by Shibuya [4] is a data structure, analogous to the traditional suffix tree [10, 11, 12, 13] and the p-suffix tree [3, 16, 17, 18], used to facilitate convenient s-matching, requiring $O(n(\log|\Sigma| + \log|\Pi|))$ construction time. Shibuya [4] shows how to use the s-suffix tree to address the structural pattern matching of RNA sequences. We advance s-string theory by proposing an s-match solution using newly introduced suffix array and LCP data structures for the s-string encodings.

2.5 Main Contributions

We advance the current state of the art in p-string theory by introducing transformative approaches of working with p-strings via coding techniques. We present efficient algorithms to construct fundamental p-string data structures for the p-match problem and generalize problems for the p-string. The techniques used in our p-string work are shown to be portable to the area of s-strings and s-match problem.

Uniquely, we represent the dynamic p-suffixes under the *prev* encoding of a p-string by special arithmetic codes. We show how to generate a parameterized arithmetic code (pAC) to represent an *m-block*, a prefix of length m, of a p-suffix and maintain the lexicographical ordering of the p-suffixes between the representative pAC codes. The relationships between adjacent p-suffixes are exploited to efficiently translate one *m-block* code to succeeding p-

suffixes in order to generate the pAC codes for all n suffixes of T. It is shown that sorting the codes is equivalent to sorting the p-suffixes and thus, constructing the p-suffix array. Our direct p-suffix sorting approach via information theoretic codes is the first algorithm to claim linear time on average, stated formally in the following theorem.

Theorem 3.3.5 *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time on average via parameterized arithmetic coding.*

As an application for the p-suffix array, we generalize the standard LPF problem to the parameterized longest previous factor (pLPF) problem defined for p-strings. We identify the similarities between the LPF and pLPF problems en route to proposing a linear time solution to construct the $pLPF$ data structure. We state our result in the following theorem:

Theorem 4.3.4. *Given an n-length p-string T, $prevT = prev(T)$, the prev encoding of T, and pSA, the parameterized suffix array for T, the algorithm compute_$pLPF$ constructs the $pLPF$ array in $O(n)$ time.*

We are the first to identify how to modify the LPF algorithm to also solve the LCP problem. We show that our pLPF algorithm can also be used to construct the $pLCP$ array, as formalized in the following.

Theorem 4.4.2. *Given an n-length p-string T, $prevT = prev(T)$, the prev encoding of T, and pSA, the parameterized suffix array for T, the compute_$pLPF$ algorithm can be used to construct the $pLCP$ array in $O(n)$ time.*

We further highlight the power of our pLPF algorithm by proving that the traditional LPF and LCP arrays may also be constructed from our pLPF solution.

In this work, we also make significant contributions to the area of s-strings, a variant of the p-string. We introduce and construct the suffix array data structure for the *compl* and *sencode* encodings fundamental to the s-match problem, identified in Theorem 5.3.13.

Theorem 5.3.13. *Given an s-string T of length n, constructing the sSA, cSA, pSA, and SA can be accomplished in $O(n)$ time on average via structural arithmetic coding.*

Then, we define the LPF problem in terms of the s-string encodings and further show how

to compute the respective LCP arrays. The resulting data structures are then utilized to propose the first s-match solution via the suffix array. Theorem 5.5.1 formalizes the s-match claim.

Theorem 5.5.1. *Given an n-length s-string T, the sSA, and the $sLCP$ data structure, it is possible to s-match, c-match, p-match, or traditional match an m-length s-string P in $O(m + \log n)$ time.*

The power of our s-string data structures and algorithms is the generalization potential that we identify to also apply to respective problems in p-strings and traditional strings, formalized in Theorems 5.4.4 and 5.4.8.

Theorem 5.4.4. *Given an n-length s-string T and the appropriate suffix array, the algorithm compute_all_LPF can construct the $sLPF$, $cLPF$, $pLPF$, and LPF array in $O(n)$ time.*

Theorem 5.4.8. *Given an n-length s-string T, the algorithm compute_all_LCP can construct the $sLCP$, $cLCP$, $pLCP$, and LCP array in $O(n)$ time.*

2.6 Preliminaries and Notation

A string on an alphabet Σ is a production $T = T[1]T[2]...T[n]$ from Σ^n with $n = |T|$ the length of T. We will use the following string notations: $T[i]$ refers to the i^{th} symbol of T, $T[i...j]$ refers to the substring $T[i]T[i+1]...T[j]$, and $T[i...n]$ refers to the i^{th} suffix $T[i]T[i+1]...T[n]$ of T. A factor refers to a nonempty substring and a prefix is defined as a leading substring of a suffix. The area of parameterized pattern matching defines the finite alphabets Σ and Π. Alphabet Σ denotes the set of constant symbols while Π represents the set of parameter symbols. Alphabets are defined such that $\Sigma \cap \Pi = \emptyset$. Furthermore, we append the terminal symbol $\$ \notin \Sigma \cup \Pi$ to the end of all strings to clearly distinguish between suffixes. For practical purposes, we can assume that $|\Sigma| + |\Pi| \leq n$ since, otherwise a single mapping can be used to enforce the condition.

Definition 2.6.1 ***Parameterized string (p-string):*** *A p-string is a production T of length n from $(\Sigma \cup \Pi)^*\$$.*

Consider the alphabet arrangements $\Sigma = \{A, B\}$ and $\Pi = \{w, x, y, z\}$. Example p-strings include $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$.

Definition 2.6.2 ([23, 24]) ***Parameterized matching (p-match):*** *A pair of p-strings S and T are p-matches with $n = |S|$ if and only if $|S| == |T|$ and each $1 \leq i \leq n$ corresponds to one of the following:*

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] == T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b))$ */* parameter bijection */*
   (a) $S[i] \neq S[j], T[i] \neq T[j]$ **for any** $1 \leq j < i$
   (b) $S[i] == S[i-q]$ **iff** $T[i] == T[i-q]$ **for any** $1 \leq q < i$

In our example, we have a p-match between the p-strings S and T since every constant/terminal symbol matches and there exists a bijection of parameter symbols between S and T. U does not satisfy the parameter bijection to p-match with S or T. The process of p-matching leads to defining the *prev* encoding.

Definition 2.6.3 ([23, 24]) ***Previous (prev) encoding:*** *Given $\mathbb{Z}$ as the set of non-negative integers, the function prev : $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ accepts a p-string T of length n and produces a string Q of length n that* 1) *encodes constant/terminal symbols with the same symbol and* 2) *encodes parameters to point to* **previous** *like-parameters. More formally, Q is constructed of individual $Q[i]$ with $1 \leq i \leq n$ where:*

$$Q[i] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \textbf{ if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } 1 \leq j < i \\ i - k, \textbf{ if } T[i] \in \Pi \wedge k = max\{j | T[i] == T[j], 1 \leq j < i\} \end{cases}$$

For a p-string T of length n, the above $O(n)$ space *prev* encoding requires the construction time of order $O(n \log(min\{n, |\Pi|\}))$, which follows from the discussions of Baker [3, 20] and Amir et al. [19] on the dependency of alphabet Π in p-match applications. Note that with an indexed alphabet and an auxiliary $O(|\Pi|)$ mapping structure, we can construct *prev* in $O(n)$ time. Using Definition 2.6.3, our working examples evaluate to $prev(S) = A0B0AB54\$$,

$prev(T) = A0B0AB54\$$, $prev(U) = A0B2AB31\$$. The relationship between p-strings and the lexicographical ordering of the *prev* encoding is fundamental to the p-match problem.

Definition 2.6.4 *prev* ***Lexicographical ordering:*** *Given the p-strings S and T and two symbols s and t from the encodings $prev(S)$ and $prev(T)$ respectively, the relationships $==$, $\neq$, $<$, and $>$ refer to lexicographical ordering between s and t. We define the ordering of symbols from a prev encoding of the production $(\Sigma \cup \mathbb{Z})^*\$$ to be $\$ < \zeta \in \mathbb{Z} < \sigma \in \Sigma$, where each ζ and σ is lexicographically sorted in their respective alphabets. The relationships $==$, $\neq$, $\prec$, and $\succ$ refer to the lexicographical ordering between strings. In the case of $prev(S)$ and $prev(T)$, $prev(S) \prec prev(T)$ when $prev(S)[1] == prev(T)[1], prev(S)[2] == prev(T)[2], ..., prev(S)[j-1] == prev(T)[j-1], prev(S)[j] < prev(T)[j]$. Similarly, we can define $==_k$, $\neq_k$, $\prec_k$, and $\succ_k$ to refer to the lexicographical relationships between a pair of p-strings considering only the first $k \geq 0$ symbols.*

The following proposition essential to the p-match problem is directly related to the established symbol ordering.

Proposition 2.6.5 ([3]) *Two p-strings S and T p-match when $prev(S) == prev(T)$. Similarly, $S \prec T$ when $prev(S) \prec prev(T)$ and $S \succ T$ when $prev(S) \succ prev(T)$.*

The example *prev* encodings show a p-match between S and T since $prev(S) = A0B0AB54\$$ and $prev(T) = A0B0AB54\$$. Also, $U \succ S$ and $U \succ T$ since $prev(U) = A0B2AB31\$ \succ prev(S) == prev(T) = A0B0AB54\$$. We use the ordering established in Definition 2.6.4 to define the parameterized suffix array and the parameterized longest common prefix array.

Definition 2.6.6 ***Parameterized suffix array (pSA):*** *The pSA for a p-string T of length n maintains a lexicographical ordering of the indices i representing individual p-suffixes $prev(T[i...n])$ with $1 \leq i \leq n$, such that $prev(T[pSA[q]...n]) \prec prev(T[pSA[q+1]...n])$ $\forall\ q, 1 \leq q < n$.*

Definition 2.6.7 ***Parameterized longest common prefix ($pLCP$) array:*** *The $pLCP$ array for a p-string T of length n maintains the length of longest common prefix between neighboring p-suffixes. We define the computation $plcp(\alpha, \beta) = max\{k \mid prev(\alpha) ==_k prev(\beta)\}$. Then, $pLCP$ is defined on each p-suffix i with $1 \leq i \leq n$ such that:*

$$pLCP[i] = \begin{cases} 0, \textbf{ if } i == 1 \\ max\{k \mid plcp(T[pSA[i]...n], T[pSA[i-1]...n])\}, \textbf{ if } 2 \leq i \leq n \end{cases}$$

For $T = AwBzABwz\$$ with $prev(T) = A0B0AB54\$$, we have $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$ and $pLCP = \{0, 0, 1, 1, 1, 0, 1, 0, 2\}$. The encoding $prev$ is supplemented by the encoding $forw$.

Definition 2.6.8 ***Forward ($forw$) encoding:*** *Let the function $rev(T)$ reverse the p-string T and $repl(T, x, y)$ replace all occurrences in T of the symbol x with y. We define the function $forw$ for the p-string T of length n as $forw(T) = rev(repl(prev(rev(T)), 0, n))$.*

The function $forw$ performs the following on a p-string T of length n: 1) encodes each constant/terminal symbol with the same symbol and 2) encodes each parameter p with the **forward** distance to the next occurrence of p or an unreachable forward distance n. Our definition of the $forw$ encoding generates output mirroring the fw encoding used by Deguchi et al. [23, 24]. Let $\mathbb{N}$ refer to the set of positive, non-zero integers. The function $fw : (\Sigma \cup \Pi)^* \rightarrow (\Sigma \cup \mathbb{N})^*$ produces an output encoding G with $fw(T) = G$ for each $1 \leq i \leq n$:

$$G[i] = \begin{cases} T[i], \textbf{ if } T[i] \in \Sigma \\ \infty, \textbf{ if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } i < j \leq n \\ k - i, \textbf{ if } T[i] \in \Pi \wedge k = min\{j | T[i] == T[j], i < j \leq n\} \end{cases}$$

The $forw$ encodings in our example with $n = 9$ are $forw(S) = A5B4AB99\$$, $forw(T) = A5B4AB99\$$, $forw(U) = A2B3AB19\$$.

Chapter 3

Parameterized Suffix Array

3.1 Introduction

Conventional pattern matching typically constitutes the matching of traditional strings over an alphabet Σ. Parameterized pattern matching using parameterized strings, introduced by Baker [3], attempts to answer pattern matching questions beyond its classical counterpart. A parameterized string (p-string) is a production of symbols from the alphabets Σ and Π with $\Sigma \cap \Pi = \emptyset$, representing the constant symbols and parameter symbols respectively. Given a pair of p-strings S and T, the parameterized pattern matching (p-match) problem is to verify whether the individual constant symbols match and whether there exists a bijection between the parameter symbols of S and T. If the two conditions are met, S is said to be a p-match of T. For example, there exists a p-match between the p-strings $z{=}y * f/{+}{+}$y; and $a{=}b * f/{+}{+}b$; that represent program statements over the alphabets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$. The incentive for studying the p-match problem is the range of problems that a single solution can address including 1) exact pattern matching when $|\Pi| = 0$, 2) mapped matching (m-matching) when $|\Sigma| = 0$ [19], and clearly, 3) p-matching when $|\Sigma| > 0 \wedge |\Pi| > 0$. Applications inherent to the p-matching problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [4], discovering cloned code segments in a program to assist with software maintenance [3], and answering critical legal questions regarding the unauthorized use of intellectual property [28].

Initial solutions to the p-match problem were based on the parameterized suffix tree (p-

suffix tree) [3]. Idury et al. [21] studied the multiple p-match problem in terms of automata. The physical space requirements of the p-suffix tree led to algorithms such as parameterized-KMP [19], parameterized-BM [20], and the parameterized suffix array [23, 24]. Analogous to standard suffix sorting, the problem of parameterized suffix sorting (p-suffix sorting) is to sort all the n parameterized suffixes (p-suffixes) of an n-length p-string into a lexicographic order. The major difficulty is that unlike the traditional suffixes of a string, the p-suffixes are *dynamic*, varying with the starting position of the p-suffix. Thus, standard suffix sorting cannot be directly applied to the p-suffix sorting problem. Current approaches to directly construct the p-suffix array, *without* a p-suffix tree, for an n-length p-string from an arbitrary alphabet require $O(n^2)$ time in the worst case [24]. Such demands the need for alternative approaches to direct p-suffix sorting.

Main Contribution: We construct p-suffix arrays by generating and sorting codes that represent the individual p-suffixes of a p-string. We propose the first theoretical linear time claims to directly p-suffix sort p-strings from non-binary parameter alphabets. We state our main result in the following theorem:

Theorem 3.3.5. *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time on average via parameterized arithmetic coding.*

3.2 p-Suffix Sorting via Fingerprints

The magic of sorting the suffixes of a string T of length n from an alphabet Σ is rooted in the notion that individual suffixes are very closely related. For instance, suffix k is a common suffix to both suffixes i and j with $1 \leq i \leq j \leq k \leq n$. Throughout this work, we are challenged with the reality that the p-suffix, more formally $prev(T[i...n])$, is not *naïvely* the suffix of the *prev* encoding of T, namely $prev(T)[i...n]$, which is formalized in Lemma 3.2.1.

Lemma 3.2.1 *Given a p-string T of length n, the suffixes of $prev(T)$ are not necessarily the p-suffixes of T. More formally, if $\pi \in \Pi$ occurs more than once in T, then $\exists i$, s.t. $prev(T[i...n]) \neq prev(T)[i...n], 1 \leq i \leq n$.*

Proof Consider that the *only* parameter symbol to occur in the p-string T is $\pi \in \Pi$, which

exists *only* at positions α and β with $\alpha < \beta$. Suppose that indeed $prev(T[\alpha...n]) == prev(T)[\alpha...n]$ and $prev(T[\beta...n]) == prev(T)[\beta...n]$. By Definition 3, the first occurrence of symbol π at position α will be *prev* encoded by 0 and the π at position β will be *prev* encoded by $\beta - \alpha$. So, in the case of suffix α, $prev(T[\alpha...n]) == prev(T)[\alpha...n]$. At suffix β, the encoding of π at position β in T will *change* to 0 in $prev(T[\beta...n])$ by Definition 3 whereas $prev(T)[\beta...n]$ will retain the *old* encoding of $\beta - \alpha$ since symbol π *still* occurs in $prev(T)$ at position α. The π at position β forces $prev(T[\beta...n]) \neq prev(T)[\beta...n]$, which is a contradiction. □

The centerpiece of this work is rooted in the notion that we *directly* construct the p-suffix array *without* the large memory footprint of the p-suffix tree by handling the dynamically changing p-suffixes, which is fundamentally different from the standard suffix sorting approaches for traditional strings. To visually identify the difference between traditional suffixes and p-suffixes, consider the example $T = zAwz\$$ as a traditional string, in which the suffixes are methodically created by removing a symbol: $\boxed{zAwz\$} \rightarrow \boxed{Awz\$} \rightarrow \boxed{wz\$} \rightarrow \boxed{z\$} \rightarrow \boxed{\$}$. If we consider the same example $T = zAwz\$$ with $\Sigma = \{A\}$ and $\Pi = \{w, z\}$, the p-suffixes defined under the *prev* encoding are dynamically changing: $\boxed{0A03\$} \rightarrow \boxed{A00\$} \rightarrow \boxed{00\$} \rightarrow \boxed{0\$} \rightarrow \boxed{\$}$.

Our idea is to modify the traditional Karp and Rabin (KR) fingerprinting scheme presented in [10, 11, 29] to accommodate the *changing* nature of p-suffixes. The KR algorithm generates an integral "signature" or "fingerprint" code to represent a string using the lexicographical ordering of symbols [29]. By representing p-suffixes through numeric fingerprints we devise a mechanism to retain a "tangible" copy of the *changing* p-suffixes under the *prev* encoding. In this section, we assume that n is not too large. That is, the KR codes can fit into standard integer representations such as long long int.

We now denote the following variables that are continually reused throughout this section for the working p-string T of length n: $prevT = prev(T)$, $forwT = forw(T)$, $maxP = maxdist(prevT)$ (see below), $R = |\Sigma| + maxP + 2$. Our fingerprinting approach relies on a lexicographical ordering implementation of Definition 2.6.4 to appropriately accommodate for the *prev* encoding alphabet $(\Sigma \cup \mathbb{Z} \cup \{\$\})$. Our ordering scheme, function *map*, is

formalized in Definition 3.2.2.

Definition 3.2.2 ***Mapping function:*** *Let* $maxP = maxdist(prevT) = max\{prevT[i] \mid prevT[i] \in \mathbb{Z}$ *for* $1 \leq i \leq |prevT|\}$. *Let function* $\alpha_i(x, X)$ *return the lexicographical order* $(1, 2, ..., |X|)$ *of the symbol* x *in alphabet* X. *We then define the function* $map : (\Sigma \cup \mathbb{Z} \cup \{\$\}) \rightarrow \mathbb{N}$ *to map a symbol, say* x, *in* $prevT$ *to an integer preserving the ordering of Definition 2.6.4. We also define the supplement function* $in(x, X)$ *to determine if* $x \in X$ *instantaneously based on the definition of* $map(x)$.

$$map(x) = \begin{cases} 1, \textbf{ if } x == \$ \\ \alpha_i(x, \mathbb{Z}) + 1, \textbf{ if } x \in \mathbb{Z} \\ \alpha_i(x, \Sigma) + maxP + 2, \textbf{ if } x \in \Sigma \end{cases}$$

$$in(x, X) = \begin{cases} true, \textbf{ if } X == \mathbb{Z} \wedge (1 < map(x) \leq maxP + 2) \\ true, \textbf{ if } X == (\Sigma \cup \{\$\}) \wedge (map(x) == 1 \vee map(x) > maxP + 2) \\ false, \textbf{ otherwise} \end{cases}$$

The function *map* is fundamental in the following definition for the parameterized Karp-Rabin fingerprint.

Definition 3.2.3 Parameterized Karp-Rabin fingerprint (pKR): *Let p-suffix* $prevT_i = prev(T[i...n])$. *We define* $pKR(i) = \sum_{k=n}^{i} \left[R^{k-1} \times map(prevT_i[n-k+1])\right]$ *to return a fingerprint generated for the p-suffix beginning at index* i.

Table 3.1 shows the parameterized KR fingerprints for the example string $T = AwBzABwz\$$. This example shows the true power of our pKR in that the ordering of the computed fingerprints for p-suffixes of T yields the correct p-suffix array $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$. We notice that using KR directly produces the array $\{1, 4, 5, 2, 3, 6, 7, 9, 8\}$, which is not the correct p-suffix array. The execution of function pKR may be *naïvely* cascaded to produce fingerprints for all p-suffixes at positions $1 \leq i \leq n$ of p-string T requiring $O(n^2)$ time, which is a theoretical bottleneck. We can intelligently construct pKR fingerprints for the p-suffixes of T by taking advantage of the relationship between p-suffixes and pKR codes. Consider q_i to be the pKR code for the p-suffix at position i. The code q_{i+1} can be used to compute the fingerprint for q_i for $i \geq 1$ by introducing a new symbol at position i. Lemmas

Table 3.1: Lexicographical ordering of p-suffixes with pKR, using $T = AwBzABwz\$$

i	pSA	T[pSA[i]...n]	prev(T[pSA[i]...n])	pKR(pSA[i])	KR(pSA[i])
1	9	$	$	43046721	43046721
2	8	z$	0$	90876411	263063295
3	7	wz$	00$	96190821	330556302
4	4	zABwz$	0AB04$	129298356	129593601
5	2	wBzABwz$	0B0AB54$	130740084	130740084
6	1	AwBzABwz$	A0B0AB54$	358900444	358900444
7	5	ABwz$	AB00$	388608030	391501431
8	6	Bwz$	B00$	398108358	424148967
9	3	BzABwz$	B0AB04$	401786973	401819778

3.2.4 and 3.2.5 identify the adjustments that dynamically change the p-suffixes between the neighboring p-suffixes at i and $(i+1)$ when considering a symbol introduced at position i.

Case 1: When the new symbol is a constant, terminal, or the only occurrence of that parameter in the suffix $T[i...n]$, Lemma 3.2.4 describes the required transition.

Lemma 3.2.4 *Given p-string T, $prevT = prev(T)$, and $prevT[i+1...n] == prev(T[i+1...n])$ where $T[i]$ is a constant, terminal, or the* only occurrence *of parameter $T[i]$ in $T[i...n]$, then $prevT[i...n] == prev(T[i...n])$ if $prevT[i] == prev(T[i])$.*

Proof For symbol $\sigma \in (\Sigma \cup \{\$\})$, $prev(\sigma) = \sigma$ by Definition 2.6.3. For symbol $\pi \in \Pi$ Definition 2.6.3 states that $prev(\pi) = 0$ for the first occurrence. When $T[i]$ is the only occurrence of π in $T[i...n]$, $\exists$ no future π to re-encode at positions $(i+1)$ to n by Definition 2.6.3. Since we are given that $prevT[i+1...n] == prev(T[i+1...n])$, and Definition 2.6.3 states that σ or π will generate a definitive encoding without impacting current encodings, then $prevT[i...n] == prev(T[i...n])$ upon adjustment of the encoding at $prevT[i]$ so that $prevT[i] == prev(T[i])$. □

Case 2: When the new symbol is a parameter with multiple occurrences in the suffix $T[i...n]$, Lemma 3.2.5 describes the required transition.

Lemma 3.2.5 *Given p-string T, $prevT = prev(T)$, $forwT = forw(T)$, and $prevT[i+1...n] == prev(T[i+1...n])$ where $T[i] \in \Pi$ occurs multiple times in $T[i...n]$, then $prevT[i...n] == prev(T[i...n])$ after* 1) *identifying the current parameter as the first occurrence of $T[i]$ ($prevT[i] = 0$) and* 2) *modifying the future occurrence of $T[i]$ ($prevT[i + forwT[i]] = forwT[i]$).*

Proof We must achieve $prev(T[i...n])$ by using $prevT[i...n]$ given that $prevT[i+1...n]$ is the correct p-suffix for position $(i+1)$. Since $T[i] \in \Pi$ is the *first* occurrence of $T[i]$ in $T[i...n]$, by Definition 2.6.3, its encoding is clearly $prev(T[i]) = 0$. So, $prevT[i] = 0$ will adjust our p-suffix. However, since we are given the fact that $T[i]$ has future occurrences in $T[i+1...n]$, then $\exists$ exactly one future occurrence of $T[i]$ to adjust. This occurrence of $T[i]$ in $T[i+1...n]$ at position, say j, $j > i$ is currently such that $prevT[j] = 0$ and by Definition 2.6.3, *only* the first occurrence of a $T[i]$ in $prev(T[i...n])$ can be 0. Then, clearly Definition 2.6.3 states that the encoding $prevT[j] = j - i$. To make this change we must locate the next forward parameter $T[i]$ in $T[i+1...n]$, which Definition 2.6.8 informs us is available at $forwT[i]$ positions ahead of the current symbol position i; i.e. $j = i + forwT[i]$. So, $prevT[j] = j - i$ must be the case. By substituting j, $prevT[i + forwT[i]] = (i + forwT[i]) - i \Rightarrow prevT[i + forwT[i]] = forwT[i]$. □

We refer to the pKR code of p-suffix i as q_i. The transitions needed to compute a p-suffix i from a p-suffix $(i+1)$ formalized in Lemmas 3.2.4 and 3.2.5 are subsequently the requirements to compute code q_i from q_{i+1}. These transitions are consolidated into δ_{pKR} and shown to efficiently generate pKR codes.

Definition 3.2.6 Function δ_{pKR}: *Let $\beta = forwT[i]$, $\lambda = (map(\beta) - map(0)) \times R^{n-\beta-1}$, and $B = \frac{q_{i+1} + map(prev(T[i]))R^n}{R}$. We define the function $\delta_{pKR}(i, q_{i+1})$ as follows to return the code q_i via a transition of the provided code q_{i+1} with the newly added symbol at position i.*

$$\delta_{pKR}(i, q_{i+1}) = \begin{cases} B, \textbf{ if } in(prevT[i], \Sigma \cup \{\$\}) \vee (in(prevT[i], \mathbb{Z}) \wedge forwT[i] \geq n) \\ B + \lambda, \textbf{ if } in(prevT[i], \mathbb{Z}) \wedge forwT[i] < n \end{cases}$$

The transition function δ_{pKR} is used to efficiently construct the fingerprints.

Theorem 3.2.7 *Given a p-string T of length n and precalculated variables $prevT$ and $forwT$, function δ_{pKR} requires $O(n)$ time to generate fingerprints for all p-suffixes in T.*

Proof The fingerprints are generated successively by the function calls $q_n = \delta_{pKR}(n, 0)$, $q_{n-1} = \delta_{pKR}(n-1, q_n)$,...,$q_1 = \delta_{pKR}(1, q_2)$. Either case of function δ_{pKR} may be computed

Listing 3.1: p-suffix sorting with fingerprints

```
1  struct pcode { int i, long long int pKR }
2  int[] p-suffix-sort-pKR(char[] T) {
3    pcode[] code, int[] pSA, long long int pKR=0
4    // A) -- generate the individual prev fingerprints
5      for int i=n to 1, step -1 {
6        pKR=δ_pKR(i, pKR)
7        code[i]=(i,pKR)
8      }
9    // B) -- sort p-suffixes
10     radix_sort the pKR attribute of each pair in code
11   // C) -- retain p-suffix array
12     for int k=1 to n, step 1
13       pSA[k]=code[k].i
14   return pSA
15 }
```

in $O(1)$ time and is called sequentially a total of n times, once for each of the n p-suffixes. The overall time is $O(n)$. □

We introduce the algorithm *p-suffix-sort-pKR* in Listing 3.1 to sort p-suffixes via the sorting of fingerprints through the transition function in Definition 3.2.6. Theorem 3.2.8 proves the time complexity of Listing 3.1.

Theorem 3.2.8 *Given a p-string T of length n, function p-suffix-sort-pKR sorts all the n p-suffixes of T in $O(n)$ time.*

Proof We assume that the fingerprints for each p-suffix are practically represented by an integer code and each computational use of the code is achieved in constant time. Thus, Section A) of *p-suffix-sort-pKR* follows from Theorem 3.2.7 to require $O(n)$ time. The radix sorting required in section B) requires $O(cn)$, where c is a constant. The loop in section C) clearly requires $O(n)$ time. Overall, *p-suffix-sort-pKR* requires $O(n)$ time. □

The idea used in the algorithm *p-suffix-sort-pKR* is unique, but assumes that the p-string fingerprints fit into practical integer representations. This assumption is primarily a limitation inherent to fingerprinting. It is well documented that KR integral fingerprints can

be large and exceed the extremes of an integer with large strings and vast alphabets. The modulo operation discussed in [10, 11, 29] is used to handle this problem. However, the modulo operation will not preserve the lexicographical ordering between fingerprints and creates a new problem with respect to suffix sorting. Even if we use fingerprints to encode prefixes of p-suffixes, the codes can still be quite large with collisions. We extend our idea using arithmetic coding to address these limitations.

3.3 p-Suffix Sorting via Arithmetic Coding

Arithmetic coding compresses a string by recursively dividing up a real number line into intervals that account for the cumulative distribution function (*cdf*), which describes the probability space of each symbol. The interval for an arithmetic code AC is (lo, hi), where lo and hi are the low and high boundaries, respectively. Any consistent choice in this region, say $tag(s) = \frac{s.hi+s.lo}{2}$, represents the arithmetic code and preserves the lexicographical ordering of strings. Arithmetic coding is further described in [30, 31]. Recently, Adjeroh and Nan [8] used a novel application of Shannon-Fano-Elias codes from information theory to address the traditional suffix sorting problem. In the work, they generate arithmetic codes for *m-blocks*, or m-length prefixes of the suffixes, to maintain the ordering of m symbols. They show how to efficiently transition one AC *m-block* code at suffix i to construct the *m-block* AC at suffix $(i+1)$ by removing the symbol at i and appending the symbol at $(i+m)$. The transitioning scheme is illustrated in Figure 3.1. Then, the suffixes are recursively partitioned and the generated *m-block* arithmetic codes are exploited to induce the ordering of the partitions in linear time. Extending the suffix sorting via arithmetic coding algorithm given in [8] to the p-suffix sorting problem is not straightforward because of the differing relationship between p-suffixes, identified in Lemma 3.2.1.

Given an n-length p-string T, we wish to generate the parameterized arithmetic code pAC for the *m-blocks*, or m-length prefixes, of the n p-suffixes of T. The distribution of symbols plays a role in the size of the intervals and hence the tag, but this does not change the lexicographic order of the generated arithmetic codes. Thus, without loss of generality, we assume each symbol $x \in (\Sigma \cup \mathbb{Z} \cup \{\$\})$ in the alphabet of a *prev* encoding to be equally

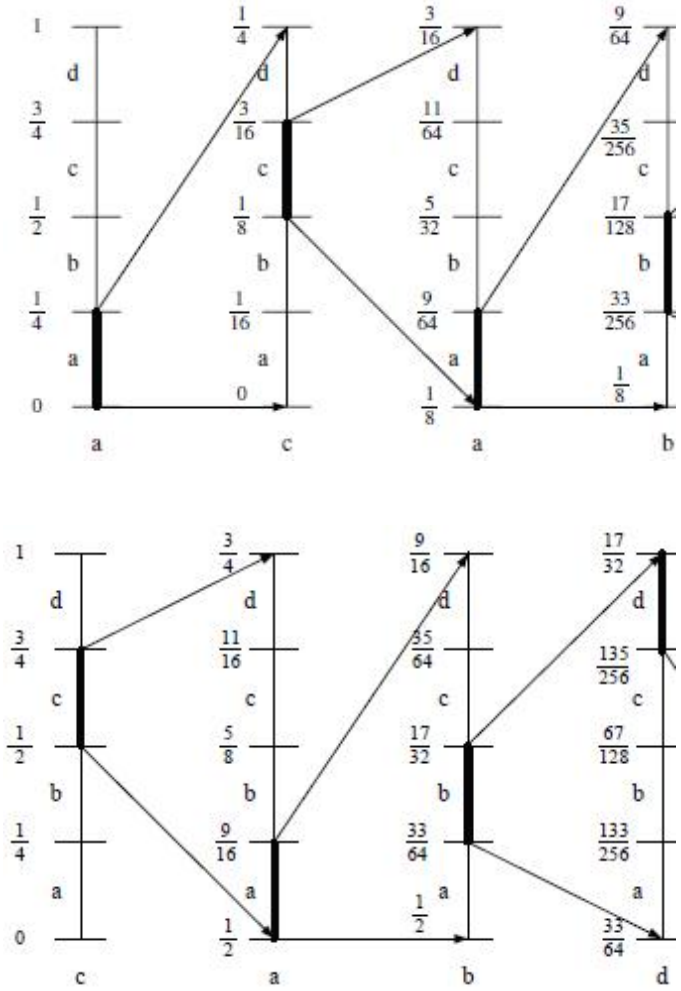

Figure 3.1: Transitioning the AC *m-block* code from $\boxed{a}cab \rightarrow cab \rightarrow cab\boxed{d}$

Listing 3.2: Generating arithmetic codes for an m-length prefix of p-suffix i

```
1  struct AC { long double lo, long double hi }
2  AC pAC(int i, int m) {
3    AC new=(0,0), AC old=(0,1), int end=min{i+m-1,n}
4    char[] prevTi=prev(T[i...end])
5    for k=i to end, step 1 {
6      new.hi=old.lo+(old.hi-old.lo)*cdf[map(prevTi[k-i+1])]
7      new.lo=old.lo+(old.hi-old.lo)*cdf[map(prevTi[k-i+1])-1]
8      old=new
9    }return new
10 }
```

probable, where p represents the probability of a symbol and the array *cdf* contains the values of the uniform cdf with respect to the neighboring lexicographical alphabet symbols. The following definition modifies the traditional AC algorithm to create an *m-block* arithmetic code for a p-suffix at position i in T.

Definition 3.3.1 Parameterized arithmetic coding (pAC) function: *For an n-length p-string T, Listing 3.2 will generate an arithmetic code interval for the m-block of the p-suffix starting at position i.*

Table 3.2 shows the pAC codes for *m-blocks* of $m = 2, 3, n$ of p-string $T = AwBzABwz\$$. We notice that a "collision" occurs for two pAC codes using $m = 2$ since the *m-blocks* are equivalent. Even though the pAC codes distinctly sort the n p-suffixes of T when m approaches n, we highlight that the practical limitation is arithmetic precision. See [8, 30] for handling this problem.

In order to use the m-block codes, we must generate them efficiently. We denote the m-block arithmetic code at p-suffix i by pAC_i. The idea is to first use function pAC to compute pAC_1 and use this code to generate the remaining $(n-1)$ codes, namely pAC_2, pAC_3, ..., and pAC_n. Iteratively, we will need to adjust the arithmetic codes to 1) remove the old symbol and 2) add the new symbol. These cases are described below. The lemmas are similar in nature to Lemmas 3.2.4 and 3.2.5 exploiting Definitions 2.6.3 and 2.6.8 and are omitted for space.

Table 3.2: Lexicographical ordering of p-suffixes with pAC, using $T = AwBzABwz\$$

i	pSA	T[pSA[i]...n]	prev(T[pSA[i]...n])	tag(pAC(pSA[i],m))		
				m=2	m=3	m=n
1	9	$	$	0.055556	0.055556	0.055556
2	8	z$	0$	0.117284	0.117284	0.117284
3	7	wz$	00$	0.129630	0.124143	0.124143
4	4	zABwz$	0AB04$	0.203704	0.209191	0.208743
5	2	wBzABwz$	0B0AB54$	0.216049	0.211934	0.212459
6	1	AwBzABwz$	A0B0AB54$	0.796296	0.801783	0.801384
7	5	ABwz$	AB00$	0.882716	0.878601	0.878076
8	6	Bwz$	B00$	0.907407	0.903292	0.902683
9	3	BzABwz$	B0AB04$	0.907407	0.911523	0.912083

Case 1: Removing a symbol s from an arithmetic code m-block requires us to simply delete s when $s \in \Sigma$ or $s \in \Pi$ and does not occur in the m-block. When $s \in \Pi$ and occurs later in the m-block, the code must accommodate for both the removed occurrence and the future occurrence of s.

Definition 3.3.2 Remove symbol δ^-_{pAC} transition: *Given the AC code A at m-block i with $(i + m - 1) \leq n$, δ^-_{pAC} supplies the transition to remove the symbol at position i and provide the new code A of the $(m$-1$)$-block at p-suffix $(i + 1)$. Let $\beta = forwT[i]$, $j = i + \beta$, $e = min\{i + m - 1, n\}$, $\lambda = (map(\beta) - map(0)) \times p^{\beta+1}$, and $c = cdf[map(prev(T[i])) - 1]$.*

$$\delta^-_{pAC}(i, A) = \begin{cases} \left(\frac{A.lo-c}{p}, \frac{A.hi-c}{p}\right), \textbf{if } (in(prevT[i], \mathbb{Z}) \wedge j > e) \vee in(prevT[i], \Sigma \cup \{\$\}) \\ \left(\frac{A.lo-\lambda-c}{p}, \frac{A.hi-\lambda-c}{p}\right), \textbf{if } in(prevT[i], \mathbb{Z}) \wedge j \leq e \end{cases}$$

Case 2: Adding (i.e. appending) symbol s to an arithmetic code m-block requires us to simply append a symbol to the code when $s \in \Sigma$ or $s \in \Pi$ and does not occur in the m-block. When $s \in \Pi$ and occurs previously in the m-block, the code must account for the new occurrence in terms of the previous occurrence of s.

Definition 3.3.3 Add symbol δ^+_{pAC} transition: *Given the AC code A at $(m$-1$)$-block $(i - m + 1) \geq 1$, δ^+_{pAC} supplies the transition to add the symbol at position i and provide the new code A of the m-block at p-suffix $(i-m+1)$. Let $b = max\{1, i-m+1\}$, $k = i-prevT[i]$, $\Delta = A.hi - A.lo$, $d = \Delta \times cdf[map(prev(T[i]))]$, $f = \Delta \times cdf[map(prev(T[i])) - 1]$, $v = \Delta \times cdf[map(prevT[i])]$, and $w = \Delta \times cdf[map(prevT[i]) - 1]$)*

$$\delta^{+}_{pAC}(i, A) = \begin{cases} (A.lo + f, A.lo + d), \textbf{if } (in(prevT[i], \mathbb{Z}) \wedge k < b) \vee in(prevT[i], \Sigma \cup \{\$\}) \\ (A.lo + w, A.lo + v), \textbf{if } in(prevT[i], \mathbb{Z}) \wedge k \geq b \end{cases}$$

With the assistance of Definitions 3.3.2 and 3.3.3, we can efficiently generate the *m-block* codes for all n p-suffixes of T. Consider the p-string $T = zwzABA\$$, $\Sigma = \{A, B\}$, $\Pi = \{w, z\}$, and $m = 4$, we successively generate the *m-block* codes in the following fashion: $\boxed{0}0\boxed{2}A \overset{\delta^{-}_{pAC}}{\rightarrow} 00A \overset{\delta^{+}_{pAC}}{\rightarrow} 00A\boxed{B} \rightarrow \cdots$. Given $prevT = prev(T)$ and $forwT = forw(T)$, we can construct all pAC codes in linear time.

Theorem 3.3.4 *Given a p-string T of length n and precalculated variables $prevT$ and $forwT$, the pAC codes for all the m-length prefixes of the p-suffixes require $O(n)$ time to generate.*

Proof Generating the first m-block code pAC_1 via $pAC_1 = pAC(1, m)$ will require $O(m)$ time. Iteratively, the neighboring pAC codes will be used to create the successive p-suffix codes. The first extension of code pAC_1 to create pAC_2 will require the removal of $prevT[1]$ via a call to $pAC_2 = \delta^{-}_{pAC}(1, pAC_1)$, which is $O(1)$ work, and the addition of symbol $prevT[2+m-1]$ via a call to $pAC_2 = \delta^{+}_{pAC}(2 + m - 1, pAC_2)$, which also demands $O(1)$ work. This process requiring two $O(1)$ steps is needed for the remaining $(n-1)$ m-block p-suffixes of T. The resulting time is $O(m + n)$. Since $m \leq n$, the theorem holds. □

The efficient preprocessing from Theorem 3.3.4 leads to our main result: an average case linear time algorithm for direct p-suffix sorting for non-binary parameter alphabets. We discuss the intricacies of the worst case p-suffix array construction in the chapter summary as an area for future work.

Theorem 3.3.5 *Given a p-string T of length n, p-suffix-sorting of T can be accomplished in $O(n)$ time on average via parameterized arithmetic coding.*

Proof We can construct $prev(T)$ in $O(n)$ time given an indexed alphabet and an $O(|\Pi|)$ auxiliary data structure. The lexicographical ordering of the m-block pAC codes follow from the notion of arithmetic coding and Definition 3.2.2. From Theorem 3.3.4, we can create all the m-block pAC codes in $O(n)$ time. Similar to [8], the individual floating-point codes may

be converted to integer codes d_i in the range $[0, c(n-1)]$ by $d_i = \left\lfloor c(n-1)\frac{pAC_i - pAC_{min}}{pAC_{max} - pAC_{min}} \right\rfloor$, where the constant c ($c > 1$) is chosen to best separate the d_i and values pAC_{min} and pAC_{max} correspond to the minimum and maximum pAC codes, respectively. From [32, 33], we know that on average, the maximum LCP for an n-length general string is $O(\log_{|\Sigma|} n)$. Let $\alpha \circ \beta$ be the string concatenation of α and β. Then, $Q = prev(T[1...n-1])\$ \circ prev(T[2...n-1])\$ \circ ... \circ prev(T[n-2...n-1])\$ \circ \$$ contains each individual p-suffix of T. Notice that Q is of length $|Q| = \frac{n(n+1)}{2} \in O(n^2)$ and since all p-suffixes are clearly represented, the symbols of Q may be mapped to a traditional string alphabet, allowing us to use the contribution of [32, 33] to obtain the length of the maximum LCP for the general string Q, which is of the same order $O(\log n^2) \in O(\log n)$. Thus by choosing $m = O(\log n)$, only the first $O(n)$ radix sort of the d_i codes is required to differentiate the p-suffixes, demanding only $O(n)$ operations on average. □

3.4 Summary

Approaching the direct p-suffix sorting problem by representing p-suffixes with fingerprints and arithmetic codes provides new mechanisms to handle the challenges of the p-string. We proposed a theoretical algorithm using fingerprints to p-suffix sort an n-length p-string in $O(n)$ time, with n and the alphabet size constrained in practice. Arithmetic codes were then used to propose an algorithm to p-suffix sort p-strings in linear time on average. An area of future research is the worst case performance for p-suffix sorting, which requires an investigation of the intricate relationship between the dynamic nature of p-suffixes and induced sorting, the fundamental mechanism in worst case linear time direct suffix sorting of traditional strings [8, 12, 14, 15].

Chapter 4

Parameterized Longest Previous Factor

4.1 Introduction

Given an n-length traditional string $W = W[1]W[2]...W[n]$ from the alphabet Σ, the longest previous factor (LPF) problem is to determine the maximum length of a previously occurring factor for each individual suffix occurring in W. More formally, for any suffix u beginning at index i in the string W, the LPF problem is to identify the length of the longest factor between u and another suffix v at some position h before i in W: that is, $1 \leq h < i$. The LPF problem, introduced by Crochemore and Ilie [7], yields a data structure convenient for fundamental applications such as string compression [25] and detecting runs [27] within a string. In order to compute the LPF array, it is shown in [7] that the suffix array SA is required to quickly identify the most lexicographically similar suffixes that constitute as previous factors for the chosen suffix in question. The use of SA expedites the work required to solve the LPF problem and likewise, is the *cornerstone* to solutions for many problems defined for traditional strings.

A generalization of traditional strings over an alphabet Σ is the parameterized string (p-string), introduced by Baker [3]. A p-string is a production of symbols from the alphabets Σ and Π with $\Sigma \cap \Pi = \emptyset$, which represent the constant symbols and parameter symbols respectively. The parameterized pattern matching (p-match) problem is to identify an equivalence

between a pair of p-strings S and T when 1) the individual constant symbols match and 2) there exists a bijection between the parameter symbols of S and T. For example, the following p-strings that represent program statements $z{=}y*f/{+}{+}$y; and $a{=}b*f/{+}{+}b$; over the alphabets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$ satisfy both conditions and thus, the p-strings p-match. The motivation for addressing a problem in terms of p-strings is the range of problems that a single solution can address, including 1) exact pattern matching when $|\Pi| = 0$, 2) mapped matching (m-matching) when $|\Sigma| = 0$ [19], and clearly, 3) p-matching when $|\Sigma| > 0 \wedge |\Pi| > 0$. Prominent applications inherent to the p-match problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [4], discovering cloned code segments in a program [9], and even answering critical legal questions regarding the unauthorized use of intellectual property [28].

In this work, we introduce the parameterized longest previous factor (pLPF) for p-strings analogous to the LPF problem for traditional strings, which can similarly be used to study compression and duplication within p-strings. Given an n-length p-string $T = T[1]T[2]...T[n]$, the pLPF problem is to determine the longest parameterized suffix (p-suffix) v at position h for a p-suffix starting at i in T with $1 \leq h < i$. Our approach uses a parameterized suffix array (pSA) [23, 24] for p-strings analogous to the traditional suffix array [6]. The major difficulty of the pLPF problem is that unlike traditional suffixes of a string, the p-suffixes are *dynamic*, varying with the starting position of the p-suffix. Thus, traditional LPF solutions cannot be directly applied to the pLPF problem.

Main Contributions: We generalize the LPF problem for traditional strings to the parameterized longest previous factor (pLPF) problem defined for p-strings. Then, we present a linear time algorithm for constructing the $pLPF$ data structure. Traditionally, the LPF is solved by using the LCP array. This was the approach used in [7]. In this work, we show how to go in the reverse direction: that is, given the pLPF solution, we now construct the $pLCP$ array. Further, we identify how to exploit our algorithm for the pLPF problem to construct the LPF and LCP arrays. Our main results are stated in the following theorems:

Theorem 4.3.4. *Given an n-length p-string T, $prevT = prev(T)$, the prev encoding of T, and pSA, the parameterized suffix array for T, the algorithm compute_$pLPF$ constructs the*

pLPF array in $O(n)$ time.

Theorem 4.4.2. *Given an n-length p-string T, $prevT = prev(T)$, the prev encoding of T, and pSA, the parameterized suffix array for T, the compute_pLPF algorithm can be used to construct the pLCP array in $O(n)$ time.*

4.2 Preliminaries

In addition to the p-string preliminaries in Section 2.6, we further define the traditional LPF problem.

Definition 4.2.1 ([7]) *Longest previous factor (LPF):* *For an n-length traditional string W, the LPF is defined for each index $1 \leq i \leq n$ such that $LPF[i] = max(\{0\} \cup \{k \mid W[i...n] ==_k W[h...n], 1 \leq h < i\})$.*

The traditional string $W = AAABABAB\$$ yields $LPF = \{0, 2, 1, 0, 4, 3, 2, 1, 0\}$.

4.3 Parameterized LPF

We define the parameterized longest previous factor (pLPF) problem below to generalize the traditional LPF problem to p-strings.

Definition 4.3.1 Parameterized longest previous factor ($pLPF$): *For a p-string T of length n, the pLPF array is defined for each index $1 \leq i \leq n$ to maintain the length of the* longest factor *between a p-suffix and a previous p-suffix occurring in T. More formally, $pLPF[i] = max(\{0\} \cup \{k \mid prev(T[i...n]) ==_k prev(T[h...n]), 1 \leq h < i\})$.*

The pLPF problem requires that we deal with p-suffixes, which are suffixes encoded with *prev*. This task is more demanding than the LPF for traditional strings because Lemma 4.3.2 proves that we cannot guarantee the individual suffixes of a single *prev* encoding to be p-suffixes. Thus, the changing nature of the *prev* encoding poses a major challenge to efficient and correct construction of the $pLPF$ array using current algorithms that construct the LPF array for traditional strings.

Lemma 4.3.2 *Given a p-string T of length n, the suffixes of $prev(T)$ are not necessarily the p-suffixes of T. More formally, if $\pi \in \Pi$ occurs more than once in T, then $\exists i$, s.t. $prev(T[i...n]) \neq prev(T)[i...n], 1 \leq i \leq n$.*

Proof Consider that the *only* parameter symbol to occur in the p-string T is $\pi \in \Pi$, which exists *only* at positions α and β with $\alpha < \beta$. Suppose that indeed $prev(T[\alpha...n]) == prev(T)[\alpha...n]$ and $prev(T[\beta...n]) == prev(T)[\beta...n]$. By Definition 3, the first occurrence of symbol π at position α will be *prev* encoded by 0 and the π at position β will be *prev* encoded by $\beta - \alpha$. So, in the case of suffix α, $prev(T[\alpha...n]) == prev(T)[\alpha...n]$. At suffix β, the encoding of π at position β in T will *change* to 0 in $prev(T[\beta...n])$ by Definition 3 whereas $prev(T)[\beta...n]$ will retain the *old* encoding of $\beta - \alpha$ since symbol π *still* occurs in $prev(T)$ at position α. The π at position β forces $prev(T[\beta...n]) \neq prev(T)[\beta...n]$, which is a contradiction. □

Table 4.1 shows the pLPF computation for a p-string $T = AAAwBxyyAAAzwwB\$$ using the previously defined alphabets. We note the intricacies of Lemma 4.3.2 since simply using the traditional LPF algorithm 1) with T yields $LPF = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 0, 1, 2, 1, 0\}$, 2) with $prev(T)$ produces $LPF = \{0, 2, 1, 0, 0, 1, 1, 0, 4, 3, 2, 1, 0, 1, 1, 0\}$, and 3) with $forw(T)$ generates the following array $LPF = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 3, 2, 1, 1, 0\}$, neither of which is the correct $pLPF$ array.

Crochemore and Ilie [7] efficiently solve the LPF problem for a traditional string W by exploiting the properties of the suffix array SA. They construct the arrays $prev_<[1...n]$ and $prev_>[1...n]$, which for each i in W maintain the suffix $h < i$ positioned respectively before and after suffix i in SA; when no such suffix exists, the element is denoted by -1. It is described in [7] how to compute the $prev_<$ and $prev_>$ arrays in linear time via deletions in a doubly linked list or without loss of generality, another dynamically-sized list data structure of the SA. We will furthermore refer to $prev_<$ and $prev_>$ as $before_<$ and $before_>$ respectively, in order to avoid confusion with the *prev* encoding for p-strings. Similarly, we also define $after_<$ and $after_>$ for each i in W to maintain the suffix $j > i$ positioned before and after suffix i in SA. For completeness, the algorithm in Listing 4.1 constructs the $before$ and $after$ ar-

Table 4.1: $pLPF$ calculation for p-string $T = AAAwBxyyAAAzwwB\$$

i	$pSA[i]$	$pLCP[i]$	$prev(T[pSA[i]...n])$	$before_<[pSA[i]]$	$before_>[pSA[i]]$	$pLPF[i]$
1	16	0	\$	-1	6	0
2	6	0	001AAA001B\$	-1	4	2
3	12	3	001B\$	6	7	1
4	7	1	01AAA001B\$	6	4	0
5	13	2	01B\$	7	8	0
6	8	1	0AAA001B\$	7	4	1
7	14	1	0B\$	8	4	1
8	4	2	0B001AAA091B\$	-1	3	1
9	11	0	A001B\$	4	3	4
10	3	2	A0B001AAA091B\$	-1	2	3
11	10	1	AA001B\$	3	2	2
12	2	3	AA0B001AAA091B\$	-1	1	3
13	9	2	AAA001B\$	2	1	2
14	1	4	AAA0B001AAA091B\$	-1	-1	2
15	15	0	B\$	1	5	1
16	5	1	B001AAA001B\$	1	-1	0

rays in the fashion $construct\_before\_after(SA, b) = \begin{cases} (before_<, before_>), \textbf{ if } b == true \\ (after_<, after_>), \textbf{ otherwise} \end{cases}$.
It is noted that the algorithm in Listing 4.1 applies to any type of suffix array (for traditional strings, p-strings, etc.) since the algorithm is only concerned with the unique existence of integers $[1, n]$ in the suffix array of an n-length string. Clearly, the algorithm runs in $O(n)$ time.

With the $before_<$ and $before_>$ arrays, the element $LPF[i]$ is simply the maximum q between $W[i...n] ==_q W[before_<[i]...n]$ and $W[i...n] ==_q W[before_>[i]...n]$. The magic of a linear time solution to constructing the LPF array is achieved through the computation of an element by *extending* the previous element, more formally $LPF[i] \geq LPF[i-1] - 1$. We show that this same property holds for the pLPF problem defined on p-strings.

Lemma 4.3.3 *The $pLPF$ for a p-string T of length n is such that $pLPF[i] \geq pLPF[i-1]-1$ with $1 < i \leq n$.*

Proof Consider $pLPF[i]$ at $i = 1$ by which Definition 4.3.1 requires that we find a previous factor at $1 \leq h < 1$ that does not exist; i.e. $pLPF[1] = 0$. At $i = 2$, indeed $pLPF[2] \geq pLPF[1] - 1 = -1$ is clearly true for all succeeding elements in which a previous factor does

not exist. For arbitrary $i = j$ with $1 < j < n$, suppose that the maximum length factor is at $g < j$ and without loss of generality, consider that the first $q \geq 2$ symbols match so that $prev(T[j...n]) =_q prev(T[g...n])$. Thus, $pLPF[j] = q$. Shifting the computation to $i = j+1$, we lose the symbols $prev(T[j])$ and $prev(T[g])$ in the p-suffixes at j and g respectively. By Proposition 2.6.5, $prev(T[j...j+q-1]) == prev(T[g...g+q-1]) \Rightarrow prev(T[j]) == prev(T[g])$ and as a consequence of the *prev* encoding in Definition 2.6.3 we have $prev(T[i...n]) =_{q-1} prev(T[g+1...n])$. Since we can guarantee that $\exists$ a factor with $(q-1)$ symbols for $pLPF[i]$ or possibly find another factor at h with $1 \leq h < i$ matching q or more symbols, the lemma holds. □

Lemma 4.3.3 permits us to adapt the algorithm *compute_LPF* given in [7] to p-strings. We introduce *compute_pLPF* in Listing 4.2 to construct the $pLPF$, which makes use of the p-matcher Λ in Listing 4.3 to properly handle the sophisticated matching of p-suffixes, the dynamic suffixes under the *prev* encoding. The role of Λ is to *extend* the matches between the p-suffixes at a and b beyond the initial q symbols by directly comparing constant/terminal symbols and comparing the *dynamically* adjusted parameter encodings for each p-suffix.

Theorem 4.3.4 *Given an n-length p-string T, $prevT = prev(T)$, the prev encoding of T, and pSA, the parameterized suffix array for T, the algorithm compute_pLPF constructs the $pLPF$ array in $O(n)$ time.*

Proof It follows from Lemma 4.3.3 that our algorithm *compute_pLPF* exploits the properties of pLPF to correctly compute and extend factors, which requires $O(n)$ time. Computing the arrays $before_<$ and $before_>$ require $O(n)$ processing [7]. What remains now is to show that, between Listing 4.2 and Listing 4.3, the total number of times that the body of the *while* loop (lines 6-15 in Listing 4.3) will be executed is in $O(n)$. The number of iterations of the *while* loop is given by the number of matching symbol comparisons, namely the number of increments of the variable q, which identifies the shift required to compare the current symbol. Without loss of generality, suppose that the initial p-suffixes at position a and b are the longest suffixes at positions 1 and 2 in T of lengths n and $(n-1)$ respectively. In the worst case, $(n-1)$ of the symbols will match between these suffixes, by which each

Listing 4.1: ($before_<$,$before_>$) and ($after_<$,$after_>$) construction

```
/*
    **** Doubly Linked List ****
    Node struct: struct Node { int suf, Node* previous, Node* next }
    Operations:
    -- void init( ) : initialization routine
    -- Node* insert(int i) : inserts Node with suf i; returns pointer
    -- void delete(Node* ptr) : removes the Node pointed to by ptr
    -- void clear( ) : removes all Nodes
*/
(int[],int[]) construct_before_after(int SA[], boolean b) {
  int q<[n], q>[n], i, j = 1
  Node* ptr[n]
  init( )
  insert(-1)
  for i = 1 to n, step 1
    ptr[SA[i]] = insert(SA[i])
  insert(-1)
  for i = n to 1, step -1 {
   if(b) { // construct (before<,before>)
      q< = ptr[i]->previous->suf
      q> = ptr[i]->next->suf
      delete(ptr[i])
   }else { // construct (after<,after>)
      q< = ptr[j]->previous->suf
      q> = ptr[j]->next->suf
      delete(ptr[j])
      j++
    }
  }clear( )
  return (q<,q>)
}
```

Listing 4.2: *pLPF* computation

```
1  int[] compute_pLPF(int before_<[], int before_>[]) {
2    int  pLPF[n], pLPF_<=0, pLPF_>=0, i, j, k
3    for i = 1 to n, step 1 {
4      j = max{0,pLPF_<−1}
5      k = max{0,pLPF_>−1}
6      pLPF_< = Λ(i,before_<[i],j)
7      pLPF_> = Λ(i,before_>[i],k)
8      pLPF[i] = max{pLPF_<,pLPF_>}
9    }return pLPF
10 }
```

Listing 4.3: p-matcher function Λ

```
1  int Λ(int a, int b, int q) {
2    boolean c = true
3    int x, y
4    if(b == −1) return 0
5    while(c ∧ (a+q)≤n ∧ (b+q)≤n) {
6      x = prevT[a+q], y = prevT[b+q]
7      if(in(x,Σ)∧ in(y,Σ)){
8          if(x == y) q++
9          else c = false
10     }else if(in(x,ℤ)∧ in(y,ℤ)){
11         if(q < x) x = 0
12         if(q < y) y = 0
13         if(x == y) q++
14         else c = false
15     }else c = false
16   }return q
17 }
```

comparison that clearly requires $O(1)$ work, will increment q. Lemma 4.3.3 indicates that succeeding calculations, or calls to Λ, already match at least $(q-1)$ symbols that are not rematched and rather, the match is *extended*. Since the decreasing lengths of the succeeding suffixes at $3, 4, ..., n$ cannot *extend* the current q, no further matching or increments of q are needed. Hence, the *while* loop iterates a total of $O(n)$ times amortized across *all* of the n iterations of the *for* loop in Listing 4.2. Thus, the total work is $O(n)$. □

Our algorithm *compute_pLPF* is motivated by the *compute_LPF* algorithm in [7]. We also observe that similar pattern matching mechanisms as the one used between the for loop and the *while* loop exist in standard string processing, for example in computing the *border* array discussed in [11].

4.4 From pLPF to pLCP

Deguchi et al. [23, 24] studied the problem of constructing the $pLCP$ array given the pSA. They showed that constructing the $pLCP$ array requires a non-trivial modification of the original LCP algorithm of Kasai et al. [34]. In [7], the LCP array was used as the basis for constructing the LPF array for traditional strings. Here, we present a simpler algorithm for constructing the $pLCP$ array. In particular, we show that, unlike in [7], it is possible to go the other way around: that is, given the pLPF solution, we now construct the $pLCP$ array. Later, we show that the same pLPF algorithm can be used to construct the LCP array and the LPF array for traditional strings.

Crochemore and Ilie [7] identify that the traditional LPF array is a permutation of the well-studied LCP array. We observe the same relationship in terms of the $pLPF$ and $pLCP$ arrays.

Proposition 4.4.1 *The $pLPF$ array is a permutation of $pLCP$.*

This observation allows us to view the $pLCP$ array from a different perspective. As a novel use of our *compute_pLPF* algorithm, we introduce a way to construct the $pLCP$ array in linear time. The key observation is integrating the notion that the $pLCP$ occurs between neighboring p-suffixes and the fact that we preprocess the $before_<$ array, which for each i

Listing 4.4: *pLCP* computation

```
1 int[] compute_pLCP(int before_<[], int after_<[]) {
2   int pLCP[n], M[n], R[n], i
3   for i = 1 to n, step 1
4     R[pSA[i]] = i
5   M = compute_pLPF(before_<, after_<)
6   for i = 1 to n, step 1
7     pLCP[R[i]] = M[i]
8   return pLCP
9 }
```

in the p-string T maintains the p-suffix $h < i$ positioned prior to the p-suffix i in pSA. We can also construct the array $after_<$ to maintain the p-suffix $j > i$ also positioned prior to the p-suffix i in pSA (see Listing 4.1). Since h and j are both positioned prior to i in pSA, we can guarantee that either h or j must be the nearest neighbor to i. So, the maximum factor determines the nearest neighbor and thus, $pLCP[R[i]]$, where R is the inverse of pSA (see Listing 4.4). Theorem 4.4.2 shows that this computation is done in linear time.

Theorem 4.4.2 *Given an n-length p-string T, $prevT = prev(T)$, the prev encoding of T, and pSA, the parameterized suffix array for T, the compute_pLPF algorithm can be used to construct the $pLCP$ array in $O(n)$ time.*

Proof We can clearly relax the p-suffix selection restrictions enforced by the problem pLPF in Lemma 4.3.3 to exploit the notion of *extending* factors. Subsequently, only the parameters of Listings 4.2 and 4.3 impose such restrictions. Let $R[1...n]$ be the rank array, the inverse of pSA. We prove that the $pLCP$ is constructed with $compute\_pLPF(before_<, after_<)$. Let $before_<[1...n]$ and $after_<[1...n]$ maintain, for all the i in T, the p-suffixes $h < i$ at position $R[h]$ in pSA and $j > i$ at position $R[j]$ in pSA, respectively, that are positioned prior to the p-suffix i at position $R[i]$ in pSA; when no such suffix exists, the element is denoted by -1. Without loss of generality, suppose that both h and j exist and $2 < i \leq n$, so we have either $R[j] == R[i] - 1$ or $R[h] == R[i] - 1$ as the neighboring p-suffix. So, $max\{plcp(prev(T[h...n]), prev(T[i...n])), plcp(prev(T[j...n]), prev(T[i...n]))\}$ distinguishes which p-suffix h or j is closer to i, identifying the nearest neighbor and in turn, $pLCP[R[i]]$. This statement is utilized in *compute_pLPF* exactly in terms of factors ex-

Listing 4.5: Improved *pLCP* computation

```
1 int[] compute_pLCP(int before_<[], int after_<[]) {
2   int pLCP[n], M[n], i
3   M = compute_pLPF(before_<, after_<)
4   for i = 1 to n, step 1
5     pLCP[i] = M[pSA[i]]
6   return pLCP
7 }
```

cept that the value will be stored in $pLCP[i]$. So, after the computation using the call to *compute_pLPF* (line 5) in Listing 4.4, the rearranging of the resulting array using the rank array R (lines 6-7) produces the required $pLCP$ array. We have yet to prove the time complexity. Since the parameter $after_<$ can be computed in $O(n)$ by deletions and indexing into a doubly linked list similar to $before_<$ (see Listing 4.1) and since *compute_pLPF* executes in $O(n)$ time via Theorem 4.3.4, the theorem holds. □

The algorithm in Listing 4.4 uses the rank array R, but this is only conceptual and thus, may be omitted for practical space. The improved solution is shown in Listing 4.5.

4.5 From pLPF to LPF and LCP

The power of defining the pLPF problem in terms of p-strings is the generalization of a p-string production. A useful property of p-strings is that a special case of the alphabet definitions or composition of symbols will yield a traditional string. Consider the case when $|\Sigma| > 0 \wedge |\Pi| = 0$, then only traditional strings are valid p-string productions. Similarly, when all of the individual symbols σ of a p-string are such that $\sigma \in \Sigma$, this also constitutes a traditional string. Such generalization by the p-string allows us to offer solutions to multiple problems with a single algorithm based on p-strings. We show in Theorems 4.5.1 and 4.5.2 that given a traditional string W, our p-string LPF and LCP algorithms can also compute the traditional LPF and LCP arrays in linear time.

Theorem 4.5.1 *Given an n-length traditional string W, the compute_pLPF algorithm constructs the LPF array in $O(n)$ time.*

Proof Since $W[i] \in \Sigma \ \forall \ i, 1 \leq i < n$ and $W[n] \in \{\$\}$, then by Definition 2.6.1 we have $W \in (\Sigma \cup \Pi)^*\$$, which labels W as a valid p-string. Given this, Theorem 4.3.4 proves that the construction of $pLPF$ for a p-string requires $O(n)$ time. In this special case, W consists of no such symbol $\pi \in \Pi$ so Lemma 4.3.2 identifies that $prev(W[i...n]) == prev(W)[i...n]$ and further $W == prev(W)$ by Definition 2.6.3, so $W[i...n] == prev(W)[i...n]$, which constrains the pLPF in Definition 4.3.1 to the LPF problem in Definition 4.2.1. Thus, Theorem 4.3.4 computes the LPF of W. □

Theorem 4.5.2 *Given an n-length traditional string W, the compute_pLCP algorithm constructs the LCP array in $O(n)$ time.*

Proof In the same manner as Theorem 4.5.1, we may label W as a valid p-string. Given this, Theorem 4.4.2 proves that the construction of $pLCP$ for a p-string requires $O(n)$ time. Mirroring the proof of Theorem 4.5.1, we have $W[i...n] == prev(W)[i...n]$, which constrains the pLCP in Definition 2.6.7 to the traditional LCP problem. Thus, Theorem 4.4.2 computes the LCP of W. □

4.6 Applications

The significance of constructing the LPF array highlighted in [7, 26] is the straightforward algorithm to compute the Lempel-Ziv (LZ) factorization [25]. In turn, the LZ computation through the LPF array benefits from the implementation of a space efficient suffix array, which has clear practical space advantages to the well-documented suffix tree solutions [10, 11] to LZ factorization. Several string applications exist that use the LZ data structure, including the detection of runs [27] and string compression [25]. Computing the $pLPF$ array will similarly assist in simple computation of the LZ array and allows us to study such applications as maximal runs in p-strings, which may be extended to source code plagiarism or redundancies in biological sequences.

4.7 Summary

We introduce the parameterized longest previous factor (pLPF) problem for p-strings, which is analogous to the longest previous factor (LPF) problem defined for traditional strings. A linear time algorithm is provided to construct the $pLPF$ array for a given p-string. The advantage of implementing our solution *compute_pLPF* is that the algorithm may be used to compute the arrays $pLPF$, $pLCP$, LPF, and LCP in linear time, which are fundamental data structures preprocessed for the efficiency of countless pattern matching applications. Each of the proposed algorithms requires $O(n)$ worst case space.

Chapter 5

Structural Matching via Suffix Arrays

5.1 Introduction

A closely related variant of the parameterized string (p-string) is the structural string (s-string), introduced by Shibuya [4, 5]. Recall that a p-string is a production of symbols from a constant symbol alphabet Σ and a parameter alphabet Π with $\Sigma \cap \Pi = \emptyset$. The s-string adds the notion of *complementary* symbols in Π that enables the pattern matching of s-strings, or structural matching (s-match), to further observe the actual intricate composition of symbols. The s-string is used in [4, 5] for RNA structural pattern matching by a structural suffix tree (s-suffix tree). An s-suffix tree is similar in nature to the p-suffix tree [3]. Both were the first solutions for pattern matching with the sophisticated s-string and p-string generalizations. Similarly, both solutions utilize an encoding scheme. The p-suffix tree requires observing suffixes in terms of a *prev* encoding to point to the previous occurrence of a parameter [3]. The s-suffix tree is built by observing the *sencode* scheme that combines the *prev* and *compl* schemes to encode the structure of complementary symbols [4]. Finally, both the p-suffix tree and s-suffix tree solutions suffer from the practical limitations of the memory footprint demanded by a suffix tree implementation.

Main Contributions: We introduce the first suffix array solution to the s-match problem. Similar to the parameterized suffix array solutions for p-strings proposed in Chapter 3, we propose a direct construction of newly defined suffix arrays for the s-string encodings *compl* and *sencode* in linear time on average, *without* the assistance of an s-suffix tree. We

then introduce the longest previous factor (LPF) problem in terms of the s-string encodings *compl* and *sencode* and further compute the respective longest common prefix (LCP) arrays using the observations in Chapter 4. It is highlighted how our individual s-string algorithms may be further used to compute the traditional and parameterized suffix array, LPF, and LCP arrays. We then show how to use the resulting suffix arrays and LCP arrays to expedite the process of s-matching. We state our main results in the following theorems:

Theorem 5.3.13 *Given an s-string T of length n, constructing the sSA, cSA, pSA, and SA can be accomplished in $O(n)$ time on average via structural arithmetic coding.*

Theorem 5.4.8 *Given an n-length s-string T, the algorithm compute_all_LCP can construct the sLCP, cLCP, pLCP, and LCP array in $O(n)$ time.*

Theorem 5.5.1 *Given an n-length s-string T, the sSA, and the sLCP data structure, it is possible to s-match, c-match, p-match, or traditional match an m-length s-string P in $O(m + \log n)$.*

5.2 Preliminaries

As an addendum to the p-string preliminaries in Section 2.6, we present the following to formalize the theory of structural strings (s-strings).

An s-string is an n-length p-string $T = T[1]T[2]...T[n]$ production from the constant symbol alphabet Σ and the parameter alphabet Π with $\Sigma \cap \Pi = \emptyset$. We terminate the string with a terminal $\$ \notin \Sigma \cup \Pi$ to clearly distinguish between suffixes. An s-string is a p-string with the added notion of *complementary* symbols, by which two symbols may uniquely correspond to one another. The notion that the s-string is a p-string allows us to apply the *prev* encoding, $forw$ encoding, and the remaining p-string theory presented in this work. The s-string definition follows.

Definition 5.2.1 ([4]) ***Structural string (s-string):*** *An s-string is a p-string T of length n from $(\Sigma \cup \Pi)^*\$$. A subset of the parameter symbols, say $\{\pi_1, \pi_2\} \subseteq \Pi = \{\pi_1, \pi_2, ..., \pi_{|\Pi|}\}$, may* uniquely *correspond to one another and behave as* complements*, such that only $complement(\pi_1) = \pi_2$ and $complement(\pi_2) = \pi_1$. We further define the alphabet Γ to rep-*

resent the complements within Π *as a set of pairs in the fashion* $\Gamma = \{(\pi_1, \pi_2)\}$.

Consider the alphabet arrangements $\Sigma = \{A, B\}$, $\Pi = \{v, w, x, y, z\}$, and $\Gamma = \{(w, x), (y, z)\}$. These alphabets are used throughout the chapter. Example s-strings include $S = AxBzzywv\$$, $T = AwByyzxv\$$, and $U = AwByyxzv\$$. The analysis of the complement symbols between two s-strings forms the added restrictions of the structural match (s-match) beyond the parameter bijection required by the p-match problem.

Definition 5.2.2 ([4]) ***Structural matching (s-match):*** *A pair of s-strings* S *and* T *are s-matches with* $n = |S|$ *if and only if* $|S| == |T|$ *and each* $1 \leq i \leq n$ *corresponds to one of the following:*

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] == T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b)) \wedge ((c) \vee (d))$ */*parameter bijection AND complement mapping*/*
   - *(a)* $S[i] \neq S[j], T[i] \neq T[j]$ **for any** $1 \leq j < i$
   - *(b)* $S[i] == S[i-q]$ **iff** $T[i] == T[i-q]$ **for any** $1 \leq q < i$
   - *(c)* $S[i] \neq complement(S[j]), T[i] \neq complement(T[j])$ **for any** $1 \leq j < i$
   - *(d)* $S[i] == complement(S[i-q])$ **iff** $T[i] == complement(T[i-q])$ **for any** $1 \leq q < i$

In our working example, S and T s-match. The s-string U does not s-match with either S or T. The act of verifying Definition 5.2.2 between a pair of s-strings is quite involved. Shibuya [4] identifies that we can use the p-string *prev* encoding in Definition 2.6.3 and the *compl* encoding in Definition 5.2.3 jointly to determine an s-match.

Definition 5.2.3 ([4]) ***Complement (compl) encoding:*** *Given* $\mathbb{Z}$ *as the set of non-negative integers, the function compl :* $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ *accepts an s-string* T *of length* n *and produces a string* Q *of length* n *that* 1) *encodes constant/terminal symbols with the same symbol and* 2) *encodes parameters to point to their* **previous** *complementary parameters. More formally,* Q *is constructed of individual* $Q[i]$ *with* $1 \leq i \leq n$ *where:*

$$Q[i] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \textbf{ if } T[i] \in \Pi \wedge T[i] \neq complement(T[j]) \textbf{ for any } 1 \leq j < i \\ i - k, \textbf{ if } T[i] \in \Pi \wedge k = max\{j \mid T[i] == complement(T[j]), 1 \leq j < i\} \end{cases}$$

We observe the similarity between the definition of the *compl* encoding (Definition 5.2.3) and the *prev* encoding (Definition 2.6.3) where $compl(T) == prev(T)$ is true with the mapping structure $(\pi, \pi) \in \Gamma \ \forall \ \pi \in \Pi$. Definition 5.2.3 presents a mechanism to point to the previous complementary symbol for a parameter symbol. By Definition 5.2.1, in the worst case, since each $\pi \in \Pi$ is only the complement of only one other symbol $\pi \in \Pi$, then $|\Gamma| = |\Pi|$ in the maximum size of alphabet Γ. Thus, the *compl* encoding for an s-string T of length n may be constructed in time $O(n \log(min\{n, |\Pi|\}))$ using a balanced tree [4], which also follows from the discussions of Baker [3, 20] and Amir et al. [19] on the dependency of alphabet Π in p-string applications. Note that with an indexed alphabet and an auxiliary $O(|\Pi|)$ mapping structure, we can construct *compl* in $O(n)$ time.

Shibuya [4] proves that the s-match may be achieved by comparing the *prev* and *compl* encodings between a pair of s-strings.

Proposition 5.2.4 ([4]) *Two s-strings S and T s-match when $prev(S) == prev(T) \wedge compl(S) == compl(T)$.*

Our example using $S = AxBzzywv\$$, $T = AwByyzxv\$$, $U = AwByyxzv\$$, and $\Gamma = \{(w, x), (y, z)\}$ yields the following *compl* encodings: $compl(S) == compl(T) = A0B00150\$$ and $compl(U) = A0B00420\$$. Moreover, $prev(S) == prev(T) == prev(U) = A0B01000\$$. By Proposition 5.2.4, we verify that S and T indeed s-match.

It is proven in [4] that comparing the encodings *sencode*, formalized in Definition 5.2.5, equivalently achieves the same s-match comparison of Proposition 5.2.4, which uses both of the encodings *prev* and *compl*. We refer to *prev*, *compl*, and *sencode* as structural encodings (s-encodings). The alternative s-match scheme is presented in Proposition 5.2.6.

Definition 5.2.5 ([4]) ***Structural encoding (sencode):*** *Given $\mathbb{Z}$ as the set of non-negative integers, the function sencode : $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ accepts an s-string T of length n and produces a string Q of length n that* 1) *encodes constant/terminal symbols with the same symbol and either* 2a) *encodes parameters to point to an existing* **previous** *parameter or* 2b) *encodes remaining parameters to point to* **previous** *complementary parameter symbols. More formally, Q is constructed of individual $Q[i]$ with $1 \leq i \leq n$ where:*

$$Q[i] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ prev(T)[i], \textbf{ if } prev(T)[i] > 0 \\ compl(T)[i], \textbf{ if } compl(T)[i] > 0 \\ 0, \textbf{ otherwise} \end{cases}$$

Proposition 5.2.6 ([4]) *Two s-strings S and T s-match when $sencode(S) == sencode(T)$.*

In our working example, $sencode(U) = A0B01420\$$ and $sencode(S) == sencode(T) = A0B01150\$$. Thus, S and T are again confirmed to s-match.

When working with suffix structures, it is a necessity to obtain any symbol of a chosen suffix. It is identified in [4] that, similar to the intricacies of p-suffixes encoded by the *prev* encoding (see Lemma 3.2.1), the suffixes at position i, $1 \leq i \leq n$, of $compl(T)$ and $sencode(T)$ are not necessarily the *compl* suffixes (c-suffixes) $compl(T[i...n])$ and *sencode* suffixes (s-suffixes) $sencode(T[i...n])$ of some n-length s-string T. The following functions permit constant time access to the individual symbols of the suffixes of the s-encodings.

Definition 5.2.7 ([4]) ***c-suffix and s-suffix symbol retrieval:*** *Given an n-length s-string T, let $prevT = prev(T)$, $complT = compl(T)$, and $\mathbb{Z}$ represent the set of non-negative integers. Further, let $i, j \in \mathbb{Z}$ such that $1 \leq i \leq n$ and $1 \leq j \leq (n - i + 1)$. The function compl : $(i, j) \rightarrow (\Sigma \cup \mathbb{Z} \cup \{\$\})$ retrieves the symbol j of the compl suffix (c-suffix) $compl(T[i...n])$.*

$$compl(i, j) = compl(T[i...n])[j] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ complT[j + i - 1], \textbf{ if } 0 < complT[j + i - 1] < j \\ 0, \textbf{ otherwise} \end{cases}$$

The function sencode : $(i, j) \rightarrow (\Sigma \cup \mathbb{Z} \cup \{\$\})$ retrieves the symbol j of the sencode suffix (s-suffix) $sencode(T[i...n])$.

$$sencode(i, j) = sencode(T[i...n])[j] = \begin{cases} T[i], \textbf{ if } T[i] \in (\Sigma \cup \{\$\}) \\ prevT[j + i - 1], \textbf{ if } 0 < prevT[j + i - 1] < j \\ complT[j + i - 1], \textbf{ if } 0 < complT[j + i - 1] < j \\ 0, \textbf{ otherwise} \end{cases}$$

We note that the alphabet membership $x \in X$ questions of Definition 5.2.7 may be answered instantaneously via function $in(x, X)$ utilizing our *map* function from Definition 3.2.2 by simply adjusting $maxP$ so that $maxP = max\{maxdist(prev(T)), maxdist(compl(T))\}$.

For completeness, we introduce the complementary matching (c-match) problem, which utilizes only the *compl* encoding of the s-match problem in Definition 5.2.2.

Definition 5.2.8 ***Complementary matching (c-match):*** *A pair of s-strings S and T are c-matches with $n = |S|$ if and only if $|S| == |T|$ and each $1 \leq i \leq n$ corresponds to one of the following:*

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] == T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b))$ */*complement mapping*/*
   (a) $S[i] \neq complement(S[j]), T[i] \neq complement(T[j])$ **for any** $1 \leq j < i$
   (b) $S[i] == complement(S[i-q])$ **iff** $T[i] == complement(T[i-q])$ **for any** $1 \leq q < i$

The following proposition identifies how to detect a c-match.

Proposition 5.2.9 *Two s-strings S and T c-match when $compl(S) == compl(T)$.*

Our example s-strings $S = AxBzzywv\$$ and $T = AwByyzxv\$$ are said to c-match since $compl(S) == compl(T) = A0B00150\$$.

In the context of s-strings, we continue to use the p-string theory previously established in this work for portability of concept and concision. We further highlight nontrivial modifications in order to utilize p-string theory in terms of s-strings.

5.3 Constructing *compl* and *sencode* Suffix Arrays

Fast and space efficient pattern matching involves the suffix array (SA) data structure. The problem of constructing the SA, known as suffix sorting, requires sorting the individual suffixes of a string into a lexicographical order. Direct suffix sorting requires constructing the suffix array *without* the use of a suffix tree. Adjeroh and Nan [8] show how to directly suffix sort traditional strings by first encoding *m-blocks*, short prefixes of the individual suffixes of

a string, with information theoretic codes and subsequently sorting the codes. We showed in Chapter 3 how to utilize this approach to p-suffix sort the individual p-suffixes of a p-string to construct the p-suffix array (pSA) in linear time on average. We utilize this same scheme for suffix sorting the s-encodings.

We now define the suffix arrays for the s-encodings.

Definition 5.3.1 *compl* ***suffix array*** *(cSA): The c-suffix array cSA for an s-string T of length n maintains a lexicographical ordering of the indices i representing individual c-suffixes $compl(T[i...n])$ with $1 \leq i \leq n$, such that $compl(T[cSA[q]...n]) \prec compl(T[cSA[q+1]...n])$ $\forall\ q, 1 \leq q < n$.*

Definition 5.3.2 *sencode* ***suffix array*** *(sSA): The s-suffix array sSA for an s-string T of length n maintains a lexicographical ordering of the indices i representing individual s-suffixes $sencode(T[i...n])$ with $1 \leq i \leq n$, such that $sencode(T[sSA[q]...n]) \prec sencode(T[sSA[q+1]...n])$ $\forall\ q, 1 \leq q < n$.*

Similar in nature to the *dynamic* p-suffixes discussed in Chapter 3, the suffixes under the *compl* and *sencode* encoding schemes also vary depending on the start of the suffix. Recall the alphabets used throughout this chapter: $\Sigma = \{A, B\}$, $\Pi = \{v, w, x, y, z\}$, and $\Gamma = \{(w, x), (y, z)\}$. Consider the s-string $Awvx\$$ and the individual suffixes: $\boxed{Awvx\$} \rightarrow \boxed{wvx\$} \rightarrow \boxed{vx\$} \rightarrow \boxed{x\$} \rightarrow \boxed{\$}$. Notice that each traditional suffix is very closely related, i.e. by removing a symbol, we simply obtain a suffix. Now, consider the *compl* encoded suffixes from the s-string $Awvx\$$, which follow: $\boxed{A002\$} \rightarrow \boxed{002\$} \rightarrow \boxed{00\$} \rightarrow \boxed{0\$} \rightarrow \boxed{\$}$. The *compl* encoded suffixes are *dynamically* changing. Likewise, the *sencode* suffixes share a similar *dynamic* behavior. This is similar to, though not exactly the same as, the behavior of adjacent p-suffixes in a p-string. The varying nature and exact relationships between the encoded suffixes is a consequence of the Definitions 5.2.3 and 5.2.5 for *compl* and *sencode* respectively, which differ from the p-suffix under the *prev* encoding in Definition 2.6.3. Thus, traditional or parameterized suffix sorting approaches cannot be applied in a straightforward manner.

At this point, we reach a crossroad. We can *naïvely* solve the cSA and sSA problems individually or further study the relationship between the *compl* and *sencode* encoding schemes,

in order to propose a single solution to address both problems. For conciseness, we further observe the relationship between the *sencode* and the *compl* schemes to introduce a common solution. Indeed, the encodings *compl* and *sencode* are related as it is obvious from Definition 5.2.5 that in addition to the *prev* encoding, *sencode* also depends on *compl*. A significant observation used in this work is that we can exploit the retrieval function in Definition 5.2.7 to force the $sencode(i, j)$ function to behave like the $compl(i, j)$ function. Lemma 5.3.3 supplies the proof that a single solution that utilizes the function $sencode(i, j)$ may be easily manipulated to address the same problem in terms of the *compl* encoding.

Lemma 5.3.3 *Given an n-length s-string T, $prevT = 0^n$, and $complT = compl(T)$, the function sencode(i, j) in Definition 5.2.7 simulates compl(i, j) for c-suffixes.*

Proof Since $prevT = 00...0 = 0^n$ and $prevT[k] = 0 \ \forall \ k, 1 \leq k \leq n$, then $0 \not< prevT[k]$ for all such k. Let σ and π represent the constant/terminal and parameter symbols in T, respectively. Thus, $sencode(i, j)$ never returns a symbol in $prevT$ and mirrors the $compl(i, j)$ function in Definition 5.2.7 by 1) encoding each $\sigma \in (\Sigma \cup \{\$\})$ with the same symbol and 2) encoding each $\pi \in \Pi$ to the distance of the previous symbol $\pi' = complement(\pi)$ within T. □

Moreover, since $sencode(i, j)$ also utilizes the *prev* encoding, we show how to further exploit the function for both p-strings in Lemma 5.3.4 and traditional strings in Lemma 5.3.5.

Lemma 5.3.4 *Given an n-length s-string T, $complT = 0^n$, and $prevT = prev(T)$, the function sencode(i, j) in Definition 5.2.7 simulates prev$(T[i...n])[j]$ for p-suffixes.*

Proof Similar to the proof of Lemma 5.3.3, since $complT[j] = 0 \ \forall \ k, 1 \leq i \leq n$, then $0 \not< complT[k]$ for all such k. Let σ and π represent the constant/terminal and parameter symbols in T, respectively. Thus, $sencode(i, j)$ will never return a symbol in $complT$ and is restricted to 1) the encoding of $\sigma \in (\Sigma \cup \{\$\})$ with the same symbol and 2) the encoding of $\pi \in \Pi$ to the distance of the previous π in T as formalized by *prev* in Definition 2.6.3. □

Lemma 5.3.5 *Given an n-length s-string T, let $\Sigma = (\Sigma \cup \Pi)$ then $\Pi = \emptyset$, and $prevT = prev(T)$, the function $sencode(i, j)$ in Definition 5.2.7 simulates $T[i + j - 1]$ for traditional suffixes.*

Proof Since the symbols in T are such that $T[k] \in (\Sigma \cup \{\$\})\ \forall\ k, 1 \leq k \leq n$, then $prev(x) = x$ with $x \in (\Sigma \cup \{\$\})$ by Definition 2.6.3 and $prevT[i...n] == prev(T[i...n])$ by Lemma 3.2.1. Subsequently, *sencode* will retrieve the constant symbols at $k = i + j - 1$ for every $k, 1 \leq k \leq n$, namely $prevT[k] == T[k]$. □

Lemmas 5.3.3, 5.3.4, and 5.3.5 prove that a solution which uses the $sencode(i, j)$ function for pattern matching can be used for various solutions. This is a significant step for us to construct the cSA and sSA using the mechanisms in Chapter 3. Since our solutions in Chapter 3 involve more than pattern matching, we will require further observations.

Recall the p-suffix sorting of Chapter 3. We use an information theoretic scheme to encode each m-length *m-block* parameterized arithmetic code (pAC). The resulting codes maintain the lexicographical ordering between the p-suffixes, which permits sorting the numbers to in turn, sort the p-suffixes in linear time on average. An example of the pAC codes is displayed in Table 5.1. The key to efficiently generating each pAC is to exploit the fact that neighboring p-suffixes and neighboring *m-block* pAC codes share the same relationship. By using this relationship, we can obtain a neighboring *m-block* code by simply shifting the neighboring code by removing the old symbol, adding a new symbol, and adjusting any changed symbol. The challenge of creating the *m-blocks* for p-suffixes is the *dynamic* nature of the symbols in each p-suffix since the positioning of a parameter in the p-suffix will alter the *prev* encoding and hence, change the p-suffix *m-block* code. We handle this in our pAC algorithm by maintaining the forward distance to the changing parameters between the pAC codes via the $forw$ data structure, which encodes the forward distance from $\pi \in \Pi$ to the succeeding π in the p-string. We employ a similar idea in our $cforw$ data structure in terms of complementary characters of the *compl* encoding, which is fundamental to the $sencode(i, j)$ function described earlier. Since neighboring c-suffixes need to adjust several complementary symbols to comply with the encoding *compl*, it is required that *we maintain the forward distance from a symbol $\pi \in \Pi$ to all of the complementary symbols*

$\pi' = complement(\pi)$ *occurring before the next instance of* π *in the s-string.*

Table 5.1: Lexicographical ordering of p-suffixes with pAC, using $T = AwxyBwzw\$$

i	pSA	$T[pSA[i]...n]$	$prev(T[pSA[i]...n])$	$tag(pAC(pSA[i], m))$		
				$m = 2$	$m = 3$	$m = n$
1	9	\$	\$	0.055556	0.055556	0.055556
2	8	w\$	0\$	0.117284	0.117284	0.117284
3	7	zw\$	00\$	0.129630	0.124143	0.124143
4	2	wxyBwzw\$	000B402\$	0.129630	0.125514	0.126135
5	6	wzw\$	002\$	0.129630	0.128258	0.127648
6	3	xyBwzw\$	00B002\$	0.129630	0.135117	0.134606
7	4	yBwzw\$	0B002\$	0.216049	0.211934	0.211452
8	1	AwxyBwzw\$	A000B402\$	0.796296	0.792181	0.791793
9	5	Bwzw\$	B002\$	0.907407	0.903292	0.903072

Definition 5.3.6 *compl* ***forward*** *(*$cforw$*)* ***encoding:*** *Given an s-string* T *of length* n*, let* $forwT = forw(T)$ *and* $complT = compl(T)$*. We define the function* $cforw$ *for the compl encoding of* T*, namely* $compl(T)$*, as an extension to the* fw *encoding by Deguchi et al.* [23, 24] *for p-strings. Function* $cforw$ 1) *encodes constant/terminal symbols with the same symbol and* 2) *encodes each parameter* p *at position* i *with the* **forward** *distance to* all *occurrences of* $complement(p)$ *prior to the next occurrence of* p *at position* $forwT[i]$ *or in the case of no future occurrences of* $complement(p)$*, an unreachable forward distance* n*. More formally,* $cforw$ *produces an output encoding* G *with* $cforw(T) = G$ *for each* $1 \leq i \leq n$*:*

$$G[i] = \begin{cases} \{complT[i]\}, \textbf{\textit{if}}\ in(complT[i], \Sigma \cup \{\$\}) \\ \{n\}, \textbf{\textit{if}}\ in(complT[i], \mathbb{Z}) \wedge \nexists\ T[k],\ s.t.\ T[i] == complement(T[k]), i < k < forwT[i] \\ \bigcup_{k=i+1}^{forwT[i]-1} \{k - i \mid T[i] == complement(T[k])\}, \textbf{\textit{otherwise}} \end{cases}$$

Our proposed $cforw$ data structure, which is constructed using the algorithm in Listing 5.1, maintains for each $\pi \in \Pi$ the **forward** distance *to all such* parameters equivalent to $complement(\pi)$ preceding the next instance of π in the s-string. For example, $T = AwxyBwzw\$$ with $n = 9$ yields $cforw(T) = \{A\}\{1\}\{3,5\}\{3\}\{B\}\{9\}\{9\}\{9\}\{\$\}$. We can represent the $cforw$ encoding by using a space-friendly 2-dimension jagged array, where constants, terminals, and some parameter symbols require only a singleton array and other parameters may require a longer array of elements. The actual number of elements in the

$cforw$ encoding is evident in Lemma 5.3.7. Indexing into the structure is illustrated in the following examples: $cforw(T)[1][1] == A$, $cforw(T)[2][1] == 1$, $cforw(T)[3] == \{3, 5\}$, $|cforw(T)[3]| == 2$, and $cforw(T)[3][2] == 5$.

Lemma 5.3.7 *Given an n-length s-string T and $complT = compl(T)$, the algorithm construct_cforw computes $cforw(T)$ in $O(n)$ time using $O(n)$ space.*

Proof It is obvious that *construct_cforw* computes the $cforw$ encoding in Definition 5.3.6. Clearly, the time complexity of algorithm *construct_cforw* is $O(n)$ and since the algorithm generates the individual elements in the encoding, the total space is also $O(n)$. Moreover, the actual space requirement is enforced by considering the worst case example for a single array in the $cforw$ structure. Without loss of generality, suppose that we only consider parameters in the s-string $T = \pi_1\pi_2^{n-2}\pi_1$ from the alphabets $\Sigma = \emptyset$, $\Pi = \{\pi_1, \pi_2\}$, and $\Gamma = \{(\pi_1, \pi_2)\}$. Consider the first instance of π_1 in T at position $\alpha = 1$. Then, the succeeding location of π_1 in T occurs at $forw(T)[\alpha] = \beta$ via Definition 2.6.8 with $\beta = n$ and all symbols in the range (α, β) are such that $T[q] == \pi_2 \ \forall \ q, \alpha < q < \beta$. Then, $cforw(T)[\alpha] = \{(q - complT[q]) == \alpha \mid \alpha < q < \beta\}$ and it is true that each $q \in cforw(T)[\alpha]$ obtains the forward distance to all of the π_2 symbols preceding the π_1 at position β in T, which requires $|cforw(T)[\alpha]| = (\beta - \alpha - 1) = (n - 2)$ elements. Further, the individual $T[q]$ containing π_2 at $\alpha < q < \beta - 1$ will encode the forward distance singleton $\{n\}$, since they are directly succeeded by another π_2 symbol, which requires $(n - 3)$ elements. Moreover, $cforw(T)[\beta - 1] = \{\beta - (\beta - 1)\} = \{1\}$ and $cforw(T)[n] = \{n\}$, a total of 2 elements. Thus, the total number of elements encoded is $(n - 2) + (n - 3) + 2 \in O(n)$. □

We identify that the algorithm *construct_cforw* may construct both $cforw$ in Definition 5.3.6 and also, the $forw$ encoding in Definition 2.6.8.

Lemma 5.3.8 *Given an n-length s-string T with all pairs in Γ of the form (π, π), the algorithm construct_cforw computes $forw(T)$ in $O(n)$ time.*

Proof Since algorithm *construct_cforw* computes $cforw(T)$ and $(\pi, \pi) \in \Gamma \ \forall \ \pi \in \Pi$, then $compl(T) == prev(T)$ and $forw(T)$ is directly computed from $prev(T)$ by Definition 2.6.8. □

Listing 5.1: *cforw* construction

```
1  int[][] construct_cforw( ) {
2    int C[n][1], i, j
3    // primary encoding
4    for i = 1 to n, step 1 {
5      if in(complT[i],Σ∪{$})
6        C[i][1] = complT[i]
7      else
8        C[i][1] = n
9    }
10   // further encode parameters
11   for i = n to 1, step −1 {
12     if in(complT[i],ℤ) {
13       j = i − complT[i]
14       if C[j][1] == n
15         C[j][1] = complT[i]
16       else
17         C[j] = complT[i] ∪ C[j]
18     }
19   }return C
20 }
```

With the generalized matching provided by the function $sencode(i, j)$ given by Lemmas 5.3.3, 5.3.4, and 5.3.5 and the generalization of the $cforw$ data structure given by Lemma 5.3.8, we can further pursue the suffix sorting for sSA, cSA, pSA and SA by encoding an *m-block* of an s-suffix with a structural arithmetic code (sAC). We can *naïvely* construct an sAC code via Definition 5.3.9.

Definition 5.3.9 ***Structural arithmetic coding (sAC) function:*** *For an n-length s-string T, the algorithm in Listing 5.2 will generate an arithmetic code interval for the m-block of the s-suffix starting at position i.*

In the same fashion as Chapter 3, we assume the use of a uniform *cdf*, where each symbol has the same probability p. Also, the function *tag* is used to determine the midpoint of the sAC code. The examples in Table 5.2 and Table 5.3 display the codes produced by the sAC function in terms of both c-suffixes and s-suffixes.

To avoid the theoretical backlog of generating the *m-block* codes using Definition 5.3.9, we introduce the δ_{sAC} functions to efficiently transition the codes between neighboring s-

Listing 5.2: Generating arithmetic codes for an m-length prefix of s-suffix i

```
1 struct AC { long double lo, long double hi }
2 AC sAC(int i, int m) {
3   AC new=(0,0), AC old=(0,1), int end=min{i+m-1,n}
4   for k=i to end, step 1 {
5     new.hi=old.lo+(old.hi-old.lo)*cdf[map(sencode(i,k))]
6     new.lo=old.lo+(old.hi-old.lo)*cdf[map(sencode(i,k))-1]
7     old=new
8   }return new
9 }
```

Table 5.2: Lexicographical ordering of c-suffixes with sAC, using $T = AwxyBwzw\$$

i	cSA	$T[cSA[i]...n]$	$compl(T[cSA[i]...n])$	$tag(sAC(cSA[i], m))$		
				$m = 2$	$m = 3$	$m = n$
1	9	\$	\$	0.055556	0.055556	0.055556
2	8	w\$	0\$	0.117284	0.117284	0.117284
3	7	zw\$	00\$	0.129630	0.124143	0.124143
4	6	wzw\$	000\$	0.129630	0.125514	0.124905
5	3	xyBwzw\$	00B335\$	0.129630	0.135117	0.135120
6	2	wxyBwzw\$	010B335\$	0.141975	0.137860	0.138470
7	4	yBwzw\$	0B030\$	0.216049	0.211934	0.211877
8	1	AwxyBwzw\$	A010B335\$	0.796296	0.793553	0.793163
9	5	Bwzw\$	B000\$	0.907407	0.903292	0.902767

suffixes, which are not straightforward extensions to the transitioning functions of Chapter 3. We denote the sAC code at m-*block* i by sAC_i. Our goal is to transition the code sAC_i to sAC_{i+1}. In performing this transition, we have to consider two cases: adding a symbol and removing a symbol.

Case 1: Removing a symbol s from the start of an arithmetic code m-block requires us to simply delete s when $s \in (\Sigma \cup \{\$\})$. When $s \in \Pi$, we must adjust the next occurrence of s and the forward occurrences of $complement(s)$ pointed to by the encoding $cforw$ that precede the next instance of s within the m-block.

Definition 5.3.10 ***Remove symbol*** δ^-_{sAC} ***transition:*** *Given the AC code A at m-block i with $q = (i + m - 1) \leq n$, δ^-_{sAC} supplies the transition to remove the symbol at position i and provide the new code A of the (m-1)-block at s-suffix $(i + 1)$. Let $cforwT = cforw(T)$, $forwT = forw(T)$, $\lambda_1 = \sum_{j=1}^{|cforwT[i]|}[\ (map(sencode(i,k)) - map(0)) \times p^{k+1}$ iff $k \leq min\{q, n\}$*

Table 5.3: Lexicographical ordering of s-suffixes with sAC, using $T = AwxyBwzw\$$

i	sSA	$T[sSA[i]...n]$	$sencode(T[sSA[i]...n])$	$tag(sAC(sSA[i], m))$		
				$m = 2$	$m = 3$	$m = n$
1	9	\$	\$	0.055556	0.055556	0.055556
2	8	w\$	0\$	0.117284	0.117284	0.117284
3	7	zw\$	00\$	0.129630	0.124143	0.124143
4	6	wzw\$	002\$	0.129630	0.128258	0.127648
5	3	xyBwzw\$	00B332\$	0.129630	0.135117	0.135114
6	2	wxyBwzw\$	010B432\$	0.141975	0.137860	0.138486
7	4	yBwzw\$	0B032\$	0.216049	0.211934	0.211910
8	1	AwxyBwzw\$	A010B432\$	0.796296	0.793553	0.793165
9	5	Bwzw\$	B002\$	0.907407	0.903292	0.903072

$\wedge\ cforwT[i][j] < forwT[i], k = i + cforwT[i][j]\ ]$, $\lambda_2 = [\ (map(sencode(i, forwT[i])) - map(sencode(i{+}1, forwT[i]{-}1))) \times p^{\beta+1}$ iff $forwT[i] < n\ ]$, *and* $c = cdf[map(sencode(i, i)) - 1]$.

$$\delta^{-}_{sAC}(i, A) = \begin{cases} \left(\frac{A.lo-c}{p}, \frac{A.hi-c}{p}\right), \textbf{if } in(sencode(1, i), \Sigma \cup \{\$\}) \\ \left(\frac{A.lo-\lambda_1-\lambda_2-c}{p}, \frac{A.hi-\lambda_1-\lambda_2-c}{p}\right), \textbf{ otherwise} \end{cases}$$

Case 2: Adding (i.e. appending) symbol at a position i to the arithmetic code is simply accomplished by adding the retrieved symbol from the *sencode* function in Definition 5.2.7.

Definition 5.3.11 ***Add symbol*** δ^{+}_{sAC} ***transition:*** *Given the AC code A at (m-1)-block $q = (i{-}m{+}1) \geq 1$, δ^{+}_{sAC} supplies the transition to add the symbol at position i and provide the new code A of the m-block at s-suffix q. Let $\Delta = A.hi{-}A.lo$, $d = \Delta \times cdf[map(sencode(q, i))]$, and $f = \Delta \times cdf[map(sencode(q, i)) - 1]$. Then, $\delta^{+}_{sAC}(i, A) = (A.lo + f, A.lo + d)$.*

At this point, we have devised all of the generalizations and developed the definitions required to pass the remaining details to the theorems of Chapter 3 in order to generate the s-encoding suffix arrays via m-block arithmetic codes that represent the encoded suffixes.

Theorem 5.3.12 *Given an s-string T of length n, the sAC codes for all the m-length prefixes of the s-suffixes can be generated in $O(n)$ time.*

Proof Similar to Theorem 3.3.4, we can generate the m-block codes for s-suffixes. We generate the first m-block code sAC_1 via $sAC_1 = sAC(1, m)$, which will require $O(m)$

time. Iteratively, we transition the codes to generate neighboring codes by first removing the leading symbol by $sAC_2 = \delta^-_{sAC}(1, sAC_1)$ and then adding the symbol at position $(2+m-1)$ via a call to $sAC_2 = \delta^+_{sAC}(2+m-1, sAC_2)$. Since we are looping to generate n codes, δ^+_{sAC} requires $O(1)$ time, and in the worst case, δ^-_{sAC} observes each element of the $O(n)$ $cforw$ encoding a single time amortized across n iterations, the theorem holds. □

Theorem 5.3.13 *Given an s-string T of length n, constructing the sSA, cSA, pSA, and SA can be accomplished in $O(n)$ time on average via structural arithmetic coding.*

Proof Since it is possible to generate arithmetic codes that represent m-block s-suffixes in $O(n)$ time from Theorem 5.3.12, we can suffix sort the s-suffixes represented by the arithmetic codes to construct the sSA by Theorem 3.3.5 in $O(n)$ time on average. Since Lemmas 5.3.3, 5.3.4, 5.3.5, and 5.3.8 prove that the $sencode(i, j)$ and $cforw$ schemes used in Definitions 5.3.10 and 5.3.11 may be generalized to handle s-suffixes, c-suffixes, p-suffixes, and traditional suffixes, then Theorem 3.3.5 can be used to construct sSA, cSA, pSA, and SA. □

5.4 Constructing *compl* and *sencode* LCP Arrays

A prerequisite for fast pattern matching with a suffix array SA is to accompany the SA with a corresponding longest common prefix (LCP) array. The LCP problem is to maintain the length of the longest prefix common between two neighboring suffixes in the SA. We show in Chapter 4 how to compute the parameterized LCP ($pLCP$) array for a p-string by working through the parameterized longest previous factor (pLPF) problem for p-strings, which is analogous to the LPF problem for traditional strings. The general problem of LPF, as defined for some string T, is to obtain the length of the longest factor between a suffix i and some suffix h starting prior to i in T. Recall that the pLCP and pLPF problems are defined similarly for the dynamic p-suffixes of a p-string under the *prev* encoding (see Chapter 4). Table 5.4 displays the $pLCP$ and $pLPF$ arrays for the p-string $T = AwxyBwzw\$$.

Table 5.4: $pLCP$ and $pLPF$ computations, using $T = AwxyBwzw\$$

i	pSA	$prev(T[pSA[i]...n])$	$pLCP[i]$	$prev(T[i...n])$	$pLPF[i]$
1	9	\$	0	A000B402\$	0
2	8	0\$	0	000B402\$	0
3	7	00\$	1	00B002\$	2
4	2	000B402\$	2	0B002\$	1
5	6	002\$	2	B002\$	0
6	3	00B002\$	2	002\$	2
7	4	0B002\$	1	00\$	2
8	1	A000B402\$	0	0\$	1
9	5	B002\$	0	\$	0

5.4.1 cLPF and sLPF

Like Chapter 4, we can compute the LCP arrays for the *compl* suffix array (cSA) and the *sencode* suffix array (sSA) by first defining the LPF problem in terms of the s-encodings.

Definition 5.4.1 *compl* **longest previous factor ($cLPF$):** *For an s-string T of length n, the cLPF array is defined for each index $1 \leq i \leq n$ to maintain the length of the* longest factor *between a c-suffix and a previous c-suffix occurring in T. More formally, $cLPF[i] = max(\{0\} \cup \{k \mid compl(T[i...n]) ==_k compl(T[h...n]), 1 \leq h < i\})$.*

Definition 5.4.2 *sencode* **longest previous factor ($sLPF$):** *For an s-string T of length n, the sLPF array is defined for each index $1 \leq i \leq n$ to maintain the length of the* longest factor *between an s-suffix and a previous s-suffix occurring in T. More formally, $sLPF[i] = max(\{0\} \cup \{k \mid sencode(T[i...n]) ==_k sencode(T[h...n]), 1 \leq h < i\})$.*

Examples of the $cLPF$ and $sLPF$ data structures are displayed in Table 5.5. We highlight that since the individual c-suffixes and s-suffixes vary in the example or more formally, $compl(T[i...n]) \neq sencode(T[i...n]) \; \forall \; i, 1 \leq i \leq n$, it is coincidentally the case that $cLPF == sLPF$ in this particular example.

To avoid redundancies and maintain concision in this work, we first observe that the *compl* and *sencode* are very closely related. The significance of Lemmas 5.3.3, 5.3.4, and 5.3.5 is that a solution to the sLPF problem using the $sencode(i, j)$ function will also provide a solution to the cLPF, pLPF, and traditional LPF problems. Using the same proof in Lemma

Table 5.5: $cLPF$ and $sLPF$ computations, using $T = AwxyBwzw\$$

i	$compl(T[i...n])$	$cLPF[i]$	$sencode(T[i...n])$	$sLPF[i]$
1	A010B335\$	0	A010B432\$	0
2	010B335\$	0	010B432\$	0
3	00B335\$	1	00B332\$	1
4	0B030\$	1	0B032\$	1
5	B000\$	0	B002\$	0
6	000\$	2	002\$	2
7	00\$	2	00\$	2
8	0\$	1	0\$	1
9	\$	0	\$	0

4.3.3 for the pLPF problem, it is evident that the neighboring elements in the $sLPF$ array may be *extended*, which is the key to linear time LPF computations (see [7] and Theorem 4.3.4). Our *compute_all_LPF* algorithm is a linear time solution to the collection of LPF data structure variants ($sLPF$, $cLPF$, $pLPF$, and LPF).

Lemma 5.4.3 *The $sLPF$ for an s-string T of length n is such that $sLPF[i] \geq sLPF[i-1]-1$ with $1 < i \leq n$.*

Proof Using Proposition 5.2.6, Definition 5.2.5, and Definition 5.4.2, the proof is identical to the proof of Lemma 4.3.3. □

Theorem 5.4.4 proves the generality of the *compute_all_LPF* algorithm in Listing 5.3.

Theorem 5.4.4 *Given an n-length s-string T and the appropriate suffix array, the algorithm compute_all_LPF can construct the $sLPF$, $cLPF$, $pLPF$, and LPF array in $O(n)$ time.*

Proof Using Lemma 5.4.3 to implement algorithm *compute_all_LPF* and upgrading the Λ matching functionality with the $sencode(i, j)$ retrieval function in Definition 5.2.7 that clearly introduces $O(1)$ work, the proof of Theorem 4.3.4 identifies that we can construct the longest previous factor for the suffixes in the given suffix array matched by Λ in $O(n)$ time. Since Lemmas 5.3.3, 5.3.4, and 5.3.5 prove that in addition to matching s-suffixes in an sSA, the function $sencode(i, j)$ can be exploited to match c-suffixes in a cSA, p-suffixes in a pSA, and traditional suffixes in a traditional SA, the theorem holds. □

Listing 5.3: Generalized LPF computation

```
1  int[] compute_all_LPF(int before_<[], int before_>[]) {
2    int  LPF[n], LPF_<=0, LPF_>=0, i, j, k
3    for i = 1 to n, step 1 {
4      j = max{0,LPF_<-1}
5      k = max{0,LPF_>-1}
6      LPF_< = Λ(i,before_<[i],j)
7      LPF_> = Λ(i,before_>[i],k)
8      LPF[i] = max{LPF_<,LPF_>}
9    }
10   return LPF
11 }
12 int Λ(int a, int b, int q) {
13   if(b == -1) return 0
14   while(sencode(a,q)==sencode(b,q))
15     q++
16   return q
17 }
```

5.4.2 cLCP and sLCP

Initially, we define the traditional LCP problem in terms of the *compl* and *sencode* encoding schemes for s-strings.

Definition 5.4.5 *compl* ***longest common prefix (cLCP) array:*** *The cLCP array for an s-string T of length n maintains the length of the longest common prefix between neighboring c-suffixes in a compl suffix array (cSA). We define the computation $clcp(\alpha, \beta) = max\{k \mid compl(\alpha) ==_k compl(\beta)\}$. Then, cLCP is defined on each c-suffix i with $1 \leq i \leq n$ such that:*

$$cLCP[i] = \begin{cases} 0, \textbf{ if } i == 1 \\ max\{k \mid clcp(T[cSA[i]...n], T[cSA[i-1]...n])\}, \textbf{ if } 2 \leq i \leq n \end{cases}$$

Definition 5.4.6 *sencode* ***longest common prefix (sLCP) array:*** *The sLCP array for an s-string T of length n maintains the length of the longest common prefix between neighboring s-suffixes in an sencode suffix array (sSA). We define the computation $slcp(\alpha, \beta) = max\{k \mid sencode(\alpha) ==_k sencode(\beta)\}$. Then, sLCP is defined on each s-suffix i with $1 \leq i \leq n$ such that:*

$$sLCP[i] = \begin{cases} 0, \textbf{ if } i == 1 \\ max\{k \mid slcp(T[sSA[i]...n], T[sSA[i-1]...n])\}, \textbf{ if } 2 \leq i \leq n \end{cases}$$

Table 5.6 displays example $cLCP$ and $sLCP$ arrays for an s-string $T = AwxyBwzw\$$. We note that even though the suffixes differ such that $compl(T[i...n]) \neq sencode(T[i...n]) \; \forall \, i, 1 \leq i \leq n$, it is coincidentally the case that $cLCP == sLCP$ for this particular example.

Table 5.6: $cLCP$ and $sLCP$ computations, using $T = AwxyBwzw\$$

i	cSA	$compl(T[cSA[i]...n])$	$cLCP[i]$	sSA	$sencode(T[sSA[i]...n])$	$sLCP[i]$
1	9	\$	0	9	\$	0
2	8	0\$	0	8	0\$	0
3	7	00\$	1	7	00\$	1
4	6	000\$	2	6	002\$	2
5	3	00B335\$	2	3	00B332\$	2
6	2	010B335\$	1	2	010B432\$	1
7	4	0B030\$	1	4	0B032\$	1
8	1	A010B335\$	0	1	A010B432\$	0
9	5	B000\$	0	5	B002\$	0

Similar to both the observation made by Crochemore and Ilie [7] to relate the LPF and LCP arrays and our Proposition 4.4.1 to connect the $pLPF$ and $pLCP$ arrays, we also identify that both the $sLPF$ and $cLPF$ arrays are permutations of the $sLCP$ and $cLCP$ arrays, respectively.

Proposition 5.4.7 *The sLPF array is a permutation of sLCP and the cLPF array is a permutation of cLCP.*

It was observed in Chapter 4 that we can use an algorithm that addresses the pLPF problem to also compute the $pLCP$ and LCP arrays. Likewise, we can employ our *compute_all_LPF* algorithm in the function *compute_all_LCP* in Listing 5.4 to compute the $sLCP$, $cLCP$, $pLCP$, and LCP arrays, since the $sencode(i, j)$ function can retrieve the appropriate types of suffix symbols necessary for matching encodings.

Theorem 5.4.8 *Given an n-length s-string T, the algorithm compute_all_LCP can construct the sLCP, cLCP, pLCP, and LCP array in $O(n)$ time.*

Listing 5.4: Generalized LCP computation

```
1 int[] compute_all_LCP(int before_<[], int after_<[]) {
2   int LCP[n], M[n], i
3   M = compute_all_LPF(before_<, after_<)
4   for i = 1 to n, step 1
5     LCP[i] = M[SA[i]]
6   return LCP
7 }
```

Proof It is clear that the concept of Theorem 4.4.2 for generating a $pLCP$ array from a pLPF algorithm is generalized by our *compute_all_LPF* implementation with the matching component Λ that uses the function $sencode(i, j)$, which may be generalized via Lemmas 5.3.3, 5.3.4, and 5.3.5. Since Theorem 5.4.4 claims $O(n)$ time, the theorem holds. □

5.5 s-Matching

Currently, the s-match problem is addressed with the s-suffix tree [4]. It is possible to search for an m-length s-string pattern P in an n-length s-string T in $O(m \log(|\Sigma| + |\Pi|))$ time using the s-suffix tree. Given the original suffix array search algorithms presented in [6], the suffix arrays for the s-encodings *prev*, *compl*, and *sencode*, and the respective LCP data structures presented in this work, we describe the first suffix array solutions to the s-match problem using Propositions 5.2.4 and 5.2.6.

Traditionally, the key to efficiently pattern matching an m-length pattern P on a suffix array of the string T of length n is to adapt the solutions introduced by Manber and Myers [6]. Recall the background discussion in Section 2.1 of pattern matching via the suffix array and binary search philosophies of [6]. We want to closely mirror the pattern matching approach in [6] requiring $O(m+\log n)$ search time, in addition to the preprocessing required for a suffix array SA and LCP data structure. This solution is simply referred to as the "MM improved algorithm" throughout this section. In the MM improved algorithm, the LCP array is used to further preprocess the LCP between each midpoint M with $1 < M < n$ and the two suffixes L and R such that $M = \lfloor \frac{L+R}{2} \rfloor$ to construct the $O(n)$ data structures $LLCP$ and $RLCP$, which signify the number of symbols that *already* match between a suffix and a midpoint

in order to avoid unnecessary re-matching. Consider $lcp(\alpha, \beta) = max\{k \mid \alpha ==_k \beta\}$. In essence, the idea is to preprocess $LLCP$ values between the left suffixes L and a midpoint M, namely $lcp(T[SA[M]...n], T[SA[L]...n])$, and also the right suffixes R for a midpoint M, namely $lcp(T[SA[M]...n], T[SA[R]...n])$. Thus, additional preprocessing of the LCP array is required to construct the $LLCP$ and $RLCP$ arrays. It was identified in [35, 36, 37] that retrieving the computation $lcp(T[SA[i]...n], T[SA[j]...n])$ for any i and j is achieved by computing the range minimum query $RMQ(i, j)$, which retrieves the minimum value in the range $[i, j]$ of the standard LCP array. The RMQ calculation was proven by [35, 36, 37] to require $O(n)$ preprocessing in $O(n)$ space with $O(1)$ query time, which is ideal since the preprocessing time is of the same order as suffix array construction and the query time is clearly absorbed in the MM improved algorithm. We observe that the RMQ computation is also relevant for the LCP data structures in this work. For discussion purposes of time and space complexity, we acknowledge that the traditional *lcp*, the *plcp* of Definition 2.6.7, the *clcp* of Definition 5.4.5, and the *slcp* of Definition 5.4.6 may be implemented with the RMQ computation.

s-Matching via *prev* and *compl*

Consider an m-length s-string P, our task is to detect an s-match between P and some prefix, say S, of a suffix in the s-string T. The first s-match method displayed in Proposition 5.2.4 states that a pair of s-strings, in our case P and S, will match when $prev(P) == prev(S) \wedge compl(P) == compl(S)$. To implement the s-match of Proposition 5.2.4 using the suffix array pattern matching approaches proposed by Manber and Myers [6], we require a suffix array and corresponding LCP array for the encodings $prev(T)$ and $compl(T)$. In this work, we have shown how to construct the p-suffix array (pSA) and c-suffix array (cSA) to suffix sort the encodings $prev(T)$ and $compl(T)$ respectively. We also show how to compute the $pLCP$ array for $prev(T)$ and the $cLCP$ array for $compl(T)$. Let $prevP = prev(P)$, $complP = compl(P)$, $prevT = prev(T)$, and $complT = compl(T)$ and then, discard P and T. Given $prevP$, $complP$, the pSA of $prevT$, the $pLCP$ of pSA, the cSA of $complT$, and the $cLCP$ of cSA, the discussion proceeds to detecting an s-match of P in T.

In order to efficiently s-match using Proposition 5.2.4, we strive to use the MM improved

search algorithm in [6] between pSA and cSA sequentially. Since the lexicographical ordering between the pSA and cSA arrays are not necessarily the same, i.e. $pSA \neq cSA \; \forall \; T \in \{\Sigma \cup \Pi\}^*\$$ with an arbitrary Γ, we cannot simply exploit the ordering of one suffix array, say pSA, to refine the binary search interval of a match in the lexicographically unrelated cSA, vice versa. To discuss this intricacy in more detail, suppose that $prevP$ matches at suffix i in $prevT$ and $complP$ does not match at the same suffix i in $complT$. Conceptually, the notion that either $complP \prec compl(T[i...i+m-1])$ or $complP \succ compl(T[i...i+m-1])$ does not identify where to continue the binary search in the cSA given the possibility that $prevP$ may occur $O(n)$ times in pSA and moreover, $complP$ may not occur at all in cSA. Thus, we must first find the leftmost and rightmost instances of $prevP$ in pSA, namely the interval $I_{prev} = [L_{prev}, R_{prev}]$ and the range of instances of $complP$ in cSA, namely $I_{compl} = [L_{compl}, R_{compl}]$. Finding the respective L and R values can be computed individually by a straightforward modification to the MM improved search algorithm in [6] using the lcp computation to require $O(m + \log n)$ time. Let $\Delta I_{prev} = R_{prev} - L_{prev}$ and $\Delta I_{compl} = R_{compl} - L_{compl}$. When $\Delta I_{prev} == 0 \vee \Delta I_{compl} == 0$, i.e. an interval from L to R does not exist in both pSA and cSA, the search can be terminated and report that P does not s-match in T. In practice, the cSA may prove to be more restrictive than the pSA since the $compl$ encoding is directly impacted by both the individual symbols of an s-string and the choice of complementary symbol mappings.

After the intervals I_{prev} and I_{compl} are targeted and $\Delta I_{prev} > 0 \wedge \Delta I_{compl} > 0$, we have reached a milestone in that P *might* s-match in T. The new challenge is to validate that an s-match has indeed occurred. So, we *align* the suffix indices between the intervals and determine if the intervals share an index in common, i.e. $\{i == j \mid i = pSA[a], j = cSA[b], L_{prev} \leq a \leq R_{prev}, L_{compl} \leq b \leq R_{compl}\}$. This is achieved simply by radix sorting the suffix indices in the intervals and walking through the elements once to report a common index between both lists. In the worst case, since we spend $O(m + \log n)$ to detect the intervals and clearly the number of suffixes in each interval may be of order $O(n)$ to demand possibly $O(n)$ time to validate a match, the search requires $O(n + m + \log n)$ time, which is quite costly for a search! The time disadvantage is compounded by the need for the data structures $prevP$ and $complP$ of $O(m)$ space plus the fundamental data structures $prev(T)$,

compl(T), pSA, cSA, $pLCP$, and $cLCP$ that require $O(n)$ space, in addition to the overhead required to implement the algorithm. The time complexity and practical space limitations provide the motivation to introduce an improved time and practical space algorithm to s-match via suffix arrays.

s-Matching via *sencode*

The core problem discovered when trying to s-match an m-length s-string P in the s-string T of length n using the pSA and cSA is the added cost to validate a match between the suffix arrays. We are regulated to validating an s-match because of notion that the pSA and cSA arrays are lexicographically unrelated, so, the existence of a possible order $O(n)$ matches of P in the pSA of T will not resolve the fact that the corresponding suffix in cSA may never match and hence, consume $O(n)$ time in the process. Proposition 5.2.6 identifies that we can detect an s-match by simply using the *sencode* encoding scheme. This eliminates the challenges of suffix array pattern matching with the encodings *prev* and *compl* by restricting the s-match problem to a single *sencode* encoding, which makes the s-match via suffix arrays mirror the $O(m + \log n)$ time complexity achieved using a p-suffix array for the p-match problem in [23, 24] analogous to the traditional pattern matching problem using suffix arrays in [6]. Since the s-string is a variant of the p-string and the MM improved algorithm can be extended for p-matching with p-strings using a single suffix array pSA [23, 24], the MM improved algorithm can clearly be extended to accommodate the s-match problem with a single suffix array sSA.

The prerequisites for s-matching with the *sencode* encoding scheme are the s-suffix array (sSA) and the $sLCP$ array for T, which are constructed in this work. In addition, we must obtain the *compl*(T) and *prev*(T) encodings for the *sencode* suffix retrieval function in Definition 5.2.7 to retrieve s-suffix symbols of T, whereas *sencode*(P) is used for the pattern P since we only work with the first s-suffix of P. Theorem 5.5.1 formalizes the claim.

Theorem 5.5.1 *Given an n-length s-string T, the sSA, and the $sLCP$ data structure, it is possible to s-match, c-match, p-match, or traditional match an m-length s-string P in $O(m + \log n)$ time.*

Proof In order to adapt the MM improved algorithm to s-match using sSA and $sLCP$, we must first consider the preprocessing required for each query. Since we are provided with the m-length s-string P, we need to perform the *sencode* on the s-string in order to compare it with s-suffixes of T, which is accomplished by $sencodeP = sencode(P)$. Since the *sencode* function uses the *prev* and *compl* functions that require $O(m)$ time via an auxiliary $O(|\Pi|)$ mapping structure with $|\Pi| \leq m$ enforced by a single mapping, then only $O(m)$ preprocessing time is needed prior to each query. To extend the MM improved algorithm for pattern matching via suffix arrays in $O(m + \log n)$ time to apply to s-suffixes given sSA and $sLCP$, we only require an additional mechanism for pattern matching that retrieves any symbol from any s-suffix of T in constant time, which is achieved by the $sencode(i, j)$ function in Definition 5.2.7. Since $sencode(i, j)$ is defined to retrieve symbols of s-suffixes and can be generalized to retrieve symbols from c-suffixes, p-suffixes and traditional suffixes via Lemmas 5.3.3, 5.3.4, and 5.3.5, respectively, the theorem holds. □

Our improved s-match algorithm is analogous to the MM improved algorithm, claiming the same time complexity. The new s-match algorithm is an advancement also in terms of practical space when compared to the approach discussed previously requiring the use of two suffix arrays and LCP data structures, since s-matching with *sencode* only requires one suffix array sSA and $sLCP$ array. In practice, the s-match algorithm is generalized for the p-match, c-match, and traditional pattern matching problems, which is similar to the other s-match algorithms presented in this work, providing the added incentive to implement a single core solution to address multiple problems with minimal adjustments to the data and alphabets.

5.6 Summary

The s-string theory, which is introduced in [4] using suffix trees as the exclusive data structure, is advanced in this work to include the suffix array data structure. We provide an information theoretic approach to construct the suffix arrays for the s-encodings *compl* and *sencode* in linear time on average. Then, the traditional LPF problem [7] is generalized for the s-encodings of an s-string. It is shown how to further manipulate the LPF algorithm

to also construct the LCP arrays for the s-encodings in linear time in the worst case. We culminate the chapter by using our data structures to offer the first s-match solution via suffix arrays. It is identified how to extend traditional pattern matching via suffix arrays in [6] to s-match with the same running time $O(m + \log n)$ per query, where n is the length of the s-string text T and m is the length of the located pattern P. A significant observation exploited throughout this work is the capability to generalize the s-string encoding *sencode* to permit all algorithms to apply also to p-strings and traditional strings.

Chapter 6

Conclusion

6.1 Summary

Our information theoretic approach to suffix sorting p-string suffixes and s-string suffixes in linear time on average is a creative approach to suffix array construction for sophisticated string encodings. The information theoretic approach to suffix sorting the suffixes of a p-string and s-string allows the sorting of numeric codes to, in turn, sort the suffixes, resembling the work of [8]. This approach is transformative for the suffixes of p-strings and s-strings because of the *dynamic* nature of the encoded suffixes. Numeric codes represent a trivial and tangible lexicographical ordering for an *m-block* prefix of a dynamically changing suffix, which would otherwise require an involved mechanism to maintain the proper lexicographical ordering.

Before this work, the LPF problem was limited to the confines of traditional strings. By redefining the traditional LPF problem in terms of p-strings and s-strings, we liberate the capabilities of the LPF data structure to apply to more generalized strings. Subsequently, we introduce an application for our previously constructed suffix arrays. Similar to [7], we show a linear time solution for our respective LPF data structures. We are the first to show that it is indeed possible to use the LPF algorithm to construct the respective LCP data structure. Such is a novel application of the LPF algorithm to reuse functionality and help construct yet another fundamental string data structure.

Prior to this thesis, the s-match problem was exclusively solved with s-suffix trees. Our

work on suffix arrays and LCP arrays for s-strings provides the necessary data structures to propose the first suffix array solution to the s-match problem, which claims the same time complexity as the mirroring pattern matching via suffix array approach for traditional strings [6].

A significant contribution of this thesis is the clear incentive to approach string applications from the standpoint of s-strings or p-strings. We consistently show how minor adjustments to the data or alphabets can adjust the behavior of an algorithm to apply to traditional strings, p-strings, and s-strings, highlighting the generality of our algorithms, data structures, and overall contributions to the string analysis community.

6.2 Future Research

We contribute to the advancement of p-string and s-string theory in this work. Our studies have introduced future research areas to continue the advancement of the p-string and s-string.

Recall that p-strings and s-strings are productions from a given constant symbol alphabet Σ and a given parameter alphabet Π, such that $\Sigma \cap \Pi = \emptyset$. The s-string introduces the notion of complementary parameter symbols in the alphabet Γ. Suppose that the alphabets are not given, the question is: how can we best choose the constants $\sigma \in \Sigma$ and the parameters $\pi \in \Pi$ with limited knowledge of the intended application? How can we further process the individual $\pi \in \Pi$ to identify which parameters are complementary symbols in Γ? It is intriguing to consider a symbol, say α, where the initial case is $\alpha \in (\Sigma \cup \Pi)$ and the further processing of a module M can "classify" the symbol as either $\alpha \in \Sigma$ or $\alpha \in \Pi$. Identifying the technique used by module M, which may possibly be influenced by studies in the areas of pattern recognition and natural language processing, is a pivotal research question.

Our work introduces new techniques to suffix sort the p-string suffix encodings and the s-string suffix encodings in linear time on average. The notion that linear time suffix sorting is achieved for traditional strings via induced sorting (see [8, 12, 14, 15]) introduces a new goal for p-strings and s-strings. To improve the worst case suffix sorting for p-strings and s-strings, it will be required to identify the intricate relationship between the *dynamically*

encoded suffixes. More specifically, the challenge is to determine the technique in which sorting a methodically chosen partition of *dynamically* encoded suffixes correctly implies the sorting of the encoded suffixes of the remaining partitions. Improvements to the worst case suffix sorting of p-strings and s-strings will further encourage the study of suffix array applications.

In this work, we present the LPF problem for p-strings and s-strings. Another area of research is to further use the proposed LPF data structures to study duplication and compression in terms of p-strings and s-strings. Perhaps, the generalization potential of the parameterized and structural string will catapult p-string and s-string solutions to become the preferred way to implement string algorithms.

References

[1] Baker, B. Parameterized pattern matching: Algorithms and applications. J. Comput. Syst. Sci. 52(1), 28-42 (1996)

[2] Baker, B. Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM J. Comput. 26(5), 1343-1362 (1997)

[3] Baker, B. A theory of parameterized pattern matching: Algorithms and applications. In Proceedings of STOC'93, pp. 71-80, ACM, New York (1993)

[4] Shibuya, T. Generalization of a suffix tree for RNA structural pattern matching. Algorithmica. 39(1), 1-19 (2004)

[5] Shibuya, T. Generalization of a suffix tree for RNA structural pattern matching. In Proceedings of SWAT'00, pp. 393-406, Springer, London (2000)

[6] Manber, U., Myers, G. Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22, 935-948 (1993)

[7] Crochemore, M., Ilie, L. Computing longest previous factor in linear time and applications. Inf. Process. Lett. 106(2), 75-80 (2008)

[8] Adjeroh, D., Nan, F. Suffix sorting via Shannon-Fano-Elias codes. Algorithms. 3(2), 145-167 (2010)

[9] Baker, B. Finding clones with dup: Analysis of an experiment. IEEE Trans. Software Eng., 33(9), 608-621 (2007)

[10] Gusfield, D. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge, UK (1997)

[11] Smyth, W. Computing Patterns in Strings. Pearson, New York (2003)

[12] Adjeroh, D., Bell, T., Mukherjee, A. The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching. Springer, New York (2008)

[13] Ukkonen, E. On-line construction of suffix trees. Algorithmica. 14, 249-260 (1995)

[14] Kärkkäinen, J., Sanders, P., Burkhardt, S. Linear work suffix array construction. J. ACM. 53, 918-936 (2006)

[15] Manzini, G., Ferragina, P. Engineering a lightweight suffix array construction algorithm. Algorithmca. 40, 33-50 (2004)

[16] Kosaraju, S. Faster algorithms for the construction of parameterized suffix trees. In Proceedings of FOCS '95, pp. 631-637, ACM, Washington, DC, (1995)

[17] Cole, R., Hariharan, R. Faster suffix tree construction with missing suffix links. SIAM J. Comput. 33(1), 26-42 (2003)

[18] Lee, T., Na, J., Park, K. On-line construction of parameterized suffix trees for large alphabets. Inf. Process. Lett. 111(5), 201-207 (2011)

[19] Amir, A., Farach, M., Muthukrishnan, S. Alphabet dependence in parameterized matching. Inf. Process. Lett. 49, 111-115 (1994)

[20] Baker, B. Parameterized pattern matching by Boyer-Moore-type algorithms. In Proceedings of SODA'95, pp. 541-550, ACM, Philadelphia, PA (1995)

[21] Idury, R., Schäffer, A. Multiple matching of parameterized patterns. Theor. Comput. Sci. 154, 203-224 (1996)

[22] Aho, A.V., Corasick, M.J. Efficient string matching: An aid to bibliographic search, Commun. ACM 18, 333-340 (1975)

[23] Deguchi, S., Higashijima, F., et al. Parameterized suffix arrays for binary strings. In Proceedings of PSC'08, pp. 84-94, Czech Republic (2008)

[24] Tomohiro, I., Deguchi, S., et al. Lightweight parameterized suffix array construction. In Proceedings of IWOCA'09. LNCS, vol. 5874, pp. 312-323. Springer, Heidelberg (2009)

[25] Ziv, J., Lempel, A. A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory. 23(3), 337-343 (1977)

[26] Crochemore, M., Ilie, L., Smyth, W. A simple algorithm for computing the Lempel Ziv factorization. In Proceedings of DCC'08, pp. 482-488 (2008)

[27] Main, M. Detecting leftmost maximal periodicities. Discrete Appl. Math. 25(1-2), 145-153 (1989)

[28] Zeidman, B. Software v. software. IEEE Spectr. 47, 32-53 (Oct. 2010)

[29] Karp, R., Rabin, M. Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31. 249-260 (1987)

[30] Moffat, A., Neal, R., Witten, I. Arithmetic coding revisited. ACM Trans. Inf. Syst. 16, 256-294 (1995)

[31] Cover, T., Thomas, J. Elements of Information Theory. Wiley (1991)

[32] Karlin, S., Ghandour, G., et al. New approaches for computer analysis of nucleic acid sequences. PNAS. 80(18), 5660-5664 (1983)

[33] Devroye, L., Szpankowski, W., Rais, B. A note on the height of suffix trees. SIAM J. Comput. 21, 48-53 (1992)

[34] Kasai, T., Lee, G., et al. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Proceedings of CPM'01. LNCS, vol. 2089, pp. 181-192 (2001)

[35] Bender M., Farach M. The lca problem revisited. In Proceedings of LATIN'00, pp. 88-94, Springer, London (2000)

[36] Sadakane, K. Succinct representations of lcp information and improvements in the compressed suffix arrays. In Proceedings of SODA'02, pp. 225-232, ACM-SIAM, Philadelphia, PA (2002)

[37] Kim, D., Sim, J., Park, H., Park, K. Linear-time construction of suffix arrays. In Proceedings of CPM'03, pp. 186-199, Springer, Heidelberg (2003)