The Parameterized Position Heap of a Trie*

Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
{noriki.fujisato, yuto.nakashima, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp

Abstract

Let Σ and Π be disjoint alphabets of respective size σ and π. Two strings over Σ ∪ Π of equal length are said to parameterized match (p-match) if there is a bijection \( f : \Sigma \cup \Pi \to \Sigma \cup \Pi \) such that (1) \( f \) is identity on \( \Sigma \) and (2) \( f \) maps the characters of one string to those of the other string so that the two strings become identical. We consider the p-matching problem on a (reversed) trie \( T \) and a string pattern \( P \) such that every path that p-matches \( P \) has to be reported. Let \( N \) be the size of the given trie \( T \). In this paper, we propose the parameterized position heap for \( T \) that occupies \( O(N) \) space and supports p-matching queries in \( O(m \log(\sigma + \pi) + m\pi + pocc) \) time, where \( m \) is the length of a query pattern \( P \) and \( pocc \) is the number of paths in \( T \) to report. We also present an algorithm which constructs the parameterized position heap for a given trie \( T \) in \( O(N(\sigma + \pi)) \) time and working space.

1 Introduction

The parameterized matching problem (p-matching problem), first introduced by Baker [2], is a variant of pattern matching which looks for substrings of a text that has “the same structure” as a given pattern. More formally, we consider a parameterized string (p-string) that can contain static characters from an alphabet \( \Sigma \) and parameter characters from another alphabet \( \Pi \). Two equal length p-strings \( x \) and \( y \) over the alphabet \( \Sigma \cup \Pi \) are said to parameterized match (p-match) if \( x \) can be transformed to \( y \) (and vice versa) by applying a bijection which renames the parameter characters. The p-matching problem is, given a text p-string \( w \) and pattern p-string \( p \), to report the occurrences of substrings of \( w \) that p-match \( p \). Studying the p-matching problem is well motivated by plagiarism detection, software maintenance, and RNA structural pattern matching [2, 15]. We refer readers to [11] for detailed descriptions about these motivations.

*YN, SI, HB, MT are respectively supported by JSPS KAKENHI Grant Numbers JP18K18002, JP17H01697, JP16H02783, JP18H04098.
Baker [2] proposed an indexing data structure for the p-matching problem, called the parameterized suffix tree (p-suffix tree). The p-suffix tree supports p-matching queries in \(O(m \log(\sigma + \pi) + pocc)\) time, where \(m\) is the length of pattern \(p\), \(\sigma\) and \(\pi\) are respectively the sizes of the alphabets \(\Sigma\) and \(\Pi\), and \(pocc\) is the number of occurrences to report [1]. She also showed an algorithm that builds the p-suffix tree for a given text \(S\) of length \(n\) in \(O(n(\pi + \log \sigma))\) time with \(O(n)\) space [2]. Later, Kosaraju [8] proposed an algorithm to build the p-suffix tree in \(O(n \log(\sigma + \pi))\) time with \(O(n)\) space. Their algorithms are both based on McCreight’s suffix tree construction algorithm [10], and hence are offline (namely, the whole text has to be known beforehand). Shibuya [15] gave an left-to-right online algorithm that builds the p-suffix tree in \(O(n \log(\sigma + \pi))\) time with \(O(n)\) space. His algorithm is based on Ukkonen’s suffix tree construction algorithm [16] which scans the input text from left to right.

Diptarama et al. [5] 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 [9] which scans the input text from left to right. Recently, Fujisato et al. [7] 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 [6] which scans the input text from right to left. Both versions of p-positions heaps support p-matching queries in \(O(m \log(\sigma + \pi) + m\pi + pocc)\) 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 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 [3] introduced the suffix tree for a CS trie which

---

\(^1\)The original claimed time bounds in Kosaraju [8] and in Shibuya [15] are \(O(n(\log \sigma + \log \pi))\). However, assuming by symmetry that \(\sigma \geq \pi\), we have \(\log \sigma + \log \pi = \log(\sigma\pi) \leq \log \sigma^2 = 2\log \sigma = O(\log \sigma)\) and \(\log(\sigma + \pi) \leq \log(2\sigma) = \log 2 + \log \sigma = O(\log \sigma)\).
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 $O(m \log \sigma + \text{occ})$ time, where $\text{occ}$ is the number of such paths to report. Shibuya [14] 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. [12] 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 $O(m \log \sigma + \text{occ})$ 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 [13].

In this paper, we propose the parameterized position heap for a CS trie $\mathcal{T}$, denoted by $\text{PPH}(\mathcal{T})$, which is the first indexing structure for p-matching on a trie. We show that $\text{PPH}(\mathcal{T})$ occupies $O(N)$ space, supports p-matching queries in $O(m \log(\sigma + \pi) + m\pi + \text{pocc})$ 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 [3] and by Fujisato et al. [7] 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 [2,8,15] in the case of a single text require somewhat involved data structures due to non-monotonicity of parameterized suffix links [11,2], 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).

2 Preliminaries

Let $\Sigma$ and $\Pi$ be disjoint ordered sets called a static alphabet and a parameterized alphabet, respectively. Let $\sigma = |\Sigma|$ and $\pi = |\Pi|$. An element of $\Sigma$ is called an s-character, and that of $\Pi$ 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 $\Sigma^*$ 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 $\varepsilon$ 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 of a (p-)string $w$ is denoted by $\text{Prefix}(w)$. The $i$-th character of a (p-)string $w$ is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a (p-)string $w$ that begins at position $i$ and ends at position $j$ is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i..j] = \varepsilon$ if $j < i$. Also, let $w[i..] = w[i..|w|]$ for any $1 \leq i \leq |w|$. For any (p-)string $w$, let $w^R$ denote the reversed string of $w$, i.e., $w^R = w[|w|] \cdots w[1]$.

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 all $1 \leq i \leq k$. For instance, let $\Sigma = \{a, b\}$ and $\Pi = \{x, y, z\}$, and consider two p-strings $x = axbzzayx$ and $y = azbyyaxz$. These two strings p-match, since $x$ can be transformed to $y$ by applying a renaming bijection $f$ such that $f(a) = a$, $f(b) = b$, $f(x) = z$, $f(y) = x$, and $f(z) = y$ to the characters in $x$. We write $x \approx y$ iff two p-strings $x$ and $y$ p-match. It is clear that $\approx$ is an equivalence relation on p-strings over $\Sigma \cup \Pi$. We denote by $[x]$ the equivalence class for p-string $x$ w.r.t. $\approx$. The representative of $[x]$ is the lexicographically smallest p-string in $[x]$, which is denoted by $\text{spe}(x)$. It is clear that two p-strings $x$ and $y$ p-match iff $\text{spe}(x) = \text{spe}(y)$. In the running example, $\text{spe}(axbzzayx) = \text{spe}(azbyyaxz) = axbyyaxz$.

A common suffix trie (CS trie) $T$ 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 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 Figure 1. CST($W$) denotes the CS trie which represents a set $W$ of (p-)strings.

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.

3.1 p-matching problem on a common suffix trie

We introduce the $p$-matching problem on a common suffix trie $T$ 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 Figure 1 when given query pattern $P = az$, 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 $p$-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 $\text{pcs}(\mathcal{T})$ of $p$-strings as follows: $\text{pcs}(\mathcal{T}) = \{ \text{spe}(w^R) | w \in W_\mathcal{T} \}$. Then, the reversed trie which we want to consider is $\text{CST}(\text{pcs}(\mathcal{T}))$. We refer to this reversed trie as the parameterized-common suffix trie of $\mathcal{T}$, and denote it by $\text{pCST}(\mathcal{T})$ (i.e., $\text{pCST}(\mathcal{T}) = \text{CST}(\text{pcs}(\mathcal{T}))$). Each node of $\text{pCST}(\mathcal{T})$ stores pointers to its corresponding node(s) of $\mathcal{T}$. Then, by solving the $p$-matching problem on $\text{pCST}(\mathcal{T})$, we can immediately answering $p$-matching queries on $\mathcal{T}$. Figure 2 shows an example of $\text{pCST}(\mathcal{T})$. In the rest of this paper, $N$ denotes the number of nodes of $\mathcal{T}$ and $N_p$ denotes the number of nodes of $\text{pCST}(\mathcal{T})$. Note that $N \geq N_p$ always holds.

### 3.2 Parameterized position heap of a common suffix trie

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

**Definition 1 (Sequence hash trees [4]).** The sequence hash tree of a sequence $S = \langle s_1, \ldots, s_k \rangle$ of strings, denoted $\text{SHT}(S) = \text{SHT}(S)^k$, is a trie structure that is recursively defined as follows: Let $\text{SHT}(S)^i = (V_i, E_i)$. Then

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

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 $u_i \notin V_{i-1}$. 

---

Figure 2: Illustration of $\text{pCST}(\mathcal{T})$ for $\mathcal{T}$ (where $\mathcal{T}$ is the common suffix trie illustrated in Figure 1). Each node of $\text{pCST}(\mathcal{T})$ corresponds to nodes of $\mathcal{T}$ which are labeled by elements in the tuple above the node of $\text{pCST}(\mathcal{T})$. For example, the node of $\text{pCST}(\mathcal{T})$ labeled 6 corresponds to the nodes of $\mathcal{T}$ labeled 11 and 14.
Note that since we have assumed that each \( s_i \in S \) 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 \( \text{SHT}(S) \) contains \( k \) nodes (including the root).

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

**Definition 2 (Parameterized positions heaps of a CST).** The parameterized position heap (p-position heap) for a common suffix trie \( T \), denoted by \( \text{PPH}(T) \), is the sequence hash tree of \( W_T \) i.e., \( \text{PPH}(T) = \text{SHT}(W_T) \).

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

**Lemma 1.** For any common suffix trie \( T \) such that the size of \( \text{pCST}(T) \) is \( N_p \), \( \text{PPH}(T) \) consists of exactly \( N_p \) nodes. Also, there is a one-to-one correspondence between the nodes of \( \text{pCST}(T) \) and the nodes of \( \text{PPH}(T) \).

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

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

Ehrenfeucht et al. [6] introduced maximal reach pointers, which are used for efficient pattern matching queries on position heaps. Diptarama et al. [5] and Fujisato et al. [7] 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 3 (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 \( \text{PPH}(T) \) such that \( v \) represents a prefix of \( \text{spe}(w_i) \).

The node which is pointed by the maximal reach pointer of node \( h_i \) is denoted by \( \text{mrp}(i) \). The augmented \( \text{PPH}(T) \) is \( \text{PPH}(T) \) with the maximal reach pointers of all nodes. For simplicity, if \( \text{mrp}(i) \) is equal to \( h_i \), then we omit this pointer. See Figure 3 for an example of augmented \( \text{PPH}(T) \).

### 3.3 P-matching with augmented parameterized position heap

It is straightforward that by applying Diptarama et al.’s pattern matching algorithm to our \( \text{PPH}(T) \) augmented with maximal reach pointers, parameterized pattern matching can be done in \( O(m \log(\sigma + \pi) + m \pi + \text{pocc'}) \) time where \( \text{pocc'} \)
Figure 3: To the left is the list of $\text{spe}(w_i)$ for $p$-strings represented by $p\text{CST}(\mathcal{T})$ of Figure 2 where $\Sigma = \{a\}$ and $\Pi = \{x, y, z\}$. To the right is an illustration for augmented $PPH(\mathcal{T})$ where the maximal reach pointers are indicated by the double-lined arrows. The underlined prefix of each $\text{spe}(w_i)$ in the left list denotes the longest prefix of $\text{spe}(w_i)$ that was represented in $PPH(\mathcal{T})$ and hence, the maximal reach pointer of the node with label $i$ points to the node which represents this underlined prefix of $\text{spe}(w_i)$.

is the number of nodes in $p\text{CST}(\mathcal{T})$ that $p$-match with the pattern. Since each node in $p\text{CST}(\mathcal{T})$ 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 [5]. These lemmas can be extended to our $PPH(\mathcal{T})$ as follows:

**Lemma 2.** Suppose $\text{spe}(p)$ is represented by a node $u$ of augmented $PPH(\mathcal{T})$. Then $p$ $p$-matches with the prefix of length $|p|$ of $w_i$ iff $\text{mrp}(i)$ is $u$ or a descendant of $u$.

**Proof.** Let $u$ be the node in augmented $PPH(\mathcal{T})$ that represents $\text{spe}(p)$.

Assume that $p$ $p$-matches with the prefix of length $|p|$ of $w_i$ and the node $v$ satisfying $id(v) = i$ represents $\text{spe}(w_i)[1..k]$. Then either $\text{spe}(w_i)[1..k]$ is a prefix of $\text{spe}(p)$ or $\text{spe}(p)$ is a prefix of $\text{spe}(w_i)[1..k]$. This implies that $v$ is either an ancestor or a descendant of $u$. If $v$ is an ancestor of $u$, then $v$ ($\text{mrp}(i)$) points to $u$ or an its descendant since $\text{spe}(w_i)[1..|p|]$ is represented by $u$. By the definition of maximal reach pointers, if $v$ is a descendant of $u$, $v$ ($\text{mrp}(i)$) points to $u$ or an its descendant.

Assume that $\text{mrp}(i)$ is $u$ or an its descendant. Let $k$ be the integer such that $\text{mrp}(i)$ represents $\text{spe}(w_i)[1..k]$. Then $\text{spe}(p)$ is a prefix of $\text{spe}(w_i)[1..k]$. This implies that $p$ $p$-matches with the prefix of length $|p|$ of $w_i$.  

**Lemma 3.** Suppose that $\text{spe}(p)$ is not represented in augmented $PPH(\mathcal{T})$. There is a factorization $q_1, \ldots, q_k$ of $p$ s.t. $q_j$ is the longest prefix of $\text{spe}(p[|q_1 \cdots q_{j-1}|+1 \cdots |p|])$.
1..|p|) that is represented in augmented \( \text{PPH}(\mathcal{T}) \). If \( p \) \( p \)-matches with the prefix of length \(|p|\) of \( w_i \), then \( \text{mrp}(i + \{q_1 \cdots q_{j-1}\}) \) is the node which represents \( \text{spe}(q_j) \) for any \( 1 \leq j < k \) and \( \text{mrp}(i + \{q_1 \cdots q_{k-1}\}) \) is the node which represents \( \text{spe}(q_k) \) or a descendant of \( \text{mrp}(i + \{q_1 \cdots q_{k-1}\}) \).

**Proof.** Assume that \( p = q_1, \ldots, q_k \) \( p \)-matches with the prefix of length \(|p|\) of \( w_i \). Since \( \text{spe}(q_1) \) is a prefix of \( \text{spe}(p) \), then \( \text{mrp}(i) \) is the node which represents \( \text{spe}(q_1) \) or an its descendant. If \( \text{mrp}(i) \) is an descendant of the node which represents \( \text{spe}(q_1) \), then \( q_1 \) is not the longest prefix of \( \text{spe}(p) \) that is represented in augmented \( \text{PPH}(\mathcal{T}) \). Thus \( \text{mrp}(i) \) is the node which represents \( \text{spe}(q_1) \). Similarly, for every \( 1 < j < k \), \( \text{spe}(q_j) \) is a prefix of \( \text{spe}(p[|q_1 \cdots q_{j-1}| + 1..|p|]) \) and \( p \)-matches with the prefix of length \(|q_j| \) of \( w_i[|q_1 \cdots q_{j-1}| + 1..|w_i|] \). Thus \( \text{mrp}(i + |q_1 \cdots q_{j-1}|) \) is the node which represents \( \text{spe}(q_j) \). Finally, \( \text{mrp}(i + |q_1 \cdots q_{k-1}|) \) has to be the node which represents \( \text{spe}(q_k) \) or an its descendant since \( q_k \) is a suffix of \( p \).

**Theorem 1.** Using our augmented \( \text{PPH}(\mathcal{T}) \), one can perform parameterized pattern matching queries in \( O(m \log(\sigma + \pi) + m\pi + \text{pocc}) \) time.

### 4 Construction of parameterized position heaps

In this section, we show how to construct the augmented \( \text{PPH}(\mathcal{T}) \) of a given common suffix trie \( \mathcal{T} \) of size \( N \). For convenience, we will sometimes identify each node \( v \) of \( \text{PPH}(\mathcal{T}) \) with the string which is represented by \( v \). In Section 4.1, we show how to compute \( \text{pCST}(\mathcal{T}) \) from a given common suffix trie \( \mathcal{T} \). In Section 4.2, we propose how to construct \( \text{PPH}(\mathcal{T}) \) from \( \text{pCST}(\mathcal{T}) \).

#### 4.1 Computing \( \text{pCST}(\mathcal{T}) \) from \( \mathcal{T} \)

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

**Lemma 4.** For any common suffix trie \( \mathcal{T} \) of size \( N \), \( \text{pCST}(\mathcal{T}) \) 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 \( \text{pCST}(\mathcal{T})^i \) \((i \leq k)\). 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 \( \text{spe}((x_j)^R)^R \) \((i.e., \; x_j[\alpha] \text{ is mapped to } \text{spe}((x_j)^R[\alpha]) \) \), and also a pointer to the corresponding node of \( \text{pCST}(\mathcal{T})^i \) \(i.e., \; \text{pointer to the node representing } \text{spe}((x_j)^R)^R \) \). 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 \( \text{spe}((x_{k+1})^R[1]) \) in \( O(\pi) \) time. Then, we can insert a new node that represents \( \text{spe}((x_{k+1})^R)^R \) as a parent of the node which represents \( \text{spe}((x_{k+1}[2..|x_{k+1}|])^R)^R \) if there does not exist such a node in \( \text{pCST}(\mathcal{T})^i \). Therefore, we can compute \( \text{pCST}(\mathcal{T}) \) in \( O(N\pi) \) time and space. \( \square \)
4.2 Computing $\text{PPH}(T)$ from $\text{pCST}(T)$

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

**Definition 4 (Reversed suffix links).** For any node $v$ of $\text{PPH}(T)$ and a character $a \in \Sigma \cup \Pi$, let

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

See Figure 4 for an example of $\text{PPH}(T)$ with reversed suffix links. In our algorithm, firstly, we insert a new node $h_i$ of $\text{PPH}(T)$ to $\text{PPH}(T)_{i-1}$. After that, we add new suffix links which point to $h_i$. When we have computed $\text{PPH}(T)$, then we compute all maximal reach pointers of $\text{PPH}(T)$.

### 4.2.1 Inserting a new node

Assume that $c_j$ (i.e., $j$-th node of $\text{pCST}(T)$) is the child of $c_i$ for any $2 \leq i \leq N_p$. Consider to insert $h_i$ (i.e., the node of $\text{PPH}(T)$ which corresponds to $c_i$) to $\text{PPH}(T)^{i-1}$. 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 5),
- $w_i[1] \in \Pi$ and $w_i[1]$ does not appear in $w_j[1..|h_j|]$ (Lemma 6),
- $w_i[1] \in \Sigma$ (Lemma 7).

**Lemma 5.** Assume that $w_i[1] \in \Pi$ appears in $w_j[1..|h_j|]$, and $a$ is the character in $\Pi$ such that $a = \text{spe}(w_j[1..|h_j|])[\alpha]$ 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 $\text{PPH}(T)^{i-1}$ 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 $\text{rsl}(a, h_k)$.
Proof. Let $\ell$ be the length of $rsl(a, h_k)$. To prove this lemma, we show that

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

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

$$rsl(a, h_k) = spe(a \cdot spe(w_j[1..\ell-1])) = spe(w_i[1] \cdot w_j[1..\ell-1]) = spe(w_i[2..\ell]) = spe(w_i[1..\ell]).$$

Thus, we have proved the first statement.

By a similar argument, we also have $spe(a \cdot spe(w_j[1..\ell])) = spe(w_i[1..\ell+1])$. Thus, if $spe(w_i[1..\ell+1])$ is represented in $PPH(T)^{i-1}$, the node representing $spe(w_j[1..\ell])$ 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 6.** Assume that $w_i[1] \in \Pi$ does not appear in $w_j[1..|h_j|]$. Let $h_k$ be the node of $PPH(T)^{i-1}$ 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 $rsl(a, h_k)$.

**Proof.** Let $\ell$ be the length of $rsl(a, h_k)$. 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

$$rsl(a, h_k) = spe(a \cdot spe(w_j[1..\ell-1])) = spe(a \cdot w_j[1..\ell-1]) = spe(w_i[1..\ell]).$$

Thus, we have proved the first statement.

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

**Lemma 7.** Assume that $w_i[1] \in \Sigma$. Let $h_k$ be the node in $PPH(T)^{i-1}$ 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 $rsl(w_i[1], h_k)$.

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

### 4.2.2 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 8.** For any node $v$ of $PPH(T)$, 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 \( \text{PPH}(T) \). Assume that \( \text{rsl}(a_1, v_1) = \text{rsl}(a_2, v_2) \) for some \( a_1, a_2 \in \Sigma \cup \Pi \) and \( v_1 \neq v_2 \) hold. By the definition of reversed suffix links, \( \text{spe}(a_1 \cdot v_1) = \text{spe}(a_2 \cdot v_2) \). Namely, \( a_1 \cdot v_1 \approx a_2 \cdot v_2 \) holds. This implies that \( v_1 \approx v_2 \), i.e., \( \text{spe}(v_1) = \text{spe}(v_2) \). Since \( v_1 \) and \( v_2 \) are node of \( \text{PPH}(T) \), \( \text{spe}(v_1) = v_1 \) and \( \text{spe}(v_2) = v_2 \) 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 \( h_j \).

4.2.3 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. [12] 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 \( h_\alpha \) holds the resulting substitutions from \( w_\alpha[1..|h_\alpha|] \) to \( \text{spe}(w_\alpha[1..|h_\alpha|]) \). 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 \( \text{pCST}(T) \). 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 \( \text{pCST}(T) \). Therefore, we have the following lemma.

**Lemma 9.** We can compute \( \text{PPH}(T) \) from \( \text{pCST}(T) \) of size \( N_p \) in \( O(N_p(\sigma + \pi)) \) time and space.

Therefore, we can obtain the following result by Lemmas 4 and 9.

**Theorem 2.** We can compute \( \text{PPH}(T) \) of a given common suffix trie \( T \) of size \( N \) in \( O(N(\sigma + \pi)) \) time and space.

Since we can also compute all maximal reach pointers of \( \text{PPH}(T) \) efficiently in a similar way to [12] (this algorithm is also similar to suffix link construction), we also have the following lemma.

**Lemma 10.** We can compute all the maximal reach pointers for \( \text{PPH}(T) \) in \( O(N_p(\sigma + \pi)) \) time and space.

Hence, we can get the following result.

**Theorem 3.** We can compute the augmented \( \text{PPH}(T) \) of a given common suffix trie \( T \) of size \( N \) in \( O(N(\sigma + \pi)) \) time and space.

5 Conclusions and open problems

This paper proposed the p-position heap for a CS trie \( T \), denoted \( \text{PPH}(T) \), which is the first indexing structure for the p-matching problem on a trie. The key idea is to transform the input CS trie \( T \) into a parameterized CS
trie $\text{pCST}(T)$ where p-matching suffixes are merged. We showed that the p-matching problem on the CS trie $T$ can be reduced to the p-matching problem on the parameterized CS trie $\text{pCST}(T)$. We proposed an algorithm which constructs $\text{PPH}(T)$ in $O(N(\sigma + \pi))$ time and working space, where $N$ is the size of the CS trie $T$. We also showed that using $\text{PPH}(P)$ one can solve the p-matching problem on the CS trie $T$ in $O(m \log(\sigma + \pi) + m\pi + pocc)$ time, where $m$ is the length of a query pattern and $pocc$ 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 $\text{PPH}(T)$ is shorter the pattern length $m$ and thus the pattern needs to be partitioned into $O(\pi)$ blocks in the current pattern matching algorithm [5].

- Can we efficiently build the p-suffix tree for a CS trie? It is noted by Baker [1, 2] 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 [3] which is based on Weiner’s algorithm [17], 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.

References

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

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

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

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

[5] 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.

[6] 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.
[7] 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.

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

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

[10] E. M. McCreight. A space-economical suffix tree construction algorithm. Journal of ACM, 23(2):262–272, 1976.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249–260, 1995.

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