2015

Parameterized Strings: Algorithms and Applications

Richard Beal

Parameterized Strings: Algorithms and Applications

by

Richard Beal

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

Doctor of Philosophy
in
Computer Science

Elaine Eschen, Ph.D.
Arun Ross, Ph.D.
Bojan Cukic, Ph.D.
William Smyth, Ph.D.
Donald Adjeroh, Ph.D., Chair

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2015

Keywords: parameterized match, structural match, equivalence parameterized match, indeterminate parameterized match, compressed parameterized match, parameterized compression, parameterized suffix array, equivalence parameterized suffix array, parameterized longest common prefix array, equivalence parameterized longest common prefix array, parameterized longest previous factor, parameterized border, structural border, compact parameterized prefix array, compact indeterminate parameterized prefix array, Forward Stem Matrix, RNA secondary structures, hairpins, pseudoknots

Abstract

Parameterized Strings:
Algorithms and Applications

by

Richard Beal

The parameterized string (p-string), a generalization of the traditional string, is composed of constant and parameter symbols. A parameterized match (p-match) exists between two p-strings if the constants match exactly and there exists a bijection between the parameter symbols. Historically, p-strings have been employed in source code cloning, plagiarism detection, and structural similarity between biological sequences. By handling the intricacies of the parameterized suffix, we can efficiently address complex applications with data structures also reusable in traditional matching scenarios. In this dissertation, we extend data structures for p-strings (and variants) to address sophisticated string computations.

We introduce a taxonomy of classes for longest factor problems. Using this taxonomy, we show an interesting connection between the parameterized longest previous factor ($pLPF$) and familiar data structures in string theory, including the border array, prefix array, longest common prefix array, and analogous p-string data structures. Exploiting this connection, we construct a multitude of data structures using the same general $pLPF$ framework.

Before this dissertation, the p-match was defined predominately by the matching between uncompressed p-strings. Here, we introduce the compressed parameterized pattern match to find all p-matches between a pattern and a text, using only the pattern and a compressed form of the text. We present parameterized compression (p-compression) as a new way to losslessly compress data to support p-matching. Experimentally, it is shown that p-compression is competitive with standard compression schemes. Using p-compression, we address the compressed p-match independent of the underlying compression routine.

Currently, p-string theory lacks the capability to support indeterminate symbols, a staple essential for applications involving inexact matching such as in music analysis. In this work, we propose and efficiently address two new types of p-matching with indeterminate symbols. (1) We introduce the indeterminate parameterized match (ip-match) to permit matching with indeterminate holes in a p-string. We support the ip-match by introducing data structures that extend the prefix array. (2) From a different perspective, the equivalence parameterized match (e-match) evolves the p-match to consider intra-alphabet symbol classes as equivalence classes. We propose a method to perform the e-match using the p-string suffix array framework, i.e. the parameterized suffix array (pSA) and parameterized longest common prefix array ($pLCP$). Historically, direct constructions of the pSA and $pLCP$ have suffered from quadratic time bounds in the worst-case. Here, we introduce new p-string theory to efficiently construct the $pSA/pLCP$ and break the theoretical worst-case time barrier.

Biological applications have become a classical use of p-string theory. Here, we introduce the structural border array to provide a lightweight solution to the biologically-oriented

variant of the p-match, i.e. the structural match (s-match) on structural strings (s-strings). Following the s-match, we show how to use s-string suffix structures to support various pattern matching problems involving RNA secondary structures. Finally, we propose/construct the forward stem matrix (FSM), a data structure to access RNA stem structures, and we apply the FSM to the detection of hairpins and pseudoknots in an RNA sequence.

This dissertation advances the state-of-the-art in p-string theory by developing data structures for p-strings/s-strings and using p-string/s-string theory in new and old contexts to address various applications. Due to the flexibility of the p-string/s-string, the data structures and algorithms in this work are also applicable to the myriad of problems in the string community that involve traditional strings.

Acknowledgments

My committee chair and advisor, Dr. Donald Adjeroh (Dr. A.), introduced me to the world of strings and always provided the guidance to see an idea through to the finish line. For the research and teaching opportunities, I am grateful. I would like to deeply thank my committee members – Dr. Elaine Eschen, Dr. Arun Ross, Dr. Bojan Cukic, and Dr. William Smyth – for their comments, suggestions, and support of my work. The knowledge acquired during your lectures was fundamental to the results achieved in this research. Together, we have advanced the state-of-the-art for strings, parameterized strings (p-strings), and structural strings (s-strings). Since this work yielded only a phD, then the next logical step must be to earn the shD! I also thank my labmates, Marco Piccirilli and Syed Ashiqur Rahman, for their support over the years. I now pass the torch to you – make Dr. A. proud!

I am infinitely appreciative of my father, Richard Sr., and mother, Pamela. You 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, situational comedies, and Matlock. I would like to deeply thank my cousin James V. Matthews for his continued confidence in my abilities.

The people 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.

This acknowledgments section would not be complete without expressing my appreciation to (President) David and (First Lady) Amy Loomis, and my wolfpack at the WVU Swing Dance Club. You were always there to remind me of the true rhythm in life; thank you. Also, I thank the WVU Community Music Program and my vocal instructors, Alaina Tetrick and Jennifer Brown, for providing me with music serenity during my PhD studies. Special thanks go out to the legends – Neil Diamond, Elton John, Eddie Money, and Billy Joel – for writing those core songs that I always reference for answers. And to Liberace, my creativity is inspired by you. Your performances and tailcoats always make me smile!

My PhD research was supported in part by the WVU Provost Fellowship. Also, this work was partly supported by a grant from the National Science Foundation (#IIS-1236983), and from the National Historical Publications & Records Commission (in collaboration with the DCAPE team at the University of North Carolina, Chapel Hill).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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. By representing data with a traditional string, we can analyze data for equivalence 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 parameterized string and variations, such as the structural string, provide the foundation for natural methods to more involved and complex string analyses.

1.1 Motivation and the Problem

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. Traditional pattern matching will not detect this relationship between the source files. Parameterized matching (p-matching) 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 p-match 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 efficiently, and (3) further advance the theory of related sophisticated strings, such as the structural string (s-string) used in the structural match (s-match) problem for analyzing RNA sequences by their secondary structure.

1.2 Main Contributions

In this dissertation, we advance the state-of-the-art in parameterized string (p-string) theory and variants, i.e. the structural string (s-string). We develop new data structures, extend popular data structures, identify beautiful relationships between data structures, propose a new context for p-string matching, and apply p-string/s-string theory in both new and old contexts to address various applications. A common theme in this work is advocating for more general data structures and acknowledging the power of string encodings to generalize solutions, which promotes flexibility and reusability of solutions. Though each chapter specifically highlights the contributions made, here, we note the core contributions of the research.

- Chapter 3: Variations of the Parameterized Longest Previous Factor
  - We define a taxonomy for longest factor problems.
  - A connection is made between the parameterized longest previous factor ($pLPF$) construction and a subset of the taxonomy. We exploit the linear time $pLPF$ construction to generate the following popular p-string data structures: the parameterized longest common prefix ($pLCP$) and the parameterized-border (p-$border$)

array. By altering the alphabets, we show how the same $pLPF$ construction can yield the longest common prefix (LCP), the permuted-longest common prefix (*permuted-LCP*), the *border* array, and the prefix array.

  - We propose other data structures also solved by the same $pLPF$ framework.

- Chapter 4: The Structural Border Array

  - Motivated to structural match without suffix structures, we propose the structural border array (*s-border*) as an extension of the traditional *border* array.
  - The traditional *border* properties are shown to also hold for the *s-border*, which leads to a linear construction of the *s-border*. By altering the alphabet, we also construct the *p-border* and *border* arrays in linear time with the same algorithm.

- Chapter 5: Compressed Parameterized Pattern Matching

  - We introduce compressed parameterized pattern matching (compressed p-matching) as the act of finding all occurrences of a pattern P in a text T compressed via p-compression, a proposed way to losslessly compress data via a p-string encoding.
  - An algorithm is provided for compressed p-matching in terms of any compression scheme where a partial symbol decompression function can be defined.

- Chapter 6: Parameterized Strings with Indeterminate Symbols

  - We propose the indeterminate parameterized match (ip-match) and the equivalence parameterized match (e-match) as p-match variants with indeterminate symbols.
  - The ip-match is addressed via extensions to the compressed prefix array (cPA) framework. Specifically, we propose and construct the compact parameterized prefix array ($cpPA$) and compact indeterminate parameterized prefix array ($cipPA$).
  - The e-match is solved via the parameterized suffix array (pSA) and parameterized longest common prefix ($pLCP$) framework. New p-string theory is introduced en route to new $pSA/pLCP$ constructions that break the worst-case time barrier.

- Chapter 7: Matching RNA Secondary Structures
  - We use the structural suffix array (sSA) to find RNA secondary structures in a text, based on a complex user query.
  - We propose the Forward Stem Matrix (FSM), which offers efficient access to potential stems in RNA secondary structures, and construct the FSM in time linear to its size via newly proposed auxiliary data structures.
  - The FSM is applied to hairpin and pseudoknot detection in an RNA sequence.

1.3 Organization

The dissertation is organized as follows. Chapter 2 presents the notations, theory, data structures, etc. used in future chapters. In Chapter 3, we generalize and extend the construction of the parameterized longest previous factor [19] from the Master's thesis [16] to also construct a family of popular data structures. Chapter 4 continues the data structure conversation as we construct a generalization of the parameterized *border* (p-border) array [56] for structural strings [86]. Subsequently, Chapter 5 proposes a method for parameterized pattern matching on compressed p-strings, using the p-border array. The p-match capabilities are extended to include indeterminate symbols in Chapter 6. In Chapter 7, we propose data structures to assist with problems involving hairpins, pseudoknots, and complex RNA secondary structures. Lastly, Chapter 8 summarizes the results and provides final remarks.

1.4 Publications/Presentations

Below, we list the current publications/presentations resulting from this work.

- Beal, R., Adjeroh, D.: Variations of the parameterized longest previous factor. Journal of Discrete Algorithms, 16, 129-150 (2012)
- Beal, R., Adjeroh, D.: Parameterized longest previous factor. Theoretical Computer Science, 437, 21-34 (2012)

- Beal, R., Adjeroh, D.: p-Suffix Sorting as Arithmetic Coding. Journal of Discrete Algorithms, 16, 151-169 (2012)

- Beal, R., Adjeroh, D.: Border array for structural strings. In: Arumugam, S., Smyth, W.F. (eds.), International Workshop on Combinatorial Algorithms (IWOCA) 2012. LNCS, vol. 7643, pp. 189-205. Springer, Heidelberg (2012)

- Beal, R., Adjeroh, D.: The structural border array. Journal of Discrete Algorithms, 23, 98-112 (2013)

- Beal, R., Adjeroh, D.: Compressed parameterized pattern matching. IEEE Data Compression Conference (DCC). pp. 461-470. IEEE. (2013)

- Beal, R., Adjeroh, D., Abbasi, A.: The forward stem matrix: An efficient data structure for finding hairpins in RNA secondary structures. ACM Conference on Bioinformatics, Computational Biology, and Biomedical Informatics (ACM-BCB). pp. 576-585. ACM. (2013)

- Beal, R., Adjeroh, D.: Suffix arrays for structural strings. Poster presented at the International Workshop on Combinatorial Algorithms (IWOCA), Duluth, MN (15-17 October 2014)

```java
public class Program {
 private static char[] alphabet
            = {'A','C','G','T'};
 private int num;

 public Program(int num)
                throws Exception
 {
   this.num = num;
   if(this.num <= 0) //invalid
     throw new Exception("!!!");
   this.dnaPermutations("");
 }

 public void dnaPermutations(
                        String str){
   if(str.length() != this.num)
   {
     for(char q : this.alphabet)
       this.dnaPermutations(str+q);
   }
   else
     System.out.println(str);
 }

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

```java
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

Preliminaries

A string on an alphabet of symbols, say $\mathcal{A}$, is a production $T = T[1]T[2]...T[n]$ from $\mathcal{A}^n$ where $n = |T|$ denotes the length of T. Later, we will introduce specific alphabets. In this work, we assume the use of indexed alphabets, where every symbol is represented by a distinct integer modeling the lexicographical ordering of the symbols. For completeness, we append strings with a terminal symbol from the set $\{\$_1, \$_2, ...\}$ or more generally \$. Note that $\$ \notin \mathcal{A}$ and \$ lexicographically precedes every symbol in $\mathcal{A}$. To simplify discussions and for brevity, we may omit the terminal symbol and subscript.

We will use the following string notations: $T[i]$ refers to the ith symbol of string T, $T[i...j]$ refers to the substring $T[i]T[i+1]...T[j]$, and $T[i...n]$ refers to the ith suffix of T: $T[i]T[i+1]...T[n]$. The m-length prefix of a suffix is the substring with the first m symbols of the suffix. A proper prefix and proper suffix of T is respectively any prefix and suffix except T. Let $S \circ T$, or simply ST, denote the concatenation of the strings S and T. This notation is suppressed for concision when the context is clear. To compute the reverse of a substring, we use the notation $T[i...j]^R = T[j]T[j-1]...T[i]$ for $1 \leq i \leq j \leq n$. Alternatively, we can reverse a string without considering the terminal symbol with the following function: $\texttt{reverse}(T[1...n-1]\$) = T[1...n-1]^R\$$. Let $\texttt{replace}(T, x, y)$ replace all occurrences in T of the symbol x with y. For notation purposes, we may specify the string to which a data structure refers. For instance, D_Q denotes that the data structure D was constructed using the string Q. This notation is omitted when the context is clear.

At the symbol level (say symbols s_1, s_2, and s_3), the exact symbol matching operator

$(=)$ is a binary operator that accepts two symbols and returns true when the symbols match and otherwise, returns false; this operator is reflexive ($s_1 = s_1$), symmetric (if $s_1 = s_2$, then $s_2 = s_1$), and transitive (if $s_1 = s_2$ and $s_2 = s_3$, then $s_1 = s_3$). We generalize this operator to the string level. For strings S_1 and S_2, $S_1 = S_2$ returns true if $|S_1| = |S_2|$ (same length) and $S_1[1] = S_2[1] \wedge S_1[2] = S_2[2] \wedge ... \wedge S_1[|S_1|] = S_2[|S_2|]$, and returns false otherwise. Similarly, we define the inexact symbol matching operator $(\approx)$ as a binary operator that returns true when two symbols s_1 and s_2, not necessarily $s_1 = s_2$, are considered equivalent; this operator is also reflexive, symmetric, and transitive, and also applies to strings. We define the $\neq$ operator to return the opposite of $=$ and the $\not\approx$ operator to return the opposite of $\approx$.

We now define operators on the lexicographical ordering of symbols and strings. The operation $s_1 \prec s_2$ (or $s_1 < s_2$) denotes that symbol s_1 lexicographically precedes symbol s_2. Further, $s_1 \succ s_2$ (or $s_1 > s_2$) denotes that symbol s_1 lexicographically succeeds symbol s_2. The $\prec$, $<$, $\succ$, and $>$ operators also apply to strings.

We extend the aforementioned operators on strings to only consider the length $k \geq 0$ prefix of the string operands: $=_k$, $\neq_k$, $\prec_k$, and $\succ_k$. For example, $S_1 \prec_k S_2$ returns true if $S_1[1...k] \prec S_2[1...k]$ and returns false otherwise. In the case that a string operand does not have k symbols, the \$ is padded to the end of the string.

For any array, say D, we can also use string notations such as $|D|$ is the size of D and $D[i...j]$ is a subarray of D. We denote D_{max} as the maximum value in the array and denote D_μ as the mean of the array. We use the following notation to recursively access array elements: $D^2[i] = D[D[i]]$, $D^3[i] = D[D[D[i]]]$, and $D^v[i] = D[D^{v-1}[i]]$.

In this work, we assume that a doubly linked list has the following node definition and basic operations. For readability, our doubly linked list will store the data suf, which are suffix indices.

```
// **** Doubly Linked List ****
// Node type: { int suf, Node* previous, Node* next }
// Operations:
// -- void init( ) : initialization routine
// -- Node* insert(int i) : inserts Node with data i; returns pointer to Node
// -- void delete(Node* ptr) : removes the Node pointed to by ptr
// -- void clear( ) : removes all Nodes in list
```

We refer to a string from a single constant alphabet Σ as a traditional, standard, regular, and exact string. In the following, we present theory for strings with other alphabets, i.e. parameterized strings and structural strings.

2.1 Parameterized Strings

Parameterized strings (p-strings) are generated from the finite alphabets sets Σ and Π. Alphabet Σ denotes the set of constant symbols and Π represents the set of parameter symbols. These alphabets are disjoint ($\Sigma \cap \Pi = \emptyset$).

Definition 2.1.1 ([11]) Parameterized string (p-string): *A p-string is a production T of length n from $(\Sigma \cup \Pi)^*\$$.*

For practical purposes, we can assume that $|\Sigma| + |\Pi| \leq n$ for large n. Consider the alphabets $\Sigma = \{A, B, C\}$ and $\Pi = \{a, b, c, v, w, x, y, z\}$. Example p-strings include $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$. For efficient access to the symbol types of the p-string, we define the α encoding.

Definition 2.1.2 Alphabet encoding (α): *Given an n-length p-string T, we define $\alpha(T)[i] = SIGMA$ if $T[i] \in (\Sigma \cup \{\$\})$ and $\alpha(T)[i] = PI$ if $T[i] \in \Pi$, for $1 \leq i \leq n$ and readable constants $SIGMA$ and PI.*

Two equal-length p-strings are said to be parameterized matches (p-matches) iff (1) the constant symbols match exactly and (2) there exists a bijection between the parameter symbols.

Definition 2.1.3 ([11, 40]) Parameterized matching (p-match): *A p-match exists between pair of p-strings S and T 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 every* $1 \leq j < i$

(b) $S[i] = S[i-q]$ iff $T[i] = T[i-q]$ *for every* $1 \leq q < i$

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. In [11], the `prev` encoding was defined to more directly identify a p-match.

Definition 2.1.4 ([11, 53, 40]) Previous (`prev`) encoding: *Given the n-length* T *and nonnegative integers* $\mathbb{Z}$*, the function* `prev` *:* $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ *(1) encodes constants/terminals, say* c*, with the same symbol* c *and (2) encodes parameters, say* π*, to the distance to the previous* π *in* T*. We formally define* $\texttt{prev}(T)$ *below for each* i*,* $1 \leq i \leq n$*.*

$$\texttt{prev}(T)[i] = \begin{cases} T[i], \text{ if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \text{ if } T[i] \in \Pi \wedge T[i] \neq T[j] \text{ for every } 1 \leq j < i \\ i - \max\{j | T[i] = T[j], 1 \leq j < i\}, \text{ otherwise} \end{cases}$$

When the Σ and Π alphabets are unclear from the context, we overload `prev` as follows: $\texttt{prev}(T, \Sigma, \Pi)$. In our working example, we have $\texttt{prev}(S)$=$A0B0AB54\$$, $\texttt{prev}(T)$=$A0B0AB54\$$, and $\texttt{prev}(U)$=$A0B2AB31\$$. To construct $\texttt{prev}(T)$ when T is from a large, non-indexed alphabet, we can use a balanced binary tree to keep a record of the previous occurrence of each $\pi \in \Pi$ and thus, $O(n \log(|\Pi|))$ time is needed when $|\Pi| \geq 1$.

Lemma 2.1.5 *Given an n-length p-string* T *from* Σ *and* Π*, the encoding* $\texttt{prev}(T)$ *can be constructed in* $O(\min\{n, n \log(|\Pi|)\})$ *time with* $O(|\Pi|)$ *extra space.*

In our case with an indexed alphabet, we can construct $\texttt{prev}(T)$ in $O(n)$ time with an auxiliary $O(|\Pi|)$ mapping structure.

Lemma 2.1.6 *Given an n-length p-string* T *from indexed alphabets* Σ *and* Π*, the encoding* $\texttt{prev}(T)$ *can be constructed in* $O(n)$ *time with* $O(|\Pi|)$ *extra space.*

Note that during the $\texttt{prev}(T)$ construction, we can also construct $\alpha(T)$ from Definition 2.1.2. While the `prev` encodes each parameter π to the distance to the previous occurrence of π in the string, the `forw` considers the forward occurrence of π.

Definition 2.1.7 ([21, 16]) Forward (`forw`) encoding: *We define the function* `forw` *for the p-string* T *of length* n *as* $\texttt{forw}(T) = \texttt{reverse}(\texttt{replace}(\texttt{prev}(\texttt{reverse}(T)), 0, n))$.

Our definition of `forw` generates output mirroring the `fw` encoding used by Deguchi et al. [53, 40]. Our example strings $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$, where $n = 9$, yield $\texttt{forw}(S) = A5B4AB99\$$, $\texttt{forw}(T) = A5B4AB99\$$, and $\texttt{forw}(U) = A2B3AB19\$$.

The relationship between p-strings and the lexicographical ordering of the `prev` encoding is fundamental to the p-match problem.

Definition 2.1.8 `prev` lexicographical ordering: *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 *are lexicographically sorted in their respective alphabets.*

In practice, p-string encodings can be realized using integer arrays with the terminal \$ represented by the minimum integer, followed by the parameter distances z, and lastly followed by constants σ. Now, we can easily deal with the lexicographical ordering of symbols and with a simple range check, we can determine both p-string and `prev` alphabet membership questions $x \in X$ via $\texttt{in}(x, X)$, where $X \in \{\Sigma, \Pi, \mathbb{Z}, \$\}$, in $O(1)$ time, an alternative to α (Definition 2.1.2). In a program, we can "decompose" these symbol integers into a class (i.e. z, σ, \$) and a value (i.e. the index of the symbol in the alphabet or the actual parameter distance). We denote decomposition using the notation C_g for each symbol, where C represents the class and g determines the value. For instance, consider the indexed alphabet $\mathcal{A} = \{A \to 1, x \to 2, Y \to 3\}$. Let $\Sigma' = \{A, Y\}$ and $\Pi' = \{x\}$. If $T = xYx$, then $\texttt{prev}(T, \Sigma', \Pi') = 0Y2$ and alternatively, we say $G = \texttt{prev}(T, \Sigma', \Pi') = z_0\sigma_3 z_2$, because z_0 and z_2 represent the respective distances 0 and 2 and further, σ_3 represents the symbol $\mathcal{A}[3]$. The expression $a_b = G[3]$ yields the assignments $a = z$ and $b = 2$. The boolean expressions $a = \sigma$, $a = z$, and $a = \$$ can be used to determine the class of the encoded symbol $G[3]$. Further, we can use b as a traditional integer to retrieve the appropriate value.

Using the `prev` encoding, we can determine a p-match and the lexicographical ordering of p-strings.

Theorem 2.1.9 ([11]) *Two p-strings S and T p-match when* $\texttt{prev}(S) = \texttt{prev}(T)$. *Also,* $S \prec T$ *when* $\texttt{prev}(S) \prec \texttt{prev}(T)$ *and* $S \succ T$ *when* $\texttt{prev}(S) \succ \texttt{prev}(T)$.

In our working example, we have a p-match between S and T since $\texttt{prev}(S) = \texttt{prev}(T) = A0B0AB54\$$. Also, $U \succ S$ and $U \succ T$ since $\texttt{prev}(U) = A0B2AB31\$ \succ \texttt{prev}(S) = \texttt{prev}(T) = A0B0AB54\$$.

2.1.1 Data Structures

When extending traditional pattern matching data structures to the p-match, we deal with a parameterized suffix (p-suffix) $\texttt{prev}(T[i...n])$, i.e. the suffix $T[i...n]$ under `prev`. The major difficulty in handling p-match problems is the `prev` encoding, which behaves on suffixes in a way that voids traditional suffix properties. Due to the dynamic nature of the p-suffixes, p-match based data structures typically cannot be constructed using approaches for traditional strings. Unlike traditional suffixes, a smaller p-suffix is not necessarily a suffix of a larger p-suffix, which is formalized below.

Lemma 2.1.10 *Given a p-string T of length n, the suffixes of* $\texttt{prev}(T)$ *are not necessarily the p-suffixes of T. More formally, if $\pi \in \Pi$ occurs more than once in T, then $\exists i$ such that* $\texttt{prev}(T[i...n]) \neq \texttt{prev}(T)[i...n], 1 \leq i \leq n$.

Proof Suppose the *only* parameter symbol to occur in the p-string T is $\pi \in \Pi$, which exists *only* at positions a and b with $a < b$. Suppose that indeed $\texttt{prev}(T[a...n]) = \texttt{prev}(T)[a...n]$ and $\texttt{prev}(T[b...n]) = \texttt{prev}(T)[b...n]$. By Definition 2.1.4, the first occurrence of π at position a will be `prev` encoded by 0 and the π at position b will be `prev` encoded by $b-a$. So, in the case of suffix a, $\texttt{prev}(T[a...n]) = \texttt{prev}(T)[a...n]$. At suffix b, the encoding of π at position b in T will *change* to 0 in $\texttt{prev}(T[b...n])$ by Definition 2.1.4 whereas $\texttt{prev}(T)[b...n]$ will retain the *old* encoding of $b - a$ since π *still* occurs in $\texttt{prev}(T)$ at position a. The π at position b forces $\texttt{prev}(T[b...n]) \neq \texttt{prev}(T)[b...n]$, a contradiction. □

For example, $T = AcbcAc$ yields the p-suffixes $\mathcal{X} = \{A002A2, 002A2, 00A2, 0A2, A0, 0\}$. Notice that $\mathcal{X}[5] = T[5] \circ \mathcal{X}[6]$, but $\mathcal{X}[4] \neq T[4] \circ \mathcal{X}[5]$ and $\mathcal{X}[4] \neq \mathcal{X}[4][1] \circ \mathcal{X}[5]$. As noted in [18], due to this dynamic nature, the n-length p-string T essentially compresses these

n dynamic p-suffixes with $O(n^2)$ total symbols, giving merit to sub-quadratic solutions to problems with p-strings.

A powerful pattern matching data structure is the suffix tree (ST). The ST on the n-length W is a compact trie (with $O(n)$ nodes) that represents all of the suffixes $W[i...n]$, for $1 \leq i \leq n$. Suffixes with common prefixes share nodes in the tree until the suffixes differentiate and ultimately, each suffix $W[i...n]$ will have its own leaf node to denote i. Here, we will work more closely with alternative lightweight representations of the ST. The suffix array (SA) and longest common prefix (LCP) array represent the ST data in an array format: the SA stores a list of the indices of the sorted suffixes and the LCP stores the length of the longest prefix common between neighboring suffixes in the SA. In the following, we define the SA and LCP data structures for p-strings.

Below, we define the parameterized suffix array (pSA), analogous to the SA, as a representation of the lexicographical ordering of the p-suffixes of p-string T.

Definition 2.1.11 ([53, 40]) Parameterized suffix array (pSA): *The pSA for a p-string T of length n stores the lexicographical ordering of the indices i representing individual p-suffixes* $\texttt{prev}(T[i...n])$ *with* $1 \leq i \leq n$*, such that* $\texttt{prev}(T[pSA[q]...n]) \prec \texttt{prev}(T[pSA[q+1]...n]) \forall q, 1 \leq q < n$*. The function* $\texttt{construct\_pSA}(T', \Sigma', \Pi')$ *returns the pSA on the p-string T' from the constant alphabet Σ' and parameter alphabet Π'.*

In general, let the rank array R on T rank each suffix index in the string T to its position in the corresponding SA, i.e. $R[SA[i]] = i$. The length of the longest common prefix of neighboring p-suffixes in the pSA is stored in the parameterized longest common prefix ($pLCP$) array, analogous to the LCP for traditional strings.

Definition 2.1.12 ([53, 40]) Parameterized longest common prefix ($pLCP$) array: *The $pLCP$ array for a p-string T of length n stores the length of the longest common prefix of neighboring p-suffixes in pSA. We define* $\texttt{plcp}(a, b) = \max\{k \mid \texttt{prev}(a) =_k \texttt{prev}(b)\}$*. Then, $pLCP[1] = 0$ and $pLCP[i] = \texttt{plcp}(T[pSA[i]...n], T[pSA[i-1]...n])$, $2 \leq i \leq n$. The function* $\texttt{construct\_pLCP}(T', \Sigma', \Pi')$ *returns the $pLCP$ on the p-string T' from the constant alphabet Σ' and parameter alphabet Π'.*

For the example $T = AwBzABwz\$$ with $\texttt{prev}(T) = A0B0AB54\$$, we have $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$, $R = \{6, 5, 9, 4, 7, 8, 3, 2, 1\}$, and $pLCP = \{0, 0, 1, 1, 1, 0, 1, 0, 2\}$. In the same way that the traditional SA and LCP help provide efficient exact pattern matching [72, 1], together, the pSA and $pLCP$ help expedite p-matching [53, 40, 16]. In fact, the matching algorithms are the same, except for properly encoding the pattern to search. Consider finding all η_{occ} instances of the m-length encoding $\texttt{prev}(P)$ in the n-length p-string T. Using only the pSA on T, we can solve this problem in $O(m \log n + \eta_{occ})$ time. Using the pSA and processing on the $pLCP$ (see [72]), the problem is solved in $O(m + \log n + \eta_{occ})$ time.

Theorem 2.1.13 ([53]) *Using the pSA and pLCP on the n-length p-string T, all η_{occ} instances of the m-length p-string pattern P in T can be found in $O(m + \log n + \eta_{occ})$ time.*

As with the LCP, we can use the $pLCP$ to identify the longest prefix common between any two arbitrary p-suffixes of the n length T by using range minimum query operations, where $\texttt{rmq}(a, b, A) = \min\{A[a], A[a+1], ..., A[b]\}$. The following $\texttt{plcp}(i, j, T)$ returns the longest prefix common between two p-suffixes $\texttt{prev}(T[i...n])$ and $\texttt{prev}(T[j...n])$.

$$\texttt{plcp}(i, j, T) = \begin{cases} 0, \text{ if } i < 1 \vee i > n \vee j < 1 \vee j > n \\ n - i + 1, \text{ if } i = j \\ \texttt{rmq}(R_T[i] + 1, R_T[j], pLCP_T), \text{ if } R_T[i] < R_T[j] \\ \texttt{rmq}(R_T[j] + 1, R_T[i], pLCP_T), \text{ if } R_T[j] < R_T[i] \end{cases}$$

The same approach is used for traditional strings, i.e. $\texttt{lcp}(i, j, T)$, to compute the longest prefix common between two arbitrary suffixes at i and j in T. The $\texttt{rmq}$ calculation was proven by [25, 84, 85] to require $O(n)$ preprocessing in $O(n)$ space with $O(1)$ query time. While theoretically efficient, the $\texttt{rmq}$ preprocessing may be considered heavy in practice. When alternative solutions are available, we approach pattern matching problems with more lightweight techniques.

The *permuted-LCP* array for traditional strings, discussed in [59], is defined as a variation of the LCP array to offer space improvements.

Definition 2.1.14 ([59]) Permuted longest common prefix (*permuted-LCP*) array: *The permuted-LCP array for a traditional string W of length n stores the length of the*

longest common prefix of neighboring suffixes in the order that they appear in W. More formally, permuted-$LCP[i] = LCP[R[i]]$ or alternatively, permuted-$LCP[SA[i]] = LCP[i]$, $1 \leq i \leq n$.

In this work, we will propose and construct a *permuted-LCP* array for p-strings. Following the LCP-related arrays, we next define a related pattern matching data structure. The longest previous factor (LPF) on an n-length text W stores at $LPF[i]$ the length of the longest prefix common between $W[i...n]$ and some $W[h...n]$ with $h < i$.

Definition 2.1.15 ([35]) 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 $LPF[i] = \max(\{0\} \cup \{k \mid W[i...n] =_k W[h...n], 1 \leq h < i\}) = \max(\{0\} \cup \{\texttt{lcp}(i, h, W) \mid 1 \leq h < i\})$.*

The traditional string $W = AAABABAB\$$ yields $LPF = \{0, 2, 1, 0, 4, 3, 2, 1, 0\}$. In [19], we extend the LPF to p-strings.

Definition 2.1.16 ([19]) Parameterized longest previous factor ($pLPF$): *For an n-length p-string T, the $pLPF$ is defined for each index $1 \leq i \leq n$ such that $pLPF[i]$ is the length of the longest prefix common between the p-suffix $\texttt{prev}(T[i...n])$ and a p-suffix occurring before i in T, i.e. $pLPF[i] = \max(\{0\} \cup \{\texttt{plcp}(i, h, T) \mid 1 \leq h < i\})$. The array $\mathcal{L}$ has the location of each previous factor, i.e. $\mathcal{L}[i] = h$, or $\mathcal{L}[i] = 0$ if no such factor exists.*

For $T = aABwxAByz$, $pLPF = \{0, 0, 0, 1, 5, 4, 3, 2, 1\}$ and $\mathcal{L} = \{0, 0, 0, 1, 1, 2, 3, 4, 1\}$. In the scenario that multiple previous equal length prefixes exist for $\texttt{prev}(T[i...n])$, depending on the implementation, it may be the case that $\mathcal{L}[i]$ points to the leftmost previous occurrence.

Another pattern matching data structure is the *border* array ($\mathcal{B}$). For a string W, each $\mathcal{B}[i]$ stores the length of the longest prefix of $W[1..i]$ that matches a suffix of $W[1..i]$, which is useful for pattern matching.

Definition 2.1.17 ([87]) border array (*border* or $\mathcal{B}$): *For an n-length traditional string W, the border array is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}[1] = 0$ and otherwise $\mathcal{B}[i] = \max(\{0\} \cup \{k \mid W[1...k] = W[i - k + 1...i], k \geq 1 \wedge i - k + 1 > 1\})$.*

From the definition, we refer to the substrings $W[1...k]$ and $W[i-k+1...i]$ as borders. For the working example $W = AAABABAB\$$, we have $\mathcal{B} = \{0,1,2,0,1,0,1,0,0\}$. The parameterized border (*p-border*) array, which was originally defined as the `pfail` function in [56], redefines the traditional *border* array for p-strings and the p-match.

Definition 2.1.18 ([56]) parameterized border array (*p-border* or $\mathcal{B}_p$): *For an n-length p-string T, the p-border array is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}_p[1] = 0$ and otherwise $\mathcal{B}_p[i] = \max(\{0\} \cup \{k \mid \texttt{prev}(T[1...k]) = \texttt{prev}(T[i-k+1...i]), k \geq 1 \wedge i-k+1 > 1\})$.*

The borders under the encodings, i.e. $\texttt{prev}(T[1...k])$ and $\texttt{prev}(T[i-k+1...i])$, are referred to as *parameterized borders* or p-borders. The p-string $T = wzwz\$$ yields the array $\mathcal{B}_p = \{0,1,2,3,0\}$. Related to the *border* is the prefix array (PA), which stores the length of the longest common prefix between a string W and each suffix of W.

Definition 2.1.19 ([88]) Prefix array (PA): *Given the n-length W, $PA[i] = \texttt{lcp}(1, i, W)$ for $1 \leq i \leq n$.*

In [88], the compressed prefix array is proposed as a representation of the PA that removes the zero entries. We further refer to this succinct structure as the compact prefix array.

Definition 2.1.20 ([88]) Compact prefix array (cPA): *Given the PA for n-length W, the cPA is a pair of integer arrays (POS, LEN) that represent the left-to-right nonzero elements of PA, i.e. $PA[POS[i]] = LEN[i] \geq 1$ for $1 \leq i \leq n$.*

For the string $W = AABBAABB$, we have PA=$\{8,1,0,0,4,1,0,0\}$, POS=$\{1,2,5,6\}$, and LEN=$\{8,1,4,1\}$. Note that given i, it requires $O(\log \eta)$ time to access $PA[i]$ via the cPA with a binary search for i on POS, where $\eta = |cPA|$. In this work, we will define/construct the PA and cPA for p-strings.

2.2 Structural Strings

A structural string (s-string), formalized below, is a p-string where the parameter symbols have complementary pairings.

Definition 2.2.1 ([86]) Structural string (s-string): *An s-string is a p-string T of length n from $(\Sigma \cup \Pi)^*\$$ with complementary parameter pairings $p = (\pi_j, \pi_k)$ in $\Gamma = \{p_1, p_2, ..., p_{|\Gamma|}\}$, where $\pi_j, \pi_k \in \Pi$. Each $\pi \in \Pi$ is used in exactly one pair $p \in \Gamma$. Two parameters π_j and π_k may either be* (a) *complementary to one another with $j \neq k$, i.e.* $\texttt{complement}(\pi_j) = \pi_k$ *and* $\texttt{complement}(\pi_k) = \pi_j$, *or* (b) *when $j = k$, simply* $\texttt{complement}(\pi_j) = \pi_j$.

For the working alphabets $\Sigma = \{A, B, C\}$ and $\Pi = \{a, b, c, v, w, x, y, z\}$, consider $\Gamma = \{(a, a), (b, b), (c, c), (v, v), (w, x), (y, z)\}$. Example s-strings include $S = AxBzzywv\$$, $T = AwByyzxv\$$, and $U = AwByyxzv\$$. The matching of s-strings is known as structural matching (s-matching), which requires a bijection of the complementary symbols in addition to the p-match.

Definition 2.2.2 ([86]) Structural matching (s-match): *A pair of s-strings S and T are s-matches with $n = |S|$ iff $|S| = |T|$ and the following are true:* (1) *constant and terminal symbols in S and T match exactly and* (2) *there exists bijections between the parameters and complements of S and T. More formally, one of the following must be true for each i, $1 \leq i \leq n$:*

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

Notice that the s-match with $\Gamma = \{(\pi_1, \pi_1), ..., (\pi_{|\Pi|}, \pi_{|\Pi|})\}$, $\pi_i \in \Pi$, yields the p-match. In our working example, S and T s-match. The s-string U does not s-match with either S or T. The `sencode` scheme was defined in [86] to more easily detect an s-match by simply comparing the encodings rather than independently satisfying the steps in Definition 2.2.2. The `sencode` scheme is composed of the encodings `prev` (Definition 2.1.4) and `compl`. We define `compl` below for the complementary symbols.

Definition 2.2.3 ([86]) Complement (`compl`) encoding: *Given the n-length T and non-negative integers $\mathbb{Z}$, the function* `compl` *:* $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ (1) *encodes constants/terminals, say c, with the same symbol c and* (2) *encodes parameters, say π, to the distance to the previous complementary symbol* $\texttt{complement}(\pi)$*. We formally define* $\texttt{compl}(T)$ *below for each i, $1 \leq i \leq n$.*

$$\texttt{compl}(T)[i] = \begin{cases} T[i], \textit{ if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \textit{ if } T[i] \in \Pi \wedge T[i] \neq \texttt{complement}(T[j]) \textit{ for every } 1 \leq j < i \\ i - \max\{j \mid T[i] = \texttt{complement}(T[j]), 1 \leq j < i\}, \textit{ otherwise} \end{cases}$$

When the alphabets are unclear from context, we overload `compl` as follows: $\texttt{compl}(T, \Sigma, \Pi, \Gamma)$. In our working example, $\texttt{compl}(S) = \texttt{compl}(T) = A0B00150\$$ and $\texttt{compl}(U) = A0B00420\$$. Also, $\texttt{prev}(S) = \texttt{prev}(T) = \texttt{prev}(U) = A0B01000\$$. Next, we define `sencode` to combine both `prev` and `compl` for s-matching.

Definition 2.2.4 ([86]) Structural encoding (`sencode`): *Given the n-length T and non-negative integers $\mathbb{Z}$, the function* `sencode` *:* $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z} \cup \bar{\mathbb{Z}})^*\$$ (1) *encodes constants/terminals, say c, with the same symbol c and either* (2a) *encodes parameters π_1 to the distance to the previous π_1 or* (2b) *encodes parameters π_2 to the distance to the previous complementary parameter* $\texttt{complement}(\pi_2)$*. We formally define* $\texttt{sencode}(T)$ *below for each i, $1 \leq i \leq n$.*

$$\texttt{sencode}(T)[i] = \begin{cases} T[i], \textit{ if } T[i] \in (\Sigma \cup \{\$\}) \\ \texttt{prev}(T)[i], \textit{ if } \texttt{prev}(T)[i] > 0 \\ \overline{\texttt{compl}(T)[i]}, \textit{ if } \texttt{compl}(T)[i] > 0 \ \wedge \ \texttt{prev}(T)[i] = 0 \\ 0, \textit{ otherwise} \end{cases}$$

In order to compare s-strings, we now formalize the lexicographical ordering of symbols from the structural encodings.

Definition 2.2.5 `sencode` **lexicographical ordering:** *We define the ordering of symbols from the* `sencode` *encoding of the production* $(\Sigma \cup \mathbb{Z} \cup \bar{\mathbb{Z}})^*\$$ *to be* $\$ < \zeta \in \mathbb{Z} < \bar{\zeta} \in \bar{\mathbb{Z}} < \sigma \in \Sigma$*, where each ζ, $\bar{\zeta}$, and σ are lexicographically sorted within their respective alphabets.*

For two s-strings S and T, we can identify an s-match by comparing the strings under `sencode`.

Theorem 2.2.6 ([86]) *Two s-strings S and T s-match when* $\texttt{sencode}(S) = \texttt{sencode}(T)$. *Also,* $S \prec T$ *when* $\texttt{sencode}(S) \prec \texttt{sencode}(T)$ *and* $S \succ T$ *when* $\texttt{sencode}(S) \succ \texttt{sencode}(T)$.

In the working example, $\texttt{sencode}(U) = A\,0\,B\,0\,1\,\overline{4}\,\overline{2}\,0\,\$$ and $\texttt{sencode}(S) = \texttt{sencode}(T) = A\,0\,B\,0\,1\,\overline{1}\,\overline{5}\,0\,\$$. Thus, S and T are confirmed to s-match.

Since Γ allows at most two parameters to be complements to one another, we can construct `compl` and `sencode` on a string from an indexed alphabet, like `prev` (Lemma 2.1.6) and α, in linear time.

Lemma 2.2.7 *Given an n-length s-string T from indexed alphabets Σ and Π, the encodings* $\texttt{prev}(T)$, $\texttt{compl}(T)$, $\texttt{sencode}(T)$, *and* $\alpha(T)$ *can be constructed in $O(n)$ time with $O(|\Pi|)$ extra space.*

The suffixes $T[i...n]$ of an s-string under `sencode`, i.e. $\texttt{sencode}(T[i...n])$, are known as structural suffixes (s-suffixes). Like p-suffixes, the s-suffixes are dynamic, i.e. we cannot guarantee that $\texttt{sencode}(T[1...n])[k...n] = \texttt{sencode}(T[k...n])$ for $2 \leq k \leq n$.

Lemma 2.2.8 *Given an s-string T of length n, the suffixes of* $\texttt{sencode}(T)$ *are not necessarily the s-suffixes of T, i.e.* $\texttt{sencode}(T[i...n])$.

Proof Consider some s-string $T = \pi_1\pi_2\pi_2\pi_2\$$ with alphabets $\Sigma = \{\sigma_1\}$, $\Pi = \{\pi_1, \pi_2, \pi_3\}$, and $\Gamma = \{(\pi_1, \pi_3), (\pi_2, \pi_2)\}$. By Definition 2.2.4, it is the case that $\texttt{sencode}(T) = \texttt{prev}(T)$ and hence, this lemma holds by Lemma 2.1.10. □

Due to the dynamic nature of s-suffixes, the following function was defined in [86] to efficiently retrieve the jth symbol from the s-suffix $\texttt{sencode}(T[i...n])$.

Definition 2.2.9 ([86]) Structural suffix (s-suffix) symbol retrieval: *Given an n-length s-string T, let $prevT = \texttt{prev}(T)$, $complT = \texttt{compl}(T)$, 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* $\texttt{sencode}(T, i, j) \rightarrow (\Sigma \cup \mathbb{Z} \cup \overline{\mathbb{Z}} \cup \{\$\})$ *retrieves the symbol at j of the s-suffix* $\texttt{sencode}(T[i...n])$, *i.e.* $\texttt{sencode}(T[i...n])[j]$.

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

The significance of $\texttt{sencode}(T,i,j)$ is that we can retrieve s-suffix symbols effortlessly in an algorithmic environment. Note that this function requires alphabet membership questions, which can be answered in constant time via $\alpha(T)$ (Definition 2.1.2). Thus, each call to $\texttt{sencode}(T,i,j)$ executes in constant time.

Lemma 2.2.10 *Each call to* $\texttt{sencode}(T,i,j)$ *requires* $O(1)$ *time.*

2.2.1 Data Structures

Similar to the SA and pSA, we define the suffix array for the s-suffixes of an s-string.

Definition 2.2.11 ([16]) Structural suffix array (sSA): *The sSA for an n-length s-string T stores the lexicographical ordering of the indices i representing the s-suffixes* $\texttt{sencode}(T[i...n])$ *with* $1\leq i\leq n$, *such that* $\texttt{sencode}(T[sSA[q]...n])\prec\texttt{sencode}(T[sSA[q+1]...n])\ \forall\ q, 1\leq q<n$.

Following the LCP and $pLCP$, we now define the longest common prefix array for s-strings.

Definition 2.2.12 ([16]) Structural longest common prefix array ($sLCP$): *The $sLCP$ array for an n-length s-string T stores the length of the longest common prefix of neighboring s-suffixes in the sSA. We define* $\texttt{slcp}(a,b)=\max\{k\mid\texttt{sencode}(a)=_k\texttt{sencode}(b)\}$. *Then,* $sLCP[1]=0$ *and* $sLCP[i]=\texttt{slcp}(T[sSA[i]...n],T[sSA[i-1]...n]),\ 2\leq i\leq n$.

Similar to pattern matching with the SA/LCP and p-matching with the $pSA/pLCP$, we show, in [16], how to s-match with the $sSA/sLCP$.

Theorem 2.2.13 ([16]) *Given an n-length s-string T, the sSA for T, and the $sLCP$ for T, the locations of all the η_{occ} occurrences of s-matches between an m-length s-string pattern P and T can be found in $O(\log n+m+\eta_{occ})$ time.*

Chapter 3

Variations of the Parameterized Longest Previous Factor

Below, we list our publications related to this chapter.

- Beal, R., Adjeroh, D.: Variations of the parameterized longest previous factor. Journal of Discrete Algorithms, 16, 129-150 (2012)
- Beal, R., Adjeroh, D.: Parameterized longest previous factor. Theoretical Computer Science, 437, 21-34 (2012)

3.1 Introduction

The longest previous factor (LPF) problem finds for each suffix at i in an n-length traditional string $W = W[1]W[2]...W[n]$ a longest factor $W[h...n]$ with $1 \leq h < i \leq n$ preceding the suffix $W[i...n]$ in W. Crochemore and Ilie [35] studied this data structure for traditional strings. In order to construct the LPF array, it was shown in [35] that the suffix array SA is useful to quickly identify the most lexicographically similar suffixes that are candidate previous factors for the chosen suffix in question. The use of SA expedites the work to construct the LPF array in linear time. The linear time construction of the LPF data structure makes it a justified choice to use in various applications. The LPF array is naturally setup for applications in string compression [100] and detecting runs [70] within a string.

In [19], we introduce the parameterized longest previous factor ($pLPF$) to extend the traditional LPF problem to parameterized strings (p-strings). A p-string, introduced by Baker [11], is a generalized form of a string produced from the constant alphabet Σ and the parameter alphabet Π. The alphabet to which a symbol belongs determines exactly *how* the symbol is matched. 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. Prominent applications concerned with the p-match problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [86], discovering cloned code segments in a program [12], and even answering critical legal questions regarding the unauthorized use of intellectual property [98]. Baker [11] identifies that the p-match bijection can be handled by using a previous (`prev`) encoding scheme where a p-match exists between S and T if and only if $\texttt{prev}(S) = \texttt{prev}(T)$. The `prev` encoding codes each symbol s with the same constant s if $s \in \Sigma$. Otherwise when $s \in \Pi$, `prev` encodes s with the integer distance to the previous s in T or 0 if it is the first instance of s in T. For example, the following p-strings that represent program statements $f/a*q{-}{+}{+}q$ and $f/b*c{-}{+}{+}c$ over the alphabets $\Sigma = \{*, /, -, +\}$ and $\Pi = \{a, b, c, f, q\}$ successfully p-match since $\texttt{prev}(\text{“}f/a*q{-}{+}{+}q\text{”}) = \text{“}0/0*0{-}{+}{+}4\text{”} = \texttt{prev}(\text{“}f/b*c{-}{+}{+}c\text{”})$. By solving a problem with the p-match, we are also solving the same problem with an exact match when $|\Pi| = 0$ and a mapped match (m-match) when $|\Sigma| = 0$ [6]. We show in [19] that the $pLPF$ problem is not a straightforward extension of the LPF problem because of the added challenges of the p-match and dynamic nature of the parameterized suffixes (p-suffixes) under the `prev` encoding. We provide an algorithm in [19] to construct the $pLPF$ data structure in linear time. A significant contribution of [19] is identifying the connection between the $pLPF$ data structure with other popular data structures such as the LPF, longest common prefix (LCP), and parameterized longest common prefix ($pLCP$). We are influenced by the variations of the traditional LPF problem studied in [38, 37] to further define variations for the $pLPF$ problem.

Main Contributions: In this chapter, we extend the power of the $pLPF$ construction by addressing variations of the data structure with the same algorithm. Initially, we consider

the $pLPF$ problem and prove its linear time construction. We are the first to introduce a taxonomy for longest factor problems, which identifies that problems satisfying certain properties can be solved with the same $pLPF$ algorithm by simply altering the preprocessing and postprocessing. The $pLPF$ framework is the basis for the data structures in this research. First, it is proven that the $pLCP$ and the newly introduced *permuted-pLCP* can be constructed with the $pLPF$ framework in linear time. Next, we introduce three new variants of the $pLPF$ array, namely the parameterized longest not-equal factor ($pLneF$), parameterized longest reverse factor ($pLrF$), and parameterized longest factor (pLF), and prove that we can use the $pLPF$ framework to construct these variants in linear time. We identify that the *border* array [87], an important data structure in string pattern matching, is also a variant of the $pLPF$. It is then shown how to compute the parameterized-*border* (*p-border*) array in linear time using the $pLPF$ framework. For simplicity, throughout this chapter, we assume the most common case of p-strings – where the lexicographical relationship between p-suffixes is like traditional that of suffixes, i.e. for the n-length S, if $S[x] = S[i] = S[y]$ and $S[x...n] \prec S[i...n] \prec S[y...n]$ then $S[x+1...n] \prec S[i+1...n] \prec S[y+1...n]$ with $1 \leq x < n$, $1 \leq i < n$, $1 \leq y < n$. This corresponds to the case of Figure 1(a) of [19]. A straightforward implementation of the details in [19] allow the algorithms to support all the other cases identified. In terms of traditional data structures such as the longest common prefix (LCP) and longest previous factor (LPF), we prove that our p-string oriented algorithms can be used to solve these standard problems. Finally, we implement our algorithms considering all of the details in [19] and confirm the linear time nature of the constructions. We also show how our newfound algorithms compare with standard algorithms in terms of the traditional LCP and LPF constructions. The following theorems formalize our core contributions.

Theorem 3.3.3. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA_T, the parameterized suffix array for T, the algorithm* `compute_pLPF` *constructs the $pLPF$ array in $O(n)$ time.*

Theorem 3.4.2. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA_T, the parameterized suffix array for T, the* `construct` *algorithm can be used to construct the $pLCP$ and permuted-$pLCP$ arrays in $O(n)$ time.*

Theorem 3.4.9. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, pSA_T, $Q_1 = T[1...n-1]\$_1$, $Q_2 = T[1...n-1]^R\$_2$, $Q = Q_1 \circ Q_2$, $prevT_1 = \texttt{prev}(Q_1)$, $prevT_2 = \texttt{prev}(Q_2)$, and pSA_Q, the pLneF, pLrF, and pLF data structures are each constructed in $O(n)$ time.*

Theorem 3.4.10. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA_T, the parameterized suffix array for T, the algorithm* `compute_p-border` *computes the p-border array in $O(n)$ expected time.*

3.2 Background / Related Work

Baker [12] identifies three types of pattern matching: (1) exact matching, (2) parameterized matching (p-match), and (3) matching with modifications. The p-match generalizes exact matching by using a parameterized string (p-string) composed of symbols from a constant symbol alphabet Σ and a parameter alphabet Π. A p-match exists between a pair of p-strings S and T of length n when (1) the constant symbols $\sigma \in \Sigma$ match and (2) there exists a bijection of parameter symbols $\pi \in \Pi$ between the p-strings. The first p-match breakthroughs, namely, the `prev` encoding and the parameterized suffix tree (p-suffix tree) were introduced by Baker [11]. The p-suffix tree construction time was improved by Baker in [13]. Other contributions in the area of parameterized suffix trees include the improved construction in [65] and the randomized algorithms in [31, 67, 68]. Like the traditional suffix tree [47, 87, 1], the p-suffix tree [11] implementation suffers from a large memory footprint. Other solutions that address the p-match problem without the space limitations of the p-suffix tree include the parameterized-KMP [6] and parameterized-BM [14], variants of traditional pattern matching approaches. Idury et al. [56] studied the multiple p-match problem using an automaton and a structure that is now referred to as the parameterized-*border* (*p-border*) array. In [54, 55], I et al. studied how to verify whether a given array is a valid *p-border* array. The parameterized suffix array (p-suffix array) and the parameterized longest common prefix ($pLCP$) array combination is analogous to the suffix array and LCP array for traditional strings [72, 47, 87, 1], which is both time and space efficient for pattern matching. Direct p-suffix array and $pLCP$ construction was first introduced by Deguchi et al. [40] for binary strings, which required $O(n)$ work. Deguchi and colleagues

[53] later proposed the first approach to p-suffix sorting and $pLCP$ construction with an arbitrary alphabet size theoretically requiring $O(n^2)$ time in the worst case. We introduce new algorithms in [21, 16] to p-suffix sort in linear time on average using coding methods from information theory. In [21], we introduce improved theoretical worst case algorithms for p-suffix sorting based on arithmetic coding techniques.

Table 3.1: Data structures for string $W = AAABABAB\$$

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

In this research, we look at auxiliary data structures constructed from the suffix array. In a novel application of the suffix array and the corresponding LCP array, Crochemore and Ilie [35] introduced the longest previous factor (LPF) problem for traditional strings. Table 3.1 shows an example of the LPF array and related data structures 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 [47]. Crochemore and Ilie [35] 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. In [38, 37], Crochemore and colleagues extended their ideas related to the LPF data structure to propose new problems regarding various types of previous factors in a string. Some of these variations observe reverse factors, which is the case with the longest previous non-overlapping reverse factor ($LPnrF$) data structure. The $LPnrF$ data structure stores, for each suffix at position i, the length of the longest reverse factor occurring before index i in the string W. The longest previous reverse factor ($LPrF$) and the longest previous non-overlapping factor ($LPnF$) are other variants of the LPF array studied in [37]. The significance of an efficient construction of the LPF data structure is

the ability to simplify computations in various string analysis procedures. Typical examples include computing the Lempel-Ziv factorization [100, 36], which is fundamental in string compression algorithms such as the UNIX *gzip* utility [47, 87] and in algorithms for detecting repeats in a string [70].

We extend the LPF data structure by generalizing the problem for p-strings with the parameterized longest previous factor $pLPF$ data structure proposed in [19]. The work in [19] also constructs the $pLCP$, traditional LPF, and traditional LCP data structures in linear time. The solutions in [19] are the first theoretical linear time claims that address the $pLCP$ array construction. In this research, we consider the $pLPF$ construction to serve as a foundation for popular data structures such as LPF, LCP, $pLCP$, and the *border* and *p-border* arrays in addition to our newly proposed variations of the longest factor problem. Our motivation to further study longest factor problems in terms of p-strings is the power of parameterization to provide data structures for p-string applications and also address problems for traditional strings.

3.3 Parameterized LPF

3.3.1 Preprocessing

Before developing the parameterized longest previous factor ($pLPF$) data structure, we begin by identifying the preprocessing involved. The intermediate data structures preprocessed assist with the efficient construction of the $pLPF$ data structure. Intuitively, the act of finding a longest previous factor demands that we oracle a few "candidate" factors. As with [35, 38, 37], the suffix array is used to efficiently find such factors for traditional strings. Crochemore and Ilie [35] efficiently solve the LPF problem for a traditional n-length 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 store the suffix $h < i$ respectively preceding and succeeding the suffix i in SA; when no such suffix exists, the element is denoted by -1. The conceptual idea to compute the $prev_<$ and $prev_>$ arrays in linear time via deletions in a doubly linked list of the SA was suggested in [35]. We provide the preprocessing required in

Algorithm 3-1. Preprocessing algorithm for $before_<$ and $before_>$ arrays

```
1  int[] preprocess1(int SA[n], int d) {
2     int q[n], i
3     Node* ptr[n]
4     init( )
5     insert(−1)
6     for(i = 1 to n)
7        ptr[SA[i]] = insert(SA[i])
8     insert(−1)
9     for(i = n to 1) {
10       switch(d){
11          case BEFORE<: q[i] = ptr[i]−>previous−>suf
12             break
13          case BEFORE>: q[i] = ptr[i]−>next−>suf
14             break
15       }delete(ptr[i])
16    }clear( )
17    return q
18 }
```

Algorithm 3-1.

It is apparent from Definition 2.1.11 that p-suffixes are different from traditional suffixes due to the `prev` encoding. Later, we will clearly identify the specific challenges introduced because of the `prev` encoding of Definition 2.1.4. At this stage, however, it is important to observe that the computations of these "candidate" factors $prev_<$ and $prev_>$ do not depend on the individual symbols and rather, only depend on the location of the factor in the string. Thus, we can also use a p-suffix array to find these factors for p-strings using Algorithm 3-1. Furthermore, we will refer to $prev_<$ and $prev_>$ respectively as $before_<$ and $before_>$ (identified by readable constants $BEFORE_<$ and $BEFORE_>$), in order to avoid confusion with the `prev` encoding for p-strings. Table 3.2 displays a general example of the $before_<$ and $before_>$ arrays.

3.3.2 Construction

The parameterized longest previous factor ($pLPF$) (Definition 2.1.16) was originally defined in [19]. In the following, we define the $pLPF$ in terms of the $before_<$ and $before_>$ arrays.

Definition 3.3.1 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 store the length of the longest factor *between a p-suffix and a previous p-suffix occurring in T. In addition to Definition 2.1.16, we define* $pLPF[i] = \max(\{0\} \cup \{k \mid \texttt{prev}(T[i...n]) =_k \texttt{prev}(T[h...n]), 1 \leq h < i\}) = \max\{\texttt{plcp}(i, before_<[i], T), \texttt{plcp}(i, before_>[i], T)\}$.

Constructing the $pLPF$ array 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 2.1.10 indicates 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. Consider the n-length p-string $T = AAAwBxyyAAAzwwB\$$ where $\Sigma = \{A, B, C\}$ and $\Pi = \{a, b, c, v, w, x, y, z\}$. The suffix at $i = 7$ is $T[i...n] = yyAAAzwwB\$$ and the suffix at $j = i+1 = 8$ is $T[j...n] = yAAAzwwB\$$. The p-suffix at index i is $\texttt{prev}(T[i...n]) = \texttt{prev}(yyAAAzwwB\$) = 01AAA001B\$$ and the p-suffix at index j is $\texttt{prev}(T[j...n]) = \texttt{prev}(yAAAzwwB\$) = 0AAA001B\$$. Notice the relationship between the traditional suffixes since $T[i...n] = T[i] \circ T[j...n]$ whereas for p-suffixes, it is the case that $\texttt{prev}(T[i...n]) \neq \texttt{prev}(T[i]) \circ \texttt{prev}(T[j...n])$ and even $\texttt{prev}(T[i...n]) \neq \texttt{prev}(T[i...n])[1] \circ \texttt{prev}(T[j...n])$. The fact that we cannot exploit the property of relating suffixes complicates p-string problems.

Table 3.2 shows the $pLPF$ for $T = AAAwBxyyAAAzwwB\$$. We note the intricacies of Lemma 2.1.10 since simply using the traditional LPF construction does not result in the correct $pLPF$ array: (1) $LPF_T = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 0, 1, 2, 1, 0\}$, (2) $LPF_{\texttt{prev}(T)} = \{0, 2, 1, 0, 0, 1, 1, 0, 4, 3, 2, 1, 0, 1, 1, 0\}$, and (3) $LPF_{\texttt{forw}(T)} = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 3, 2, 1, 1, 0\}$. Essentially, Lemma 2.1.10 demands that we individually handle p-suffixes in a way differing from the approach for traditional suffixes used in LPF construction.

Given the $before_<$ and $before_>$ arrays, the element $LPF[i]$ of the traditional LPF 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 $LPF[i] \geq LPF[i-1] - 1$, which is a variant of the extension property used in LCP construction

Table 3.2: $pLPF$ calculation for p-string $T = AAAwBxyyAAAzwwB\$$

i	$pSA[i]$	$pLCP[i]$	$\texttt{prev}(T[pSA[i]...n])$	$before_<[pSA[i]]$	$before_>[pSA[i]]$	$pLPF[i]$
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

proven by Kasai et al. [61]. We prove that this same property holds for the $pLPF$ problem defined on p-strings. The proof assumes that p-strings follow Figure 1(a) in [19] (for the n-length p-string S, if $S[x] = S[i] = S[y]$ and $S[x...n] \prec S[i...n] \prec S[y...n]$ then $S[x+1...n] \prec S[i+1...n] \prec S[y+1...n]$ with $1 \leq x < n$, $1 \leq i < n$, $1 \leq y < n$).

Lemma 3.3.2 *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 3.3.1 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 $\texttt{prev}(T[j...n]) =_q \texttt{prev}(T[g...n])$. Thus, $pLPF[j] = q$. Shifting the computation to $i = j+1$, we lose the symbols $\texttt{prev}(T[j])$ and $\texttt{prev}(T[g])$ in the p-suffixes at j and g respectively. By Theorem 2.1.9, $\texttt{prev}(T[j...j+q-1]) = \texttt{prev}(T[g...g+q-1]) \Rightarrow \texttt{prev}(T[j]) = \texttt{prev}(T[g])$ and as a consequence of the `prev` encoding in Definition 2.1.4 we have $\texttt{prev}(T[i...n]) =_{q-1} \texttt{prev}(T[g+1...n])$. 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. □

Algorithm 3-2. *pLPF* computation

```
1  int[] compute_pLPF(char T[n], int pSA[n]) {
2     return construct(preprocess1(pSA,BEFORE_<),preprocess1(pSA,BEFORE_>),
3                                                              prev(T),prev(T))
4  }
```

Algorithm 3-3. General construction

```
1  boolean extend=true
2  int[] construct(int arr1[n], int arr2[n], int prevT1[n], int prevT2[n]) {
3     int  z[n], z1=0, z2=0, i, j=0, k=0
4     for(i = 1 to n) {
5        if(extend) {
6           j = max{0,z1−1}, k = max{0,z2−1}
7           /* see [19] for additional details */
8        }if(arr1 ≠ null) z1 = Λ(i,arr1[i],j,prevT1,prevT2)
9        if(arr2 ≠ null) z2 = Λ(i,arr2[i],k,prevT1,prevT2)
10       z[i] = max{z1,z2}
11    }return z
12 }
```

Algorithm 3-4. p-matcher function Λ

```
1  int Λ(int a, int b, int q, int prevT1[n], int prevT2[n]) {
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 = prevT1[a+q], y = prevT2[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 }
```

At this point, we identify that the individual extensions of the p-matches $\texttt{prev}(T[i...n]) =_j \texttt{prev}(T[before_<[i]...n])$ and $\texttt{prev}(T[i...n]) =_k \texttt{prev}(T[before_>[i]...n])$ for sequential i and some $j > 0$ and $k > 0$ require special consideration from the Ω function details in our research [19]. As noted earlier in this chapter, we assume the most common case of p-strings where p-suffixes are classified by Figure 1(a) of [19], in which the lexicographical relationship between p-suffixes is like that of traditional suffixes; we note that Lemma 3.3.2 would change slightly if the other relationships are considered. A straightforward implementation of the details in [19] will extend this to support all classifications. As a result, Lemma 3.3.2 permits us to adapt the basic algorithm `compute_LPF` given in [35] for our $pLPF$ problem by extending the solution to incorporate the dynamic matching of p-suffixes. In [19], we provide the construction of the $pLPF$ data structure by comparing both "candidate" p-suffixes using data from the same `prev` encoding. Since we will be looking at a multitude of $pLPF$ variations in this research, we give a more generalized $pLPF$ solution here that compares "candidate" p-suffixes on two separate, individual `prev` encodings. In [19], we take advantage of the fact that we can reuse the same `prev` encoding during the construction of the $pLPF$ array. Extending the algorithm to use individual `prev` encodings does not change the behavior of the algorithm and rather, it gives us the extra flexibility to perform the same type of construction on parallel `prev` encodings. The `compute_pLPF` in Algorithm 3-2 shows how to correctly perform the preprocessing of $before_<$ and $before_>$ via Algorithm 3-1 and call the construction routine in Algorithm 3-3. This construction routine makes use of the p-matcher Λ in Algorithm 3-4 to properly handle the sophisticated matching of p-suffixes, the dynamic suffixes under the `prev` encoding.

The main difference between our $pLPF$ solution and the traditional LPF solution in [35] is the actual pattern matching performed. For the $pLPF$, the p-suffixes must be p-matched as formalized in Definition 2.1.3. The role of Λ is to *extend* the matches between the p-suffix at a, constructed from the $prevT_1$ parameter, and at b, constructed from the $prevT_2$ parameter, beyond the initial q symbols by directly comparing constant/terminal symbols and comparing the dynamically adjusted parameter encodings for each p-suffix. For example, consider $prevT_1 = prevT_2 = \texttt{prev}(T)$. Then, the compared symbols at position $(q + 1)$ in the p-suffixes $\texttt{prev}(T[a...n])$ and $\texttt{prev}(T[b...n])$ are adjusted to 0 when they are

either already 0, i.e. $prevT_1[a+q] = 0$ or $prevT_2[b+q] = 0$, or the symbol at $prevT_1[a+q]$ or $prevT_2[b+q]$ encodes the distance to the previous occurrence of a parameter before the initial symbol of the p-suffix a or b in T, i.e. the compared symbols are the first occurrences of that parameter in the p-suffix. Otherwise, the compared parameter symbols are simply encoded to $prevT_1[a+q]$ and $prevT_2[b+q]$, signifying that the parameters have a previous occurrence in the p-suffix. See [21, 16] for an extended discussion and proof on the relationship between p-suffixes. Nonetheless, this dynamic adjusting does not add to the theoretical complexity of the algorithm.

Theorem 3.3.3 *Given an n-length p-string T, $prevT =$* `prev`(T)*, the* `prev` *encoding of T, and pSA_T, the parameterized suffix array for T, the algorithm* `compute_pLPF` *constructs the $pLPF$ array in $O(n)$ time.*

Proof It follows from Lemma 3.3.2 that `compute_pLPF` exploits the properties of $pLPF$ to correctly compute and extend factors. Computing the arrays $before_<$ and $before_>$ clearly require $O(n)$ processing via Algorithm 3-1. What remains now is to show that, starting with the call in Algorithm 3-2 between Algorithm 3-3 and Algorithm 3-4, the total number of times that the body of the **while** loop (lines 6-15 in Algorithm 3-4) will be executed is in $O(n)$. Let $prevT_1 = prevT_2 = prevT$. The number of iterations of the **while** loop is given by the number of symbols matched between $prevT_1$ and $prevT_2$, 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-suffix at position a retrieved from $prevT_1$ and at position b retrieved from $prevT_2$ are the longest p-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 p-suffixes and each comparison, which clearly requires $O(1)$ work, will increment q. Lemma 3.3.2 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 p-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 Algorithm 3-3. Thus, the total work is $O(n)$. □

Our algorithm `compute_pLPF` is motivated by the `compute_LPF` algorithm in [35]. We also observe that similar pattern matching mechanisms as the one used between the **for** loop in Algorithm 3-3 and the **while** loop in Algorithm 3-4 exist in standard string processing, for example in computing the *border* array discussed in [87].

3.3.3 A Taxonomy for Longest Factor Problems

The construction algorithm for the $pLPF$ data structure that we provide is very powerful. In the future sections of this chapter, we show how to use this very construction to approach various significant and other newly proposed data structures. Influenced by the variants of the LPF discussed in [38, 37], we take longest factor problems to a more theoretical level by introducing a class of problems referred to as "longest factor symbol comparison problems" and appropriately identify the subclasses. This discussion is aimed to show how we may "reuse" the $pLPF$ construction. By no means is this section a complete compilation of thoughts and theory on the classes and subclasses of longest factor problems, which is additional research beyond the scope of this work. Instead, we provide the first true look at longest factor problems in terms of classes and identify types of problems that can be solved by our $pLPF$ construction.

A longest factor symbol comparison problem, or longest factor problem for short, produces a data structure d on the n-length text T, where each $d[i]$ stores the maximum number of symbols (longest factor) common between $T[i...n]$ and some other suffix in T. Without loss of generality, these problems can be defined for p-strings or traditional strings. We say that problems like the $pLPF$ with the property of Lemma 3.3.2 have the *extension property*. See Definition 3.3.4 for a more formal description of the property.

Definition 3.3.4 Extension property: *Let p be a longest factor problem yielding the data structure d on the n-length T. Let $y = \max\{0, d[i-1]-1\}$ and some $\mathcal{J} \geq y$. If each $d[i] = \max\{y, \mathcal{J}\}$, the y symbols at $d[i-1]$ are first considered in the answer $d[i]$ until the true solution $\mathcal{J}$ is found, and $d[i] \geq d[i-1]-1$ for all possible d and $1 < i \leq n$, then we say that problem p has the* extension property*.*

Considering all of the p-suffix classifications in [19] will slightly modify this property. We

identify that problems possessing the *extension property* may be of two main classes: (1) with restrictions or (2) without restrictions. Specifically, longest factor problems with restrictions may have a requirement on perhaps the location of the match. The way in which these restrictions are handled create the additional subclasses of problems: (type-1) pre-satisfied with the preprocessing of arrays, (type-2) post-satisfied requiring additional verification and work afterwards, or (type-3) both pre-satisfied and post-satisfied. We do not consider type-2 problems since they do not take advantage of preprocessing.

Problems like the $pLPF$ are said to have type-1 pre-satisfied restrictions since the $before_<$ and $before_>$ arrays handle the only restriction that the longest factor precede the suffix in question within the p-string. In other words, after the execution of the `construct` routine in Algorithm 3-3, there is no real additional work required since all of the restrictions are handled beforehand. Examples of new data structures that we introduce which fit this scheme include the $pLneF$ and $pLrF$.

Consider for example a problem like the longest previous non-overlapping factor ($LPnF$) in [38, 37]. We can view this problem as satisfying restrictions by either type-1 or type-3. More specifically, the $LPnF$ problem requires that the longest factor precede the suffix at position i in the string *and* also, this match must not overlap the suffix at i. In this case, we can easily pre-satisfy the restriction that a "candidate" suffix occurs beforehand however, the fact that the match must be both the longest and not-overlap the suffix at i will require more sophisticated preprocessing as in [38, 37]. We can view this problem in terms of either type-1 preprocessing [38, 37] or a type-3 mixture of preprocessing and postprocessing. By observing the $LPnF$ as a type-3 problem, we can preprocess "candidate" suffixes to check and afterwards, decide if they meet the non-overlapping criteria and re-evaluate where necessary. We view LCP-related data structures in this way.

By utilizing preprocessed arrays in the same way as used by the $pLPF$ construction, it is possible to solve longest factor problems possessing the *extension property* with type-1 pre-satisfied restrictions simply using Algorithms 3-3 and 3-4.

Proposition 3.3.5 *Let p be a longest factor problem satisfying the* extension property*. Further, let p be in a class of problems with type-1 pre-satisfied restrictions similar in form to*

the pLPF preprocessed arrays, which pre-satisfy the restrictions of the pLPF problem. If the preprocessed arrays are constructed in linear time, then p can be solved in linear time with Algorithm 3-3 and Algorithm 3-4, the same construction routines as the pLPF problem.

Proof Algorithm 3-3 and Algorithm 3-4 clearly implement the *extension property* in Definition 3.3.4. Further, the algorithms compute a longest factor with preprocessed arrays to pre-satisfy restrictions. Since p is a type-1 pre-satisfied restriction problem with the *extension property*, the algorithms will compute p given the correct preprocessed arrays. It follows from Theorem 3.3.3 that Algorithms 3-3 and 3-4 execute in linear time given that the preprocessed arrays are constructed in linear time. □

Similar in nature to Proposition 3.3.5, we can prove that any type-3 longest factor problem, satisfying conditions with both preprocessing and postprocessing, is solved in linear time given that the preprocessing and postprocessing execute in linear time. In passing, we briefly mention that our $pLPF$ construction can solve various other p-matching problems by setting *extend* =**false** in Algorithm 3-3. (These problems can be grouped into a separate classification within the taxonomy.) We use this technique to construct the *p-border* array. Throughout the remainder of this chapter, we consider variations of the $pLPF$ data structure that can be addressed with the $pLPF$ construction algorithm.

3.4 Variations on a Theme - Parameterized Strings

Our taxonomy of longest factor problems provides a way to view these problems in groups or classes. In terms of the $pLPF$ problem, such a classification gives insight into a group of related problems that may be solved with the same general framework. Earlier in the chapter, we discussed the capability to solve related variations of the $pLPF$ array with the same construction algorithm. This was formalized in Proposition 3.3.5. In this section, we construct popular p-string data structures (such as the $pLCP$ and the *p-border* array) and newly proposed data structures defined for p-strings with the same framework used to construct the $pLPF$ array. Note that throughout this section, we are able to dynamically and efficiently construct all p-suffixes from $prevT = \texttt{prev}(T)$, as utilized in the construction

of the $pLPF$ array (see the p-matcher Λ in Algorithm 3-4). For flow of discussion and clarity, where appropriate, we choose to simply represent these dynamically constructed p-suffixes by the suffix under the `prev` encoding, i.e. $\texttt{prev}(T[i...n])$, rather than rehashing the relationship between $prevT = \texttt{prev}(T)$ and the individual p-suffixes $\texttt{prev}(T[i...n])$.

3.4.1 Preprocessing

The preprocessing used by the algorithms throughout this chapter is handled by Algorithm 3-5. Since the preprocessing does not depend on the individual symbols of a string, we may use this algorithm for either traditional strings or p-strings. So, the following discussion applies to both p-strings with the pSA and traditional strings with SA. The algorithm is used by providing a suffix array SA for a string (or alternatively a p-suffix array pSA for a p-string) and the constant d representing the array to construct (see the constants later). Algorithm 3-5 has the ability to construct each of the following preprocessed arrays: $before_<$, $before_>$, $after_<$, $after_>$, $neq_<$, $neq_>$, $rev_<$, and $rev_>$. In the algorithm, each of these arrays are identified by the respective readable constants $BEFORE_<$, $BEFORE_>$, $AFTER_<$, $AFTER_>$, $NEQ_<$, $NEQ_>$, $REV_<$, and $REV_>$. A trivial analysis of the algorithm given a valid suffix array will prove that any of these arrays are clearly computed in time linear to the length of the string.

Now, we briefly discuss the makeup of each array given an n-length string T. The $before_<$ and $before_>$ arrays were previously discussed in this research (see the example in Table 3.2) since they play a significant role in our computation of the $pLPF$ data structure. Recall that $before_<$ and $before_>$ store a list for each index i in T of the lexicographically closest suffixes *before* i in T; if no such index exists, this is signified in the array by -1. In the case of $before_<$, when we say *lexicographically closest*, we are referring to such a suffix at h with $h < i$ that precedes the location of i in the SA. Similarly, when we say *lexicographically closest* in the case of $before_>$, we are referring to a suffix at $h < i$ that succeeds the location of i in SA. All of the arrays share the form $X_<$ and $X_>$ where the subscript determines if the array refers to indices that precede or succeed the element in terms of the SA. The $after_<$ and $after_>$ arrays are defined exactly the same as $before_<$ and $before_>$ respectively, except

that we are reporting the suffix j *after* i, namely $j > i$. The following arrays are slightly different. The $neq_<$ and $neq_>$ arrays relax the conditions regarding the index of the suffix and instead, simply store a list of the preceding and succeeding suffixes in the SA. The $rev_<$ and $rev_>$ arrays store the indices of a suffix in $T[n-1]...T[2]T[1]\$$ that is lexicographically closest to a suffix in T. Table 3.3 provides a basic example of these arrays.

Table 3.3: Preprocessed arrays for p-string $T = AAAwBxyyAAAzwwB\$$

i	$pSA[i]$	$after_<[pSA[i]]$	$after_>[pSA[i]]$	$neq_<[pSA[i]]$	$neq_>[pSA[i]]$	$rev_<[pSA[i]]$	$rev_>[pSA[i]]$
1	16	-1	-1	-1	6	-1	1
2	6	16	12	16	12	1	14
3	12	16	13	6	7	1	14
4	7	12	13	12	13	9	5
5	13	16	14	7	8	9	5
6	8	13	14	13	14	5	13
7	14	16	15	8	4	13	7
8	4	14	11	14	11	13	7
9	11	14	15	4	3	2	10
10	3	11	10	11	10	10	3
11	10	11	15	3	2	3	11
12	2	10	9	10	9	11	4
13	9	10	15	2	1	4	12
14	1	9	15	9	15	12	16
15	15	16	-1	1	5	12	16
16	5	15	-1	15	-1	12	16

3.4.2 Parameterized LCP and Permuted Parameterized LCP

In this section, we identify the connection between the $pLCP$ and the $pLPF$. We show a construction of the $pLCP$ (identified by the readable constant $PLCP$) and propose/construct the *permuted-pLCP* array (identified by constant $PERMUTED\_PLCP$), a *permuted-LCP* array for p-strings.

It was discovered in [61] that the LCP data structure for a traditional n-length string W may be computed in linear time by *extending* each element of the array in the order that the suffixes appear in the string. More specifically, each element at index i of the LCP array is defined of neighboring suffixes, i.e. the suffixes at i and $i-1$ for $i > 1$ in the suffix array SA. This means that the elements of the LCP array are defined in same order as the SA. However, since the suffixes of a string are related, Kasai et al. [61] identified that the following property holds: $LCP[R[i]] \geq LCP[R[i-1]] - 1$ for $i > 1$ where R is the rank array, the inverse of the SA. This is the very same relationship that we term as the *extension property* in Definition 3.3.4. Rather than constructing the LCP array elements in order $LCP[1], LCP[2], ..., LCP[n]$, it is advantageous to reuse comparisons

Algorithm 3-5. Preprocessing algorithm

```
1  int[] preprocess(int SA[h], int n, int d) {
2     int q[n], i, j = 1, c = −1, s = 1, e = h
3     Node *ptr[h]
4     if(d = BEFORE< ∨ d = BEFORE>)
5        q = preprocess1(SA,d)
6     else if(d = REV< ∨ d = REV>){
7        if(d = REV>) {
8           s = h
9           e = 1
10       }for(i = s to e) {
11          if(SA[i] > n)
12             c = SA[i] − n
13          else
14             q[SA[i]] = c
15       }
16    }else{
17       init( )
18       insert(−1)
19       for(i = 1 to h)
20          ptr[SA[i]] = insert(SA[i])
21       insert(−1)
22       for(i = h to 1) {
23          switch(d) {
24             case AFTER<:
25                q[j] = ptr[j]−>previous−>suf
26                delete(ptr[j++])
27                break
28             case AFTER>:
29                q[j] = ptr[j]−>next−>suf
30                delete(ptr[j++])
31                break
32             case NEQ<:
33                q[i] = ptr[i]−>previous−>suf
34                break
35             case NEQ>:
36                q[i] = ptr[i]−>next−>suf
37                break
38          }
39       }clear( )
40    }return q
41 }
```

at previous stages by constructing adjacent elements in the order that they appear in the string: $LCP[R[1]], LCP[R[2]], ..., LCP[R[n]]$. This particular ordering was later defined as the *permuted-LCP* data structure in [59]. Earlier, we defined the traditional *permuted-LCP* in Definition 2.1.14. Now, we are the first to define the analogous *permuted-pLCP* data structure for p-strings.

Definition 3.4.1 Permuted parameterized longest common prefix (*permuted-pLCP*) array: *The permuted-pLCP array for a p-string T of length n stores the length of the parameterized longest common prefix of neighboring p-suffixes in the order that they appear in T. More formally, permuted-$pLCP[i] = pLCP[R[i]]$ or alternatively, permuted-$pLCP[pSA[i]] = pLCP[i]$, $1 \leq i \leq n$.*

In retrospect, Kasai et al. [61] actually constructed the LCP array by filling in elements as they appear in the *permuted-LCP* array. Deguchi et al. [53, 40] studied the problem of constructing the $pLCP$ array given the pSA. They showed a construction algorithm to compute the $pLCP$ array via a non-trivial modification of the traditional LCP construction by Kasai et al. [61] that executes theoretically in $O(n^2)$ time. By reusing the same `construct` algorithm used by our `compute_pLPF` routine, we introduce a way to construct the $pLCP$ array in linear time. This linear time construction is possible because of two key observations. First, the p-matcher Λ in Algorithm 3-4 shows how we can reuse elements from a single encoding `prev`(T) to access the individual symbols of the dynamically changing p-suffixes. Second, by Proposition 3.3.5, if we can view the *permuted-LCP* and also, the *permuted-pLCP* data structure as a longest factor problem with type-1 pre-satisfied restrictions, then we can construct the data structure in linear time.

In terms of the pre-satisfied conditions, we can mirror the array construction used by the `compute_pLPF` algorithm. The only true difference is in the content of the arrays. Consider the arrays $before_<$, which for each i in the p-string T stores the p-suffix $h < i$ positioned prior to the p-suffix i in pSA, and $after_<$, which for each i in T stores the p-suffix $j > i$ also positioned prior to the p-suffix i in pSA. 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, the element $pLCP[R[i]]$ or *permuted-*$pLCP[i]$ is constructed – no additional restrictions are needed on the *way* that the longest factors are found. Rather, we only need to transition the array from the *permuted-pLCP* to the $pLCP$. From Definition 3.4.1, this transition is accomplished with a simple linear time reordering

Algorithm 3-6. *pLCP* and *permuted-pLCP* computations

```
1  int[] compute_pLCP1(char T[n], int pSA[n], int d) {
2     int pLCP[n], X[n], Y[n], R[n], prevT[n]=prev(T), i
3     for(i = 1 to n)
4        R[pSA[i]] = i
5     X = construct(preprocess(pSA,n,BEFORE<),null,prevT,prevT)
6     Y = construct(preprocess(pSA,n,AFTER<),null,prevT,prevT)
7     for(i = 1 to n) {
8        switch(d){
9           case PERMUTED_PLCP:
10             pLCP[i] = max{X[i],Y[i]}
11             break
12          case PLCP:
13             pLCP[R[i]] = max{X[i],Y[i]}
14             break
15       }
16    }return pLCP
17 }
```

of elements. The construction of the arrays $before_<$ and $after_<$ is shown in Algorithm 3-5. Theorem 3.4.2 proves that the *pLCP* and *permuted-pLCP* constructions can be performed in linear time.

Theorem 3.4.2 *Given an n-length p-string T, prevT =* `prev`(T)*, the* `prev` *encoding of T, and* pSA_T*, the parameterized suffix array for T, the* `construct` *algorithm can be used to construct the pLCP and permuted-pLCP arrays in* $O(n)$ *time.*

Proof We can clearly relax the p-suffix selection restrictions enforced by the *pLPF* problem in Lemma 3.3.2 to exploit the idea of *extending* factors. Subsequently, only the parameters of Algorithms 3-3 and 3-4 impose such restrictions. Let $R[1...n]$ be the rank array representing the inverse of pSA_T. Let $before_<[1...n]$ and $after_<[1...n]$ store, for all the i in T, the p-suffixes at $h < i$ at position $R[h]$ in pSA_T and $j > i$ at position $R[j]$ in pSA_T, respectively, that are positioned prior to the p-suffix i at position $R[i]$ in pSA_T; when no such p-suffix exists, the element is denoted by -1. Let $X =$ `construct`(`preprocess`$(pSA_T, n, BEFORE_<)$, **null**$, prevT, prevT)$ and $Y =$ `construct`(`preprocess`$(pSA_T, n, AFTER_<)$, **null**$, prevT, prevT)$. We prove that the *pLCP* is constructed by the statement $pLCP[R[i]] = \max\{X[i], Y[i]\}$. The *permuted-pLCP* result is readily obtained from the computation. Without loss of generality, suppose that both h and j exist and $2 < i \leq n$, so we have either $R[h] = R[i] - 1$ or $R[j] = R[i] - 1$ as the neighboring p-suffix. When $x =$ `plcp`(h, i, T) and $y =$ `plcp`(j, i, T)

Algorithm 3-7. Improved *pLCP* and *permuted-pLCP* computations

```
1  int[] compute_pLCP(char T[n], int pSA[n], int d) {
2     int pLCP[n], M[n], prevT[n]=prev(T), i
3     M[pSA[1]] = -1
4     for(i = 2 to n)
5        M[pSA[i]] = pSA[i-1]
6     M = construct(M, null, prevT, prevT)
7     switch(d){
8        case PERMUTED_PLCP: return M
9        case PLCP:
10          for(i = 1 to n)
11             pLCP[i] = M[pSA[i]]
12    }return pLCP
13 }
```

then $\max\{x, y\}$ distinguishes which p-suffix at h or j is closer to i, identifying the nearest neighbor and in turn, $pLCP[R[i]]$, which by Definition 3.4.1 is the value of *permuted-pLCP*[i]. Since it is the case that $X[i] = x$ and $Y[i] = y$, then it follows that *permuted-pLCP*$[i] = \max\{X[i], Y[i]\}$. Thus, maintaining the natural ordering of the elements yields the *permuted-pLCP*. By Definition 3.4.1, a simple and clear linear time reordering of the elements will transform the *permuted-pLCP* to the *pLCP*. Since the parameters $before_<$ and $after_<$ are clearly computed in $O(n)$ steps via Algorithm 3-5, the routine `construct` executes in $O(n)$ time given a p-suffix array via Theorem 3.3.3, and the reordering of elements is clearly linear, the theorem holds. □

For discussion purposes, Algorithm 3-6 uses a rank array R to index with the arrays $before_<$ and $after_<$ to determine the neighboring p-suffix. Further, notice that from Theorem 3.4.2, exactly one of the elements in either $before_<$ or $after_<$ is guaranteed to be the neighboring p-suffix. In practice, we do not need to indirectly go through the preprocessed arrays to find the neighboring p-suffixes since this can be found trivially with a p-suffix array. These observations are incorporated into the improved solution shown in Algorithm 3-7. The improved solution also eliminates the need for the rank array R and similar to Algorithm 3-6, permits the computation of the *permuted-pLCP* array – a direct result of the `construct` method. For further improved space consumption, the implementation of Algorithm 3-7 may incorporate a variation of the *LCP* indexing contributions of [73]. Even though the *pLCP* and *permuted-pLCP* data structures are defined differently from their traditional counterparts, they are still arrays of integers. We acknowledge the possibility that these

data structures may benefit from the other LCP contributions in [59, 80, 84, 41]. In this research, we are strictly concerned with the relationship between the data structures. It is apparent from Algorithm 3-7 that the $pLCP$ and *permuted-pLCP* data structures are very closely related to the $pLPF$ array in terms of the problem, restrictions, and construction.

3.4.3 $pLneF$, $pLrF$, and pLF

Parameterized Longest Not-Equal Factor

Consider the n-length p-string $T = AAAwBAAy\$$. In the case that we want to find the longest p-match or duplication within the p-string, we can view the problem naturally as a longest factor problem. Suppose that we wish to find the longest factor in common with the p-suffix $\texttt{prev}(T[6...n]) = AA0\$$, not including the p-suffix itself. Exhaustively, we find that the length of the longest factor is 3, since it is the case that the longest factor is at index 2 and $\texttt{prev}(T[2...n]) = AA0BAA0\$ =_3 AA0\$ = \texttt{prev}(T[6...n])$. We define this particular problem as the parameterized longest not-equal factor ($pLneF$).

Definition 3.4.3 Parameterized longest not-equal factor ($pLneF$): *For an n-length p-string T, the pLneF is defined for each index $1 \leq i \leq n$ such that $pLneF[i] = \max(\{0\} \cup \{k \mid \texttt{prev}([i...n]) =_k \texttt{prev}(T[j...n]), 1 \leq j \leq n, i \neq j\}) = \max\{\texttt{plcp}(i, neq_<[i], T), \texttt{plcp}(i, neq_>[i], T)\}$.*

Similar to the other variants in this chapter, the $pLneF$ data structure has much in common with the $pLPF$ array and may be computed with the same construction algorithm. A general example is given in Table 3.4. As shown in Definition 3.4.3, the $pLneF$ data structure is defined in terms of the preprocessed arrays $neq_<$ and $neq_>$, which for each position i in the p-string, respectively store the lexicographically closest p-suffix preceding and succeeding i. These preprocessed arrays handle all of the restrictions of the $pLneF$ problem in a similar fashion as the $pLPF$. Lemma 3.4.4 proves that the $pLneF$ array is constructed in linear time with the same construction algorithm used for the $pLPF$ array. Algorithm 3-8 displays this computation.

Lemma 3.4.4 *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, and pSA_T, the parameterized suffix array for T, the algorithm* `construct` *computes the pLneF array in $O(n)$ time.*

Algorithm 3-8. *pLneF* computation

```
1 int[] compute_pLneF(char T[n], int pSA[n]) {
2    return construct(preprocess(pSA,n,NEQ<),preprocess(pSA,n,NEQ>),
3                                                          prev(T),prev(T))
4 }
```

Proof It is clear that the *pLneF* problem is a type of longest factor problem. Let $prevT_1 = prevT_2 = prevT$. Suppose that $pLneF[a] = u > 0$ is known and $pLneF[b]$ is unknown for any $a \geq 1$ with $b = a + 1 \leq n$, then $pLneF[a] = \max(\{c, d \mid \texttt{prev}(T[a...n]) =_c \texttt{prev}(T[neq_<[a]...n]), \texttt{prev}(T[a...n]) =_d \texttt{prev}(T[neq_>[a]...n])\}) = u$. Consequently, $pLneF[b] = pLneF[a+1] = \max(\{c, d \mid \texttt{prev}(T[a+1...n]) =_c \texttt{prev}(T[neq_<[a+1]...n]), \texttt{prev}(T[a+1...n]) =_d \texttt{prev}(T[neq_>[a+1]...n])\}) \geq u-1 \Rightarrow pLneF[a+1] \geq pLneF[a]-1 \Rightarrow pLneF[b] \geq pLneF[b-1]-1$. Hence, the *pLneF* problem has the *extension property* in Definition 3.3.4. Further, Definition 3.4.3 defines the *pLneF* problem restrictions in terms of the preprocessed arrays $neq_<$ and $neq_>$, which are clearly computed in $O(n)$ time from Algorithm 3-5. Thus, it follows from Proposition 3.3.5 that the `construct` routine (Algorithm 3-3) on $prevT_1$ and $prevT_2$, which executes in $O(n)$ time via Theorem 3.3.3, also computes the *pLneF* data structure in $O(n)$ time. □

Parameterized Longest Reverse Factor

Perhaps the longest factor computation is to be considered between a p-suffix and some reversed p-suffix in the p-string T. For example, consider the n-length p-string $T = aabayzyy\$_1$ and in particular, the p-suffix at $i = 1$. The longest factor between $\texttt{prev}(T[1...n])$ and a p-suffix in the set $\{\texttt{prev}(T[1...n-1]^R\$_2), \texttt{prev}(T[1...n-2]^R\$_2), ..., \texttt{prev}(T[1]\$_2)\}$ is $\texttt{prev}(T[1...n-1]^R\$_2) = 01020021\$_2 =_8 01020021\$_1 = \texttt{prev}(T[1...n])$. We define this longest factor variant as the parameterized longest reverse factor (*pLrF*) problem.

Definition 3.4.5 Parameterized longest reverse factor (*pLrF*): *For an n-length p-string T, let $Q_1 = T[1...n-1]\$_1$, $Q_2 = T[1...n-1]^R\$_2$, and $Q = Q_1 \circ Q_2$. The pLrF is defined for each index $1 \leq i \leq n$ such that $pLrF[i] = \max(\{0\} \cup \{k \mid \texttt{prev}(T[i...n]) =_k \texttt{prev}(T[1...j]^R), 1 \leq j < n\}) = \max\{\texttt{plcp}(i, rev_<[i]+n, Q), \texttt{plcp}(i, rev_>[i]+n, Q)\} = \max\{\texttt{plcp}(Q_1[i...n], Q_2[rev_<[i]...n]), \texttt{plcp}(Q_1[i...n], Q_2[rev_>[i]...n])\}$.*

This problem compares with the text reverse data structures in [38, 37], but is different since we do not require the match to be overlapping or non-overlapping. A general example is provided in Table 3.4. The $pLrF$ problem is similar to the $pLPF$ problem because it may be defined through preprocessed arrays. These preprocessed arrays $rev_<$ and $rev_>$ provide for each forward p-suffix at i in the p-string T, the index of the reverse p-suffix preceding and succeeding i in pSA_Q. Algorithm 3-5 constructs the arrays $rev_<$ and $rev_>$ by exploiting the lexicographical closeness between the forward p-suffixes in Q_1 and the reverse p-suffixes in Q_2, which are both contained in pSA_Q. Notice that from Definition 3.4.5, the $pLrF$ problem is described in terms of preprocessed arrays in multiple ways. First, the $pLrF$ problem is defined by taking the `plcp` between the questioned p-suffix at i and the p-suffixes at $(rev_<[i]+n)$ and $(rev_>[i]+n)$ in Q. Also, the problem is defined by the `plcp` between the questioned p-suffix at i and the p-suffixes at $rev_<[i]$ and $rev_>[i]$ in Q_1 and Q_2, respectively. These definitions are the same since any `plcp` computation on Q or the pair Q_1 and Q_2 is identical because $Q[n] = \$_1 \neq Q[y]\ \forall\ y, n+1 \leq y \leq 2n$ and hence, it is guaranteed that any `plcp` result, say r, is such that $0 \leq r < n$. The advantage of considering the definition of the $pLrF$ array in terms of $rev_<$ and $rev_>$ with the pair of p-strings Q_1 and Q_2 is the direct setup of the problem to make use of the `construct` framework, also used to construct the $pLPF$ array. Lemma 3.4.6 proves that Algorithm 3-9 constructs the $pLrF$ data structure in linear time.

Lemma 3.4.6 *Given an n-length p-string T, $Q_1 = T[1...n-1]\$_1$, $Q_2 = T[1...n-1]^R\$_2$, $Q = Q_1 \circ Q_2$, $prevT_1 = \texttt{prev}(Q_1)$, $prevT_2 = \texttt{prev}(Q_2)$, and pSA_Q, the parameterized suffix array for Q, the algorithm* `construct` *computes the $pLrF$ array in $O(n)$ time.*

Proof This proof is similar to that of Lemma 3.4.4. The $pLrF$ problem is clearly a longest factor problem. Algorithm 3-5 constructs the arrays $rev_<$ and $rev_>$ in $O(n)$ time to store the index of the lexicographically closest p-suffix of Q_2 preceding and succeeding a p-suffix of Q_1 within pSA_Q. These preprocessed arrays are the only restrictions of the $pLrF$ problem in Definition 3.4.5 and hence, the $pLrF$ problem is a longest factor problem with pre-satisfied restrictions. Suppose that $pLrF[a] = u > 0$ is known and $pLrF[b]$ is unknown for any $a \geq 1$ with $b = a+1 \leq n$, then $pLrF[a] = \max(\{c, d \mid \texttt{prev}(Q_1[a...n]) =_c \texttt{prev}(Q_2[rev_<[a]...n]), \texttt{prev}(Q_1[a...n]) =_d \texttt{prev}(Q_2[rev_>[a]...n])\}) = u$. Consequently, $pLrF[b] = pLrF[a+1] = \max(\{c, d \mid \texttt{prev}(Q_1[a+1...n]) =_c \texttt{prev}(Q_2[rev_<[a+1]...n]), \texttt{prev}(Q_1[a+1...n]) =_d \texttt{prev}(Q_2[rev_>[a+1]...n])\}) \geq u-1 \Rightarrow pLrF[a+1] \geq pLrF[a]-1 \Rightarrow pLrF[b] \geq$

Algorithm 3-9. *pLrF* computation

```
1 int[] compute_pLrF(char T[n], int pSA_Q[|Q|]) {
2    return construct(preprocess(pSA_Q,n,REV_<),preprocess(pSA_Q,n,REV_>),
3                                                          prev(Q_1),prev(Q_2))
4 }
```

$pLrF[b-1]-1$. So, the $pLrF$ problem has the *extension property* in Definition 3.3.4. Therefore, it follows from Proposition 3.3.5 and Theorem 3.3.3 that the `construct` routine (Algorithm 3-3) uses $O(n)$ operations to compute the $pLrF$ data structure. □

Parameterized Longest Factor

Consider an example n-length p-string $T = AabcBCxyzA\$$. From the surface, we may find that the longest factor is between the p-suffix at $i = 2$ and the p-suffix at $j = 7$, i.e. $\texttt{prev}(T[i...n]) = 000BC000A\$ =_3 000A\$ = \texttt{prev}(T[j...n])$. If we analyze further, we can find a longer factor since $\texttt{prev}(T[1...10]\$_1) = A000BC000A\$_1 =_4 A000CB000A\$_2 = \texttt{prev}(T[1...10]^R\$_2)$. In the case where we are trying to find the longest factor in a p-string, whether that be in the forward or reverse direction, it is beneficial to observe both the $pLneF$ and $pLrF$ arrays. We define this problem as the parameterized longest factor (pLF) problem.

Definition 3.4.7 Parameterized longest factor (pLF): *For an n-length p-string T, the pLF is defined for each index $1 \leq i \leq n$ such that* $pLF[i] = \max(\{0\} \cup \{k \mid \texttt{prev}(T[i...n]) =_k \texttt{prev}(T[j...n]), 1 \leq j \leq n, i \neq j\} \cup \{k \mid \texttt{prev}(T[i...n]) =_k \texttt{prev}(T[1...j]^R), 1 \leq j < n\}) = \max\{pLneF[i], pLrF[i]\}$.

From the definition, the pLF problem is indirectly related to the $pLPF$ problem. This is due to the fact that the pLF data structure is broken down into a simple computation between elements of the $pLneF$ and $pLrF$ arrays, which are two longest factor arrays that we recently computed with the same construction algorithm used for the $pLPF$ array. See Table 3.4 for a general example. Algorithm 3-10 constructs the pLF array and the computation is formalized in Lemma 3.4.8.

Lemma 3.4.8 *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, pSA_T, $Q_1 = T[1...n-1]\$_1$, $Q_2 = T[1...n-1]^R\$_2$, $Q = Q_1 \circ Q_2$, $prevT_1 = \texttt{prev}(Q_1)$, $prevT_2 = \texttt{prev}(Q_2)$, and pSA_Q, the algorithm* `construct` *computes the pLF array in $O(n)$ time.*

Algorithm 3-10. *pLF* computation

```
1 int[] compute_pLF(char T[n], int pSA_T[n], int pSA_Q[|Q|]) {
2     int pLF[n], pLneF[n], pLrF[n], i
3     pLneF = compute_pLneF(T, pSA_T)
4     pLrF = compute_pLrF(T, pSA_Q)
5     for(i = 1 to n)
6         pLF[i] = max{pLneF[i], pLrF[i]}
7     return pLF
8 }
```

Proof The correctness and running time of the algorithm follow directly from Definition 3.4.7, Lemma 3.4.4, and Lemma 3.4.6. □

We summarize the foregoing results in the following theorem:

Theorem 3.4.9 *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, pSA_T, $Q_1 = T[1...n-1]\$_1$, $Q_2 = T[1...n-1]^R\$_2$, $Q = Q_1 \circ Q_2$, $prevT_1 = \texttt{prev}(Q_1)$, $prevT_2 = \texttt{prev}(Q_2)$, and pSA_Q, the pLneF, pLrF, and pLF data structures are each constructed in $O(n)$ time.*

Proof The proof follows from Lemma 3.4.4, Lemma 3.4.6, and Lemma 3.4.8. □

3.4.4 Parameterized Border Array

The *p-border* array [56] is a data structure that stores the length of the maximum border between a prefix of a p-string T and a proper p-suffix of the prefix (see Definition 2.1.18). The challenge of the problem, not involved in the traditional *border* array [87], is that all prefixes and suffixes are under the `prev` encoding.

In the closing comments of the section discussing our longest factor taxonomy, we mention that the *pLPF* construction may also be used to solve other p-matching oriented problems by setting *extend* = **false**. We view the construction of the *p-border* array in this way – as a special p-matching problem with some postprocessing. By independently computing the longest factor (maximum p-match length) between p-suffixes 1 and 2, between p-suffixes 1 and 3, between p-suffixes 1 and 4, etc., the result will be non-extendable p-matches. These particular p-matches happen to be lengths of longest proper p-suffixes with a prefix of T – select elements in the *p-border* array. The other *p-border* elements are obtained by removing the rightmost symbols from the end of the non-extendable p-matches, since removing rightmost

Algorithm 3-11. *p-border* array computation

```
1  int[] compute_p-border(char T[n], int pSA[n]) {
2     int fixed[n]={-1,1,...,1}, b[n]={0,0,...,0}, b2[n] i, c
3     extend=false
4     b2 = construct(fixed,null,prev(T),prev(T))
5     for(i = 2 to n) {
6        c = b2[i]
7        while(c > 0 ∧ c > b[i+c-1]) b[i+c-1]=c--
8     }extend=true
9     return b
10 }
```

symbols of a `prev` encoding does not modify previous parameter distances in the encoding. Since *p-border*$[1] = 0$ by Definition 2.1.18, consider initially $i = 2$. Let $c > 1$ be the length of the non-extendable p-match between the p-suffixes 1 and i. Then, *p-border*$[i + c - 1] = c$ is the current longest p-match and hence, an element in *p-border*. Also, while $c = c - 1 > 0$, we have *p-border*$[i + c - 1] = c$. After this initial match, we can consider successive non-extendable p-matches beginning with $i = i + 1$. These successive p-matches cannot replace the elements already populated in *p-border* because these previously populated elements are from an earlier, longer p-match. We obtain these non-extendable p-matches from the $pLPF$ construction and use the aforementioned observations to construct the *p-border* array. We give the construction algorithm of the *p-border* array in Algorithm 3-11 and prove its correctness and complexity in Theorem 3.4.10.

Theorem 3.4.10 *Given an n-length p-string T, $prevT =$* `prev`(T)*, the* `prev` *encoding of T, and pSA_T, the parameterized suffix array for T, the algorithm* `compute_p-border` *computes the p-border array in $O(n)$ expected time.*

Proof Let *extend* = **false**, $fixed[1...n] = \{-1, 1, ..., 1\}$, and *p-border*$[1...n] = \{0, 0, ..., 0\}$. By following Algorithm 3-3, we know that $b2 =$ `construct`$(fixed, \textbf{null}, prevT, prevT)$ yields $b2 = \{b2[1] = 0, b2[2] =$ `plcp`$(T[1...n], T[2...n]), ..., b2[n] =$ `plcp`$(T[1...n], T[n])\}$, in which each $b2[i]$ corresponds to the length of the longest factors: 0, `prev`$(T[1...n]) =_{b2[2]}$ `prev`$(T[2...n])$, ..., `prev`$(T[1...n]) =_{b2[n]}$ `prev`$(T[n])$. (Note that *extend* = **false** only changes the *way* in which previous p-matches are used to assist in future p-matching and since $fixed[2...n]$ is always a fixed p-suffix, we need to always restart p-matches from the beginning.) Since *p-border*$[1] = 0$ by definition, consider the value $b2[i] = \mathcal{J}$ (obviously $\mathcal{J} < n$) for $1 < i \leq n$ beginning with $i = 2$. Assume that $\mathcal{J} > 1$ without loss of generality. This signifies that

$\texttt{prev}(T[1...n]) =_{\mathcal{J}} \texttt{prev}(T[i...n])$, where $\mathcal{J}$ is the length of the longest factor and thus, $\texttt{prev}(T[1...n])[\mathcal{J}+1] \neq \texttt{prev}(T[i...n])[\mathcal{J}+1]$ is the case. In particular, this longest factor implies that the p-match is not extendable to the left (since *p-border*$[1] = 0$ and *p-border*$[i = 2]$ is the first element to consider a p-match) and clearly not extendable to the right (i.e. the longest factor). Hence, the value $b2[i] = \mathcal{J}$ generated from `construct` is actually the length of the longest proper p-suffix (since $1 < i \leq n$) that matches a prefix ending at $(i + \mathcal{J} - 1)$, which is the element *p-border*$[i + \mathcal{J} - 1] = b2[i] = \mathcal{J}$ from Definition 2.1.18. Further, it is also the case that this $b2[i]$ result tells us about the other longest proper p-suffixes that match a prefix of T, namely *p-border*$[i + \mathcal{J} - 2] = \mathcal{J} - 1$ (since $\texttt{prev}(T[1...n]) =_{\mathcal{J}-1} \texttt{prev}(T[i...n])$), *p-border*$[i + \mathcal{J} - 3] = \mathcal{J} - 2$ (since $\texttt{prev}(T[1...n]) =_{\mathcal{J}-2} \texttt{prev}(T[i...n])$), ..., *p-border*$[i + \mathcal{J} - k - 1] = \mathcal{J} - k$ (since $\texttt{prev}(T[1...n]) =_{\mathcal{J}-k} \texttt{prev}(T[i...n])$) for $(\mathcal{J} - k) > 0$ and non-negative k. Since this p-match between the p-suffixes at 1 and $i = 2$ cannot be extended to the left or right, it is guaranteed that $b2[i = 2] = \mathcal{J}$ will imply the elements *p-border*$[i = 2]$ through *p-border*$[i + \mathcal{J} - 1 = 1 + \mathcal{J}]$ since removing rightmost p-matched symbols will not change any encodings earlier in the p-suffix by Definition 2.1.4 and hence, will not be subject to the p-suffix intricacies formalized in Lemma 2.1.10. It is not possible for the elements at $b2[i + 1...i + \mathcal{J} - 1]$ and even $b2[i + 1...n]$ to provide longer p-matches for the elements *p-border*$[1...i + \mathcal{J} - 1]$ since the earlier p-matches that populated these elements are clearly longer because we considered earlier, longer p-suffixes in T. However, the elements $b2[i + 1...i + \mathcal{J} - 1]$ may still provide non-populated *p-border* elements for entries at some j, $j > i + \mathcal{J} - 1$. Consider now, $i = 3, ..., n$, $b2[i] = \mathcal{K}$, and let $\mathcal{K}_2 = \mathcal{K}$. While $\mathcal{K}_2 >$ *p-border*$[i + \mathcal{K}_2 - 1]$ and $\mathcal{K}_2 = \mathcal{K}_2 - 1$, there are new longer p-matches identified. If these p-matches could be extended to the left, it would have been previously identified in earlier $b2[h]$ for $h < i$ and already recorded in *p-border*. Now considering the current $b2[i]$, we know that $\mathcal{K}$ gives the length of the longest factor between the p-suffixes 1 and i and thus, cannot be extended to the right. These facts clearly indicate that this new longest factor gives the elements *p-border*$[i + \mathcal{K} - 1] = \mathcal{K}$, *p-border*$[i + \mathcal{K} - 2] = \mathcal{K} - 1$, etc. while *p-border*$[i + \mathcal{K} - k - 1] < \mathcal{K} - k$ for non-negative k. Since successive $b2[i]$ give the lengths of the longest factors between the p-suffixes 1 and i, it follows from the previous discussion that the `compute_p-border` construction generates the *p-border* array.

In terms of time, the notion that *extend* = **false** forces the `construct` algorithm to require B steps to construct the array $b2$ where $B = b2[1] + b2[2] + ... + b2[n] = 0 + \texttt{plcp}(T[1...n], T[2...n]) + ... + \texttt{plcp}(T[1...n], T[n])$. Since Σ and Π are finite, each $\sigma \in \Sigma$ and

$\pi \in \Pi$ are reused within a general p-string and the `prev` encoding is composed of various distances. The first p-suffix `prev`$(T[1...n])$, in particular, is composed of each unique distance. The successive p-suffixes that are compared with `prev`$(T[1...n])$ have numerous parameter distances replaced with the distance 0, which decreases the likeliness of long individual $b2[i]$. The same is true for the occurrence of infrequent constant sequences in the first p-suffix. Thus, it is expected that $B \in O(n)$. Further, the use of $b2$ to compute *p-border* requires $O(n)$ time since each element of *p-border* is populated at most once. Therefore, the algorithm constructs *p-border* in $O(n)$ expected time. □

Table 3.4: Arrays for p-string $T = AAAwBxyyAAAzwwB\$$ (* denotes newly proposed; $^+$ denotes new construction algorithm)

i	$pLneF[i]$ *	$pLrF[i]$ *	$pLF[i]$ *	*permuted-pLCP* *	*p-border* $^+$
1	4	4	4	4	0
2	3	3	3	3	1
3	2	2	2	2	2
4	2	3	3	2	0
5	1	2	2	1	0
6	3	2	3	0	0
7	2	2	2	1	0
8	1	5	5	1	0
9	4	4	4	2	1
10	3	3	3	1	2
11	2	2	2	0	3
12	3	2	3	3	4
13	2	2	2	2	0
14	2	2	2	1	0
15	1	1	1	0	0
16	0	0	0	0	0

3.5 Variations on a Theme - Traditional Strings

Recall that we have constructed the following p-string data structures using the same `construct` algorithm: *pLPF*, *pLCP*, *permuted-pLCP*, *pLneF*, *pLrF*, *pLF*, and the *p-border* array. In this section, we construct these same data structures for traditional strings. For concision, we exploit a special relationship between p-strings and traditional strings to address traditional string data structures with the *same* algorithms used for their p-string counterparts. This further adds to the power and utility of the `construct` algorithm. Lemma 3.5.1 formalizes the special relationship between p-strings and traditional strings.

Lemma 3.5.1 *Given a p-string alphabet Σ and Π as the set of constant and parameter symbols respectively, let $\Sigma = \Sigma \cup \Pi$ and afterwards, $\Pi = \emptyset$. For an n-length p-string T, the p-suffixes of T are traditional suffixes.*

Proof Since $T[i] \in \Sigma \ \forall \ i, 1 \leq i < n$ and $T[n] \in \{\$\}$, then by Definition 2.1.1 we have $T \in (\Sigma \cup \Pi)^*\$$, which classifies T as a valid p-string. In this special case, T consists of no such symbol $\pi \in \Pi$ so Lemma 2.1.10 identifies that $\texttt{prev}(T[i...n]) = \texttt{prev}(T)[i...n]$ and further $T = \texttt{prev}(T)$ by Definition 2.1.4, so $T[i...n] = \texttt{prev}(T[i...n]) = \texttt{prev}(T)[i...n] \ \forall \ i, \ 1 \leq i \leq n$. □

Lemma 3.5.1 formalizes the power of defining problems in terms of p-strings. By solving a p-string problem, we also solve the same problem for traditional strings since a string is a special case of a p-string. This generalization allows us to offer solutions to multiple problems with a single algorithm based on p-strings. Due to this generalization and the analogous definitions of the p-suffix array and the standard suffix array, our preprocessed arrays constructed in Algorithms 3-1 and 3-5 are also *still* valid for traditional strings.

We now, where necessary, redefine the data structures of this chapter using traditional strings and formalize the construction of the arrays. All of the following proofs use Lemma 3.5.1 to confirm that traditional strings are specific cases of p-strings and the corresponding p-string theorems still hold. These now trivial proofs are omitted. Let us begin by redefining the LPF data structure originally defined in [35] in terms of the $before_<$ and $before_>$ arrays.

Definition 3.5.2 ([35]) Longest previous factor (LPF): *In addition to Definition 2.1.15, the LPF is defined for each index $1 \leq i \leq n$ of an n-length traditional string W such that $LPF[i] = \max\{\texttt{lcp}(i, before_<, W), \texttt{lcp}(i, before_>, W)\}$.*

The LPF data structure is also computed in linear time with the `compute_pLPF` algorithm.

Theorem 3.5.3 *Given an n-length traditional string W, the* `compute_pLPF` *algorithm constructs the LPF array in $O(n)$ time.*

Similar to the generalization of the $pLPF$ problem to solve the LPF problem, it is also the case that the *permuted-LCP* array in Definition 2.1.14 and the traditional LCP array, analogous to Definition 2.1.12, are also constructed with the generalized p-string solution.

Theorem 3.5.4 *Given an n-length traditional string W, the* `compute_pLCP` *algorithm constructs the LCP and permuted-LCP arrays in $O(n)$ time.*

Further, we can define our newly proposed p-string data structures $pLneF$, $pLrF$, and pLF for traditional strings.

Definition 3.5.5 Longest not-equal factor ($LneF$): *For an n-length traditional string W, the LneF is defined for each index $1 \leq i \leq n$ such that $LneF[i] = \max(\{0\} \cup \{k \mid W[i...n] =_k W[j...n], 1 \leq j \leq n, i \neq j\}) = \max\{\texttt{lcp}(i, neq_<, W), \texttt{lcp}(i, neq_>, W)\}$.*

Definition 3.5.6 Longest reverse factor (LrF): *For an n-length traditional string W, let $Q_1 = W[1...n-1]\$_1$, $Q_2 = W[1...n-1]^R\$_2$, and $Q = Q_1 \circ Q_2$. The LrF is defined for each index $1 \leq i \leq n$ such that $LrF[i] = \max(\{0\} \cup \{k \mid W[i...n] =_k W[1...j]^R, 1 \leq j < n\}) = \max\{\texttt{lcp}(i, rev_<[i] + n, Q), \texttt{lcp}(i, rev_>[i] + n, Q)\} = \max\{\texttt{lcp}(Q_1[i...n], Q_2[rev_<[i]...n]), \texttt{lcp}(Q_1[i...n], Q_2[rev_>[i]...n])\}$.*

Definition 3.5.7 Longest factor (LF): *For an n-length traditional string W, the LF is defined for each index $1 \leq i \leq n$ such that $LF[i] = \max(\{0\} \cup \{k \mid W[i...n] =_k W[j...n], 1 \leq j \leq n, i \neq j\} \cup \{k \mid W[i...n] =_k W[1...j]^R, 1 \leq j < n\}) = \max(LneF[i], LrF[i])$.*

The aforementioned data structures are respectively constructed with the $pLneF$, $pLrF$, and pLF construction algorithms.

Theorem 3.5.8 *Given an n-length traditional string W, the algorithms* `compute_pLneF`*,* `compute_pLrF`*, and* `compute_pLF` *respectively construct the $LneF$, LrF, and LF data structures in $O(n)$ time.*

Finally, the *p-border* array [56] is a general case of the *border* array [87], so we can likewise prove that the *p-border* array construction theorem still holds for the traditional *border* array.

Theorem 3.5.9 *Given an n-length traditional string W, the* `compute_p-border` *algorithm constructs the border array in $O(n)$ expected time.*

Lastly, we note that during Theorem 3.4.10, the $b2$ array (see Algorithm 3-11 line 4) generated for the construction of *p-border* is another important string data structure for the n-length T. Since each $b2[i]$, for $2 \leq i \leq n$, has the length of the longest prefix common

between T and $T[i...n]$, then $b2[i] = PA[i]$, where PA is the prefix array (Definition 2.1.19). So by setting $b2[1] = n$, we have that the prefix array is constructed via the $pLPF$ framework under the same constraints as the *p-border*.

Theorem 3.5.10 *Given an n-length traditional string W, the* `compute_p-border` *algorithm constructs the prefix array (PA) in $O(n)$ expected time.*

3.6 Experiments

In this chapter, we develop theory that establishes a connection between the parameterized longest previous factor ($pLPF$) data structure and popular data structures, such as LCP and the *border* array. We also show the connection between $pLPF$ and other newly introduced data structures. Our core contribution is a single construction algorithm that can develop data structures related to $pLPF$. For ease of discussion in this chapter, we consider the most common case of p-strings – where the lexicographical relationship between p-suffixes is like that of traditional suffixes (Figure 1(a) of [19]). In our implementation of the algorithms, we handle all of the p-string scenarios/details in the work [19]. The implementations are written in C. We have executed our algorithms in the Cygwin environment running on a Dell Inspiron 570 desktop with 3.10 GHz clock speed and 8 GB RAM. This section discusses experimental results of our programs on various files from the Large Corpus (http://corpus.canterbury.ac.nz/descriptions/#large), the theoretical Fibonacci string, and strings from random distributions. Table 3.5 contains details that describe the nature of the data. We discuss the experimental results and compare them with our theoretical results. We then implement traditional algorithms that construct LCP and LPF and compare them with our newly introduced parameterized constructions. For each experiment, we do not consider the construction of the p-suffix array (pSA), the rank array (R), or `prev`(T) in the execution time because it is assumed that these data structures are readily available to a prior to construction of $pLPF$ variants. Other auxiliary data structures needed in the construction, such as those from Algorithm 3-5, are considered in the execution time. In terms of practical space, we consider the resident memory used by the process.

Let us first consider the Ecoli sequence from the Large Corpus. In our experiment, we constructed the $pLPF$ variants for prefixes of the text. For this text, we considered the entire alphabet to be parameters: $\Sigma = \emptyset$ and $\Pi = \{a, c, g, t\}$. In Figure 3.1, we display the

Table 3.5: Select data attributes

Attribute	Ecoli $n = 4638690$ $\|\Sigma\| = 0, \|\Pi\| = 4$	Bible $n = 3445275$ $\|\Sigma\| = 27, \|\Pi\| = 14$	Fibonacci $n = 200000$ $\|\Sigma\| = 0, \|\Pi\| = 2$	$\mathcal{N}(24, 12)$ $n = 1000000$ $\|\Sigma\| = 0$
permuted-pLCP$_{max}$	2815	487	121391	29
permuted-pLCP$_{\mu}$	19.3	13.3	52287.6	16.6
$pLCP_{max}$	2815	487	121391	29
$pLCP_{\mu}$	19.3	13.3	52287.6	16.6
p-border$_{max}$	15	14	121391	15
p-border$_{\mu}$	2.3	0.3	52287.6	5.4
$pLPF_{max}$	2815	487	121391	29
$pLPF_{\mu}$	19.3	13.3	52287.6	16.6
$pLneF_{max}$	2815	487	121391	29
$pLneF_{\mu}$	24.1	15.7	69681.4	17.3
$pLrF_{max}$	3027	15	196415	31
$pLrF_{\mu}$	22.7	4.0	97888.8	17.3
pLF_{max}	3027	487	196415	31
pLF_{μ}	26.2	15.7	97889.0	17.7

execution time for the construction of each $pLPF$ variant. Each data structure is created in linear time, which confirms our theoretical results. We notice that indeed, the pLF data structure requires more time to construct since it primarily requires the construction of both $pLrF$ and $pLneF$. For $pLCP$ and *permuted-pLCP*, we display results using the construction of Algorithm 3-7. For concision, we omit the construction of Algorithm 3-6 from the results since we observe that indeed Algorithm 3-7 is a practical improvement. We notice that the quickest algorithms construct the $pLCP$, *permuted-pLCP*, and *p-border* data structures. This is the case since these particular constructions (Algorithm 3-7 and Algorithm 3-11) do not require the preprocessing of Algorithm 3-5. That is, the light preprocessing is apparent in the overall execution time.

Next, we considered the Bible text from the Large Corpus. Since the Bible is composed of words, it is inappropriate to use individual symbols as parameters. So, the original Bible text was preprocessed and transformed into a text more suitable for p-matching and our program. First, only letters and spaces in the original text remained in the transformed text. All letters were forced to lowercase letters. Next, a unigram was constructed and each word appearing in the new text with a frequency $f \geq 7500$ was replaced with a unique uppercase symbol. (Thus, the size of the transformed Bible is slightly smaller than the original.) For example, the word `in` was replaced with the letter `G` and the word `the` was replaced with the letter `L`. These 14 frequent words were considered parameters. Since the parameter words are now replaced with unique symbols not used by constants, there is no real need to adjust

the remaining words since the remaining symbols must match anyway to detect a constant. So, we have constructed a new p-match problem where frequent words may be substituted. We constructed the $pLPF$ variants on prefixes of the processed Bible sequence. Figure 3.2 shows the results, which are nearly identical to the linear time results of the previous Ecoli experiment. The *p-border* construction on the Bible sequence is so quick because of the fact that the first p-suffix consists of the introductory sequence `G_L_beginning_`, which does not p-match with any longer sequences in the text. Overall, we also notice the same results for random strings from Normal and Uniform discretized distributions. Figure 3.3 shows the construction times for the $pLPF$ variants on random input strings with symbols from the Normal distribution. Very similar results were obtained for sequences with symbols from the Uniform distribution.

The qth Fibonacci sequence (or Fibostring) [87] is denoted by f_q and satisfies the recurrence $f_0 = b$, $f_1 = a$, and $f_q = f_{q-1}f_{q-2}$ for $q \geq 2$. The recurrence relation makes the resulting string naturally repetitive. We see from Table 3.5 that the maximum and mean values of each data structure is significant when compared to the text length. This would classify a Fibonacci sequence as a worst case string for a p-matching application. In terms of time, Figure 3.4 shows the $pLPF$ variants are constructed very quickly. For the prefixes considered, we see that even some constructions do not add noticeable time to the plots when increasing prefix length. We note from Table 3.5 and the plots that the Fibonacci sequence is shorter than the other sequences ($n = 200000$). Considering sequences at around the same prefix size, there is typically more time required for the construction of $pLPF$ variants on the Fibonacci sequence. This very relationship can be viewed more clearly when considering the Ecoli and Bible results, in which the slopes of the linear time results of Ecoli are greater than that of the Bible. This is due to the nature of the resulting arrays of the $pLPF$ variants, detailed in Table 3.5. The resident memory required by the data structures on the Fibonacci sequence is displayed in Figure 3.5. Similar to the time of pLF, the construction of pLF also requires more space than the other constructions. The space required for the other experiments is similar.

Throughout this research, we are focused on relating the constructions of data structures to the $pLPF$ construction. This includes traditional data structures with $|\Pi| = 0$. The LPF and LCP are popular traditional data structures that our algorithms can construct. For purposes of comparison, we have implemented the direct LPF construction presented in [35]. This particular algorithm was chosen since it influenced this work. We also imple-

mented the LCP construction algorithm of [61]. We compared these constructions with our parameterized constructions on the Fibonacci sequence in terms of time and memory required. Figure 3.6 displays the execution time. We see that the parameterized constructions do not add significantly to the constructions of the traditional algorithms. That is, there is a very small cost for using a parameterized construction. As the prefix size increases, this cost becomes smaller. We observed that the parameterized constructions do require more memory, which is expected since p-string solutions also require additional data structures such as $\texttt{prev}(T)$.

First and foremost, this research considers the theoretical relationship between data structures and the linear time $pLPF$ construction. Our experimental results confirm the linear time construction of $pLPF$ variants. The fact that solutions to p-string data structures support constructions for traditional data structures is an important feature. To be viable in practice, a p-string solution needs to be efficient when considering the traditional counterpart. Even though our codes are not optimized, our experimental results still show that our construction algorithms are competitive with standard algorithms for constructing the popular data structures LCP and LPF. This gives practical significance to our algorithms as a foundation for solving numerous combinatorial problems for p-strings and standard strings.

3.7 Conclusions

In this chapter, we consider the $pLPF$ array and prove that its construction algorithm is strongly connected with other string data structures. The framework that is used to solve the $pLPF$ problem is also proven to construct the $pLCP$ array and the newly proposed *permuted-pLCP* data structure. We also define a multitude of variations of the $pLPF$ data structure that are computed with the same basic framework. We implement our algorithms and confirm our theoretical results on various texts. In terms of applications, our $pLPF$ construction framework is a viable option for computing the LCP, LPF, and *border* arrays, which are prominent data structures in efficient pattern matching. In this work, we are considering longest factor problems that yield arrays with the lengths of the longest factors. We can easily define parallel arrays to also point to the position of the longest factor to permit easy access to these factors. Direct applications of our introduced data structures may include pattern substitution, detecting duplication [12], LZ decomposition in text compres-

sion [100], studying periodicity in strings [70, 87], biological sequence compression [3, 44], and analysis of repetition structures in DNA sequences [47, 2]. Specifically, our pLF data structure may be used to identify how to best substitute a pattern or even determine if duplication is "hidden" by reversal or with parameterization. Moreover, the choice of the Π alphabet adds to the possibilities of string analysis, a step beyond traditional exact matching. Since we have defined our data structures for p-strings, we have the ability to answer traditional string problems and also address more sophisticated applications like analyzing parameterized duplication in source code, DNA, and RNA.

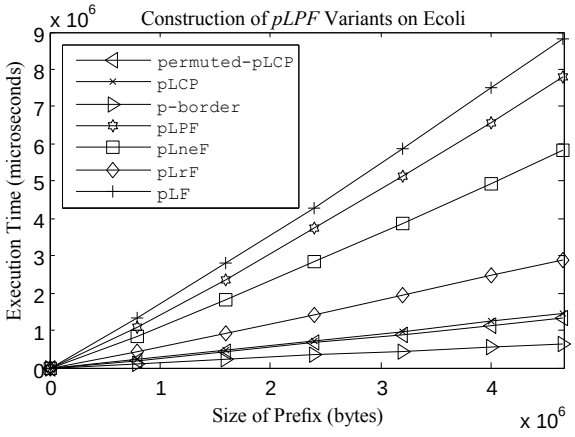

Figure 3.1: Time for construction of $pLPF$ variants on Ecoli sequence with $|\Sigma| = 0$ and $|\Pi| = 4$

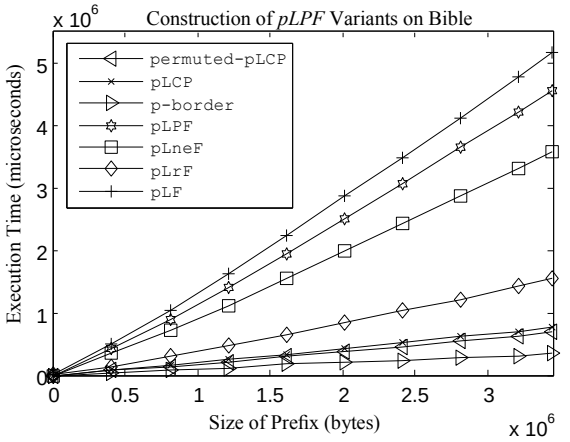

Figure 3.2: Time for construction of $pLPF$ variants on Bible with $|\Sigma| = 27$ and $|\Pi| = 14$

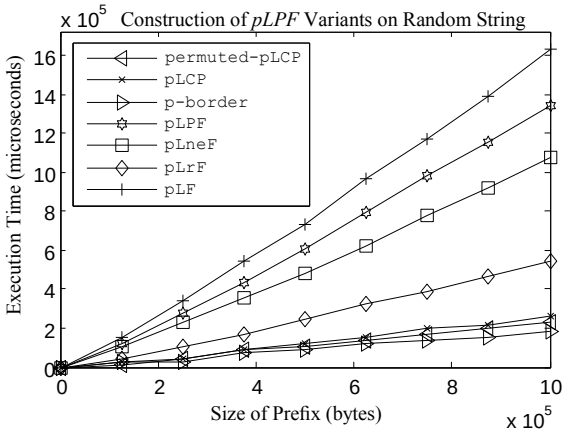

Figure 3.3: Time for construction of $pLPF$ variants on random sequence from $\mathcal{N}(24, 12)$, where $\mathcal{N}(a, b)$ is a discretized Normal distribution with mean a and variance b, and $|\Sigma| = 0$

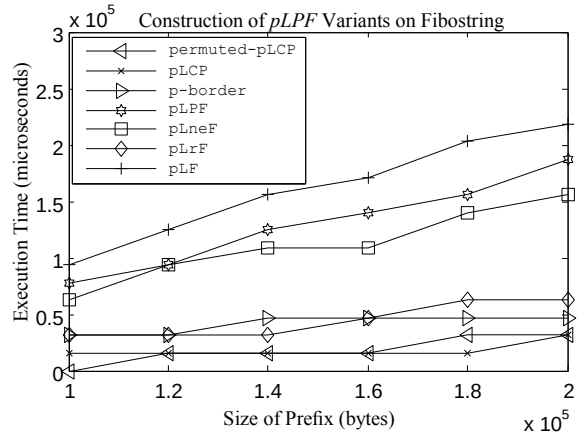

Figure 3.4: Time for construction of $pLPF$ variants on Fibonacci sequence with $|\Sigma| = 0$ and $|\Pi| = 2$

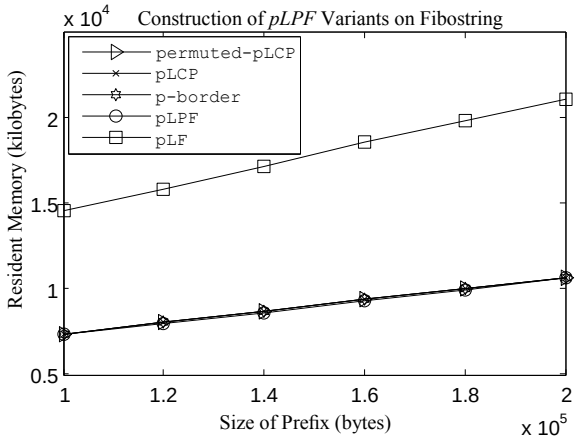

Figure 3.5: Space for construction of $pLPF$ variants on Fibonacci sequence with $|\Sigma| = 0$ and $|\Pi| = 2$

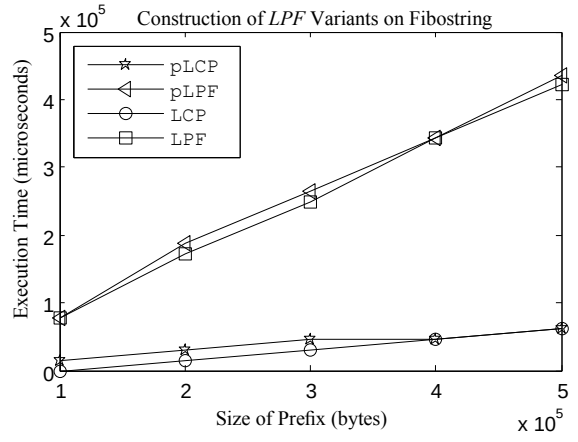

Figure 3.6: Comparison of traditional constructions ($|\Pi| = 0$) of LCP and LPF with the same data structures built using parameterized constructions

Chapter 4

The Structural Border Array

The work reported in this chapter yielded the following publications.

- Beal, R., Adjeroh, D.: Border array for structural strings. In: Arumugam, S., Smyth, W.F. (eds.), International Workshop on Combinatorial Algorithms (IWOCA) 2012. LNCS, vol. 7643, pp. 189-205. Springer, Heidelberg (2012)
- Beal, R., Adjeroh, D.: The structural border array. Journal of Discrete Algorithms, 23, 98-112 (2013)

4.1 Introduction

The *border* array is a fundamental data structure in string theory used for pattern matching, classifying strings, etc. [87]. A parameterized string (p-string), as identified by Baker [11], is a generalized string from the constant alphabet Σ and the parameter alphabet Π. The parameterized border array (*p-border*) is the traditional *border* array problem observed in terms of p-strings [56, 54, 55]. Similarly, *p-border* is also useful in parameterized pattern matching (p-matching), which is a type of pattern matching where constant symbols $\sigma \in \Sigma$ match and there exists a bijection between the parameter symbols $\pi \in \Pi$. Consider the example p-strings that represent program statements $z{=}y * f/{+}{+}$y; and $a{=}b * f/{+}{+}b$; over the alphabet sets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$. Here, a p-match exists because constant symbols $\sigma \in \Sigma$ match and parameter symbols $\pi \in \Pi$ properly align, namely in the first statement z, y, and f are consistently substituted by a, b, and f respectively in the second statement. The p-match problem offers a new way to address pattern matching in

significant applications regarding the identification of plagiarism in academia and industry [12] and also, detecting unauthorized use of source code [98].

A variation of the p-match problem is known as structural matching (s-matching) between structural strings (s-strings) [86]. The s-string adds the notion of complementary pairs of parameter symbols in some alphabet Γ. Detecting an s-match requires identifying a p-match and ensuring that the parameter complements are consistent. For instance, consider the alphabets $\Sigma = \emptyset$, $\Pi = \{A, U, C, G\}$, and $\Gamma = \{(A, U), (C, G)\}$ and consider the sequences $S = UAUAU$ and $T = GCGCG$. Notice that where parameters U and A exist in S, there exist substitutions of parameters G and C respectively in T. Also, notice that where the complements U and A exist in S, the complement symbols G and C align in T. These observations identify that S and T s-match. This type of matching is relevant for analyzing biological data such as RNA sequences or secondary structures, since the complementary base pairing can be analyzed using the s-match [86]. Currently, the s-match problem is handled via structural suffix trees (s-suffix trees) [86]. In many situations, the huge practical space of an s-suffix tree poses a significant problem, which led to the development of the structural suffix array (s-suffix array) [16]. In this work, we are motivated to introduce yet another significant data structure for the s-match problem: the structural border array.

Main Contributions: We introduce the structural border array (*s-border*) as defined for an n-length structural string (s-string) T. Initially, we provide constructions that execute in time $O(n^3)$ and $O(n^2)$ to build the *s-border* array. We establish theory to improve the result to $O(n)$ by proving particular properties of the *s-border* data structure. Using the same construction algorithm, we show how to modify the s-string alphabets to also construct both the parameterized border (*p-border*) and the traditional *border* array in linear time. Our solution to the *p-border* problem is a symbol-based approach different from the automaton-oriented solution presented in [56]. The following formalizes our main results.

Theorem 4.3.7. *Given an n-length s-string T, there is an algorithm that constructs the s-border array $\mathcal{B}_s$ in $O(n)$ time.*

Theorem 4.4.3. *Given an n-length s-string T, the algorithm* `construct_`$\mathcal{B}_s$ *constructs the p-border array $\mathcal{B}_p$ and the traditional border array $\mathcal{B}$ each in $O(n)$ time.*

4.2 Background

Baker [12] identifies three types of pattern matching: (1) exact matching, (2) parameterized matching (p-match), and (3) matching with modifications. The first p-match breakthroughs, namely, the `prev` encoding and the parameterized suffix tree (p-suffix tree), were introduced by Baker [11]. Additional improvements to the p-suffix tree are given in [65, 32, 68]. Like the traditional suffix tree [87, 47, 1], the p-suffix tree [11] implementation suffers from a large practical memory footprint. One p-matching solution to address the space problem is the parameterized suffix array (p-suffix array) in [53, 40]. An expected linear time p-suffix array construction is given in [20]. The work of [21] proves the existence of sub-quadratic and near-linear time worst case p-suffix array constructions. Other solutions that address the p-match problem without the space limitations of the p-suffix tree include the parameterized-KMP [6] and parameterized-BM [14], variants of traditional pattern matching approaches. These particular approaches use a variety of heuristics for shifting the matches to p-match efficiently. Further, the p-match problem is addressed via the Shift-OR mechanism in [42]. Idury et al. [56] studied a heuristic known as the `pfail` function to address the multiple p-match problem using an automaton. This `pfail` function is now known as the parameterized border array (*p-border*), analogous to the traditional *border* array [87], and has been studied in a variety of combinatorial problems in [54, 55]. Other p-match data structures are studied in [19]. A closely related variant of the p-match problem is the structural pattern matching (s-match) problem, introduced by Shibuya [86]. The s-match is used in [86] for RNA analysis by a structural suffix tree (s-suffix tree). An s-suffix tree is similar in nature to the p-suffix tree [11] and constructed in similar time. The practical space used by the s-suffix tree was the motivation to introduce a more lightweight data structure known as the structural suffix array (s-suffix array) [16]. In this work, we introduce the structural border array (*s-border*) for the s-match problem and provide a linear time construction. We show how to use our algorithm to also construct, in linear time, the *p-border* and the traditional *border* arrays.

4.3 Structural Border Array

The traditional *border* array as defined in Definition 2.1.17 for traditional strings compares prefixes of a string, say W, with proper suffixes of those prefixes. Working with the

individual symbols of the prefixes and suffixes of W is trivial because $W[j...n]$ is always a suffix of $W[1...n]$ for any j, $1 \leq j \leq n$. This trivial use of symbols is not the case with the parameterized border (*p-border*) of Definition 2.1.18. In the case of a p-string, the *p-border* identifies the maximum p-match between borders, in which these borders are under the `prev` encoding by Theorem 2.1.9. The challenge of working with the *p-border* is the dynamic nature of `prev` (see Lemma 2.1.10), which is fundamentally different from the suffixes in traditional *border* construction. As a result, the *way* in which symbols are handled in the traditional *border* construction is not correct for the *p-border*.

We define the *s-border* array using the encoding `sencode`, which is the encoding that identifies a structural match (s-match) by Theorem 2.2.6.

Definition 4.3.1 Structural border array (*s-border* or $\mathcal{B}_s$): *For an n-length s-string T, the s-border array is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}_s[1] = 0$ and otherwise* $\mathcal{B}_s[i] = \max(\{0\} \cup \{k \mid \texttt{sencode}(T[1...k]) = \texttt{sencode}(T[i-k+1...i]), k \geq 1 \wedge i-k+1 > 1\})$.

The substrings $T[1...k]$ and $T[i-k+1...i]$ in the definition are referred to as borders. When these borders are under the encoding `sencode`, they are known as *structural borders* or s-borders. Since *s-border* is defined on `sencode`, which from Definition 2.2.4 is a combination of symbols from the text T, `prev`(T), and `compl`(T), we encounter the same difficulties as *p-border*. The difference this time is that the *pair* of encodings `prev` and `compl` dynamically change depending on the locations of parameters in an s-suffix (see Lemma 2.2.8).

4.3.1 Naïve Algorithm

Without investigating properties of our defined *s-border* array, we can still compute the data structure in a naïve way as shown in Algorithm 4-1. For an n-length s-string T, the algorithm computes *s-border* in roughly $O(n^3)$ time, since the $O(n)$ `sencode` construction by Lemma 2.2.7 is nested within a **while** loop that is bounded by n iterations and this $O(n^2)$ computation is nested within a **for** loop with n iterations. This particular algorithm makes the case for just how difficult it can be to compute *s-border*.

To improve this algorithm, we introduce the s-match related functions Ψ and ψ in Algorithm 4-2. Note that the Ψ and ψ functions correctly implement s-matching by incorporating elements of Definition 2.2.2, Theorem 2.2.6, Definition 2.2.9, and Definition 2.1.2. Specifically, function $\psi(a, b, j)$ compares the symbols $T[a]$ and $T[b]$ as they occur in s-suffixes at

Algorithm 4-1. Naïve construction of $\mathcal{B}_s$

```
1  int[] construct_Bs_naive(char T[n]){
2      int i,k,m,Bs[n]={0,0,...,0}
3      for i=2 to n {
4          k=1, m=0
5          while(i–k+1>1) {
6              if(sencode(T[1...k])=sencode(T[i–k+1...i])) m=k
7              k++
8          }Bs[i]=m
9      }return Bs
10 }
```

symbol j, where **true** is returned when the symbols s-match. This comparison is accomplished in constant time.

Lemma 4.3.2 *Each call to ψ executes in $O(1)$ time.*

The $\Psi(i, j, k)$ function, which uses a sequence of constant time calls to ψ, returns the number of symbols m such that $\texttt{sencode}(T[i+k-1...i+k+m-2]) = \texttt{sencode}(T[j+k-1...j+k+m-2])$.

Lemma 4.3.3 *Each call to Ψ executes in $O(m)$ time, where m is the length of the current s-match.*

We emphasize the inability to obtain a correct *s-border* solution by trivially plugging an s-string into a *border* or *p-border* construction algorithm. To put this problem into perspective for an s-string T, let $U = \texttt{sencode}(T[1...n]) \circ \$_1 \circ \texttt{sencode}(T[2...n]) \circ \$_2 \circ \texttt{sencode}(T[3...n]) \circ \$_3 \circ ... \circ \$_{n-1} \circ \texttt{sencode}(T[n])$ where $\{\$_1, \$_2, ..., \$_{n-1}\}$ is the set of unique terminal symbols where $\$_i \notin \{\Sigma \cup \Pi \cup \{\$\}\}$. Notice that U contains each s-suffix and thus, Lemma 2.2.8 does not apply. Let $\mathcal{B} = \texttt{border}(U)$ compute the traditional *border* array $\mathcal{B}$ for text U. Since U clearly represents each s-suffix, the resulting array $\mathcal{B}$ contains the correct results for $\mathcal{B}_s$ within the multitude of elements in $\mathcal{B}$. However, the problems with this approach are (1) computing the *s-border* elements $\mathcal{B}_s[i]$ will require us to postprocess the resulting $\mathcal{B}$ to find the maximum border ending at symbol i in the original T and (2) the construction of $\mathcal{B}_s$ can do no better than the length of U, which is of length $O(n^2)$. Note that the *s-border* array only has n elements. From the previous example with running time $O(n^2)$ and the naïve $O(n^3)$ approach in Algorithm 4-1, we are motivated to further investigate properties of *s-border*. This leads to an improved algorithm for constructing the *s-border*.

Algorithm 4-2a. s-matching function Ψ

```
1  char T[n] /* given */
2  char prevT[n]=prev(T)
3  char complT[n]=compl(T)
4  char αT[n]=α(T)
5
6  int Ψ(int i,int j,int k){
7     int a,b,m=−1,q=k−1
8     do{
9        q++
10       a=i+q−1,b=j+q−1
11       m++
12    }while(ψ(a,b,q));
13    return m
14 }
15
```

Algorithm 4-2b. s-matching function ψ

```
boolean ψ(int a,int b,int j){
   boolean match=false
   if(1≤a≤n ∧ 1≤b≤n ∧ αT[a]=αT[b]) {
      if(αT[a]=SIGMA∧T[a]=T[b]) match=true
      else if(αT[a]=PI){ /* distances */
         if(prevT[a]=prevT[b]=0) match=true
         else if(prevT[a]<j ∧ prevT[b]<j ∧
              prevT[a]=prevT[b]) match=true
         else if(prevT[a]≥j∧prevT[b]≥j) match=true
         else if(complT[a]<j ∧ complT[b]<j ∧
              complT[a]=complT[b]) match=true
         else if(complT[a]≥j∧complT[b]≥j) match=true
      }
   }return match
}
```

4.3.2 Improved Algorithm

A key property used to construct the traditional *border* array $\mathcal{B}$ is the property that $\mathcal{B}[i+1] \leq \mathcal{B}[i]+1$. This property helps progress matches by oracling the previous array element and comparing the subsequent symbols. We prove that even though the s-suffixes change from Lemma 2.2.8, this property still holds when considering the *s-border* array, which is defined on prefixes of suffixes under `sencode`. That is, the *way* in which the `sencode` is defined, which is a combination of `prev` and `compl`, does not invalidate this traditional *border* property. In Figure 4.1, we illustrate that a prefix named $prefix$ of an s-suffix is such that $prefix = \texttt{sencode}(T)[1...|prefix|]$. More specifically, the way in which distances refer previously in the text allows us to treat $prefix$ as a *valid* encoding itself. This is exactly what is needed for the *s-border* construction. Such is not true for encodings like `forw` from Definition 2.1.7. For construction algorithms, we emphasize that one must always consider the impact of the encoding scheme (see [21, 19]).

Lemma 4.3.4 *Given an s-string T of length n, the individual s-border elements $\mathcal{B}_s[i]$ are such that $\mathcal{B}_s[i+1] \leq \mathcal{B}_s[i]+1 \ \forall\ i, 1 \leq i < n$.*

Proof Initially, $\mathcal{B}_s[1] = 0$ by Definition 4.3.1. Consider that $\mathcal{B}_s[i] = k$ for some $i, 1 < k < n$. Without loss of generality, assume that $k > 2$. Then, by Definition 4.3.1, $\texttt{sencode}(T[1...k]) = \texttt{sencode}(T[i-k+1...i])$ is the maximum s-border of $T[1...i]$. Consider $j = i+1$. What we

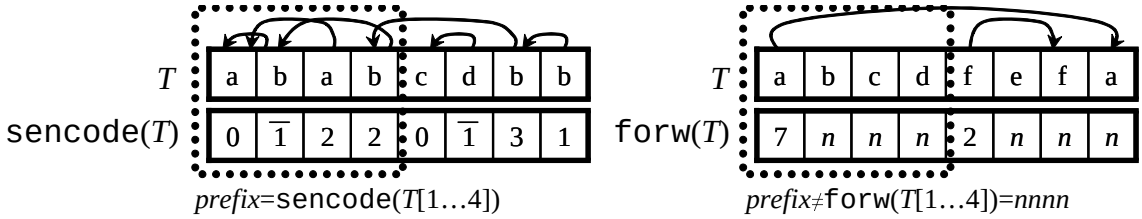

Figure 4.1: Displaying the intricacies of prefixes for s-string encoded suffixes using $\Sigma = \emptyset$, $\Pi = \{a, b, c, d, e, f\}$, and $\Gamma = \{(a, b), (c, d), (e, e), (f, f)\}$, where n is fixed: $n = 8$

know is that (1) already $\texttt{sencode}(T[1...k]) = \texttt{sencode}(T[i-k+1...i])$ and (2) `prev` (from Definition 2.1.4) and `compl` (from Definition 2.2.3) are defined such that the dynamically changing elements $T[i] \in \Pi$ of Lemma 2.2.8 are encoded to point to *previous* elements in the s-suffix, which means that appending elements to the s-suffix does not modify the encodings of the already existing s-suffix (see Figure 4.1). From (2) and Definition 4.3.1, there cannot exist any s-border of $T[1...j]$ longer than one symbol beyond the maximum s-border at i, that is, the s-match of length $\mathcal{B}_s[i]+1$. Let $a = \texttt{sencode}(T[1...1+k])[k+1]$ and $b = \texttt{sencode}(T[j-k...j])[k+1]$. It follows that if $a = b$, then also $\texttt{sencode}(T[1...1+k]) = \texttt{sencode}(T[j-k...j])$. Thus, $\mathcal{B}_s[j] = k+1$. If $a \neq b$, then it follows that $0 \leq \mathcal{B}_s[j] \leq \mathcal{B}_s[i] = k$. Therefore, $\mathcal{B}_s[j] \leq k+1$. □

The previous lemma gives us the ability to offer the improvement `construct_`$\mathcal{B}_s$`_improved` in Algorithm 4-3. This algorithm makes use of the s-match related functions in Algorithm 4-2. Using the Ψ and ψ functions, the technique behind Algorithm 4-3 is to start from the left of the s-string, find the longest s-matches, and populate the elements of $\mathcal{B}_s$. A heuristic is used to determine whether or not s-matching may yield new elements of $\mathcal{B}_s$. The following theorem formalizes the algorithm and its running time.

Theorem 4.3.5 *Given an n-length s-string T, the algorithm* `construct_`$\mathcal{B}_s$`_improved` *constructs the s-border array $\mathcal{B}_s$ in time $O(\max\{n, b\phi\})$, where b is the length of the longest s-border and ϕ is based on the s-string.*

Proof We first prove that the technique behind the algorithm is correct. Let $\mathcal{B}_s = \{0, 0, ..., 0\}$. Maintain a pointer to index h such that the *s-border* subarray $\mathcal{B}_s[1...h-1]$ is always complete. Initially, $h = 2$ since $\mathcal{B}_s[1] = 0$ by Definition 4.3.1. Next, we find the longest s-match of say m symbols between $T[1...n]$ and $T[j...n]$ where $j = 2$. Let $q = m$. By Lemma 4.3.4, we know that while $m > 0$, then we can assign $\mathcal{B}_s[j+m-1] = m$ and let $m = m-1$. The

previously populated elements are the longest s-borders for indices $j+q-1$, $j+q-2$, etc. because we have started at $j=2$ and no s-border can be longer for these elements. Thus, $\mathcal{B}_s[1...j+q-1]$ is now complete. Set $h=j+q$. We continue at $j=3$ considering the following cases.

- If $j=h$, we continue the same process to find $\mathcal{B}_s[j]$ and populate other $\mathcal{B}_s$ elements exactly like the initial part of the proof for $j=h$ because only $\mathcal{B}_s[1...h-1=j-1]$ is complete.

- When $j<h$ and an s-match exists between the symbols at $h-j+1$ in the s-suffixes at 1 and j, i.e. $\texttt{sencode}(T[1...n])[h-j+1]=\texttt{sencode}(T[j...n])[h-j+1]$, then the following s-match of m symbols can be conducted: $\texttt{sencode}(T[1...m])=\texttt{sencode}(T[j...j+m-1])$. (Otherwise, no longer s-border is possible.) Since $\mathcal{B}_s[1...j...h-1]$ is already complete with the longest s-borders, only when the m exceeds the complete section of $\mathcal{B}_s$ is there a newly introduced s-border, i.e. when $j+m-1\geq h$. So, any newly introduced s-border to the incomplete part of $\mathcal{B}_s$ must be maximum because in previous steps $j-1$, $j-2$, etc. we have considered, but not found, s-borders that could be only longer. Considering s-borders for future $j+1$, $j+2$, etc. will only introduce shorter s-borders than the current s-border of length m. Now that we have new maximum s-borders, populate only the new findings. It follows from Lemma 4.3.4 that we can let $q=m$ and assign the other known maximum s-borders, that is, while $m>0\ \wedge j+m-1\geq h$, assign $\mathcal{B}_s[j+m-1]=m$ and let $m=m-1$. Finally, set $h=j+q$ to signify that now $\mathcal{B}_s[1...h-1=j+q-1]$ is complete.

- Otherwise, no s-matching is necessary because it is not possible to introduce a longer s-border with the current s-suffixes considered.

The previous cases are considered for subsequent $j=4,5,...,n$. In each case, we are finding the longest s-matches between the s-suffixes at 1 and j with each $\mathcal{B}_s$ element populated at most once using the first appropriate value. Since j increases and we populate $\mathcal{B}_s$ using the earliest relevant s-match found, subsequent j will only produce smaller s-borders in `sencode` due to the fact that appending symbols in `prev` (from Definition 2.1.4) and `compl` (from Definition 2.2.3) does not modify the encodings of the already existing s-suffix (see Figure 4.1). Thus, the algorithm correctly computes $\mathcal{B}_s$.

We now analyze the running time via the displayed Algorithm 4-3. Assume an indexed alphabet. Then, $\texttt{prev}(T)$, $\texttt{compl}(T)$, $\texttt{sencode}(T)$, and $\alpha(T)$ are constructed in $O(n)$ time by Lemma 2.2.7. Since ψ executes in constant time via Lemma 4.3.2, the running time of the entire algorithm is clearly dependent on the s-matching of Ψ in line 9. This line is responsible for a sequence of symbol comparisons to conduct s-matches of which will require $O(b)$ comparisons in the worst case with b as the length of the maximum s-match by Lemma 4.3.3 and in this case, b is the also the longest s-border. This line is called when either (1) $j = h$ or (2) $j < h$ and there exists an s-match between two symbols. In case (1), at most b symbols may be matched, but h will be incremented so $h = j + b$ will force case (2). Case (2) is where any additional *rematching* is performed, that is, matching symbols in T that may have already been visited in case (1). So, the *total* comparisons by line 9 during an execution of the algorithm in case (1) is in $O(n)$. Further, the time required for the comparisons in case (1) is absorbed by the time bound required by the construction of the initial encodings. Let ϕ be the number of times that case (2) is executed. Then, the algorithm executes in $O(\max\{n, b\phi\})$ time. □

Depending on the s-string, there are cases in which Algorithm 4-3 will execute in linear time. That is, when either b or ϕ is small, the $\texttt{construct\_}\mathcal{B}_s\texttt{\_improved}$ algorithm executes in $O(n)$ time for an n-length s-string T. This is a significant improvement from the previously discussed solutions requiring time $O(n^2)$ and $O(n^3)$, but there are still cases when the algorithm will require more than linear time. Below, we discuss yet a further improvement to the $\mathcal{B}_s$ construction algorithm.

4.3.3 Further Improvement

We now investigate another fundamental property used in the traditional *border* construction. That is, the way in which the next longest border is found. Consider the longest border b_1 of T. If b_2 is the longest border of b_1, then b_2 is the next longest border of T. With this property, we can oracle previous *border* elements, $\mathcal{B}[e]$ with $e = \mathcal{B}[i] > 0$, to find the next longest border of $T[1...i]$. This property is used when the current longest border cannot be extended further and so, we can try the next longest border of which the final symbols may possibly match. Should the last symbol of the second longest border not match, we can oracle the third longest border, etc. So, the oracle may be recursive to the v^{th} level: $\mathcal{B}^1[i] = \mathcal{B}[i]$, $\mathcal{B}^2[i] = \mathcal{B}[\mathcal{B}[i]]$,..., $\mathcal{B}^v[i] = \mathcal{B}[\mathcal{B}^{v-1}[i]]$. Even with the changing s-suffixes by

Lemma 2.2.8, we prove that this property also holds for s-strings and the *s-border*.

Lemma 4.3.6 *Given an s-string T of length n, we find the length q_v of the v^{th} longest s-border by $q_v = \mathcal{B}_s^v[i]$ while $\mathcal{B}_s[i] > 0$ and $q_v = 0$ otherwise.*

Proof Consider the *s-border* array $\mathcal{B}_s$ for some element $\mathcal{B}_s[i] = q_1 > 0$. Then, $\texttt{sencode}(T[1...q_1]) = \texttt{sencode}(T[i - q_1 + 1...i])$ is the maximum s-border, i.e. the first longest s-border, of $T[1...i]$ by Definition 4.3.1. Now, consider the second longest s-border of $T[1...i]$ of length q_2, $0 < q_2 < q_1$. Let $\mathcal{B}_{s_1} = \texttt{sborder}(T[1...q_1])$ compute the *s-border* array for the input s-string. From the previous discussion of Figure 4.1, it follows now that the first longest s-border of $T[1...q_1]$ is also the second longest s-border of $T[1...i]$, i.e. $\mathcal{B}_{s_1}[q_1]$. Since $\texttt{sencode}(T[1...\mathcal{B}_{s_1}[q_1]]) = \texttt{sencode}(T[1...\mathcal{B}_s[q_1]])$ by Definition 2.2.4 and Theorem 2.2.6, the element is *already* known from the original $\mathcal{B}_s$ array element $\mathcal{B}_s[q_1]$, i.e. $\mathcal{B}_s[\mathcal{B}_s[i]] = \mathcal{B}_s^2[i]$. Thus, additional constructions of $\mathcal{B}_{s_j}$ via `sborder` are excessive and unnecessary. For the v^{th} longest border of $T[1...i]$, we must take the first longest s-border of $T[1...i]$, i.e. $q_1 = \mathcal{B}_s[i] > 0$, then the first longest s-border of $T[1...q_1]$, i.e. $q_2 = \mathcal{B}_s[q_1] > 0$, then the first longest s-border of $T[1...q_2]$, i.e. $q_3 = \mathcal{B}_s[q_2] > 0$, ..., then the first longest s-border of $T[1...q_{v-1}]$, i.e. $q_v = \mathcal{B}_s[q_{v-1}] > 0$. Overall, $q_v = \mathcal{B}_s^v[i]$. In any case, when $\mathcal{B}_s[j] = 0$ for some j, then no such longest s-border and subsequent s-borders can exist. So, $q_v = 0$. □

With the previously proven properties in Lemma 4.3.4 and Lemma 4.3.6, we are now able to propose a further improved solution in Algorithm 4-4 to compute the *s-border*. This is analogous to traditional *border* construction with the core difference being *how* the individual s-suffix symbols are observed and compared. Essentially, the proofs of Lemma 4.3.4 and Lemma 4.3.6 in addition to the s-match related functions in Algorithm 4-2 "evolve" the traditional *border* properties and construction algorithm to now construct the *s-border* array in $O(n)$ time.

Theorem 4.3.7 *Given an n-length s-string T, there is an algorithm that constructs the s-border array $\mathcal{B}_s$ in $O(n)$ time.*

Proof Algorithm `construct_`$\mathcal{B}_s$ builds the required *s-border* array. The correctness of the algorithm follows from the proofs relating *s-border* properties to traditional *border* construction properties in Lemma 4.3.4 and Lemma 4.3.6 and also, the correctness of the s-matching functions in Algorithm 4-2, which are developed using the theoretical foundations in Definition 2.2.2, Theorem 2.2.6, Definition 2.2.9, and Definition 2.1.2. We now analyze the running

time of `construct_`$\mathcal{B}_s$ from Algorithm 4-4. The key to the analysis is observing how many times Ψ in line 7 executes in relation to how quickly the array $\mathcal{B}_s$ is filled. Respectively, the variables that correspond to these events are m and h. Initially, $h = 2$, $j = 2$, and $k = 1$. Say that originally Ψ executes m_1 comparisons. Then, by Lemma 4.3.3, there exists a current longest s-match of length m_1, i.e. $\texttt{sencode}(T[1...m_1]) = \texttt{sencode}(T[j...j + m_1 - 1])$. Now, m_1 elements of $\mathcal{B}_s$ are populated and then h is advanced beyond the populated elements: $h = j + m_1$. Since $\texttt{sencode}(T[1...m_1 + 1]) \neq \texttt{sencode}(T[j...j + m_1])$, then either (1) line 12 is executed as an attempt to extend the next longest s-border starting at element k or (2) line 13 resets the algorithm to consider the s-suffix starting at h because no longer s-border exists. When case (1) executes, there are at most m_1 next longest s-borders to try. From Lemma 4.3.6, the next longest s-border is known to match and so Ψ continues the s-match at k so that no *rematching* is done. That is, now $j = h-k+1$ and subsequent s-matches of length m_2, m_3, etc., generally m_g, via Ψ are performed by s-matching the substrings of the encodings $\texttt{sencode}(T[i...i+k+m_g-2])[k...k+m_g-2] = \texttt{sencode}(T[j...j+k+m_g-2])[k...k+m_g-2]$ rather than rematching the complete encodings, including the already known s-border by $\texttt{sencode}(T[i...i + k + m_g - 2]) = \texttt{sencode}(T[j...j + k + m_g - 2])$. In other words, each m_g is the number of symbols that the s-match is extended, rather the the complete length of the s-match. When case (2) executes, even less work is done. Thus, Ψ performs a *total* of $O(n)$ comparisons during the execution of the `construct_`$\mathcal{B}_s$ algorithm. Since advances in h directly correspond to the s-match comparisons by Ψ and since there are at most a *total* of $O(n)$ next longest s-borders amortized across the $O(n)$ *total* work by Ψ, then the theorem holds. □

4.4 Generalization

From the previous section, the facts that s-suffix symbols are oracled efficiently and *s-border* shares fundamental construction properties used in traditional *border* construction leads to a $O(n)$ construction of *s-border* for an n-length s-string T. This result is significant not only for the *s-border*, but also for *p-border* and the traditional *border*. Such is the case because by modifying the alphabet for s-strings, the s-match problem becomes tailored for p-matching and even traditional matching. The following lemmas formalize the generalization possibilities for the `construct_`$\mathcal{B}_s$ algorithm.

Algorithm 4-3. Improved $\mathcal{B}_s$ construction

```
1  char prevT[n],complT[n],αT[n]
2  int[] construct_Bs_improved(char T[n]){
3   int h=2,i=1,j,k=1,m,q,x
4   int Bs[n]={0,0,...,0}
5   prevT=prev(T),complT=compl(T),αT=α(T)
6   for j=2 to n {
7    x=h−j+1,m=q=0
8    if(j=h ∨ (j<h ∧ ψ(i+x−1,j+x−1,x))){
9     q=m=Ψ(i,j,k)
10    while(m>0 ∧ j+m−1≥h){
11     Bs[j+m−1]=m, m−−
12    }h=j+q
13   }
14  }return Bs
15 }
```

Algorithm 4-4. Further improved $\mathcal{B}_s$ construction

```
char prevT[n],complT[n],αT[n]
int[] construct_Bs(char T[n]){
 int h=2,i=1,j=2,k=1,m,q,w=0
 int Bs[n]={0,0,...,0}
 prevT=prev(T),complT=compl(T),αT=α(T)
 while(h≤n){
  q=m=Ψ(i,j,k)
  while(m>0 ∧ j+m−1≥h){
   Bs[j+m−1]=m+w, m−−
  }h=j+q
  if(w+q>0 ∧ Bs[w+q]>0){
   k=Bs[w+q]+1,j=h−k+1,w=k−1
  }else{ k=1,j=h,w=0 }
 }return Bs
}
```

Lemma 4.4.1 *Given an n-length s-string T, the algorithm* `construct_`$\mathcal{B}_s$ *constructs the p-border array $\mathcal{B}_p$ in $O(n)$ time.*

Proof Set the alphabet of complement symbols to $\Gamma = \{(\pi_1, \pi_1), ..., (\pi_{|\Pi|}, \pi_{|\Pi|})\}$, $\pi_i \in \Pi$. Now, $\texttt{compl}(T) = \texttt{prev}(T)$. So, either $\texttt{sencode}(T)[i] = T[i]$ for $T[i] \in (\Sigma \cup \{\$\})$ or $\texttt{sencode}(T)[i] = \texttt{prev}(T)[i]$ for $T[i] \in \Pi$ by Definition 2.2.4. Already $\texttt{prev}(T)[i] = T[i]$ for $T[i] \in (\Sigma \cup \{\$\})$ by Definition 2.1.4. So, under these conditions $\texttt{sencode}(T) = \texttt{prev}(T)$ and s-matching by Theorem 2.2.6 is equivalent to p-matching by Theorem 2.1.9. Then, the *s-border* problem of Definition 4.3.1 becomes the *p-border* problem of Definition 2.1.18. Therefore, `construct_`$\mathcal{B}_s$ constructs the *p-border* array $\mathcal{B}_p$ in $O(n)$ time by Theorem 4.3.7. □

Lemma 4.4.2 *Given an n-length s-string T, the algorithm* `construct_`$\mathcal{B}_s$ *constructs the traditional border array $\mathcal{B}$ in $O(n)$ time.*

Proof Collect the parameter symbols into the set of constant symbols: $\Sigma = \Sigma \cup \Pi$. Now, set the alphabets of parameter and complement symbols to null: $\Pi = \Gamma = \emptyset$. In these conditions, $\texttt{sencode}(T)[i] = T[i]$ for $T[i] \in (\Sigma \cup \Pi \cup \{\$\})$ by Definition 2.2.4. Then, s-matching by Theorem 2.2.6 simplifies to traditional matching and the *s-border* problem of Definition 4.3.1 becomes the traditional *border* of Definition 2.1.17. Therefore, `construct_`$\mathcal{B}_s$ constructs the *border* array $\mathcal{B}$ in $O(n)$ time by Theorem 4.3.7. □

We summarize the foregoing results in the following theorem:

Theorem 4.4.3 *Given an n-length s-string T, the algorithm* `construct_`$\mathcal{B}_s$ *constructs the p-border array $\mathcal{B}_p$ and the traditional border array $\mathcal{B}$ each in $O(n)$ time.*

4.5 Conclusions

In this chapter, we introduce the structural border array ($\mathcal{B}_s$) for structural strings (s-strings). We provide numerous algorithms that continually improve our $\mathcal{B}_s$ construction for the n-length text T by exploiting the properties of the *s-border* array, ultimately arriving to an $O(n)$ solution. Finally, we provide a connection between $\mathcal{B}_s$, the traditional *border* array ($\mathcal{B}$), and the parameterized border array ($\mathcal{B}_p$) by showing that each array can be constructed with the same $\mathcal{B}_s$ construction algorithm.

Chapter 5

Compressed Parameterized Pattern Matching

This work resulted in the following publication.

- Beal, R., Adjeroh, D.: Compressed parameterized pattern matching. IEEE Data Compression Conference (DCC). pp. 461-470. IEEE. (2013)

5.1 Introduction and Background

Consider a pattern P and a text T, where T is compressed to form T_c. In applications where we wish to detect similar biological sequences [86] or identify plagiarism [11, 98] between P and T, we can solve the problem with parameterized pattern matching (p-match) [11]. To some extent we can address this problem following the work of Apostolico et al. [8, 7] for p-matching between P and T compressed via run-length encoding. There is currently no work to support p-matching between an uncompressed P and compressed T for arbitrary compression schemes.

The p-match problem was introduced by Baker [11] as a special form of inexact matching not correctly and efficiently supported by approximate matching schemes. More specifically, the p-match compares strings considering both exact symbols from the constant alphabet Σ and potentially inexact symbols from the parameter alphabet Π. The first p-match solutions revolved around the parameterized suffix tree (p-suffix tree) [11]. Various works, including [68], have improved the p-suffix tree construction. Due to the large memory footprint of the p-suffix tree [11] implementation, the p-match was also solved using extensions of traditional

pattern matching schemes by Baker [11] and Amir et al. [6]. The multiple p-match problem is addressed by Idury et al. [56]. The parameterized suffix array (p-suffix array) was introduced by I et al. [53] as a lightweight alternative to the p-suffix tree. We offer theoretical improvements to p-suffix array construction in [21] and study other p-match data structures in [19, 17].

In this work, we focus on lossless compression. Denote the length of T_c, the compressed form of T, as n_c and denote the number of p-matches of the m-length P in the n-length T as n_{occ}. Consider the LZ family. Navarro et al. [76] perform compressed exact matching in $O(\min\{n, mn_c\} + n_{occ})$ time for LZ77 whereas Kärkkäinen et al. [60] address the compressed approximate matching problem in $O(mkn_c + n_{occ})$ time for LZ78, where k is the permitted number of edits, insertions, and deletions. Matching in LZW compressed text is studied in [43]. Other compression schemes have been studied on similar grounds. The relationship between the BWT and pattern matching is detailed in [1]. Dictionary based compression is the focus of the compressed matching of [63, 64]. Variable-to-fixed length coding is studied in [64, 97, 62, 92, 91]. Apostolico et al. [8] address the p-match in terms of fully compressed run-length encodings in $O(n+(r_P \times r_T)\alpha(r_T)\log(r_T))$ time, where α is the inverse of Ackermann's function and r_T and r_P respectively denote the number of runs in the encodings for T and P. In [7], Apostolico et al. provide an alternative solution in $O(r_P \times r_T)$ time.

Main Contributions: We formally define the compressed parameterized pattern matching (compressed p-matching) problem to find all of the p-matches between a pattern P and text T, using only the uncompressed P and the compressed text T_c. Initially, we introduce parameterized compression (p-compression) as a way to losslessly compress a text to support p-matching. Experimentally, it is shown that p-compression is competitive with standard lossless compression schemes. Our solution to the compressed p-matching problem is developed using a general partial decompression method ∂. With the recent interest in variable-to-fixed length coding [64, 97, 62, 92], we show how to define ∂ for Tunstall codes. Our main results are formalized below.

Theorem 5.2.5. *Given T_c, the compressed form of the n-length text T, and P, an m-length pattern, compressed p-matching can be performed in $O(n\mu)$ time with $O(\max\{m, \upsilon\})$ extra space, where the partial decompression function ∂ executes in $O(\mu)$ time with $O(\upsilon)$ extra space.*

Theorem 5.3.2. *Given T_c, the compressed form of the n-length text T using Tunstall codes,*

and P, an m-length pattern, compressed p-matching can be performed in $O(n)$ time with $O(m)$ extra space.

5.2 Compressed Parameterized Pattern Matching

In the case of identifying similar biological sequences or detecting plagiarism, we may wish to compress the underlying data, say T, and conduct p-matching between the compressed form of T, say T_c, and some arbitrary pattern P. This is the notion of the compressed parameterized pattern matching (compressed p-matching) problem, which cannot be directly supported by current compressed matching schemes. We now define the `cppm` function to formalize the compressed p-matching problem.

Definition 5.2.1 Compressed p-matching function (`cppm`(T_c, P)): *Consider a traditional n-length string T with $T[1...(n-1)] \in X$ for the alphabet $X = \{x_1, x_2, ..., x_{|X|}\}$ and $T[n] = \$$. Let T be compressed into T_c of length n_c. Given a pattern P of length m where each symbol $P[i]$ $(1 \leq i \leq m)$ is from either the chosen parameter alphabet $\Pi \subseteq X$ or the chosen constant alphabet $\Sigma \subseteq X$ where $(\Sigma \cup \Pi) = X$ and $(\Sigma \cap \Pi) = \emptyset$, the compressed p-matching problem is to use P and T_c to compute the following array: $atIndex = \{i \mid \texttt{prev}(T[i...i+m-1]) = \texttt{prev}(P[1...m])\ \forall\ i, 1 \leq i \leq n-m+1\}$.*

5.2.1 Windowed-Previous Encoding

We introduce parameterized compression as the compression of a p-string encoding. Conducting parameterized compression on the `prev` encoding may appear counter-intuitive at first, given that for an n-length T, the number of distances in $\texttt{prev}(T)$ may be in $O(n)$. The fact that the alphabet may be of the same order as the uncompressed string itself is not beneficial for dictionaries or symbol representation and heavily limits compression. Now, we introduce the windowed-previous encoding scheme (`wprev`) to encode a p-string with limited and finite distances up to a maximum distance d. As a result, `wprev` is a transformation of T that can be compressed.

Definition 5.2.2 Windowed-previous encoding (`wprev`) function: *Let function $\mathcal{J}(y, Y)$ return the lexicographical order $(1, 2, ..., |Y|)$ of the symbol y in alphabet Y. Let $\mathcal{K}$ reverse the process, i.e. $\mathcal{K}(\mathcal{J}(y, Y), Y) = y$. Then, function `wprev` generates a set of integers for each i, $1 \leq i \leq n = |T|$ where:*

$$\texttt{wprev}(T,q,d)[i] = \begin{cases} \mathcal{J}(T[i], \Sigma \cup \Pi \cup \{\$\}) + d, \text{ if } T[i] \in (\Sigma \cup \{\$\}) \ \vee (T[i] \ \in \ \Pi \ \wedge \\ \quad ((d < \min\{k \mid T[i-k] = T[i], (i-k) \geq 1 \wedge k = 1, 2, ...\}) \\ \quad \vee \ (q = true \ \wedge \ T[i] \neq T[j] \text{ for every } 1 \leq j < i))) \\ 0, \text{ if } q = false \ \wedge \ T[i] \in \Pi \wedge T[i] \neq T[j] \text{ for every } 1 \leq j < i \\ i - k, \text{ if } T[i] \in \Pi \wedge k = \max\{j \mid T[i] = T[j], 1 \leq j < i\} \end{cases}$$

Omitting the terminal for brevity, consider $\Sigma = \emptyset, \Pi = \{x \rightarrow 1, y \rightarrow 2\}$, and $T = xyxyyyyyx$, where $\texttt{prev}(T) = 002211116$. Here, we have $\texttt{wprev}(T, true, 3) = 452211114$ and $\texttt{wprev}(T, false, 3) = 002211114$. The parameter d identifies the number of parameter distances $0, 1, ..., d$ that are used in `wprev`. In practice, d is chosen based on the alphabet size and the underlying data type used. The boolean variable q determines how to encode the parameters in the transformation. When $q = true$, all of the data is encoded in $\texttt{wprev}(T, true, d)$ that is necessary to obtain the original T since constants are encoded verbatim and distances are used to reconstruct the parameters. When $q = false$, we need to prepend a header h to $\texttt{wprev}(T, false, d)$ to identify the order of occurrence of the (at most $|\Pi|$) parameters that are encoded by 0. The header h tells us how to replace the 0 distances when converting $\texttt{wprev}(T, false, d)$ to T.

To compress and decompress the `wprev` transformation, we derive the following routines: (1) `p_compress` in Algorithm 5-1 which uses T to prepare the `wprev` encoding to be compressed by some standard compression scheme `compress` and (2) `p_decompress` in Algorithm 5-2 to `decompress` the compressed sequence into the original `wprev` encoding used to reconstruct the original T. The advantage of $T_c = \texttt{p\_compress}(T)$ for p-matching is that (1) a number of parameter distances in $\texttt{prev}(T)$ are already identified for p-matching and (2) a compressor may exploit the transformation to discover "hidden" patterns within a text, reveal relationships between the distances of like-symbols, and alter symbol probabilities based on the way in which the parameters are chosen by $\texttt{choose}(\Sigma, \Pi, q, d)$. It is assumed that these values are relayed between `p_compress` and `p_decompress`. Since `wprev` can be constructed in linear time for an indexed alphabet with $O(1)$ mapping operations $\mathcal{J}$ and $\mathcal{K}$, then functions `p_compress` and `p_decompress` clearly execute in the time of the underlying `compress` and `decompress` routines, respectively. Thus, the cost of p-compression in terms of time is negligible.

To see how the `wprev` compression compares to the compression of T, we performed experiments using numerous compression schemes on various ASCII data sets and identified

Algorithm 5-1. Compressing an encoding.

```
choose(Σ,Π,q,d)
String p_compress(String T){
    String y = wprev(T,q,d),h=null
    int i
    if(q) return compress(y)
    for i=1 to |y|
        if(y[i]=0)
            h = h ∘ (J(T[i],Σ∪Π∪{$})+d)
    return compress(h ∘ SEPARATOR ∘ y)
}
```

Algorithm 5-2. Decompressing an encoding.

```
choose(Σ,Π,q,d)
String p_decompress(String z){
    String y=decompress(z),T=null
    int i,j=1,k=max{1,y.indexOf(SEPARATOR)+1},p=1
    for i=k to |y| {
        if(y[i]=0){ T = T ∘ K(y[p]−d,Σ∪Π∪{$}), p++ }
        else if(y[i]≤d) T = T ∘ T[j−y[i]]
        else T = T ∘ K(y[i]−d,Σ∪Π∪{$})
        j++
    }return T
}
```

the number of bytes required for the compression of T and the number of bytes required for the compression of `wprev` transformations. The compression schemes used were `winrar` from win-rar.com, `Tunstall` developed by Yoshida and Kida [97, 62], and `lzma`, `bzip2`, and `ppmd` from 7-zip.org. We tested on the Ecoli sequence, the Bible text, and a random sequence $\mathcal{U}(1, 32)$ where $\mathcal{U}(a, b)$ is on the range $[a, b]$. Each sequence was appended with a terminal symbol. The Bible sequence was altered for p-matching in the following way: only letters and spaces remained in the text, all uppercase letters were converted to lowercase, and the most frequently occurring 14 words, considered parameters, were replaced with a corresponding unique uppercase letter. The experimental results are shown in Table 5.1. We observe that T and the `wprev` transformation have very similar compression results in most cases, especially for small d. In the case of Bible and `winrar`, for example, we see that the transformation $\texttt{wprev}(T, true, 2)$ achieves better compression than T. In the case of the Ecoli sequences with `Tunstall`, we see that `wprev` transformations do not compress as well as T, even though the compressions of T and `wprev` are nearly identical for the other cases of `Tunstall`. We attribute this to the very small alphabet of the Ecoli sequence and the fact that the `wprev` scheme introduces additional elements to this alphabet for a variable-to-fixed length coder to handle. In the case of $\mathcal{U}(1, 32)$, we note that the compression schemes do not compress the *random* data. Nonetheless, the results suggest that compressing the `wprev` transformation does not deter much from traditional compression and in fact, may even offer better compression.

5.2.2 Challenges

Earlier, we discussed how to construct the compressed T_c, i.e. $T \overset{\texttt{wprev}}{\rightarrow} W \overset{\texttt{compress}}{\rightarrow} T_c$ or $T \overset{\texttt{p\_compress}}{\rightarrow} T_c$. We now focus on devising a solution to $\texttt{cppm}(T_c, P)$ by first identifying the differences between compressed p-matching and compressed matching that require a new approach: (1) our compressed p-matching problem uses a compressed transformation T_c and (2) finding all p-matches of P in the n-length T requires handling parameterized suffixes (p-suffixes) $\texttt{prev}(T[i...n])$, which are dynamic since $\texttt{prev}(T[i...n]) = \texttt{prev}(T)[i...n]$ is not always true for each i, $1 \leq i \leq n$ (see Lemma 2.1.10).

Consider for example $\Sigma = \{A\}$, $\Pi = \{v, w\}$, $T = wAww\$$, and so, $\texttt{prev}(T) = 0A21\$$. Unlike the traditional suffixes $\boxed{wAww\$} \rightarrow \boxed{Aww\$} \rightarrow \boxed{ww\$} \rightarrow \boxed{w\$} \rightarrow \boxed{\$}$, the p-suffixes are dynamically changing: $\boxed{0A21\$} \rightarrow \boxed{A01\$} \rightarrow \boxed{01\$} \rightarrow \boxed{0\$} \rightarrow \boxed{\$}$. Now, consider finding

Table 5.1: Comparison of various compression schemes on texts T and transformations $\texttt{wprev}(T, q, d)$ for $q = true, false$ and $d = 1, 2, 3, 4$ where $C = \texttt{compress}(T)$.

text (T)	`compress` routine	$\lvert C\rvert$ in bytes	$\lvert\Pi\rvert$	$\lvert\Sigma\rvert$	$\lvert\texttt{p\_compress}(T)\rvert$ in bytes							
					q=*true*, d=1	q=*true*, d=2	q=*true*, d=3	q=*true*, d=4	q=*false*, d=1	q=*false*, d=2	q=*false*, d=3	q=*false*, d=4
Bible	`winrar`	877778	14	27	877782	877726	877733	877907	877872	877807	877823	877999
n=3445275	`lzma`	1029564			1036725	1037996	1036499	1036647	1036767	1038038	1036541	1036690
	`bzip2`	777355			777345	777345	777345	777222	778411	778411	778411	777254
	`ppmd`	757062			757012	757012	757012	757063	757081	757081	757081	757128
	`Tunstall`	2019356			2019356	2019356	2019356	2023660	2023496	2023496	2023496	2025712
Ecoli	`winrar`	1327266	4	0	1360435	1425487	1507922	1574267	1360432	1425483	1507873	1574231
n=4638690	`lzma`	1358668			1411071	1499807	1604996	1682661	1411089	1499828	1605014	1682679
	`bzip2`	1250993			1235106	1279065	1360862	1436509	1235082	1278279	1360948	1436670
	`ppmd`	1138456			1140917	1175065	1258703	1315264	1140940	1175086	1258720	1315285
	`Tunstall`	1209650			1375638	1521040	1614234	1700618	1408450	1538108	1636836	1723960
$\mathcal{U}(1, 32)$	`winrar`	679408	32	0	681505	684396	687843	691440	681542	684429	687884	691484
n=1000000	`lzma`	695218			696808	701039	706087	711155	696839	701069	706117	711185
	`bzip2`	630713			631997	634021	640727	646038	632190	634158	640730	646395
	`ppmd`	647486			647180	650047	656233	662194	647228	650104	656287	662240
	`Tunstall`	659900			661286	662682	663666	664394	661662	662976	663978	664668

all p-matches of $P = v$ in T. This problem requires that we first determine $\texttt{prev}(P) = 0$. We report an occurrence at i of P in T when $\texttt{prev}(T[i...i + |P| - 1]) = \texttt{prev}(P) = 0$, which yields the occurrences $\{1, 3, 4\}$. In this example, we see that taking substrings directly from $\texttt{prev}(T)$ will overlook the occurrences $\{3, 4\}$. For compressed texts, the p-matching task is even more difficult since we are analyzing $O(n)$ p-suffixes with a total of $O(n^2)$ p-suffix symbols in a compressed form. This gives further merit to efficient solutions for compressed p-matching.

5.2.3 Algorithm

Before we begin the discussion of our `cppm` algorithm, we list some initial notes and assumptions. Consider the indexed alphabet $\mathcal{A} = \{a_1, a_2, ..., a_{|\mathcal{A}|}\}$. For ease of discussion, assume that $T_c = \texttt{p\_compress}(T)$ is such that the function $\texttt{choose}(\Sigma_T, \Pi_T, q, d)$ uses $\Sigma_T = \emptyset$ and $\Pi_T = \mathcal{A}$ as the `wprev` alphabets for the T transformation prior to compression, $q = true$, and a fixed d. By extending our future discussion, other cases can also be supported. Now, let the m-length pattern P have alphabets Σ and Π from $\mathcal{A}$ such that $\Sigma \subseteq \mathcal{A}$ and $\Pi \subseteq \mathcal{A}$ where $(\Sigma \cup \Pi) = \mathcal{A}$ and $(\Sigma \cap \Pi) = \emptyset$. These particular alphabets determine how the compressed p-matching is to be performed. To avoid unnecessary complications to the discussion, we do not append a terminal \$ to P. Finally, we assume the existence of a special partial decompression function $\partial(T_c, i)$ that, using T_c, will retrieve the original `wprev` encoded symbol at i, say $W[i]$. It is assumed that ∂ will be called sequentially for $i = 1, 2, ..., n$. The actual development of ∂ is a core contribution discussed later.

Consider an m-length pattern P and an n-length text T. Algorithm 5-3 addresses the compressed p-matching (`cppm`) problem by constructing the set: $atIndex = \{i \mid \texttt{prev}(T[i...i+m-1]) = \texttt{prev}(P[1...m])\ \forall\ i, 1 \leq i \leq n-m+1\}$. The algorithm accepts the compressed text T_c, the uncompressed pattern P, and the array A, which determines the pattern alphabets: $A[l] = 0$ if the pattern alphabet symbol mapped to l is a constant and $A[l] = 1$ if the symbol mapped to l is a parameter. We initially generate $\mathcal{B}_p$, the *p-border* array for P, and also, $prevP$, the `prev` encoding for P. Let $L[|\Sigma| + |\Pi|] = \{0, 0, ..., 0\}$ and $j = 0$. The algorithm iterates through each original symbol at $i = 1, 2, ..., n$ and works with one decompressed `wprev` symbol $W[i]$ at a time. That is, we retrieve the symbol $sym = \partial(T_c, i)$. This sym is now the original $W[i]$ and so, the next task will be to convert the sym to a `prev` encoded symbol for p-matching purposes. This is done by observing the integer

ranges in Definition 5.2.2. When $sym > d$, we are working with an encoded symbol (not a distance) and so, we can decrement sym to determine the integer that represents the symbol: $val = sym - d$. When $sym \leq d$, we are working with a distance and so, we must look through L to determine the symbol in which the distance refers: val. At this point, we have obtained, in val, the actual integer representation of the symbol $T[i]$.

We continue by adjusting val to the valid encoded value $\texttt{prev}(T[i-j...i])[j+1]$ with two cases. (1) If $A[val] = 0$, we are working with a constant in P, so the correct encoding is the symbol itself: $C_g = \sigma_{val}$ (recall notation from Chapter 2). (2) Otherwise, val refers to a parameter. Initially, say that the parameter is the first occurrence, so simply let $C_g = z_0$. Now, if the value occurred before, $L[val] \geq 1$, and if the distance is such that $(i - L[val]) < j + 1$, then the distance is valid and belongs in the p-suffix prefix: $C_g = z_{i-L[val]}$. In both cases, record the location of this symbol: $L[val] = i$. At this point, we have the current `prev` encoded symbol C_g from the $j+1$ symbol of the p-suffix at $i-j$ in T and we can begin the p-match with $prevP$. A p-match exists by Theorem 2.1.9 iff all of the `prev` encoded symbols match. So, if $C_g = prevP[j+1]$, then we know that $j{+}{+}$ symbols p-match. However, a mismatch must halt the process because a p-match cannot exist between P and the current p-suffix in T. So, we now need to find the length b of the longest p-suffix that is a prefix of the failed p-match, which will assist in advancing the p-match. That is, when $j > 0 \wedge b > 0$, find the maximum b where $\texttt{prev}(T[i-j...i-j+b-1]) = \texttt{prev}(T[i-b...i-1]) = prevP[1...b] = \texttt{prev}(P[1...b])$. By Definition 2.1.18, $b = \mathcal{B}_p[j]$. So, we set $j = \mathcal{B}_p[j]$ and appropriately reset C_g for parameters considering a new p-suffix $\texttt{prev}(T[i-j...i])$ until either $j = 0$ or $\mathcal{B}_p[j] = 0$ or $C_g = prevP[j+1]$. If we reach a point where indeed $C_g = prevP[j+1]$, set $j{+}{+}$ to signify a new matching symbol. As we oracle elements in $\mathcal{B}_p$, we are "attempting" to begin the next p-match with the $1st$ longest p-border, and if this fails, the $2nd$ longest p-border, etc. At any time, when m symbols match, i.e. $j = m$, then we can report a p-match: $atIndex = atIndex \cup (i - m + 1)$. In this situation, we know that the prefix of the p-suffix in T p-matches fully with P, i.e. $\texttt{prev}(T[i-j+1...i]) = prevP = \texttt{prev}(P)$, and so, we know that $\mathcal{B}_p[j]$ gives us the maximum known p-match to continue at the next iteration, i.e. if $\mathcal{B}_p > 0$, then the maximum p-border is $\texttt{prev}(T[i - \mathcal{B}_p[j] + 1...i]) = prevP[1...\mathcal{B}_p[j]] = \texttt{prev}(P[1...\mathcal{B}_p[j]])$. When a p-match exists, let $j = \mathcal{B}_p[j]$. In any case, continue to the next iteration $i{+}{+}$.

Essentially, the aforementioned `cppm` computation (see Algorithm 5-3) performs three major tasks. (1) Each `wprev` compressed symbol is decompressed and converted to a `prev`

Algorithm 5-3. Parameterized pattern matching using pattern P and $T_c = \texttt{p\_compress}(T)$, the compressed form of T, where $n = |T|$ and $m = |P|$.

```
1  // for brevity, assume no use of terminal $
2  List cppm(String Tc,String P,bit A[|Σ|+|Π|]){
3     int Cg,val,sym,h,i,j=0,L[|Σ|+|Π|]={0,0,...,0}
4     int Bp[m]=pborder(P),prevP[m]=prev(P)
5     List atIndex; boolean flag
6     for i=1 to n {
7        // A - prepare new symbol
8        Cg=$, sym=∂(Tc,i), val=sym-d
9        if(sym≤d) {
10          h=1, val=-1, Cg=z_sym
11          while(h≤(|Σ|+|Π|) ∧ val=-1) {
12             if(L[h]=i-sym) val=h
13             h++
14          }
15       } // B - adjust prev encoded value
16       if(A[val]=1) { // symbol is parameter in P
17          Cg=z0
18          if(L[val]≥1 ∧ (i-L[val])<j+1) Cg=z_{i-L[val]}
19       }else // symbol is constant in P
20           Cg=σ_val
21       L[val]=i
22       // C - p-match
23       if(Cg=prevP[j+1]) //...match
24          j++
25       else { //...mismatch
26          flag=true
27          while(flag ∧ j>0 ∧ Bp[j]>0) {
28             j=Bp[j]
29             if(C=z ∧ g≥j+1) Cg=z0
30             if(Cg=prevP[j+1]){ flag=false, j++ }
31          }if(j>0 ∧ Bp[j]=0) j=0
32       }if(j=m) { //...report p-match
33             atIndex=atIndex ∪ (i-m+1), j=Bp[j]
34       }
35    }return atIndex
36 }
```

Algorithm 5-4. Partial decompression ∂^k_{TC} for Tunstall codes when $W = \texttt{wprev}(T, true, d)$ and T_c=$\texttt{p\_compress}(T)$=$\texttt{compress}(W)$=$\texttt{Tunstall}_k(W)$.

```
1  // given dictionary D = {d_1, d_2, ..., d_D}
2  // let l_i = |d_i| and L = sum_{i=1}^{D} l_i
3  struct pair { int start, int length }
4  int start=0,length=0,u=0,r=0,S[L]
5  pair DT[D] ; bit index[k]
6
7  int ∂^k_TC(bit T_c[n_c], int i){
8      int s=i*k-k,h,j,q=1,sym
9      if(r=0){
10         // construction of DT and S
11         for j=1 to D {
12             DT[j]=(q,l_j)
13             for h=1 to l_j
14                 S[q+h-1]=d_j[h]
15             q=q+l_j
16         }
17     }if(i=1)
18         r++
19     sym=0
20     if(length=0 ∧ i≤n){
21         // locate new D substring
22         for j=1 to k
23             index[j]=T_c[j+s]
24         (start,length)=DT[index]
25         u=0
26     }// retrieve one symbol
27     if(u<length){
28         u++
29         sym=S[start+u]
30     }// consider new D substring
31     if(u=length)
32         start=length=u=0
33     return sym
34 }
```

encoded symbol for future p-matching purposes. (2) Each `prev` encoded symbol is adjusted in terms of the p-suffix considered. (3) All p-matches are performed by comparisons between `prev` as in Theorem 2.1.9 and p-matches are advanced with the *p-border* array $\mathcal{B}_p$. The correctness of the algorithm follows from the fact that each symbol in the original text T is accessed left-to-right, p-matches are made via Theorem 2.1.9, and p-matches are advanced by successive next longest candidates via $\mathcal{B}_p$ in Definition 2.1.18. The following lemma formalizes the time complexity of the algorithm. The proof uses an amortized analysis.

Lemma 5.2.3 *For the n-length text T, Algorithm 5-3 executes in $O(n\mu)$ time, where the function ∂ executes in $O(\mu)$ time.*

Proof Let the notation $t_{\texttt{fun}}$ denote the time required to execute function `fun`. Given the function parameters T_c and the m-length P, the algorithm clearly executes in $O(t_{\texttt{pborder}} + t_{\texttt{prev}} + n \times (t_\partial + (|\Sigma| + |\Pi|) + \mathcal{G}))$ time, where $\mathcal{G}$ denotes the number of times that lines 28-30 execute. By Lemma 2.2.7 and [17], $t_{\texttt{pborder}} = t_{\texttt{prev}} = O(m)$ on P. Let $t_\partial = O(\mu)$. Finally, we must determine $\mathcal{G}$. The **while** loop at line 27 is *true* for nonzero j and nonzero $\mathcal{B}_p[j]$, where successively $j = \mathcal{B}_p[j]$. It readily follows from Definition 2.1.18 that for nonzero $\mathcal{B}_p[j]$, strictly $\mathcal{B}_p[\mathcal{B}_p[j]] < \mathcal{B}_p[j]$ because the next longest p-border must be shorter than the longest p-border. So, the loop terminates because successive p-border lengths decrease. Further, there is a relationship between the number of successful matches (incrementing j) in lines 23-24 and the execution of lines 28-30. Since n symbols are each processed exactly once in the **for** loop (lines 6-35), then there are at most n successful matches by lines 23-24 that increment j during the entire algorithm. It follows that there are at most $(n-1)$ p-borders used throughout the entire algorithm by lines 28-30. So, the total number of executions of lines 28-30 amortize so that $O(1)$ time is contributed to each iteration of the **for** loop (lines 6-35). Then, $\mathcal{G} = O(1)$. Thus, the overall time of the algorithm is $O(m + n \times \mu) \in O(n\mu)$ since $m \leq n$. □

Below, we formalize the space required by `cppm`.

Lemma 5.2.4 *For the n_c-length compressed text T_c and the m-length pattern P, Algorithm 5-3 requires $O(\max\{n_c, m, n_{occ}, \upsilon\})$ space, where n_{occ} is the total number of p-matches that exist and where the function ∂ demands $O(\upsilon)$ space.*

Proof Let the notations s_{var} and $s_{\texttt{fun}}$ respectively denote the total space of the variable *var* and the extra space, beyond function parameters and return values, required to execute

function `fun`. Thus, the algorithm requires $O(s_{T_c} + s_P + s_{prevP} + s_A + s_L + s_{\mathcal{B}_p} + s_{atIndex} + s_{\texttt{prev}} + s_{\texttt{pborder}})$ space. Collectively, we know that $s_{T_c} = O(n_c)$ and $s_P = s_{prevP} = s_{\mathcal{B}_p} = O(m)$. By [17], $s_{\texttt{pborder}} = O(1)$. Also, $s_A = s_L = O(|\Sigma| + |\Pi|)$ from the algorithm and $s_{\texttt{prev}} = O(|\Pi|)$ by Lemma 2.2.7, and if we consider the alphabets as practical constants, $s_A = s_L = s_{\texttt{prev}} = O(1)$. Let $s_{atIndex} = O(n_{occ})$ and $s_\partial = O(\upsilon)$. Therefore, the overall space is $O(n_c + m + n_{occ} + \upsilon) \in O(\max\{n_c, m, n_{occ}, \upsilon\})$. □

Since $O(\upsilon)$ space is used by ∂ and both $\mathcal{B}_p$ and $prevP$ are *extra* declarations, only $O(\max\{m, \upsilon\})$ extra space is required. The results are summarized below.

Theorem 5.2.5 *Given T_c, the compressed form of the n-length text T, and P, an m-length pattern, compressed p-matching can be performed in $O(n\mu)$ time with $O(\max\{m, \upsilon\})$ extra space, where the partial decompression function ∂ executes in $O(\mu)$ time with $O(\upsilon)$ extra space.*

Earlier, we stated that the compressed p-matching problem is combinatorial because we need to detect occurrences of P in a total of $O(n^2)$ symbols over n dynamically changing p-suffixes in the n-length T. The *p-border* expedites the time required and also attributes to the extra space required by our algorithm. The time required for our compressed p-matching solution is linear in T when the partial decompression function ∂ is done in a constant number of steps per symbol. Our solution requires extra space linear in P when ∂ is done with at most $|P|$ space. Even though [8] addresses a different problem, i.e. fully compressed p-matching, we also achieve a time result based on $|T|$. In the case of [7], the time complexity is a term based on the number of runs in the encodings for P and T. These results are different from traditional compressed matching solutions that can achieve time results based on $|T_c|$. In contrast, our approach is more general, applicable for any compression scheme where ∂ can be defined. If we define ∂ as the `catenate` utility, we achieve regular p-matching in $O(n)$ time with $O(1)$ extra space, which is analogous to the conventional p-matching of [11, 6]. Given the recent interest and pattern matching abilities of variable-to-fixed length coding [64, 97, 62, 92], we define ∂ for Tunstall codes, i.e. ∂^k_{TC}, in the following section.

5.3 Tunstall Code Partial Decompression

A Tunstall code [91] is devised by developing a dictionary $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ of variable length sequences for the originally uncompressed string T. Each $d_i \in \mathcal{D}$ is represented

with a fixed k-length code from 2^k sequential codes, where k is chosen in practice so that $k \mod 8 = 0$. The original sequence T is divided into substrings, which are candidates to be entries in $\mathcal{D}$. The construction of $\mathcal{D}$ is beyond the scope of this work. We denote this compression scheme as $\texttt{Tunstall}_k$.

Klein and Shapira [64] provide an algorithm to decompress text encoded with Tunstall codes, making use of the aforementioned variables in addition to a dictionary table DT of pairs $(start, length)$ used to locate text within the string $S = d_1 \circ d_2 \circ ... \circ d_D$. Their algorithm reads the compressed sequence of Tunstall codes k bits at a time and assigns this value to i. Then, the Tunstall code is decoded from the substring $S[start...start + length - 1]$ by simply indexing S with $DT[i] = (start, length)$. By repeating this process, the concatenation of the individual $S[start...start + length - 1]$ will produce the original sequence. The problem of using the described decompression scheme directly in Algorithm 5-3 is the rate of decompression. Recall that the ∂ function is designed to be called sequentially $i = 1, 2, ..., n$ and use T_c to retrieve the symbol originally compressed at i within the n-length `wprev` encoding. So, we develop ∂^k_{TC} in Algorithm 5-4 to work left to right in T_c, partially decompress T_c, and return a single symbol sequentially. The following lemma formalizes the time and extra space required to execute each call to ∂^k_{TC}.

Lemma 5.3.1 *Each call to ∂^k_{TC} requires $O(k)$ time and $O(D)$ extra space, where D is the number of substrings in the dictionary $\mathcal{D}$.*

Proof Consider the time analysis first. It is clear that DT and S are constructed only once in $O(\mathcal{L})$ time, where $\mathcal{L} = l_1 + l_2 + ... + l_D$. This construction time is amortized across all n calls to the function. During each execution in the worst case, we must read a Tunstall code that requires $O(k)$ time, so $O(k + \frac{\mathcal{L}}{n})$ time is required. Since the construction of Tunstall codes is designed to make $\mathcal{L}$ negligible with respect to the uncompressed text size n, then $O(k + \frac{\mathcal{L}}{n}) \in O(k)$. Now consider space. Since T_c of length n_c is required for any partial decompression function, it is clear from Algorithm 5-4 that only the extra structures DT, S, $\mathcal{D}$, and *index* are significant. The $\mathcal{D}$ is utilized in the compression routines, so we can omit this as extra space due to ∂. Further, we may replace $\mathcal{D}$ with the concatenated representation of S and DT, which can perform the same functions. Since we can replace $\mathcal{D}$ with S of the same total elements $O(\mathcal{L})$, then only DT of length $O(D)$ and *index* of length $O(k)$ are considered extra space. Since $D > k$, then the extra space is in $O(D)$. □

Since k is chosen in practice as a fixed bit length and $D \leq 2^k$, we can consider both k

and D as constants. Thus, each call to ∂_{TC}^{k} requires $O(1)$ time and $O(1)$ extra space. The next theorem specializes Theorem 5.2.5 to the case of Tunstall coding.

Theorem 5.3.2 *Given T_c, the compressed form of the n-length text T using Tunstall codes, and P, an m-length pattern, compressed p-matching can be performed in $O(n)$ time with $O(m)$ extra space.*

5.4 Conclusion

We formally define the compressed parameterized pattern matching (compressed p-matching) problem to find all of the p-matches between a pattern P and text T, using only the uncompressed P and the compressed text T_c. We introduce the idea of parameterized compression – a way to losslessly compress T via a new p-string encoding `wprev` to better support p-matching. It is shown experimentally that T and `wprev` on T have similar compression performance. Our solution to the compressed p-matching problem is provided for any compression scheme via a general partial decompression function ∂. We then develop ∂ for Tunstall coding as an example. By more closely investigating other compression schemes to define ∂, we can add to the utility of our compressed p-matching result.

Chapter 6

Parameterized Strings with Indeterminate Symbols

In this chapter, we pioneer the notion of indeterminate symbols with parameterized strings, which are composed of symbols from the constant alphabet Σ and the parameter alphabet Π. Below, we propose and address two new variations of the parameterized match (p-match) with indeterminate symbols: the indeterminate parameterized match (ip-match) and the equivalence parameterized match (e-match). The ip-match allows constant and parameter holes in a string. The e-match allows disjoint groups of symbols from Σ or Π to be indeterminate. While the p-match itself is a special type of inexact matching, we propose, in this chapter, to extend the inexact capabilities of the p-match via matching with indeterminate symbols.

6.1 Parameterized Prefix Array

A parameterized string (p-string) is composed of symbols from a constant alphabet Σ and a parameter alphabet Π. Two p-strings S and T are parameterized matches (p-matches) if (1) all of the constant symbols match exactly and (2) the parameter symbols form a bijection. In other words, the p-match is a special form of inexact pattern matching, where the constants match exactly and the parameters consistently correspond. While the already inexact p-match framework has been taken to *further* inexact forms such as approximate matching [49] and matching with mismatches [8, 7], the research in this area is very limited. Currently, the notion of allowing indeterminate symbols in p-strings with indeterminate

pattern matching is not supported.

We are motivated to study this area because of the generalization of the p-match and the intricate applications that the p-match naturally addresses. By using only the Σ alphabet, any p-match algorithm/result also applies to traditional string theory [87, 47, 1], the foundation of solutions to various applications. Further, the p-match naturally addresses problems in biology [86], software engineering [12], academia [11], and music. Various music applications have been approached from the perspective of string theory and pattern matching [30, 4]; we will later make the connection between music and the p-match. By allowing another level of inexactness to the p-match, we will add to the capability of the p-match framework and support more sophisticated applications.

Here, we consider the prefix array (PA), which was used to find repetitions in [71], used within the Z-algorithm for pattern matching [47], used in algorithms by [34], and formally defined as a data structure in [88]. For an n-length T, $PA[i]$ is the length of the longest prefix common between T and $T[i...n]$. The PA computation is powerful for traditional pattern matching applications and, unlike the related *border* array [87], where each $border[i]$ is the length of the longest prefix of $T[1...i]$ that matches a suffix of $T[1...i]$, the PA also supports indeterminate pattern matching [88, 26]. The PA is represented more compactly in [88]. Currently, there is no PA or compact PA for p-strings.

Main Contributions: In this section, we introduce the parameterized prefix array (pPA) and a succinct form, the compact parameterized prefix array ($cpPA$), whose length η is the number of nonzero entries in pPA. This extension is challenging because prefixes and suffixes of p-strings do not necessarily behave like traditional strings. Our constructions make use of the longest previous factor (LPF) data structure, which was primarily used for LZ factorization [36] and has recently motivated much research [35, 37, 38, 19, 23, 24]. We introduced the parameterized longest previous factor ($pLPF$) [19], which for an n-length p-string T, each $pLPF[i]$ for $1 \leq i \leq n$ is the length of the longest parameterized prefix common between $T[i...n]$ and some $T[h...n]$ with $1 \leq h < i$. The array $\mathcal{L}$ (see Definition 2.1.16) keeps the index of this occurrence, i.e. $\mathcal{L}[i] = h$. We show in [23] the power of the $pLPF/LPF$ data structures, as they can be used to construct many data structures such as the longest common prefix (LCP), parameterized longest common prefix ($pLCP$), *border*, parameterized-*border*, etc. In this section, we show a connection between $pLPF$, $\mathcal{L}$, and pPA. We exploit this connection to construct the pPA via the $pLPF$. We then directly construct the $cpPA$ via the $pLPF$ to conserve space. Next, we introduce the indeterminate parameterized string

(ip-string), a p-string with indeterminate symbols, and further define the indeterminate parameterized pattern matching (ip-match) problem, i.e. the matching of ip-strings. We then propose and construct the compact indeterminate parameterized prefix array (*cipPA*). The construction of *cPA* data structures on indeterminate strings is a difficult problem. We note that *cPA* construction is not simply searching for a pattern T in $T[2...n]\$\$...\$$ and recording an element prior to a mismatch. When a mismatch occurs, an efficient search will shift the result to the next best place to continue matching, which can skip necessary *cPA* computations. The addition of indeterminate symbols complicates the *cPA* construction on strings [88, 27] and p-strings; so the *cipPA*, like the *cPA* on indeterminate strings, is constructed in quadratic time in the worst case [88]. Further, the (compact) prefix array is a staple for indeterminate pattern matching because the linear space data structure can be used to obtain all borders of each suffix with indeterminate symbols, which is a problem that is also solved in quadratic time in the very worst case (for strings with wildcards) [57]. Lastly, we identify applications for our data structures in music and software engineering. Our main theoretical results are formalized below.

Theorem 6.1.6. *Given the pLPF and $\mathcal{L}$ on the n-length text T, the pPA on T can be constructed in $O(n)$ time and $O(1)$ extra space.*

Theorem 6.1.8. *Given the pLPF and $\mathcal{L}$ on the n-length text T, the η length cpPA on T can be constructed in $O(n \log \eta)$ time and $O(1)$ extra space.*

Theorem 6.1.14. *For the n-length ip-string T, the cipPA on T can be constructed in $O(n^2)$ time and $O(|\Pi|)$ extra space.*

6.1.1 Background

Baker [12] identifies three types of pattern matching: (1) exact matching, (2) parameterized matching (p-match), and (3) matching with modifications. In this section, we focus on (2) and a new fusion between (2) and (3). The first p-match breakthroughs revolved around suffix structures on the previous (`prev`) encoding. The parameterized suffix tree (p-suffix tree) was proposed and constructed in [11]. Additional improvements to the p-suffix tree are given in [65, 32, 68]. Like the traditional suffix tree [87, 47, 1], the p-suffix tree [11] implementation suffers from a large practical memory footprint. One p-matching solution to address the space problem is the parameterized suffix array (p-suffix array) in [53, 40]. The work

in [21] gives sub-quadratic and near-linear time worst-case p-suffix array constructions. The structural string (s-string) was introduced by Shibuya [86] as an extension to the p-match that incorporates complementary symbols to support structural matching (s-matching) of RNA secondary structures. Originally, the structural suffix tree was constructed in [86] to perform the s-match. The structural suffix array was proposed in [16] as an alternative suffix structure for the s-match.

Other solutions that address the p-match without the space limitations of the p-suffix tree include the parameterized-KMP [6] and parameterized-BM [14], variants of traditional pattern matching schemes. Further, the p-match problem is addressed via the Shift-OR mechanism in [42]. Idury et al. [56] studied a heuristic known as the `pfail` function to address the multiple p-match problem using the traditional Aho-Corasick automaton. This `pfail` function is viewed as the parameterized border array (*p-border*), analogous to the traditional *border* array [87], and has been used for p-matching and studied in a variety of combinatorial problems in [54, 55]. In [22], we construct the structural border (*s-border*) array, a border array with respect to the s-string, and provide the first s-match algorithm without suffix structures.

The aforementioned p-matching is specifically for uncompressed texts. In [8, 7], fully compressed p-matching is proposed on run-length encodings. In [22], we showed how to p-match on run-length encoded strings as an application of the *s-border* array. Compressed p-matching between a compressed text and an uncompressed pattern is addressed in [18]. While the problem of p-matching with mismatches is studied in [49, 8, 7], the existence of indeterminate symbols in p-strings is currently not examined. Even though the *border* array supports standard pattern matching [87] and extensions support p-matching/s-matching [22], the *border* definition fails to support indeterminate symbols [88]. However, the prefix array (PA) [71, 47, 34, 27], though closely related to the *border* array, does support indeterminate pattern matching [88, 26]. Indeterminate pattern matching can also be addressed without the prefix array [47, 52]. In this section, we propose and address indeterminate parameterized pattern matching via an extention of the PA and the compressed prefix array (cPA) [88] – data structures not currently supported for p-strings. In [23], we showed how to build various data structures with the construction of the parameterized longest previous factor ($pLPF$) [19], a p-string data structure related to the longest factor data structures by Crochemore et al. [35, 37, 38]. In this section, we show how to construct our new data structures via the $pLPF$ array. We then introduce p-strings with indeterminate symbols, a form of

indeterminate matching for these p-strings, and construct a prefix array data structure to support the proposed match.

6.1.2 Parameterized Prefix Array

The prefix array is a core data structure in string processing that is used to solve traditional exact string matching problems and can be extended to support indeterminate strings. To address both exact matching problems and more challenging inexact matching scenarios dealing with complex biological strings [86], source code [12], prose (detecting academic plagiarism [11]), and music theory, the p-string and the p-matching problem naturally provide the necessary capabilities. To extend the prefix array data structure to support p-matching, we introduce the parameterized prefix array below by encoding all prefixes under `prev`.

Definition 6.1.1 Parameterized prefix array (pPA): *The pPA for the p-string T of length n is defined for each position $1 \leq i \leq n$, such that $pPA[i]$ is the length of the longest common prefix between the p-suffixes,* $\texttt{prev}(T)$ *and* $\texttt{prev}(T[i...n])$. *i.e. $pPA[i] = \texttt{plcp}(1, i, T)$ for $1 \leq i \leq n$.*

For an example p-string $T = AcbcAzazAb$ from the working alphabets $\Sigma = \{A, B, C\}$ and $\Pi = \{a, b, c, w, x, y, z\}$, we have $pPA = \{10, 0, 0, 0, 6, 0, 0, 0, 2, 0\}$. Following [88], we define the compact parameterized prefix array below to represent the nonzero elements of the pPA.

Definition 6.1.2 Compact parameterized prefix array ($cpPA$): *Given the pPA for the n-length T, the $cpPA$ is a pair of integer arrays $(pPOS, pLEN)$ that represent the left-to-right nonzero elements of pPA, i.e. $pPA[pPOS[i]] = pLEN[i] \geq 1$ for $1 \leq i \leq n$.*

For the example $T = AcbcAzazAb$, we have $pPOS = \{1, 5, 9\}$ and $pLEN = \{10, 6, 2\}$. Note that converting between $cpPA$ and pPA only requires a linear scan. Also note that by first computing $\eta = |pPOS| = |pLEN|$, we can algorithmically decide to construct the compact data structure over the full-fledged prefix array when $2\eta < n$.

Construction

To put the $cpPA$ construction into perspective, first consider the PA/cPA construction on traditional strings with no indeterminate symbols. Two types of solutions are available:

(a) direct constructions of the cPA and PA are provided in [88, 27] and (b) the $\mathcal{B}$ is used to construct the PA in [27]. In this section, we construct our new p-string data structures (pPA and $cpPA$) using different methods.

First, we identify a simple way to construct the $pPA/cpPA$ via parameterized suffix structures. With these data structures, the length of the prefixes common between the prefix of the n length T and all p-suffixes of T is readily available. Consider the parameterized suffix tree (pST) [11]. We walk the path $T[1], T[2], ..., T[n]$ and when at symbol $T[i]$, we say that each p-suffix (say at k) that differentiates at this node have $\texttt{plcp}(1, k, T) = i$. The problem with the pST approach is that like the traditional suffix tree [87, 47, 1], the pST suffers from a large practical footprint. While using the lightweight pST alternative, i.e. the pSA and $pLCP$, will resolve this issue, powerful mechanisms [25] to yield arbitrary `plcp` computations can require heavy processing.

Here, we will identify more interesting constructions of the $pPA/cpPA$ via the parameterized longest previous factor ($pLPF$) [19]. The $pLPF$ is a p-string extension of the longest previous factor (LPF) data structure [35]. Traditionally, the LPF was used for LZ factorization [36]. However, the LPF's utility in string matching has yielded various related data structures [38, 37, 23]. Further, we show in [23] that one $pLPF$ construction algorithm can also construct many data structures, including the LPF, $pLCP$, LCP, *p-border*, and *border*. Now, we look at the contents of the $pLPF$ data structure and identify a connection with the $pPA/cpPA$.

For the n-length text T, each $pLPF[i]$ is defined as the maximum prefix length common between $\texttt{prev}(T[i...n])$ and $\texttt{prev}(T[h...n])$ for some $1 \leq h < i$, where the $pPA[i]$ is the maximum prefix common between $\texttt{prev}(T[i...n])$ and $\texttt{prev}(T)$. So, in the case that $h = 1$, we have $pPA[i] = pLPF[i]$. For other cases, $h > 1$ and thus $0 \leq pPA[i] \leq pLPF[i]$. To determine these other cases, we establish further theory. Recall that $\mathcal{L}[i]$ stores the position in T of the longest previous p-match with the p-suffix at i (see Definition 2.1.16). The observation, formalized below, is that we can use the $\mathcal{L}$ to successively "hop" along the $pLPF$ and compute an element of pPA.

Lemma 6.1.3 *Given the n-length text T, the $pLPF$ and $\mathcal{L}$ on T, let $H_i = \{pLPF[i], pLPF[\mathcal{L}[i]], pLPF[\mathcal{L}^2[i]], ..., pLPF[\mathcal{L}^f[i]]\}$ be the elements visited during the* hop-from-i. *If $\mathcal{L}^f[i] = 1$ then $pPA[i] = \min(H_i)$, else $pPA[i] = 0$.*

Proof Define the *hop-from-i* as the act of starting at $pLPF[i]$, proceeding to $pLPF[\mathcal{L}[i]]$,

proceeding to $pLPF[\mathcal{L}^2[i]]$, etc. until a non-existing entry is accessed (when $i = 0$ or $\mathcal{L}^y[i] = 0$ for $y \geq 1$). Thus, the list of elements visited during the *hop-from-i* is $H_i = \{pLPF[i], pLPF[\mathcal{L}[i]], pLPF[\mathcal{L}^2[i]], ..., pLPF[\mathcal{L}^f[i]]\}$, where $pLPF[\mathcal{L}^f[i]]$ is the last element visited.

In words, the *hop-from-i* is an operation to successively visit the length of longest parameterized prefix common with the current prefix until no such prefix exists. So, $pLPF[i]$ is the length and $\mathcal{L}[i]$ is the location in T of the longest prefix common with $\texttt{prev}(T[i...n])$. By Definition 2.1.16, $pLPF[\mathcal{L}[i]]$ must then be the length and $\mathcal{L}^2[i]$ must be the location of the longest prefix common with $\texttt{prev}(T[i...i + pLPF[i] - 1])$. Further, $pLPF[\mathcal{L}^2[i]]$ must then be the length and $\mathcal{L}^3[i]$ must be the location of the longest prefix common with $\texttt{prev}(T[i...i + pLPF[\mathcal{L}[i]] - 1])$, etc. Thus, H_i is the list of lengths of longest prefixes common with $\texttt{prev}(T[i...n])$ occurring before i in T. So, $\mathcal{L}^f[i] \neq 1$ means that, by Definition 2.1.16, $\texttt{prev}(T[i...n])$ does not share any common symbols with $\texttt{prev}(T)$ and we must have $pPA[i] = 0$. Otherwise, $\mathcal{L}^f[i] = 1$ and we have $pPA[i] > 0$. When $|H_i| = 1$, $pPA[i] = H_i[1]$. When $|H_i| > 1$, we have $\mathcal{L}[i] \neq 1$, and so the longest prefix matching with $\texttt{prev}(T[i...n])$ did not occur directly with $\texttt{prev}(T)$. Thus, the minimum of H_i is the longest prefix common between $\texttt{prev}(T[i...n])$ and $\texttt{prev}(T)$, i.e. $pPA[i] = \min(H_i)$. □

Algorithm 6-1 constructs each element $pPA[i]$ by calling the `hop` function, which uses Lemma 6.1.3 to compute $pPA[i]$. In the following, we formalize the running time of the algorithm.

Theorem 6.1.4 *Given the pLPF and $\mathcal{L}$ on the n-length text T, the pPA on T can be constructed in $O(nu)$ time and $O(1)$ extra space, where $u = \texttt{unique\_positive}(\mathcal{L})$ is the number of unique positive integers in $\mathcal{L}$.*

Proof The correctness of Algorithm 6-1 follows from Lemma 6.1.3. The running time is clearly bounded by the n calls in line 3 to the function `hop`. In `hop`, the running time is bounded by the number of "hops", or iterations of the loop in lines 10-13. Suppose the maximum number of iterations is h. Then the running time is in $O(nh)$. We now bound h. So, $pLPF[i]$ is visited during the first iteration, $pLPF[\mathcal{L}[i]]$ is visited at the second iteration, $pLPF[\mathcal{L}^2[i]]$ at the third iteration, etc. until no such element exists or $\mathcal{L}[i] = 0$. By Definition 2.1.16, we know that each $\mathcal{L}[i]$ points to a previous longest factor or zero when one does not exist, i.e. $0 \leq \mathcal{L}[i] < i$. Since the loop stops when a previous

factor does not exist, then clearly $h \in \texttt{unique\_positive}(\mathcal{L})$. Thus, the running time is in $O(n \times \texttt{unique\_positive}(\mathcal{L}))$, Since the algorithm does not allocate any space beyond the required pPA, the extra space is in $O(1)$. □

The previous approach shows a new and interesting connection between the $pLPF$ and the pPA, albeit the running time is not on par with the traditional PA linear time constructions [88, 27]. The bottleneck of Algorithm 6-1 is the need to construct each $pPA[i]$ by visiting/hopping multiple indices of $pLPF$ via $\mathcal{L}$. Notice that each hop moves to previous elements in the $pLPF$. With this, we consider the relationship between the elements visited during the computations of say $pPA[i]$ and $pPA[j]$. We observe that the computations from the *hop from* j, needed to compute $pPA[j]$, can help solve $pPA[i]$ when $i > j$ and $pLPF[j]$ is visited during the *hop from* i. The observation is formalized in the following lemma.

Lemma 6.1.5 *Given the n-length text T, the $pLPF$ and $\mathcal{L}$ on T, let H_i and H_j be the elements visited during the* hop-from-i *and* hop-from-j*, respectively. If $j = \mathcal{L}[i]$, then $pPA[i] = \min(pLPF[i], pPA[j])$.*

Proof Following Lemma 6.1.3, the list of elements visited during the *hop-from-i* is $H_i = \{pLPF[i], pLPF[\mathcal{L}[i]], pLPF[\mathcal{L}^2[i]], ..., pLPF[\mathcal{L}^f[i]]\}$. Similarly, the list of elements visited during the *hop-from-j* is $H_j = \{pLPF[j], pLPF[\mathcal{L}[j]], pLPF[\mathcal{L}^2[j]], ..., pLPF[\mathcal{L}^g[j]]\}$. By Lemma 6.1.3, we know how to compute $pPA[j]$, i.e. if $\mathcal{L}^g[j] = 1$ then $pPA[j] = \min(H_j)$ and otherwise $pPA[j] = 0$. In the case when $j = \mathcal{L}[i]$, we have $H_i[2...f+1] = H_j$ and thus $pPA[i] = \min(H_i[1], pPA[j]) = \min(pLPF[i], pPA[j])$. □

By Lemma 6.1.5, we can first compute a pPA element and then use the value to compute another. We implement this in Algorithm 6-2. By Definition 6.1.1, we know that $pPA[1] = n$, so the algorithm uses a single left-to-right scan of the $pLPF$ from $i = 2...n$ to compute each element $pPA[2...n]$. When we reach an i with a previous matching prefix, i.e. a nonzero $\mathcal{L}[i]$, we compute $pPA[i]$ by taking the minimum of the *current* longest prefix value, i.e. $pLPF[i]$, and the minimum of the *old* longest prefixes, i.e. the elements visited during the *hop-from-*$\mathcal{L}[i]$ already computed in $pPA[\mathcal{L}[i]]$ (line 6). The running time and extra space complexities are formalized below.

Theorem 6.1.6 *Given the $pLPF$ and $\mathcal{L}$ on the n-length text T, the pPA on T can be constructed in $O(n)$ time and $O(1)$ extra space.*

Algorithm 6-1. Construct pPA via $pLPF$ and $\mathcal{L}$.

```
1  int[] construct_pPA(int pLPF[n],int ℒ[n]){
2     int pPA[n]={0,0,...,0}, i
3     for(i=1 to n) pPA[i]=hop(i,pLPF,ℒ)
4     return pPA
5  }
6  //  Construct plcp(1,i,T)
7  int hop(int i,int pLPF[n],int ℒ[n]){
8     int plcp=n
9     if(i≥2){
10       do{
11          plcp=min(plcp,pLPF[i])
12          i=ℒ[i]
13       }while(i≠0 ∧ ℒ[i]≠0)
14       if(i≠1) plcp=0
15    }
16    return plcp
17 }
```

Algorithm 6-2. Improved pPA construction via $pLPF$ and $\mathcal{L}$.

```
1  int[] construct_pPA_imp(int pLPF[n],int ℒ[n]){
2     int pPA[n]={n,0,...,0}, i, v
3     for(i=2 to n){
4        if(ℒ[i]≥1) v=pPA[ℒ[i]]
5        else v=0
6        pPA[i]=min(v,pLPF[i])
7     }
8     return pPA
9  }
```

Algorithm 6-3. Count length of each array of the $cpPA$.

```
1  int get_cppa_length(char T[n],bit αT[n]){
2     int η=0,i,sym=T[1],type=αT[1]
3     for i=1 to n {
4        if( (type=SIGMA ∧ sym=T[i]) ∨ (type=PI ∧ αT[i]=PI) ) η++
5     }
6     return η
7  }
```

Algorithm 6-4. Construct $cpPA = (pPOS, pLEN)$ via $pLPF$ and $\mathcal{L}$.

```
1  (int[],int[]) construct_cpPA(char T[n],bit αT[n],int pLPF[n],int ℒ[n]){
2     int η=get_cppa_length(T,αT)
3     if(2*η<n){ // when more efficient to construct cpPA
4        int pPOS[η]={1,0,...,0}, pLEN[η]={n,0,...,0}, i, j=1, v, w
5        for(i=2 to n){
6           v=0
7           if(ℒ[i]≥1) {
8              w=binary_search(pPOS,ℒ[i])
9              if(w≥1) v=pLEN[w]
10          }
11          v=min(v,pLPF[i])
12          if(v≥1) {
13             pPOS[j]=i
14             pLEN[j]=v
15             j++
16          }
17       }return (pPOS,ℒ)
18    }else return (construct_pPA_imp(pLPF,ℒ),null)
19 }
```

Proof The correctness of our pPA construction in Algorithm 6-2 follows from Lemma 6.1.5. Since the algorithm uses a single scan of the n-length $pLPF$ and $\mathcal{L}$ arrays and does not allocate any arrays in addition to the pPA, then the algorithm requires $O(n)$ time and $O(1)$ extra space. □

At this point, we have constructed the pPA. We can very easily generate the $cpPA$ by running Algorithm 6-2 to obtain the pPA first and making a $cpPA$ element for each $pPA[i] > 0$. While this has the same $O(n)$ construction time as Algorithm 6-2, we have used the n-length pPA to construct the η-length $cpPA$. Thus, construction of the $cpPA$ in this manner will use $O(n)$ extra space.

Theorem 6.1.7 *Given the pLPF and $\mathcal{L}$ on the n-length text T, the cpPA on T can be constructed $O(n)$ time and $O(n)$ extra space.*

To resolve the excessive extra space of Theorem 6.1.7, in Algorithm 6-4, we construct the $cpPA = (pPOS, pLEN)$ data structure directly, i.e. without first allocating/constructing the pPA. The algorithm first, in line 2, computes the length η of the $cpPA$ via Algorithm 6-3. When $T[i]$ is a constant, η is simply the count of all exact $T[i]$ symbols in T, which is necessary to begin a p-match. When $T[i]$ is a parameter, η is the total of all parameter

symbols in T because any length-1 prefix beginning with a parameter will p-match with any other parameter by Definition 2.1.4. Following the η computation, line 3 determines, for the given T, if the compact pPA representation is better than just pPA, i.e. when $2\eta < n$. If not, we compute the pPA in the regular way (line 18). Otherwise, we allocate the arrays $pPOS$ and $pLEN$ with the known first entries (line 4), i.e. $pPOS[1] = 1$ and $pLEN[1] = n$. The remainder of the algorithm (lines 5-17) operates in the same manner as Algorithm 6-2 using Lemma 6.1.5 (line 11); the key modification is replacing the n-length pPA with the η-length arrays $pPOS$ and $pLEN$, i.e. removing the excessive extra space of Theorem 6.1.7. In line 8, we need to find the location in $cpPA$ of $pPA[\mathcal{L}[i]]$, so we first find the location of $\mathcal{L}[i]$ in $pPOS$ via a binary search in order to find the position of the element $pPA[\mathcal{L}[i]]$ in $pLEN$. We compute the current $cpPA$ value v in line 11 (Lemma 6.1.5). Finally in lines 12-16, we only store the current value v in $pPOS$ and $pLEN$ if $v \geq 1$. The complexities of the algorithm follow.

Theorem 6.1.8 *Given the pLPF and $\mathcal{L}$ on the n-length text T, the η length cpPA on T can be constructed in $O(n \log \eta)$ time and $O(1)$ extra space.*

Proof The correctness of Algorithm 6-4 follows from that of Algorithm 6-2 and Lemma 6.1.5. We now consider the time complexity. Since line 2 is an $O(n)$ operation, the running time is bounded by the time of line 8, which is called $(n-1)$ times. Let `binary_search`(A, a), in $O(\log |A|)$ time, return the location of a in the sorted array A or return 0 when a does not exist. Since $|pPOS| \in O(\eta)$, then line 8 requires $O(\log \eta)$ time and the entire algorithm is bounded by $O(n \log \eta)$. Lastly, the algorithm operates with $O(1)$ extra space since only the required $pPOS$ and $pLEN$ arrays are allocated. □

In passing, we note that by manipulating the alphabets, i.e. using only the Σ alphabet, all of our algorithms and results apply to the PA and cPA for traditional strings.

6.1.3 A Prefix Array for Indeterminate p-strings

In this section, we introduce a variation of the p-match with indeterminate symbols and propose/construct analogous prefix array data structures. The notion of indeterminate strings and indeterminate pattern matching is traditionally described in the following way [52]. Indeterminate strings contain symbols from Σ and *holes*. These holes are represented by another symbol, say a placeholder Z. Each placeholder corresponds to a subset of symbols

from Σ, i.e. $z \subseteq \Sigma$. Two indeterminate strings match when all of the corresponding symbols match in the following way: two symbols from Σ match exactly, two identical holes can match, or a hole Z can match with any symbol $s \in z$.

For parameterized strings, incorporating indeterminate symbols can take many forms. We can allow each hole to correspond to symbols from only Σ, only Π, or $\Sigma \cup \Pi$. The way this is defined can impact how the holes and symbols are matched. We can allow holes and symbols to match in the traditional way. Alternatively, we can enforce another parameter bijection on only the indeterminate matches within the strings. While applications may exist for each variation, we will propose the most direct way to introduce indeterminate symbols into p-strings, which is useful in various applications (discussed later). Now, we define the indeterminate parameterized string, which consists of constants from Σ, parameters from Π, and holes from $I = \{\widehat{I^{\Sigma}}, \widehat{I^{\Pi}}\}$.

Definition 6.1.9 Indeterminate parameterized string (ip-string): *An ip-string is a production T of length n from $(\Sigma \cup \Pi \cup I)^*\$$, where $I = \{\widehat{I^{\Sigma}}, \widehat{I^{\Pi}}\}$.*

To know the alphabet of each symbol in the n-length ip-string T, we redefine the p-string α encoding (Definition 2.1.2). For each $1 \leq j \leq n$, (a) $i\alpha(T)[j] = SIGMA$ if $T[j] \in \Sigma$, (b) $i\alpha(T)[j] = PI$ if $T[j] \in \Pi$, (c) $i\alpha(T)[j] = \widehat{I^{\Sigma}}$ if $T[j] = \widehat{I^{\Sigma}}$, and (d) $i\alpha(T)[j] = \widehat{I^{\Pi}}$ if $T[j] = \widehat{I^{\Pi}}$.

Two ip-strings S and T form an indeterminate parameterized match (ip-match) when a p-match occurs between the string symbols from Σ and Π, and the holes match or the hole $\widehat{I^{\Sigma}}$ matches a symbol in Σ or $\widehat{I^{\Pi}}$ matches a symbol in Π. Definition 6.1.10 formalizes the ip-match.

Definition 6.1.10 Indeterminate parameterized matching (ip-match): *An ip-match exists between the n-length ip-strings S and T if 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. $S[i] = S[i-q]$ iff $T[i] = T[i-q]$ for every $1 \leq q < i$ with $S[i], T[i] \in \Pi$
3. $(S[i] = T[i] = \widehat{I^{\Sigma}}) \vee (S[i] = \widehat{I^{\Sigma}} \wedge T[i] \in \Sigma) \vee (T[i] = \widehat{I^{\Sigma}} \wedge S[i] \in \Sigma)$
4. $(S[i] = T[i] = \widehat{I^{\Pi}}) \vee (S[i] = \widehat{I^{\Pi}} \wedge T[i] \in \Pi) \vee (T[i] = \widehat{I^{\Pi}} \wedge S[i] \in \Pi)$

For example, let $\Sigma = \{A, B, C\}$ and $\Pi = \{a, b, c, v, w, x, y, z\}$, $S = abb\widehat{I^{\Pi}}Aba\widehat{I^{\Sigma}}$, $T = wxx\widehat{I^{\Pi}}Axw\widehat{I^{\Sigma}}$, $U = bcczAcbB$, $V = bcccAcbB$, $W = bcczCcbB$, and $X = ccczAcbb$. The

ip-strings S, T, U, and V are all ip-matches. Our definition of the ip-match generalizes to many other matching schemes. When no holes are used, the ip-match becomes the p-match. When no parameters are used, the ip-match becomes exact matching with wildcards. When no holes and no parameters are used, the ip-match becomes the exact match.

Unlike the p-match (see Definition 2.1.4), we cannot perform the ip-match by creating a single encoding scheme and blindly comparing the encodings mainly because we do not know if a symbol will be matched with another symbol or with a hole, i.e. a factor that will change the parameter distances of `prev` and thus the bijection. In Algorithm 6-5 (`iplcp`), we compute the longest common prefix between any two suffixes at A and B in an ip-string T by considering each symbol in terms of Definition 6.1.10 (1)-(4).

Lemma 6.1.11 *For the ip-string T of length n and $i\alpha T = i\alpha(T)$, the length of the ip-match between $T[A...n]$ and $T[B...n]$ is computed in $O(n - \max(A, B))$ time and $O(|\Pi|)$ extra space.*

Proof The `iplcp` in Algorithm 6-5 compares at most $O(n - \max(A, B))$ symbols via (1)-(4) of Definition 6.1.10 each in $O(1)$ time with the notated alphabet sets in $i\alpha T$. The parameter bijection (2) is supported by two auxiliary size $|\Pi|$ arrays. Thus, $O(n - \max(A, B))$ time and $O(|\Pi|)$ extra space is required. □

In the following, we define the prefix array and compact prefix array for ip-strings.

Definition 6.1.12 Indeterminate parameterized prefix array ($ipPA$): *The $ipPA$ for the p-string T of length n is defined for each position $1 \leq i \leq n$, such that $ipPA[i]$ is the length of the longest ip-match between the $T[1...n]$ and $T[i...n]$, i.e. $ipPA[i] = \texttt{iplcp}(T, i\alpha(T), 1, i)$.*

Definition 6.1.13 Compact indeterminate parameterized prefix array ($cipPA$): *Given the $ipPA$ for the ip-string T, the $cipPA$ is a pair of integer arrays $(ipPOS, ipLEN)$ that represent the left-to-right nonzero elements of $ipPA$, i.e. $ipPA[ipPOS[i]] = ipLEN[i] \geq 1$.*

As with the cPA, we want to directly construct the $cipPA$ without the $ipPA$. Algorithm 6-4 cannot construct the $cipPA$ for ip-strings because (1) the length of the $cipPA$ is different due to the presence of indeterminate symbols, (2) only p-matching is supported, and (3) the $pLPF$ only supports standard p-strings.

We modify the η calculation `get_cpPA_length` (Algorithm 6-3) to `get_cipPA_length` (Algorithm 6-6), which handles the indeterminate cases. For instance, when $T[1]$ is a hole, it

can ip-match another like hole or a symbol in the respective alphabet. During each iteration of the scan to find η, when a new length-1 ip-match is found we also find the next value of $ipPOS$. For clarity, we show this $ipPOS$ generation in `get_nextval` of Algorithm 6-6. Our $cipPA$ construction in Algorithm 6-5 first computes η and $ipPOS$, and uses the `iplcp` (Lemma 6.1.11) to ip-match T with each element of $ipPOS$.

Theorem 6.1.14 *For the n-length ip-string T, the $cipPA$ on T can be constructed in $O(n^2)$ time and $O(|\Pi|)$ extra space.*

Proof Our $cipPA$ construction (Algorithm 6-5) computes the length η of the $cipPA$, computes the $ipPOS$, and uses the `iplcp` to perform the ip-matching between T and the individual $ipPOS$. The line 2 computation of η and $ipPOS$ (respectively, `get_cpPA_length` and `get_nextval` in Algorithm 6-6) are scans that clearly require $O(n)$ time. For $h = 1$ to η, we loop through all positions of the $ipPOS$ to compute. Each element is computed by ip-matching (line 4) between T and $T[ipPOS[h]...n]$ via `iplcp`, which requires $O(n-ipPOS[h])$ time and $O(|\Pi|)$ extra space by Lemma 6.1.11. Since $\eta \leq n$, the algorithm requires $O(n^2)$ time and $O(|\Pi|)$ extra space in the worst case. □

At this point, we note that while the PA and cPA are constructed in linear time [88, 27] and the new pPA and $cpPA$ are constructed in linear time in the worst case, the addition of indeterminate symbols introduces challenges to the theoretical worst case complexity. First, we highlight that the cPA construction is not as easy as performing an efficient/fast search of T in $T[2...n]\$\$...\$$ and reporting a cPA element prior to a mismatch. Simply put, when a mismatch occurs, we shift the result to the next best place to continue matching, which can occur after smaller matches; these smaller matches are skipped during efficient pattern matching, but are valid cPA elements. Even though we can efficiently construct the cPA for strings and p-strings, the introduction of indeterminate symbols adds complications to the solution and worst case running time. The compact prefix array data structures are powerful tools that can be used to find all borders of suffixes of a string with indeterminate symbols. This difficult computation requires quadratic time in the very worst case and is linear in the average case for traditional strings with wildcards [57]. For traditional strings with indeterminate symbols, the cPA is constructed by [88] in quadratic time in the worst case. In [88], it was shown experimentally that computing the cPA for indeterminate strings is faster with a standard brute-force approach than when employing more sophisticated

shifting mechanisms that do not improve the worst case time complexity. Unlike the LPF definition where a match can occur between a suffix and a previous suffix in the string, the definition of the prefix array is more restrictive, i.e. requiring a match between a suffix and *only* prefixes of the text. Intuitively, this restriction will play a role in the practical running time. A future research problem is to investigate alternate approaches to improve the worst case theoretical construction of prefix array-based data structures on indeterminate strings.

6.1.4 Applications

The *border* cannot be used for strings with indeterminate symbols because its representation and construction are based on matching transitivity. The prefix array does allow us to handle indeterminate symbols because we can directly obtain the necessary longest common prefixes between these strings – the ultimate shifting mechanism. To find a pattern P in the text T, whether a traditional string, p-string, or ip-string, we can find all instances of the m-length P in the n-length T, where both can have indeterminate symbols, by constructing our $cipPA = (ipPOS, ipLEN)$ on $P \circ \# \circ T$ with $\# \notin \Sigma \cup \Pi$. For any $ipLEN[i] = |P|$, we report that P occurred in T at position $(i - |P| - 1)$. Due to the $\#$ symbol, this will require $O(nm)$ time in the very worst case (and $O(|\Pi|)$ extra space), but uses the $cipPA$ data structure to solve the difficult scenario where P and T can have any number of holes. Indeterminate pattern matching and even matching with wildcards suffer from similar worst case time bounds and can limit holes to appear exclusively in the pattern P [47].

An advantage of the $cipPA$, like the cPA as noted in [88], is that it can be used alternatively as a shifting mechanism for pattern matching. The PA based data structures, in at most n elements, represent all of the borders of suffixes of a string and indeterminate string; this is a difficult problem that is solved in $O(n^2)$ time in the very worst case in [57]. Our ip-string data structure can be used as a shifting mechanism in the following way. Compute the $cipPA = (ipPOS, ipLEN)$ for P. From left-to-right in T, independently ip-match the symbols until we find a mismatch. Suppose that $P[1...k]$ and $T[i...i+k-1]$ are ip-matches for the max k; there is a mismatch after k symbols. We can then scan the compact $ipLEN[1...k]$ until we find the smallest j such that $ipPOS[j] + ipLEN[j] - 1 \geq k$. We then use this to determine the longest border of $P[1...k]$ and attempt rematching. This method is a variation of matching with the *border* arrays [22] using the prefix array representation.

Due to the multiple alphabets and sophisticated matching scheme, p-string theory is a

Algorithm 6-5. The $cipPA = (ipPOS, ipLEN)$ construction.

```
1  (int[],int[]) construct_cipPA(char T[n],char iαT[n]){
2     int η=get_cipPA_length(T,iαT),ipPOS[η]=get_nextval(T,iαT,η),ipLEN[η],k
3     for h=1 to η
4        k=iplcp(T,iαT,1,ipPOS[h])
5        ipLEN[h]=k
6     }return (ipPOS,ipLEN)
7  }
8
9  int iplcp(char T[n],char iαT[n],int A,int B){
10    boolean valid_sym=true
11    int pmap1[|Π|]={0,...,0},pmap2[|Π|]={0,...,0}
12    int k=1,sym1,type1,sym2,type2
13    while(valid_sym ∧ k≤n){
14       sym1=T[k+A−1], type1=iαT[k+A−1]
15       sym2=T[k+B−1], type2=iαT[k+B−1]
16       if(type1=type2=SIGMA ∧ sym1=sym2) k++ // Definition 6.1.10 (1)
17       else if(type1=type2=PI ∧ pmap1[sym1]=pmap2[sym2]){
18          // Definition 6.1.10 (2)
19          pmap1[sym1]=pmap2[sym2]=k
20          k++
21       }
22       else if( (type1=type2=$\widehat{I^{\Sigma}}$) ∨ (type1=$\widehat{I^{\Sigma}}$ ∧ type2=SIGMA) ∨
23                (type2=$\widehat{I^{\Sigma}}$ ∧ type1=SIGMA) ) k++ // Definition 6.1.10 (3)
24       else if( (type1=type2=$\widehat{I^{\Pi}}$) ∨ (type1=$\widehat{I^{\Pi}}$ ∧ type2=PI) ∨
25                (type2=$\widehat{I^{\Pi}}$ ∧ type1=PI) ) k++ // Definition 6.1.10 (4)
26       else valid_sym=false
27    }return k−1
28 }
```

Algorithm 6-6. Preprocessing prior to $cipPA$ construction.

```
1  int get_cipPA_length(char T[n],char iαT[n]){
2     int η=0,j,sym=T[1],type=iαT[1]
3     for j=1 to n {
4        if( ((type=SIGMA ∨ type=$\widehat{I^{\Sigma}}$) ∧ (sym=T[j] ∨ iαT[j]=$\widehat{I^{\Sigma}}$)) ∨
5            ((type=PI ∨ type=$\widehat{I^{\Pi}}$) ∧ (iαT[j]=PI ∨ iαT[j]=$\widehat{I^{\Pi}}$)) ) η++
6     }return η
7  }
8
9  int[] get_nextval(char T[n],char iαT[n],int η){
10    int j,curr=1,sym=T[1],type=iαT[1],nextval[η]
11    for j=1 to n {
12       if( ((type=SIGMA ∨ type=$\widehat{I^{\Sigma}}$) ∧ (sym=T[j] ∨ iαT[j]=$\widehat{I^{\Sigma}}$)) ∨
13           ((type=PI ∨ type=$\widehat{I^{\Pi}}$) ∧ (iαT[j]=PI ∨ iαT[j]=$\widehat{I^{\Pi}}$)) ){
14          nextval[curr]=j
15          curr++
16       }
17    }
18    return nextval
19 }
```

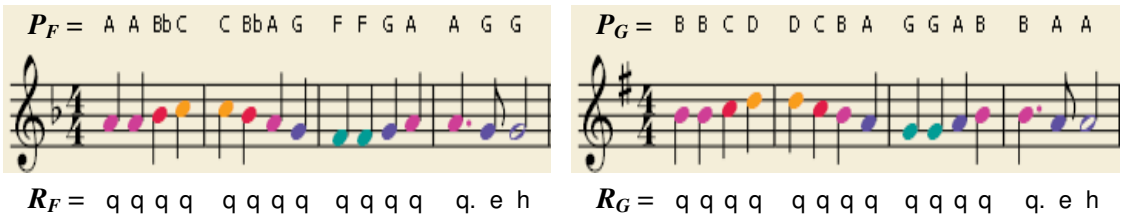

Figure 6.1: Transpositions of Beethoven's *Ode to Joy*, where P_F are pitches in the key of F Major and P_G are pitches in G Major. R_F and R_G are the respective rhythm sequences with quarter (q), dotted quarter ($q.$), eighth (e), and half (h) notes.

very powerful field that is utilized to solve all standard string problems and to naturally address various other problems. In the early beginnings of the p-string, the core applications were source code cloning and detecting plagiarism in academia [11], which make use of the Π alphabet as program variable names and particular words, respectively. A decade later, the p-string was used to help identify the structural similarity of complex RNA secondary structures [86]. More recently, the p-string was applied as a transformation for data compression [18]. We now identify music as a new area for p-string theory.

Music is organized sound over time. A score, or sheet music, represents music on paper as a collection of notes and other symbols/annotations. At the basic level, notes will have a pitch and a beat. There are 12 pitches that are repeated at different octaves: $\mathbb{P} = \{A, A\sharp/B\flat, B, C, C\sharp/D\flat, D, D\sharp/E\flat, E, F, F\sharp/G\flat, G, G\sharp/A\flat\}$. For most musical arrangements, the pitch alphabet will be between A_0 and C_8 (with p_o as pitch p at octave o), the respective low and high pitches of the piano. A musical composition is a collection of several pitches with beats. The pitches share a tonic (root) note and a key signature, i.e. a collection of pitches with respect to the tonic. The pitches in the key signature control the relative sound of a composition. Each pitch is to elapse a certain time determined by the beat, which is described by the time signature of the piece. A problem in music theory and practice is the need to transpose a composition from one key signature to another. Simply put, music is transposed for multiple instruments (pitched differently) to sound correct together in a band/orchestra setting or music can be transposed to help better support the range of an instrument/voice. Thus, the same music can exist in many key signatures with variations, introducing challenges to music analysis and extraction. The method to transpose between two major or minor key signatures say K_1 and K_2 is to find the interval between K_1 and K_2 and move each pitch in the music up/down that exact interval. Computationally, we can count the number of semitones $\mathcal{S}$ that make up this interval and appropriately add/subtract

$\mathcal{S}$ from each note in the sheet music. If we let $\Sigma = \emptyset$ and $\Pi = \mathbb{P}$, then there is clearly a one-to-one correspondence, and thus a p-match, between the pitches of transposed pieces. A sample transposition of Beethoven's *Ode to Joy* between F Major and G Major is displayed in Figure 6.1. Observe that $\texttt{prev}(P_F) = \texttt{prev}(P_G) = 010013500135131$ with pitches in the same octave $\Pi = \{F, G, A, B\flat, B, C, D\}$. For simplicity, we can keep the same rhythm, i.e. time signature; so $R_F = R_G$. However, the music can be written in different, but equivalent time signatures, again requiring a p-match to identify. Our newly proposed data structures for the ip-match can be of assistance in various music scenarios with indeterminate symbols. For instance, when music is originally drafted, sometimes a composer may not be certain of a note or, it may be possible that two notes sound just as pleasing to the ear. These segments of sheet music will have indeterminate symbols. Indeterminate symbols may also be present in digital music converted from audio due to background noise. Also, indeterminate symbols can help us query and extract different variations of a musical segment.

Various other applications of the ip-match are possible. One of the problems of trying to protect software and detect software infringement is the difficulty of dealing with multiple languages. When we preprocess these texts for p-matching, we will set Σ as the syntactic elements and Π as the identifiers, and replace all multiple letter entries by an agreed symbol. In order to work with multiple languages, we must convert to a uniform language – the problem here is: the lower level the language, the less natural language exists and so, it can be more difficult to say that code cloning indeed exists. With the ip-match, we can perform the match on the original processed source code. Suppose are looking for a proprietary assignment statement in the C language, `G=299*sqrt(F+T)+Q*T/F;`. If this statement were present in an unauthorized clone of our system, it would appear in Java as `Y=299*Math.sqrt(Z+T)+R*T/Z;`, in Pascal as `A:=299*sqrt(B+C)+D*C/B;`, and in BASIC as `LET Z=299*SQR(Y+X)+W*X/Y`. We can use the ip-match to find all of these clones by searching for the pattern `Y` $\widehat{I^{\Sigma}}$ `299*` $\widehat{I^{\Pi}}$ `(Z+T)+R*T/Z`.

6.1.5 Conclusions

While the p-match problem has been investigated for many years, originating in STOC'93 [11] and SODA'95 [14], the p-match is still lacking the capabilities of the well-studied traditional exact match. Even though forms of inexactness have been applied to the p-match [49, 8, 7], the area has limited research. The ability to apply further inexactness to the

p-match can address sophisticated applications. This motivates our study of indeterminate symbols, indeterminate pattern matching, and the p-string/p-match, a previously unstudied problem. Leading to our study of indeterminate symbols and parameterized strings, we study the prefix array (PA) and its succinct representation, the compact prefix array (cPA) [88], which has been used for indeterminate pattern matching for traditional strings. In this section, we extend the PA/cPA framework for p-strings and provide a linear time construction via a new, novel use of the parameterized longest previous factor ($pLPF$) [19] data structure. The results generalize to also yield a new linear time construction of the PA/cPA data structures. We introduce the indeterminate parameterized string (ip-string) and a form of indeterminate pattern matching (ip-match) on ip-strings. Next, we propose the PA and cPA for ip-strings and show a construction that executes in quadratic time in the worst case. This worst case result compares to other related algorithms on traditional strings. The cPA construction for traditional strings with indeterminate symbols also has a quadratic worst case time bound [88]. Further, prefix based data structures compactly encode the borders of all suffixes of a string and this computation also requires quadratic time in the worst case, for traditional strings with wildcards [57]. Nonetheless, the ip-match can address interesting applications with indeterminate symbols in both patterns and texts of RNA secondary structures [86], source code [12], prose [11], and now music compositions, as identified in this section. Lastly, we note that our results also apply to traditional strings when the alphabets only contain constant symbols.

6.2 The Equivalence Parameterized Match

The area of pattern matching has evolved to help support a multitude of applications requiring different forms of string analyses. Traditional exact matching is useful in many applications, such as database retrieval. Approximate pattern matching, a helpful scheme in spell-checking applications, is much less restrictive as two strings S and T can vary by at most k symbols, which may be added, modified, or deleted to transition between S and T. Other variations of inexact matching exist, such as matching with wildcards, indeterminate matching, and degenerate matching, and are helpful in applications from biology to cryptography. However, there are a number of applications that require a pattern matching scheme between the strict exact matching and the less-stringent inexact matching. The parameterized match (p-match) [11] helps solve some of these applications that require

a combination of traditional exact matching and inexact matching with bijections.

A parameterized string (p-string) is composed of symbols from a constant alphabet Σ and a parameter alphabet Π. Two p-strings S and T are said to be a parameterized match (p-match) if (1) all of the constant symbols match exactly, i.e. $S[i], T[i] \in \Sigma \ \wedge \ S[i] = T[i]$, and (2) the parameter symbols form a bijection between S and T, i.e. for $S[i], T[i] \in \Pi$ and possibly $S[i] \neq T[i]$, where parameter $S[i]$ reappears in S say at position j, then also parameter $T[i]$ reappears in T at position j. This scheme is needed in applications that involve detecting source code cloning and deep software analysis [12, 98], identifying academic plagiarism [11], and determining the structural equivalence of RNA sequences as a component of the structural match (s-match) [86]. In some cases, we may need to add a level of inexactness to the p-match. Suppose that we are searching natural language source codes in different programming languages and at the token level, we want to treat all of the superfluous assignment operators `:=`, `=:`, `=`, `<-`, etc. in the same manner. Suppose that are searching vocal music compositions in the bass and baritone range, then we can consider all corresponding pitches above high-C in the same manner. Thus, we need to employ the p-match with a group of constant or parameter symbols that behave in the same way, i.e. to add a form of indeterminacy to the p-match.

Main Contributions: In this section, we propose a form of indeterminate p-matching known as the equivalence parameterized match (e-match) on equivalence strings (e-strings). We show how to address the e-match using the parameterized string (p-string) suffix array framework. Before this section, the main bottleneck of approaching the p-match via the parameterized suffix array (pSA) and parameterized longest common prefix ($pLCP$) was the theoretical worst-case quadratic time needed to *directly* construct the data structures, without the need of the parameterized suffix tree (pST). The $pSA/pLCP$ open problem posed by [53] is quoted below.

> "From a theoretical viewpoint, a naïve radix sort would give an $O(n^2)$ time algorithm. It is an open problem if there exists better worst-case time algorithms for p-suffix array [pSA] construction. Similarly, for PLCP [$pLCP$] arrays, the P-Kasai algorithm runs in $O(n^2)$ time. However, we do not know if this bound is tight, or if there exist linear time algorithms for PLCP [$pLCP$] array construction."

The fact that the parameterized suffix (p-suffix) is dynamic and invalidates the properties of traditional suffixes makes efficient construction of the $pSA/pLCP$ quite difficult. Further,

it is challenging to tightly bound the construction analysis due to these dynamic p-suffixes. In fact, following the previous quote in [53], it is difficult to bound novel p-string algorithms more tightly than naïve solutions! Nonetheless, we show an improved pSA construction in [21], but with practical considerations. Also, we construct the $pLPF$ in linear expected time in [19]. However, the theoretical worst-case bound for the $pSA/pLCP$ is still a problem.

In this section, we initially identify a relationship between p-suffixes and then propose the parameterized cover (p-cover). This newfound relationship and p-cover are exploited to construct the pSA in stages. The idea is extended to yield $O(n|\Pi|)$ worst-case constructions of the pSA and $pLCP$ for an n-length p-string from the constant alphabet Σ and parameter alphabet Π, breaking the previous $O(n^2)$ time barrier. Our main results, formalized below, are also applicable to the traditional suffix array (SA) and longest common prefix (LCP) array.

Theorem 6.2.18. *Given an n-length e-string E from Σ and Π with u_Π unique parameters, we can construct the e-suffix array (eSA) in $O(nu_\Pi)$, or generally $O(n|\Pi|)$, time and extra space.*

Theorem 6.2.20. *Given an n-length e-string E from Σ and Π with u_Π unique parameters, the RMQ r on LCP_{prE} where $prE = \texttt{prev}(\texttt{rename}(E))$, and the first parameters data structure F on E, the equivalence longest common prefix array ($eLCP$) is constructed in $O(nu_\Pi)$ time and extra space, or generally $O(n|\Pi|)$ time and extra space.*

6.2.1 Background

Baker [12] identifies three types of pattern matching: (1) exact matching, (2) parameterized matching (p-match), and (3) matching with modifications. In this section, we focus on (2) and a new fusion between (2) and (3). The p-match generalizes exact matching with the parameterized string (p-string), composed of symbols from a constant symbol alphabet Σ and a parameter alphabet Π. A p-match exists between a pair of p-strings S and T of length n when (1) the constant symbols $\sigma \in \Sigma$ match and (2) there exists a bijection of parameter symbols $\pi \in \Pi$ between S and T. The first p-match breakthroughs revolved around suffix structures on the p-string previous (`prev`) encoding. The parameterized suffix tree (pST) was proposed and constructed in $O(n(|\Pi| + \log(|\Pi| + |\Sigma|)))$ time in [11]. Additional improvements to the pST are given in [65, 32, 68]. Like the traditional suffix tree [87, 47, 1],

the pST [11] implementation suffers from a large practical memory footprint.

One p-matching solution to address the space problem is the parameterized suffix array (pSA) and parameterized longest common prefix ($pLCP$) array. To avoid the memory footprint of the pST, the pSA and $pLCP$ must be constructed directly, that is, without the assistance of the pST. Initially, the pSA and $pLCP$ were defined for p-strings from a binary alphabet and constructed directly in linear time [40]. The same group [53] later defined the pSA and $pLCP$ for p-strings from any alphabet and proposed direct constructions requiring quadratic time in the worst-case. The question of whether or not sub-quadratic direct constructions of the pSA and $pLCP$ exist was also posed as an open problem in [53]. A transformative approach to pSA construction via arithmetic codes was introduced in [20]. Later in [21], the arithmetic coding approach was exploited further, with practical considerations, to construct the pSA in sub-quadratic and near-linear time in the worst-case. The parameterized longest previous factor ($pLPF$) data structure was studied in [19] and a connection was made between the $pLPF$ and the construction of other p-string data structures, including the $pLCP$, in expected linear time for particular p-strings [23]. The structural string (s-string) was introduced by Shibuya [86] as an extension to the p-match that incorporates complementary symbols to support matching RNA secondary structures. Originally, the structural suffix tree (sST) was constructed in [86] to perform the s-match. The structural suffix array (sSA) and structural longest common prefix ($sLCP$) array were proposed in [16] as an alternative suffix structure for the s-match. Historically, algorithms directly constructing the pSA and sSA have suffered theoretical time lags due to the dynamic nature of encoded suffixes. This is also true for algorithms constructing data structures such as $pLCP$ and $sLCP$. In this section, we propose a novel method to address the challenges of directly constructing the pSA and $pLCP$ for an n-length p-string from Σ and Π in $O(n|\Pi|)$ time, breaking the $O(n^2)$ previous worst-case theoretical time barrier.

Other solutions that address the p-match without the space limitations of the pST include the parameterized-KMP [6] and parameterized-BM [14], variants of traditional pattern matching schemes. These particular approaches use a variety of heuristics for shifting the comparisons to p-match efficiently. Further, the p-match problem is addressed via the Shift-OR mechanism in [42]. Idury et al. [56] studied a heuristic known as the `pfail` function to address the multiple p-match problem using the traditional Aho-Corasick automaton. This `pfail` function is viewed as the parameterized border array (*p-border*), analogous to the traditional *border* array [87], and has been used for p-matching and studied in a variety

of combinatorial problems in [54, 55]. A relationship between the *pLPF* and the *p-border* was shown in [23]. In [22], we construct the structural border (*s-border*) array, a border array with respect to the s-string, and provide the first s-match algorithm without suffix structures.

The aforementioned p-matching is specifically for uncompressed texts. In [8, 7], fully compressed p-matching is proposed on run-length encodings. In [22], we show how to p-match on run-length encoded strings as an application of the *s-border* array. Compressed p-matching between a compressed text and an uncompressed pattern is addressed in [18]. The problem of p-matching with mismatches is studied in [49, 8, 7]. In this section, we propose the equivalence parameterized match (e-match) to integrate indeterminate symbols with the p-match; we address the e-match via suffix structures.

6.2.2 The Equivalence Parameterized Match

The p-match is helpful in applications involving source code analysis [12], academic plagiarism [11], and RNA structural similarity [86]. In these applications, there are situations where we may wish to treat a group of symbols, whether constants or parameters, in the same way. To implement a form of indeterminacy with the p-match, we propose the equivalence parameterized match (e-match). Discussion on the e-match begins with formalizing the notion of an equivalence class for alphabets defined for pattern matching, which closely follows from mathematics.

Definition 6.2.1 Equivalence class (e-class): *Given an alphabet* $A = \{a_1, a_2, ..., a_{\mathcal{A}}\}$*, an equivalence class on* A *is a partition of one or more symbols from* A*, say* $\widehat{A} = \{a_i, a_j, ..., a_k\}$*, where every pair of symbols* $a_y, a_z \in \widehat{A}$ *are considered to be* inexact matches*, i.e.* $a_y \approx a_z$*. Since an e-class is a partition of symbols from* A*, then any two e-classes* $\widehat{A}_1$ *and* $\widehat{A}_2$ *from* A *are such that* $\widehat{A}_1 \cap \widehat{A}_2 = \emptyset$*.*

Note that from the preliminaries, the $\approx$ operator, like the $=$ operator for symbols/strings, is reflexive, symmetric, and transitive. Using the previous definition, we now define equivalence alphabet sets for our constant alphabet Σ and parameter alphabet Π to group symbols by e-classes.

Definition 6.2.2 Equivalence alphabet sets (e-alphabet sets): *For the constants* Σ*, we denote the e-alphabet set as a set of e-classes from* Σ*, i.e.* $\widehat{\Sigma} = \{\widehat{\Sigma}_1, \widehat{\Sigma}_2, ..., \widehat{\Sigma}_{\widehat{\mathcal{S}}}\}$*. For*

the parameters Π*, we denote the e-alphabet set as a set of e-classes from* Π*, i.e.* $\widehat{\Pi} = \{\widehat{\Pi}_1, \widehat{\Pi}_2, ..., \widehat{\Pi}_{\widehat{p}}\}$.

In other words, the e-alphabets $\widehat{\Sigma}$ and $\widehat{\Pi}$ are respectively partitions of Σ and Π of symbols that are considered equivalent. From Definition 6.2.1, we can also have any or all $|\widehat{\Sigma}_i| = 1$ and $|\widehat{\Pi}_j| = 1$ to model the standard scenario with no indeterminate symbols. In the following, we define the equivalence parameterized string as a p-string with the e-alphabets $\widehat{\Sigma}$ and $\widehat{\Pi}$.

Definition 6.2.3 Equivalence parameterized string (e-string): *An e-string* E *is an* n*-length p-string from* $(\Sigma \cup \Pi)^*\$$*, where each* $E[i] \in \Sigma$ *belongs to an e-class* $\widehat{\Sigma}_i \in \widehat{\Sigma}$ *and each* $E[i] \in \Pi$ *belongs to an e-class* $\widehat{\Pi}_j \in \widehat{\Pi}$.

At this point, we highlight that, in addition to composing the e-string, the user is free to define the p-string alphabet Σ and Π, in addition to the e-alphabets $\widehat{\Sigma}$ and $\widehat{\Pi}$ (following Definition 6.2.2) as desired for the application considered. Below, we define the equivalence parameterized match to formalize the way that the various alphabets and e-string symbols are oracled to determine a match.

Definition 6.2.4 Equivalence parameterized matching (e-match): *An e-match exists between pair of e-strings* E_1 *and* E_2 *with* $n = |E_1|$ *if and only if* $|E_1| = |E_2|$ *and each* $1 \leq i \leq n$ *corresponds to one of the following:*

1. $E_1[i], E_2[i] \in (\Sigma \cup \{\$\}) \wedge E_1[i] \approx E_2[i]$
2. $E_1[i], E_2[i] \in \Pi \wedge ((a) \vee (b))$ */* parameter bijection */*
   (a) $E_1[i] \not\approx E_1[j], E_2[i] \not\approx E_2[j]$ *for every* $1 \leq j < i$
   (b) $E_1[i] \approx E_1[i-q]$ *iff* $E_2[i] \approx E_2[i-q]$ *for every* $1 \leq q < i$

Consider $\Sigma = \{A, B, C, D, E\}$, $\Pi = \{u, v, w, x, y, z\}$, $\widehat{\Sigma} = \{\{A, B\}, \{C\}, \{D, E\}\}$ (where $\widehat{\Sigma}_1 = \{A, B\}$, $\widehat{\Sigma}_2 = \{C\}$, and $\widehat{\Sigma}_3 = \{D, E\}$), $\widehat{\Pi} = \{\{u\}, \{v, w, x\}, \{y, z\}\}$ (where $\widehat{\Pi}_1 = \{u\}$, $\widehat{\Pi}_2 = \{v, w, x\}$, and $\widehat{\Pi}_3 = \{y, z\}$), $E_1 = AvzuxyE$, $E_2 = BuvzuxE$, and $E_3 = AvzzvzE$. Note that only E_1 and E_2 are e-matches because the constants match exactly with another constant in the same e-class and parameters from the same e-class form a bijection. Essentially, the e-match is the p-match where constant symbols can match any respective symbol in the same e-class from $\widehat{\Sigma}$ and parameter bijections are formed between symbols from the

same e-class from $\widehat{\Pi}$. Notice that the e-match definition is the p-match where the $=$ and $\neq$ symbol based operators are replaced respectively by the $\approx$ and $\not\approx$ symbol operators. We observe that since the e-alphabets are partitions of the original alphabets, we can perform a renaming of the e-string with the following encoding.

Definition 6.2.5 Rename e-string encoding (`rename`)**:** *Given an n-length e-string E with e-alphabet sets $\widehat{\Sigma}$, of size $\widehat{\mathcal{S}}$, and $\widehat{\Pi}$, of size $\widehat{\mathcal{P}}$, we define the alphabets $X_{\widehat{\Sigma}} = \{x_1, x_2, ..., x_{\widehat{\mathcal{S}}}\}$, $X_{\widehat{\Pi}} = \{x_{\widehat{\mathcal{S}}+1}, ..., x_{\widehat{\mathcal{S}}+\widehat{\mathcal{P}}}\}$, and $X = X_{\widehat{\Sigma}} \cup X_{\widehat{\Pi}}$. Define A_X as an $(\widehat{S}+\widehat{P})$-length bit array where $A_X[i] = PI$ if $x_i \in X_{\widehat{\Pi}}$ and $A_X[i] = SIGMA$ when $x_i \in X_{\widehat{\Sigma}}$. The* $\texttt{rename}(E)[i]$ *function is defined for $1 \leq i \leq n$ such that:*

$$\texttt{rename}(E)[i] = \begin{cases} x_i, \text{ if } E[i] \in \Sigma \wedge E[i] \in \widehat{\Sigma}_i \\ x_{\widehat{\mathcal{S}}+j}, \text{ if } E[i] \in \Pi \wedge E[i] \in \widehat{\Pi}_j \end{cases}$$

The $\texttt{rename}(E)$ will rename each symbol in E according to its e-class. For the working example, we have $X_{\widehat{\Sigma}} = \{x_1, x_2, x_3\}$, $X_{\widehat{\Pi}} = \{x_4, x_5, x_6\}$, $\texttt{rename}(E_1) = x_1x_5x_6x_4x_5x_6x_3$, $\texttt{rename}(E_2) = x_1x_4x_5x_6x_4x_5x_3$, and $\texttt{rename}(E_3) = x_1x_5x_6x_6x_5x_6x_3$. At this point, we notice that since the e-alphabets are partitions, the `rename` encoding handles all of the matching of e-classes from Definition 6.2.4. All that is left is to ensure that the renamed symbols indeed form a p-match via `prev` (as Definition 2.1.4). Thus, we perform the e-match by performing a p-match on the `rename` encodings.

Lemma 6.2.6 *Two e-strings E_1 and E_2, from Σ and Π with e-alphabets $\widehat{\Sigma}$ and $\widehat{\Pi}$ and* `rename` *alphabets $X_{\widehat{\Sigma}}$ and $X_{\widehat{\Pi}}$, are e-matches if* $\texttt{prev}(\texttt{rename}(E_1), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}}) = \texttt{prev}(\texttt{rename}(E_2), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}})$.

In our example, $\texttt{prev}(\texttt{rename}(E_1), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}}) = \texttt{prev}(\texttt{rename}(E_2), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}}) = x_100033x_3$ to verify the e-match, and $\texttt{prev}(\texttt{rename}(E_3), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}}) = x_100132x_3$.

Using the previous lemma and the fact that suffixes of `rename` behave as traditional suffixes, i.e. $\texttt{rename}(E)[i...n] = \texttt{rename}(E[i...n])$, we can support efficient e-matching by first computing $prE = \texttt{rename}(E)$ and constructing standard parameterized suffix structures on traditional suffixes of the standard string prE, which is the same as handling the equivalence parameterized suffixes (e-suffix) $\texttt{prev}(\texttt{rename}(E[i...n]))$. Following the trends of handling big data, we choose to use the more space-friendly parameterized suffix array (Definition 2.1.11) and parameterized longest common prefix array (Definition 2.1.12) over the parameterized suffix tree (pST) since currently, a compressed form of the pST is not directly constructed. Using the p-string framework, we define the suffix array for e-strings below.

Definition 6.2.7 Equivalence parameterized suffix array (eSA): *Given the n-length e-string E, the e-suffix array (eSA) is* $eSA = \texttt{construct\_pSA}(\texttt{rename}(\mathrm{E}), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}})$.

Note that the rank array R on the eSA is defined exactly as the R on the pSA. Now, we define the $eLCP$ to record the length of the longest prefix common between two neighboring e-suffixes in the eSA.

Definition 6.2.8 Equivalence longest common prefix array ($eLCP$): *Given the n-length e-string E, the $eLCP$ is* $eLCP = \texttt{construct\_pLCP}(\texttt{rename}(\mathrm{E}), X_{\widehat{\Sigma}}, X_{\widehat{\Pi}})$.

We can now answer e-matching problems via the eSA and $eLCP$ in the same way that we use the $pSA/pLCP$ and SA/LCP. The standard pattern matching problem is to find all occurrences of an m-length pattern P in an n-length text T. After constructing the eSA and $eLCP$ on T, we can find all instances of any e-string pattern P in T by searching for the encoding $\texttt{prev}(\texttt{rename}(P))$. As with the p-match, using the eSA and processing on the $eLCP$ of the n-length e-string T, we can find all of the n_{occ} instances of the m-length e-string pattern P in T in $O(m + \log n + \eta_{occ})$ time.

Theorem 6.2.9 *Using the eSA and $eLCP$ on the n-length e-string T, all η_{occ} instances of the m-length e-string pattern P in T can be found in $O(m + \log n + \eta_{occ})$ time.*

Motivated by the theoretical challenges of directly constructing the pSA and $pLCP$, we next explore new, novel construction methods.

6.2.3 eSA Construction

Previously, we discussed the close relationship between the e-string/p-string, e-suffix/p-suffix, eSA/pSA, and $eLCP/pLCP$. Recall that handling e-strings involves working with the p-string encoding `prev` of the e-string encoding `rename`; so, we will use the p-string and e-string terms interchangeably. Below, we address the eSA construction on e-strings via the pSA construction on p-strings (see Definition 6.2.7). The core challenge of direct p-suffix sorting is the dynamic nature of the p-suffixes (see Lemma 2.1.10). Thus, the clearest way to sort the p-suffixes is to first compute each $\texttt{prev}(T[i...n])$ and sort. The bottleneck with this approach is that $O(n^2)$ time and space is needed to sort the n p-suffixes of T. The problems with current direct p-suffix sorting approaches are that: (1) they are limited to

binary strings [40], (2) they identify improved constructions unable to break the $O(n^2)$ worst-case time bound [53], or (3) they exploit finite integral representations and computations to sort p-suffixes in practice [21].

In [53], the open question was posed whether sub-quadratic worst-case linear time constructions of the pSA exist. The research in [21] proposes arithmetic coding approaches to sort p-suffixes in practice in sub-quadratic time. Here, we propose two new pSA constructions with significantly improved theoretical worst-case time bounds over previous direct p-suffix sorting approaches.

Relationship between p-suffixes

Both of our p-suffix sorting approaches exploit the following observation. Let $prevT = \texttt{prev}(T)$. While Lemma 2.1.10 shows the disconnect between $prevT$ and the p-suffixes of T, Lemma 6.2.10 shows the relationship between $prevT$ and all of the p-suffixes of T.

Lemma 6.2.10 *Given an n-length p-string T with u_Π unique parameters, let $prevT = \texttt{prev}(T)$. Any p-suffix $\texttt{prev}(T[i...n])$ can be derived from $prevT[i...n]$ in at most $u_\Pi \in O(|\Pi|)$ changes.*

Proof Let $prevT = \texttt{prev}(T)$. Recall that in $prevT$, each parameter is encoded by 0 for the first occurrence of that parameter or otherwise, is encoded by the distance of the previous occurrence of that parameter. Let $T_i = T[i...n]$ and $prevT_i = prevT[i...n]$. So, $prevT_i$ can be converted to the p-suffix $\texttt{prev}(T_i)$ by resetting $prevT_i[j] = DISTANCE\_0$ if $T_i[j] \in \Pi$ and $T_i[j]$ is the first occurrence of that parameter in T_i (see Definition 2.1.4). Since there are u_Π symbols from Π in T, then at most $u_\Pi \in O(|\Pi|)$ changes are needed. □

For example, consider $T = AxyyzBxyyBzA$ from $\Sigma = \{A, B\}$ and $\Pi = \{w, x, y, z\}$, with $n = 12$, $prevT = \texttt{prev}(T) = A0010B541B6A$, and $prevT_4 = prevT[4...n] = 10B541B6A$. To find $\texttt{prev}(T[4...n])$ from $prevT_4$, we locate the first occurrences of parameters x, y, and z at positions 4, 1, and 2 respectively in $prevT_4$. Note that the parameter w does not occur in $prevT_4$ so we do not consider this parameter. Now, we simply set $prevT_4[1] = prevT_4[2] = prevT_4[4] = DISTANCE\_0$ and then $prevT_4 = \texttt{prev}(T[4...n])$.

To efficiently derive the p-suffix $\texttt{prev}(T[i...n])$ from suffixes of $prevT$, we need to know the first location of each unique parameter for each p-suffix of T in order to reset that parameter distance in $prevT_i$. We define the first unique parameters (F) data structure to

access the unique parameters in their order of occurrence. Note that while this specific order is not necessary for converting $prevT[i...n]$ to $\texttt{prev}(T[i...n])$, the order will be important for future algorithms.

Definition 6.2.11 First unique parameters (F): *For the n-length p-string T, F is a jagged array where each $F[i]$ has the locations in $T[i...n]$ of each unique parameter. Let $\texttt{locPI}(Y, y)$ return the index of the first location of the yth occurring unique parameter in Y or 0 in the case that no such parameter exists. More formally for $1 \leq i \leq n$, $F[i][j]$ exists if and only if $loc = \texttt{locPI}(T[i...n], j) > 0$ and so $F[i][j] = loc$.*

An example F is shown in Figure 6.2. Let u_Π be the number of unique parameters in the n-length text T. Since there are at most u_Π unique parameters in any $T[i...n]$, then the space required by F is at most $O(nu_\Pi)$ for the specific T, or generally $O(n|\Pi|)$ since $u_\Pi \in O(|\Pi|)$. We acknowledge the possibility of further compressing the F due to the fact that $F[i+1]$ does not add any new parameter information to $F[i]$ when both $T[i]$ and $T[i+1]$ are constants. To remove this redundancy, we can define $F[i]$ as before when $T[i]$ is a parameter and when $T[i]$ is a constant, we set $F[i]$ to the position $k > i$ of the next parameter in T. To make our algorithms and discussions more readable, we use the F as formalized in Definition 6.2.11.

Given the n-length e-string E and the $(|\Sigma| + |\Pi|)$-length bit array A, where $A[i] = SIGMA$ if symbol $i \in \Sigma$ and $A[i] = PI$ if symbol $i \in \Pi$, we construct the F in Algorithm 6-7, i.e. $F = \texttt{construct\_F}(E, A)$. The F is constructed via a right-to-left scan of E. As a new leftmost symbol $E[i]$ is encountered for the suffix $E[i...n]$, we determine how to include this symbol in the current $F[i]$ based on whether or not $E[i] \in \Pi$ and $E[i]$ already occurred in $E[i+1...n]$. Let $W = \texttt{incr}(w)$ accept array w and return array W where $W[i] = w[i] + 1$. (a) When $E[i] \notin \Pi$, we simply have that $F[i] = \texttt{incr}(F[i+1])$. (b) When $E[i] \in \Pi$ and does not occur in $E[i+1...n]$, we have found the first parameter; let $\mathcal{F} = F[i+1]$, $F[i][1] = 1$, and $F[i][2...(|\mathcal{F}| + 1)] = \texttt{incr}(\mathcal{F})$. (c) Otherwise, $E[i] \in \Pi$ and already occurred in $E[i+1...n]$, so we let $\mathcal{F}' = F[i+1]$ and delete the entry denoting parameter $E[i]$, i.e. remove $\mathcal{F}'[j]$ if $E[i + \mathcal{F}'[j]] = E[i]$ and then execute (b) with $\mathcal{F} = \mathcal{F}'$. We do this for $i = n, n-1, ..., 1$. In terms of running time, during each of the i-of-n iterations, at most $O(\max\{1, |F[i+1]|\})$ steps are needed and $|F[i+1]| \in O(u_\Pi)$ for E with u_Π unique parameters, and generally, $|F[i+1]| \in O(|\Pi|)$. So, the construction of F clearly requires $O(nu_\Pi)$ time for E, or generally $O(n|\Pi|)$ time.

Algorithm 6-7. Constructing F.

```
1  int[n][] construct_F(char E[n],bit A[a]){
2     int F[n][], i, k, size=0, v;  bit found_pi[a]={0,...,0};  char curr
3     for(i=n to 1){
4        curr=E[i]
5        if(A[curr]=PI ∧ found_pi[curr]=0){ // new first parameter
6           found_pi[curr]=1
7           size++
8           F[i]=new int[size]
9           F[i][1]=1
10          for(k=1 to size-1) F[i][k+1]=F[i+1][k]+1
11       }else if(A[curr]=PI){ // reoccurring first parameter
12          F[i]=new int[size]
13          F[i][1]=1
14          v=2
15          for(k=1 to size){
16             if(E[i+F[i+1][k]-1]≠curr){
17                F[i][v]=F[i+1][k]+1
18                v++
19             }
20          }
21       }else if(size>0){ // no new parameters
22          F[i]=new int[size]
23          for(k=1 to size) F[i][k]=F[i+1][k]+1
24       }
25    }return F
26 }
```

Lemma 6.2.12 *For the n-length p-string T with u_Π unique parameters, F is constructed in $O(nu_\Pi)$ time and space for T, or generally, $O(n|\Pi|)$ time and space.*

Construction via Parameterized Cover: A 3-Stage Sort

Our first *pSA* construction will integrate the notion of traditional *static* suffix sorting, i.e. the case where suffixes behave as standard suffixes, and *dynamic* suffix sorting, i.e. the case where suffixes behave differently from standard suffixes (see Lemma 2.1.10). We observe that each p-suffix of the n-length T can be segmented based on the parameterized cover, which we define in the following.

Definition 6.2.13 Parameterized cover (p-cover): *The p-cover for a p-string T is the length of the contiguous substring required to cover the first occurrence of each unique parameter within T. Let* $\texttt{first}(Y, y)$ *(respectively,* $\texttt{last}(Y, y)$*) return* (a) *the index in Y of the*

first (respectively, last) occurrence of symbol of y or (b) 0 *in the case that y does not exist. Let* $i_1 = \texttt{first}(\texttt{prev}(T), DISTANCE\_0)$ *and* $i_2 = \texttt{last}(\texttt{prev}(T), DISTANCE\_0)$. *Formally, if* $i_1 > 0$, *the p-cover substring is* $\texttt{prev}(T)[i_1...i_2]$ *and its length is* $C_i = \texttt{pcover}(T) = i_2 - i_1 + 1$. *Otherwise,* $C_i = 0$.

In essence, the p-cover is the *dynamic* part of the p-suffix. We observe that all p-suffixes can be divided into at most three parts: a static prefix, a static suffix, and the p-cover, i.e. the dynamic substring between the prefix and suffix.

Lemma 6.2.14 *For an n-length p-string T, any p-suffix of T can be segmented into at most one part that is dynamic and at most two parts that are traditional substrings of* $prevT = \texttt{prev}(T)$.

Proof There are four possible cases for segmenting a p-suffix $\texttt{prev}(T_i)$ of length $n_i = n-i+1$ with p-cover C_i, where $T_i = T[i...n]$.

1. 1 traditional part and 0 dynamic parts: Suppose that no parameters exist in T_i. Then, $C_i = 0$ and so the p-suffix is one traditional suffix of $prevT$, i.e. $\texttt{prev}(T_i) = T_i = prevT[i...n]$.

2. 0 traditional parts and 1 dynamic part: Suppose that $C_i = n_i$. Then, the p-suffix has only one dynamic part.

3. 1 traditional part and 1 dynamic part: There are two possibilities.

   (a) Suppose that the p-cover of T_i begins at position $j > 1$ and ends at position n_i. We know that since the prefix $T_i[1...j-1]$ is not included in the p-cover, then these symbols must be constants and thus, $T_i[1...j-1] = prevT[i...i+j-2]$. So, one dynamic part and one traditional part divide the p-suffix.

   (b) Suppose that the p-cover of T_i begins at position 1 and ends at position $k < n_i$. Since the `prev` encoding encodes a parameter π in T_i by either 0 for the first occurrence of π or otherwise, the distance to the previous π in T_i, then we can guarantee that $\texttt{prev}(T_i)[k+1...n_i] = prevT[i+k...n]$ since the nonzero parameter distances are already represented in $prevT[i+k...n]$, a traditional suffix of $prevT$. Thus, one dynamic part and one traditional part exist.

T	A	x	y	y	z	B	x	y	y	B	z	A

i	prev($T[i...n]$)												C_i	F		
1	A	0	0	1	0	B	5	4	1	B	6	A	4	2	3	5
2		0	0	1	0	B	5	4	1	B	6	A	4	1	2	4
3			0	1	0	B	0	4	1	B	6	A	5	1	3	5
4				0	0	B	0	4	1	B	6	A	4	1	2	4
5					0	B	0	0	1	B	6	A	4	1	3	4
6						B	0	0	1	B	0	A	5	2	3	6
7							0	0	1	B	0	A	5	1	2	5
8								0	1	B	0	A	4	1	4	
9									0	B	0	A	3	1	3	
10										B	0	A	1	2		
11											0	A	1	1		
12												A	0			

Figure 6.2: Computing the p-covers (which are shaded) and F data structure for $T = AxyyzBxyyBzA$ with $n = 12$. In this example, $C_{max} = 5$.

4. 2 traditional parts and 1 dynamic part: Suppose that the p-cover is such that $0 < C_i < n_i - 1$ and begins at position $j > 1$ and ends at position $k < n_i$. By combining cases 3(a) ($T_i[1...j-1] = prevT[i...i+j-2]$) and 3(b) ($\texttt{prev}(T_i)[k+1...n_i] = prevT[i+k...n]$), we have one dynamic part and two traditional parts.

□

We say that the p-cover for T is the length, $C \geq 0$, of the substring required to "cover" each of the first parameters T. Figure 6.2 shows the p-covers for all of the p-suffixes of an example p-string T. In the figure, you can also see the Lemma 6.2.14 segmentation, that is, how prefixes prior to the p-cover and suffixes following the p-cover are exactly substrings from $prevT = \texttt{prev}(T)$. We now identify the notion of a maximum p-cover. The maximum p-cover for the p-suffixes of T is $C_{max} = \max\{C_i \mid 1 \leq i \leq n\}$. Using the F data structure on T, we can trivially find the C_i and thus, easily find C_{max}. Algorithm 6-8 computes the maximum p-cover for T in linear time.

Lemma 6.2.15 *Given F, the maximum p-cover for the n-length p-string T is computed in $O(n)$ time.*

We now propose a three-stage sort to sort the e-suffixes of E, i.e. construct the pSA, by exploiting Lemma 6.2.14 and the partial sorting of the e-suffixes in SA_{prE}, where $prE =$

Algorithm 6-8. Finding the maximum p-cover C_{max}.

```
1  int max_pcover(int F[n][]) {
2      int i, len, C_i, C_max=0
3      for(i=1 to n){
4          len=|F[i]|
5          if(len>0){
6              C_i=F[len]-F[1]+1
7              if(C_i>C_max) C_max=C_i
8          }
9      }return C_max
10 }
```

$\texttt{prev}(\texttt{rename}(T))$. The proposed construction is displayed in Algorithm 6-9 with helper functions in Algorithm 6-10. Below, we describe the construction.

The first part of the eSA construction initializes data and constructs some data structures. Let $prE = \texttt{prev}(\texttt{rename}(E))$. Now, compute the traditional suffix structures SA_{prE}, LCP_{prE}, and R_{prE} on prE using a standard linear time suffix sort and LCP construction, i.e. `construct_SA_LCP_R`. Then, we construct the F data structure, compute the number of parameters in the string η_Π, and compute the maximum p-cover C_{max}. In the case that there are no parameters, i.e. $\eta_\Pi = 0$, then the eSA construction is already complete, i.e. $SA_{prE} = eSA$. Otherwise $\eta_\Pi \geq 1$ and we need to use perform additional steps to construct the eSA.

We continue the eSA construction now in step A, which places the traditional prefixes/suffixes into the same lexicographical group or bin. That is, we look at the suffixes in the SA_{prE} and group those neighboring suffixes with a common prefix extending to the first parameter of each suffix. In other words, the group is a contiguous region in the SA_{prE} with suffixes that share the same traditional prefix (segment before the p-cover), which is determined by oracling the LCP_{prE} and F data structures; in SA_{prE}, the traditional suffixes are grouped since they are indeed special traditional prefixes (with no p-cover). We observe that since parameter distances lexicographically precede constants, (1) the e-suffixes that form singleton groups are traditional suffixes (Lemma 6.2.14 Case 1) and are in their final resting place in the eSA and (2) all other e-suffixes (Lemma 6.2.14 Cases 2-4) can only be repositioned within the designated contiguous grouping. So, we will complete the sort by reordering/differentiating the e-suffixes within their respective contiguous groupings found in step A.

Recall from Lemma 6.2.14 that e-suffixes can be segmented into a traditional prefix, a p-cover (dynamic part), and a traditional suffix. At this point in the construction, step A has given us a partial ordering (ranking) of the e-suffixes with respect to the traditional prefixes and traditional suffixes. In step B, we prepare the ordering (ranking) of the p-covers. There are η_Π total parameters in the string and thus η_Π different p-covers. For ordering/ranking purposes, we will compute/store the substrings that represent each p-cover beginning with the first parameter and extending C_{max} symbols; in the case that no such trailing symbol exists, we pad substring with the TERMINAL symbol. Next, we sort the substrings and rank the p-covers.

Observe that now, all segments of the e-suffixes are sorted: the traditional prefixes and traditional suffixes are sorted (SA_{prE} constructed during initialization and grouped in step A) and we manually sorted the p-covers (step B). At this point, we need to use the aforementioned sortings/rankings in three stages to completely differentiate the e-suffixes and construct the eSA. First, use the traditional prefix grouping from step A and the goal is to reorder the e-suffixes within the contiguous group. For each group, we do the following. For all e-suffixes in the group, i.e. not yet differentiated, resort the group with respect to the p-cover ranks. In the case that only one e-suffix exists in the group, the e-suffix is in its final place in the eSA. When e-suffixes have the same p-cover rank, we need to further differentiate this new contiguous group by the last segment, i.e. the traditional suffix. So, we collect the traditional suffix ranks and resort these particular e-suffixes. Since all three segments of the e-suffix have been considered, each e-suffix is in its final position in the eSA after differentiating with respect to the traditional suffix. In the following, we formalize the time requirement for this construction.

Theorem 6.2.16 *Given an n-length e-string E from Σ and Π with η_Π parameters and maximum p-cover C_{max}, we can construct the e-suffix array (eSA) in $O(\max\{n|\Pi|, \eta_\Pi C_{max}\})$ time and extra space for $\eta_\Pi \geq 1$, and $O(n)$ time and extra space for $\eta_\Pi = 0$.*

Proof Clearly, a sort of the segments (Lemma 6.2.14) of the n e-suffixes of E yields the eSA. When $\eta_\Pi = 0$, only $O(n)$ scans performed and $O(n)$ data structures are constructed/allocated, so the eSA is constructed in $O(n)$ time and $O(n)$ extra space. When $\eta_\Pi = 0$, clearly the initialization is dominated by the construction of F, which requires $O(n|\Pi|)$ time and $O(n|\Pi|)$ extra space in addition to the $O(n)$ needed for the eSA. Step A requires one $O(n)$ scan of the SA_{prE}. Step B allocates space for the η_Π total C_{max}-length p-covers and

Algorithm 6-9. Constructing the eSA.

```
1  struct block_Z{ int index, char block[Z] }
2  struct pair{ int a, int b }
3  int[n] construct_eSA(char E[n]) {
4     int SA_prE[n], LCP_prE[n], R_prE[n], rE[n]=rename(E), prE[n]=prev(rE), η_Π=count_PI(rE,A_X)
5     (SA_prE,LCP_prE,R_prE)=construct_SA_LCP_R(prE)
6     if(η_Π=0) return SA_prE
7     // ---- when parameters exist in rE (η_Π>0)
8     int i, j, k, k1, k2, q, x, y, suf, len, curr, eSA[n], tmp[n], t1, t2
9     int YR[n]={0,...,0}, F=construct_F(rE,A_X), C_max=max_pcover(F)
10    struct block_{C_max} YS[η_Π], tmp4[n]; struct pair X[n], Y[n], tmp2[n], tmp3[n]
11    // ----A: bin the traditional prefixes and traditional suffixes
12    i=j=1
13    do{
14       tmp[i]=j, t1=|F[SA_prE[i]]|, t2=|F[SA_prE[i+1]]|
15       while(i+1≤n ∧ t1>0 ∧ t2>0 ∧ F[SA_prE[i]][1]=F[SA_prE[i+1]][1]
16                   ∧ LCP_prE[i+1]≥F[SA_prE[i]][1]-1) { i++, tmp[i]=j }
17       i++,j++
18    }while(i≤n)
19    x=binning(tmp,n,X)
20    // ----B: sort the p-cover (dynamic part) of the e-suffixes
21    for(i=1 to n){
22       if(A_X[rE[i]]=PI){
23          YS[i].index=i
24          for(j=1 to C_max){
25             if(i+j-1>n) YS[i].block[j]=TERMINAL
26             else YS[i].block[j]=prE[i+j-1]
27          }for(j=1 to |F[i]|) YS[i].block[F[i][j]]=DISTANCE_0
28       }
29    }sort(YS,1,η_Π,2) // sort rows of YS[1...n] by the 2nd variable, i.e. blocks
30    for(i=1 to η_Π) YR[YS[i].index]=i
31    // ----C: differentiating e-suffixes with tiebreakers
32    k=1
33    for(i=1 to x){
34       // ---- tiebreaker 1: differentiate same prefixes by the p-covers
35       curr=0
36       for(j=X[i].a to X[i].b){
37          curr++
38          tmp2[curr].a=SA_prE[j]
39          tmp2[curr].b=YR[SA_prE[j]+F[SA_prE[j]][1]-1]
40       }sort(tmp2,1,curr,2) // sort rows of tmp2[1...curr] by 2nd variable, i.e. b
41       len=X[i].b-X[i].a+1
42       for(k1=k to k+len-1) eSA[k1]=tmp2[k1-k+1].a
43       // ---- tiebreaker 2: differentiate same prefix/p-cover by suffix
44       k1=k
45       for(j=1 to curr) { t1=eSA[k+j-1], tmp4[j]=YS[YR[t1]+F[t1][1]+1] }
46       tmp=same(tmp4)
47       y=binning(tmp,curr,Y)
48       for(q=1 to y){
49          curr=0
50          for(j=Y[q].a to Y[q].b){
51             curr++
52             tmp3[curr].a=eSA[k1+j-1]
53             suf=eSA[k1+j-1]+F[eSA[k1+j-1]][1]+C_max-1
54             if(suf≤n) tmp3[curr].b=R_prE[suf]
55             else tmp3[curr].b=-suf // handle terminal padding
56          }sort(tmp3,1,curr,2) // sort rows of tmp3[1...curr] by 2nd variable, i.e. b
57          len2=Y[i].b-Y[i].a+1
58          for(k2=k1 to k1+curr-1) eSA[k2]=tmp3.a[k2-k1+1]
59          k1=k1+curr
60       }
61       k=k+len
62    }return eSA
63 }
```

Algorithm 6-10. Helper functions.

```
1  // given S[n] with block attribute sorted, return nondecreasing
2  // int array G, where G[j]=G[j+1] iff S[j].block=S[j+1].block
3  int[n] same(struct block_C S[n]){
4     int i=1, G[n], g=1
5     while(i≤n){
6        G[i]=g
7        while(i+1≤n ∧ match(S[i].block,S[i+1].block)) { i++, G[i]=g }
8        g++
9     }return G
10 }
11
12 // Z has z nondecreasing ints and j distinct ints; we evaluate Z[1...y]
13 // and post in BP the first/last indices where each distinct int occurs
14 int binning(int Z[z],int y,struct pair BP[]){
15    int i=1,j=0;  struct pair curr
16    do{
17       curr.a=curr.b=i
18       while(i+1≤y ∧ Z[i]=Z[i+1]) { i++, curr.b=i }
19       i++
20       j++
21       BP[j]=curr
22    }while(i≤y)
23    return j
24 }
25
26 // given alphabet type identifier A, which for parameter s has A[s]=PI
27 // and for constant s has A[s]=CONSTANT, we count the parameters in E
28 int count_PI(char E[n],bit A[a]){
29    int count=0, i
30    for(i=1 to n){
31       if(A[E[i]]=PI) count++
32    }return count
33 }
```

sorts them in $O(\eta_\Pi C_{max})$ time and space. Finally, step C considers each e-suffix at most twice to differentiate; so the total work required is in $O(n)$. Thus, $O(\max\{n|\Pi|, \eta_\Pi C_{max}\})$ time and extra space is needed. □

In the worst case, $\eta_\Pi = C_{max} = n$ and so the algorithm runs in $O(n^2)$ time and extra space. In practice, the resource consumption can vary depending on the alphabets and e-string composition. For alphabet symbols from a uniform distribution, we expect $C_{max} \in O(|\Sigma|+|\Pi|)$ and $\eta_\Pi \in O(\frac{n|\Pi|}{|\Sigma|+|\Pi|})$, and then, with $\eta_\Pi \geq 1$ the algorithm will execute in $O(n|\Pi|)$ time and extra space on average.

We acknowledge the opportunity to compute the equivalence longest common prefix array ($eLCP$) during this eSA construction. However, due to the lengthiness of Algorithm 6-9 and the fact that we use this algorithm as motivation for an improved eSA construction, we omit the $eLCP$ computation at this stage.

Improved Construction: An $O(|\Pi|)$-Stage Sort

Historically, it has been challenging to determine a better/different way to bound the worst case for pSA construction on the n-length T. For instance, in [53] the pSA construction is said to have a worst case behavior of $O(n^2)$. In [21], we propose a different way to bound the worst case in practice in $O(\frac{n^2 \log n}{m})$, based on arithmetic codes representing m-length blocks of the text. From another perspective, the worst case running time analysis of Algorithm 6-9 is an achievement because it bounds the p-suffix sorting in $O(nC_{max})$ time, where C_{max} is the length of the maximum p-cover. In this section, we are motivated by Algorithm 6-9 to propose a better construction with an improved worst case time bound.

Recall that Algorithm 6-9 segments a p-suffix into three parts (i.e. traditional prefix, p-cover, traditional suffix) and sorts the p-suffixes, i.e. differentiates the segments, in a constant number of steps. The bottleneck is the time to prepare the information necessary for the p-cover part of the sort. The idea of our improved algorithm is to upgrade the segmentation concept of Algorithm 6-9 so that we can discard the p-cover bottleneck. In the following, we formalize a different segmentation based on the zeros of the p-suffix $\texttt{prev}(T[i...n])$, which represent the first parameters in $T[i...n]$. We call this the *zeros segmentation* or *first parameters segmentation.*

Lemma 6.2.17 *For an n-length p-string T with the first parameters data structure F, any p-suffix of T can be segmented into exactly $|F[i]|$ zeros and at most $(|F[i]|+1)$ traditional*

substrings of $prevT = \texttt{prev}(T)$.

Proof Consider segmenting the p-suffix $\texttt{prev}(T[i...n])$ by the zero distances. By Definition 6.2.11, $\texttt{prev}(T[i...n])$ has exactly $|F[i]|$ zero segments. For the other segments, we know from Lemma 6.2.10 that any p-suffix $\texttt{prev}(T[i...n])$ can be derived from the single p-suffix $prevT = \texttt{prev}(T)$ by resetting the first parameter distances in $prevT[i...n]$. So, the non-zero segments must be traditional substrings of $prevT$. At most $(|F[i]| - 1)$ traditional substrings are between all the zero segments since there can be either one traditional substring or nothing between consecutive zero segments. Since a traditional substring can precede the first zero segment or follow the last zero segment, then at most $(|F[i]| + 1)$ traditional substring segments exist. □

The aforementioned segmentation (displayed in Figure 6.3) splits a p-suffix by the zero distances, yielding at most $(2|F[i]| + 1)$ total segments. If we say that T has u_Π unique parameters, then the most segments existing in any p-suffix will be $(2u_\Pi + 1) \in O(u_\Pi)$, or more generally $(2|\Pi| + 1) \in O(\Pi)$, segments will exist. Our improved pSA construction in Algorithm 6-11 is a depth-first sort that differentiates the p-suffixes in at most $O(u_\Pi)$, or more generally $O(|\Pi|)$, stages based on the segmentation of Lemma 6.2.17. We describe the algorithm below.

Given the n-length e-string E, we first perform an initialization step. We prepare the encoding $prE = \texttt{prev}(\texttt{rename}(E))$, compute the first parameters data structure F on E, and compute the number of unique parameters in E, i.e. $u_\Pi = |F[1]|$. Now, using standard suffix array techniques, we compute the suffix array data structures on prE with $(tmp, LCP_{prE}, R_{prE}) = \texttt{construct\_SA\_LCP\_R}(prE)$, where we call the suffix array tmp because we will be manipulating this array en route to the desired eSA. The LCP_{prE} and R_{prE} are respectively the LCP and R arrays on the string prE. Next, we construct the RMQ r from the LCP_{prE}; this RMQ information is necessary for efficient queries to compare the traditional substring segments of Lemma 6.2.17. At this point, we note that when $u_\Pi = 0$, then $eSA = tmp$ and even $eLCP = LCP_{prE}$, so the work is complete; we acknowledge that additional measures can be taken to make this step more lightweight in terms of memory. When $u_\Pi \geq 1$, we need additional steps to sort the dynamic e-suffixes.

Initially, tmp will provide a partial ordering of the e-suffixes, specifically by segments prior to the first zero segment, i.e. the traditional prefix, so that we can refine and continue to sort via the recursive function `pbinning`. The `pbinning` preconditions are: (pre1) some permu-

tation of the e-suffixes in the partition $tmp[intervalL...intervalH]$ will populate the desired $eSA[intervalL...intervalH]$, (pre2) the e-suffixes in the partition $tmp[intervalL...intervalH]$ share the same prefix based on all segments prior to and including the $(curr_param - 1)$st zero segment, and (pre3) the e-suffixes in $tmp[intervalL...intervalH]$ are currently ordered by rank of the traditional suffix following the $(curr_param - 1)$st zero segment. The function postcondition is that the given e-suffixes in the partition $tmp[intervalL...intervalH]$ will be completely sorted within $eSA[intervalL...intervalH]$. Now, call `pbinning` with $intervalL = 1$, $intervalH = n$, and $curr\_param = 1$ to signify that we will work to sort all e-suffixes in tmp starting with the first unique parameter in each e-suffix.

(*1) If $curr\_param > u_{\Pi}$, we return from the recursive function (goto (*5)) because no such parameter exists and so the e-suffixes must be completely differentiated/sorted. Otherwise, we continue to sort the e-suffixes in a depth-first manner based on the zeros segmentation, formalized in Lemma 6.2.17. Let $i = intervalL$.

(*2) If the considered suffix $tmp[i]$ has no parameter, i.e. $|F[tmp[i]]| = 0$, then by (pre3) this traditional suffix is in its final resting place in the eSA, i.e. $eSA[i] = tmp[i]$. Otherwise, we need to identify a subpartition of $tmp[intervalL...intervalH]$ that shares a prefix up until the $(curr_param)$th zero segment. That is, up until the (off)th symbol, where $off = 0$ if $curr\_param = 1$ and otherwise, $off = F[tmp[i]][curr\_param - 1]$. The partition will start at $iL = i$ and have at least $len = 1$ e-suffixes. In lines 27-30, we loop to create a subpartition with respect to the e-suffix $tmp[i]$, as described in (*3).

(*3) We include the e-suffix $tmp[i + 1]$ into the subpartition if (i) both e-suffixes $tmp[i]$ and $tmp[i + 1]$ have more than off symbols (otherwise, they are already differentiated), (ii) have at least $curr\_param$ parameters (otherwise, they are already differentiated), and (iii) the currently considered segments preceding the $(curr_param)$th zero segment match exactly, i.e. $\texttt{rmq}(r, tmp[i] + off, tmp[i + 1] + off) \geq (F[tmp[i]][curr\_param] - off - 1)$, with the $(curr_param)$th zero segment in the same location. If (i), (ii), and (iii) are all true, then $tmp[i]$ and $tmp[i + 1]$ are indistinguishable and are thus in the same subpartition, so increment i, increment len, set $iH = i$, and try to add another e-suffix to this subpartition; repeat (*3). Otherwise, the subpartition is complete and we process in the following way.

(*4) If $len = 1$, then the subpartition is a singleton, which by (pre1) means that the e-suffix is in its final place in the eSA, i.e. $eSA[i] = tmp[i]$. Otherwise, we have that the e-suffixes in $tmp[iL...iH]$, which share the same prefix by (pre2), cannot be distinguished by the $(curr_param)$th zero segment. So, we set $off = F[tmp[iL]][curr\_param]$ and in

lines 34-42 reorder the subpartition within *tmp* with respect to the rank of the currently unconsidered suffixes following the first off symbols of each e-suffix. This updates the lexicographical ordering to satisfy the `pbinning` precondition (pre3) so that the function can be called recursively (goto ($*^1$)) to distinguish these $tmp[iL...iH]$ e-suffixes with the same prefix in a depth-first manner, where $intervalL = iL$, $intervalH = iH$, and the next zero is $curr\_param + 1$.

($*^5$) We return from the recursive function. Now, the subpartition $eSA[iL...iH]$ is completely sorted. So, we increment i and if $i \leq intervalH$, we continue to ($*^2$) to repeat the same subpartitioning process on $tmp[iH + 1...end]$, where $end \leq intervalH$.

After `pbinning` is complete, the n e-suffixes are sorted into the desired eSA. The resources consumed by this eSA construction are formalized below.

Theorem 6.2.18 *Given an n-length e-string E from Σ and Π with u_Π unique parameters, we can construct the e-suffix array (eSA) in $O(nu_\Pi)$, or generally $O(n|\Pi|)$, time and extra space.*

Proof Algorithm 6-11 correctly computes the eSA, since it performs a depth-first sort of the first parameter segmentations (Lemma 6.2.17) of the n e-suffixes in E. The initialization of the main function `construct_eSA_improved`, lines 6-11, is dominated by the F construction, which requires $O(nu_\Pi)$ time by Lemma 6.2.12. (Note that `construct_SA_LCP_R` requires only $O(n)$ time [1] and also, `setup_rmq` is only a linear time operation [25, 84, 85].) The last step (line 12) of `construct_eSA_improved` calls the helper function `pbinning`, which recursively sorts the n e-suffixes.

Now, consider the function `pbinning`. Each of the $O(u_\Pi)$ segments of an e-suffix are considered at most once by the main loop (lines 19-47). In lines 21-26, the function determines if and where the next segment occurs in the current e-suffix in $O(1)$ time. Then, in lines 27-30, the function uses this e-suffix to create a partition of, say len, e-suffixes that share the same prefix in $O(len)$ time, since the `rmq` query is an $O(1)$ operation (see [25, 84, 85]). These partitions are possibly singletons (line 31), which are placed into the eSA in $O(1)$ time. Otherwise, (lines 32-44) the rank of the next segments are considered in each of the e-suffixes in the current partition, then we `sort` the ranks in $O(len)$ time to lexicographically refine the e-suffix ordering, and finally recursively call `pbinning` to consider differentiating the current ordering of the size len partition based on the next segment. Note that in the aforementioned steps, the work required by each of the len e-suffixes in the partition is in

$O(1)$. After returning from the recursive call, the aforementioned partition is completely sorted/stored in the eSA and we continue the procedure by considering the next e-suffix not yet processed. Further, since the `pbinning` function clearly processes each of the $O(u_\Pi)$ segments of the n e-suffixes at most $O(1)$ times, then `pbinning` requires $O(nu_\Pi)$ time. Thus, the main function `construct_eSA_improved` will then require $O(nu_\Pi)$ time in the worst case.

In terms of the extra space needed, i.e. the memory in addition to the desired eSA, we require the $O(n)$ data structures prE, LCP_{prE}, r on LCP_{prE} (see [25, 84, 85]), etc. and the $O(nu_\Pi)$ sized F data structure (see Lemma 6.2.12). The recursive space needed for the parameters and local variables of the `pbinning` function is in $O(u_\Pi)$ since only $O(1)$ variables are required at each of the $O(u_\Pi)$ depths of recursion. Thus, the total extra space $O(nu_\Pi + u_\Pi)$ is in $O(nu_\Pi)$.

Further, the time and extra space required is in $O(n|\Pi|)$ since $u_\Pi \in O(|\Pi|)$. □

6.2.4 $eLCP$ Algorithm

Due to the connection between the e-string/p-string, e-suffix/p-suffix, eSA/pSA, and $eLCP/pLCP$, we will often refer to p-string terms and theory when constructing the $eLCP$. While we can construct the $eLCP$ by standard $pLCP$ construction algorithms (see Definition 6.2.8), the bottleneck is the theoretical worst case time. For example the p-Kasai algorithm [53] constructs the $pLCP$ for the n-length p-string T in $O(n^2)$ time in the worst case. In [19, 23], the $pLCP$ is constructed in $O(n)$ expected time for certain groups of p-strings. Like the pSA construction, the $pLCP$ construction suffers from Lemma 2.1.10. Due to the dynamic nature of p-suffixes from the n-length T, it is not necessarily the case that the information from $pLCP[R[i]]$ can be used to help compute $pLCP[R[i+1]]$ for $1 \leq i < n$ (see [23]). If we compute $pLCP[i]$ individually and independently without using information from other $pLCP[j]$, the task is to find the maximum k-length prefix common between the p-suffixes $\texttt{prev}(T[pSA[i]...n])$ and $\texttt{prev}(T[pSA[i-1]...n])$, which will require $O(k)$ time. Since $n-1$ of these computations are needed and since $k \in O(n)$ in the worst case, then we naively construct the $pLCP$ in $O(n^2)$ time. While this result may refrain us from constructing the $pLCP$ via individual and independent computations, we show that when using the data structures created after executing our `construct_eSA_improved` in Algorithm 6-11, each $eLCP[i]$ can be constructed individually and independently in $O(|\Pi|)$ time. Then, the $eLCP$ is constructed in $O(n|\Pi|)$ time, breaking the worst case time barrier.

Algorithm 6-11. Improved eSA construction for the n-length e-string E.

```
1  int u_Π, eSA[n], tmp[n], LCP_prE[n], R_prE[n], prE[n], F[n][]
2  struct pair ptmp[n]
3  struct RMQ r
4
5  int[n] construct_eSA_improved(char E[n]) {
6    tmp=rename(E)
7    prE=prev(rename(tmp))
8    F=construct_F(tmp,A_X)
9    u_Π=unique_PI(F)
10   (tmp,LCP_prE,R_prE)=construct_SA_LCP_R(prE)
11   r=setup_rmq(LCP_prE)
12   pbinning(1,n,1)
13   return eSA
14 }
15
16 void pbinning(int intervalL,int intervalH,int curr_param){
17   int i, j, iL, iH, e, off, len
18   if(curr_param≥1 ∧ curr_param>u_Π) return
19   i=intervalL
20   do{
21     if(|F[tmp[i]]|=0){
22       eSA[i]=tmp[i]
23     }else{
24       iL=i, len=1
25       if(curr_param=1) off=0
26       else off=F[tmp[i]][curr_param-1]
27       while(i+1≤intervalH ∧ tmp[i]+off≤n ∧ tmp[i+1]+off≤n
28         ∧|F[tmp[i]]|≥curr_param ∧ F[tmp[i+1]]≥curr_param
29         ∧rmq(r,tmp[i]+off,tmp[i+1]+off)≥(F[tmp[i]][curr_param]-off-1)
30         ∧F[tmp[i]][curr_param]=F[tmp[i+1]][curr_param]){i++,len++,iH=i}
31       if(len=1) eSA[i]=tmp[i]
32       else{
33         off=F[tmp[iL]][curr_param]
34         for(j=iL to iH){
35           ptmp[j-iL+1].a=tmp[j]
36           e=tmp[j]+off
37           if(e>n) ptmp[j-iL+1].b=e
38           else if(|F[tmp[j]]|>curr_param ∧ F[tmp[j]][curr_param+1]=e){
39             if(e+1>n) ptmp[j-iL+1].b=e+n
40             else ptmp[j-iL+1].b=R_prE[e+1]+2n
41           }else ptmp[j-iL+1].b=R_prE[e]+3n
42         }sort(ptmp,1,len,2) // sort rows of ptmp by 2nd type, i.e. b
43         for(j=1 to len) tmp[iL+j-1]=ptmp[j].a
44         pbinning(iL,iH,curr_param+1)
45       }
46     }i++
47   }while(i≤intervalH)
48 }
49
50 int unique_PI(int F[n][]) {
51   int u=0
52   if(n≥1) u=|F[1]|
53   return u
54 }
```

Segments	*Z*: Zero	*N*: Nonzero Blocks

T	B	x	y	y	B	z	y	x
prevT	B	0	0	1	B	0	3	6
i	prev(*T*[*i*...*n*])							
1	B	0	0	1	B	0	3	6
2		0	0	1	B	0	3	6
3			0	1	B	0	3	0
4				0	B	0	3	0
5					B	0	0	0
6						0	0	0
7							0	0
8								0

Figure 6.3: The zeros segmentation for each p-suffix of $T = BxyyBzyx$.

Our proposed $eLCP$ construction is shown in Algorithm 6-12. We describe the approach in the following. Consider the n-length e-string E. After constructing the eSA with function `construct_eSA_improved`, the following data structures are generated: the intended eSA on E, the RMQ r on LCP_{prE} where $prE = \texttt{prev}(\texttt{rename}(E))$, and the first parameters data structure F on E. Let us now focus on computing the function $\texttt{elcp}(T_a, T_b) = \max\{k \mid \texttt{prev}(\texttt{rename}(T_a)) =_k \texttt{prev}(\texttt{rename}(T_b))\}$ for $\texttt{elcp}(E[i...n], E[j...n])$ and any i and j. We use the segmentation in Lemma 6.2.17, which splits the p-suffix by the zero distances (the first parameters), to compute the `elcp`. For the purposes of discussion, we segment the p-suffix at i into I_k and the p-suffix at j into J_k. For instance, with $T = AwBzABwz\$$, the segments of the p-suffixes at $i = 1$ ($\texttt{prev}(T[1...n]) = A0B0AB54\$$) and $j = 5$ ($\texttt{prev}(T[j...n]) = AB00\$$) are respectively $I = \{A, 0, B, 0, AB54\$\}$ and $J = \{AB, 0, 0, \$\}$. We highlight that computing/storing the segmentation is unnecessary since these boundaries are already represented in F (see Definition 6.2.11) and further, our approach will not need to touch the segment symbols. For convenience, we discuss our `elcp` approach in terms of I and J.

Our improved `elcp` computation is a left-to-right match on the first parameter segments, i.e. I and J. From Lemma 6.2.10, we can derive any p-suffix from $\texttt{prev}(T)$ by resetting only the zero locations. So, when comparing two p-suffixes derived from a common text, we need to be sure that the zero segments (dynamic segments) are properly aligned. The other

segments in I and J are then candidates for the exact match, a task that can be done in $O(1)$ time via the given RMQ r.

Let $\mathcal{U} = \texttt{prev}(\texttt{rename}(T[i...n]))$ and $\mathcal{V} = \texttt{prev}(\texttt{rename}(T[j...n]))$. Let $common = 0$, $b = 1$, and $\Delta = 0$. $(*^1)$ Let $minseg = \min\{|I_b|, |J_b|\}$ and $maxseg = \max\{|I_b|, |J_b|\}$. If both I_b and J_b are first parameter, zero segments, then we have successfully aligned/matched first parameters between $\mathcal{U}$ and $\mathcal{V}$; indicate this by extending the match (increment *common*) and consider the next position to match (increment Δ and increment b). Else if either I_b or J_b is a zero distance segment, then the zero segment *cannot* match with the other nonzero segment, so the `elcp` computation is complete (goto $(*^2)$). Otherwise, we exact match the segments via $h = \min\{minseg, \texttt{rmq}(r, i + \Delta, j + \Delta)\}$ (in $O(1)$ time) and accumulate the match length in *common*, i.e. $common = common + h$. In the case that $h < minseg$, then the segments mismatched at some point so the `elcp` computation is complete (goto $(*^2)$). In the case that $h < maxseg$, then the segments are of different length, so the `elcp` is complete (goto $(*^2)$). Otherwise, $h = minseg = maxseg$ and the segments matched completely, so we need to consider the next segment (increment b) starting at positions $(i + \Delta)$ and $(j + \Delta)$ in the text with $\Delta = \Delta + h$; if another segment exists ($b \leq \min\{|I|, |J|\}$) goto $(*^1)$ and otherwise, goto $(*^2)$. $(*^2)$ Return *common*, which contains the maximum prefix common between $\mathcal{U}$ and $\mathcal{V}$. The `elcp` in Algorithm 6-12 performs the previous procedure by identifying the segment location on-demand, i.e. iterating the individual $F[i]$ and $F[j]$ to determine where the zero locations occur, which eliminates the need to construct/store the I and J. The running time and extra space are formalized below.

Lemma 6.2.19 *Given an n-length e-string E with u_Π unique parameters, the RMQ r on LCP_{prE} where $prE = \texttt{prev}(\texttt{rename}(E))$, and the first parameters data structure F on E, let $\mathcal{U} = \texttt{prev}(\texttt{rename}(T[i...n]))$ and $\mathcal{V} = \texttt{prev}(\texttt{rename}(T[j...n]))$. The computation $\texttt{elcp}(\mathcal{U}, \mathcal{V})$ requires $O(\max\{|F[i]|, |F[j]|\})$ time and $O(nu_\Pi)$ space.*

Proof The correctness of the `elcp` computation (implemented in Algorithm 6-12) follows from a comparison of the first parameter segmentations (Lemma 6.2.17) of the e-suffixes $\mathcal{U}$ and $\mathcal{V}$ of E. The algorithm iterates and attempts to align/compare the $O(|F[i]|)$ first parameter segments of $\mathcal{U}$ and the $O(|F[j]|)$ parameter segments of $\mathcal{V}$; there are a total of $O(\max\{|F[i]|, |F[j]|\})$ iterations. Since the segment comparison performed during each iteration is a constant time `rmq` query [25, 84, 85] or another $O(1)$ operation, the running time of `elcp` is in $O(\max\{|F[i]|, |F[j]|\})$. With respect to space, the algorithm only allocates

$O(1)$ space, so the required space is due to the parameters. Since the RMQ data structure r has size $O(n)$ [25, 84, 85], the space required is dominated by the $O(nu_\Pi)$ size data structure F (see Lemma 6.2.12). □

Using Lemma 6.2.19, we compute the $eLCP$ array (see `construct_eLCP` in Algorithm 6-12) with $(n-1)$ `elcp` computations: $eLCP[i] = \texttt{elcp}(T[eSA[i]...n], T[eSA[i-1]...n])$ for $1 < i \leq n$. Denote the number of unique parameters in T by u_Π, i.e. $u_\Pi = |F[1]|$. The $eLCP$ construction requires $(n-1)$ `elcp` operations, each requiring $O(u_\Pi)$ time. Thus, $O(nu_\Pi)$ time is required. In terms of space, `construct_eLCP` allocates the $eLCP$ in addition to $O(1)$ space. Thus, the extra space is dominated by the $O(nu_\Pi)$ space of `elcp`. More generally, since $u_\Pi \in O(|\Pi|)$, the algorithm requires $O(n|\Pi|)$ time and extra space.

Theorem 6.2.20 *Given an n-length e-string E from Σ and Π with u_Π unique parameters, the RMQ r on LCP_{prE} where $prE = \texttt{prev}(\texttt{rename}(E))$, and the first parameters data structure F on E, the equivalence longest common prefix array ($eLCP$) is constructed in $O(nu_\Pi)$ time and extra space, or generally $O(n|\Pi|)$ time and extra space.*

6.2.5 Generalization

The e-match is a powerful matching scheme. By altering the e-match alphabets, we can also solve the p-match and the traditional match via the same scheme, with the same data structures and respective constructions. When we compose the e-alphabets of only singleton e-classes, then no indeterminacy exists and so, the e-match becomes the p-match.

Lemma 6.2.21 *The e-match simplifies to the p-match by first letting $\widehat{\Sigma} = \widehat{\Pi} = \emptyset$ and then, letting $\widehat{\Sigma}_i = \{\Sigma[i]\}$ for all $1 \leq i \leq |\Sigma|$ and $\widehat{\Pi}_j = \{\Pi[j]\}$ for all $1 \leq j \leq |\Pi|$.*

By forcing *all* symbols to be constants and then applying the alphabet rearrangement of Lemma 6.2.21, we remove indeterminacy and the original e-match scheme becomes the traditional exact match.

Lemma 6.2.22 *The e-match simplifies to exact matching by applying Lemma 6.2.21 to the alphabets $\Sigma = \Sigma \cup \Pi$ and $\Pi = \emptyset$.*

Further, we can manipulate the $\widehat{\Sigma}$ and $\widehat{\Pi}$ alphabets to allow p-matching and exact matching with a form of indeterminacy. The significance of the previous lemmas is that our e-match

Algorithm 6-12. The *eLCP* construction.

```
1  int[n] construct_eLCP(int eSA[n],struct RMQ r,int F[n][]){
2    int eLCP[n]={0,...,0}, i, a, b
3    for(i=2 to n) {
4      a=eSA[i-1], b=eSA[i]
5      eLCP[i]=elcp(a,F[a],b,F[b],r)
6    }return eLCP
7  }
8  int elcp(int a,int Fa[y],int b,int Fb[z],struct RMQ r){
9    int common=0, k=1, min0s=min(y,z), loc0a=0, loc0b=0, pvloc0a, pvloc0b
10   boolean more
11   do{
12     more=false
13     if(min0s=0 ∨ k>min0s){
14       if(a+loc0a≤n ∧ b+loc0b≤n) common=common+rmq(r,a+loc0a,b+loc0b)
15     }else{
16       pvloc0a=loc0a, pvloc0b=loc0b
17       loc0a=Fa[k], loc0b=Fb[k]
18       if(a+pvloc0a≤n ∧ b+pvloc0b≤n)
19         common=min(loc0a-1,loc0b-1,common+rmq(r,a+pvloc0a,b+pvloc0b))
20       if(loc0a=loc0b=(common+1)){ more=true, common++ }
21     }k++
22   }while(more ∧ k≤min0s+1)
23   return common
24 }
```

data structures can be used to address p-matching and traditional matching problems with the same data structure and approach. For example, the eSA construction can also yield the pSA and traditional suffix array SA. From a software development standpoint, the e-match problem has great utility since it can be reused in a variety of applications by simply altering the alphabet sets.

6.2.6 Conclusions

In this section, we propose the equivalence parameterized match (e-match) as an extension of the parameterized match (p-match) between parameterized strings (p-strings), from the constant alphabet Σ and parameter alphabet Π, to add a level of indeterminacy to the p-match. A rearrangement of the alphabets will generalize the e-match to address the p-match and the traditional match. Thus, the e-match can address traditional exact matching problems such as database search, in addition to routine p-match applications such as RNA structural similarity, software analysis, and plagiarism detection, and also variations of the aforementioned problems with a form of indeterminate matching. To solve the e-match, we apply a new encoding with the standard p-string encodings in order to use the parameterized suffix array (pSA) and parameterized longest common prefix ($pLCP$) matching framework. The historical bottleneck of the pSA and $pLCP$ construction for the n-length text T is the theoretical worst case $O(n^2)$ construction time, due to handling dynamically changing suffixes that can invalidate traditional suffix properties. In fact, an open problem was posed in [53] of whether or not sub-quadratic, i.e. $o(n^2)$, constructions exist. In this section, we introduce new theory leading to an improved $O(n|\Pi|)$ worst case construction of the pSA and $pLCP$ to break the time barrier. This result is a breakthrough for directly sorting dynamic suffixes under encodings, a task required for direct parameterized suffix sorting. Our construction result is comparable to constructing suffix structures on traditional strings, which can also require a time factor linear in the alphabet.

Chapter 7

Matching RNA Secondary Structures

In this chapter, we propose/construct data structures and algorithms to address pattern matching problems involving RNA secondary structures. The work in this chapter was published/presented in the following.

- Beal, R., Adjeroh, D., Abbasi, A.: The forward stem matrix: An efficient data structure for finding hairpins in RNA secondary structures. ACM Conference on Bioinformatics, Computational Biology, and Biomedical Informatics (ACM-BCB). pp. 576-585. ACM. (2013)
- Beal, R., Adjeroh, D.: Suffix arrays for structural strings. Poster presented at the International Workshop on Combinatorial Algorithms (IWOCA), Duluth, MN (15-17 October 2014)

7.1 The Structural Suffix Array

RNA (ribonucleic acid) is an important molecule known to mediate transfer of cellular information from DNA-encoded genes to functional proteins and, in the case of non-coding RNA, play a role in various cellular processes such as translation and splicing. Like proteins, RNA can fold into potentially complex three-dimensional structures. The function of a given RNA, such as in transcription, splicing, cellular localization and translation depends critically on its structure [15, 90, 93]. Figure 7.1 shows the basic RNA secondary structure elements.

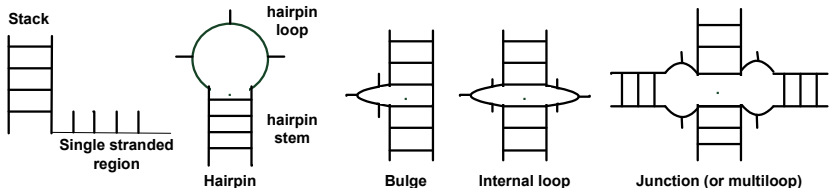

Figure 7.1: Basic structural elements of an RNA secondary structure.

A given RNA sequence is made up of four types of nucleotides (bases), namely adenine (A), cytosine (C), guanine (G), and uracil (U). The Watson-Crick base pairs (A,U) and (C,G) are complementary bases which pair up to form stacks or stems during RNA folding. We consider RNA as a string from the alphabet $\Sigma_{RNA} = \{A, C, G, U\}$. Thus, methods used in string pattern matching can be applied (with appropriate modifications) to the problem of analysis of RNA sequences and/or structure. For instance, there has been a lot of interest in predicting RNA secondary and tertiary structures [29, 39], and in alignment of RNA sequences [95, 9, 47]. Combinatorial properties of RNA secondary structures were studied in [51, 79], showing the difficulty involved in most problems that require matching RNA structures.

Let $T = T[1...n]$ and $P = P[1...m]$ be the database and query RNA strings, respectively. Heyne et al. [50] proposed an algorithm for determining the longest collinear sequence of substructures common to P and T in $O(m^2n^2)$ time and $O(mn)$ space. Their method required exact matches between substructures, while inexact matches are allowed between structural motifs. Given that the structure of T is known, Bafna et al. [10] inferred the structure of P using an alignment procedure in $O(n^2m^2 + nm^3)$ time. In [45, 58], algorithms were proposed for finding arc-preserving common subsequences in arc-annotated sequences. Special edit distances for matching RNA structures were proposed in [99, 5], whereby edit operations were performed on both individual bases, and on arcs (complementary base pairs).

The specific problem we address is more closely related to the recent work on pattern matching on RNA secondary structures [96, 74, 86, 89]. Xu et al. [96] used the notion of secondary expressions to represent the RNA structures, and proposed an algorithm that finds exact matches of a given secondary expression in $O(nm^2)$ time, where m is the size of the secondary expression, and n is the size of the text. The affix tree, which allows bidirectional matches for exact and inexact pattern matching on RNA structures, was studied in [74, 69].

The more recent use of affix trees in [74] requires worst case time in $O(nm)$ and $O(n)$ space to find the structural pattern P of length m in a database string T of length n. Answering similar queries for inexact matching required time in $O(HK^{2p-2}s^pn)$, where H is the maximum length of a hairpin loop, K is the maximum length of an interior element, $s =$ maximum length of a stack or stem, and p is the total number of fragments in the secondary structure. Though the space complexity is linear, the space requirement for the affix tree is a practical bottleneck. Influenced by the suffix tree with a memory footprint known to be quite high [47, 1], the affix tree suffers from the same space problem (requiring about 45 bytes per node [89]), since it is composed of two suffix trees - one regular suffix tree and the suffix tree of the reversed sequence. Strothmann proposed a more space efficient variant called the affix array [89] with the same basic functionality as the affix tree; this new structure was also used to answer RNA structure queries. Shibuya [86] introduced the notion of structural strings for RNA-based pattern matching and constructed the structural suffix tree (s-suffix tree) for related pattern matching problems, i.e. the structural matching (s-match) problem. Figure 7.2 shows an example of a structural match between RNA sequences. In terms of matching with the s-suffix tree, the problem of inexact matching for RNA structures was not considered by Shibuya. The s-suffix trees were constructed by extending the parameterized suffix trees (p-suffix trees) of Baker [11, 14] to support the complementary base pairs needed for RNA structures. Like traditional suffix trees [1], the p-suffix tree construction has been the focus of much work [65, 31, 67, 68]. Further, as with suffix trees and affix trees, the practical space requirement for the s-suffix tree is a practical bottleneck. In [16], we propose and construct the structural suffix array (sSA) and structural longest common prefix ($sLCP$) array for s-matching via more lightweight data structures. Here, we discuss methods using the sSA and $sLCP$ to address various pattern matching queries on RNA structures for both exact and inexact matching. Our results could have applications in prediction and discovery of RNA structural motifs [81, 78], classification of RNA structures [81], functional annotation of RNA, etc.

Main Contributions: In this research, we use the sSA and $sLCP$ s-matching framework to address various RNA pattern matching problems, including the detection of exact RNA sequences, structurally similar RNA sequences, and in answering inexact RNA structural queries. The following formalizes our main result.

Theorem 7.1.1. *Given an n-length RNA database D, an inexact structural query P of*

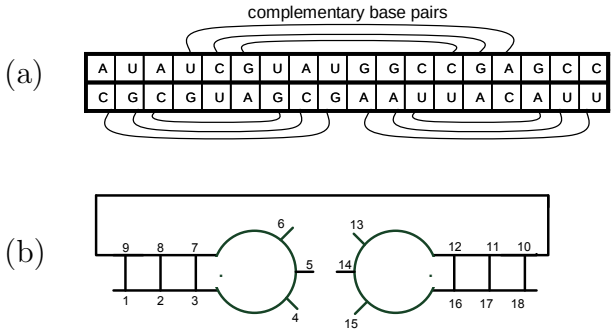

Figure 7.2: Example by Shibuya [86] of two RNA sequences with the same structure, despite having different symbols at each position: (a) shows the relationship between the complementary bases of the RNA sequences and (b) shows the structure formed by each sequence, which folds to form touching loops or kissing hairpins.

maximum length m with f fragments F_i, $i = 1, 2, \ldots f$, and the sSA and sLCP for D, it is possible to determine all locations in D of P in $O(\eta_1\phi + fm \times (\log n + \mathcal{M}m))$ time and $O(f \times n + \eta)$ space, where ϕ represents the work to integrate all possible fragments from a single matching fragment F_1 in D, $\eta = \eta_1 + \eta_2 + ... + \eta_f$, η_i is the number of matching fragments F_i in D, and $\mathcal{M}$ is the maximum number of matches found during the algorithm.

7.1.1 RNA Pattern Matching

In this section, we describe how to use the s-match (Theorem 2.2.6) to solve various combinatorial queries in RNA pattern matching. The following nucleotides are present in a strand of RNA: adenine (A), cytosine (C), guanine (G), and uracil (U). We denote an RNA sequence as an element of $(\Sigma_{RNA})^*$, where $\Sigma_{RNA} = \{A, C, G, U\}$. The Watson-Crick pairings are such that A and U pair ($A \leftrightarrow U$) and C and G pair ($C \leftrightarrow G$). For symbols $s_1, s_2 \in \Sigma_{RNA}$, we say that $\texttt{pair}(s_1, s_2)$ returns $true$ when s_1 and s_2 pair and $false$ otherwise.

Exact Pattern Matching

Problem A: Given an RNA sequence Q and an RNA database of sequences say $D = d_1\$ \circ d_2\$ \circ ... \circ d_z\$$ for some z with each d_i as an RNA sequence, $|Q| = m$, and $|D| = n$, the problem is to determine all n_{occ} locations of Q in D. That is, given an RNA pattern and a database of sequences (without information on the secondary structures formed), the problem is to find all sequences in the database that matches the pattern.

We observe that the s-match problem in Definition 2.2.2 with $\Sigma \neq \emptyset$, $\Pi = \emptyset$, and $\Gamma = \emptyset$

is restricted to only the verbatim matching of constants $\sigma \in \Sigma$, which is the essence of exact matching. With the selection of alphabets as $\Sigma = \Sigma_{RNA}$, $\Pi = \emptyset$, and $\Gamma = \emptyset$, we can answer the specified problem in $O(\log n + m + \eta_{occ})$ time via Theorem 2.2.13, where η_{occ} is the number of occurrences of Q in D. This result is analogous to traditional matching using the provided SA and LCP arrays as discussed in [72, 1]. For similar problems that find all locations of a match, we can solve related existential problems by answering yes if $n_{occ} > 0$ and no otherwise.

Problem B: RNA secondary structures are formed by twisting and turning an RNA sequence to form various structural elements (see Figure 7.1) such as stems, loops, etc. [15]. Given an RNA sequence Q and the RNA database D, $|Q| = m$, and $|D| = n$, the problem is to identify all n_{occ} structurally similar (or the same) occurrences of Q that exist in D. That is, given an RNA pattern and a database of sequences (without information on the secondary structures formed), the problem is to find all occurrences in the database that are structurally similar with the pattern. An example was provided in Figure 7.2.

It is identified in [86] that we can equate RNA structures via the s-match problem with $\Sigma = \emptyset$, $\Pi = \Sigma_{RNA}$, and $\Gamma = \{(A, U), (C, G)\}$. Using the s-match between Q and D, we can find structurally similar RNA sequences, including the identical pattern. It follows from our s-matching theory that the problem is answered in $O(\log n + m + \eta_{occ})$ time via Theorem 2.2.13.

Inexact Pattern Matching

We define an *inexact structure query* as an inexact pattern that can match several RNA sequences by allowing *varying lengths* of structural elements. More specifically, a structure query is an m-length pattern P that is decomposed into f different structural elements or fragments (F), for short, labeled as a stem, loop, etc. Each fragment F_i in P is allowed to be some sequence of a length in the range (s_i, e_i). In this case of P, each fragment F_i includes a *specified sequence*, say Q, of symbols in Σ_{RNA} so that when some l is chosen, such that $s_i \leq l \leq e_i$, only l-length substrings of Q are permitted to match.

Problem C: Given an inexact structure query P of a known RNA structure with varying lengths and specified sequences, the RNA database D, $|P| = m$, and $|D| = n$, the problem is to determine all occurrences of P in D. Figure 7.3 shows an example structural query pattern. Our idea is to (1) use the developed s-matching approach to search for individual

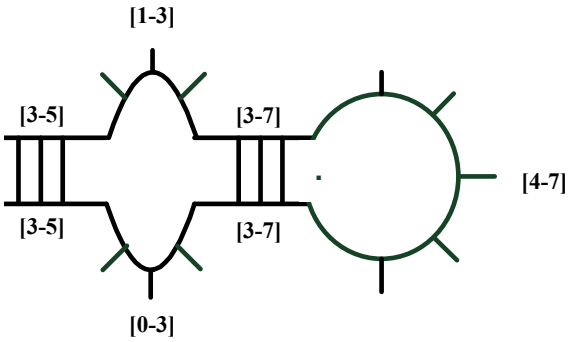

Figure 7.3: Example definition for an approximate structure pattern. Numbers in brackets indicate the range for the given fragment length.

fragments F_i in D, (2) integrate the fragment occurrences $(F_1, F_2, ..., F_f)$, and (3) validate the results prior to reporting. In this situation, our s-match needs to detect only complementary base pairs for stems and exact matches for other structural elements. Since both matches are reported when comparing `prev` encodings, we need to construct sSA and $sLCP$ on T with $\Sigma = \emptyset$, $\Pi = \Sigma_{RNA}$, and $\Gamma = \{(A,A),(C,C),(G,G),(U,U)\}$. In the following, we sketch the data types and algorithm steps.

Define V as an $f \times n$ matrix where the f rows represent the F_i and the n columns represent the locations in D. Each individual element $V(i,j)$ represents a set of lengths $\{l_1, l_2, ..., l_g\}$, where each l_x $(s_i \leq l_x \leq e_i)$ means that a length-l_x fragment F_i exists at position j in D. Initially, we set $V(i,j) = \emptyset$ to denote the existence of no such match. We also define L to be of f rows where each $L[i]$ consists of a varying number of tuples $(j, \{l_1, l_2, ..., l_g\})$ that correspond to the nonempty set at $V(i,j)$. Let η_i denote the number of occurrences of the fragment F_i in D and let $\eta = \eta_1 + \eta_2 + ... + \eta_f$. Then, the data structure V has $O(f \times n + \eta)$ elements and L has $O(\eta)$ elements. Let Ξ be an $n \times |\Sigma_{RNA}|$ matrix (with $O(n \times |\Sigma_{RNA}|)$ elements), with mappings $A \rightarrow 1$, $C \rightarrow 2$, $G \rightarrow 3$, and $U \rightarrow 4$, that for each $\Xi(i,1)$, we are detecting the location (index) of the first occurrence of symbol A in $D[i...n]$ or -1 if the symbol does not exist. $\Xi(i,2)$, $\Xi(i,3)$, and $\Xi(i,4)$ are defined similarly for the respective symbols C, G, and U. We also define the following: $e_{max} = \max\{e_1, e_2, ..., e_f\}$, $e_{min} = \min\{e_1, e_2, ..., e_f\}$, $l_{max} = e_1 + e_2 + ... + e_f$, and $l_{min} = s_1 + s_2 + ... + s_f$.

In our algorithm, we begin by finding all possible occurrences of F_i in D and logging in both V and L the resulting length and position in D of the occurrence of F_i. We assume that we can process P left-to-right as a pattern and for algorithmic concision, we assume that P does not have start/end biological markers. Also, we assume that P includes a

pointer from each half of a stem to the other stem half for quick access. We expect RNA sequences and (s_i, e_i) to be given in P for all such F_i except for the ending half of a stem, which can be derived easily from the beginning half of the stem. Consider each successive F_i $(i = 1, 2, ..., f)$ in terms of steps (a) or (b).

(a) Let $q = 0$ and $v = 0$. If F_i is any structural element except the ending half of a stem, we need to begin logging occurrences of sequences of F_i as they appear in D. Say that the RNA sequence associated with F_i begins at position r and ends at position z in P. Compute $S = \texttt{sencode}(P[r + v...r + s_i + q + v - 1])$. (Note, we always require that $r + s_i + q + v - 1 \leq z$.) Next, we use the sSA and $sLCP$ via Theorem 2.2.13 to find all occurrences of S in D. Since these structural elements must occur exactly, we must filter the aforementioned results for exact matches only. Say an occurrence of S in D is at position u. Let the function $\mathcal{F}(a, b)$ return $true$ if $(\Xi(a, b) = -1) \vee (r+v+\Xi(a, b)-1 > z) \vee (T[a + \Xi(a, b) - 1] = P[r + v + \Xi(a, b) - 1])$ and return $false$ otherwise. We can efficiently detect an exact match by knowing that the first A, C, G, and U align, i.e. $\mathcal{F}(u, 1) \wedge \mathcal{F}(u, 2) \wedge \mathcal{F}(u, 3) \wedge \mathcal{F}(u, 4)$. If the result is $true$, log in both $V(i, u)$ and $L[i]$ the length $|S| = s_i + q$ of each found occurrence of S in D. Next, we extend the match to $q = 1, 2, ..., (e_i - s_i)$. For each increment of q, walk through each tuple $(y, \{l_1, l_2, ..., l_g\})$ in $L[i]$ and log (in both $L[i]$ and $V(i, y)$) a longer length result $(s_i + q)$ iff $T[y + q] = P[r + s_i + q + v - 1]$ and $r + s_i + q + v - 1 \leq z$. Repeat the step beginning with the computation of S for $v = 1, 2, ..., e_i - s_i$ to consider remaining substrings that may possibly match with F_i.

(b) Otherwise, F_i is the ending half of a stem. Via a pointer, we can determine that the other half of the stem is say F_h and its sequence begins at position w and ends at position x in P. Since the RNA is a single strand of symbols, the only way for some F_h substring, say $P[w...w+s_h-1]$, to form complementary base pairs with the current structure F_i is if the sequence loops back. That is, only sequences can pair if they exist as a `reverse` in D. To find these results, we need to perform a slight variation of step (a) since that step filters results for exact matches only and in this scenario, we must consider matching stems that pair. Recall that we are given sSA and $sLCP$ on D with $\Sigma = \emptyset$, $\Pi = \Sigma_{RNA}$, and $\Gamma = \{(A, A), (C, C), (G, G), (U, U)\}$. This forces the s-match scheme of Theorem 2.2.6 to simply compare the `prev` encoding. So, if we search for S, by Definition 2.1.4, we will also report similar results that are equal under `prev`, such as AUA, CGC, UGU, etc.

Now to find the complementary pairing, we need to filter the results by considering the first A, C, G, and U symbol. Thus, we can perform step (a) except that now, (1) $r = w$ and $z = x$, (2) compute $S = \texttt{sencode}(\texttt{reverse}(P[r + v...r + s_i + q + v - 1]))$, (3) filter pairings by redefining $\mathcal{F}(a, b)$ to return *true* if $(\Xi(a, b) = -1) \vee (r + v + \Xi(a, b) - 1 > z) \vee \texttt{pair}(T[a + \Xi(a, b) - 1], P[r + v + \Xi(a, b) - 1])$, and (4) log extensions when $\texttt{pair}(T[y + q], P[r + v - q])$ and $v - q \geq 0$.

Next, we integrate occurrences of $F_1, F_2, ..., F_f$ to form a path of indices a_z and lengths l_z: $H = (a_1, l_1) \rightarrow (a_2, l_2) \rightarrow ... \rightarrow (a_f, l_f)$. Initially, if $\eta_i = 0$ for any i, then a fragment is not found and we can exit, reporting $n_{occ} = 0$. Otherwise, we integrate the paths in a depth-first manner. Initially, we are at depth $d = 0$. Define $\mathcal{A}_d = 0$ and $l_{path_d} = 0$ for each depth level. The path H is formed by appending the (index,length) pair at each increment of d and removing the last pair at each decrement of d. One at a time, consider each length l_d of $L[1]$, for discussion, at position j_d. For each l_d, compute $l_{path_d} = l_d$ and $\mathcal{A}_d = j_d + l_d$. Now, let $d = d + 1$ and consider a length l_d of F_2 at position j_d. If $V(2, \mathcal{A}_{d-1})$ is empty, we backtrack to $d = d - 1$ and explore other lengths in F_1. If F_2 is the ending part of a stem, we need to make sure that we choose the same l_d at $V(2, \mathcal{A}_{d-1})$ as the beginning part of the stem. For any other structural element, choose any l_d at $V(2, \mathcal{A}_{d-1})$. Update $\mathcal{A}_d = \mathcal{A}_{d-1} + l_d$ and $l_{path_d} = l_{path_{d-1}} + l_d$. If at any time, $l_{path_d} > l_{max}$ or perhaps the V element is empty, we backtrack and explore other length options; otherwise, we continue the current path considering successive F. When $d = f - 1$, a final length is added to the path, and both $\mathcal{A}_d$ and l_{path_d} are updated, we need to verify the correctness of the resulting structure length, i.e. $l_{min} \leq l_{path_d} \leq l_{max}$. Prior to reporting paths as valid occurrences of P in D, we need to verify that the chosen stem halves, which were chosen to be of equal length, pair symbol-by-symbol via function `pair`. We continue this process until all paths from F_1 are explored or discounted. Each resulting RNA secondary structure can be subsequently scanned for further biological validation.

We formalize the running time of the aforementioned algorithm in the following theorem.

Theorem 7.1.1 *Given an n-length RNA database D, an inexact structural query P of maximum length m with f fragments F_i, $i = 1, 2, \ldots f$, and the sSA and sLCP for D, it is possible to determine all locations in D of P in $O(\eta_1\phi + fm \times (\log n + \mathcal{M}m))$ time and $O(f \times n + \eta)$ space, where ϕ represents the work to integrate all possible fragments from a single matching fragment F_1 in D, $\eta = \eta_1 + \eta_2 + ... + \eta_f$, η_i is the number of matching fragments F_i in D,*

and $\mathcal{M}$ is the maximum number of matches found during the algorithm.

Proof The running time of the algorithm can be divided into the summation of the running time of the following: (1) detecting all occurrences in D of fragments in P and (2) integrating/validating the fragments. Consider (1) first for F_i. Let $\mathcal{M}$ denote the maximum number of s-match occurrences detected at any part of (1). In this part, we construct `sencode` for an s_i-length string, which requires $O(s_i)$ time by Lemma 2.2.7. Then, an s-match is done in $O(s_i + \log n + \mathcal{M})$ time in the worst case by Theorem 2.2.13 and filtered for matches in $O(|\Sigma_{RNA}|\mathcal{M}) \in O(\mathcal{M})$ time. Extending the match for each fragment F_i requires $\mathcal{M}(e_i - s_i)$ work. Collectively, the aforementioned steps are executed a total of $(e_i - s_i + 1)$ time for the other sequence substrings and then, executed f times for each F_i. Thus, (1) requires $O(fe_{max} \times (\log n + \mathcal{M}e_{max}))$ time, in $O(fm \times (\log n + \mathcal{M}m))$ since $e_{max} = \max\{e_1, e_2, ..., e_f\} \leq m$. Consider (2) next. Let ϕ represent the work to integrate all possible fragments from a single matching fragment F_1 in D. Variable ϕ is limited in nature since at each step, we only continue generating paths if the total path length l_{path} is such that $0 \leq l_{path} \leq l_{max}$. Then, the time to find all paths is in $O(\eta_1\phi)$. We perform a final scan to verify that the chosen stem occurrences indeed pair, which takes $O(m)$ time. Thus, the total time to execute (1) and (2) is then in $O(\eta_1\phi + fm \times (\log n + \mathcal{M}m))$. We note that the space required for the algorithm, aside from the provided D and P, is dominated by $|V|$, $|L|$, $|\Xi|$, and $O(f)$ space for the depth-first search. Thus, $O(f \times n + \eta + n \times |\Sigma_{RNA}| + f)$ space is required and since $|\Sigma_{RNA}| = 4$, the space required is in $O(f \times n + \eta)$. □

7.1.2 Discussion and Conclusion

We acknowledge that it is possible to rank our search results according to energetics to add a level of biological validation to our procedure [29, 39]. Additionally, we can also use the solution presented for Problem C as a basis for matching structural queries with no specified sequences by iteratively populating the structural query with contiguous elements from D and executing the algorithm. In terms of the original Problem C, we can also handle the case where wildcard symbols are at the end of a fragment sequence by a slight modification to our algorithm. Overall, the advantage of addressing RNA pattern matching applications using the sSA and $sLCP$ as the foundation is the flexibility to propose numerous extensions to a problem with slight modifications to the solution. Suppose that we complicate the structure query and allow some structural element sequences, say those for stems, to match

also sequences with a similar, though inexact, form. To address this situation, we can simply make a minor adjustment to how our algorithm filters s-matches. Handling such complications with the traditional SA and LCP will require *additional* searches to be added to the algorithm.

In summary, we showed how to use the s-match to handle inexact queries with RNA structural patterns as the input, which was not considered in [86]. A task of future research will be to study complementary base pairing schemes beyond the standard Watson-Crick pairings, and incorporate those into new data structures for other applications in biology.

7.2 The Forward Stem Matrix

RNA secondary structures are sequences composed of nucleotides with different regions of the RNA strand bonding to form structural elements such as stems, loops, bulges, etc. The core challenge of finding RNA secondary structures in a sequence is to integrate pattern matching with structure analysis. A philosophy used to find RNA secondary structures is to first find the stems and then postprocess the resulting structures. The results of such an exercise will have significant implications in various problems in computational biology, such as in prediction of RNA secondary and tertiary structures [28], RNA structure design [66], functional classification of RNA structures [78], micro RNA target prediction [94], and discovery of RNA structural motifs [81], among others. The recent interest in long non-coding RNAs [82] with significantly complicated structures [77] motivates the need to efficiently find RNA secondary structures.

We introduce the Forward Stem Matrix to efficiently store and permit quick access to all k-length stem possibilities, for $k \in K$, in an n-length RNA sequence T. Each $FSM[i][j]$ either (a) contains a set of indices for the $K[i]$-length closing-stem halves $\mathcal{C} = \texttt{reverse}(\texttt{complement}(T[j...j + K[i] - 1]))$ that correspond to the opening-stem half $\mathcal{O} = T[j...j + K[i] - 1]$ if $T[j...j + K[i] - 1]$ is the first occurrence of $\mathcal{O}$ in T, (b) contains $\emptyset$ if (a) yields no such indices, or (c) contains the negated index of the furthest previous occurrence of $\mathcal{O}$ in T, otherwise. In this research, we construct the FSM data structure and provide example RNA applications.

Other data structures and methods have been proposed to address variations of RNA secondary structure matching. The Structural Suffix Tree (sST) [86] data structure was developed to handle an extension of the exact pattern matching between an RNA pattern P

and text T, in that the individual symbols are used to determine *how* the RNA secondary structure may occur. In cases where we want to specify search for an RNA secondary structure, we require pattern matching between a text and an approximate structure query. Along the lines of other suffix structures [1, 72], the Affix Tree [74] and Affix Array [89] (implemented in the structator system [75]) were constructed specifically for matching with RNA and shown to apply to matching structure queries (also handled in [75]). In an analysis of the time and space resources needed to match RNA secondary structures, the use of even the most elegant data structures still shows the true combinatorial nature of the problem [51, 79]. For example, the methods reported in [74, 89] require exponential time with respect to the number of fragments in an RNA structure query to find inexact matches of an input structure query pattern. This motivates the need for more efficient data structures that more naturally integrate pattern matching with RNA structure analysis.

Our construction of the FSM makes use of the Furthest Previous Non-Overlapping Factor ($FPnF_k$) and the Furthest Previous Factor (FPF_k). The $FPnF_k$ stores, for each k-length substring at index i in the n-length string T, the minimum index in T where $T[i...i+k-1]$ occurred in a non-overlapping fashion, i.e. $T[\min(x)...\min(x)+k-1] = T[i...i+k-1]$ such that $x+k-1 < i$. The Furthest Previous Factor (FPF_k) omits the non-overlapping condition. The $FPnF_k$ and FPF_k arrays are related to the Longest Previous Factor (LPF) [35] array that stores information on the longest factors in a string, which is useful in data compression, string factorization, etc. [36]. From the LPF, a family of data structures were derived, including the Longest Previous Non-Overlapping Factor ($LPnF$) [38], Longest Previous Reverse Factor ($LPrF$) [37], Longest Previous Non-Overlapping Reverse Factor ($LPnrF$) [37], and *Prior* [47] for traditional strings from the symbol alphabet Σ. This family was extended in [19] to include the Parameterized Longest Previous Factor ($pLPF$) and variations [23] for the parameterized string (p-string) [13] from a constant alphabet Σ and a parameter alphabet Π. In this section, we provide new constructions for the $FPnF_k$ and FPF_k arrays.

Main Contributions: The Forward Stem Matrix (FSM) is proposed to efficiently store stem options in an RNA sequence for quick access. To assist with the FSM construction, we first introduce the Furthest Previous Non-Overlapping Factor ($FPnF$) and the Furthest Previous Factor (FPF) arrays. We then provide a construction of the $FPnF$ and FPF arrays by making an interesting connection with p-string theory. An improved linear time construction of the $FPnF$ and FPF arrays is shown via suffix trees. Next, the

aforementioned results are integrated with suffix arrays (SA) to yield a linear time construction of the FSM. Afterwards, we provide example applications of the FSM: to find hairpin and pseudoknot structures in an RNA sequence. Our main results are formalized below.

Theorem 7.2.7. Given an n-length T and a constant k, the $FPnF_k$ array is constructed in $O(n)$ time and $O(n)$ space.

Theorem 7.2.9. Given the n-length T and the set of stem lengths K, the Forward Stem Matrix (FSM) is constructed in $O(n|K|)$ time using $O(n)$ extra space.

In all of our construction algorithms, we avoid data structures that tend to require heavy preprocessing in practice, such as the processing required to determine the longest prefix common between *any* two suffixes of a string via range minimum query (RMQ) or lowest common ancestor (LCA) computations.

7.2.1 Forward Stem Matrix

An RNA secondary structure, composed of the nucleotides adenine (A), cytosine (C), guanine (G), and uracil (U), is formed when nucleotides from different regions bond within a strand of RNA; the Watson-Crick complementary base pairs between the nucleotides are (A,U) and (C,G). Figure 7.1 shows the basic structural elements [15] that can make up an RNA secondary structure: single strands, stems (or stacks), loops, and bulges. To search for an RNA structure in a text, an approximate search query (shown in Figure 7.3) is used. These queries are composed of the basic structural elements, each of which is a fragment of the query. For each fragment, we either specify (1) the exact RNA sequence desired, (2) a range $[l, h]$ of the fragment length, or (3) a hybrid of both (1) and (2) with wildcard symbols. A philosophy used for matching structure queries is to begin with identifying the stems and filter the results as appropriate.

A quick look at Figure 7.1 shows that the stem is a fundamental secondary structure. All of the double stranded RNA structures can basically be viewed in terms of a stem, or a set of stems. A stem or stack (see Figure 7.1) forms on a single strand of RNA, say T, when some k-length substring of RNA at position i in T, i.e. $\mathcal{O} = T[i...i+k-1]$, forms a complementary base pairing with a forward, non-overlapping, reversed substring in T, i.e. some $j > i+k-1$ where $\mathcal{C} = T[j...j+k-1] = \texttt{reverse}(\texttt{complement}(\mathcal{O}))$, `reverse` reverses a string, and `complement` forms the base pairings (see Definition 7.2.1). For discussion, we

call $\mathcal{O}$ the opening-stem half and $\mathcal{C}$ the closing-stem half.

Definition 7.2.1 `complement` **function:** *We define the following function to return a complementary RNA sequence:* $\texttt{complement}(U[1] \circ U[2] \circ ... \circ U[|U|]) = \overline{U[1]} \circ \overline{U[2]} \circ ... \circ \overline{U[|U|]}$, *where* $1 \leq i \leq |U|$ *and each* $\overline{U[i]} =$*'C'* *if* $U[i] =$*'G'*, $\overline{U[i]} =$*'G'* *if* $U[i] =$*'C'*, $\overline{U[i]} =$*'A'* *if* $U[i] =$*'U'*, *and* $\overline{U[i]} =$*'U'* *if* $U[i] =$*'A'*.

The RNA sequence $T = ACCCCUGGGGU$, for example, has a stem of length $k = 5$ at indices $i = 1$ and $j = 7$ since $ACCCC = \texttt{reverse}(\texttt{complement}(GGGGU))$. For a structure query, we need to find all stems in T of lengths within the ranges for the considered stem fragments: $k_1, k_2, ..., k_c \in K$. Since a stem is non-overlapping, then $\lfloor \frac{n}{2} \rfloor \geq \max\{k \mid k \in K\}$. We define the Forward Stem Matrix (FSM) to find all k-length closing-stem halves in T for any opening-stem half in T and every $k \in K$.

Definition 7.2.2 Forward stem matrix (FSM)**:** *Consider the n-length RNA sequence* T *from* $\Sigma_{RNA} = \{A, C, G, T\}$ *and a set of integers* K, *with* $1 \leq k \leq \lfloor \frac{n}{2} \rfloor$ *for* $k \in K$. *From any* $K[i]$*-length opening-stem half in* T *starting at position* j, *i.e.* $T[j...j + K[i] - 1]$, $FSM[i][j]$ *provides access to the indices of all forward, non-overlapping, complementary closing-stem halves in* T, *i.e.* $\texttt{reverse}(\texttt{complement}(T[j...j + K[i] - 1]) = T[w...w + K[i] - 1]$ *for* $j + K[i] - 1 < w$. *Formally, each* $FSM[i][j] = y$ *is defined below for* $1 \leq i \leq |K|$ *and* $1 \leq j \leq n$.

$$y = \begin{cases} \{w \mid \texttt{reverse}(\texttt{complement}(T[j...j + K[i] - 1])) = \\ \qquad\qquad T[w...w + K[i] - 1] \ \wedge \ j + K[i] - 1 < w\}, \\ \qquad\qquad \text{if } T[j...j + K[i] - 1] \neq T[a...a + K[i] - 1] \ \forall \ a < j \\ -\min\{x \mid T[j...j + K[i] - 1] = T[x...x + K[i] - 1] \ \wedge \ x < j\}, \text{ otherwise} \end{cases}$$

Consider using this data structure when we want to find all $K[\widehat{i}]$-length closing-stem halves for the opening-stem half $\mathcal{O} = T[\widehat{j}...\widehat{j} + K[\widehat{i}] - 1]$. In the case that $FSM[\widehat{i}][\widehat{j}] = \emptyset$, then there are no such closing-stems. In the case that $|FSM[\widehat{i}][\widehat{j}]| = 1$ and $FSM[\widehat{i}][\widehat{j}][1] < 0$, then we know that $Q = FSM[\widehat{i}][-FSM[\widehat{i}][\widehat{j}][1]]$ stores all closing-stem halves for an earlier opening-stem; our only task is to report the appropriate closing-stem halves for $\mathcal{O}$ at $\widehat{j}$ in T, i.e. $q \in Q$ such that $\widehat{j} + K[\widehat{i}] - 1 < q$. Otherwise, either $|FSM[\widehat{i}][\widehat{j}]| > 1$ or $|FSM[\widehat{i}][\widehat{j}]| = 1$ and $FSM[\widehat{i}][\widehat{j}][1] > 0$, so we have encountered the furthest previous occurrence of the opening-stem half $\mathcal{O}$; here, $FSM[\widehat{i}][\widehat{j}]$ stores the indices of all closing-stem halves. By accessing closing-stem halves of opening-stem halves $\mathcal{O}$ via the results from the

furthest previous occurrence of $\mathcal{O}$, space is saved. The space complexity of the data structure is formalized below.

Lemma 7.2.3 *Given the n-length string T and the set of considered stem lengths K, the space required by FSM is in $O(n|K|)$.*

Proof Consider one array $FSM[i]$ for some $1 \leq i \leq |K|$. By Definition 7.2.2, we have that each $FSM[i][j]$, $1 \leq j \leq n$, is either (1) a single negated integral index pointing to the furthest previous occurrence of the opening-stem half $T[j...j + K[i] - 1]$ or (2) a set of indices denoting the location of all closing-stem halves. Since a unique opening-stem half $\mathcal{O}$ has a unique closing-stem half $\mathcal{C}$ by Definition 7.2.1, then, in the worst case, there are exactly n elements contributed by (2) across all $FSM[i][j]$ for the considered i. In the worst case that all of the n elements appear in one set $FSM[i][j]$, then exactly $n-1$ elements are contributed by (1) for the remaining $FSM[i][h]$ with $1 \leq h \leq n$ and $h \neq j$. Therefore, at most $2n - 1$ elements are needed for each array $FSM[i]$. Since there are $|K|$ such arrays, then $O(n|K|)$ space is required. □

A naïve $O(n^3|K|)$ algorithm to construct the FSM would, for each of the $|K|$ rows in FSM, perform an $O(nk)$ search (with $k \in K$) to find the closing-stem halves for a single opening-stem half, since in the worst case, all of the opening-stem halves are unique in T and $k \in O(n)$. By expediting the pattern matching via the *border* array [17, 87], the algorithm can be improved to $O(n^2|K|)$. To more efficiently compute the FSM, we must compute some new data structures. In the following, we introduce and construct the Furthest Previous Non-Overlapping Factor (FPF) and the Furthest Previous Factor (FPF) arrays. Then, we use these new arrays in our improved construction of the FSM.

Furthest Previous Non-Overlapping Factor

Motivated by string factor related data structures, we define the Furthest Previous Non-Overlapping Factor ($FPnF_k$) to store, for each k-block in T, the index of the first non-overlapping occurrence of that k-block in T.

Definition 7.2.4 Furthest previous non-overlapping factor ($FPnF$): *For an n-length string T and a chosen integer $1 \leq k \leq \lfloor \frac{n}{2} \rfloor$, the $FPnF_k$ array stores an index x, for each $1 \leq i \leq n$, that locates the furthest previous non-overlapping occurrence of $T[i...i + k - 1]$ in*

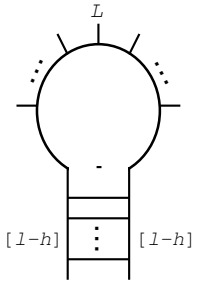

Figure 7.4: Hairpin query.

T. *More formally,* $FPnF_k[i] = \min\{x \mid T[x...x+k-1] = T[i...i+k-1] \ \wedge \ x+k-1 < i\}$. *In the case that no such* x *exists, define* $FPnF_k[i] = 0$.

Omitting the non-overlapping guard yields the Furthest Previous Factor (FPF).

Definition 7.2.5 Furthest previous factor (FPF): *For a chosen* k *and an* n*-length string* T*, the* FPF_k *array stores an index* x*, for each* $1 \leq i \leq n$*, that locates the furthest previous occurrence of* $T[i...i+k-1]$ *in* T*. More formally,* $FPF_k[i] = \min\{x \mid T[x...x+k-1] = T[i...i+k-1] \ \wedge \ x < i\}$. *In the case that no such* x *exists, define* $FPF_k[i] = 0$.

To construct the $FPnF$ and FPF, we can first construct the $FPnF$ and obtain FPF by omitting the non-overlapping restrictions. In a naïve algorithm, we would compute $FPnF$ by considering the $n-k+1$ total k-blocks, which begin at i, and compute the earliest exact match of the k-block in the text via a left-to-right scan. This algorithm would require $O(n^2k)$ time. We can improve this to $O(n^2)$ time using a *border* array for pattern matching [17, 87]. In the following, we improve on this construction time of $FPnF$ and note how to modify the construction to also obtain the FPF.

—Construction

To construct the $FPnF_k$, we provide a novel application of p-string theory. Consider a string T from the alphabet A. The $FPnF_k$ stores, for each ith k-block $T[i...i+k-1]$, the earliest index in T in which the k-block occurred in a non-overlapping fashion. When $\Sigma = \emptyset$ and $\Pi = A$, the $\texttt{prev}(T)$ stores, for each ith symbol $T[i]$, the distance to the previous $T[i]$ in T. If the `prev` would record the *first* previous occurrence rather than the *immediate* previous occurrence, it would be a special flavor of $FPnF_k$ where $k = 1$. Our method, shown in Algorithm 7-1, exploits the `prev` encoding to construct $FPnF_k$. Consider a new

Algorithm 7-1. Constructing $FPnF_k$ array.

```
1  int[n] construct_FPnF_k(int k,char T[n]){
2       int FPnF_k[n], T̂[n],i=1,j,v,z=0
3       int SA[n]=construct_SA(T), LCP[n]=construct_LCP(T)
4       do{
5            v=z++, T̂[SA[i]]=v
6            while(i<n ∧ LCP[i+1]≥k){
7                 T̂[SA[i+1]]=v, i++
8            }i++
9       }while(i≤n)
10      Σ=∅, Π=A, FPnF_k=prev(T̂)
11      for(i=1 to n){
12           j=FPnF_k[i]
13           if(j>0){
14                FPnF_k[i]=i−j
15                if(FPnF_k[FPnF_k[i]]>0)
16                     FPnF_k[i]=FPnF_k[FPnF_k[i]]
17           }
18      }for(i=1 to n){
19           j=FPnF_k[i]
20           if(j>0 ∧ j+k−1≥i) FPnF_k[i]=0
21      }return FPnF_k
22 }
```

array $\widehat{T}$ where we assign integers to the $O(n)$ k-blocks of T such that two integers are equal only when the corresponding k-blocks are equal. When $\Sigma = \emptyset$ and $\Pi = A$, we can use $\texttt{prev}(\widehat{T})$ to find not only the previous occurrence of the k-blocks, but via a left-to-right scan, we can successively use the previous occurrences to find the furthest previous occurrence of each k-block. After we find the first occurrence of the k-block, we only must check the non-overlapping condition. The following lemma formalizes the complexity of the algorithm, which is dominated by the computation $\texttt{prev}(\widehat{T})$.

Lemma 7.2.6 *Given an n-length T from alphabet A and parameter k, the $FPnF_k$ array is constructed in $O(n \log n)$ time and $O(n)$ extra space.*

Proof The SA and LCP on T require $O(n)$ time and $O(n)$ space to construct (see [1]). Also, the desired array $FPnF_k$ requires $O(n)$ space. Constructing $\widehat{T}$ in lines 4-9 clearly requires $O(n)$ time since both nested loops are controlled by a variable that is incremented by both loops. The alphabet assignment in line 10 is simply an $O(1)$ operation via pointers. Since $\widehat{T}$ is from a new alphabet of size $O(n)$ (because the variable z is incremented at most n

times), then `prev` is constructed in $O(n \log n)$ time with $O(n)$ extra space by Lemma 2.1.5. The time required to find the furthest previous indices for each k-block (lines 11-17) and the time required to check the non-overlapping condition (lines 18-20) clearly require a single $O(n)$ scan. Thus, $FPnF_k$ requires $O(n \log n)$ time and $O(n)$ extra space. □

In passing, we note that by omitting the last loop in Algorithm 7-1, we remove the non-overlapping condition and construct the FPF_k array. Even though we will improve the time complexity of the $FPnF$ and FPF constructions, we highlight that the previous construction is a new application of p-string theory, which traditionally deals with detecting source code redundancy [13, 98] and similar biological sequences [86].

—Improved Construction

We now present a more efficient solution to construct the n-length $FPnF_k$ array using the suffix tree ST for the n-length text T. The idea is to introduce computations in addition to a typical preorder traversal of the ST, in which, after we reach a k-block, i.e. a collection of suffixes sharing a prefix of k symbols, we collect these suffix indices and report the first index (the furthest previous occurrence) for all such k-blocks in T. The algorithm is as follows:

- Step (1): Initially, set $FPnF_k = \{0, 0, ..., 0\}$.
- Step (2): Now, construct the suffix tree ST for T.
- Step (3a): Perform a preorder traversal on ST (with the root as depth-0). In addition to the traversal, when visiting a new node n_j at depth-k, do:
  - (3a*) Set $S = \emptyset$. For each leaf, i.e. the suffix index say i, reached as a descendant from node n_j in the ST, we append this to S via $S = S \cup i$. Upon encountering n_j again in the traversal, the array S will have the list of all the suffixes beginning with the same k-length prefix. By scanning $S = \{s_1, s_2, ..., s_q\}$, we can determine the earliest occurrence of the currently traversed prefix in T at $z = \min(S)$. That is, we can determine elements $FPnF_k[s_v]$, for each $1 \leq v \leq q$ with $s_v \neq z$, by setting $FPnF[s_v] = z$ if $z+k-1 < s_v$. After these $|S|-1$ elements are populated, the preorder traversal of all children of n_j is complete. Continue to (3b).

- Step (3b): Continue with the preorder traversal. When a different n_j at depth-k is encountered, we execute (3a*).

The complexity of the algorithm is analyzed in the following.

Theorem 7.2.7 *Given an n-length T and parameter k, the $FPnF_k$ array is constructed in $O(n)$ time and $O(n)$ space.*

Proof The time needed for initialization in step (1) is in $O(n)$. The ST construction for T in step (2) requires $O(n)$ time and $O(n)$ space (see [1]). In step (3), all of the work is clearly done alongside an $O(n)$ preorder traversal. All that is left to consider is the time required to scan the S structures for the minimum occurrences z throughout the algorithm. Since the S is composed of the indices from the $O(n)$ leaves of the ST, then there are exactly n of the $S = S \cup i$ operations and so, each leaf node is considered exactly once. So, an $O(n)$ scan to find the minimum z of each S and populate the other $FPnF_k[s_v]$ is amortized across the entire algorithm. Thus, the $FPnF_k$ construction requires $O(n)$ time and $O(n)$ space. □

By omitting the check of $z+k-1 < s_v$ in step (3a*), we construct the FPF_k array. This omission does not alter the construction time or space complexity.

Corollary 7.2.8 *The FPF_k array is constructed in $O(n)$ time and $O(n)$ space.*

In passing, we note that, similar to this suffix tree solution, the $FPnF$ and FPF arrays may also be constructed via the suffix array (SA) and the longest common prefix (LCP) array.

FSM Construction

By looking at the definitions for $FPnF$ and FPF, and comparing them with FSM, we see a striking resemblance. In $FSM[i][j]$, we either point to the furthest previous occurrence of a $K[i]$-length opening-stem half, i.e. exactly $FPF_{K[i]}[j]$, or store the set of indices of the forward, non-overlapping, complementary $K[i]$-length closing-stem halves, a variation of $FPnF_{K[i]}$. In the following, we use our $FPnF$ related data structures to construct FSM.

Each $FSM[i][j]$ either (a) contains a set of indices for the $K[i]$-length closing-stem halves $\mathcal{C} = \texttt{reverse}(\texttt{complement}(T[j...j+K[i]-1]))$ that correspond to the opening-stem half $\mathcal{O} = T[j...j+K[i]-1]$ if $T[j...j+K[i]-1]$ is the first occurrence of $\mathcal{O}$ in T, (b) contains

$\emptyset$ if (a) yields no such indices, or otherwise, (c) contains the negated index of the furthest previous occurrence of $\mathcal{O}$ in T. We construct $FSM[i][j]$ individually by row $FSM[i]$ for $K[i]$. Initially $i = 1$.

- Step (1): Define $\overline{A} = \texttt{negate}(A)$ to negate each element of A in $\overline{A}$. Construct SA_T, LCP_T, and $FPF^T_{K[i]} = \texttt{construct\_FPF}_k(K[i], T, SA_T, LCP_T)$. We handle case (c) by setting $FSM[i] = \texttt{negate}(FPF^T_{K[i]})$, which will populate all $FSM[i][j]$ for an i and every j with the negated index of the furthest previous occurrence of the $K[i]$-length opening-stem half $\mathcal{O}$ in T.

- Step (2): Here, we handle the case (a). Let $crT = \texttt{reverse}(\texttt{complement}(T))$ and $\widehat{T} = T \circ \$_1 \circ crT$. Now, construct $SA_{\widehat{T}}$ and $LCP_{\widehat{T}}$. Let S be a stack with `push` and `pop` operations, where initially $S = \emptyset$. Observation (*): By using $\widehat{T}$, it is guaranteed that for each opening-stem from the beginning half of $\widehat{T}$, there will be at least one closing-stem located in the ending half of $\widehat{T}$. The trick is to use the $LCP_{\widehat{T}}$ and (*) to group closing-stem halves in T with the guaranteed occurrence(s) O in the ending half of $\widehat{T}$ and use this O to determine where the corresponding opening-stem is in T.

  - Step (2.1) Let $d = -1$. Perform one scan through $LCP_{\widehat{T}}$ to collect all suffixes in T (suffix indices in the range $[1, n]$) that match at least $K[i]$ symbols and `push` them on S. When we find a suffix in the ending half of $\widehat{T}$ (suffix indices in the range $[n + 2, |\widehat{T}|]$), say at t, that matches at least $K[i]$ symbols, set $d = SA_{\widehat{T}}[t]$. Continue this until we find a suffix that matches less than $K[i]$ symbols to end the grouping. Now, S has the indices of closing-stems in T that should be paired with the furthest previous occurrence of $T[l...l + K[i] - 1]$ with $l = |\widehat{T}| - d - K[i] + 2$. We can find this using $f = l$ if $FPF^T_{K[i]}[l] = 0$ or $f = FPF^T_{K[i]}[l]$ otherwise. Until S is empty, `pop` all elements with values less than $|T|$ into V. Only a subset $\overline{V}$ of V is actually a closing-stem half for $\mathcal{O}$, i.e. $\overline{V} = \{v \mid v \in V \ \wedge \ f + K[i] - 1 < v\}$. Note that we can collect $\overline{V}$ from V with a simple scan to yield an unsorted $\overline{V}$; we will sort this efficiently in step (4). Set $FSM[i][f] = \overline{V}$ to enforce (a) and continue the scan in (2.1) until all elements in $LCP_{\widehat{T}}$ are considered once.

- Step (3): Now, we handle the final case (b). From the previous steps, the elements where $FSM[i][j] = 0$ (for the considered i and any j) represent those furthest previ-

ous opening-stem halves in T where no closing-stem half exists. Thus, we set these $FSM[i][j] = \emptyset$ to uphold (b).

- Step (4): Finally, sort the index sets in the current row $FSM[i]$. Let $M = \emptyset$. For $1 \leq z \leq n$, when $|FSM[i][z]| \geq 1 \ \wedge \ FSM[i][z][1] \geq 1$, let $M = M \ \cup \ (FSM[i][z][y], z)$ for each y with $1 \leq y \leq |FSM[i][z]|$, then let $FSM[i][z] = \emptyset$. Now M has all pairs (c, o), within the row $FSM[i]$, with closing-stem half indices c and the corresponding opening-stem half index o. Perform one radix sort on the c attribute of the pairs in M. Work left-to-right on the sorted M and set $FSM[i][M[w].o] = FSM[i][M[w].o] \ \cup \ M[w].c$ with $1 \leq w \leq |M|$. Now, each set in the row $FSM[i]$ has been sorted. Afterwards, each condition (a), (b), and (c) is handled for the row $FSM[i]$. Now increment i and if $i \leq |K|$, consider the next row by (1). Otherwise, FSM is complete.

The time and space complexities of the algorithm are formalized below.

Theorem 7.2.9 *Given the n-length T and the set of stem lengths K, the Forward Stem Matrix (FSM) is constructed in $O(n|K|)$ time using $O(n)$ extra space.*

Proof Consider the work performed for each of the $|K|$ rows to construct FSM. Step (1) requires $O(n)$ time, since SA_T and LCP_T are constructed in linear time (see [1]), $FPF^T_{K[i]}$ is constructed in linear time by Corollary 7.2.8, and `negate` is an $O(n)$ operation. In step (2), crT is built in linear time due to the $O(n)$ operations `reverse` and `complement`. Also, $SA_{\widehat{T}}$ and $LCP_{\widehat{T}}$ are linear time constructions (see [1]). In this step, a single $O(n)$ scan is done on the $LCP_{\widehat{T}}$, pushing indices onto S that represent suffixes that match at least $K[i]$ symbols. When $K[i]$ symbols do not match, S is popped onto V and the resulting indices are selected to appear in an element of FSM. Here, exactly n elements are pushed onto S, popped, and selected to appear in $FSM[i][j]$ throughout the entire scan. Thus, this work is amortized across the entire $O(n)$ scan. Then, a simple $O(n)$ scan is done in step (3). By Lemma 7.2.3, step (4) will collect $O(n)$ pairs into M, which are composed of indices from the alphabet $O(n)$. So, a radix sort of M requires $O(n)$ time. Scanning M to repopulate the $FSM[i]$ entries is also an $O(n)$ operation. Since there are $|K|$ rows in FSM, then the algorithm constructs FSM in $O(n|K|)$ time. In terms of extra space (beyond the FSM), the algorithm shares the following structures when computing each row of FSM: the SA, LCP, and FPF arrays based on $|T|, |crT|, |\widehat{T}| \in O(n)$ and the S, V, and $\overline{V}$ based on $|\widehat{T}| \in O(n)$. Thus, $O(n)$ extra space is used. $\square$

Applications

—Finding Hairpins with a Known Loop Sequence

A hairpin is made up of a stem and a loop (see Figure 7.1). The hairpin structure [90] is of great significance in guiding RNA folding, assisting in protein recognition, and even protecting messenger RNA. Consider the hairpin structure query in Figure 7.4 where the loop pattern is known and the stem sequence is unknown with a length in some range. Formally, we address the problem of finding all hairpins H in the RNA sequence T, where H has a known loop sequence L and an unknown stem sequence with length in the range $[l, h]$ with $h \geq l$. Here, we find these hairpins in an RNA sequence using FSM.

Given SA_T, LCP_T, and FSM with $K = \{l, l+1, ..., h-1, h\}$, the general method is to use SA_T and LCP_T to find all occurrences of the loop L in T and using these occurrences, oracle FSM elements and report matches when there exists an opening-stem half just before the occurrence of L and a corresponding closing-stem half just after the occurrence of L.

- Step (1): Use the SA_T and LCP_T to find the range $[occ_L, occ_H]$ in SA_T where all suffixes have the prefix L. In the case that $occ_H \leq 0$ or $occ_L \leq 0$, there does not exist a match of L and so, no hairpins H with the given loop exist in T. Otherwise, let $i = occ_L$ and continue to step (2).

- Step (2): This step requires that we find some $K[j]$-length stem surrounding the considered occurrence L at $T[SA_T[i]...SA_T[i] + |L| - 1]$. Observation (*): For any stem with length $u > 1$, all of the corresponding length suffixes of the opening-stem half and prefixes of the closing-stem half are also stems. Since for each L occurrence, there can be at most one stem of each length $[l, h]$, then the goal is to find the longest such stem and simply report that also smaller stems exist for this occurrence of L. To do this, we need nested binary searches: the outer binary search varies the stem length, say g, and the inner binary search oracles an element of FSM to say whether or not the stem length g exists around L. Set the boundaries of the outer binary search to the range of $[l, h]$ in K, i.e. $s = 1$ and $e = h - l + 1$, and initially say that no maximum stem exists, i.e. $k = 0$.

  - Step (2.1) If $s \leq 0$ or $e \leq 0$ or $s > e$, then the outer binary search is complete; now, report the results by continuing to step (2.2). Otherwise, set $m = \lfloor \frac{s+e}{2} \rfloor$. The next

task is to see if a stem of length $K[m]$ surrounds the loop L at $T[SA_T[i]...SA_T[i]+|L|-1]$, i.e. the opening-stem half of length $K[m]$ must exist at $a = SA_T[i]-K[m]$ and the closing-stem must start at $b = SA_T[i] + |L|$. Let $Q = FSM[m][a]$. From Definition 7.2.2, we can find a $K[m]$-length closing-stem starting at b for a $K[m]$-length opening-stem starting at a. If $|Q| = 1$ and $Q[1] < 0$, then let $S = FSM[-(Q[1])][a]$; otherwise, let $S = Q$. Binary search for b in S. If b exists, then set $k = K[m]$ if $K[m] > k$ and consider larger possible stem lengths, i.e. set $s = m + 1$ and go back to step (2.1). Otherwise, b does not exist, so we consider smaller stem lengths, i.e. $e = m - 1$ and go back to step (2.1).

- Step (2.2): If still $k = 0$, then no stem of any length was found to surround this occurrence of loop L; proceed to step (3) to try to find a hairpin for the next occurrence of L. If the maximum length $k = l$, then report $(l, SA_T[i])$, signifying that there exists a hairpin H in T where the opening-stem of length l is at $SA_T[i] - l$ in T that is followed by L and the length l closing-stem. Otherwise, the maximum length $k > l$, so, report $(l, k, SA_T[i])$, signifying that there exists hairpins in T with stem lengths in the range $[l, k]$ with L starting at $SA_T[i]$ in T, i.e. the hairpins with opening-stem length q, for $l \leq q \leq k$, that start at $SA_T[i] - q$ in T are followed by L and a q-length closing-stem.

- Step (3): We increment i and if $i \leq occ_H$, we loop to step (2) to consider finding stems for $T[SA_T[i]...SA_T[i] + |L| - 1]$, the currently considered occurrence of loop L. Otherwise, there are no more occurrences of L to consider and all H in T have already been reported.

The time complexity for the previously detailed algorithm follows.

Theorem 7.2.10 *Given SA_T, LCP_T, and FSM on T with $K = \{l, l+1, ..., h-1, h\}$, the act of finding all hairpins H (with a known loop sequence L and an unknown stem sequence with length in the range $[l, h]$) in the RNA sequence T can be accomplished in $O(\max\{|L|, \eta_L \log(h-l) \log n\})$ time with η_L as the number of times that L occurs in T.*

Proof Step (1) executes in $O(|L| + \log n + \eta_L)$ time (see [72]). For step (2), each of the $O(\log(h-l))$ individual iterations from the outer binary search executes the inner binary search in $O(\log(\max\{|FSM[c][d]| \mid 1 \leq c \leq |K| \ \wedge \ 1 \leq d \leq n\})) \in O(\log n)$ time (by the

proof of Lemma 7.2.3). Thus, step (2) executes in $O(\log(h-l)\log n)$ time. Lastly, step (3) loops to step (2) a total of η_L times. Collectively, the time is in $O(|L| + \log n + \eta_L + \eta_L(\log(h-l)\log n)) \in O(\max\{|L|, \eta_L \log(h-l)\log n\})$. □

We acknowledge that by scanning the FSM on the stem lengths K and modifying each index set, i.e. where $|FSM[i][j]| \geq 1 \wedge FSM[i][j][1] \geq 1$ for $1 \leq i \leq |K|$ and $1 \leq j \leq n$, with a two-level hash scheme for perfect hashing [33], we can omit the $\log n$ work of the innermost binary search in the previous hairpin detection algorithm, since a query into each index set will only require $O(1)$ time. Given a satisfactory collection of hash functions, the extra work to build and store hash tables will not change the time or space complexity of the FSM. Thus, the hairpin detection algorithm may be improved to require $O(\max\{|L|, \eta_L \log(h-l)\})$ time. In practice, this improvement uses some extra resources in the FSM construction to offer faster querying to the FSM for applications.

Without the FSM, the act of finding all H in T would also begin by finding all L in T (see step (1)). For each of the η_L occurrences, we begin by matching stems with lengths $1, 2, ..., (l-1)$ that surround L and we continue matching until we have attempted to find an h-length stem surrounding L. When discovering any stems surrounding L with a length in the range $[l, h]$, we report an occurrence of H in T during this sequential matching process. This algorithm requires $O(\max\{|L|, \log n, \eta_L h\})$ time. There are two key problems here: (A) for large stem lengths $h \in O(n)$, the algorithm executes in time based on $|T|$ and further, cannot take advantage of cases when both $l, h \in O(n)$ and (B) the problems of (A) are magnified when we extend the result to more complex RNA secondary structures. The FSM data structure expedites the time to find stems so that matching complex RNA secondary structures with many fragments like Figure 7.3 becomes a problem of finding non-stems (bulges, loops, etc.) that align correctly rather than finding a large number of possible structures and filtering by stems with expensive pattern matching routines.

In passing, we note that the FSM data structure may be augmented with additional data for validating the RNA structures. For example, each opening-stem and closing-stem in the FSM can include data on the probability that the stem occurs in nature. When building an RNA structure, the probabilities of the stems can be assessed to determine how frequently a structure may exist; those structures with a collection of stems occurring together with some probability p less than a threshold θ may be removed from further consideration. We may further include a separate table of conditional stem probabilities to further validate how

frequently a collection of stems occur together in a structure; this addition will help filter infrequently occurring and potentially invalid RNA structures. A further study is required to analyze how the FSM may be augmented to apply more biological approaches to the validation.

—Finding All Hairpins with Arbitrary Loops

We now find all hairpins in the n-length RNA sequence R with a minimum loop size l and a stem length in the range $[s_L, s_H]$ via the following algorithm. We want to report hairpins that satisfy three conditions: (a) the stem size is in the range $[s_L, s_H]$, (b) the loop size is at least l, and (c) the stem complementary base pairings are inner maximal, in that the stem pairing does not extend to the loop. Note that (c) must be true because if the stems may be extended, these extended base pairs could not be considered part of the hairpin loop.

- Step (1): Identify all possible stems with lengths in the range $[s_L, s_H]$ by constructing the FSM on R with $K = \{s_L, s_L + 1, ..., s_H\}$ (this handles condition (a)). Define H as a $|K| \times n$ matrix to store in each $H[i][j]$ the hairpins that will be generated from $FSM[i][j]$, grouped by the opening stem half $R[j...j+K[i]-1]$. Set $i = 1$ and $j = |K|$.

- Step (2): If $FSM[i][j] = \emptyset$, then there cannot be any hairpins for this opening stem half; go to (4). If $|FSM[i][j]| = 1$ and $FSM[i][j] < 0$, then we already reported hairpins with this opening-stem half before, so set $\widehat{Q} = FSM[i][-FSM[i][j][1]]$. Otherwise, we need to find hairpins considering these opening-stems, so let $\widehat{Q} = FSM[i][j]$. In both cases, we need to binary search in $\widehat{Q}$ to find $p = j + K[i] + l$ and list in a new Q all elements $\widehat{p} \in \widehat{Q}$ such that $\widehat{p} \geq p$ and go to (3) with this Q (this handles condition (b)). Continue to (3).

- Step (3): We report hairpins with the $K[i]$-length opening-stem half at j in R with all closing-stem halves $Q[y]$ for $1 \leq y \leq |Q|$ only when the complementary base pairing between the opening-stem half $R[j...j + K[i] - 1]$ and the closing-stem half $R[x...x + K[i] - 1]$ is inner maximal, where $x = Q[y]$. Let t be the index of some opening-stem half, u be the index of some closing-stem half, and v be the length of this stem. We can determine that the pairing is inner maximal by defining the function $\texttt{imaximal}(t, u, v)$ to return true if $\texttt{complement}(R[t+v]) \neq R[u-1]$ and return false otherwise. So, when $\texttt{imaximal}(j, x, K[i])$ returns true, the pairing is inner maximal (handles condition (c))

and we report a hairpin $\widehat{h} = (j, x, K[i])$ (and record the hairpin via $H[i][j] = H[i][j] \cup \widehat{h}$), which signifies that the $K[i]$-length opening-stem half begins at position j, followed by the $(x - j - K[i])$-length loop, followed by the $K[i]$-length closing-stem half beginning at position x in R. When all forward stems have been considered in Q, go to (4).

- Step (4): If $i < n$, then increment i and go to (2). Else, decrement j and if $j \geq 1$, reset $i = 1$ and go to (2). When $j = 0$, all hairpin triples have been both reported and also logged in H.

The analysis of the algorithm follows.

Theorem 7.2.11 *Given the n-length RNA string R, all η hairpins with stem size in the range $[s_L, s_H]$ and with loop size at least l can be found in $O(n^2\Delta)$ time and $O(n\Delta + \eta)$ space, where $\Delta = s_H - s_L + 1$.*

Proof Step (1) requires $O(n\Delta)$ time by Theorem 7.2.9. Steps (2), (3), and (4) report all hairpins by scanning each element of FSM, with the element types (i), (ii), and (iii). (i) For elements with no forward stem information, i.e. $FSM[i][j] = \emptyset$, we omit these in $O(1)$ time in step (2). (ii) For $FSM[i][j]$ directly containing forward stem information, we binary search once to find a minimum index $p = j + K[i] + l$ where a forward stem may occur in ($O(\log(|FSM[i][j]|))$ time since $K[j]$ and l do not impact the binary search time), then check at most two pairs of elements to ensure that the stem is an inner maximal match for each of the, say z, indices at least as large as $p = j + K[i] + l$, and finally report the hairpins (overall $O(z)$ time). Since small $K[j]$ and l produce a larger z in the worst case, we can bound all of these operations for one $FSM[i][j]$ by $O(|FSM[i][j]|)$. (iii) For $FSM[i][j]$ referencing previous forward stem information, we consider $FSM[i][-FSM[i][j][1]]$ in (ii). Overall, since there are $O(n\Delta)$ total elements in FSM with $O(n)$ elements in each row by Lemma 7.2.3, then processing each (ii) or (iii) considers order $O(n^2)$ hairpins (compressed by FSM). So, $O(n^2\Delta)$ time is required. In terms of space, since the FSM on R with K requires $O(n\Delta)$ space by Lemma 7.2.3 and since we store the η hairpins in H as they are reported, then $O(n\Delta + \eta)$ space is required. □

—Detecting All Pseudoknots

A pseudoknot is an RNA secondary structure with multiple hairpins, which are structures with a stem and a loop (see Figure 7.1). For a pseudoknot, the hairpin loops, say L_1 and

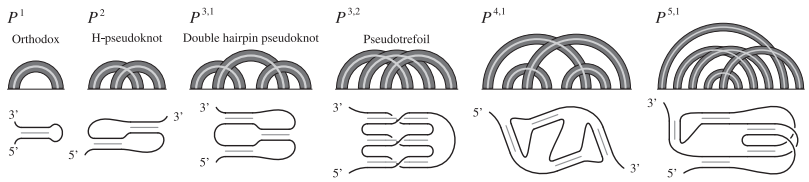

Figure 7.5: Pseudoknot configurations (figure from [83]), where P^u and $P^{u,x}$ denote pseudoknots with u stems and configuration type x.

L_2, form a complementary bond, i.e. L_1 and L_2 form a stem. In one context, L_1 and L_2 are loops of a hairpin and in another context, sections of L_1 and L_2 are the stems of another hairpin. This is the essence of a pseudoknot. The process of dealing with all pseudoknots is a combinatorial challenge with complicated processes and exponential resource complexities. Due to this difficulty, many contexts do not handle pseudoknots at all, or only consider small pseudoknots. Here, we provide a string based algorithm answer the following: given an RNA sequence R of length n, find all pseudoknots that R forms. We address this with a two step process: (1) find all hairpins with an arbitrary loop and (2) report hairpin configurations that satisfy a pseudoknot form in Figure 7.5. This process is expedited (and conceptually simplified) with the assistance of the FSM data structure to find stem structures that form in R.

To find pseudoknots, we (a) find the hairpins, (b) ensure that the hairpins form the desired pseudoknot configuration, and (c) identify the segments of the hairpin loops that form complementary bonds. Consider the case of finding the pseudoknot configuration P^2 in Figure 7.5. Previously, we computed (a) in Theorem 7.2.11. Recall that each hairpin was reported in the form (x, y, z), signifying that the z-length opening-stem half begins at position x in the RNA sequence R, followed by the $(y - x - z)$-length loop, followed by the z-length closing-stem half beginning at position y in R. For (b), we choose two of these hairpins H_1 and H_2 and send them to a $\texttt{config}_{P^2}$ function to determine if the hairpins form the correct configuration. We define the function $\texttt{config}_{P^2}(H_1, H_2)$ to accept two hairpins where $H_1.x < H_2.x$ and returns true if $H_1.x + H_1.z - 1 < H_2.x + H_2.z - 1 < H_1.y + H_1.z - 1 < H_2.y + H_2.z - 1$ and returns false otherwise. The final step (c) is to determine the hairpin loops forming complementary base pairs and report the P^2 pseudoknots. We will report/represent the P^2 pseudoknots in the form $(H_1, H_2; S_1, S_2, ...)$, indicating that a P^2 pseudoknot is formed between hairpins H_1 and H_2 when one of the $S_{\mathcal{J}} = (o, c, g)$ hairpin loop pairings is made. To find these hairpin loop pairings, we define the `pair` function to report all maximal stem

pairings between the o opening-stem half indices from $R[H_1.x...H_1.x + H_1.z - 1]$ and the c closing-stem half indices from $R[H_2.y...H_2.y + H_2.z - 1]$ for possible g-length stems. This can be accomplished via the FSM and `imaximal` method leading to Theorem 7.2.11. We omit the details.

In terms of time, step (a) requires $O(n^2\Delta)$ by Theorem 7.2.11. Step (b) executes an $O(h) \in O(1)$ (since the number of hairpins is $h = 2$) $\texttt{config}_{P^2}$ function for each pair of the hairpins. We can bound (b) by $O(\binom{n^2\Delta}{2})$. Let $l_1 = H_1.y - H_1.x - H_1.z$ and $l_2 = H_2.y - H_2.x - H_2.z$ respectively be the lengths of the loops for hairpins H_1 and H_2. Then, the time for (c) is $O(\Delta l_1(\log n + l_2))$ for each of the $O(\binom{n^2\Delta}{2})$ possible hairpin pairs found by (b). The running time is thus $O(\Delta l_1(\log n + l_2)\binom{n^2\Delta}{2}) \in O(n^4\Delta^3 l_1(\log n + l_2))$. By analysis, since step (c) can be done as soon as we find any two hairpins in the proper configuration, we can immediately store and report the η_p total pseudoknots. So, we require $O(n\Delta + \eta + \eta_p)$ overall space, where η is the number of hairpins from step (a).

For configuration $P^{h,1}$ for larger h, we can extend the representation $(H_1, H_2, ..., H_h; S_1^1, S_2^1, ...; S_1^2, S_2^2, ...; ...; S_1^{h-1}, S_2^{h-1}, ...)$ and require that to report a pseudoknot with hairpins $H_1, H_2, ..., H_h$, exactly one hairpin loop pairing from $S^{\mathcal{K}}$, for each $\mathcal{K}$, is chosen such that none overlap. We can find all $P^{h,1}$ pseudoknots using the same philosophy as the P^2 case. Essentially, only the `config` and `pair` functions would need to be extended to support pseudoknot configurations with larger h. We report the general result below.

Theorem 7.2.12 *A hairpin requires a stem size in the range $[s_L, s_H]$ and loop size at least l. Let $\Delta = s_H - s_L + 1$. Given the n-length RNA string R, all $P^{h,1}$ pseudoknots with h hairpins can be found in $O(\frac{n^{2h+2}\Delta^{h+1}}{(h-1)!})$ time and $O(n\Delta + \eta + \eta_p)$ space, where η and η_p are respectively the number of such hairpins and pseudoknots in R.*

Note that the space is linear when Δ is constant and when η and η_p are linear for some RNA string R. Since all stem lengths may be reported, then possibly $\Delta = O(n)$ and so, the previous result would require $O(\frac{n^{3h+3}}{(h-1)!})$ time. Rodland [83] showed that although the number of possible pseudoknots in a random sequence could be exponential with respect to the sequence length, only a few of the possible configurations are observed in practical sequences. For instance, most pseudoknots found in real non-coding RNA are of the simpler types, namely the H-pseudoknots (P^2) and some double hairpin pseudoknots ($P^{3,1}$), and only a few cases with more complex configurations. See also [48]. Most practical pseudoknots also contained a few more stems than would be expected from a random sequence. These

observations suggest that we can safely assume that h, the number of hairpins in the pseudoknot, is a small constant, relative to n, the length of the sequence. Thus, our pseudoknot detection requires polynomial time.

—Other Applications

Though different variations of hairpin matching problems have been addressed in [74, 75, 89], our algorithm for detecting hairpins with a known loop sequence via FSM is very natural. We note that in addition to finding hairpins and pseudoknots, there are a multitude of other applications that the FSM can help address. For instance, the FSM can also be used to identify hairpins in an RNA sequence consisting of loops with wildcards, stems with wildcards, etc. In fact, the FSM can be used to find more sophisticated RNA secondary structure patterns (such as Figure 7.3) in a large RNA database or even assist in predicting the structure of an RNA given a sequence of nucleotides. These patterns can be composed of various RNA structural elements such as stems, loops, bulges, simple hairpins, etc. The patterns can also specify all, some, or none of the nucleotides present in each structural element. To find these RNA secondary structure patterns in a large database, we may take many approaches. One approach is to use the FSM to find a collection of stems matching between the pattern and areas of the database, and postprocess to determine if these stem regions coexist with the other structural elements of the pattern. Let us refer to the stems of an RNA secondary structure pattern and their locations as a *stem signature* of that pattern. Another approach is to use the FSM to first process the pattern and identify the simple hairpins, that is those hairpins that can be extracted from the pattern which look like Figure 7.4. Let us refer to the simple hairpins of an RNA secondary structure pattern and their locations as the *hairpin signature* of that pattern. Next, we can use the FSM to locate these simple hairpins in the database and finally, postprocess to determine if the locations of the hairpins allow the other RNA structural elements to coexist. We highlight that with the latter approach, the FSM is treated as *an index to hairpins*. That is, the FSM is oracled for natural and efficient access to simple hairpins with a stem and a loop. In a practical sense, using the FSM to locate regions of a signature ultimately requires resolving collisions between RNA secondary structure patterns sharing the same signature. Augmentations to the signature can reduce the probability of collisions and improve use in practice.

7.2.2 Experiments

To show the practical performance of our algorithms, we implemented the FSM construction and the hairpin (with known loop sequence) detection algorithm each in two ways. First, we implemented the FSM using the proposed algorithm with the SA, LCP, and $FPnF$-related arrays. The $FPnF$-related arrays were constructed using the SA and LCP. The hairpin detection was also implemented as proposed in this research with the FSM, SA, and LCP data structures. Second, we implemented the FSM construction in a semi-naïve way using a Boyer Moore (BM) pattern matcher [47, 87] to find both the first occurrence of opening-stem halves and all corresponding closing-stem halves. We also implemented the hairpin (with known loop sequence) detection algorithm, which executes like the proposed algorithm except that the loop sequence is found with BM and we replace the outer binary search to find the length of the stem by a sequential search. The programs were written in Java because of the natural way to represent the FSM with a matrix of jagged arrays. For convenience, we also use some basic functions from the Arrays and Collections classes; we could implement these basic utility functions for better performance. The algorithms were executed in a Cygwin environment running on a Dell Inspiron 570 desktop with 3.10 GHz clock speed and 8 GB RAM. Below, we discuss the performance of the algorithms on the following sequences from Rfam [46]: V01555.2 Epstein-Barr Virus (EBV), CR548612.1 Paramecium Tetraurelia Macronuclear (PTM), and DQ792504.1 Horsepox Virus (HPOX). Table 7.1 shows some characteristics of the aforementioned sequences.

First, we discuss the FSM construction time, which is composed of (1) the time for SA and LCP preprocessing and (2) the time for the FSM algorithm. Figure 7.6 shows the construction time for the FSM on prefixes of the EBV sequence with $K = \{1, 2, ..., 10\}$ using both the proposed approach (SA) and the semi-naïve approach (BM). It is obvious that over time, the proposed approach, which appears linear, performs better. Further, the EBV sequence has a large amount of repetition according to Table 7.1, which is quite taxing on the semi-naïve implementation. The improved construction is not impacted by this repetition. The FSM is constructed for the PTM sequence on $K = \{1, 2, ..., 5\}$ and $K = \{1, 2, ..., 10\}$ in Figure 7.7 and Figure 7.8, respectively. We see that changing the value of K, i.e. the considered stem lengths, alters the execution time. Recall that the running time for the FSM is directly related to the number of elements in the FSM (see Theorem 7.2.9 and Lemma 7.2.3). This is clear when observing the behavior between the FSM construction

Table 7.1: Characteristics of sequences, where $||FSM||_K$ denotes the total number of elements in the FSM data structure on $K = \{1, 2, ..., 10\}$.

T	$\|T\|$	$\texttt{max}(LCP_T)$	$\texttt{mean}(LCP_T)$	$\|\|FSM\|\|_K$
EBV	172281	32442	3071.4	2973170
HPOX	212633	71	8.6	3775665
PTM	984602	372	10.4	18746709

time in Figure 7.8 and the respective number of FSM elements in Figure 7.10. Further, we can bound the FSM construction times by $2n|K|$, which is the most possible elements in the FSM. In practice, we would achieve this bound in the worst case if, at each prefix of the sequence, the number of elements in the FSM is maximum. In the case of HPOX (see Figure 7.9), we see that the FSM algorithm actually executes faster than the SA and LCP linear time preprocessing. In our implementation, the provided Java sort methods are used along with some other Java utility functions, for convenience. Performance improvements for the FSM construction are possible by implementing these functions along with an integer `radixsort`.

Next, we discuss the time for hairpin detection. In this experiment, we use the FSM structures on $K = \{1, 2, ..., 10\}$ to find all hairpins in the form of Figure 7.4 with $L = \{A\}^s$, $l = K[1] = 1$, and $h = K[|K|] = 10$. In other words, we search for hairpins with a known loop sequence L as a run of s adenine symbols and with a stem length in the range of $[1, 10]$. In the case of the EBV sequence in Figure 7.11, we detect hairpins with both the proposed approach (SA) and the semi-naïve approach (BM). Notice that for both hairpin detection approaches, the running time is similar. We observe that the running time for hairpin detection is fast for all of the sequences considered (see Figure 7.12). In general, we see that hairpin detection is simple given the FSM. Thus, we may suggest that the FSM data structure is *an index to hairpins* in the form of Figure 7.4. In passing, we note that the way in which we find hairpins is exclusively symbol-based and additional biological validation may be required. Nonetheless, we can quickly provide a multitude of hairpins to a biological filter given the FSM. Also, extensions to the hairpin detection algorithm can identify more sophisticated RNA secondary structures such as Figure 7.3.

7.2.3 Conclusions

In this section, we propose the Forward Stem Matrix (FSM), which stores information regarding all k-length stem ($k \in K$) options within an n-length RNA sequence. We pro-

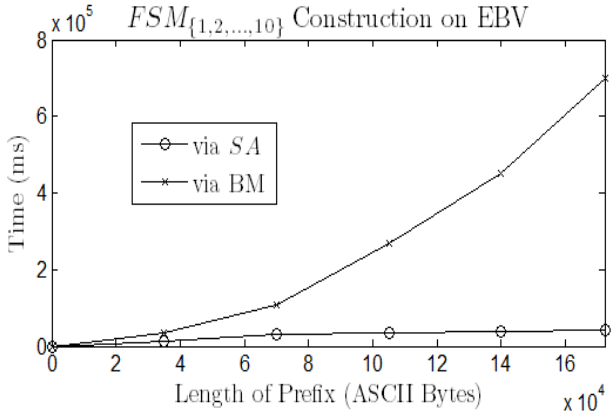

Figure 7.6: FSM construction on EBV sequence.

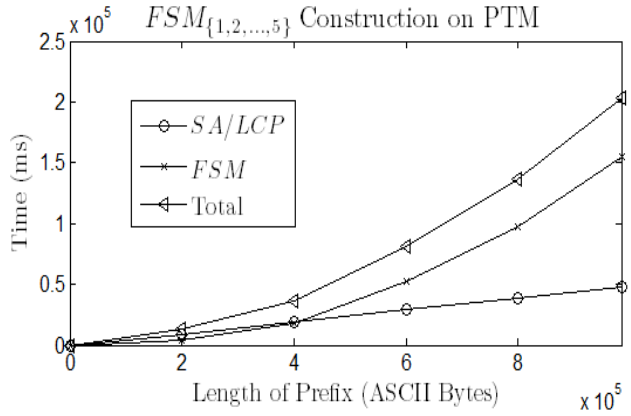

Figure 7.7: FSM construction on PTM sequence.

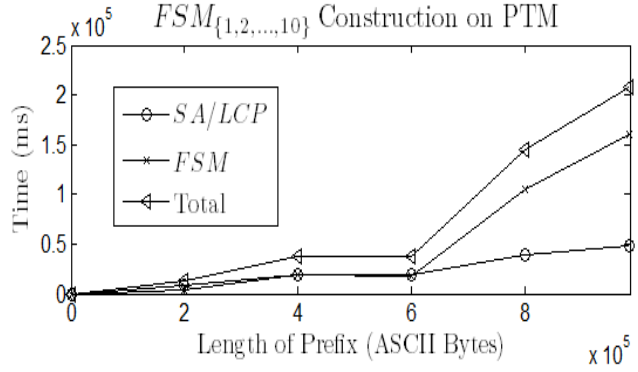

Figure 7.8: FSM construction on PTM sequence.

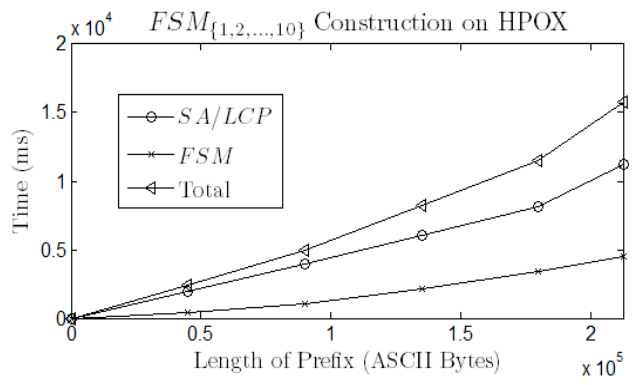

Figure 7.9: FSM construction on HPOX sequence.

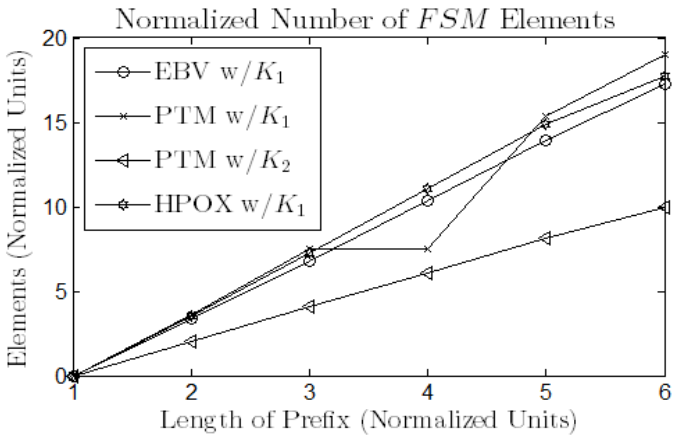

Figure 7.10: Number of elements in the FSM, where $K_1 = \{1, 2, ..., 10\}$ and $K_2 = \{1, 2, ..., 5\}$. Both the number of elements and the prefix length are normalized by the length of each sequence.

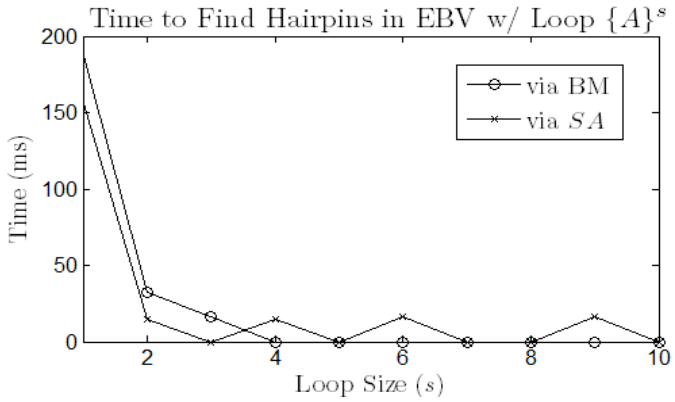

Figure 7.11: Time to find hairpins in EBV with $L = \{A\}^s$, $l = 1$, and $h = 10$ (see Figure 7.4).

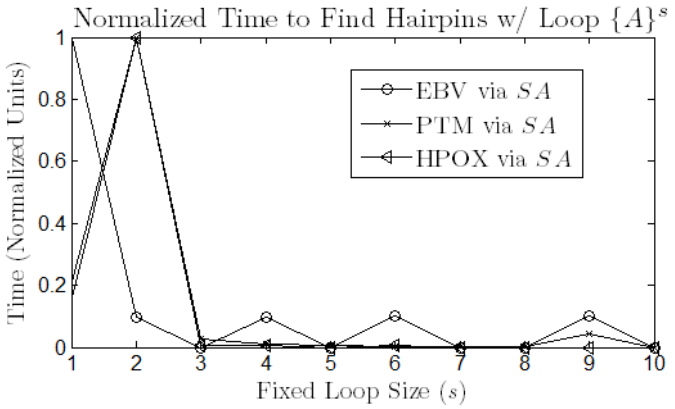

Figure 7.12: Time to find hairpins with $L = \{A\}^s$, $l = 1$, and $h = 10$ (see Figure 7.4), normalized by the maximum time to find hairpins within each sequence.

vide a linear $O(n|K|)$ construction for the FSM via suffix arrays and arrays related to the Longest Previous Factor (LPF), in particular, the Furthest Previous Non-Overlapping Factor ($FPnF$) and Furthest Previous Factor (FPF) arrays. We provide two new constructions for the $FPnF$ and FPF including a novel use of parameterized string (p-string) theory and an improved linear solution with suffix trees. Then, we devise methods to find hairpins and pseudoknots within an RNA sequence. Experimental results are included to show the empirical performance of the proposed data structure. Using the FSM, other RNA secondary structure problems can also be addressed. For future work, we are interested in extending our data structures to support non-Watson-Crick wobble pairings and applying the data structures to assist in RNA alignment. Also, we wish to investigate how to efficiently filter discovered RNA structures based on their biological relevance.

Chapter 8

Conclusions

The parameterized match (p-match) is a generalization of traditional string matching; the structural match (s-match) is a generalization of the p-match. By developing algorithms using the p-match/s-match and defining data structures to support the parameterized string (p-string) and structural string (s-string), we develop powerful solutions with the flexibility to support numerous problems, including parameterized duplication [12], RNA structural analysis [86], and traditional pattern matching problems. In this dissertation, we advance the state-of-the-art for p-string theory by developing data structures for p-strings/s-strings and using p-string/s-string theory in new and old contexts to address various applications.

8.1 Summary

First, we recall the parameterized longest previous factor ($pLPF$) and develop a taxonomy for longest factor problems. We show the connection between the construction of the $pLPF$ and the construction of various data structures in the taxonomy. Using the $pLPF$ construction as the foundation, we show how to efficiently construct important p-string oriented data structures such as the parameterized longest common prefix ($pLCP$) and the parameterized-border (*p-border*) array, in addition to the following popular data structures for traditional strings: the longest common prefix (LCP), the permuted-longest common prefix (*permuted-LCP*), the *border* array, and the prefix array (PA). Also, we propose a number of new data structures including the longest not-equal factor ($LneF$), longest reverse factor (LrF), longest factor (LF), and equivalent p-string extensions. The $pLPF$ framework is exploited to provide constructions of the aforementioned data structures as

well. In summary, there is an important connection between the $pLPF$ array and that of other longest factor problems, including several popular data structures. This allows us to exploit, i.e. reuse, the efficient $pLPF$ construction for numerous data structures, which is a transformative result that makes the p-string quite appealing to the string community and enlightening for software developers.

Next, we extend the *border* array for the s-string to support s-matching, i.e. construct the structural border (*s-border*) array. Earlier, we showed how to construct *border*-related arrays with the $pLPF$. Here, we provide direct constructions of *border*-related arrays. En route to improved constructions, we prove that even with the prefix and suffix intricacies of the s-string encodings, the basic *border* properties still hold for the *s-border*. Thus, we show a progression of constructions of the *s-border* from requiring a cubic construction time, to quadratic time, ultimately to linear time. This is a significant result because of the generalization of the s-string, i.e. we show how to modify the s-string alphabets to also construct the *p-border* and the traditional *border* arrays in linear time. Similar to efficient left-to-right matching and p-matching, the *s-border* gives another efficient way to s-match, in addition to the structural suffix tree [86] and the structural suffix array [16].

Then, we investigate the problem of p-matching on compressed texts, which is well studied for traditional strings, and for p-strings is studied in a slightly different context in terms of RLE strings [7]. We define the compressed parameterized pattern matching (compressed p-matching) problem to find all of the p-matches between a pattern P and text T, using only P and the compressed text T_c. We support p-matching by introducing parameterized compression (p-compression). That is, rather than compressing T to form the compressed T_c, we compress a new p-string encoding of T that is practical for compression. In essence, a p-string encoding is applied to T as a transformation prior to compression. Experimentally, we show that compressing this transformation is competitive with compressing T and in some cases, leads to even better compression. This is a significant contribution because it is a new application of p-string theory. Using p-compression, we develop a solution to the compressed p-matching problem for any general compression scheme where a partial symbol decompression function can be defined. Our results are examined for the specific case of Tunstall codes; we acknowledge that other compression schemes are possible.

The p-match is already a special and strict form of inexact matching. Next, we add another form of inexactness to the p-match by introducing/addressing two new variations of the p-match with indeterminate symbols. First, we propose the indeterminate parameterized

match (ip-match), which allows indeterminate holes in a p-string. We address this problem by extending the prefix array, a data structure known for traditional pattern matching with indeterminate symbols. Prior to addressing the ip-match, we introduce/construct the parameterized prefix array (pPA) and its succinct representation, the compact parameterized prefix array ($cpPA$). We initially construct the $cpPA$ via parameterized suffix structures and then, improve on the result with a novel construction via the $pLPF$ data structure. Later, we extend the $cpPA$ to solve the ip-match. Finally, we discuss applications for our data structures. Second, we propose the equivalence parameterized match (e-match), which incorporates equivalence classes of symbols within each p-string alphabet. We propose a method to perform the e-match using the p-match suffix array framework. Historically, the parameterized suffix array (pSA) and parameterized longest common prefix ($pLCP$) have suffered from direct constructions, i.e. without the use of the parameterized suffix tree, requiring quadratic theoretical time bounds in the worst case. The main problem is dealing with the dynamic parameterized suffixes (p-suffixes), which differ drastically from traditional suffixes. Whether or not improved direct pSA and $pLCP$ constructions exist has been posed as an open problem by the string community. In this research, we first prove a special relationship between p-suffixes and then propose the parameterized cover (p-cover). Together, this theory is exploited to construct the pSA in stages. We extend this idea to construct the pSA and $pLCP$ for an n-length text from alphabets Σ and Π in $O(n|\Pi|)$ time, breaking the previous worst-case theoretical time barrier of $O(n^2)$. Due to the generality of the e-match, our results support the e-match, p-match, and traditional matching.

Lastly, we explore applications in computational biology, specifically dealing with RNA secondary structures. Using the structural suffix array (sSA), we propose a solution to find RNA secondary structures in a text, based on a complex user query. Next, we define the Forward Stem Matrix (FSM) on K to permit quick access to RNA stem structures with length $k \in K$. To more compactly define and efficiently construct the FSM, we develop the Furthest Previous Non-Overlapping Factor ($FPnF$) and Furthest Previous Factor (FPF) arrays. We show how to first construct the $FPnF$ and FPF with a novel application of p-string theory. An improved solution is then given with suffix structures. The FPF-related arrays are then used to construct the FSM in time linear to its length, $O(n|K|)$. Like the s-string, we can use the FSM to analyze the structure of RNA. We then show how to use the FSM to find hairpin and pseudoknot structures in an RNA sequence. Experimental results for the FSM conclude the chapter.

8.2 Final Remarks

The overall theme of this work is to tackle the challenges of the p-string/s-string encodings and advocate for more general data structures, in order to support other intricate applications. Even though we differentiate between traditional and p-string/s-string solutions, we highlight and exploit the deep connection between traditional string and p-string/s-string data structures. Before this work, the p-string was mostly known for its role in plagiarism detection and analysis of biological sequences. Now, the p-string will also be known for exciting applications in compression, music, and inexact matching. In this work, we not only advance p-string theory and the capabilities of the p-match, but at times, we also present beautiful and novel applications of p-string theory. Our hope is that this work will trigger more interest in the use of generalized p-string/s-string solutions to address the myriad of problems in the string community.

Interesting topics of future work include establishing a connection between data structure taxonomies and the construction of other p-string data structures. In the way that our single $pLPF$ construction algorithm can yield many other data structures, we believe that this scheme can apply to other types of powerful p-string data structures. Also, the p-match and s-match are special types of inexact matches. In this work, we introduce further inexactness to the p-match via the e-match and ip-match schemes. A future research problem is to propose and address other ways to extend the inexactness of the p-match/s-match for applications in music. Lastly, the relationship between p-string/s-string theory and RNA analysis is evident in this work. An important research problem is to extend these RNA-oriented data structures to support non-Watson-Crick base pairings for more complex RNA queries.

References

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

[2] Adjeroh, D., Feng, J: Locating all tandem repeat families in a sequence. In: CSB'04, pp. 676-681 (2004)

[3] Adjeroh, D., Zhang, Y., Mukherjee, A., Powell, M., Bell, T.: DNA sequence compression using the Burrows-Wheeler Transform. In: CSB'02, pp. 303-313 (2002)

[4] Allali, J., Antoniou, P., Ferraro, P., Iliopoulos, C., Michalakopoulos, S.: Overlay problems for music and combinatorics. In: International Conference on Auditory Display, pp. 138-143 (2009)

[5] Allali, J., Sagot, M.: A new distance for high level RNA secondary structure comparison. IEEE/ACM Trans. Comput. Biology Bioinform. 2(1), 3-14 (2005)

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

[7] Apostolico, A., Erdős, P., Jüttner, A.: Parameterized searching with mismatches for run-length encoded strings. Theor. Comput. Sci. 454(5), 23-29 (2012)

[8] Apostolico, A., Erdős, P., Lewenstein, M.: Parameterized matching with mismatches. J. of Discrete Algorithms. 5(1), 135-140 (2007)

[9] Backofen, R., Siebert, S.: Fast detection of common sequence structure patterns in RNAs. J. of Discrete Algorithms. 5(2), 212-228 (2007)

[10] Bafna, V., Muthukrishnan, S., Ravi, R.: Computing similarity between RNA strings. In: CPM'95, pp. 1-16 (1995)

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

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

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

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

[15] Batey, R., Rambo, R., Doudna, J.: Tertiary motifs in RNA structure and folding. Angewandte Chemie International Edition. 38, 2326-2343 (1999)

[16] Beal, R.: Parameterized Strings: Algorithms and Data Structures. MS Thesis. West Virginia University (2011)

[17] Beal, R., Adjeroh, D.: Border array for structural strings. In: IWOCA'12, pp. 189-205 (2012)

[18] Beal, R., Adjeroh, D.: Compressed parameterized pattern matching. DCC'13, pp. 461-470. (2013)

[19] Beal, R., Adjeroh, D.: Parameterized longest previous factor. Theor. Comput. Sci. 437, 21-34 (2012)

[20] Beal, R., Adjeroh, D.: p-Suffix sorting as arithmetic coding. In: IWOCA'11, pp. 44-56, Springer (2011)

[21] Beal, R., Adjeroh, D.: p-Suffix sorting as arithmetic coding. J. of Discrete Algorithms. 16, 151-169 (2012)

[22] Beal, R., Adjeroh, D.: The structural border array. J. of Discrete Algorithms. 23, 98-112 (2013)

[23] Beal, R., Adjeroh, D.: Variations of the parameterized longest previous factor. J. of Discrete Algorithms. 16, 129-150 (2012)

[24] Beal, R., Adjeroh, D., Abbasi, A.: The forward stem matrix: An efficient data structure for finding hairpins in RNA secondary structures. In: ACM-BCB'13, pp. 576-585 (2013)

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

[26] Blanchet-Sadr, F., Bodnar, M., De Winkle, B.: New bounds and extended relations between prefix arrays, border arrays, undirected graphs, and indeterminate strings. In: STACS '14, pp. 162-173 (2014)

[27] Bland, W., Kucherov, G., Smyth, W.: Prefix table construction & conversion. In: IWOCA'13, pp. 41-53 (2013)

[28] Capriotti, E., Marti-Renom, M.: Computational RNA structure prediction. Current Bioinformatics. 3, 32-45 (2008)

[29] Chen, H., Condon, A., Jababri, H.: An O(n^5) algorithm for MFE prediction of kissing hairpins and 4-chains in nucleic acids. J. Comput. Biol. 16(6), 803-815 (2009)

[30] Christodoulakis, M, Iliopoulos, C. S., Sohel Rahman, M., Smyth, W. F.: Identifying rhythms in musical texts. Internat. J. Foundations of Computer Sci. 19(1), 37-52 (2008)

[31] Cole, R., Hariharan, R.: Faster suffix tree construction with missing suffix links. In: STOC'00, pp. 407-415, ACM, New York (2000)

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

[33] Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. McGraw-Hill, Boston (2002)

[34] Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)

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

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

[37] Crochemore, M., Iliopoulos, C., Kubica, M., Rytter, W., Waleń, T.: Efficient algorithms for three variants of the LPF table. J. of Discrete Algorithms. 11, 51-61 (2012)

[38] Crochemore, M., Tischler, G. Computing longest previous non-overlapping factors. Inf. Process. Lett. 111(6), 291-295, (2011)

[39] Do, C., Woods, D., Batzoglou, S.: CONTRAfold: RNA secondary structure prediction without physics-based models. ISMB (Supplement of Bioinformatics). 22(14), 90-98 (2006)

[40] Deguchi, S., Higashijima, F., Bannai, H., Inenaga, S., Takeda, M.: Parameterized suffix arrays for binary strings. In: PSC'08, pp. 84-94, Czech Republic (2008)

[41] Fischer, J.: Wee LCP. Inf. Process. Lett. 110(8-9), 317-320 (2010)

[42] Fredriksson, K., Mozgovoy, M.: Efficient parameterized string matching. Inf. Process. Lett. 100(3), 91-96 (2006)

[43] Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. In: SODA'11, pp. 362-372 (2011)

[44] Giancarlo, R., Scaturro, D., Utro, F.: Textual data compression in computational biology: A synopsis. Bioinformatics. 25(13), 1575-1586 (2009)

[45] Gramm, J., Guo, J., Niedermeier, R.: Pattern matching for arc-annotated sequences. ACM Transactions on Algorithms. 2(1), 44-65 (2006)

[46] Griffiths-Jones, S., Bateman, A., Marshall, M., Khanna, A., Eddy, S.: Rfam: An RNA family database. Nucleic Acids Research. 31(1), 439-441 (2003)

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

[48] Han, K., Byun, Y.: PSEUDOVIEWER2: Visualization of RNA pseudoknots of any type. Nucleic Acid Research. 31(13), 3432-3440 (2003)

[49] Hazay, C., Lewenstein, M., Sokol, D: Approximate parameterized matching. ACM Trans. Algorithms, 3(3), article 29 (2007)

[50] Heyne, S., Will, S., Beckstette, M., Backofen, R.: Lightweight comparison of RNAs based on exact sequence-structure matches. Bioinformatics. 25(16), 2095-2102 (2009)

[51] Hofacker, I., Schuster, P., Stadler, P.: Combinatorics of RNA secondary structures. Discrete Applied Mathematics. 88(1-3), 207-237 (1998)

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

[53] I, T., Deguchi, S., Bannai, H., Inenaga, S., Takeda, M.: Lightweight parameterized suffix array construction. In: IWOCA'09. LNCS, vol. 5874, pp. 312-323. Springer, Heidelberg (2009)

[54] I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting parameterized border arrays for a binary alphabet. In: LATA'09, pp. 422-433 (2009)

[55] I, T., Inenaga, S., Bannai, H., Takeda, M.: Verifying a parameterized border array in $O(n^{1.5})$ time. In: CPM'10, pp. 238-250 (2010)

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

[57] Iliopoulos, C.S., Mohamed, M., Mouchard, L., Perdikuri, K.G., Smyth, W.F., Tsakalidis, A.K.: String regularities with don't cares. Nordic J. Comput. 10(1), 40-51 (2003)

[58] Jiang, T., Lin, G., Ma, B., Zhang, K.: The longest common subsequence problem for arc-annotated sequences. J. of Discrete Algorithms. 2(2), 257-270 (2004)

[59] Kärkkäinen, J., Manzini, G., Puglisi, S.: Permuted longest-common-prefix array. In: CPM'09, pp. 181-192, Springer, Heidelberg (2009)

[60] Kärkkäinen, J., Navarro, G., Ukkonen, E.: Approximate string matching on Ziv-Lempel compressed text. J. of Discrete Algorithms. 1(3-4), 313-338 (2003)

[61] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: CPM'01. LNCS, vol. 2089, pp. 181-192 (2001)

[62] Kida, T.: Suffix tree based VF-coding for compressed pattern matching. In: DCC'09, p. 449 (2009)

[63] Klein, S., Shapira, D.: Compressed matching in dictionaries. Algorithms. 4(1), 61-74 (2011)

[64] Klein, S., Shapira, D.: On improving Tunstall codes. Inf. Process. Manage. 47(5), 777-785 (2011)

[65] Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In: FOCS'95, pp. 631-637 (1995)

[66] Laing, C., Schlick,T.: Computational approaches to RNA structure prediction, analysis, and design. Current Opinion in Structural Biology. 21(3), 306-318 (2011)

[67] Lee, T., Na, J., Park, K.: On-line construction of parameterized suffix trees. In: SPIRE'09, pp. 31-38, Springer, Heidelberg (2009)

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

[69] Maass, M.: Linear bidirectional on-line construction of affix trees. Algorithmica. 37, 320-334 (2003)

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

[71] Main, M., Lorentz, R. An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms. 5(3), 422-432 (1984)

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

[73] Manzini, G.: Two space saving tricks for linear time LCP array computation. In: SWAT'04. LNCS, vol. 3111, pp. 372-383 (2004)

[74] Mauri, G., Pavesi, G.: Algorithms for pattern matching and discovery in RNA secondary structure. Theor. Comput. Sci. 335(1), 29-51 (2005)

[75] Meyer, F., Kurtz, S., Backofen, R., Will, S., Beckstette, M.: Structator: Fast index-based search for RNA sequence-structure patterns. BMC Bioinformatics. 12, 214 (2011)

[76] Navarro, G., Raffinot, M.: A general practical approach to pattern matching over Ziv-Lempel compressed text. In: CPM'99, pp. 14-36, Springer, London (1999)

[77] Novikova, I., Hennelly, S., Tung, C., Sanbonmatsu, K.: Rise of the RNA machines: Exploring the structure of long non-coding RNAs. Journal of Molecular Biology, 425(19), 3731-3746 (2013)

[78] Pedersen, J., Bejerano, G., Siepel, A., Rosenbloom, K., Lindblad-Toh, K., Lander, E., Kent, J., Miller, W., Haussler, D.: Identification and classification of conserved RNA secondary structures in the human genome. PLoS Computational Biology. 2(4), 0251-0262 (2006)

[79] Penner, R., Waterman, M.: Spaces of RNA secondary structures. Advances in Mathematics. 101(1), 31-49 (1993)

[80] Puglisi, S., Turpin, A.: Space-time tradeoffs for longest-common-prefix array computation. In: ISAAC'08, pp. 124-135, Springer, Heidelberg (2008)

[81] Rabani, M., Kertesz, M., Segal, E.: Computational prediction of RNA structural motifs involved in posttranscriptional regulatory processes. PNAS. 105(39), 14885-14890 (2008)

[82] Rinn, J., Chang, H.: Genome regulation by long noncoding RNAs. Annual Review of Biochemistry. 81, 145-166 (2012)

[83] Rødland, E.: Pseudoknots in RNA secondary structures: Representation, enumeration, and prevalence. Journal of Computational Biology. 13(6), 1197-1213 (2006)

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

[85] Schieber, B., Vishkin, U. On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253-1262 (1988)

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

[87] Smyth, B.: Computing Patterns in Strings. Pearson Addison-Wesley, Essex, England (2003)

[88] Smyth, W., Wang, S.: New perspectives on the prefix array. In: SPIRE '08, pp. 133-143 (2008)

[89] Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theor. Comput. Sci. 389(1-2), 278-294 (2007)

[90] Svoboda, P., Di Cara, A.: Hairpin RNA: A secondary structure of primary importance. Cellular and Molecular Life Sciences. 63, 901-918 (2006)

[91] Tunstall, B.: Synthesis of noiseless compression codes. PhD dissertation. Georgia Institute of Technology (1967)

[92] Uemura, T., Yoshida, S., Kida, T., Asai, T., Okamoto, S.: Training parse trees for efficient VF coding. In: SPIRE'10, pp. 179-184 (2010)

[93] Wan, Y., Kertesz, M., Spitale, R., Segal, E., Chang, H.: Understanding the transcriptome through RNA structure. Nature Review. 12, 641-655 (2011)

[94] Wang, X.: Computational prediction of microRNA targets. Methods in Molecular Biology. 667, 283-295 (2010)

[95] Wang, Z., Zhang, K.: Multiple RNA structure alignment. J. Bioinformatics and Computational Biology. 3(3), 609-626 (2005)

[96] Xu, Y., Wang, L., Zhao, H., Li, J.: Exact matching of RNA secondary structure patterns. Theor. Comput. Sci. 335(1), 53-66 (2005)

[97] Yoshida, S., Kida, T.: On performance of compressed pattern matching on VF codes. In: DCC'11, p. 486 (2011)

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

[99] Zhang, K., Wang, L., Ma, B.: Computing similarity between RNA structures. In: CPM'99, pp. 281-293 (1999)

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