Compact String Indexing Structures for Parameterized Pattern Matching Problems

藤里, 法輝

Compact String Indexing Structures for Parameterized Pattern Matching Problems

Noriki Fujisato

February, 2024

Abstract

In recent years, the amount of various digital data has drastically been increasing due to the improvement of information and communication technology. However, the CPU performance has improved only slightly in recent years in contract to the rapid increase in data. Therefore, it seems infeasible to process all such data only with hardware machine powers.

On computers, most, if not all, data can be regarded as a sequence of symbols, or a string. Based on this observation, it is important and useful to develop efficient algorithms and data structures for processing large-scale string data sets.

A fundamental and important problem in string processing is the pattern matching problem, that is formalized as follows: Given a text and a pattern, report the beginning positions of all substrings of the text that match the pattern. The pattern matching problem is ubiquitous in the real-world applications including information retrieval and bioinformatics.

On the other hand, there is a requirement for pattern matching based on approximate models, not limited to the exact matching. For example, two programs in which only the names of functions and/or variables differ can be seen "identical". There can be demands to detect such approximated matching in software maintenance [3]. This problem was first introduced by Baker [3] in 1993, and was named as *parameterized pattern matching* (*PPM*).

This present thesis also deals with the PPM problems. PPM is formally defined as follows: Let Σ and Π be disjoint alphabets. Two equal length strings x and y on $\Sigma \cup \Pi$ are said to parameterized match if x can be changed to y by applying a bijection on Π. PPM is to report to report the beginning positions of all substrings of the text that parameterized match a pattern [3]. For example, consider the text $t = \texttt{uyAuBuuuAyByAyBy}$ and the pattern $p = \texttt{xAvBv}$ over the static alphabet $\Sigma = \{\texttt{A}, \texttt{B}\}$ and the parameterized alphabet $\Pi = \{\texttt{u}, \texttt{v}, \texttt{x}, \texttt{y}\}$, then the positions to output are $\{2, 8\}$. To see why, observe that for the substring $t[2:6] = \texttt{yAuBu}$ there is a bijection that maps $\texttt{y} \to \texttt{x}$ and $\texttt{u} \to \texttt{v}$ with which the substring becomes equal to p. Also, observe that for the other substring $t[8:12] = \texttt{uAyBy}$, there is a bijection that maps $\texttt{u} \to \texttt{x}$ and $\texttt{y} \to \texttt{v}$ with

which the substring becomes equal to p. We remark that the bijections have to be the identity for the characters in the static alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ in PPM, and the two aforementioned bijection satisfy this property.

Two major methods for solving PPM are known. The first method is to scan the full text t and report the position of a substring that parameterized match the pattern p each time it is found. The second method is to preprocess the input string t by constructing an indexing data structure that can quickly answer PPM queries for a given pattern p. The method to construct an indexing data structure can require more space and time to preprocess the whole text than the first method. However, the methods of constructing indexing data structures have the advantage of requiring less query time than the first method, which becomes evident in particular when we receive a number of queries over the same text.

The contribution of this thesis is two fold:

(A) Faster constructions of existing indexing data structures for PPM on strings.

(B) New indexing data structure for PPM on strings and trees.

In more details, we propose the following algorithms and data structures for PPM. Hereafter, let n be the length of the text, m be the pattern length, and occ be the number of matching positions to report.

(A-1) We propose a fast algorithm for constructing the indexing data structure called the parameterized suffix array (PSA). The best known construction algorithm for PSAs works in $O(n \log(|\Sigma| + |\Pi|))$ time and $O(n)$ words of working space (see Table 1.2). The new algorithm proposed in this thesis builds PSAs in $O(n|\Pi|)$ time and $O(n)$ words of space. When $|\Sigma|$ is constant and $|\Pi| = O(n)$, the existing algorithm requires $O(n \log n)$ time but our algorithm works in optimal $O(n)$ time.

(B-1) We propose a new indexing data structure for PPM called the right-to-left parameterized position heap (RLPPH). Our RLPPHs require $O(n)$ words of space, support PPM queries in $O(m|\Pi| + m \log(|\Pi| + |\Sigma|) + occ)$ time, and can be constructed in $O(n \log(|\Sigma| + |\Pi|))$ time and $O(n)$ words of working space in a right-to-left online manner. This contribution will also be used as a building block for (B-4).

(B-2) We propose another new indexing data structure called the parameterized directed acyclic word graph (PDAWG), which is a dual structure for the PST. Showing a data structure

having this duality had been a long standing open question. We present an offline algorithm that builds PDAWGs in $O(n \min\{\log(|\Sigma| + |\Pi|), |\Pi|\})$ time with $O(n)$ words of working space.

(B-3) We propose yet another new indexing data structure called the parameterized suffix tray (PSTr). We show that the PSTr can be constructed in $O(n \min\{\log(|\Sigma| + |\Pi|), |\Pi|\})$ time and $O(n)$ words of working space offline. We then show how the PSTr supports PPM queries in $O(m + \log(|\Pi| + |\Sigma|) + occ)$ time, which is the fastest among the existing PPM indexing structures.

(B-4) We present the first indexing structure for PPM on trees (tries), which is the PPH for tries. Let N be the number of nodes in the input tree. We develop an offline construction of the PPH for the input tree running in $O(n(|\Pi| + |\Sigma|))$ time and $O(n(|\Pi| + |\Sigma|))$ words of working space.

Acknowledgments

I would like to express my appreciation to everyone who supported my research life at Kyushu University. First of all, I am deeply grateful to Professor Shunsuke Inenaga who is my supervisor. I would also like to express my appreciation to Professor Masayuki Takeda who is my former supervisor.

I would also like to express my appreciation to Professor Eiji Takimoto who is the committee chair of my thesis. I would also like to express my appreciation to Professor Shunsuke Inenaga, Professor Hideo Bannai and Professor Kohei Hatano, who are the members of the committee of my thesis. I also thank all of those in Department of Informatics, Kyushu University, for their generous support.

I also thank specially Professor Masayuki Takeda, Professor Shunsuke Inenaga, Professor Hideo Bannai, Specially Appointed Associate Professor Dominik Köppl, Assistant Professor Yuto Nakashima and Assistant Professor Takuya Mieno. They taught how to research to me. I thank Professor Eiji Takimoto and Professor Kohei Hatano. They gave many comments to me at weekly seminar.

The results in the thesis were partially published in the Proc. of PSC'18, the Proc. of SPIRE'19, the Proc. of CIAC'19, the Proc. of CPM'20, and the Proc. of CIAC'21. I thank all editors, committees, anonymous referees, and publishers.

Last but not least, I thank my friends for their support.

Contents

Chapter 1

Introduction

1.1 Background

In recent years, the amount of various digital data has drastically been increasing due to the improvement of information and communication technology. However, the CPU performance has improved only slightly in recent years in contract to the rapid increase in data. Therefore, it seems infeasible to process all such data only with hardware machine powers.

On computers, most, if not all, data can be regarded as a sequence of symbols, or a string. Based on this observation, it is important and useful to develop efficient algorithms and data structures for processing large-scale string data sets.

A fundamental and important problem in string processing is the pattern matching problem, that is formalized as follows: Given a text and a pattern, report the beginning positions of all substrings of the text that match the pattern. The pattern matching problem is ubiquitous in the real-world applications including information retrieval and bioinformatics.

On the other hand, there is a requirement for pattern matching based on approximate models, not limited to the exact matching. For example, two programs in which only the names of functions and/or variables differ can be seen "identical". There can be demands to detect such approximated matching in software maintenance [3]. This problem was first introduced by Baker [3] in 1993, and was named as *parameterized pattern matching* (*PPM*).

This present thesis also deals with the PPM problems. PPM is formally defined as follows: Let Σ and Π be disjoint alphabets. Two equal length strings x and y on $\Sigma \cup \Pi$ are said to parameterized match if x can be changed to y by applying a bijection on Π. PPM is to report to report the beginning positions of all substrings of the text that parameterized match a pattern [3]. For example, consider the text $t = \texttt{uyAuBuuuAyByAyBy}$ and the pattern $p = \texttt{xAvBv}$ over the

static alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ and the parameterized alphabet $\Pi = \{\mathtt{u}, \mathtt{v}, \mathtt{x}, \mathtt{y}\}$, then the positions to output are $\{2, 8\}$. To see why, observe that for the substring $t[2:6] = \mathtt{yAuBu}$ there is a bijection that maps $\mathtt{y} \rightarrow \mathtt{x}$ and $\mathtt{u} \rightarrow \mathtt{v}$ with which the substring becomes equal to p. Also, observe that for the other substring $t[8:12] = \mathtt{uAyBy}$, there is a bijection that maps $\mathtt{u} \rightarrow \mathtt{x}$ and $\mathtt{y} \rightarrow \mathtt{v}$ with which the substring becomes equal to p. We remark that the bijections have to be the identity for the characters in the static alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ in PPM, and the two aforementioned bijection satisfy this property.

Two major methods for solving PPM are known. The first method is to scan the full text t and report the position of a substring that parameterized match the pattern p each time it is found. The second method is to preprocess the input string t by constructing an indexing data structure that can quickly answer PPM queries for a given pattern P. The method to construct an indexing data structure can require more space and time to preprocess the whole text than the first method. However, the methods of constructing indexing data structures have the advantage of requiring less query time than the first method, which becomes evident in particular when we receive a number of queries over the same text.

This current thesis focuses on this second type of the PPM problem, which is formalized as follows:

Definition 1 (Indexing version of the PPM problem)**.**
Preprocess: Text t of length n over $\Pi \cup \Sigma$.
Query input: Pattern P of length m over $\Pi \cup \Sigma$.
Query output: All text positions i such that $t[i : i + m - 1]$ parameterized match p.

There exist various indexing data structures for PPM including *parameterized suffix trees* (*PSTs*) [3], *parameterized suffix arrays* (*PSAs*) [15], *parameterized position heaps* (*PPHs*) [16], and *parameterized Burrows-Wheeler Transforms* (*PBWTs*) [23, 30]. Hereafter, let n be the length of the text, m be the pattern length, and occ be the number of pattern occurrences. Below we review the complexities of these existing data structure for PPM.

1.2 Existing Indexing Structures for PPM

1.2.1 Parameterized Suffix Trees (PSTs)

The common tool for PPM is the *previous encoding* [3], that transforms a given string t of length n into another sequence s of length n such that each $s[i]$ represents the distance to the

most recent occurrence of $t[i] \in \Pi$ in $t[1..i-1]$ (a more formal definition will be given in Chapter 2).

Baker [3] proposed the parameterized suffix trees (PSTs) which are analogue to the suffix trees [45] for exact pattern matching. The PST of a text t is basically the rooted tree that represents the previous encodings of all the suffixes of t, where the edge labels are non-empty substrings of these previous encodings and the labels of the out-going edges of the same node begin with distinct symbols. By representing each edge label with a corresponding interval $[i..j]$ of t, one can store the PST of t in $O(n)$ words of space. PSTs support PPM queries in $O(m \log(|\Pi| + |\Sigma|) + occ)$ time [3]. As for constructions, Baker [3] showed that PSTs can be constructed in $O(n(|\Pi| + \log|\Sigma|))$ time and $O(n)$ words of working space. Kosaraju [33] claimed an online algorithm for constructing PSTs but later it was pointed out that his method contained errors [44]. Cole and Hariharan [11] proposed an online algorithm for constructing PSTs in $O(n \log(|\Pi| + |\Sigma|))$ time and $O(n)$ words of working space, that processes the given text from right to left. Shibuya [44] proposed a different version of an online algorithm for constructing PSTs in $O(n \log(|\Pi| + |\Sigma|))$ time and $O(n)$ words of working space, that processes the given test from left to right. Lee et al. [35] proposed an algorithm for constructing PSTs in *randomized* $O(n)$ time and $O(n)$ words of working space.

In this thesis, we present an offline algorithm for constructing the PST in $O(n|\Pi|)$ time and $O(n)$ words of working space. This has a merit over the existing methods when Σ is an integer alphabet of polynomial size in n, and Π is a constant-size alphabet. This PST construction is via our PSA construction algorithm running in $O(n|\Pi|)$ time and $O(n)$ words of working space, which will be discussed in the next subsection.

See Table 1.1 for a summary of comparisons of time and space complexities of PST constructions.

Algorithm	Construction Time	Working Space	Online or Offline
Baker [3]	$O(n(\|\Pi\| + \log\|\Sigma\|))$	$O(n)$ words	Offline
Cole and Hariharan [11]	$O(n \log(\|\Pi\| + \|\Sigma\|))$	$O(n)$ words	Online
Shibuya [44]	$O(n \log(\|\Pi\| + \|\Sigma\|))$	$O(n)$ words	Online
This Paper (via PSA)	$O(n\|\Pi\|)$	$O(n)$ words	Offline

Table 1.1: Algorithms for constructing PSTs.

1.2.2 Parameterized Suffix Arrays (PSAs)

The *parameterized suffix array* (*PSA*) of a text t stores the sorted text-positions in the lexicographical order of the previous encoded suffixes. In other words, the PSA for t is an array of length n that is equivalent to the lexicographically sorted list of the leaves of the PST for t.

PSAs were first introduced by Deguchi et al. [15], as an analogue to suffix arrays [36] for exact pattern matching. The space complexity of the PSA is $O(n)$ words, and supports PPM queries in $O(m + \log n + occ)$ time [15].

It is obvious that one can construct the PSA for t in $O(n \log(|\Pi|+|\Sigma|))$ time and $O(n)$ words of working space via constructing the edge-sorted PST for t [11]. The challenge is to build PSAs "directly", without the need to first build PSTs as an intermediate structure. I et al. [25] proposed a practically efficient algorithm that directly constructs PSAs, but the worst-case time was still in $O(n^2)$. Beal and Adjeroh [5] proposed a PSA construction algorithm based on arithmetic coding that runs in $O(n)$ time on average. However, the proved worst-case upper bound is $O(n^2(\log(n-\log^{1+\epsilon} n)/\log^{1+\epsilon} n))$ for $0 < \epsilon < 1$, which is $O(n^2/\log^{\epsilon} n) \subseteq o(n^2)$ (Theorem 26 and Corollary 27 of [5]). However, no previous methods have achieved *strongly sub-quadratic* $O(n^{2-\epsilon})$-time direct constructions for PSAs.

In this thesis, we propose an offline algorithm for constructing PSAs in $O(n|\Pi|)$ time and $O(n)$ words of space in this paper. This is the first direct PSA construction algorithm that runs in strongly sub-quadratic $O(n^{2-\epsilon})$ time when $|\Pi| = O(n^{1-\epsilon})$, and in particular, our method runs in optimal $O(n)$ time for $|\Pi| = O(1)$.

See Table 1.2 for a summary of comparisons of time and space complexities of PSA constructions.

Algorithm	Construction Time	Working Space	Online or Offline
Deguchi et al. [15] (when $\|\Pi\| = 2, \|\Sigma\| = 0$)	$O(n)$	$O(n)$ words	Offline
Cole and Hariharan [11] (via PST)	$O(n \log(\|\Pi\| + \|\Sigma\|))$	$O(n)$ words	Online
Beal and Adjeroh [5]	$O(n^2 \frac{\log(n-\log^{1+\epsilon} n)}{\log^{1+\epsilon} n})$	$O(n)$ words	Offline
This Paper	$O(n\|\Pi\|)$	$O(n)$ words	Offline

Table 1.2: Algorithms for constructing PSAs.

1.2.3 Parameterized Position Heaps (PPHs)

The *parameterized position heap* (*PPH*) of a text t is a rooted tree with $n+1$ nodes where each edge is labeled by a symbol and there is a one-to-one correspondence between the text positions and the nodes except for the root.

Diptarama et al. [16] were the first who proposed a version of PPHs called the left-to-right parameterized position heap (*LRPPH*), which is analogue to the left-to-right position heap [34] for exact matching, which can be constructed in a left-to-right online manner. They showed that LRPPHs take $O(n)$ words of space and supports PPM queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time [16].

We propose an alternative version of PPHs, called the right-to-left parameterized position heap (*RLPPH*) which is analogue to the right-to-left position heap [17] for exact matching, which can be built in a right-to-left online manner. The RLPPH requires $O(n)$ words of space and supports PPM queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time. This right-to-left update property will then be a key to an extended problem of indexing a labeled tree, which will later be discussed in details.

1.2.4 Parameterized Burrows-Wheeler Transforms (PBWTs)

Ganguly et al. [23] proposed an indexing data structure for PPM called the *parameterized Burrows-Wheeler Transform* (*PBWT*), which is analogue to the Burrows-Wheeler Transform (BWT) for exact matching [9]. They showed that the PBWTs take $n \log(|\Sigma| + |\Pi|) + O(n)$ bits of space, and supports queries in $O(m \log(|\Sigma| + |\Pi|) + occ)$ time [23]. Kim and Cho [30] proposed the simplified PBWT that takes $2n \log(|\Sigma| + |\Pi|) + 2n + o(n)$ bits of space.

It is known offline algorithm for constructing PBWT in $O(n \min\{\log(|\Pi| + |\Sigma|), |\Pi|\})$ time and $O(n)$ words of space [11, 21]. Hashimoto et al. [24] presented an online algorithm for directly constructing the PBWT in $O(n|\Pi| \log n / \log\log n)$time using $O(n \log(|\Sigma| + |\Pi|))$ bits of space. Iseri et al. [26] gave an online algorithm for directly constructing the PBWT in $O(n\log |\Pi| \log n / \log\log n)$time with $O(n \log(|\Sigma| + |\Pi|))$ bits of space.

See Table 1.3 for a summary of comparisons of time and space complexities of PBWT constructions.

Algorithm	Construction Time	Working Space	Online or Offline
via PST [11]	$O(n \log(\lvert\Pi\rvert + \lvert\Sigma\rvert))$	$O(n)$ words	Offline
via PSA [this work]	$O(n\lvert\Pi\rvert)$	$O(n)$ words	Offline
Hashimoto et al. [24]	$O(n\frac{\lvert\Pi\rvert \log n}{\log \log n})$	$O(n \log(\lvert\Sigma\rvert + \lvert\Pi\rvert))$ bits	Online
Iseri et al. [26]	$O(n\frac{\log \lvert\Pi\rvert \log n}{\log \log n})$	$O(n \log(\lvert\Sigma\rvert + \lvert\Pi\rvert))$ bits	Online

Table 1.3: Algorithms for constructing PBWTs.

1.3 Our Contributions

The contribution of this thesis is two fold:

(A) Faster constructions of existing indexing data structures for PPM on strings.

(B) New indexing data structure for PPM on strings and trees.

In what follows, we describe our contributions in more details.

See Tables 1.4 and 1.5 for comparisons of space requirements, query times, online/offline construction time and working space for the existing and proposed indexing structures for PPM.

1.3.1 (A-1) Faster Parameterized Suffix Array Construction

As was described above, it was known that PSAs can be built in $O(n \log(|\Pi| + |\Sigma|))$ time and $O(n)$ words of working space, via construction for PSTs [11].

We present an offline algorithm for constructing PSAs without constructing PSTs, in $O(n|\Pi|)$ time and $O(n)$ words of working space. In addition, we show how to build the *parameterized longest common prefix* (*PLCP*) from the SA for the text t in $O(n|\Pi|)$ time. Further, we can build the PST for t from the PSA and PLCP for t in linear time. These algorithms we will propose are one of the fastest known algorithms for building PSAs and PSTs.

1.3.2 (B-1) Right-to-Left Parameterized Position Heaps (RLPPHs)

Kucherov [34] proposed an indexing data structure called the left-to-right position heap (LRPH), for the exact pattern matching problem. The LRPH is constructed online by reading the text from left to right. Kucherov [34] showed that the query time of the LRPH is $O(m \log |\Sigma| + occ)$ time, and proposed an online algorithm for constructing the RLPH in $O(n \log |\Sigma|)$ time and $O(n)$ words of working space.

Ehrenfeucht et al. [17] proposed an indexing data structure called the right to left position heap (RLPH), for the exact pattern matching problem. The RLPH for text t is constructed online by reading the text t from right to left. Ehrenfeucht et al. [17] showed that the query time of the RLPH is $O(m \log |\Sigma| + occ)$ time, and proposed an online algorithm for constructing the RLPH in $O(n \log |\Sigma|)$ time and $O(n)$ words of working space.

The LRPPH of Diptarama et al. [16] is the PPM version of the LRPH of Kucherov [34]. LRPPHs require $O(n)$ words of space, support PPM queries in $O(m|\Pi|+m \log(|\Pi|+|\Sigma|)+occ)$ time, and can be built in $O(n \log(|\Pi| + |\Sigma|))$ time and $O(n)$ words of working space [16].

In contrast, our RLPPH is the PPM version of the RLPH of Ehrenfeucht et al. [17]. RLPPHs achieve the same time and space complexities for queries and constructions for the aforementioned LRPPHs. The advantage of our RLPPHs over the LRPPHs counterpart is that our right-to-left online nature permits us to extend the construction scheme from string inputs to tree inputs (see our contribution (B-4) at Chapter 1.3.5 below).

1.3.3 (B-2) Parameterized Directed Acyclic Word Graphs (PDAWGs)

The *directed acyclic word graph* (*DAWG*) of a text t, proposed by Blumer et al. [6], is a (partial) DFA that accepts all suffixes of t and can thus be used as text indexing structure. The DAWGs and suffix trees are dual structures, in the combinatorial sense that the nodes of the DAWG for t represent the left-maximal substrings of t, while the nodes of the suffix tree for t represent the right-maximal substrings of t. In the data structure point of view, the *Weiner links* [45] of the suffix tree for t are equivalent to the DAWG for the reversal of t.

It had been a long standing open question whether there exist an $O(n)$-space PPM version of DAWGs, and quite recently, *parameterized directed acyclic word graph* (*PDAWGs*) have been proposed [38]. PDAWGs take $O(n)$ words of space, support queries in $O(m \log(|\Pi|+|\Sigma|)+occ)$ time, can be build in $O(n|\Pi| \log(|\Pi| + |\Sigma|))$ time and $O(n)$ words of space in a left-to-right online manner [38].

In this paper, we will reveal the PPM version of the duality of the two indexing structures, namely, the PST for t augmented with parameterized reversed suffix links is equivalent to the PDAWG for the reversal of t. Also, we will show an offline algorithm that constructs PDAWGs in $O(n \min\{\log(|\Pi| + |\Sigma|), |\Pi|\})$ time and $O(n)$ words of working space.

1.3.4 (B-3) Parametrized Suffix Trays (PSTrs)

The suffix tray [12] for a text t is a "hybrid" indexing data structure for exact pattern matching that is built on the suffix tree [45] and the suffix array [36] for t. The suffix tray takes $O(n)$ words of space, supports queries in $O(m + \log|\Sigma| + occ)$ time, and can be build in $O(n)$time and $O(n)$ words of working space [12]. The query time of the suffix tray is faster than any other linear indexing data structure which is constructed in $O(n)$ time.

In this paper, we propose the PPM version of the suffix tray, called the *parameterized suffix trays* (*PSTrs*). Our PSTr requires $O(n)$ words of space, supports queries in $O(m + \log(|\Pi| + |\Sigma|) + occ)$ time, and can be build in $O(n \min\{\log(|\Pi| + |\Sigma|), |\Pi|\})$ time and $O(n)$ words of working space, respectively. It is notable that the query time of the PSTr is faster than any other other known indexing data structure for PPM.

Index	Index Space	Query Time
PST [3]	$O(n)$ words [3]	$O(m \log(\|\Sigma\| + \|\Pi\|) + occ)$ [3]
PSA [15]	$O(n)$ words [15]	$O(m + \log n + occ)$ [15]
PDAWG [38]	$O(n)$ words [38]	$O(m \log(\|\Sigma\| + \|\Pi\|) + occ)$ [38]
LRPPH [16]	$O(n)$ words [16]	$O(m \log(\|\Sigma\| + \|\Pi\|) + m\|\Pi\| + occ)$ [16]
RLPPH [this paper]	$O(n)$ words [this paper]	$O(m \log(\|\Sigma\| + \|\Pi\|) + m\|\Pi\| + occ)$ [this paper]
PSTr [this paper]	$O(n)$ words [this paper]	$O(m + \log(\|\Sigma\| + \|\Pi\|) + occ)$ [this paper]
PBWT [23]	$O(n \log(\|\Sigma\| + \|\Pi\|))$ bits [23]	$O(m \log(\|\Sigma\| + \|\Pi\|) + occ)$ [23]

Table 1.4: Query times and space requirements of indexing data structures for PPM.

1.3.5 (B-4) Parameterized Position Heaps for Trees

A trie is a rooted tree where each edge is labeled by a single character and the out-going edges of the same node are labeled by distinct characters. Tries are natural extensions to strings, and are a space-efficient representation of multiple strings. Thus, the indexing version of the pattern matching problem on tries (trees) is important, and suffix trees for trees [8] and position heaps for trees [40] have already been proposed, both of which take $O(N)$ words of space for a given tree with N nodes.

For a given tree with N nodes, the suffix tree of the tree can be built in $O(N)$ time and $O(N)$ words of working space [43], and the position heap of the tree can also be build in $O(N)$ time and $O(N)$ words of working space [40] from the suffix tree for the tree.

Index	Online Construction Time, Working Space	Offline Construction Time, Working Space
PST [3]	$O(n \log(\|\Pi\| + \|\Sigma\|))$, $O(n)$ words [11]	$O(n\|\Pi\|)$, $O(n)$ words [this paper]
PSA [15]	$O(n \log(\|\Pi\| + \|\Sigma\|))$, $O(n)$ words [11]	$O(n\|\Pi\|)$, $O(n)$ words[this paper]
PDAWG [38]	$O(n\|\Pi\| \log(\|\Pi\|) + n \log \|\Sigma\|)$, $O(n)$ words [38]	$O(n \min\{\log(\|\Pi\| + \|\Sigma\|), \|\Pi\|\})$, $O(n)$ words[this paper]
LRPPH [16]	$O(n \log(\|\Pi\| + \|\Sigma\|)$, $O(n)$ words [16]	N/A
RLPPH [this paper]	$O(n \log(\|\Pi\| + \|\Sigma\|)$, $O(n)$ words [this paper]	N/A
PSTr [this paper]	N/A	$O(n \min\{\log(\|\Pi\| + \|\Sigma\|), \|\Pi\|\})$, $O(n)$ words [this paper]
PBWT [23]	$O(n\frac{\log \|\Pi\| \log n}{\log \log n})$, $O(n \log(\|\Sigma\| + \|\Pi\|))$ bits [23]	$O(n \min\{\log(\|\Pi\| + \|\Sigma\|), \|\Pi\|\})$, $O(n)$ words [11, 21]

Table 1.5: Construction times and working space of indexing data structures for PPM.

In this thesis, we present the first indexing data structure for the PPM problem on tree inputs. The proposed indexing structure, the PPH for trees, is an analogue to the PPH for strings (B-1) and the position heap for trees [40]. Our PPH for trees require $O(N)$ words of space, supports PPM queries on trees in $O(m \log(|\Pi| + |\Sigma|) + m|\Pi| + occ)$ time, and can be constructed in $O(N(|\Sigma| + |\Pi|))$ time and $O(N(|\Sigma| + |\Pi|))$ words of working space.

1.4 Organizations

The rest of this thesis is organized as follows: In Chapter 2 we give basic definitions ad notations. Chapter 3 deals with RLPPHs (B-1), and then we discuss PPHs for trees (B-4) in Chapter 4. Chapter 5 is devoted to PSAs (A-1), and Chapter 6 deals with PDAWGs (B-2). We present PSTrs in Chapter 7, and conclude in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we give notations to be used in this thesis.

2.1 Strings

Let Σ and Π be disjoint sets called a *static alphabet* and a *parameterized alphabet*, respectively. An element of Σ is called an *s-character*, and that of Π is called a *p-character*. In the sequel, both an s-character and a p-character are sometimes simply called a *character*. An element of Σ^* is called a *string*, and an element of $(\Sigma \cup \Pi)^*$ is called a *p-string*. The length of a (p-)string w is the number of characters contained in w. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a (p-)string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The set of prefixes, substrings, and suffixes of a (p-)string S is denoted by $\mathsf{Prefix}(w)$, $\mathsf{Substr}(w)$, and $\mathsf{Suffix}(w)$, respectively. The i-th character of a (p-)string w is denoted by $w[i]$ for $1 \leq i \leq |w|$. The factor of w that begins at position i and ends at position j is $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, we abbreviate $w[1 : i]$ to $w[: i]$ and $w[i : |w|]$ to $w[i :]$ for $1 \leq i \leq |w|$. The empty string is denoted by ε, that is $|\varepsilon| = 0$. Moreover, let $w[i : j] = \varepsilon$ if $i > j$. The *reverse* w^R of $w \in A^*$ is inductively defined by $\varepsilon^R = \varepsilon$ and $xa^R = ax^R$ for $a \in A$ and $x \in A^*$.

2.2 Parameterized Pattern matching

Two p-strings x and y of length k each are said to *parameterized match* (*p-match*) iff there is a bijection f on $\Sigma \cup \Pi$ such that $f(a) = a$ for any $a \in \Sigma$ and $f(x[i]) = y[i]$ for any $1 \leq i \leq k$ [3]. For instance, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, and consider two p-strings $s_1 = \mathtt{axbyyzaxz}$

and $s_2 = \texttt{azbxxyazy}$. These two strings p-match, since s_1 can be transformed to s_2 by applying a renaming bijection f such that $f(\texttt{a}) = \texttt{a}$, $f(\texttt{b}) = \texttt{b}$, $f(\texttt{x}) = \texttt{z}$, $f(\texttt{y}) = \texttt{x}$, and $f(\texttt{z}) = \texttt{y}$ to the characters in s_1.

Definition 1 (Previous encoding [3]). *For a p-string w, the previous encoding $\mathsf{prev}(w)$ is a string of length $|w|$ such that for each $1 \leq i \leq |w|$,*

$$\mathsf{prev}(w)[i] = \begin{cases} w[i] & \text{if } w[i] \in \Sigma, \\ 0 & \text{if } w[i] \in \Pi \text{ and } w[j] \neq w[i] \text{ for any } 1 \leq j < i, \\ i - j & \text{otherwise, } w[i] = w[j] \text{ and } w[i] \neq w[k] \text{ for any } j < k < i. \end{cases}$$

It is known that two p-strings s_1 and s_2 p-match iff $\mathsf{prev}(s_1) = \mathsf{prev}(s_2)$. The representative of $[x]$ is the lexicographically smallest p-string in $[x]$, which is denoted by $\mathsf{spe}(x)$ [22]. It is known that two p-strings s_1 and s_2 p-match iff $\mathsf{spe}(s_1) = \mathsf{spe}(s_2)$.

The *parameterized pattern matching* problem is to find all positions i in t such that $t[i : i + m - 1] \approx P$.

2.3 Indaxing Data Structures

Definition 2 (Trie). *Let Σ be a alphabet. Then, Trie is defined as follow $Tr = (V, E)$:*
1. each edge is labeled with a character in Σ ;
2. any two out-going edges of any node are labeled with distinct characters;
3.each node $v \in V$ is associated with a string that is obtained by concatenating the edge labels in the path from root to the v;

Definition 3 (Compact Trie). *Let Σ be a alphabet. Then, Trie is defined as follow $CT = (V, E)$:*
1. each edge is labeled with a string in Σ^* ;
2. any two out-going edges of any node are labeled with strings whose first letter is different;
3.each node $v \in V$ is associated with a string that is obtained by concatenating the edge labels in the path from root to the v;

2.4 Computation Model

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\lceil \log_2 n \rceil$, and hence, standard operations on values representing lengths and positions

of strings can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

Chapter 3

Right-to-Left Parameterized Position Heaps

3.1 Introduction

Ehrenfeucht et al. [17] proposed a text indexing structure called *position heaps*. Ehrenfeucht et al.'s position heap is constructed in a *right-to-left* online manner, where a new node is incrementally inserted to the current position heap for each decreasing position $i = n, \ldots, 1$ in the input string t of length n. In other words, Ehrenfeucht et al.'s position heap is defined over a sequence $\langle \varepsilon, t[n\ :], \ldots, t[1\ :] \rangle$ of the suffixes of t in increasing order of their length, where ε is the empty string of length 0. Kucherov [34] proposed another variant of position heaps. Kucherov's position heap is constructed in a *left-to-right* online manner, where a new node is incrementally inserted to the current position heap for each increasing $i = 1, \ldots, n$. In other words, Kucherov's position heap is defined over a sequence $\langle t[1\ :], \ldots, t[n\ :], \varepsilon \rangle$ of the suffixes of t in decreasing order of their length. We will call Ehrenfeucht et al.'s position heap as the RL position heap, and Kucherov's position heap as the LR position heap. Both of the RL and LR position heaps for a text string t of length n require $O(n)$ space and can be constructed in $O(n \log |\Sigma|)$. By augmenting the RL and LR position heaps of t with auxiliary links called maximal reach pointers, pattern matching queries can be answered in $O(m \log |\Sigma| + occ)$ time, where m is the length of a query pattern P and occ is the number of occurrences of P in t.

Nakashima et al. [39] proposed position heaps for a set of strings that is given as a reversed trie, and proposed an algorithm that constructs the position heap of a given trie in $O(|\Sigma|N)$ time and space, where N is the size of the input trie. Later, the same authors showed how to construct the position heap of a trie in $O(N)$ time and space, for integer alphabets of size polynomialy

bounded in N [40].

Of various algorithms and indexing structures for the parameterized pattern matching (see [37] for a survey), we focus on Diptarama et al.'s *parameterized position heaps* [16]. Diptarama et al.'s *parameterized position heaps* are based on Kucherov's LR position heaps, which are constructed in a *left-to-right* online manner. Let us call their structure the *LR p-position heaps*. Diptarama et al. showed how to construct the LR p-position heap for a given text of length n in $O(n \log(|\Sigma| + |\Pi|))$ time with $O(n)$ space. They also showed that the LR p-position heap augmented with maximal reach pointers can support parameterized pattern matching queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time, where occ is the number of occurrences to report.

In this paper, we propose *RL p-position heaps* which are constructed in a *right-to-left* online manner. We show how to construct our RL position heap for a given text string t of length n in $O(n \log(|\Sigma| + |\Pi|))$ time with $O(n)$ space. Our construction algorithm is based on Ehrenfeucht et al.'s construction algorithm for RL position heaps [17], and Weiner's suffix tree construction algorithm [45]. Namely, we use reversed suffix links defined for the nodes of RL p-position heaps. The key to our algorithm is how to label the reversed suffix links, which will be clarified in Definition 11. Using our RL p-position heap augmented with maximal reach pointers, one can perform parameterized pattern matching queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.

3.2 Parameterized position heaps

Let $\mathcal{S} = \langle S_1, \ldots, S_k \rangle$ be a sequence of strings such that for any $1 < i \leq k$, $S_i \notin \mathsf{Prefix}(S_j)$ for any $1 \leq j < i$. For convenience, we assume that $S_1 = \varepsilon$.

Definition 4 (Sequence hash trees [10])**.** *The* sequence hash tree *of a sequence* $\mathcal{S} = \langle S_1, \ldots, S_k \rangle$ *of strings, denoted* $\mathsf{SHT}(\mathcal{S})$*, is a trie structure that is recursively defined as follows: Let* $\mathsf{SHT}(\mathcal{S})^i = (V_i, E_i)$*. Then*

$$\mathsf{SHT}(S)^i = \begin{cases} (\{\varepsilon\}, \emptyset) & \text{if } i = 1, \\ (V_{i-1} \cup \{p_i\}, E_{i-1} \cup \{(q_i, c, p_i)\}) & \text{if } 1 \leq i \leq k, \end{cases}$$

where q_i *is the longest prefix of* S_i *which satisfies* $q_i \in V_{i-1}$*,* $c = S_i[|q_i| + 1]$*, and* p_i *is the shortest prefix of* S_i *which satisfies* $p_i \notin V_{i-1}$*.*

Note that since we have assumed that each $S_i \in \mathcal{S}$ is not a prefix of S_j for any $1 \leq j < i$, the new node p_i and new edge (q_i, c, p_i) always exist for each $1 \leq i \leq k$. Clearly $\mathsf{SHT}(\mathcal{S})$

$\mathsf{prev}(t[17:])$	0
$\mathsf{prev}(t[16:])$	00
$\mathsf{prev}(t[15:])$	a00
$\mathsf{prev}(t[14:])$	0a03
$\mathsf{prev}(t[13:])$	00a33
$\mathsf{prev}(t[12:])$	000a33
$\mathsf{prev}(t[11:])$	0100a33
$\mathsf{prev}(t[10:])$	00104a33
$\mathsf{prev}(t[9:])$	010104a33
$\mathsf{prev}(t[8:])$	0013104a33
$\mathsf{prev}(t[7:])$	01013104a33
$\mathsf{prev}(t[6:])$	001313104a33
$\mathsf{prev}(t[5:])$	0101313104a33
$\mathsf{prev}(t[4:])$	00131313104a33
$\mathsf{prev}(t[3:])$	002131313104a33
$\mathsf{prev}(t[2:])$	0022131313104a33
$\mathsf{prev}(t[1:])$	a0022131313104a33

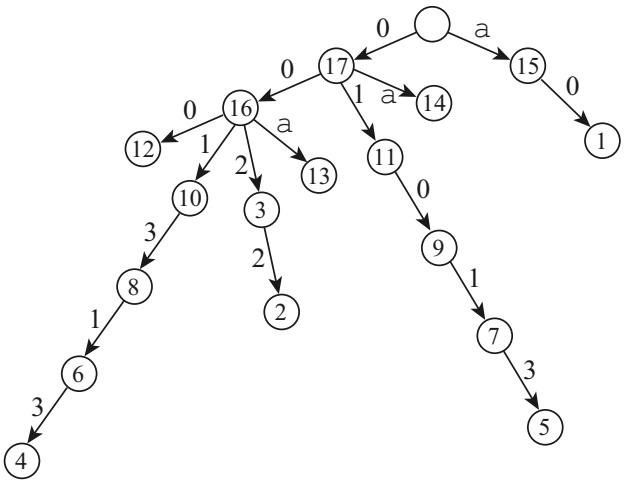

Figure 3.1: To the left is the list of $\mathsf{prev}(t[i:])$ for p-string $t = \mathtt{axyxyyxxyyxxzyazy}$ of length 17, where $\Sigma = \{\mathtt{a}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. To the right is an illustration for $\mathsf{PPH}(t)$. The underlined prefix of each $\mathsf{prev}(t[i:])$ in the left list denotes the longest prefix of $\mathsf{prev}(t[i:])$ that was inserted to $\mathsf{PPH}(t[i+1:])$ and hence, the node with id i represents this underlined prefix of $\mathsf{prev}(t[i:])$.

contains k nodes (including the root).

In what follows, we will define our indexing data structure for a text p-string t of length n. Let $\mathcal{P}_t = \langle \varepsilon, \mathsf{prev}(t[n:]), \ldots, \mathsf{prev}(t[1:]) \rangle$ be the sequence of previous encoded suffixes of t arranged in *increasing* order of their length. It is clear that $\mathsf{prev}(t[i:]) \notin \mathsf{Prefix}(\mathsf{prev}(t[j:]))$ for any $1 \leq j < i$ and $\mathsf{prev}(t[i:]) \notin \mathsf{Prefix}(\varepsilon)$ for any $1 \leq i \leq n$. Hence we can naturally define the sequence hash tree for $\mathcal{P}_t$, and we obtain our data structure:

Definition 5 (Parameterized positions heaps)**.** *The* parameterized position heap *(*p-position heap*) for a p-string t, denoted $\mathsf{PPH}(t)$, is the sequence hash tree of $\mathcal{P}_t$ i.e., $\mathsf{PPH}(t) = \mathsf{SHT}(\mathcal{P}_t)$.*

See Figure 3.1 for an example of our p-position heap.

Note that we can obtain $\mathcal{P}_{t[i-1:]}$ by adding $\mathsf{prev}(t[i-1:])$ at the beginning of $\mathcal{P}_{t[i:]}$. This also means that $\mathsf{PPH}(t[i:]) = \mathsf{SHT}(\mathcal{P}_{t[i:]})$ for each $1 \leq i \leq n$. Hence, we can construct $\mathsf{PPH}(t)$ by processing the input string t from right to left. We remark that we can easily compute $\mathsf{prev}(t[i-1:])$ from $\mathsf{prev}(t[i:])$ in a total of $O(n \log |\Pi|)$ time for all $2 \leq i \leq n$ using $O(\min\{|\Pi|, n\})$ extra space, e.g., by maintaining a balanced search tree that stores the distinct p-characters that have occurred in $t[i:]$ and records the leftmost occurrences of these p-character in the nodes.

Diptarama et al. [16] proposed another version of parameterized position heap for a sequence of previous encoded suffixes of the input p-string t arranged in *decreasing* order of their length. Since their algorithm processes t from left to right, we sometimes call their structure as a *left-to-right p-position heap* (*LR p-position heap*), while we call our $\mathsf{PPH}(t)$ as a *right-to-left p-position heap* (*RL p-position heap*) since our construction algorithm processes t from right to left.

For any p-string $P \in (\Sigma \cup [0 : n-1])^+$, we say that P is *represented* by $\mathsf{PPH}(t)$ iff $\mathsf{PPH}(t)$ has a path which starts from the root and spells out P.

Lemma 1. *For any string t of length n, $\mathsf{PPH}(t)$ consists of exactly $n+1$ nodes. Also, there is a one-to-one correspondence between the positions $1, \ldots, n$ in t and the non-root nodes of $\mathsf{PPH}(t)$.*

Proof. Initially, $\mathsf{PPH}(\varepsilon)$ consists only of the root that represents ε. For each $1 \leq i \leq n$, since $|\mathsf{prev}(t[i:])| = n - i + 1 > n - j + 1 = |\mathsf{prev}(t[j:])|$ for any $1 \leq i < j \leq n$, it is clear that there is a prefix of $\mathsf{prev}(t[i:])$ that is not represented by $\mathsf{PPH}(t[i+1:])$. Therefore, when we construct $\mathsf{PPH}(t[i:])$ from $\mathsf{PPH}(t[i+1:])$, then exactly one node is inserted, which corresponds to position i. □

Let V be the set nodes of $\mathsf{PPH}(t)$. Based on Lemma 9, we define a bijection $\mathsf{id} : V \to [0 : n]$ such that $\mathsf{id}(r) = 0$ for the root r and $\mathsf{id}(v) = i$ iff v was the node that was inserted when constructing $\mathsf{PPH}(t[i:])$ from $\mathsf{PPH}(t[i+1:])$.

Unlike our RL p-position heap, Diptarama et al.'s LR p-position heap can have *double nodes* to which two positions of the text p-string are associated.

We remark that the pattern matching algorithm of Diptarama et al. [16] can be applied to our RL p-position heap $\mathsf{PPH}(t)$ for a text p-string t, and this way one can solve the parameterized pattern matching problem in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time, where occ is the number of positions in text t such that the pattern p-string P of length m and the corresponding substring $t[i : i+m-1]$ p-match. We note that since our RL p-position heap does not have double nodes, the pattern matching algorithm can be somewhat simplified.

The following lemma is an analogue to Lemma 6 of [16] for Diptarama et al.'s LR p-position heap.

Lemma 2. *For any $1 \leq i \leq j \leq n$ if $\mathsf{prev}(t[i:j])$ is represented by $\mathsf{PPH}(t)$, then for any substring X of $t[i:j]$, $\mathsf{prev}(X)$ is represented by $\mathsf{PPH}(t)$.*

Proof. The lemma can be shown in a similar way to Lemma 6 of [16]. For the sake of completeness, we provide a full proof below.

First, we show that for any proper prefix $t[i : i + k]$ of $t[i : j]$ with $0 \leq k < j - i$, $\mathsf{prev}(t[i : i + k])$ is represented by $\mathsf{PPH}(t)$. It follows from the definition of previous encoding that $\mathsf{prev}(t[i : i + k]) = \mathsf{prev}(t[i : j])[1 : k + 1]$, and hence $\mathsf{prev}(t[i : i + k])$ is a prefix of $\mathsf{prev}(t[i : j])$. Since $\mathsf{prev}(t[i : j])$ is represented by $\mathsf{PPH}(t)$ and $i \leq i + k < j$, $\mathsf{prev}(t[i : i + k])$ is also represented by $\mathsf{PPH}(t)$.

Now it suffices for us to show that for any proper suffix $t[i + h : j]$ of $t[i : j]$ with $0 < h \leq j - i$, $\mathsf{prev}(t[i + h : j])$ is represented by $\mathsf{PPH}(t)$, since then we can inductively apply the above discussion for the prefixes. By the above discussions for the prefixes of $t[i : j]$, there exist positions $i = b_{j-i} < \cdots < b_0 \leq n$ in t such that $\mathsf{prev}(t[i : i + k]) = \mathsf{prev}(t[b_k : b_k + k])$ for $0 \leq k \leq j - i$. By the definition of $\mathsf{PPH}(t)$, the root has an out-going edge labeled by $\mathsf{prev}(t[b_1 + 1 : b_1 + 1])$, and this is the base case for our induction. Since $\mathsf{prev}(t[i : i + k]) = \mathsf{prev}(t[b_k : b_k + k])$, we have $\mathsf{prev}(t[i + 1 : i + k]) = \mathsf{prev}(t[b_k + 1 : b_k + k])$. Now since $\mathsf{prev}(t[b_{k+1}{+}1 : b_{k+1}{+}k{+}1]) = \mathsf{prev}(t[i{+}1 : i{+}k{+}1])$ and $\mathsf{prev}(t[b_k{+}1 : b_k{+}k]) = \mathsf{prev}(t[i{+}1 : i + k])$, $\mathsf{prev}(t[b_k + 1 : b_k + k])$ is a prefix of $\mathsf{prev}(t[b_{k+1} + 1 : b_{k+1} + k + 1])$. This implies that if $\mathsf{prev}(t[b_k + 1 : b_k + k])$ is represented by $\mathsf{PPH}(t)$, then $\mathsf{prev}(t[b_{k+1} + 1 : b_{k+1} + (k + 1)])$ is also represented by $\mathsf{PPH}(t)$. By induction, we have that $\mathsf{prev}(t[b_{j-i} + 1 : b_{j-i} + j - i]) = \mathsf{prev}(t[i + 1 : j])$ is represented by $\mathsf{PPH}(t)$. Applying the same argument inductively, it is immediate that $\mathsf{prev}(t[i + h : j])$ with $2 \leq h \leq j - i$ are also represented by $\mathsf{PPH}(t)$. □

In the next section, we show how to construct our RL p-position heap $\mathsf{PPH}(t)$ for an input text p-string t of length n in $O(n \log(|\Sigma| + |\Pi|))$ time and $O(n)$ space.

3.3 Right to left construction of parameterized position heaps

In this section, we present our algorithm which constructs $\mathsf{PPH}(t)$ of a given p-string t in a right-to-left online manner. The key to our construction algorithm is the use of *reversed suffix links*, which will be defined in the following subsection.

3.3.1 Reversed suffix links

For convenience, we will sometimes identify each node v of $\mathsf{PPH}(t)$ with the path label from the root to v. In our right-to-left online construction of $\mathsf{PPH}(t)$, we use the *reversed suffix links*,

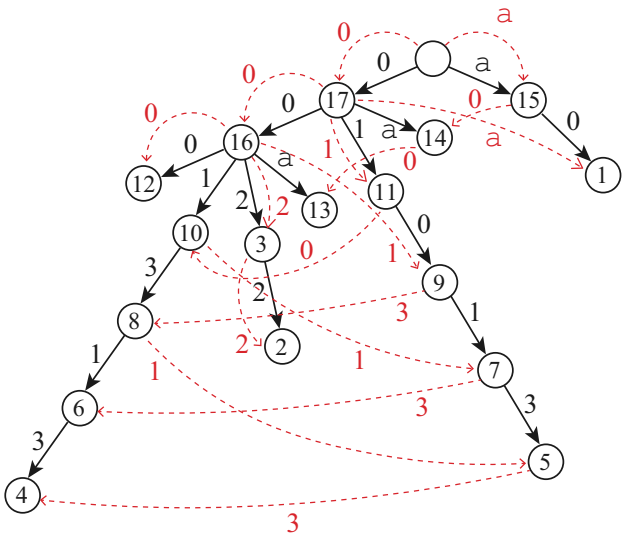

Figure 3.2: Illustration of the reversed suffix links of $\mathsf{PPH}(t)$ with the same p-string $t =$ `axyxyyxxyyxxzyazy` as in Figure 3.1. The reversed suffix links and their labels are shown in red.

which are a generalization of the *Weiner links* that are used in right-to-left construction of the suffix tree [45] for (standard) string matching:

Definition 6 (Reversed suffix links)**.** *For any node v of $\mathsf{PPH}(t)$ and a character $a \in \Sigma \cup [0 : n-1]$, let*

$$\mathsf{rslp}(a, v) = \begin{cases} av & \text{if } a \in \Sigma \cup \{0\} \text{ and } av \text{ is represented by } \mathsf{PPH}(t), \\ u & \text{if } a \in [1 : n-1],\ v[a] = 0 \text{ and} \\ & u = 0v[1 : a-1]av[a+1 : |v|] \text{ is represented by } \mathsf{PPH}(t), \\ \text{undefined} & \text{otherwise}. \end{cases}$$

It is clear that by taking one rslp link from a node, then the node depth (and hence the string length) increases exactly one.

Observe that the first case of of the definition of $\mathsf{rslp}(a, v)$ is a direct extension of the Weiner links, where $\mathsf{rslp}(a, v)$ points to the node av that is obtained by prepending a to v. The second case, however, is a special case that arises in parameterized pattern matching. The following lemma ensures that our reversed suffix links rslp are well defined:

Lemma 3. *For any node v in $\mathsf{PPH}(t)$ and a character $a \in \Sigma \cup [0 : n-1]$, let $\mathsf{rslp}(a, v) = u$, where u is a node of $\mathsf{PPH}(t)$. Then, for any string X such that $\mathsf{prev}(X) = u$, $\mathsf{prev}(X[2 : |X|]) = v$.*

Proof. In the first case of the definition of $\mathsf{rslp}(a, v)$ where $a \in \Sigma \cup \{0\}$, we have $\mathsf{prev}(X) = u = av$. Hence, $\mathsf{prev}(X[2 : |X|]) = \mathsf{prev}(X)[2 : |X|] = u[2 : |u|] = v$.

In the second case of the definition of $\mathsf{rslp}(a, v)$ where $a \in [1 : n-1]$, we have $\mathsf{prev}(X) = u = 0v[1 : a-1]av[a+1 : |v|]$, which implies that $X[1] = X[a+1]$ and $X[1] \neq X[i]$ for any $2 \leq i \leq a$. Thus, $\mathsf{prev}(X[2 : |X|]) = v[1 : a-1]0v[a+1 : |v|] = v$. □

The next proposition shows that there is a monotonicity in the labels of the reversed suffix links that come from the nodes in the same path of $\mathsf{PPH}(t)$.

Proposition 1. *Suppose there is a reversed suffix link* $\mathsf{rslp}(a, v)$ *of a node* v *with* $a \in \Sigma \cup [0 : n-1]$. *Let* u *be any ancestor of* v. *Then, if* $a \in \Sigma \cup \{0\}$, u *has a reversed suffix link* $\mathsf{rslp}(a, u)$. *Also, if* $a \in [1 : n-1]$ *and* $|u| \geq a$, *then* u *has a reversed suffix link* $\mathsf{rslp}(a, u)$, *and if* $a \in [1 : n-1]$ *and* $|u| < a$, *then* u *has a reversed suffix link* $\mathsf{rslp}(0, u)$.

Proof. It suffices for us to show that the lemma holds for the parent v' of v, since then the lemma inductively holds for any ancestor of v. Note that $v' = v[1 : |v|-1]$. Let $w = \mathsf{rslp}(a, v)$.

If $a \in \Sigma \cup \{0\}$, then $w = av$. Hence, the parent of w is $w[1 : |w|-1] = av[1 : |v|-1] = av'$. Therefore, there is a reversed suffix link $\mathsf{rslp}(a, v')$.

If $a \in [1 : n-1]$ and $|v'| = |v| - 1 \geq a$, then it follows from the definition of $\mathsf{rslp}(a, v)$ that $v[a] = 0$ and $w = 0v[1 : a-1]av[a+1 : |v|]$. Since $|v'| \geq a$, we have that $v'[a] = 0$ and $|v| \geq a+1$. Thus $w[1 : |w|-1] = 0v[1 : a-1]av[a+1 : |v|-1]$ is represented by $\mathsf{PPH}(t)$. Consequently, there is a reversed suffix link $\mathsf{rslp}(a, v')$.

If $a \in [1 : n-1]$ and $|v'| = |v| - 1 = a - 1$, then it follows from the definition of $\mathsf{rslp}(a, v)$ that $v[a] = v[|v|] = 0$ and $w = 0v[1 : |v|-1]a$. Thus $w[1 : |w|-1] = 0v[1 : |v|-1] = 0v'$ is represented by $\mathsf{PPH}(t)$. Consequently, there is a reversed suffix link $\mathsf{rslp}(a, v')$. □

3.3.2 Adding a new node

Our algorithm processes a given p-string t of length n from right to left and maintains $\mathsf{PPH}(t[i :])$ in decreasing order of $i = n, \ldots, 1$. Initially, we begin with $\mathsf{PPH}(\varepsilon)$ which consists of the root r representing the empty string ε. For convenience, we use an auxiliary node $\perp$ as a parent of the root r, and create reversed suffix links $\mathsf{rslp}(a, \perp) = r$ for every $a \in \Sigma \cup \{0\}$.

Now suppose we have constructed $\mathsf{PPH}(t[i :])$ for $1 < i \leq n$, and we will update it to $\mathsf{PPH}(t[i-1 :])$. In so doing, we begin with node v_i such that $\mathsf{id}(v_i) = i$. We know the locus of this node v_i since v_i is the node that was inserted at the last step when $\mathsf{PPH}(t[i :])$ was

constructed from $\mathsf{PPH}(t[i+1:])$. Note also that this node v_i is a leaf in $\mathsf{PPH}(t[i:])$. We climb up the path from v_i until finding its lowest ancestor v'_i that satisfies the following. There are three cases:

1. If $t[i-1] \in \Sigma$, then v'_i is the lowest ancestor of v_i such that $\mathsf{rslp}(t[i-1], v_i)$ is defined.
2. If $t[i-1] \in \Pi$ and $t[i-1] \neq t[j]$ for any $i \leq j \leq n$, then v'_i is the lowest ancestor of v_i such that $\mathsf{rslp}(0, v_i)$ is defined.
3. Otherwise, let $d = j - i$ where j is the smallest position such that $i \leq j \leq n$ and $t[i-1] = t[j]$. Then v'_i is the lowest ancestor of v_i such that $\mathsf{rslp}(d, v'_i)$ is defined if it exists, and v'_i is the lowest ancestor of v_i such that $\mathsf{rslp}(0, v'_i)$ is defined otherwise.

Let u_i be the node of $\mathsf{PPH}(t[i:])$ that is pointed by the reversed suffix link of v'_i as above. Then, we create a new node v_{i-1} as a child of u_i such that $\mathsf{id}(v_{i-1}) = i-1$. The new edge (u_i, v_{i-1}) is labeled by $\mathsf{prev}(t[i-1:])[|u_i|+1]$. We repeat the above procedure for all positions i in t in decreasing order. See also Figure 3.3 for concrete examples.

Lemma 4. *The above algorithm correctly updates* $\mathsf{PPH}(t[i:])$ *to* $\mathsf{PPH}(t[i-1:])$.

Proof. Note that v_i and v'_i are prefixes of $\mathsf{prev}(t[i:])$. Let a be the character in $\Sigma \cup [0:n-1]$ that is used in the reversed suffix link as above.

In Cases 1 and 2 above, we have $a = t[i-1] \in \Sigma$ or $a = 0$. Then it is clear that av'_i is a prefix of $\mathsf{prev}(t[i-1:])$. Since v'_i is the lowest ancestor of v_i for which $\mathsf{rslp}(a, v'_i)$ is defined, $u_i = av'_i$ is the longest prefix of $\mathsf{prev}(t[i-1:])$ that is represented by $\mathsf{PPH}(t[i:])$. Hence, the new node v_{i-1} and its incoming edge labeled by $\mathsf{prev}(t[i-1:])[|u_i|+1]$ are correctly inserted.

Consider Case 3 above. We first try to find v'_i in the first sub-case, where $a = d \geq 1$. If it exists, then v'_i is the lowest ancestor of v_i such that $\mathsf{rslp}(d, v'_i)$ is defined, and thus $\mathsf{rslp}(d, v'_i) = 0v'_i[1:d-1]dv'_i[d+1:|v'_i|]$. It now follows from Lemma 2 that $u_i = 0v'_i[1:d-1]dv'_i[d+1:|v'_i|]$ is the longest prefix of $\mathsf{prev}(t[i-1:])$ that is represented by $\mathsf{PPH}(t[i:])$. Hence, the new node v_{i-1} and its incoming edge labeled by $\mathsf{prev}(t[i-1:])[|u_i|+1]$ are correctly inserted in this sub-case. It is clear that v'_i in the first sub-case is at least of depth d. Hence, if we arrive at the ancestor of v_i of depth $d-1$ without encountering the lowest ancestor satisfying the condition of the first sub-case, then we try to find the lowest ancestor of v_i that has a reversed suffix link labeled by 0 (second sub-case). Thus, by a similar argument to Case 2, the new node v_{i-1} its incoming edge labeled by $\mathsf{prev}(t[i-1:])[|u_i|+1]$ are correctly inserted in this second sub-case. □

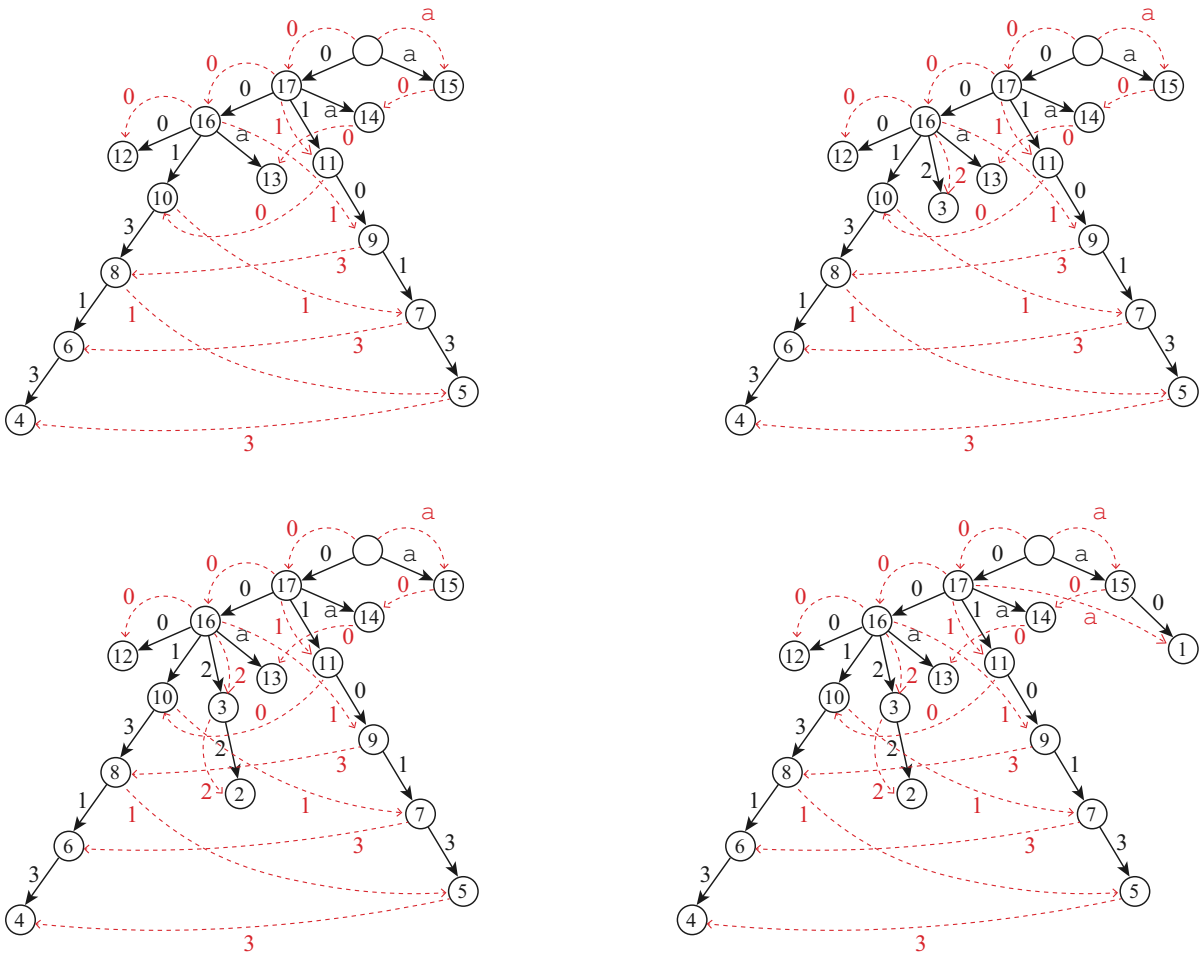

Figure 3.3: A snapshot of updating $\mathsf{PPH}(t[i:])$ for $i = 4, 3, 2, 1$ with the same p-string $t =$ axyxyyxxyyxxzyazy as in Figures 3.1 and 3.2. First, we update $\mathsf{PPH}(t[4:])$ (upper left) to $\mathsf{PPH}(t[3:])$ (upper right). Since $t[3] = t[5] =$ y and $d = 5-3 = 2$, we first try to find the lowest ancestor of the node with id 4 that has a reversed suffix link labeled with $d = 2$ by climbing up the path. However, it does not exist, and then we arrive at the lowest ancestor with id 17 whose depth is 1 (< 2). Hence the second sub-case of Case 3 is applied, and using its reversed suffix link we move to the node with id 16. The new node with id 3 is inserted as its child. Next, we update $\mathsf{PPH}(t[3:])$ (upper right) to $\mathsf{PPH}(t[2:])$ (lower left). Since $t[2] = t[4] =$ x and $d = 4 - 2 = 2$, we first try to find the lowest ancestor of the node with id 3 that has a reversed suffix link labeled with $d = 2$ by climbing up the path, and we arrive at the node with id 16. Hence the first sub-case of Case 3 is applied, and using its reversed suffix link we move to the node with id 3. The new node with id 2 is inserted as its child. Finally, we update $\mathsf{PPH}(t[2:])$ (lower left) to $\mathsf{PPH}(t[1:])$ (lower right). Since $t[1] =$ a $\in \Sigma$, Case 1 is applied. Thus we try to find the lowest ancestor of the node with id 2 that has a reversed suffix link labeled with a by climbing up the path, and we arrive at the root. Using its reversed suffix link, we move to the node with id 15. The new node with id 1 is inserted as its child.

3.3.3 Adding a new reversed suffix link

After inserting the new node v_{i-1}, we need to maintain the reversed suffix links corresponding to v_{i-1}.

Lemma 5. *There is exactly one reversed suffix link that points to the new node v_{i-1} in $\mathsf{PPH}(t[i-1:])$. Moreover, this reversed suffix link comes from the ancestor of v_i of depth $|v'_i|+1$.*

Proof. Suppose on the contrary that there are two distinct nodes x and y each of which has a reversed suffix link pointing to v_{i-1}. The label of any reversed suffix link that points to v_{i-1} is uniquely determined by the path label from the root to v_{i-1}. Therefore, the reversed suffix links of x and y that point to v_{i-1} are both labeled by the same symbol. This means that $x = y$, however, this contradicts the definition of the p-position heap. Hence, there is at most one node which has a reversed suffix link that points to v_{i-1}.

Let z_i be the ancestor of v_i of depth $|v'_i|+1$. Also, let $x = (t[i:])[|u_i|] = (t[i-1:])[|u_i|+1] = t[i+|u_i|-1]$, namely, x is the text character that corresponds to the label of the edge (v'_i, z_i) that is on the path from the root to v_i, [1] and to the label of the new edge (u_i, v_{i-1}). If $x \in \Pi$ and $i+|u_i|-1$ is the smallest position in $t[i-1:]$ such that $t[i-1] = t[i+|u_i|-1]$, then (v'_i, z_i) is labeled with 0 while (u_i, v_{i-1}) is labeled with $|u_i|$. Otherwise, the label of the new edge (u_i, v_{i-1}) must be equal to that of (v'_i, z_i). It follows from the definition of reversed suffix links that in both cases the reversed suffix link to v_{i-1} comes from z_i. □

Lemma 6. *There is no reversed suffix link that comes from the new node v_{i-1} in $\mathsf{PPH}(t[i-1:])$.*

Proof. Suppose on the contrary that there is a reversed suffix link from v_{i-1} in $\mathsf{PPH}(t[i-1:])$, and let w be the node that is pointed by this reversed suffix link. Notice that $|w| = |v_{i-1}|+1$. Let $t[j:]$ be the suffix of t for which this node w was inserted, namely, $\mathsf{id}(w) = j > i-1$. By Lemma 2, for any substring X of $t[j : j+|w|-1]$, $\mathsf{prev}(X)$ is represented by $\mathsf{PPH}(t[j:])$, and hence it is also represented by $\mathsf{PPH}(t[i:])$ since $j \le i$. Recall that $\mathsf{prev}(t[j+1 : j+|w|-1]) = \mathsf{prev}(t[i-1 : i+|v_{i-1}|])$, which implies that the node v_{i-1} existed already in $\mathsf{PPH}(t[i:])$. However, this contradicts that v_{i-1} is the node that was inserted when $\mathsf{PPH}(t[i:])$ was updated to $\mathsf{PPH}(t[i-1:])$. □

Due to Lemmas 5 and 6, there is only one reversed suffix link that is newly inserted in $\mathsf{PPH}(t[i-1:])$.

[1] HB Note: please check

3.3.4 Complexity analysis

Lemma 7. *The proposed algorithm runs in a total of* $O(n \log(|\Sigma|+|\Pi|))$ *time with* $O(n)$ *space.*

Proof. For each $i = n, \ldots, 1$, the algorithm updates $\mathsf{PPH}(t[i:])$ to $\mathsf{PPH}(t[i-1:])$. The update begins with node v_i such that $\mathsf{id}(v_i) = i$, and climbs up the path to v'_i. It takes a reversed suffix link from v'_i and moves to u_i of depth $|v'_i| + 1$, and the new node v_{i-1} of depth $|v'_i| + 2$ with $\mathsf{id}(v_{i-1}) = i - 1$ is inserted. Hence the total number of nodes visited when updating $\mathsf{PPH}(t[i:])$ to $\mathsf{PPH}(t[i-1:])$ is $|v_i| - |v'_i| + 2 = |v_i| - |v_{i-1}| + 4$. Thus, the total number of nodes visited for all $i = n, \ldots, 1$ sums up to $\sum_{i=n}^{2}(|v_i| - |v_{i-1}| + 4) = |v_n| - |v_1| + 4(n-1) = O(n)$. At each node that we visit, it takes $O(\log(|\Sigma|+|\Pi|))$ time to search for the corresponding reversed suffix link, as well as inserting a new edge. Hence, the total time cost is $O(n \log(|\Sigma| + |\Pi|))$.

It is clear that the number of nodes in $\mathsf{PPH}(t)$ is $n + 2$, including the root and the auxiliary node $\perp$. It follows from Lemmas 5 and 6 that the number of reversed suffix links coming out from the root, the internal nodes, and the leaves is $n + 1$. As for the reversed suffix links that come from $\perp$ to the root, we add a new reversed suffix link labeled with $t[i-1]$ only if $t[i-1] \in \Sigma$ and $t[i-1] \neq t[j]$ for any $j < i - 1$. This way, we can maintain these reversed suffix links from $\perp$ in an online manner, using $O(n)$ space. □

We have proven the following theorem, which is the main result of this paper.

Theorem 1. *For an input p-string* t *of length* n*, the proposed algorithm constructs* $\mathsf{PPH}(t[i:])$ *in a right-to-left online manner for* $i = n, \ldots, 1$*, in a total of* $O(n \log(|\Sigma| + |\Pi|))$ *time with* $O(n)$ *space.*

3.4 Parameterized pattern matching with augmented $\mathsf{PPH}(t)$

Ehrenfeucht et al. [17] introduced *maximal reach pointers*, which used for efficient pattern matching queries on position heaps. Diptarama et al. [16] introduced maximal reach pointers for their LR p-position heaps, and showed how to perform pattern matching queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time, where m is the length of a given pattern p-string and occ is the number of occurrences to report. We can naturally extend the notion of maximal reach pointers to our RL p-position heaps, as follows:

Definition 7 (Maximal reach pointers)**.** *For each position* $1 \leq i \leq n$ *in* t*, the* maximal reach

$\mathsf{prev}(t[12:])$	a
$\mathsf{prev}(t[11:])$	0a
$\mathsf{prev}(t[10:])$	00a
$\mathsf{prev}(t[9:])$	a00a
$\mathsf{prev}(t[8:])$	0a03a
$\mathsf{prev}(t[7:])$	00a33a
$\mathsf{prev}(t[6:])$	a00a33a
$\mathsf{prev}(t[5:])$	0a03a33a
$\mathsf{prev}(t[4:])$	00a33a33a
$\mathsf{prev}(t[3:])$	a00a33a33a
$\mathsf{prev}(t[2:])$	0a03a33a33a
$\mathsf{prev}(t[1:])$	01a03a33a33a

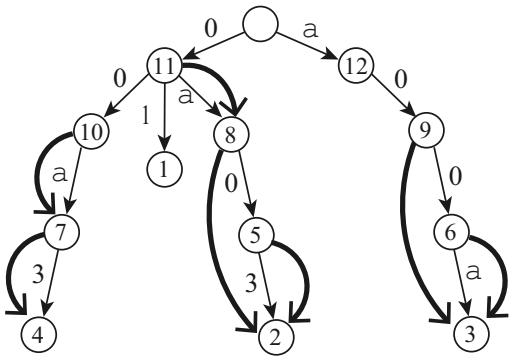

Figure 3.4: To the left is the list of $\mathsf{prev}(t[i:])$ for p-string $t = \texttt{xxayxayxayxa}$ of length 12, where $\Sigma = \{\texttt{a}\}$ and $\Pi = \{\texttt{x}, \texttt{y}\}$. To the right is an illustration for augmented $\mathsf{PPH}(t)$, where the maximal reach pointers are indicated by the bold arrows. The wavy underlined prefix of each $\mathsf{prev}(t[i:])$ in the left list denotes the longest prefix of $\mathsf{prev}(t[i:])$ that is represented by $\mathsf{PPH}(t)$, and hence it is the destination of $\mathsf{mrp}(i)$.

pointer *of the node* v *with* $\mathsf{id}(v) = i$ *points to the deepest node* u *of* $\mathsf{PPH}(t)$ *such that* u *is a prefix of* $\mathsf{prev}(t[i:])$.

We denote by $\mathsf{mrp}(i)$ the pointer of node v such that $\mathsf{id}(v) = i$. The *augmented* $\mathsf{PPH}(t)$ is $\mathsf{PPH}(t)$ with the maximal reach pointers of all nodes. For simplicity, if $\mathsf{mrp}(i)$ points to the node with id i, then we omit this pointer. See Figure 3.4 for an example of maximal reach pointers and augmented $\mathsf{PPH}(t)$.

Lemma 8. *For every* $1 \leq i \leq n$*, we can compute* $\mathsf{mrp}(i)$ *in a total of* $O(n \log(|\Sigma| + |\Pi|))$ *time with* $O(n)$ *space.*

Proof. We compute $\mathsf{mrp}(i)$ for each position $i = 1, \ldots, n$ increasing order. In so doing, we use the *forward* suffix link that are the reversals of the reversed suffix links. For simplicity, we will call forward suffix links as suffix links. Since there is exactly one in-coming reversed suffix link to each node, there is also exactly one out-going suffix link from each node. Let $\mathsf{slp}(v)$ denote the node that the suffix link of v points to.

We begin with node v_1 such that $\mathsf{id}(v_1) = 1$. Since we have built $\mathsf{PPH}(t[i:])$ in decreasing order of i, v_1 is a leaf of $\mathsf{PPH}(t)$ and it is the deepest node that is a prefix of $\mathsf{prev}(t[1:])$. Now

we take the suffix link of v_1, and let $u_1 = \mathsf{slp}(v_1)$. Since $\mathsf{prev}(t[1 : |v_1|]) = v_1$, it follows from Lemma 3 that $u_1 = \mathsf{prev}(t[2 : |v_1|])$, which implies that u_1 is a prefix of $\mathsf{prev}(t[2 :])$. Then the deepest node v_2 that is a prefix of $\mathsf{prev}(t[2 :])$ can be found by traversing the corresponding path from node u_1. Then, we make a pointer to v_2 from the node w with $\mathsf{id}(w) = 2$. We iteratively perform the same procedure for all positions i in increasing order.

To analyze the time complexity, we can use a similar argument as in Lemma 7. For each i, the number of nodes traversed is $|v_{i+1}| - |u_i| + 1 = |v_{i+1}| - |v_i| + 2$. Thus, the total number of nodes visited sums up to $\sum_{i=1}^{n-1}(|v_{i+1}| - |v_i| + 2) = |v_n| - |v_1| + 2(n-1) = O(n)$. Since it takes $O(\log(|\Sigma| + |\Pi|))$ time to search for each corresponding edge in the traversal, the total running time is $O(n \log(|\Sigma| + |\Pi|))$.

The space requirement is clearly $O(n)$. □

It is straightforward that by applying Diptarama et al.'s pattern matching algorithm to our $\mathsf{PPH}(t)$ augmented with maximal reach pointers, parameterized pattern matching can be done in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.

Corollary 1. *Using our augmented* $\mathsf{PPH}(t)$*, one can perform parameterized pattern matching queries in* $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ *time.*

3.5 Conclusions and further work

This paper proposed a new indexing structure for parameterized pattern matching, called RL p-position heaps, that are built in a right-to-left online manner. We proposed a Weiner-type construction algorithm for our RL p-position heaps that runs in $O(n \log(|\Sigma| + |\Pi|))$ time with $O(n)$ space, for a given text p-string of length n over a static alphabet Σ and a parameterized alphabet Π. The key to our efficient construction is how to label the reversed suffix links. By augmenting our position heap with maximal reach pointers, one can perform parameterized pattern matching in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time, where m is the length of a query pattern and occ is the number of occurrence to report.

Our future work includes the following:

- Would it be possible to shave the $m|\Pi|$ term in the pattern matching time using parameterized position heaps? Other data structures such as parameterized suffix trees achieve better $O(m \log(|\Sigma| + |\Pi|) + occ)$ time [4].

Chapter 4

Parameterized Position Heap for a Trie

4.1 Introduction

Diptarama et al. [16] proposed a new indexing structure called the *parameterized position heap* (*p-position heap*). They showed how to construct the p-position heap of a given p-string S of length n in $O(n \log(|\Sigma| + |\Pi|))$ time with $O(n)$ space in a *left-to-right online* manner. Their algorithm is based on Kucherov's position heap construction algorithm [34] which scans the input text from left to right. Recently, Fujisato et al. [20] presented another variant of the p-position heap that can be constructed in a *right-to-left online* manner, in $O(n \log(|\Sigma| + |\Pi|))$ time with $O(n)$ space. This algorithm is based on Ehrenfeucht et al.'s algorithm [17] which scans the input text from right to left. Both versions of p-positions heaps support p-matching queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.

This paper deals with indexing on *multiple* texts; in particular, we consider the case where those multiple texts are represented by a *trie*. It should be noted that our trie is a so-called *common suffix trie* (*CS trie*) where the common suffixes of the texts are merged and the edges are reversed (namely, each text is represented by a path from a leaf to the root). See also Figure 4.1 for an example of a CS trie. There are two merits in representing multiple texts by a CS trie: Let N be the size of the CS trie of the multiple strings of total length Z. (1) N can be as small as $\Theta(\sqrt{Z})$ when the multiple texts share a lot of common long suffixes. (2) The number of distinct suffixes of the texts is equal to the number of the nodes in the CS trie, namely N. On the other hand, this is not the case with the ordinal common prefix trie (CP trie), namely, the number of distinct suffixes in the CP trie can be super-linear in the number of its nodes. Since most, if not all, indexing structures require space that is dependent of the number of distinct suffixes, the CS trie is a more space economical representation for indexing than its

CP trie counterpart.

Let N be the size of a given CS trie. Due to Property (1) above, it is significant to construct an indexing structure *directly* from the CS trie. Note that if we expand all texts from the CS trie, then the total string length can blow up to $O(N^2)$. Breslauer [8] introduced the suffix tree for a CS trie which occupies $O(N)$ space, and proposed an algorithm which constructs it in $O(N|\Sigma|)$ time and working space. Using the suffix tree of a CS trie, one can report all paths of the CS trie that exactly matches with a given pattern of length m in $O(m \log |\Sigma| + occ)$ time, where occ is the number of such paths to report. Shibuya [44] gave an optimal $O(N)$-time construction for the suffix tree for a CS trie in the case of integer alphabets of size $N^{O(1)}$. Nakashima et al. [39] proposed the position heap for a CS trie, which can be built in $O(N|\Sigma|)$ time and working space and supports exact pattern matching in $O(m \log |\Sigma| + occ)$ time. Later, an optimal $O(N)$-time construction algorithm for the position heap for a CS trie in the case of integer alphabets of size $N^{O(1)}$ was presented [40].

In this paper, we propose the *parameterized position heap* for a CS trie $\mathcal{T}$, denoted by $\mathsf{PPH}(\mathcal{T})$, which is the *first* indexing structure for p-matching on a trie. We show that $\mathsf{PPH}(\mathcal{T})$ occupies $O(N)$ space, supports p-matching queries in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time, and can be constructed in $O(N(|\Sigma| + |\Pi|))$ time and working space. Hence, we achieve optimal pattern matching and construction in the case of constant-sized alphabets. The proposed construction algorithm is fairly simple, yet uses a non-trivial idea that converts a given CS trie into a smaller trie based on the p-matching equivalence. The simplicity of our construction algorithm comes from the fact that each string stored in (p-)position heaps is represented by an explicit node, while it is not the case with (p-)suffix trees. This nice property makes it easier and natural to adopt the approaches by Brealauer [8] and by Fujisato et al. [20] that use *reversed suffix links* in order to process the texts from left to right. We also remark that all existing p-suffix tree construction algorithms [4, 32, 44] in the case of a single text require somewhat involved data structures due to non-monotonicity of parameterized suffix links [3, 4], but our p-position heap does not need such a data structure even in the case of CS tries (this will also be discussed in the concluding section).

4.2 Preliminaries

A *common suffix trie* (*CS trie*) $\mathcal{T}$ is a reversed trie such that (1) each edge is directed towards the root, (2) each edge is labeled with a character from $\Sigma \cup \Pi$, and (3) the labels of the in-coming

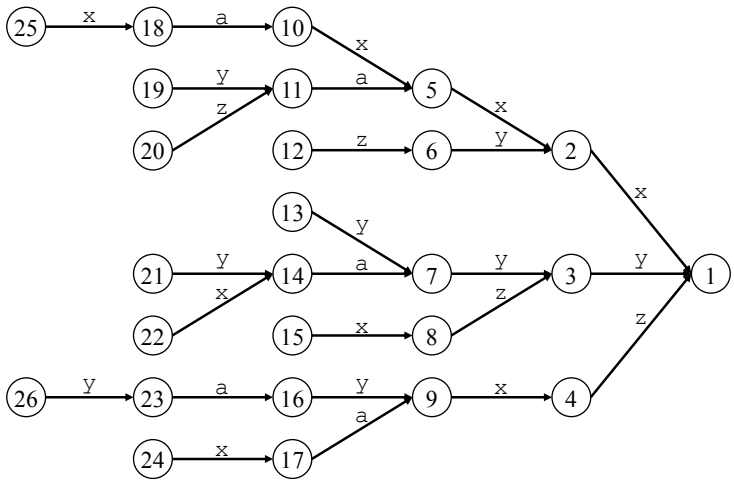

Figure 4.1: The CS trie for a set $\{\mathtt{xaxxx}, \mathtt{yaxx}, \mathtt{zaxx}, \mathtt{zyx}, \mathtt{yyy}, \mathtt{yayy}, \mathtt{xayy}, \mathtt{xzy}, \mathtt{yayxz}, \mathtt{xaxz}\}$ of 10 p-strings over $\Sigma \cup \Pi$, where $\Sigma = \{\mathtt{a}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.

edges to each node are mutually distinct. Each node of the trie represents the (p-)string obtained by concatenating the labels on the path from the node to the root.

An example of a CS trie is illustrated in Fig. 4.1. $\mathsf{CST}(W)$ denotes the CS trie which represents a set W of (p-)strings.

4.3 Parameterized position heap of a common suffix trie

In this section, we introduce the parameterized pattern matching (p-matching) problem on a common suffix trie that represents a set of p-strings, and propose an indexing data structure called a parameterized position heap of a trie.

4.3.1 p-matching problem on a common suffix trie

We introduce the *p-matching problem* on a common suffix trie $\mathcal{T}$ and a pattern p. We will say that a node v in a common suffix trie p-matches with a pattern p-string p if the prefix of length $|p|$ of the p-string represented by v and p p-match. In this problem, we preprocess a given common suffix trie $\mathcal{T}$ so that later, given a query pattern p, we can quickly answer every node v of $\mathcal{T}$ whose prefix of length $|p|$ and p p-match. For the common suffix trie in Fig. 4.1, when given query pattern $P = \mathtt{azy}$, then we answer the nodes 17 and 23.

Let $W_{\mathcal{T}}$ be the set of all p-strings represented by nodes of $\mathcal{T}$. By the definition of the common suffix trie, there may exist two or more nodes which represent different p-strings, but

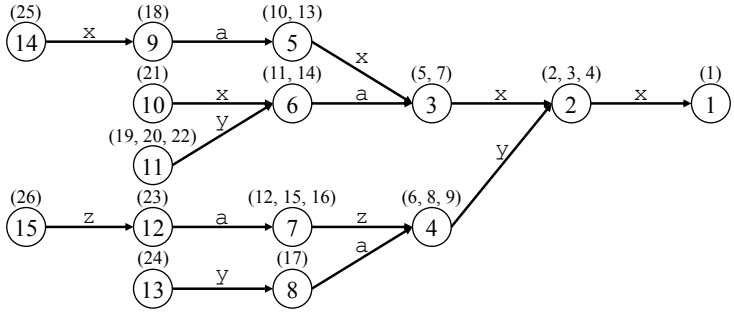

Figure 4.2: Illustration of $\mathsf{pCST}(\mathcal{T})$ for $\mathcal{T}$ (where $\mathcal{T}$ is the common suffix trie illustrated in Fig. 4.1). Each node of $\mathsf{pCST}(\mathcal{T})$ corresponds to nodes of $\mathcal{T}$ which are labeled by elements in the tuple above the node of $\mathsf{pCST}(\mathcal{T})$. For example, the node of $\mathsf{pCST}(\mathcal{T})$ labeled 6 corresponds to the nodes of $\mathcal{T}$ labeled 11 and 14.

p-match. We consider the common suffix trie which merges such nodes into the same node by using the representative of the parameterized equivalent class of these strings. We define the set $\mathsf{pcs}(\mathcal{T})$ of p-strings as follows: $\mathsf{pcs}(\mathcal{T}) = \{\mathsf{spe}(w^R)^R \mid w \in W_{\mathcal{T}}\}$. Then, the reversed trie which we want to consider is $\mathsf{CST}(\mathsf{pcs}(\mathcal{T}))$. We refer to this reversed trie as the *parameterized-common suffix trie* of $\mathcal{T}$, and denote it by $\mathsf{pCST}(\mathcal{T})$ (i.e., $\mathsf{pCST}(\mathcal{T}) = \mathsf{CST}(\mathsf{pcs}(\mathcal{T}))$). Each node of $\mathsf{pCST}(\mathcal{T})$ stores pointers to its corresponding node(s) of $\mathcal{T}$. Then, by solving the p-matching problem on $\mathsf{pCST}(\mathcal{T})$, we can immediately answering p-matching queries on $\mathcal{T}$. Fig. 4.2 shows an example of $\mathsf{pCST}(\mathcal{T})$. In the rest of this paper, N denotes the number of nodes of $\mathcal{T}$ and N_p denotes the number of nodes of $\mathsf{pCST}(\mathcal{T})$. Note that $N \geq N_p$ always holds.

4.3.2 Parameterized position heap of a common suffix trie

Let $\mathcal{S} = \langle s_1, \ldots, s_k \rangle$ be a sequence of strings such that for any $1 < i \leq k$, $s_i \notin \mathsf{Prefix}(s_j)$ for any $1 \leq j < i$.

Definition 8 (Sequence hash trees [10])**.** *The* sequence hash tree *of a sequence* $\mathcal{S} = \langle s_1, \ldots, s_k \rangle$ *of strings, denoted* $\mathsf{SHT}(\mathcal{S}) = \mathsf{SHT}(\mathcal{S})^k$*, is a trie structure that is recursively defined as follows: Let* $\mathsf{SHT}(\mathcal{S})^i = (V_i, E_i)$*. Then*

$$\mathsf{SHT}(\mathcal{S})^i = \begin{cases} (\{\varepsilon\}, \emptyset) & \text{if } i = 1, \\ (V_{i-1} \cup \{u_i\}, E_{i-1} \cup \{(v_i, a, u_i)\}) & \text{if } 2 \leq i \leq k, \end{cases}$$

where v_i *is the longest prefix of* s_i *which satisfies* $v_i \in V_{i-1}$*,* $a = s_i[|v_i| + 1]$*, and* u_i *is the shortest prefix of* s_i *which satisfies* $u_i \notin V_{i-1}$*.*

Note that since we have assumed that each $s_i \in \mathcal{S}$ is not a prefix of s_j for any $1 \leq j < i$, the new node u_i and new edge (v_i, a, u_i) always exist for each $1 \leq i \leq k$. Clearly $\mathsf{SHT}(\mathcal{S})$ contains k nodes (including the root).

Let $\mathcal{W}_\mathcal{T} = \langle \mathsf{spe}(w_1), \ldots, \mathsf{spe}(w_{N_p}) \rangle$ be a sequence of p-strings such that $\{w_1, \ldots, w_{N_p}\} = \mathsf{pcs}(\mathcal{T})$ and $|w_i| \leq |w_{i+1}|$ for any $1 \leq i \leq N_p - 1$. $\mathcal{W}_\mathcal{T}(i)$ denote the sequence $\langle \mathsf{spe}(w_1), \ldots, \mathsf{spe}(w_i) \rangle$ for any $1 \leq i \leq N_p$, and $\mathsf{pCST}(\mathcal{T})^i$ denote the common suffix trie of $\{\mathsf{spe}(w_1), \ldots, \mathsf{spe}(w_i)\}$, namely, $\mathsf{pCST}(\mathcal{T})^i = \mathsf{CST}(\{\mathsf{spe}(w_1), \ldots, \mathsf{spe}(w_i)\})$. The node of $\mathsf{pCST}(\mathcal{T})$ which represents w_i is denoted by c_i. Then, our indexing data structure is defined as follows.

Definition 9 (Parameterized positions heaps of a CST)**.** *The* parameterized position heap *(*p-position heap*) for a common suffix trie $\mathcal{T}$, denoted by $\mathsf{PPH}(\mathcal{T})$, is the sequence hash tree of $\mathcal{W}_\mathcal{T}$ i.e., $\mathsf{PPH}(\mathcal{T}) = \mathsf{SHT}(\mathcal{W}_\mathcal{T})$.*

Let $\mathsf{PPH}(\mathcal{T})^i = \mathsf{SHT}(\mathcal{W}_\mathcal{T}(i))$ for any $1 \leq i \leq N_p$ (i.e., $\mathsf{PPH}(\mathcal{T})^{N_p} = \mathsf{PPH}(\mathcal{T})$). The following lemma shows the exact size of $\mathsf{PPH}(\mathcal{T})$.

Lemma 9. *For any common suffix trie $\mathcal{T}$ such that the size of $\mathsf{pCST}(\mathcal{T})$ is N_p, $\mathsf{PPH}(\mathcal{T})$ consists of exactly N_p nodes. Also, there is a one-to-one correspondence between the nodes of $\mathsf{pCST}(\mathcal{T})$ and the nodes of $\mathsf{PPH}(\mathcal{T})$.*

Proof. Initially, $\mathsf{PPH}(\mathcal{T})^1$ consists only of the root that represents ε since $w_1 = \varepsilon$. Let i be an integer in $[1 : N_p]$. Since w_i does not p-match with w_j and $|\mathsf{spe}(w_i)| \geq |\mathsf{spe}(w_j)|$ for any $1 \leq j < i$, there is a prefix of $\mathsf{spe}(w_i)$ that is not represented by any node of $\mathsf{PPH}(\mathcal{T})^{i-1}$. Therefore, when we construct $\mathsf{PPH}(\mathcal{T})^i$ from $\mathsf{PPH}(\mathcal{T})^{i-1}$, then exactly one node is inserted, which corresponds to the node representing w_i. □

Let h_i be the node of $\mathsf{PPH}(\mathcal{T})$ which corresponds to w_i. For any p-string $p \in (\Sigma \cup \Pi)^+$, we say that p is *represented* by $\mathsf{PPH}(\mathcal{T})$ iff $\mathsf{PPH}(\mathcal{T})$ has a path which starts from the root and spells out p.

Ehrenfeucht et al. [17] introduced *maximal reach pointers*, which are used for efficient pattern matching queries on position heaps. Diptarama et al. [16] and Fujisato et al. [20] also introduced maximal reach pointers for their p-position heaps, and showed how efficient pattern matching queries can be done. We can naturally extend the notion of maximal reach pointers to our p-position heaps:

Definition 10 (Maximal reach pointers)**.** *For each $1 \leq i \leq N_p$, the* maximal reach pointer *of the node h_i points to the deepest node v of $\mathsf{PPH}(\mathcal{T})$ such that v represents a prefix of $\mathsf{spe}(w_i)$.*

$\mathsf{spe}(w_1)$	$\underline{\varepsilon}$
$\mathsf{spe}(w_2)$	$\underline{\mathtt{x}}$
$\mathsf{spe}(w_3)$	$\underline{\mathtt{xx}}$
$\mathsf{spe}(w_4)$	$\underline{\mathtt{xy}}$
$\mathsf{spe}(w_5)$	$\underline{\mathtt{xxx}}$
$\mathsf{spe}(w_6)$	$\underline{\mathtt{axx}}$
$\mathsf{spe}(w_7)$	$\underline{\mathtt{xyz}}$
$\mathsf{spe}(w_8)$	$\underline{\mathtt{axy}}$
$\mathsf{spe}(w_9)$	$\underline{\mathtt{axxx}}$
$\mathsf{spe}(w_{10})$	$\underline{\mathtt{xaxx}}$
$\mathsf{spe}(w_{11})$	$\underline{\mathtt{xayy}}$
$\mathsf{spe}(w_{12})$	$\underline{\mathtt{axyz}}$
$\mathsf{spe}(w_{13})$	$\underline{\mathtt{xaxy}}$
$\mathsf{spe}(w_{14})$	$\underline{\mathtt{xaxxx}}$
$\mathsf{spe}(w_{15})$	$\underline{\mathtt{xaxyx}}$

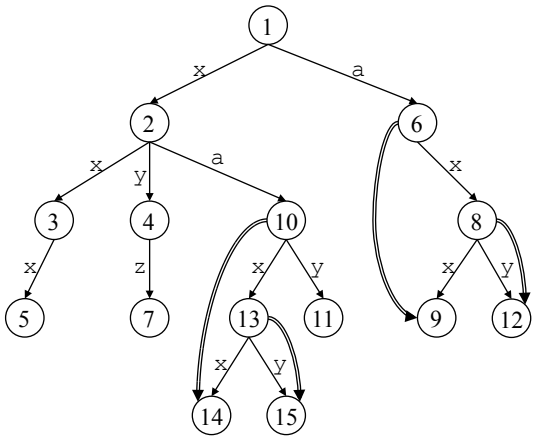

Figure 4.3: To the left is the list of $\mathsf{spe}(w_i)$ for p-strings represented by $\mathsf{pCST}(\mathcal{T})$ of Fig. 4.2, where $\Sigma = \{\mathtt{a}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. To the right is an illustration for augmented $\mathsf{PPH}(\mathcal{T})$ where the maximal reach pointers are indicated by the double-lined arrows. The underlined prefix of each $\mathsf{spe}(w_i)$ in the left list denotes the longest prefix of $\mathsf{spe}(w_i)$ that was represented in $\mathsf{PPH}(\mathcal{T})$ and hence, the maximal reach pointer of the node with label i points to the node which represents this underlined prefix of $\mathsf{spe}(w_i)$.

The node which is pointed by the maximal reach pointer of node h_i is denoted by $\mathsf{mrp}(i)$. The *augmented* $\mathsf{PPH}(\mathcal{T})$ is $\mathsf{PPH}(\mathcal{T})$ with the maximal reach pointers of all nodes. For simplicity, if $\mathsf{mrp}(i)$ is equal to h_i, then we omit this pointer. See Fig. 4.3 for an example of augmented $\mathsf{PPH}(\mathcal{T})$.

4.3.3 P-matching with augmented parameterized position heap

It is straightforward that by applying Diptarama et al.'s pattern matching algorithm to our $\mathsf{PPH}(\mathcal{T})$ augmented with maximal reach pointers, parameterized pattern matching can be done in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ')$ time where occ' is the number of nodes in $\mathsf{pCST}(\mathcal{T})$ that p-match with the pattern. Since each node in $\mathsf{pCST}(\mathcal{T})$ stores the pointers to the corresponding nodes in $\mathcal{T}$, then we can answer all the nodes that p-match with the pattern.

Diptarama et al.'s algorithm stands on Lemmas 13 and 14 of [16]. These lemmas can be extended to our $\mathsf{PPH}(\mathcal{T})$ as follows:

Lemma 10. *Suppose* $\mathsf{spe}(p)$ *is represented by a node* u *of augmented* $\mathsf{PPH}(\mathcal{T})$. *Then* p *p-matches with the prefix of length* $|p|$ *of* w_i *iff* $\mathsf{mrp}(i)$ *is* u *or a descendant of* u.

Lemma 11. *Suppose that* $\mathsf{spe}(p)$ *is not represented in augmented* $\mathsf{PPH}(\mathcal{T})$*. There is a factorization* $q_1, \ldots, q_k$ *of* p *s.t.* q_j *is the longest prefix of* $\mathsf{spe}(p[|q_1 \cdots q_{j-1}|+1 : |p|])$ *that is represented in augmented* $\mathsf{PPH}(\mathcal{T})$*. If* p *p-matches with the prefix of length* $|p|$ *of* w_i*, then* $\mathsf{mrp}(i+|q_1 \cdots q_{j-1}|)$ *is the node which represents* $\mathsf{spe}(q_j)$ *for any* $1 \leq j < k$ *and* $\mathsf{mrp}(i + |q_1 \cdots q_{k-1}|)$ *is the node which represents* $\mathsf{spe}(q_k)$ *or a descendant of* $\mathsf{mrp}(i + |q_1 \cdots q_{k-1}|)$*.*

Theorem 2. *Using our augmented* $\mathsf{PPH}(\mathcal{T})$*, one can perform parameterized pattern matching queries in* $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ *time.*

4.4 Construction of parameterized position heaps

In this section, we show how to construct the augmented $\mathsf{PPH}(\mathcal{T})$ of a given common suffix trie $\mathcal{T}$ of size N. For convenience, we will sometimes identify each node v of $\mathsf{PPH}(\mathcal{T})$ with the string which is represented by v. In Section 4.4.1, we show how to compute $\mathsf{pCST}(\mathcal{T})$ from a given common suffix trie $\mathcal{T}$. In Section 4.4.2, we propose how to construct $\mathsf{PPH}(\mathcal{T})$ from $\mathsf{pCST}(\mathcal{T})$.

4.4.1 Computing $\mathsf{pCST}(\mathcal{T})$ from $\mathcal{T}$

Here, we show how to construct $\mathsf{pCST}(\mathcal{T})$ of a given $\mathcal{T}$ of size N.

Lemma 12. *For any common suffix trie* $\mathcal{T}$ *of size* N*,* $\mathsf{pCST}(\mathcal{T})$ *can be computed in* $O(N|\Pi|)$ *time and space.*

Proof. We process every node of $\mathcal{T}$ in a breadth first manner. Let x_j be the p-string which is represented by j-the node of $\mathcal{T}$. Suppose that we have processed the first k nodes and have computed $\mathsf{pCST}(\mathcal{T})^i$ ($i \leq k$). We assume that the j-th node of $\mathcal{T}$, for any $1 \leq j \leq k$, holds the resulting substitutions from x_j to $\mathsf{spe}((x_j)^R)^R$ (i.e., $x_j[\alpha]$ is mapped to $\mathsf{spe}((x_j)^R)^R[\alpha]$), and also a pointer to the corresponding node of $\mathsf{pCST}(\mathcal{T})^i$ (i.e., pointer to the node representing $\mathsf{spe}((x_j)^R)^R$). We consider processing the $(k+1)$-th node of $\mathcal{T}$. Since x_{k+1} is encoded from right to left, we can determine a character $\mathsf{spe}((x_{k+1})^R)^R[1]$ in $O(|\Pi|)$ time. Then, we can insert a new node that represents $\mathsf{spe}((x_{k+1})^R)^R$ as a parent of the node which represents $\mathsf{spe}((x_{k+1}[2 : |x_{k+1}|])^R)^R$ if there does not exist such a node in $\mathsf{pCST}(\mathcal{T})^i$. Therefore, we can compute $\mathsf{pCST}(\mathcal{T})$ in $O(N|\Pi|)$ time and space. □

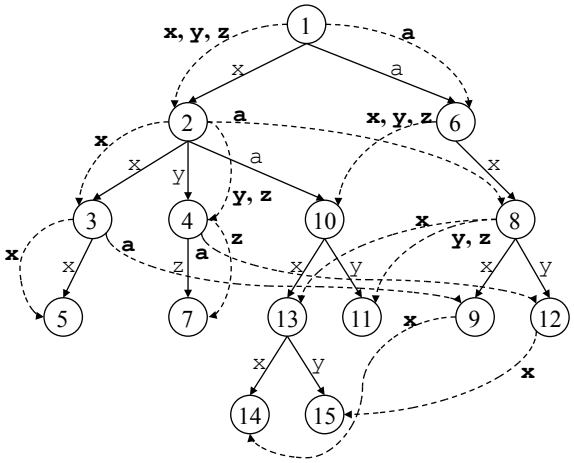

Figure 4.4: $\mathsf{PPH}(\mathcal{T})$ with all reversed suffix links is illustrated in this figure. Each dashed arrow shows a reversed suffix link. The label of a suffix link is drawn by a bold character.

4.4.2 Computing $\mathsf{PPH}(\mathcal{T})$ from $\mathsf{pCST}(\mathcal{T})$

For efficient construction of our $\mathsf{PPH}(\mathcal{T})$, we use *reversed suffix links* defined as follows.

Definition 11 (Reversed suffix links)**.** *For any node v of $\mathsf{PPH}(\mathcal{T})$ and a character $a \in \Sigma \cup \Pi$, let*

$$\mathsf{rslp}(a, v) = \begin{cases} \mathsf{spe}(av) & \textit{if } \mathsf{spe}(av) \textit{ is represented by } \mathsf{PPH}(\mathcal{T}), \\ \textit{undefined} & \textit{otherwise}. \end{cases}$$

See Fig. 4.4 for an example of $\mathsf{PPH}(\mathcal{T})$ with reversed suffix links. In our algorithm, firstly, we insert a new node h_i of $\mathsf{PPH}(\mathcal{T})^i$ to $\mathsf{PPH}(\mathcal{T})^{i-1}$. After that, we add new suffix links which point to h_i. When we have computed $\mathsf{PPH}(\mathcal{T})$, then we compute all maximal reach pointers of $\mathsf{PPH}(\mathcal{T})$.

Inserting a new node

Assume that c_j (i.e., j-th node of $\mathsf{pCST}(\mathcal{T})$) is the child of c_i for any $2 \leq i \leq N_p$. Consider to insert h_i (i.e., the node of $\mathsf{PPH}(\mathcal{T})$ which corresponds to c_i) to $\mathsf{PPH}(\mathcal{T})^{i-1}$. We show how to find the parent of h_i by starting from h_j. There are 3 cases based on $w_i[1]$ as follows:

- $w_i[1] \in \Pi$ and $w_i[1]$ appears in $w_j[1 : |h_j|]$ (Lemma 13),
- $w_i[1] \in \Pi$ and $w_i[1]$ does not appear in $w_j[1 : |h_j|]$ (Lemma 14),
- $w_i[1] \in \Sigma$ (Lemma 15).

Lemma 13. *Assume that* $w_i[1] \in \Pi$ *appears in* $w_j[1 : |h_j|]$*, and* a *is the character in* Π *such that* $a = \mathsf{spe}(w_j[1 : |h_j|])[\alpha]$ *and* $w_i[1] = w_j[\alpha]$ *for some* $1 \leq \alpha \leq |w_j[1 : |h_j|]|$*. Let* h_k *be the node of* $\mathsf{PPH}(\mathcal{T})^{i-1}$ *which is the lowest ancestor of* h_j *that has a reversed suffix link labeled with* a*. Then,* h_i *is a child of the node representing* $\mathsf{rslp}(a, h_k)$*.*

Proof. Let ℓ be the length of $\mathsf{rslp}(a, h_k)$. To prove this lemma, we show that

1. $\mathsf{rslp}(a, h_k) = \mathsf{spe}(w_i)[1 : \ell]$, and
2. There does not exist a node which represents $\mathsf{spe}(w_i)[1 : \ell + 1]$ in $\mathsf{PPH}(\mathcal{T})^{i-1}$.

By the definition of reversed suffix links and spe, we have

$$\begin{aligned} \mathsf{rslp}(a, h_k) &= \mathsf{spe}(a \cdot \mathsf{spe}(w_j[1 : \ell - 1])) = \mathsf{spe}(w_i[1] \cdot w_j[1 : \ell - 1]) \\ &= \mathsf{spe}(w_i[1] \cdot w_i[2 : \ell]) = \mathsf{spe}(w_i[1 : \ell]). \end{aligned}$$

Thus, we have proved the first statement.

By a similar argument, we also have $\mathsf{spe}(a \cdot \mathsf{spe}(w_j[1 : \ell])) = \mathsf{spe}(w_i[1 : \ell + 1])$. Thus, if $\mathsf{spe}(w_i)[1 : \ell + 1]$ is represented in $\mathsf{PPH}(\mathcal{T})^{i-1}$, the node representing $\mathsf{spe}(w_j[1 : \ell])$ must have a reversed suffix link labeled with a. This contradicts the fact that h_k is the lowest ancestor of h_j which has a reversed suffix link labeled with a. □

Lemma 14. *Assume that* $w_i[1] \in \Pi$ *does not appear in* $w_j[1 : |h_j|]$*. Let* h_k *be the node of* $\mathsf{PPH}(\mathcal{T})^{i-1}$ *which is the lowest ancestor of* h_j *that has a reversed suffix link labeled with* $a \in \Pi \setminus \{h_j[\alpha] \mid 1 \leq \alpha \leq |h_j|\}$*. Then,* h_i *is a child of the node representing* $\mathsf{rslp}(a, h_k)$*.*

Proof. Let ℓ be the length of $\mathsf{rslp}(a, h_k)$. We show similar statements to the proof of the previous lemma hold, but with different assumptions on $w_i[1]$ and a. By the definition of reversed suffix links and spe, we have

$$\mathsf{rslp}(a, h_k) = \mathsf{spe}(a \cdot \mathsf{spe}(w_j[1 : \ell - 1])) = \mathsf{spe}(a \cdot w_j[1 : \ell - 1]) = \mathsf{spe}(w_i[1 : \ell]).$$

Thus, we have proved the first statement.

By a similar argument, we also have $\mathsf{spe}(a \cdot \mathsf{spe}(w_j[1 : \ell])) = \mathsf{spe}(w_i[1 : \ell + 1])$. This implies that the second statement holds (similar to the proof of the previous lemma). □

Lemma 15. *Assume that $w_i[1] \in \Sigma$. Let h_k be the node in $\mathsf{PPH}(\mathcal{T})^{i-1}$ which is the lowest ancestor of h_j that has a reversed suffix link labeled with $w_i[1]$. Then, h_i is a child of the node representing $\mathsf{rslp}(w_i[1], h_k)$.*

Proof. Since $w_i[1] \in \Sigma$, we can show the lemma in a similar way to the above proofs. □

Inserting new reversed suffix links

In our algorithm, we will add reversed suffix links which point to h_i after inserting a new node h_i. The following lemma shows the number of nodes which point to h_i by reversed suffix links is at most one.

Lemma 16. *For any node v of $\mathsf{PPH}(\mathcal{T})$, the number of nodes which point to v by reversed suffix links is at most one.*

Proof. Let v_1, v_2 be nodes of $\mathsf{PPH}(\mathcal{T})$. Assume that $\mathsf{rslp}(a_1, v_1) = \mathsf{rslp}(a_2, v_2)$ for some $a_1, a_2 \in \Sigma \cup \Pi$ and $v_1 \neq v_2$ hold. By the definition of reversed suffix links, $\mathsf{spe}(a_1 \cdot v_1) = \mathsf{spe}(a_2 \cdot v_2)$. Namely, $a_1 \cdot v_1 \approx a_2 \cdot v_2$ holds. This implies that $v_1 \approx v_2$, i.e., $\mathsf{spe}(v_1) = \mathsf{spe}(v_2)$. Since v_1 and v_2 are node of $\mathsf{PPH}(\mathcal{T})$, $\mathsf{spe}(v_1) = v_1$ and $\mathsf{spe}(v_2) = v_2$ hold. This contradicts the fact that $v_1 \neq v_2$. □

By the above lemma and arguments of insertion, the node which points to the new node h_i by reversed suffix links is only a child of h_k which is an ancestor of h_j.

Construction algorithm

Finally, we explain our algorithm of constructing our position heap. From the above lemmas, we can use similar techniques to Nakashima et al. [39] which construct the position heap of a trie of normal strings. One main difference is the computation of the label of inserted edges/reversed suffix links. In so doing, each node h_α holds the resulting substitutions from $w_\alpha[1 : |h_\alpha|]$ to $\mathsf{spe}(w_\alpha[1 : |h_\alpha|])$. By using these substitutions, we can compute the corresponding label in $O(|\Pi|)$ time. Thus, we can insert new nodes and new suffix links in $O(|\Pi|)$ time for each node of $\mathsf{pCST}(\mathcal{T})$. In fact, since we need to use $(|\Sigma| + |\Pi|)$-copies of the position heap for nearest marked ancestor queries on each character, we use $O(|\Sigma| + |\Pi|)$ time to update the data structures needed for each node of $\mathsf{pCST}(\mathcal{T})$. Therefore, we have the following lemma.

Lemma 17. *We can compute $\mathsf{PPH}(\mathcal{T})$ from $\mathsf{pCST}(\mathcal{T})$ of size N_p in $O(N_p(|\Sigma| + |\Pi|))$ time and space.*

Therefore, we can obtain the following result by Lemmas 12 and 17.

Theorem 3. *We can compute* $\mathsf{PPH}(\mathcal{T})$ *of a given common suffix trie* $\mathcal{T}$ *of size* N *in* $O(N(|\Sigma| + |\Pi|))$ *time and space.*

Since we can also compute all maximal reach pointers of $\mathsf{PPH}(\mathcal{T})$ efficiently in a similar way to [39] (this algorithm is also similar to suffix link construction), we also have the following lemma.

Lemma 18. *We can compute all the maximal reach pointers for* $\mathsf{PPH}(\mathcal{T})$ *in* $O(N_p(|\Sigma| + |\Pi|))$ *time and space.*

Hence, we can get the following result.

Theorem 4. *We can compute the augmented* $\mathsf{PPH}(\mathcal{T})$ *of a given common suffix trie* $\mathcal{T}$ *of size* N *in* $O(N(|\Sigma| + |\Pi|))$ *time and space.*

4.5 Conclusions and open problems

This paper proposed the p-position heap for a CS trie $\mathcal{T}$, denoted $\mathsf{PPH}(\mathcal{T})$, which is the first indexing structure for the p-matching problem on a trie. The key idea is to transform the input CS trie $\mathcal{T}$ into a parameterized CS trie $\mathsf{pCST}(\mathcal{T})$ where p-matching suffixes are merged. We showed that the p-matching problem on the CS trie $\mathcal{T}$ can be reduced to the p-matching problem on the parameterized CS trie $\mathsf{pCST}(\mathcal{T})$. We proposed an algorithm which constructs $\mathsf{PPH}(\mathcal{T})$ in $O(N(|\Sigma|+|\Pi|))$ time and working space, where N is the size of the CS trie $\mathcal{T}$. We also showed that using $\mathsf{PPH}(\mathcal{P})$ one can solve the p-matching problem on the CS trie $\mathcal{T}$ in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time, where m is the length of a query pattern and occ is the number of occurrences to report.

Examples of open problems regarding this work are the following:

- Would it be possible to shave the $m|\Pi|$ term in the pattern matching time using p-position heaps? This $m|\Pi|$ term is introduced when the depth of the corresponding path of $\mathsf{PPH}(\mathcal{T})$ is shorter the pattern length m and thus the pattern needs to be partitioned into $O(|\Pi|)$ blocks in the current pattern matching algorithm [16].

- Can we efficiently build the p-suffix tree for a CS trie? It is noted by Baker [3, 4] that the *destination* of a parameterized suffix link (p-suffix link) of the p-suffix tree can be

an *implicit node* that lies on an edge, and hence there is no monotonicity in the chain of p-suffix links. If we follow the approach by Breslauer [8] which is based on Weiner's algorithm [45], then we need to use the reversed p-suffix link. It is, however, unclear whether one can adopt this approach since the *origin* of a reversed p-suffix link may be an implicit node. Recall that in each step of construction we need to find the nearest (implicit) ancestor that has a reversed p-suffix link labeled with a given character. Since there can be $\Theta(N^2)$ implicit nodes, we cannot afford to explicitly maintain information about the reversed p-suffix links for all implicit nodes.

Chapter 5

Direct Linear Time Construction of Parameterized Suffix and LCP Arrays for Constant Alphabets

5.1 Introduction

Deguchi et al. [15] proposed the parameterized suffix array (PSA). Given the PST of a string, the PSA can be constructed in linear time, but as in the case for standard strings, the direct construction of PSAs has been a topic of interest. Deguchi et al. [15] showed a linear time algorithm for the special case of $|\Pi| = 2$ and $\Sigma = \emptyset$. I et al. [25] proposed a lightweight and practically efficient algorithm for larger Π, but the worst-case time was still quadratic in n. Beal and Adjeroh [5] proposed an algorithm based on arithmetic coding that runs in $O(n)$ time on average. Furthermore, they claimed a worst-case running time of $o(n^2)$. However, the proved upperbound is $O(n^2(\frac{\log(n-\log^{1+\varepsilon} n)}{\log^{1+\varepsilon} n}))$ for a very small $\varepsilon > 0$ (Corollary 27 of [5]), so it is only slightly better than quadratic.

In this paper, we break the worst-case quadratic time barrier considerably, and present the first worst-case linear time algorithm for constructing the parameterized suffix and LCP arrays of a given p-string, when the number of distinct parameterized symbols in the string is constant. Namely, when assuming that Π and Σ are linearly sortable integer alphabets, our algorithm runs in $O(n|\Pi_t|)$ time and $O(n)$ words of space, where Π_t is the set of distinct symbols of Π in t. Furthermore, when $|\Pi|$ is constant and Σ is an integer alphabet, our algorithm runs in $O(n)$ time, which is faster than first building the PST which takes $O(n \log |\Sigma|)$ time. Since the PST can be constructed from the parameterized suffix and LCP arrays in linear time using essentially the

same algorithm for standard strings by Kasai et al. [28], our algorithm is the fastest algorithm for constructing PST in such cases.

5.2 Preliminaries

Let $\prec$ denote a total order on A, as well as the lexicographic order it induces. For two strings $x, y \in A^*$, $x \prec y$ if and only if x is a proper prefix of y, or there is some position $1 \leq k \leq \min\{|x|, |y|\}$ such that $x[1:k-1] = y[1:k-1]$ and $x[k] \prec y[k]$.

We assume that Π and Σ are disjoint integer alphabets, where $\Pi = \{0, \ldots, n^{c_1}\}$ for some constant $c_1 \geq 1$ and $\Sigma = \{n^{c_1}+1, \ldots, n^{c_2}\}$ for some constant $c_2 \geq 1$. This way, we can distinguish whether a symbol of a given prev encoding belongs to Σ or not. Also, given p-string x of length n, we can compute $\mathsf{prev}(x)$ in $O(n)$ time and space (in words), by sorting the pairs $(x[i], i)$ using radix sort, followed by a simple scan of the result.

The following are the data structures that we consider in this paper.

Definition 12 (Parameterized Suffix Array [15])**.** *The parameterized suffix array of a p-string x of length n, is an array $\mathsf{PSA}[1:n]$ of integers such that $\mathsf{PSA}[i] = j$ if and only if $\mathsf{prev}(x[j:n])$ is the ith lexicographically smallest string in $\{\mathsf{prev}(x[i:n]) \mid i = 1, \ldots, n\}$.*

Definition 13 (Parameterized LCP Array [15])**.** *The parameterized LCP array of a p-string x of length n, is an array $\mathsf{PLCP}[1:n]$ of integers such that $\mathsf{PLCP}[1] = 0$, and $\mathsf{PLCP}[i]$, for any $i \in \{2, \ldots, n\}$, is the longest common prefix between $\mathsf{prev}(x[\mathsf{PSA}[i-1]:n])$ and $\mathsf{prev}(x[\mathsf{PSA}[i]:n])$.*

The difficulty when dealing with the prev encoding of suffixes of a string, is that they are not necessarily the suffixes of the prev encoding of the string. It is important to notice however, that, given the prev encoding $\mathsf{prev}(x)$ of the whole string x, any specific value of the prev encoding of an arbitrary suffix of x can be retrieved in constant time, i.e., for any $1 \leq i \leq n$ and $1 \leq k \leq n - i + 1$,

$$\mathsf{prev}(x[i:n])[k] = \begin{cases} 0 & \text{if } x[k'] \in \Pi \text{ and } \mathsf{prev}(x)[k'] \geq k, \\ \mathsf{prev}(x)[k'] & \text{otherwise,} \end{cases}$$

where $k' = i + k - 1$. The critical problem for suffix sorting is that even if two prev encodings $\mathsf{prev}(x[i:n])$ and $\mathsf{prev}(x[j:n])$ share a common prefix and satisfies $\mathsf{prev}(x[i:n]) \prec \mathsf{prev}(x[j:n])$, it may still be that $\mathsf{prev}(x[j+1:n]) \prec \mathsf{prev}(x[i+1:n])$.

Fig. 5.1 shows an example of PSA and PLCP for the string `stssAtssAs`. For example, we have that $\mathsf{prev}(x[6 : 10]) \prec \mathsf{prev}(x[1 : 10])$, which share a common prefix of length 2, yet $\mathsf{prev}(x[2 : 10]) \prec \mathsf{prev}(x[7 : 10])$.

i	PSA$[i]$	prev$(x[\mathsf{PSA}[i] : n])$	PLCP$[i]$
1	10	0	0
2	6	0 0 1 A 2	1
3	2	0 0 1 A 4 3 1 A 2	4
4	1	0 0 2 1 A 4 3 1 A 2	2
5	3	0 1 A 0 3 1 A 2	1
6	7	0 1 A 2	3
7	4	0 A 0 3 1 A 2	1
8	8	0 A 2	2
9	9	A 0	0
10	5	A 0 0 1 A 2	2

Figure 5.1: An example of the parameterized suffix and LCP arrays for a p-string $x =$ `stssAtssAs`, where $\Sigma = \{\mathtt{A}\}, \Pi = \{\mathtt{s}, \mathtt{t}\}$.

5.3 Algorithms

In this section we describe our algorithms for constructing the parameterized suffix and LCP arrays. First, we mention a simple observation below.

From the definition of $\mathsf{prev}(t)$, we have that $\mathsf{prev}(t)[i] = 0$ for some position i if and only if i is the first occurrence of symbol $t[i] \in \Pi$. Therefore, the following observation can be made.

Observation 1. *For any p-string t, the prev encoding $\mathsf{prev}(t')$ of any substring t' of t contains at most $|\Pi_t|$ positions that are 0's, where Π_t is the set of distinct symbols of Π in t.*

5.3.1 PSA Construction

Based on this observation, we can see that the prev encoding of each suffix $t[i : n]$ can be partitioned into $z_i + 1 \leq |\Pi_t| + 1$ *blocks*, where z_i is the number of 0's in $\mathsf{prev}(t[i : n])$, and the jth block is the substring of $\mathsf{prev}(t[i : n])$ that ends at the jth 0 in $\mathsf{prev}(t[i : n])$ for $j = 1, \ldots, z_i$, and the (possibly empty) remaining suffix for $j = z_i + 1$. For technical reasons, we will append 0 to the last block as well. That is, we can write

$$\mathsf{prev}(t[i : n])0 = B_{i,1} \cdots B_{i,z_i+1} \tag{5.1}$$

where, $B_{i,j}$ denotes the jth block of $\mathsf{prev}(t[i:n])$. Also, for any $z_i + 1 < j \leq |\Pi_t| + 1$, we let $B_{i,j} = 0$. Furthermore, for each j, let B_j denote the set of all jth blocks for all $i = 1, \ldots, n$, and let $C_{i,j}$ denote the lexicographic rank of $B_{i,j}$ in B_j. Note that if $B_{i,j} = B_{k,j}$ for some i, k, then $C_{i,j} = C_{k,j}$. Finally, let C_i denote the string over the alphabet $\{1, \ldots, n\}$ obtained by renaming each block $B_{i,j}$ of the string $\mathsf{prev}(t[i:n])0$ with its lexicographic rank $C_{i,j}$. More formally,

$$\begin{aligned} B_j &= \{B_{i,j} \mid i = 1, \ldots, n\} \\ C_{i,j} &= |\{B_{i',j} \in B_j \mid B_{i',j} \prec B_{i,j}\}| + 1 \\ C_i &= C_{i,1} \cdots C_{i,z_i+1}. \end{aligned}$$

Fig. 5.2 shows an example.

i	$\mathsf{prev}(t[i:\lvert n\rvert])$	$B_{i,1}$	$B_{i,2}$	$B_{i,3}$	C_i
1	0 0 2 1 A 4 3 1 A 2	0	0	2 1 A 4 3 1 A 2 0	1 1 4
2	0 0 1 A 4 3 1 A 2	0	0	1 A 4 3 1 A 2 0	1 1 3
3	0 1 A 0 3 1 A 2	0	1 A 0	3 1 A 2 0	1 2 5
4	0 A 0 3 1 A 2	0	A 0	3 1 A 2 0	1 4 5
5	A 0 0 1 A 2	A 0	0	1 A 2 0	2 1 2
6	0 0 1 A 2	0	0	1 A 2 0	1 1 2
7	0 1 A 2	0	1 A 2 0	0	1 3
8	0 A 2	0	A 2 0	0	1 5
9	A 0	A 0	0	0	2 1
10	0	0	0	0	1 1

Figure 5.2: An example of C_i for a p-string $t = \mathtt{stssAtssAs}$, where $\Sigma = \{\mathtt{A}\}, \Pi = \{\mathtt{s}, \mathtt{t}\}$. $C_{4,2} = 4$, because $B_{4,2} = \mathtt{A0}$ is the lexicographically fourth smallest string in B_2, i.e., B_2 consists of 5 strings: $0 \prec \mathtt{1A0} \prec \mathtt{1A20} \prec \mathtt{A0} \prec \mathtt{A20}$.

Lemma 19. *For any* $1 \leq i_1, i_2 \leq n$,

$$\mathsf{prev}(t[i_1:n]) \prec \mathsf{prev}(t[i_2:n]) \iff C_{i_1} \prec C_{i_2}.$$

Proof. Notice that 0 is the smallest symbol in the two strings, so

$$\begin{aligned} \mathsf{prev}(t[i_1:n]) \prec \mathsf{prev}(t[i_2:n]) &\Leftrightarrow \mathsf{prev}(t[i_1:n])0 \prec \mathsf{prev}(t[i_2:n])0 \\ &\Leftrightarrow B_{i_1,1} \cdots B_{i_1,z_{i_1}+1} \prec B_{i_2,1} \cdots B_{i_2,z_{i_2}+1}. \end{aligned}$$

Also notice that since any block must end with a 0, if two blocks are not identical, it holds that

one cannot be a prefix of the other. Thus, if $B_{i_1,1} \cdots B_{i_1,z_{i_1}+1} \prec B_{i_2,1} \cdots B_{i_2,z_{i_2}+1}$, this implies that there is some block k such that $B_{i_1,j} = B_{i_2,j}$, for all $1 \leq j < k$, and $B_{i_1,k} \prec B_{i_2,k}$, where $B_{i_1,k}$ is not a prefix of $B_{i_2,k}$. By definition, $B_{i_1,k} \preceq B_{i_2,k} \Leftrightarrow C_{i_1,k} \leq C_{i_2,k}$. Therefore, we have, $B_{i_1,1} \cdots B_{i_1,z_{i_1}+1} \prec B_{i_2,1} \cdots B_{i_2,z_{i_2}+1} \Leftrightarrow C_{i_1} \prec C_{i_2}$. □

From Lemma 19, the problem of lexicographically sorting the set of strings $\{\mathsf{prev}(t[1 : n]), \ldots, \mathsf{prev}(t[n : n])\}$ reduces to the problem of lexicographically sorting the set of strings $\{C_1, \ldots, C_n\}$. The latter can be done in $O(n|\Pi_t|)$ time using radix sort, since the strings are over the alphabet $\{1, \ldots, n\}$ and the total length of the strings is at most $n|\Pi_t|$.

What remains is to compute $C_{i,j}$ for all i, j in the same time bound. A problem is that the total length of all $B_{i,j}$ is $\Theta(n^2)$, so we cannot afford to naively process all of them.

Denote by $b_{i,j}$ and $e_{i,j}$ the beginning and end positions of $B_{i,j}$ with respect to their (global) position in t. Note that for any $1 \leq i \leq n$, we have $b_{i,1} = i$, and $b_{i,j} = e_{i,j-1} + 1$ for all $2 \leq j \leq z_i + 1$. Our algorithm depends on the following simple yet crucial lemma.

Lemma 20. *For any $1 < i \leq n$ and $1 \leq j \leq z_i + 1$, we have that either*

1. *$b_{i,j} = e_{i-1,j} + 1$, or,*
2. *$b_{i,j} \geq b_{i-1,j}$, $e_{i,j} = e_{i-1,j}$, and $B_{i,j}$ is a suffix of $B_{i-1,j}$*

holds.

Proof. If $t[i-1] \in \Sigma$, then, $\mathsf{prev}(t[i : n])$ is a suffix of $\mathsf{prev}(t[i-1 : n])$, i.e., $\mathsf{prev}(t[i : n]) = \mathsf{prev}(t[i-1 : n])[2 : |n-i+2|]$ and $\mathsf{prev}(t[i-1 : n])[1] \neq 0$. Thus, $B_{i,1}$ is a suffix of $B_{i-1,1}$, and $B_{i,j} = B_{i-1,j}$ for all $2 \leq j \leq z_i$ and the second case of the claim holds.

If $t[i-1] \in \Pi$, the values in $\mathsf{prev}(t[i : n])$ are equivalent to the corresponding values of $\mathsf{prev}(t[i-1 : n])[2 : |n-i+2|]$, except possibly at some (global) position $k \geq i$ when there is a second occurrence of the symbol $t[i-1]$ at $t[k]$ which becomes the first occurrence in $t[i : n]$. (In other words, the value corresponding to $t[k]$ in $\mathsf{prev}(t[i-1 : n])$ is $k - i + 1$.) Since there is no previous occurrence of $t[i-1]$ in $t[i-1 : n]$, $\mathsf{prev}(t[i-1 : n])[1] = 0$. The situation is depicted in Fig. 20.

Let $B_{i-1,j'}$ be the block of $\mathsf{prev}(t[i-1 : n])$ that contains (global) position k. Because, as mentioned previously, $\mathsf{prev}(t[i] : n)$ and $\mathsf{prev}(t[i-1 : n])[2 : |n-i+2|]$ are equivalent except for the value corresponding to (global) position k, the block structure of $\mathsf{prev}(t[i-1 : n])$ is preserved in $\mathsf{prev}(t[i : n])$, except that (1) the first block $B_{i-1,1}$ disappears, and (2) the block

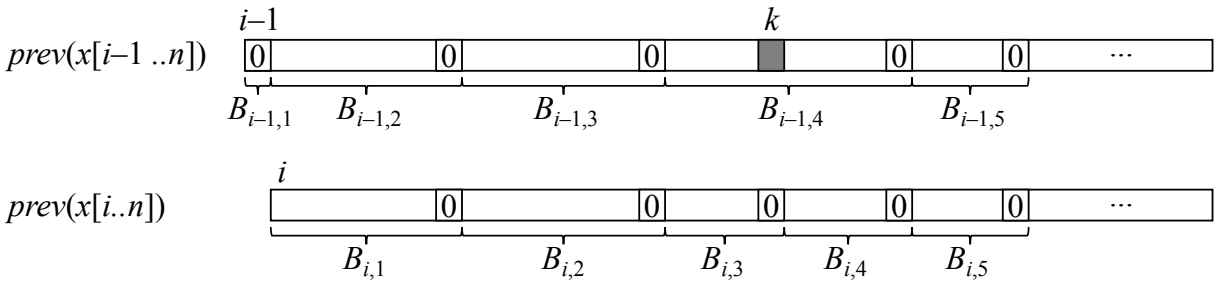

Figure 5.3: A case in the proof of Lemma 20, where $t[i] \in \Pi$, and $t[k]$ is the first occurrence of $t[i]$ in $t[i+1:n]$. The value corresponding to (global) position k in $\mathsf{prev}(t[i:n])$, shown as a shaded box, is $k-i$, while it is 0 in $\mathsf{prev}(t[i+1:n])$. All other values in $\mathsf{prev}(t[i:n])$ and $\mathsf{prev}(t[i+1:n])$ at the same (global) position are equivalent.

$B_{i-1,j'}$ is split into two blocks, corresponding to $B_{i,j'-1}$ and $B_{i,j'}$. Therefore, the first case of the claim is satisfied for $1 \leq j \leq j'$, since $b_{i,j} = b_{i-1,j+1} = e_{i-1,j} + 1$ for any $1 \leq j < j'$. Also, we can see that the second case of the claim is satisfied for $j' \leq j \leq z_i$, since $B_{i,j'}$ is a suffix of $B_{i-1,j'}$, and $B_{i,j} = B_{i-1,j}$ for $j' < j \leq z_i$.

Finally, the case when such k does not exist can be considered to be included above by simply assuming we are looking at a prefix of a longer string and $k > |t|, j' > z_i$, since the prev encoding is preserved for prefixes, i.e., the prev encoding of a prefix of any p-string y is equivalent to the corresponding prefix of the prev encoding of y. Thus, the lemma holds. □

Lemma 20 implies that if we fix some j, we can represent $B_{i,j}$ for all i as suffixes (in the standard sense) of strings of total length $O(n)$.

Corollary 2. *For any j, there exists a set of strings S_j with total length $n+1$ over the alphabet $\Sigma \cup \{0, \ldots, n-1\}$ such that $B_{i,j}$ is a suffix of some string in S_j for all $i \in \{1, \ldots, n\}$.*

Proof. We include $B_{i,j}$ in S_j, if $i = 1$, or, if $i > 1$ and $B_{i,j}$ satisfies the first case of Lemma 20. Since the first case implies that the (global) positions $[b_{i-1,j} : e_{i-1,j}]$ and $[b_{i,j} : e_{i,j}]$ are disjoint, the total length of strings in S_j is at most $n+1$ (including the 0 appended to B_{i,z_i+1}). On the other hand, if $B_{i,j}$ satisfies the second case is, it is a suffix of an already included string. □

Thus, computing $C_{i,j}$ for all i can be done by computing the generalized suffix array for the set S_j. This can be done in $O(n)$ time given S_j [29, 31, 27] and thus, for all j, the total is $O(n|\Pi_t|)$ time.

Theorem 5. *The parameterized suffix array of a p-string of length n can be computed in $O(n|\Pi_t|)$ time and $O(n)$ words of space.*

Proof. We compute a *forward encoding* of t, analogous to the prev encoding, defined as follows

$$fwd(t)[i] = \begin{cases} t[i] & \text{if } t[i] \in \Sigma, \\ \infty & \text{if } t[i] \in \Pi \text{ and } t[i] \neq t[j] \text{ for any } i < j \leq n, \\ j - i & \text{if } t[i] \in \Pi,\ t[i] = t[j] \text{ and } t[i] \neq t[k] \text{ for any } i < k < j. \end{cases}$$

This is done once, and can be computed in $O(n)$ time. Next, for any fixed j, we show how to compute the set S_j in linear time. This is done by using fwd and Lemma 20. We can first scan $\mathsf{prev}(t)$ to obtain $B_{1,j}$. Suppose for some $i \geq 2$, we know the beginning and end positions $b_{i-1,j}$, $e_{i-1,j}$ of $B_{i-1,j}$. Notice that when $t[i-1] \in \Pi$, k in the proof of Lemma 20 is $i+fwd(t)[i-1]-2$. Based on this value, we know that if $k < b_{i-1,j}$, then $B_{i,j} = B_{i-1,j}$ and if $b_{i-1,j} \leq k \leq e_{i-1,j}$ $B_{i,j}$ is a suffix of $B_{i-1,j}$, which corresponds to the second case of Lemma 20. When $k > e_{i-1,j}$, this corresponds to the first case of Lemma 20, so we scan $\mathsf{prev}(t[i:n])$ starting from position corresponding to the global position $b_{i,j} = e_{i-1,j} + 1$ (i.e., $e_{i-1,j} - i$ in $\mathsf{prev}(t[i:n])$) until we find the first 0, which gives us $B_{i,j}$ which we include in S_j. Since we only scan each position once, the total time for computing S_j is $O(n)$.

The time complexity follows from arguments for sorting C_j based on radix sort. Since, for a single step of the radix sort, we only require the values $C_{i,j}$ for a fixed j and all $1 \leq i \leq n$ and from Corollary 2, the space complexity is $O(n)$ words. □

5.3.2 PLCP Construction

Given PSA, we describe below how to construct PLCP in $O(n|\Pi_t|)$ time and $O(n)$ words of space.

As a tool, we use longest common extension (LCE) queries. For a string t of length n, a *longest extension query*, given positions $1 \leq i, j \leq n$ asks for the longest common prefix between $t[i:n]$ and $t[j:n]$. It is known that the string can be pre-processed in $O(n)$ time so that the longest extension query can be answered in $O(1)$ time for any i, j (e.g. [19]).

We recompute S_j for $j = 1, \ldots, |\Pi_t|$, and each time process it for LCE queries, so that the longest common prefix between $B_{i_1,j}$ and $B_{i_2,j}$ for some $1 \leq i_1, i_2 \leq n$ can be computed in constant time. This can be done in time linear in the total length of S_j, so in $O(n|\Pi_t|)$ total time for all j. We compute the longest common prefix between each adjacent suffix in PSA block by block. Since each block takes constant time, and there are $O(|\Pi_t|)$ blocks for each suffix, the total is $O(n|\Pi_t|)$ time for all entries of the PLCP array. The space complexity is $O(n)$ words,

since, as for the case of PSA construction, we only process the jth block at each step.

A note on a previous algorithm

A linear time algorithm for computing PLCP given PSA, is claimed in [5, Theorem 20]. However, we note that their proof of correctness seems to be incomplete, and we could not verify the correctness of their algorithm.

Their algorithm follows the idea for computing the Longest Previous Factor (LPF) array by Crochemore and Ilie [14], where, for all $1 \leq i \leq n$,

$$LPF[i] = \max\{l \geq 0 \mid t[i : i + l - 1] = t[j : j + l - 1], 1 \leq j < i\}.$$

Their algorithm computes two arrays:

$$\begin{aligned} X &= \max\{l \geq 0 \mid t[i : i + l - 1] \approx t[j : j + l - 1], 1 \leq j < i, \mathsf{PSA}^{-1}[j] < \mathsf{PSA}^{-1}[i]\}, \\ Y &= \max\{l \geq 0 \mid t[i : i + l - 1] \approx t[j : j + l - 1], i < j \leq n, \mathsf{PSA}^{-1}[j] < \mathsf{PSA}^{-1}[i]\}, \end{aligned}$$

where $\mathsf{PSA}^{-1}[\mathsf{PSA}[i]] = i$ for any $1 \leq i \leq n$. Then, $\mathsf{PLCP}[i] = \max\{X[\mathsf{PSA}^{-1}[i]], Y[\mathsf{PSA}^{-1}[i]]\}$. Here, X can also be denoted as $pLPF_<$, i.e., the longest prefix of the suffix $t[i : n]$ that occurs (in the parameterized sense) before i, and is a prefix of a lexicographically smaller suffix. Although the rationale for their algorithm is claimed to be ([5, Lemma 15])

$$pLPF[i] \geq pLPF[i - 1] - 1,$$

which holds, it seems that the required property actually is

$$pLPF_<[i] \geq pLPF_<[i - 1] - 1$$

which does not hold. For example, in the example `stssAtssAs` of Fig. 5.1, $pLPF_<[9] = 0$, while $pLPF_<[8] = 2$.

Chapter 6

Offline Construction of Parameterized DAWG

6.1 Introduction

The *parameterized suffix tree* (*p-suffix tree*) [3] is the fundamental indexing structure for p-matching, which supports p-matching queries in $O(m \log(|\Pi| + |\Sigma|) + occ)$ time, where m is the length of pattern P, and occ is the number of occurrences to report. It is known that the p-suffix tree of a text w of length n can be built in $O(n \log(|\Pi| + |\Sigma|))$ time with $O(n)$ space in an offline manner [32] and in a *left-to-right online* manner [44]. A *randomized* $O(n)$-time left-to-right online construction algorithm for p-suffix trees is also known [35]. Indexing p-strings has recently attracted much attention, and the p-matching versions of other indexing structures, such as *parameterized suffix arrays* [15, 25, 5, 21], *parameterized BWTs* [23], and *parameterized position heaps* [16, 20, 22], have also been proposed.

This paper fills in the missing pieces of indexing structures for p-matching, by proposing the parameterized version of the *directed acyclic word graphs* (*DAWGs*) [6, 13], which we call the *parameterized directed acyclic word graphs* (*PDAWGs*).

We propose an alternative *offline* algorithm which builds the PDAWG in $O(n)$ time with $O(n)$ working space, provided that the p-suffix tree of the reversal of the input string is given. While the proposed offline algorithm itself works in the pointer machine, there are two different complexities for p-suffix tree construction in the pointer machine and in the word RAM: The p-suffix tree can be built offline, in $O(n \log(|\Pi| + |\Sigma|))$ time and $O(n)$ space in the pointer machine [1] and $O(n|\Pi|)$ time and $O(n)$ space in the word RAM with word size $\Omega(\log n)$ [21]. Putting these together, we obtain $O(n \min\{\log(|\Pi| + |\Sigma|), |\Pi|\})$-time $O(n)$-space offline con-

struction of the PDAWG in the word RAM.

6.2 Preliminaries

We first introduce definitions and notation used in this paper and then briefly review indexing data structures of parameterized strings. In this section, we call a string $x \in (\Sigma \cup \mathcal{N})^*$ a *pv-string* if $x = \mathsf{prev}(S)$ for some p-string S.

Let $w, x, y \in (\Sigma \cup \mathcal{N})^*$. The set of the *end positions* or *p-occurrences* of x in a pv-string w is defined by $\mathsf{RPos}_w(x) = \{\, i \in \{0, \ldots, |w|\} \mid x = \mathsf{prev}(w[i - |x| + 1 : i]) \,\}$. We say x *occurs* in w at i if $i \in \mathsf{RPos}_w(x)$. Note that $0 \in \mathsf{RPos}_w(x)$ iff $x = \varepsilon$. We write $x \equiv^{\mathrm{R}}_w y$ iff $\mathsf{RPos}_w(x) = \mathsf{RPos}_w(y)$ and the equivalence class of x under $\equiv^{\mathrm{R}}_w$ as $[x]^{\mathrm{R}}_w$. Note that for any $x \notin \mathsf{PFactor}(w)$, which can be a non-pv-string, $\mathsf{RPos}_w(x) = \emptyset$. For an equivalent class $u = [x]^{\mathrm{R}}_w$, we may write $\mathsf{RPos}_w(u)$ to mean $\mathsf{RPos}_w(x)$ for $x \in u$. For a finite nonempty set X of strings which has no distinct elements of equal length, the shortest and longest elements of X are denoted by $\lfloor X \rfloor$ and $\lceil X \rceil$, respectively.

Example 1. *Let* $w = \mathsf{prev}(\mathtt{xaxaya}) = 0\mathtt{a}2\mathtt{a}0\mathtt{a}$ *where* $\Sigma = \{\mathtt{a}\}$ *and* $\Pi = \{\mathtt{x}, \mathtt{y}\}$. *Then,* $\mathsf{RPos}_w(\mathtt{a}) = \mathsf{RPos}_w(0\mathtt{a}) = \{2, 4, 6\}$ *and* $[\mathtt{a}]^{\mathrm{R}}_w = \{\mathtt{a}, 0\mathtt{a}\}$. *On the other hand,* $\mathsf{RPos}_w(2\mathtt{a}) = \emptyset$ *and* $[2\mathtt{a}]^{\mathrm{R}}_w = (\Sigma \cup \mathcal{N})^* \setminus \mathsf{PFactor}(w)$.

The *parameterized pattern matching problem* is to enumerate all the end positions of $\mathsf{prev}(p)$ in $\mathsf{prev}(t)$, i.e., elements of $\mathsf{RPos}_{\mathsf{prev}(t)}(\mathsf{prev}(p))$, for given two p-strings t and p. The weaker version of the problem is to decide whether $\mathsf{RPos}_{\mathsf{prev}(t)}(\mathsf{prev}(p)) \neq \emptyset$.

The basic properties on end positions of factors in static strings presented in [6] also hold for pv-strings.

Lemma 21. *Let* x, y *and* w *be pv-strings. If* $\mathsf{RPos}_w(x) \cap \mathsf{RPos}_w(y) \neq \emptyset$, *then either* $x \in \mathsf{PSuffix}(y)$ *or* $y \in \mathsf{PSuffix}(x)$. *If* $x \in \mathsf{PSuffix}(y)$, *then* $\mathsf{RPos}_w(y) \subseteq \mathsf{RPos}_w(x)$. *For any* $x \in \mathsf{PFactor}(w)$, *there is* $k \in \mathbb{N}$ *such that* $[x]^{\mathrm{R}}_w = \{\, \mathsf{prev}(y[i :]) \mid 1 \leq i \leq k \,\}$ *where* $y = \lceil [x]^{\mathrm{R}}_w \rceil$.

We will use the above lemma implicitly in arguments in this paper.

We define notions symmetric to RPos, $\equiv^{\mathrm{R}}$, and $[\cdot]^{\mathrm{R}}$. The start position set of x in a pv-string w is $\mathsf{LPos}_w(x) = \{\, i \in \{0, \ldots, |w|\} \mid \mathsf{prev}(w[i : i + |x| - 1]) = x \,\}$. We write $x \equiv^{\mathrm{L}}_w y$ iff $\mathsf{LPos}_w(x) = \mathsf{LPos}_w(y)$. The equivalence class of x under $\equiv^{\mathrm{L}}_w$ is denoted by $[x]^{\mathrm{L}}_w$.

6.3 Parameterized directed acyclic word graphs

In this subsection, we present a new indexing structure for parameterized strings, which we call *parameterized directed acyclic word graphs* (*PDAWGs*). In this subsection, we fix a text t and its pv-encoding $w = \mathsf{prev}(t)$.

Definition 2 (Parameterized directed acyclic word graphs)**.** *The* parameterized directed acyclic word graph *(*PDAWG*)* $\mathsf{PDAWG}(t)$ *of* t *is a triple* (V, E, F) *where*

$$V = \{\, [x]^{\mathrm{R}}_w \mid x \in \mathsf{PFactor}(w) \,\},$$
$$E = \{\, ([x]^{\mathrm{R}}_w, a, [xa]^{\mathrm{R}}_w) \in V \times (\Sigma \cup \mathcal{N}) \times V \mid x = \lceil [x]^{\mathrm{R}}_w \rceil \text{ and } xa \in \mathsf{PFactor}(w) \,\},$$
$$F = \{\, (u, v) \in (V \setminus \{\{\varepsilon\}\}) \times V \mid v = [\mathsf{prev}(x[2:])]^{\mathrm{R}}_{)} w \text{ for } x = \lfloor u \rfloor \,\}.$$

Elements of E *are called edges and those of* F *are suffix links.*

Note that (V, E) is a directed acyclic graph and $(V, \overline{F})$ is a tree. Since each $u \in V \setminus \{[\varepsilon]^{\mathrm{R}}_w\}$ has unique v such that $(u, v) \in F$, we often write $F(u)$ for v regarding F as a function.

6.4 Duality of PDAWGs and p-suffix trees

This section establishes the duality between parameterized suffix trees [3] and PDAWGs. This will give us the following two merits: the *first* bidirectional index for parametrized pattern matching (Section 6.4.1), and efficient offline construction of PDAWGs (Section 6.4.3).

6.4.1 Parameterized suffix trees and Weiner links

In this subsection, we first recall the basic properties of p-suffix trees (Section 6.4.1), and then the suffix links of p-suffix trees (Section 6.4.1). Then, we introduce our Weiner links of p-suffix trees (Section 6.4.1), and show that the Weiner links are equal to the PDAWG edges (Section 6.4.1). This immediately leads us to bidirectional indexing structure for p-matching.

Basics of p-suffix trees

Let t be a p-string and consider its reversal t^R. The *parameterized suffix tree (p-suffix tree)* $\mathsf{PSTree}(t^R)$ of t^R is the path-compacted (or Patricia) tree for $\mathsf{PSuffix}(t^R)$. Below let us recall the

definition of $\mathsf{PSTree}(t^R)$, which is the edge-labeled tree (V, E) of t^R such that for $w = \mathsf{prev}(t^R)$

$$V = \{ \lceil [x]^{\mathrm{L}}_w \rceil \mid x \in \mathsf{PFactor}(t^R) \},$$
$$E = \{ (x, y, xy) \in V \times (\Sigma \cup \mathcal{N})^+ \times V \mid xy = \lceil [xa]^{\mathrm{L}}_w \rceil \in V \text{ for some } a \in \Sigma \cup \mathcal{N} \}.$$

Recall that each edge of $\mathsf{PSTree}(t)$ is labeled by an element of $\mathsf{Factor}(\mathsf{PSuffix}(t)) \setminus \{\varepsilon\}$. An example for $\mathsf{PSTree}(t^R)$ is given in Figure 6.1(a).

Remark 1. *Since each node of* $\mathsf{PSTree}(t^R)$ *is defined as the longest member* $\lceil [x]^{\mathrm{L}}_w \rceil$ *of the equivalence class* $[x]^{\mathrm{L}}_w$, $\mathsf{PSTree}(t^R)$ *may contain an internal node that has only a single child. Such an internal node corresponds to a suffix of* t^R *that has internal p-matching occurrences in* t^R. *For instance,* $\mathtt{a0}$ *of the parameterized suffix tree shown in Figure 6.1(a) has only a single child. This is because* $\mathtt{a0}$ *has three p-matching occurrences in* $\mathtt{baxayay}$ *at positions* $2, 4, 6$, *where the last occurrence corresponds to the suffix* $\mathtt{ay}$ *and all of the other internal p-matching occurrences of* $\mathtt{a0}$ *are immediately followed by* $\mathtt{a}$. *If we use a common convention that* t^R *terminates with a unique character* \$, *all internal nodes of* $\mathsf{PSTree}(t^R)$ *become branching.*

Recall that each node of $\mathsf{PSTree}(t^R)$ is a pv-string. The *string depth* of a node x of $\mathsf{PSTree}(t^R)$ is $|x|$. We store the string depth $|x|$ in each node x of $\mathsf{PSTree}(t^R)$. To represent $\mathsf{PSTree}(t^R)$ in linear space, in an actual implementation, the label y of an edge (x, y, xy) is represented by two integers i and j such that $y = \mathsf{prev}(t^R[i - |x| : j])[|x| + 1 : j - i + |x| + 1]$. In other words, y is the length-$(j - i + 1)$ suffix of the pv-encoding of $\langle t^R[i - |x| : j] \rangle = xy$ which labels the path from the root to the node xy.

Suffix links of p-suffix trees

Let us recall the *suffix links* of the p-suffix tree, which were first introduced by Baker [3]. Let u be a non-root node of $\mathsf{PSTree}(t^R)$ and let S be any substring of t^R such that $\mathsf{prev}(S) = u$. The suffix link of u, denoted $\mathsf{sl}(u)$, is a pointer from u to $v = \mathsf{prev}(S[2 : |S|])$. Notice that v may not be a node of the p-suffix tree. For instance, see Figure 6.1(a). Consider node $u = 0\mathtt{a}$, in which we have $S = \mathtt{ya}$ or $S = \mathtt{xa}$. In either case $v = \mathsf{prev}(S[2 : 2]) = \mathsf{prev}(a) = \mathtt{a}$, however, there is no node representing $\mathtt{a}$. In this paper, we define the suffix link of a node $u = \mathsf{prev}(S)$ only if $v = \mathsf{prev}(S[2 : |S|])$ is a node.

We can characterize the target node v of the suffix link $\mathsf{sl}(u)$ depending on the first character $u[1]$ of u, as follows:

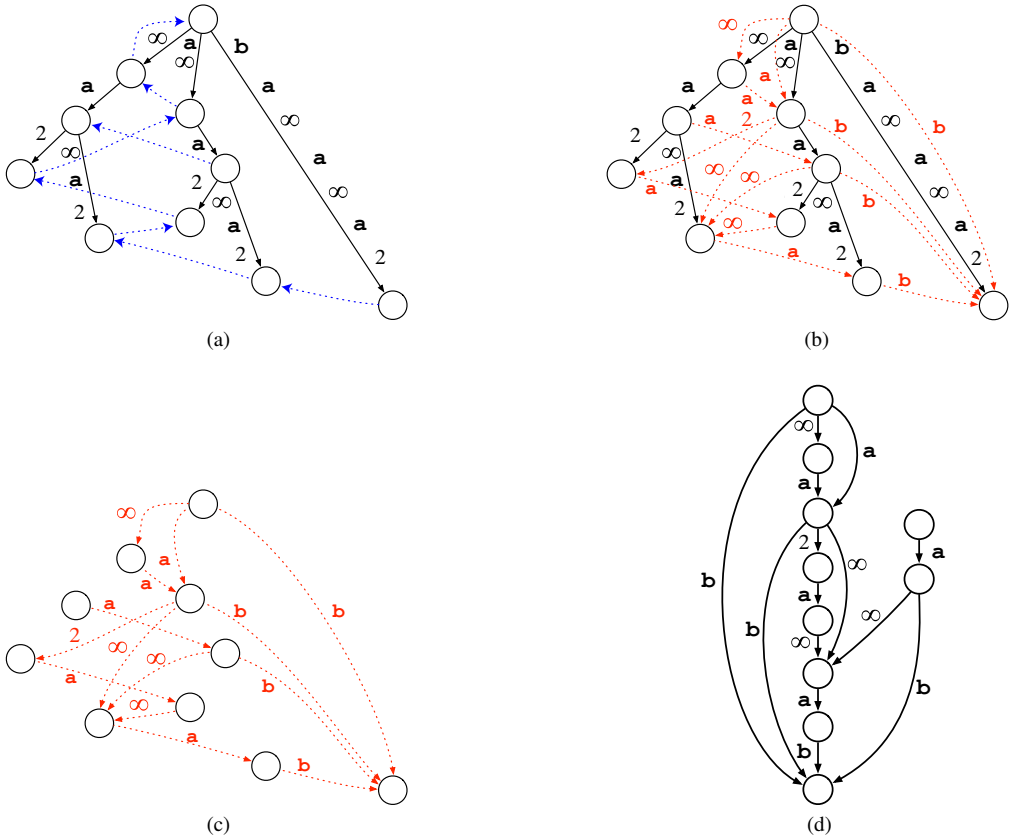

Figure 6.1: Consider string $t = \texttt{yayaxab}$ over $\Sigma = \{\texttt{a}, \texttt{b}\}$ and $\Pi = \{\texttt{x}, \texttt{y}\}$. (a) The parameterized suffix tree $\mathsf{PSTree}(t^R)$ with $t^R = \texttt{baxayay}$ augmented with the suffix links (dashed blue arcs). (b) The parameterized suffix tree $\mathsf{PSTree}(t^R)$ augmented with the Weiner links (dashed red arcs). (c) The DAG consisting of the p-suffix tree nodes and the Weiner-links. (d) The PDAWG $\mathsf{PDAWG}(t)$ with $t = \texttt{yayaxab}$. Observe that the edge-labeled graphs (c) and (d) are isomorphic.

(i) If $u[1] \in \Sigma$, then $v = u[2 : |u|]$. This is the same as the suffix link of a standard suffix tree for exact matching.

(ii) If $u[1] = 0$ and there exists a position a in u such that $u[a] = a - 1$, then $u[a]$ "points" to the first position of u in the prev encoding. Notice that such a position a is unique and that a is the smallest position in S that is larger than 1 with $S[1] = S[a]$. Now, the target node v is $u[2 : a - 1] \cdot 0 \cdot u[a + 1 : |u|]$, that is obtained by removing $u[1]$ from u and replacing $u[a]$ with 0.

(iii) If $u[1] = 0$ and there is no position a in u such that $u[a] = a - 1$, then $v = u[2 : |u|]$.

For examples of suffix links, see Figure 6.1(a). The suffix link of node $\texttt{a0a}$ points to node $0\texttt{a}$, which is Case (i). The suffix link of node $0\texttt{a2}$ points to node $\texttt{a0}$, which is Case (ii). The suffix

link of node $0\mathtt{a}0\mathtt{a}2$ points to node $\mathtt{a}0\mathtt{a}2$, which is Case (iii).

Weiner links of p-suffix trees

We enhance p-suffix trees by introducing *Weiner links*, which are key tools in this whole section. The *reversals* of the suffix links of Section 6.4.1 are explicit Weiner links. In Case (i), the corresponding explicit Weiner link is labeled with the static character $u[1] \in \Sigma$. In Case (iii), the corresponding explicit Weiner link is labeled with $u[1] = 0$. In Case (ii), we label the corresponding explicit Weiner link with the position a. In a unified manner for all these three cases, we can represent each explicit Weiner link by a triple (v, a, u), where $a \in \Sigma \cup \mathcal{N}$ and $|u| = |v| + 1$.

Now, we are to extend the notion of Weiner links to the case where there is no node u with $|u| = |v| + 1$ for a given node v and $a \in \Sigma \cup \mathcal{N}$. In such a case, we use the shortest (i.e. shallowest) possible node as u, as follows. Let v be a node in $\mathsf{PSTree}(t^R)$ such that $v = \mathsf{prev}(S)$ for some substring S of t^R, and $a \in \Sigma \cup \mathcal{N}$. Let $\alpha(a, v)$ be the pv-string such that

$$\alpha(a,v) = \begin{cases} av & \text{if } a \in \Sigma \cup \{0\} \text{ and } av \in \mathsf{PFactor}(t), \\ \mathsf{prev}(S[a] \cdot S) & \text{if } a \in \mathcal{N} \setminus \{0\} \text{ and } \mathsf{prev}(S[a] \cdot S) \in \mathsf{PFactor}(t), \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{6.1}$$

This function α corresponds to the reversals of the suffix links. When $a \in \Sigma$, it "prepends" label a to string (node) v. When $a \in \mathcal{N}$, it gives the pv-encoding of the p-string which is obtained by prepending the parameter character indicated by a to the p-string S whose pv-encoding is v. For $a = 0$, the indicated parameter character is a fresh one that occurs nowhere in S. For $a \in \mathcal{N} \setminus \{0\}$, the parameter character is $S[a]$. Then, the Weiner link from node v to node $u = \lceil [\alpha(a,v)]^{\mathrm{L}}_{\mathsf{prev}(t^R)} \rceil$ is labeled with a, which is represented by the triple (v, a, u). Note that the operator $\lceil [\cdot]^{\mathrm{L}}_{\mathsf{prev}(t^R)} \rceil$ brings us to the shallowest node u from the locus for $\alpha(a, v)$ in $\mathsf{PSTree}(t^R)$. Thus, each Weiner link increases the string depth by at least one, and hence $|u| \geq |v| + 1$ always holds.

The Weiner link (v, a, u) is said to be *explicit* if $|u| = v + 1$ (or equivalently $u = \alpha(a, v)$), and *implicit* if $|u| > |v|+1$. Namely, $u = \alpha(a, v)$ if and only if v is obtained by simply removing the first character a from $u = av$ (the first case in equation (6.1)), or by replacing $u[a + 1] = a$ with 0 and removing the first character 0 from u (the second case in equation (6.1)). Basically the same arguments hold for implicit Weiner links, except in that we need to cut off the suffix

of u to adjust the length to $|v| + 1$.

For examples of Weiner links of a p-suffix tree, see Figure 6.1(b). The node $v = \mathtt{a0}$ has an explicit Weiner link which is labeled 2 and points to the node $u = 0\mathtt{a}2$. This is because by replacing $u[2 + 1] = 2$ with 0 and by removing the first character 0 from u, we obtain $v = \mathtt{a0}$. This corresponds to the second case in equation (6.1). For another example, consider the node $u' = \mathtt{0a0a}2$ which has three in-coming Weiner links all labeled 0. The Weiner link from the node $v' = \mathtt{a0a}2$ is explicit, while the other two from the node $v'' = \mathtt{a0a}$ and $v''' = \mathtt{a0}$ are implicit. Note that all these Weiner links correspond to the first case in equation (6.1), namely, one can obtain v' by simply removing the first 0 from u, and can obtain v'' and v''' by removing the first 0 from u and removing the suffixes of u accordingly.

Duality between PDAWGs and p-suffix trees and bidirectional searches

To establish the correspondence between $\mathsf{PDAWG}(t)$ and $\mathsf{PSTree}(t^R)$, we define the *p-reverse* $\widetilde{x}$ of a pv-string x so that $\widetilde{x} = \mathsf{prev}(S^R)$ iff $x = \mathsf{prev}(S)$ for any p-string $S \in (\Sigma \cup \Pi)^*$. For example, for $t = \mathtt{xaxy}$ with $\mathtt{a} \in \Sigma$ and $\mathtt{x}, \mathtt{y} \in \Pi$, we have $\widetilde{\mathsf{prev}(t)} = \widetilde{0\mathtt{a}20} = 00\mathtt{a}2 = \mathsf{prev}(\mathtt{yxax}) = \mathsf{prev}(t^R)$.

For technical convenience, we rename the nodes $[x]^{\mathrm{R}}_{\mathsf{prev}(t)}$ of $\mathsf{PDAWG}(t)$ to be $\lceil [x]^{\mathrm{R}}_{\mathsf{prev}(t)} \rceil$ in this section. Moreover, we call an edge (x, a, y) of $\mathsf{PDAWG}(t)$ *primary* if $y = xa$, and *secondary* otherwise.

Theorem 6. *The following correspondence between* $\mathsf{PDAWG}(t) = (V_{\mathsf{D}}, E_{\mathsf{D}})$ *and* $\mathsf{PSTree}(t^R) = (V_{\mathsf{t}}, E_{\mathsf{t}})$ *holds.*

(1) $\mathsf{PDAWG}(t)$ *has a node* $x \in V_{\mathsf{D}}$ *iff* $\mathsf{PSTree}(t^R)$ *has a node* $\widetilde{x} \in V_{\mathsf{t}}$.

(2) $\mathsf{PDAWG}(t)$ *has a primary edge* $(x, a, y) \in E_{\mathsf{D}}$ *iff* $\mathsf{PSTree}(t^R)$ *has an explicit Weiner link* $(\widetilde{x}, a, \widetilde{y})$.

(3) $\mathsf{PDAWG}(t)$ *has a secondary edge* $(x, a, y) \in E_{\mathsf{D}}$ *iff* $\mathsf{PSTree}(t^R)$ *has an implicit Weiner link* $(\widetilde{x}, a, \widetilde{y})$.

(4) $\mathsf{PDAWG}(t)$ *has a suffix link from* xy *to* y *iff* $\mathsf{PSTree}(t^R)$ *has an edge* $(\widetilde{y}, \widetilde{x}, \widetilde{xy}) \in E_{\mathsf{t}}$.

Proof. To make the arguments simpler, we assume for now that t begins with a unique character \$ that does not occur elsewhere in t. The case without \$ can be shown similarly.

(1) By the symmetry $\mathsf{RPos}_{\mathsf{prev}(t)}(x) = \{\, n+1-k \mid k \in \mathsf{LPos}_{\mathsf{prev}(t^R)}(\widetilde{x}) \,\}$, we have $x = \lceil [x]^{\mathrm{R}}_{\mathsf{prev}(t)} \rceil$ if and only if $\widetilde{x} = \lceil [\widetilde{x}]_{\mathsf{prev}(t^R)}{}^{\mathrm{L}} \rceil$. To see why this holds more intuitively, let $\mathsf{parent}(\widetilde{x})$ be the parent of $\widetilde{x}$ in $\mathsf{PSTree}(t^R)$, and let ℓ be the edge label from $\mathsf{parent}(\widetilde{x})$ to $\widetilde{x}$. Then, for any locus on this edge representing $\widetilde{z_i} = \mathsf{parent}(\widetilde{x}) \cdot \ell[1:i]$, with $1 \le i \le |\ell|$, there are the same leaves below it. Since each leaf of $\mathsf{PSTree}(t^R)$ corresponds to a distinct position in $\mathsf{prev}(t^R)$, every $\widetilde{z_i}$ has the same set of beginning positions in $\mathsf{prev}(t^R)$ (note that $\widetilde{z_\ell} = \widetilde{x}$). By symmetry, this in turn means that z_i has the same set of ending positions in $\widetilde{\mathsf{prev}(t^R)} = \mathsf{prev}(t)$, i.e. $\{z_i \mid 1 \le i \le |\ell|\} = [x]^{\mathrm{R}}_{\mathsf{prev}(t)}$. The other way (from PDAWG nodes to p-suffix tree nodes) can be shown analogously.

(2) Because $(\widetilde{x}, a, \widetilde{y})$ is an explicit Weiner link, $\alpha(a, \widetilde{x}) = \widetilde{y}$. By the definition of operator $\widetilde{\cdot}$, we obtain $xa = y$. Hence there is a primary edge from node x to y labeled a in $\mathsf{PDAWG}(t)$. The other way (from PDAWG primary edges to p-suffix tree explicit Weiner links) can be shown analogously.

(3) Similar to (2).

(4) Immediately follows from the proof for (1).

Further, we obtain the following corollary that allows for bidirectional parameterized pattern searches:

Corollary 3. *Using a pair of* $\mathsf{PDAWG}(t)$ *and* $\mathsf{PSTree}(t^R)$ *with pv-encodings* $\mathsf{prev}(t)$ *and* $\mathsf{prev}(t^R)$*, one can perform forward and backward search to find all substrings of* t *that p-match a given pattern* P *in* $O(m \log(|\Pi| + |\Sigma|) + occ)$ *time, where* m *is the length of pattern* P *and* occ *is the number of occurrences to report.*

Proof. For a query pattern P that grows in both directions, one can easily maintain $\langle P \rangle$ in $O(\log |\Pi|)$ time per added character using $O(|\Pi|)$ working space.

It is known that $\mathsf{PSTree}(t^R)$ allows for the navigation of a (reversed) pattern P^R of length m in $O(m \log(|\Pi| + |\Sigma|))$ time [3]. This can be translated to an amortized $O(\log(|\Pi| + |\Sigma|))$-time navigation per input character that is added to the *left end* of P (which is the right-end of P^R.) A symmetric argument holds for our $\mathsf{PDAWG}(t)$, which leads to an amortized $O(\log(|\Pi| + |\Sigma|))$-time navigation per input character that is added to the *right end* of P.

6.4.2 Size of PDAWGs

Blumer et al. [6] have shown that the DAWG for a static string of length n has at most $2n-1$ nodes and $3n-4$ edges. We show that PDAWGs have the same size bound.

Theorem 7. $\mathsf{PDAWG}(T)$ *has at most* $2n-1$ *nodes and* $3n-3$ *edges when* $n = |T| \geq 3$. *Those bounds are tight.*

Proof. From the Theorem 6, there is a duality between the number of nodes in $\mathsf{PDAWG}(t)$ and the number of nodes in $\mathsf{PSTree}(t^R)$. It is obvious that the number of nodes in $\mathsf{PSTree}(t^R)$ is $2n-1$.

On the number of edges, we first give a weaker upper bound $3n-3$, just like Blumer et al. [6] have done. Let $\mathsf{PDAWG}(T) = G = (V, E, F)$, $\mathsf{PSTrie}(T) = H = (U, D)$, $V' \subsetneq V$ the set of non-sink nodes of G, $U' \subsetneq U$ the set of internal nodes of H, and $d_G(v)$ denote the out-degree of node v in G. We have $|E| = \sum_{v \in V'} d_G(v) = \sum_{v \in V'} d_H(\lceil v \rceil)$, since $d_G(v) = d_H(\lceil v \rceil)$ for all $v \in V$. Since H has at most n leaves,

$$n \geq |U \setminus U'| = 1 + \sum_{u \in U'} (d_H(u) - 1) \geq 1 + \sum_{v \in V'} (d_H(\lceil v \rceil) - 1) = 1 + |E| - |V'|,$$

which implies $|E| \leq n + |V'| - 1 \leq 3n - 3$.

6.4.3 Offline construction of PDAWGs via p-suffix trees

In this section, we present a fast offline construction algorithm for $\mathsf{PDAWG}(t)$, provided that $\mathsf{PSTree}(t^R)$ (without suffix links) has already been built.

By the definition of our Weiner links on parameterized suffix trees, the following monotonicity holds.

Lemma 22. *Suppose that a node* v *in* $\mathsf{PSTree}(t^R)$ *has an (implicit or explicit) Weiner link* (v, k, u) *with label* $k \in \Sigma \cup \mathcal{N}$. *Then, any ancestor* v' *of* v *has an (implicit or explicit) Weiner link* $(v', \langle\langle k \rangle\rangle_{|v'|}, u')$ *where* u' *is the shallowest ancestor of* u *with* $|u'| \geq |v'| + 1$.

It follows from Theorem 6 and Lemma 22 that there is a simple *offline* algorithm that builds the PDAWG by computing Weiner links in a bottom-up manner over the PST for the reversed text string.

Theorem 8. *Let* t *be a p-string of length* n. *Given* $\mathsf{PSTree}(t^R)$ *(without suffix links), we can build* $\mathsf{PDAWG}(t)$ *in* $O(n)$ *time and space.*

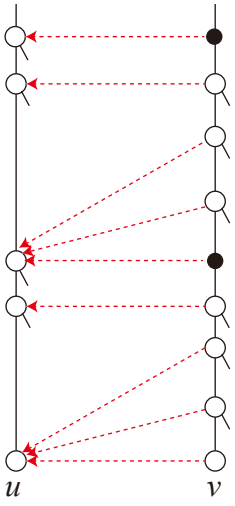

Figure 6.2: Illustration for our algorithm of Theorem 8 that propagates the Weiner links between the ancestors of v and u, in a bottom up manner. The white circles represent nodes of $\mathsf{PSTree}(t^R)$ and the black circles represent imaginary nodes. The red arcs represent Weiner links.

Proof. We first compute the (reversed) suffix links of the nodes of $\mathsf{PSTree}(t^R)$ that corresponds to $\mathsf{prev}(S)$ for all suffixes of t, together with their labels which are the characters of $\mathsf{prev}(t)$.

For each Weiner link $L = (v, k, u)$ between two nodes corresponding to two consecutive suffixes of t, perform the following:

(1) Perform the following $\mathsf{update}(v, u)$ function:

  (a) If $|\mathsf{parent}(u)| = \mathsf{parent}(v) + 1$, set $v \leftarrow \mathsf{parent}(v)$ and $u \leftarrow \mathsf{parent}(u)$.

  (b) If $|\mathsf{parent}(u)| < \mathsf{parent}(v) + 1$, set $v \leftarrow \mathsf{parent}(v)$.

  (c) If $|\mathsf{parent}(u)| > \mathsf{parent}(v) + 1$, then create an imaginary node w at string depth $|\mathsf{parent}(u)| - 1$ between $\mathsf{parent}(v)$ and v. Set $v \leftarrow w$ and $u \leftarrow \mathsf{parent}(u)$.

(2) (a) If there is no Weiner link between v and u, create a new Weiner link $(v, \langle\langle k \rangle\rangle_{|v|}, u)$. Go to (1).

  (b) Otherwise, stop the propagation for L.

See also Figure 6.2 that illustrates our algorithm. The correctness of this algorithm is immediate from Lemma 22.

It is clear from Lemma 22 that the complexity of this algorithm is linear in the number of Weiner links created. It follows from Theorem 7 and Theorem 6 that the number of Weiner links (v, k, u) such that v is *not* an imaginary node is $O(n)$. It also follows from our duality discussion in Section 6.4.3. that each Weiner link (w, k, u) such that w is an imaginary node corresponds to an edge in $E' \setminus E$, and E is the set of edges of $\mathsf{PDAWG}(t)$. Since $|E'| = O(n)$.the total time complexity of this algorithm is $O(n)$.

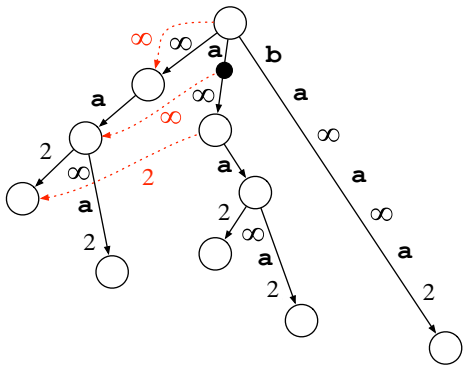

Figure 6.3: An example for how the Weiner links are propagated in a bottom-up manner, with the same string as in Figure 6.1. We pick the Weiner link $(\mathtt{a0}, 2, \mathtt{0a2})$ between two nodes $\mathtt{a0} = \mathsf{prev}(\mathtt{ay})$ and $\mathtt{0a2} = \mathsf{prev}(\mathtt{yay})$. An imaginary node (black circle) is created at string depth $|\mathtt{0a}| - 1 = 1$. By removing the imaginary nodes and the Weiner links from them, we obtain $\mathsf{PDAWG}(t)$ (see Figure 6.1).

See also Figure 6.3 for a concrete example of our offline algorithm that computes $\mathsf{PDAWG}(t)$ from $\mathsf{PSTree}(t^R)$.

Chapter 7

Parameterized Suffix Tray

7.1 Introduction

Basically, the existing indexing structures for p-matching are designed upon indexing structure for exact pattern matching. Namely, *parameterized suffix trees* [4], *parameterized suffix arrays* [15], *parameterized DAWGs* [38], *parameterized CDAWGs* [38], *parameterized position heaps* [34, 20], and *parameterized BWTs* [23] are based on their exact matching counterparts: suffix trees [45], suffix arrays [36], DAWGs [6], CDAWGs [7], position heaps [34, 17], and BWTs [9], respectively. It should be emphasized that extending exact-matching indexing structures to parameterized matching is not straightforward and poses algorithmic challenges. Let n, m, $|\Pi|$ and $|\Sigma|$ be the lengths of a text t, a pattern P, the number of distinct symbols of $|\Pi|$ that appear in t and the number of distinct symbols of $|\Sigma|$ that appear in t, respectively. While there exist a number of algorithms which construct the suffix array for t in $O(n)$ time in the case of integer alphabets of polynomial size in n [18, 27, 31, 29, 41, 2], the best known algorithms build the parameterized suffix array for t (denoted $\mathsf{PSA}(t)$) in $O(n \log(|\Sigma| + |\Pi|))$ time via the suffix tree [4, 44], or directly in $O(n|\Pi|)$ time [21]. The existence of a pure linear-time algorithm for building $\mathsf{PSA}(t)$ and the parameterized suffix tree (denoted $\mathsf{PSTree}(t)$) in the case of integer alphabets remains open.

PPM queries can be supported in $O(m + \log n + occ)$ time by $\mathsf{PSA}(t)$ coupled with the parameterized LCP array (denoted $\mathsf{PLCP}(t)$) [15], or in $O(m \log(|\Sigma| + |\Pi|) + occ)$ time by $\mathsf{PSTree}(t)$ [4], where occ is the number of occurrences to report.

In this paper, we propose a new indexing structure for p-matching, the *parameterized suffix tray* for t (denoted $\mathsf{PSTray}(t)$). $\mathsf{PSTray}(t)$ is a combination of $\mathsf{PSTree}(t)$ and $\mathsf{PSA}(t)$ and is an analogue to the *suffix tray* indexing structure for exact matching [12]. We show that our

$\mathsf{PSTray}(t)$

(1) occupies $O(n)$ space,

(2) supports PPM queries in $O(m + \log(|\Sigma| + |\Pi|) + occ)$, and

(3) can be constructed in $O(n)$ time from $\mathsf{PSTree}(t)$ and $\mathsf{PSA}(t)$.

Result (3) implies that $\mathsf{PSTray}(t)$ can be constructed in $O(n \min\{\log(|\Sigma| + |\Pi|), |\Pi|\})$ time using $O(n)$ working space [4, 44, 21]. Results (1) and (2) together with this imply that our $\mathsf{PSTray}(t)$ is the fastest linear-space indexing structure for PPM which can be built in time linear in n.

We emphasize that extending suffix trays for exact matching [12] to parameterized matching is also not straightforward. The suffix tray of a string $t \in \Sigma^*$ is a hybrid data structure of the suffix tree and suffix array of t, designed as follows: Each of the $O(\frac{n}{|\Sigma|})$ carefully-selected nodes of the suffix tree stores an array of fixed size $|\Sigma|$, so that pattern traversals within these selected nodes take $O(m)$ time (this also ensures a total space to be $O(\frac{n}{|\Sigma|} \times |\Sigma|) = O(n)$). Once the pattern traversal reaches an unselected node, then the search switches to the sub-array of the suffix array of size $O(|\Sigma|)$. This ensures a worst-case $O(m + \log |\Sigma| + occ)$-time pattern matching with the suffix tray.

Now, recall that the previous-encoded suffixes of t are sequences over an alphabet $\Sigma \cup \{0, \ldots, n-1\}$ of size $\Theta(|\Sigma| + n) \subseteq O(n)$, while the alphabet size of t is $|\Sigma| + |\Pi|$. This means that naïve extensions of suffix trays to PPM would only result in either super-linear $O(\frac{n^2}{|\Sigma|+|\Pi|})$ space, or $O(m + \log n + occ)$ query time which can be achieved already with the parameterized suffix array. We overcome this difficulty by using the *smallest parameterized encoding* (*spe*) of strings which was previously proposed by the authors in the context of PPM on labeled trees [22], and this leads to our $O(n)$-space parameterized suffix trays with desired $O(m + \log(|\Sigma| + |\Pi|) + occ)$ query time.

7.2 Parameterized suffix trays

In this section, we propose a new indexing structure called the *parameterized suffix tray* for PPM, and we discuss its space requirements.

Our parameterized suffix trays are a "hybrid" data structure of parameterized suffix trees and parameterized suffix arrays, which are defined as follows:

Definition 14 (Parameterized suffix trees [4])**.** *The* parameterized suffix tree *for a p-string t, denoted* $\mathsf{PSTree}(t)$*, is a compact trie that stores the set* $\{\mathsf{prev}(t[i:]) \mid 1 \leq i \leq |t|\}$ *of the previous encodings of all suffixes of t.*

See Fig. 7.1 for examples of $\mathsf{PSTree}(t)$. We assume that the leaves of $\mathsf{PSTree}(t)$ are sorted in lexicographical order, so that the sequence of the leaves corresponds to the parameterized suffix array for t, which is defined below.

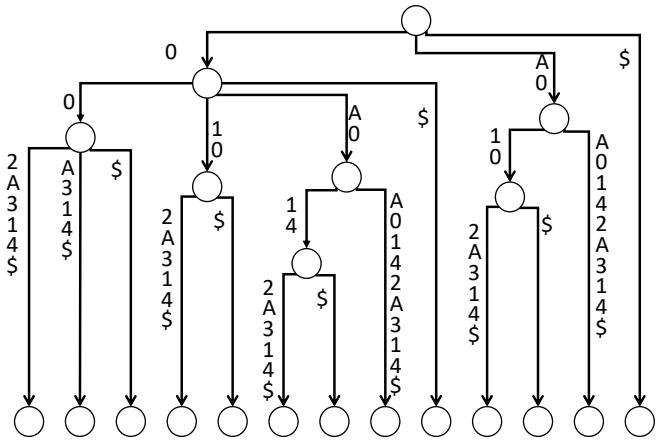

Figure 7.1: $\mathsf{PSTree}(t)$ for a p-string $t = \mathtt{zAxAyyxyAxxy}$, where $\Sigma = \{\mathtt{A}, \$\}, \Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.

Definition 15 (Parameterized suffix arrays [15])**.** *The* parameterized suffix array *of a p-string t, denoted* $\mathsf{PSA}(t)$*, is an array of integers such that* $\mathsf{PSA}(t)[i] = j$ *if and only if* $\mathsf{prev}(t[j:])$ *is the ith lexicographically smallest string in* $\{\mathsf{prev}(t[i:]) \mid 1 \leq i \leq |t|\}$.

Definition 16 (Parameterized longest common prefix arrays [15])**.** *The parameterized longest common prefix array of a p-string t, denoted* $\mathsf{PLCP}(t)$*, is an array of integers such that* $\mathsf{PLCP}(t)[1] = 0$ *and* $2 \leq i \leq |t|$ $\mathsf{PLCP}(t)[i]$ *stores the length of the longest common prefix between* $\mathsf{prev}(t[\mathsf{PSA}(t)[i-1]:])$ *and* $\mathsf{prev}(t[\mathsf{PSA}(t)[i]:])$.

See Fig. 7.2 for examples of $\mathsf{PSA}(t)$ and $\mathsf{PLCP}(t)$.

In addition to the above data structures from the literature, we introduce the following new notions and data structures. For convenience, we will sometimes identify each node of the parameterized suffix tree with the string which is represented by that node.

In what follows, let $\Pi_t = \{t[i] \in \Pi \mid 1 \leq i \leq |t|\}$ and $\Sigma_t = \{t[i] \in \Sigma \mid 1 \leq i \leq |t|\}$, namely, Π_t (resp. Σ_t) is the set of distinct characters of Π (resp. Σ) that occur in t.

i	$\mathsf{PSA}(t)[i]$	$t[\mathsf{PSA}(t)[i]:]$	$\mathsf{PLCP}(i)$
1	6	0 0 2 A 3 1 4 $	0
2	7	0 0 A 3 1 4 $	2
3	11	0 0 $	2
4	5	0 1 0 2 A 3 1 4 $	1
5	10	0 1 0 $	3
6	3	0 A 0 1 4 2 A 3 1 4 $	1
7	8	0 A 0 1 4 $	5
8	1	0 A 0 A 0 1 4 2 A 3 1 4 $	3
9	12	0 $	1
10	4	A 0 1 0 2 A 3 1 4 $	0
11	9	A 0 1 0 $	4
12	2	A 0 A 0 1 4 2 A 3 1 4 $	2
13	13	$	0

Figure 7.2: $\mathsf{PSA}(t)$ and $\mathsf{PLCP}(t)$ for a p-string $t = \texttt{zAxAyyxyAxxy}$, where $\Sigma = \{\texttt{A}, \$\}, \Pi = \{\texttt{x}, \texttt{y}, \texttt{z}\}$.

Definition 17 (P-nodes, branching p-nodes)**.** *Let t be a p-string over $\Sigma \cup \Pi$. A node v in $\mathsf{PSTree}(t)$ is called a* p-node *if the number of leaves in the subtree of $\mathsf{PSTree}(t)$ rooted at v is at least $\max\{|\Sigma|, |\Pi|\}$. A p-node v is called a* branching p-node *if at least two children of v in $\mathsf{PSTree}(t)$ are p-nodes.*

See Fig. 7.3 for examples of p-nodes and branching p-nodes.

For any $x \in \Pi_t$, let $\mathsf{rank}_t(x)$ denote the lexicographical rank of x in $\Pi_t \cup \Sigma_t$. Assuming that Π and Σ are integer alphabets of polynomial size in n, we can compute $\mathsf{rank}_t(x)$ for every $x \in \Pi_t$ in $O(n)$ time by bucket sort. We will abbreviate $\mathsf{rank}_t(x)$ as $\mathsf{rank}(x)$ when it is not confusing.

Definition 18 (P-array)**.** *Let $\mathsf{prev}(v)$ be any branching p-node of $\mathsf{PSTree}(t)$, where v is some substring of t. The* p-array *$A(\mathsf{prev}(v))$ for $\mathsf{prev}(v)$ is an array of length $|\Sigma| + |\Pi|$ such that for each $x \in \Sigma \cup \Pi$, $A(\mathsf{prev}(v))[\mathsf{rank}(x)]$ stores a pointer to the child u of $\mathsf{prev}(v)$ such that $\mathsf{prev}(\mathsf{spe}(v)x)$ is a prefix of u if such a child exists, and $A(\mathsf{prev}(v))[\mathsf{rank}(x)]$ stores nil otherwise.*

See Fig. 7.4 for an example of a p-array.

Definition 19 (Parameterized suffix tray)**.** *The parameterized suffix tray of a p-string t, denoted $\mathsf{PSTray}(t)$, is a hybrid data structure consisting of $\mathsf{PSA}(t)$, $\mathsf{PLCP}(t)$, and $\mathsf{PSTree}(t)$ where each branching p-node is augmented with the p-array.*

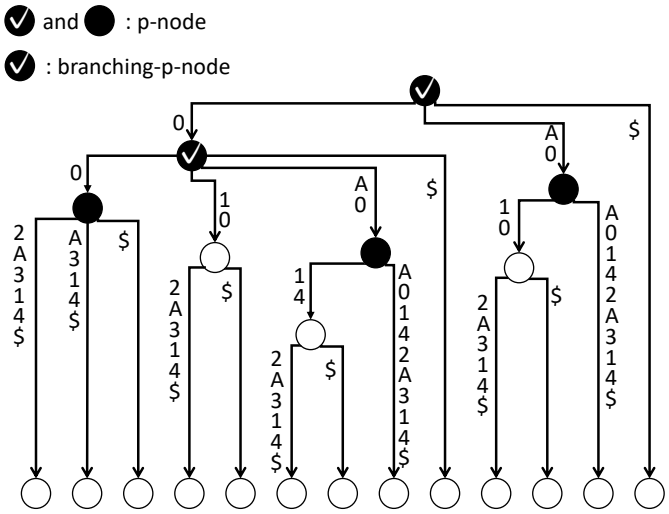

Figure 7.3: $\mathsf{PSTree}(t)$ for a p-string $t = \mathtt{zAxAyyxyAxxy}$, where $\Sigma = \{\mathtt{A}, \$\}, \Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.Then black nodes are p-nodes because the number of leaves in the subtree of $\mathsf{PSTree}(t)$ rooted at them are at least $\max\{|\Sigma|, |\Pi|\} = 3$. Checked nodes are branching p-nodes because at least two children of them in $\mathsf{PSTree}(t)$ are p-nodes.

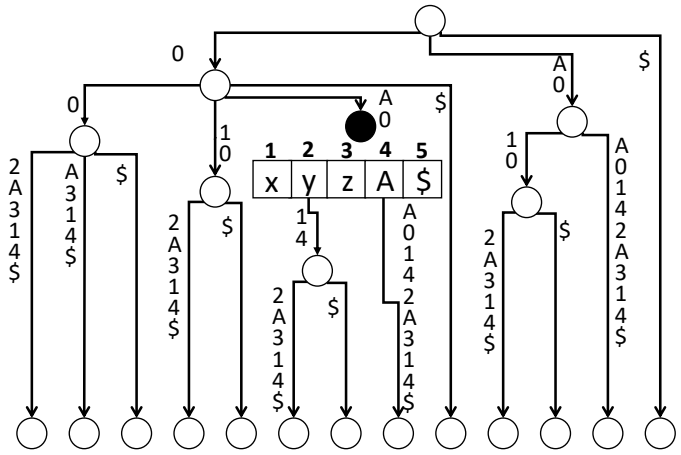

Figure 7.4: $\mathsf{PSTree}(t)$ for a p-string $t = \mathtt{zAxAyyxyAxxy}$, where $\Sigma_t = \{\mathtt{A}, \$\}, \Pi_t = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Consider a branching p-node $\mathsf{prev}(v) = \mathtt{0A0}$ where v is e.g. $\mathtt{zAx}$. Then, $A(\mathtt{0A0})[\mathsf{rank}(\mathtt{y})] = A(\mathtt{0A0})[2]$ stores a pointer to node $\mathtt{0A014}$ because $\mathsf{spe}(v) = \mathtt{xAy}$ and $\mathsf{prev}(\mathsf{spe}(v)\mathtt{y} = \mathtt{xAyy}) = \mathtt{0A01}$ is a prefix of node $\mathtt{0A014}$.

We can show the following lemmas regarding the space requirements of $\mathsf{PSTray}(t)$, by similar arguments to [12] for suffix trays on standard strings.

Lemma 23. *For any p-string t of length n over $\Sigma \cup \Pi$, the number of branching p-nodes in $\mathsf{PSTree}(t)$ is $O(\frac{n}{|\Pi|+|\Sigma|})$.*

Lemma 24. *For any p-string t of length n, $\mathsf{PSTray}(t)$ occupies $O(n)$ space.*

7.3 PPM using parameterized suffix trays

In this section, we present our algorithm for parameterized pattern matching (PPM) on $\mathsf{PSTray}(t)$. For any node v in $\mathsf{PSTray}(t)$, let $l_v = (i, j)$ denote the range of $\mathsf{PSA}(t)$ that v corresponds, namely, $l_v = (i, j)$ iff the leftmost and rightmost leaves in the subtree rooted at v correspond to the ith and jth entries of $\mathsf{PSA}(t)$, respectively. For any *p-node* v, we store l_v in v. Also, for any *non-branching* p-node u, we store a pointer to the unique child of u that is a p-node. These can be easily computed in a total of $O(n)$ time by a standard traversal on $\mathsf{PSTree}(t)$.

The basic strategy for PPM with $\mathsf{PSTray}(t)$ follows the (exact) pattern matching algorithm with suffix trays on standard strings [12]. Namely, we traverse $\mathsf{PSTree}(t)$ with a given pattern P from the root, and as soon as we encounter a node that is not a p-node, then we switch to the corresponding range of $\mathsf{PSA}(t)$ and perform a binary search to locate the pattern occurrences. The details follow.

Let P be a pattern p-string of length m. We assume that Π and Σ are disjoint integer alphabets, where $\Pi = \{0, \ldots, c_1 n\}$ and $\Sigma = \{c_1 n + 1, \ldots, n^{c_2}\}$ for some positive constants c_1 and c_2. Using an array (bucket) B of size $|\Pi| = c_1 n \in O(n)$, we can compute $\mathsf{prev}(P)$ in $O(m)$ time by scanning P from left to right and keeping the last occurrence of each character $x \in \Pi$ in P in $B[x]$. We can compute $\mathsf{spe}(P)$ in $O(m)$ time in a similar manner with a bucket. These buckets are a part of our indexing structure that occupies $O(n)$ total space.

After computing $\mathsf{prev}(P)$ and $\mathsf{spe}(P)$, we traverse $\mathsf{prev}(P)$ on $\mathsf{PSTray}(t)$. If $\mathsf{prev}(P[:i])$ for prefix $P[:i]$ $(1 \leq i \leq m)$ is represented by a p-node, we can find the out-going edge whose label begins with $\mathsf{prev}(P)[i+1]$ in constant time by accessing the p-array entry $A(\mathsf{prev}(P[:i]))[\mathsf{spe}(P)[i+1]]$. Therefore, we can solve PPM in $O(m + occ)$ time if $\mathsf{prev}(P)$ is a prefix of some p-node in $\mathsf{PSTray}(t)$. Otherwise (if $\mathsf{prev}(P)$ is not a prefix of any p-node), there exists integer i such that $\mathsf{prev}(P[:i])$ is not a p-node but the parent of $\mathsf{prev}(P[:i])$ is a p-node. In this case, we will use the next lemma.

Lemma 25 (PPM in PSA range (adapted from [15]))**.** *Given a pattern p-string P of length m and a range $[j, k]$ in $\mathsf{PSA}(t)$ such that the occ occurrences of P in t lie in the range $[j, k]$ of $\mathsf{PSA}(t)$, we can find them in $O(m + \log(k - j) + occ)$ time by using $\mathsf{PSA}(t)$ and $\mathsf{PLCP}(t)$.*

Let $I_{\mathsf{prev}(P[:i])} = (j, k)$ denote the range in $\mathsf{PSA}(t)$ where $\mathsf{prev}(P)$ is a prefix of the suffixes in the range. We apply Lemma 25 to this range so we can find the parameterized occurrences of P in t in $O(m+\log(k-j)+occ)$ time. By Definition 17 we have $k-j \leq |\Pi|+|\Sigma|$ (recall that $\mathsf{prev}(P[: i])$ is not a p-node). Thus, $O(m + \log(k - j) + occ) \subseteq O(m + \log(|\Pi| + |\Sigma|) + occ)$, implying the next theorem.

Theorem 9. *Suppose $|\Pi| = O(n)$. Then, $\mathsf{PSTray}(t)$ supports PPM queries in $O(m+\log(|\Pi|+|\Sigma|) + occ)$ time each, where m is the length of a query pattern P and occ is the number of occurrences to report.*

7.4 Construction of parameterized suffix trays

Let t be a p-string of length n. In this section, we show how to construct $\mathsf{PSTray}(t)$ provided that $\mathsf{PSTree}(t)$ has already been built. Throughout this section we assume that Π and Σ are disjoint integer alphabets, both being of polynomial size in n, namely, $\Pi = \{0, ..., n^{c_1}\}$ and $\Sigma = \{n^{c_1} + 1, ..., n^{c_2}\}$ for some positive constants c_1 and c_2. For convenience, we define the following two notions.

Definition 20 (P-function)**.** *Let q, r be p-strings such that $q \approx r$. The* p-function $\mathsf{f}_{q,r} : \Sigma \cup \Pi \to \Sigma \cup \Pi$ *transforms q to r, namely, for every $1 \leq i \leq h$*

$$\mathsf{f}_{q,r}(q[i]) = r[i].$$

For instance, if $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, $q = \mathtt{xyxzyyxz}$, $r = \mathtt{zxzyxxzy}$ and $q \approx r$, then $\mathsf{f}_{q,r}(\mathtt{x}) = \mathtt{z}$, $\mathsf{f}_{q,r}(\mathtt{y}) = \mathtt{x}$, $\mathsf{f}_{q,r}(\mathtt{z}) = \mathtt{y}$ since q can be transformed r by this function.

Definition 21 (F-array)**.** *Let q be a p-string and $x \in \Pi_t$. The first (left-most) occurrence of x in q is denoted by $i_{q,x}$. The* f-array *of q, denoted $\mathsf{fpos}(q)$, is an array of length $|\Pi|$ such that $\mathsf{fpos}(q)[\mathsf{rank}(x)] = i_{q,x}$.*

For instance, if $\Pi_t = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and $q = \mathtt{xyxzyyxz}$, then $\mathsf{fpos}(q)[\mathsf{rank}(\mathtt{x})] = \mathsf{fpos}(q)[1] = 1$, $\mathsf{fpos}(q)[\mathsf{rank}(\mathtt{y})] = \mathsf{fpos}(q)[2] = 2$, and $\mathsf{fpos}(q)[\mathsf{rank}(\mathtt{z})] = \mathsf{fpos}(q)[3]$
$= 4$.

Given $\mathsf{PSTree}(t)$, we show how to construct $\mathsf{PSTray}(t)$. It is well known that $\mathsf{PSA}(t)$ and $\mathsf{PLCP}(t)$ can be constructed from $\mathsf{PSTree}(t)$ in $O(n)$ time. In the following, we consider how to compute $A(\mathsf{prev}(v))$ for every p-node $\mathsf{prev}(v)$ in $\mathsf{PSTree}(t)$.

First, we consider how to compute (branching) p-nodes in $\mathsf{PSTree}(t)$. This can be done by a similar method to the suffix tray for exact matching [12], namely:

Lemma 26 (Computing p-node)**.** *We can compute all p-nodes and branching p-nodes in* $\mathsf{PSTree}(t)$ *in* $O(n)$ *total time.*

Our algorithm performs a bottom-up traversal on $\mathsf{PSTree}(t)$ and propagates pairs $(\mathsf{fpos}(t[i:]), i)$ from leaves to their ancestors. Each internal p-node $\mathsf{prev}(v)$ will store only a single pair $(\mathsf{fpos}(t[i:]), i)$, where i is the largest position in t such that $\mathsf{prev}(t[i:])$ is a leaf in the subtree rooted at $\mathsf{prev}(v)$ [1]. See also Fig. 7.5. One can easily compute the pairs for all p-nodes in a total of $O(n)$ time. Then, we compute $\mathsf{f}_{v,\mathsf{spe}(v)}$ for every p-node $\mathsf{prev}(v)$ from the pair $(\mathsf{fpos}(t[i:]), i)$

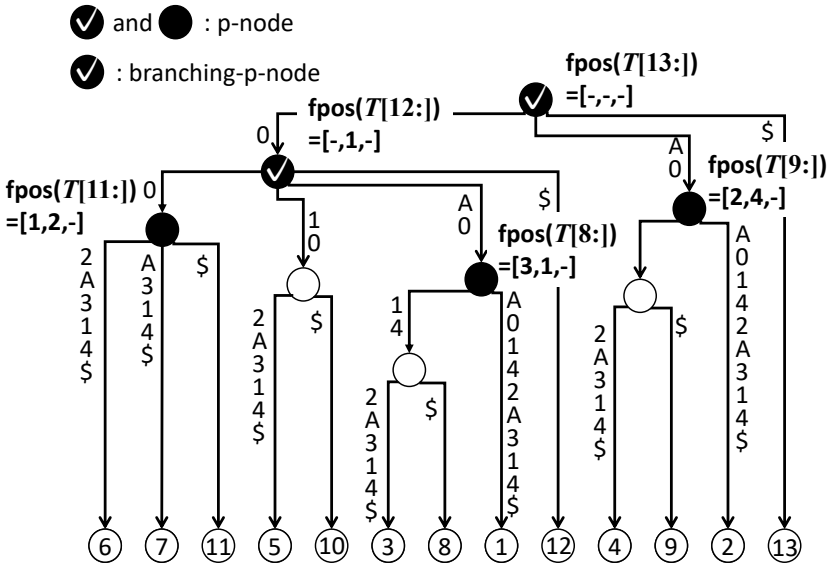

Figure 7.5: $\mathsf{PSTree}(t)$ for a p-string $t = \mathtt{zAxAyyxyAxxy}$, where $\Sigma = \{\mathtt{A}, \$\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. For instance, we propagate $\mathsf{fpos}(t[12:])$ (coupled with the corresponding position 12) to the p-node 0.

that is stored in the p-node $\mathsf{prev}(v)$. Finally, for every p-node $\mathsf{prev}(v)$ we compute $A_{\mathsf{prev}(v)}$ from $\mathsf{f}_{v,\mathsf{spe}(v)}$ and i.

[1] Indeed, our $\mathsf{PSTray}(t)$ construction algorithm works with any position i in the subtree rooted at $\mathsf{prev}(v)$, and we propagate the largest leaf position i to each internal p-node for simplicity.

In what follows, we first show how to compute $A_{\mathsf{prev}(v)}$ from $\mathsf{f}_{v,\mathsf{spe}(v)}$ and i in Lemmas 27 and 28. We then present how to compute $\mathsf{f}_{v,\mathsf{spe}(v)}$ and i from $\mathsf{fpos}(t[i:])$ in Lemma 29, and how to compute $\mathsf{fpos}(t[i:])$ in Lemma 30. These lemmas will ensure the correctness and time complexity of our algorithm.

We consider how to compute $A(\mathsf{prev}(v))$ for a given p-node $\mathsf{prev}(v)$.

Lemma 27. *Let s be a p-string. If $\mathsf{prev}(s)[|s|] = k \in \{0, \ldots, |t|-1\}$, then $\mathsf{spe}(s)[|s|] = \mathsf{spe}(s)[|s|-k]$.*

Proof. Clear from the definitions of $\mathsf{prev}(\cdot)$ and $\mathsf{spe}(\cdot)$. □

In the sequel, let $\mathsf{prev}(t[i:])$ be any leaf in the subtree rooted at $\mathsf{prev}(v)$, where $1 \leq i \leq |t|$. By Lemma 27, we can compute $A(\mathsf{prev}(v))$ if we know $\mathsf{spe}(v)[|v|-k+1]$, where $k = \mathsf{prev}(t[i:])[|v|+1]$.

Lemma 28. $\mathsf{spe}(v)[|v|-k+1] = \mathsf{f}_{t[i:i+|v|-1],\mathsf{spe}(t[i:i+|v|-1])t[i+|v|+k-2]}$.

Proof. Clear from the definitions of $\mathsf{f}_{\cdot,\cdot}$ and $\mathsf{spe}(\cdot)$. □

We can compute $\mathsf{spe}(v)[|v|-k+1]$ if we know $\mathsf{f}_{t[i:i+|v|-1],\mathsf{spe}(v)}$. In the next lemma, we show how to compute $\mathsf{f}_{t[i:i+|v|-1],\mathsf{spe}(v)}$ for all p-nodes $\mathsf{prev}(v)$.

Lemma 29. *For every p-node $\mathsf{prev}(v)$, we can compute $\mathsf{f}_{t[i:i+|v|-1],\mathsf{spe}(t[i:i+|v|-1])}$ with $\mathsf{prev}(v) = \mathsf{prev}(t[i:i+|v|-1]))$ in amortized $O(|\Pi|)$ time if we know pair $\mathsf{fpos}(t[i:], i)$.*

Proof. Let x_j be the jth smallest element of Π in lexicographically order. Let y_l denote the parameterized character in $S = t[i:i+|v|-1]$ such that if j is the left-most occurrence of y_l in S (i.e. $j = \min\{h \mid S[h] = y_l\}$), then $|\Pi_{S[1:j]}| = l$. For instance, for $S = \mathtt{xAzAyA}$, then $y_1 = \mathtt{x}$, $y_2 = \mathtt{z}$, and $y_3 = \mathtt{y}$. Let $(i_{\mathsf{prev}(v)}, \mathsf{fpos}(t[i_{\mathsf{prev}(v)}:])$ denote the pair stored in p-node $\mathsf{prev}(v)$. Consider a set $\mathbf{I} = \{(i_{\mathsf{prev}(v)}, \mathsf{fpos}(t[i_{\mathsf{prev}(v)}:])[k]) \mid \mathsf{prev}(v) \text{ is a p-node}, 1 \leq k \leq |\Pi|\}$ of integer pairs. We sort the elements of $\mathbf{I}$ so that we can compute in $O(1)$ time $\mathsf{f}_{t[i:i+|v|-1],\mathsf{spe}(v)}(y_l) = x_l$ for all p-nodes $\mathsf{prev}(v)$, where x_l is the lth smallest parameterized character that occurs in $\mathsf{spe}(v)$. We can sort the elements of $\mathbf{I}$ in a total of $O(n)$ time by radix sort, since there are $O(\frac{n}{|\Sigma|+|\Pi|})$ p-nodes and each f-array $\mathsf{fpos}(t[i:])$ is of length $|\Pi|$. This completes the proof. □

We can easily compute $\mathsf{fpos}(t[i:])$ by the following lemma:

Lemma 30. *Let q be a p-string. Let $x \in \Pi$, $y \in \Pi \cup \Sigma$ and $x \neq y$. Then the following equations hold:*

$$\begin{aligned} \mathsf{fpos}(xq)[\mathsf{rank}(x)] &= 1, \\ \mathsf{fpos}(xq)[\mathsf{rank}(y)] &= \mathsf{fpos}(q)[\mathsf{rank}(y)] + 1. \end{aligned}$$

Thus we can compute f-arrays $\mathsf{fpos}(t[i:])$ for all $1 \leq i \leq n$ in a total of $O(n)$ time.

Since all the afore-mentioned procedures take $O(n)$ time each, we obtain the main theorem of this section.

Theorem 10. *Given a p-string t of length n over alphabet $\Sigma \cup \Pi$ with $\Pi = \{0, ..., n^{c_1}\}$ and $\Sigma = \{n^{c_1} + 1, ..., n^{c_2}\}$ for some positive constants c_1 and c_2, we can construct $\mathsf{PSTray}(t)$ in $O(n)$ time from $\mathsf{PSTree}(t)$.*

7.5 Conclusions and open questions

In this paper, we proposed an indexing structure for parameterized pattern matching (PPM) called the parameterized suffix tray $\mathsf{PSTray}(t)$, where t is a given text string. Our $\mathsf{PSTray}(t)$ uses $O(n)$ space and supports pattern matching queries in $O(m + \log(|\Sigma| + |\Pi|) + occ)$ time, where $n = |t|$, m is the query pattern length, $|\Sigma|$ and $|\Pi|$ are respectively the numbers of distinct static characters and distinct parameterized characters occurring in t, and occ is the number of pattern occurrences to report. We also showed how to construct $\mathsf{PSTray}(t)$ in $O(n + s(n))$ time, where $s(n)$ denotes the time complexity to build the parameterized suffix tree $\mathsf{PSTree}(t)$ for t. It is known that $s(n) = \min\{n|\Pi|, n(\log(|\Pi| + |\Sigma|)\}$ [4, 44, 21].

On the other hand, if we use hashing for implementing the branches of the parameterized suffix tree $\mathsf{PSTree}(t)$, one can trivially answer PPM queries in $O(m+occ)$ time with $O(n)$ space. The best linear-space deterministic hashing we are aware of is the one by Ružić [42], which can be built in $O(n(\log\log n)^2)$ time for a set of n keys in the word RAM model with machine word size $\Omega(\log n)$. By associating each node of $\mathsf{PSTree}(t)$ with a unique integer (e.g. the pre-order rank), one can regard each branch in $\mathsf{PSTree}(t)$ as an integer from the universe of polynomial size in n, each fitting in a constant number of machine words. This gives us a deterministic $O(n(\log\log n)^2 + s(n))$-time algorithm for building $\mathsf{PSTree}(t)$ with $O(m + occ)$-time PPM queries. Still, it is not known whether a similar data structure can be build in $O(n + s(n))$ time. We conjecture that our $O(m + \log(|\Sigma| + |\Pi|) + occ)$ PPM query time would be the best possible

for any indexing structure that can be build in $O(n + s(n))$ time. Proving or disproving such a lower bound is an intriguing open problem.

Chapter 8

Conclusion

In this thesis, we studied indexing data structures for the parameterized pattern matching problem.

Let n be the length of the text, m be the pattern length, and occ be the number of pattern occurrences, N be the number of nodes of the CST, Σ be the static alphabet, Π be the parameterized alphabet.

In Chapter 3, we proposed the RLPPH. We showed that RLPPHs require $O(n)$ words of space, support PPM queries in $O(m|\Pi| + m\log(|\Pi| + |\Sigma|) + occ)$ time, and can be built in $O(n\log(|\Pi| + |\Sigma|))$ time and $O(n)$ words of working space.

In Chapter 4, we presented the first indexing data structure for the PPM problem on tree inputs which is called the PPH for trees. We showed our PPH for trees require $O(N)$ words of space, supports PPM queries on trees in $O(m\log(|\Pi| + |\Sigma|) + m|\Pi| + occ)$ time, and can be constructed in $O(N(|\Sigma| + |\Pi|))$ time and $O(N(|\Sigma| + |\Pi|))$ words of working space.

In Chapter 5, we proposed an offline algorithm for constructing PSAs without constructing PSTs, in $O(n|\Pi|)$ time and $O(n)$ words of working space. In addition, we showed how to build the PLCP from the PSA for the text t in $O(n|\Pi|)$ time. Further, we showed we can build the PST for t from the PSA and PLCP for t in linear time.

In Chapter 6, we showed an offline algorithm that constructs PDAWGs in $O(n\min\{\log(|\Pi|+|\Sigma|), |\Pi|\})$ time and $O(n)$ words of working space.

In Chapter 7, we proposed the PSTr. We showed our PSTr requires $O(n)$ words of space, supports queries in $O(m + \log(|\Pi| + |\Sigma|) + occ)$ time, and can be build in $O(n\min\{\log(|\Pi| + |\Sigma|), |\Pi|\})$ time and $O(n)$ words of working space, respectively.

Our future work includes the following:

- Would it be possible to shave the $m|\Pi|$ term in the pattern matching time using parameterized position heaps? Other data structures such as parameterized suffix trees achieve better $O(m \log(|\Sigma| + |\Pi|) + occ)$ time [4].
- Would it be possible to construct PSA in $O(n)$ when Π and Σ are integer alphabets ?

Bibliography

[1] A. Apostolico and Z. Galil. *Pattern Matching Algorithm*. Oxford University Press, New York, 1997.

[2] U. Baier. Linear-time suffix sorting - A new approach for suffix array construction. In *CPM 2016*, pages 23:1–23:12, 2016.

[3] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *STOC 1993*, pages 71–80, 1993.

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

[5] R. Beal and D. Adjeroh. Variations of the parameterized longest previous factor. *Journal of Discrete Algorithms*, 16:129 – 150, 2012. Selected papers from the 22nd International Workshop on Combinatorial Algorithms (IWOCA 2011).

[6] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[7] A. Blumer, J. Blumer, D. Haussler, R. Mcconnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[8] D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191(1–2):131–144, 1998.

[9] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm, 1994.

[10] E. Coffman and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13:427–432, 1970.

[11] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *SIAM Journal on Computing*, 33(1):26–42, 2003.

[12] R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015.

[13] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[14] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75 – 80, 2008.

[15] S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda. Parameterized suffix arrays for binary strings. In *PSC 2008*, pages 84–94, 2008.

[16] Diptarama, T. Katsura, Y. Otomo, K. Narisawa, and A. Shinohara. Position heaps for parameterized strings. In *Proc. CPM 2017*, pages 8:1–8:13, 2017.

[17] A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo. Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, 9(1):100–121, 2011.

[18] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[19] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In M. Lewenstein and G. Valiente, editors, *17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006.

[20] N. Fujisato, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Right-to-left online construction of parameterized position heaps. In *Proc. PSC 2018*, pages 91–102, 2018.

[21] N. Fujisato, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Direct linear time construction of parameterized suffix and LCP arrays for constant alphabets. In *SPIRE 2019*, pages 382–391, 2019.

[22] N. Fujisato, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. The parameterized position heap of a trie. In *Proceedings of the International Conference on Algorithms and Complexity*, pages 237–248, 2019.

[23] A. Ganguly, R. Shah, and S. V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 397–407, 2017.

[24] D. Hashimoto, D. Hendrian, D. Köppl, R. Yoshinaka, and A. Shinohara. Computing the parameterized burrows–wheeler transform online. In D. Arroyuelo and B. Poblete, editors, *String Processing and Information Retrieval*, pages 70–85, Cham, 2022. Springer International Publishing.

[25] T. I, S. Deguchi, H. Bannai, S. Inenaga, and M. Takeda. Lightweight parameterized suffix array construction. In *Proc. IWOCA 2009*, pages 312–323, 06 2009.

[26] K. Iseri, T. I, D. Hendrian, D. Köppl, R. Yoshinaka, and A. Shinohara. Breaking a barrier in constructing compact indexes for parameterized pattern matching, 2023.

[27] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[28] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In A. Amir and G. M. Landau, editors, *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.

[29] D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2-4):126–142, 2005.

[30] S.-H. Kim and H.-G. Cho. Simpler fm-index for parameterized string matching. *Information Processing Letters*, 165:106026, 2021.

[31] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210, 2003.

[32] S. Kosaraju. Pattern matching in compressed texts. In *Proc. Foundation of Software Technology and Theoretical Computer Science*, pages 349–362. Springer-Verlag, 1995.

[33] S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *FOCS 1995*, pages 631–637, 1995.

[34] G. Kucherov. On-line construction of position heaps. *J. Discrete Algorithms*, 20:3–11, 2013.

[35] T. Lee, J. C. Na, and K. Park. On-line construction of parameterized suffix trees for large alphabets. *Information Processing Letters*, 111(5):201–207, 2011.

[36] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.

[37] J. Mendivelso and Y. Pinzón. Parameterized matching: Solutions and extensions. In *Proc. PSC 2015*, pages 118–131, 2015.

[38] K. Nakashima, N. Fujisato, D. Hendrian, Y. Nakashima, R. Yoshinaka, S. Inenaga, H. Bannai, A. Shinohara, and M. Takeda. DAWGs for parameterized matching: Online construction and related indexing structures. In *CPM 2020*, pages 26:1–26:14, 2020.

[39] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. The position heap of a trie. In *Proc. SPIRE 2012*, volume 7608 of *Lecture Notes in Computer Science*, pages 360–371, 2012.

[40] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inf. Process. Lett.*, 115(9):655–659, 2015.

[41] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[42] M. Ružić. Constructing efficient dictionaries in close to sorting time. In *ICALP 2008*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008.

[43] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Transactions on Fundamentals of Electronics*, E86-A(5):1061–1066, 2003.

[44] T. Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica*, 39(1):1–19, 2004.

[45] P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.