

An Extension of Linear-size Suffix Tries for Parameterized Strings

Katsuhito Nakashima, Diptarama Hendrian, Ryo Yoshinaka, and
Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University, Japan
katsuhito_nakashima@shino.ecei.tohoku.ac.jp
{diptarama,ryoshinaka,ayumis}@tohoku.ac.jp

Abstract. In this paper, we propose a new indexing structure for parameterized strings which we call *PLSTs*, by generalizing linear-size suffix tries for ordinary strings. Two parameterized strings are said to match if there is a bijection on the symbol set that makes the two coincide. PLSTs are applicable to the parameterized pattern matching problem, which is to decide whether the input parameterized text has a substring that matches the input parameterized pattern. The size of PLSTs is linear in the text size, with which our algorithm solves the parameterized pattern matching problem in linear time in the pattern size. PLSTs can be seen as a compacted version of parameterized suffix tries and a combination of linear-size suffix tries and parameterized suffix trees. We experimentally show that PLSTs are more space efficient than parameterized suffix trees for highly repetitive strings.

Keywords: indexing data structure · linear-size suffix trie · parameterized pattern matching.

1 Introduction

The pattern matching problem is to check whether a pattern string occurs in a text string or not. To efficiently solve the pattern matching problem, a numerous number of text indexing structures have been proposed. Suffix trees are the most widely used data structures and provide many applications including several variants of pattern matching problems [5,10]. They can be seen as a compacted type of suffix tries, where two branching nodes that have no other branching nodes between them in a suffix trie are directly connected in the suffix tree. The new edges have a reference to an interval of the text so that the original path label of the suffix trie can be recovered. Recently, Crochemore et al. [4] proposed a new indexing structure, called a *linear-size suffix trie (LST)*, which is another compacted variant of a suffix trie. An LST replaces paths consisting only of

This research was partially supported by JSPS KAKENHI Grant Numbers JP15H05706 and JP19K20208.

Copyright 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

non-branching nodes by edges like a suffix tree, but the original path labels are recovered by referring to other edge labels in the LST itself unlike suffix trees. LSTs use less memory space than suffix trees for indexing the same highly repetitive strings [14]. LSTs may be used as an alternative of suffix trees for various applications, like computing the longest common substrings, not limited to the pattern matching problem.

On the other hand, different types of pattern matching have been proposed and intensively studied. The variant this paper is concerned with is the *parameterized pattern matching problem*, introduced by Baker [3]. Considering two disjoint sets of symbols Σ and Π , we call a string over $\Sigma \cup \Pi$ a *parameterized string* (*p-string*). In the parameterized pattern matching problem, given p-strings T and P , we must check whether substrings of T can be transformed into P by applying a one-to-one function that renames symbols in Π . The parameterized pattern matching is motivated by applying to the software maintenance [2,3], the plagiarism detection [8], the analysis of gene structure [13], and so on. Similarly to the basic string matching problem, several indexing structures that support the parameterized pattern matching have been proposed, such as parameterized suffix trees [3], structural suffix trees [13], parameterized suffix arrays [6,12], and parameterized position heaps [7,9].

In this paper, we propose a new indexing structure for p-strings, which we call *PLST*. A PLST is a tree structure that combines a linear-size suffix trie and a parameterized suffix tree. We show that the size of a PLST is $O(n)$ and give an algorithm for the parameterized pattern matching problem for given a pattern and a PLST, to find the occurrences of a pattern in the text, that runs in $O(m)$ time, where n is the length of the text and m is the length of the pattern. Furthermore, we experimentally show that PLSTs are more space efficient than parameterized suffix trees for highly repetitive strings such as Fibonacci strings.

2 Preliminaries

2.1 Basic definitions and notation

We denote the set of all non-negative integers by \mathcal{N} . Let Δ be an alphabet. For a string $w = xyz \in \Delta^*$, x , y , and z are called *prefix*, *substring*, and *suffix* of w , respectively. The length of w is denoted by $|w|$ and the i -th symbol of w is denoted by $w[i]$ for $1 \leq i \leq |w|$. The substring of 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, 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 $i > j$. For a string u and an extension uv , we write $\text{str}(u, uv) = v$.

Throughout this paper, we fix two alphabets Σ and Π . We call elements of Σ *constant* symbols and those of Π *parameter* symbols. An element of Σ^* is called a *constant string* and that of $(\Sigma \cup \Pi)^*$ is called a *parameterized string*, or *p-string* for short. We assume that the size of Σ and Π are constant.

Given two p-strings w_1 and w_2 of length n , w_1 and w_2 are a *parameterized match* (*p-match*), denoted by $w_1 \approx w_2$, if there is a bijection f on $\Sigma \cup \Pi$ such

that $f(a) = a$ for any $a \in \Sigma$ and $f(w_1[i]) = w_2[i]$ for all $1 \leq i \leq n$ [3]. We can determine whether $w_1 \approx w_2$ or not by using an encoding called *prev-encoding* defined as follows.

Definition 1 (Prev-encoding [3]). For a p -string w of length n over $\Sigma \cup \Pi$, the *prev-encoding* for w , denoted by $\text{prev}(w)$, is defined to be a string over $\Sigma \cup \mathcal{N}$ of length n such that for each $1 \leq i \leq n$,

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

We call strings over $\Sigma \cup \mathcal{N}$ *pv-strings*.

For any p -strings w_1 and w_2 , $w_1 \approx w_2$ if and only if $\text{prev}(w_1) = \text{prev}(w_2)$. For example, given $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and $\Pi = \{\mathbf{u}, \mathbf{v}, \mathbf{x}, \mathbf{y}\}$, $s_1 = \mathbf{uvvvaauvb}$ and $s_2 = \mathbf{xyyaxxyb}$ are p -matches by f such that $f(\mathbf{u}) = \mathbf{x}$ and $f(\mathbf{v}) = \mathbf{y}$, where $\text{prev}(s_1) = \text{prev}(s_2) = 0011\mathbf{a}514\mathbf{b}$.

We define *parameterized pattern matching* as follows.

Definition 2 (Parameterized pattern matching [3]). Given two p -strings, text T and pattern P , decide whether T has a substring that p -matches P .

For example, considering a text $T = \mathbf{auvaubauvbv}$ and a pattern $P = \mathbf{xayby}$ over $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and $\Pi = \{\mathbf{u}, \mathbf{v}, \mathbf{x}, \mathbf{y}\}$, T has two substrings $T[3 : 7] = \mathbf{vaub}$ and $T[7 : 11] = \mathbf{uavbv}$ that p -match P .

Throughout this paper, we assume that a text T ends with a sentinel symbol $\$ \in \Sigma$, which occurs nowhere else in T .

2.2 Suffix tries, suffix trees, and linear-size suffix tries

This subsection briefly reviews tree structures for indexing all the substrings of a constant string $T \in \Sigma^*$.

The suffix trie $\text{STrie}(T)$ is a tree with nodes corresponding to all the substrings of T . Figure 1 (a) shows an example of a suffix trie. Throughout this paper, we identify a node with its corresponding string for explanatory convenience. Note that each node does not explicitly remember its corresponding string. For each nonempty substring ua of T where $a \in \Sigma$, we have an edge from u to ua labeled with a . Then by reading the labels on the path from the root to a node u , one can obtain the string u the node corresponds. Then the path label from the node u to a descendant uv is $\text{str}(u, uv) = v$ for $u, v \in \Sigma^*$. Since there are $\Theta(|T|^2)$ substrings of T , the size of $\text{STrie}(T)$ is $\Theta(|T|^2)$.

The suffix tree $\text{STree}(T)$ is a tree obtained from $\text{STrie}(T)$ by removing all non-branching internal nodes and replacing each path with no branching nodes by a single edge whose label refers to a corresponding interval of the text T . That is, the label on the edge (u, v) is a pair (i, j) such that $T[i : j] = \text{str}(u, v)$.

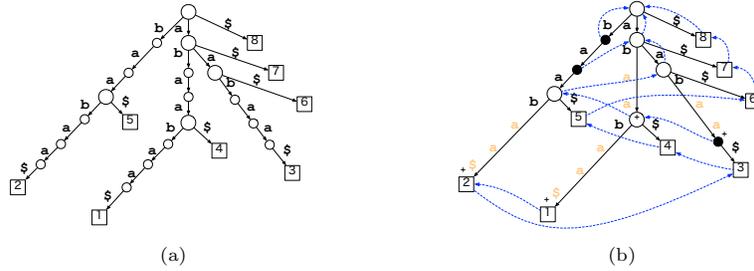


Fig. 1. (a) The suffix trie for $T = abaabaa\$$. (b) The LST for T . Solid and broken arrows represent the edges and suffix links, respectively. The LST keeps the first symbol (black) on each edge, while the succeeding symbols (orange) are discarded. Big white and small black circles represent nodes of Type 1 and Type 2, respectively. The ‘+’ signs represent the 1-bit flag. If a node v has ‘+’ sign, the edge (u, v) has a path label of length greater than 1 in $STrie(T)$ where u is the parent node of v in $LST(T)$.

Since there are at most $O(|T|)$ branching nodes, the size of $STree(T)$ is $\Theta(|T|)$. An important auxiliary map on nodes is called *suffix links*, denoted by SL , which is defined by $SL(aw) = w$ for each node aw with $a \in \Sigma$ and $w \in \Sigma^*$.

The *linear-size suffix trie (LST)* [4] $LST(T)$ of a string T is another compact variant of a suffix trie (see Figure 1 (b)). An LST suppresses (most) non-branching nodes and replaces paths with edges like a suffix tree, but the labels of those new edges do not refer to intervals of the input text. Each edge (u, v) retains only the first symbol $str(u, v)[1]$ of the original path label $str(u, v)$. To recover the original label $str(u, v)$, we refer to another edge or a path in the LST itself following a suffix link, using the fact that $str(u, v) = str(SL(u), SL(v))$. The reference will be recursive, but eventually one can regain the original path label by collecting those retained symbols. For this sake, $LST(T)$ keeps some non-branching internal nodes from $STrie(T)$ and thus it may have more nodes than $STree(T)$, but still the size is linear in $|T|$. The nodes of $LST(T)$ consist of those of $STree(T)$ and non-branching node whose suffix links point at a branching node. We call the former *Type 1* and the latter *Type 2*. Each edge (u, v) has a 1-bit flag that tells whether $|v| - |u| = 1$. If it is the case, one knows the complete label $str(u, v) = str(u, v)[1]$. Otherwise, one needs to follow the suffix link to regain the other symbols. An LST uses suffix links to regain the original path label in the suffix trie. If we had only Type 1 nodes, for some edge (u, v) , there may be a branching node between $SL(u)$ and $SL(v)$, which makes it difficult to regain the original path label. Having Type 2 nodes, there is no branching node between $SL(u)$ and $SL(v)$ for every edge (u, v) . Then it is enough to go straight down from $SL(u)$ to regain the original path label.

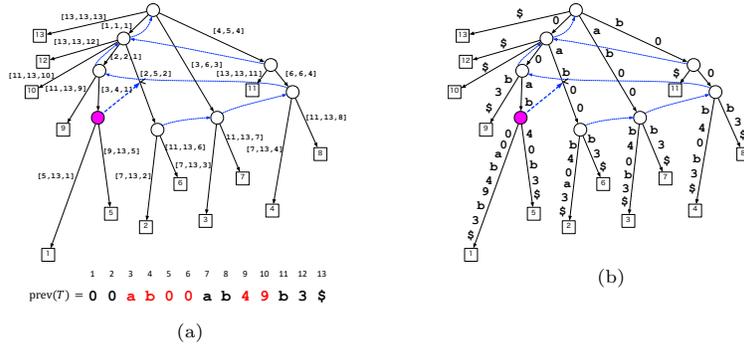


Fig. 2. (a) The parameterized suffix tree $\text{PSTree}(T)$ for $T = \text{xyabzwbabzxbz}\$$ where $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{\$}\}$ and $\Pi = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w}\}$. (b) The referenced substrings are shown on edges. Broken blue arrows denote suffix links. Some suffix links do not point to a branching node shown with a bold broken arrow.

2.3 Parameterized suffix tries and parameterized suffix trees

For a p-string $T \in (\Sigma \cup \Pi)^*$, a *prev-encoded substring* (*pv-substring*) of T is the prev-encoding $\text{prev}(w)$ of a substring w of T . The set of pv-substrings of T is denoted by $\text{PrevSub}(T)$.

A parameterized suffix trie of T , denoted by $\text{PSTrie}(T)$, is the trie that represents all the pv-substrings of T . The size of $\text{PSTrie}(T)$ is $\Theta(|T|^2)$.

For a pv-string $u \in (\Sigma \cup \mathcal{N})^*$, the *k-re-encoding* for u , denoted by $\langle u \rangle_k$, is defined to be the pv-string of length $|u|$ such that for each $1 \leq i \leq |u|$,

$$\langle u \rangle_k[i] = \begin{cases} 0 & \text{if } u[i] \in \mathcal{N} \text{ and } u[i] \geq i - k + 1, \\ u[i] & \text{otherwise.} \end{cases}$$

When $k = 1$, we omit k . We then have $\langle \text{prev}(w)[i : j] \rangle = \text{prev}(w[i : j])$ for any p-string $w \in (\Sigma \cup \Pi)^*$ and $i, j \leq |w|$.

Usually suffix links are defined on nodes of suffix trees, but it is convenient to have “implicit suffix links” on all nodes except the root of $\text{STrie}(T)$, i.e., all the nonempty substrings of T , as well. For a nonempty pv-string $u \in (\Sigma \cup \mathcal{N})^+$, let $\text{sl}(u)$ denote the re-encoding $\langle u[2 :] \rangle$ of the string obtained by deleting the first symbol. This operation on strings will define real suffix links in indexing structures for parameterized strings based on parameterized suffix tries. Differently from constant strings, $u \in \text{PrevSub}(T)$ does not necessarily imply $u[2 :] \in \text{PrevSub}(T)$. What we actually have is $\text{sl}(u) = \langle u[2 :] \rangle \in \text{PrevSub}(T)$.

A *parameterized suffix tree* (*p-suffix tree*) [3] of T , denoted by $\text{PSTree}(T)$, is a compacted variant of the parameterized suffix trie. Figure 2 shows an example of a p-suffix tree. Like the suffix tree for a constant string over Σ , $\text{PSTree}(T)$ is obtained from $\text{PSTrie}(T)$ by removing non-branching internal nodes and giving each edge as a label a reference to some interval of the prev-encoded text $\text{prev}(T)$. The reference is represented by a triple (i, j, k) of a text start position, end position, and suffix number, which refers to the pv-string $\langle \text{prev}(T)[k :] \rangle[i : j]$.

3 PLSTs

We now introduce our indexing tree structures for p-strings, which we call *PLSTs*, based on LSTs and p-suffix trees reviewed in Sections 2.2 and 2.3. There are two difficulties in extending LSTs to deal with p-strings. Figure 3(a) shows the LST-like structure obtained from $\text{PSTrie}(T)$ in the same way as $\text{LST}(T)$ is obtained from $\text{STrie}(T)$. We want to know $\text{str}(u, v)$ for an edge (u, v) by “reduction by suffix links”, but

1. it is *not* necessarily that $\text{str}(u, v) = \text{str}(\text{sl}(u), \text{sl}(v))$,
2. there can be a branching node u of $\text{PSTrie}(T)$ such that $\text{sl}(u)$ is not branching.

An example edge (u, v) exhibiting the first difficulty consists of $u = 00$ and $v = 00\mathbf{b}3\mathbf{\$}$, where $\text{str}(u, v) = \mathbf{b}3\mathbf{\$}$ but $\text{str}(\text{sl}(u), \text{sl}(v)) = \mathbf{b}0\mathbf{\$}$. This is caused by the fact that $\text{sl}(u) = \langle u[2 :] \rangle$ rather than $\text{sl}(u) = u[2 :]$. Then, the path label $\text{str}(\text{sl}(u), \text{sl}(v))$ referenced by the suffix link may not give exactly what we want. We solve this problem by giving the node v a “re-encoding sign” with which one can recover $\text{str}(u, v)$ from $\text{str}(\text{sl}(u), \text{sl}(v))$. An example node for the second case is $00\mathbf{ab}$. This is a branching node but $\langle \text{sl}(00\mathbf{ab}) \rangle = 0\mathbf{ab}$ does not appear as a node. To handle this case, we simply refer to the corresponding interval of the original text T by keeping the necessary subsequence, where, as we will observe in experiments, the necessary subsequence tends to be rather small. Our proposed structure PLST is shown in Figure 3(b). In what follows we explain PLSTs.

3.1 Definition and properties of PLSTs

Let $U = \text{PrevSub}(T)$ be the set of nodes of $\text{PSTrie}(T)$. The set V of nodes of the PLST $\text{PLST}(T)$ for T is a subset of U , which is partitioned as $V = V_1 \cup V_2 \subseteq U$. Nodes in V_i are called *Type i* for $i = 1, 2$. The definition of Type 1 and Type 2 nodes follows the one for original LSTs [4].

1. A node $u \in U$ is Type 1 if u is a leaf or a branching node in $\text{PSTrie}(T)$.
2. A node $u \in U$ is Type 2 if $u \notin V_1$ and $\text{sl}(u) \in V_1$.

Edges of $\text{PLST}(T)$ are trivially determined: we have $(u, uv) \in V \times V$ as an edge if and only if $v \neq \varepsilon$ and there is no proper nonempty prefix v' of v such that $uv' \in V$. We will show in Section 3.3 that $|V| \in O(|T|)$. We say that $u \in V$ is *good* if $\text{sl}(u) \in V$, and $u \in V$ is *bad* otherwise. Note that any $u \in V_2$ is good by the definition of V_2 , and that the root ε is bad.

To obtain $\text{str}(u, v)$ for an edge (u, v) , if $|v| - |u| = 1$, we simply read the edge label $v[1]$ like an LST. Otherwise, if both u and v are good, we basically use the technique of “reduction by suffix links”. An important observation is that the equation $\text{str}(u, v) = \text{str}(\text{sl}(u), \text{sl}(v))$, which was a key property to regain the original label in LSTs, does not necessarily hold for PLSTs. Figure 4 shows an example, where $\text{str}(u, v) = \mathbf{cb}40 \neq \mathbf{cb}00 = \text{str}(\text{sl}(u), \text{sl}(v))$; the third symbol 4 in $\text{str}(u, v)$ is re-encoded to 0 in $\text{str}(\text{sl}(u), \text{sl}(v))$, because the first symbol $v[1] = 0$ of v , that is referenced by the symbol 4, is cut out in $\text{sl}(v)$. Fortunately, the possible difference between $\text{str}(u, v)$ and $\text{str}(\text{sl}(u), \text{sl}(v))$ is limited.

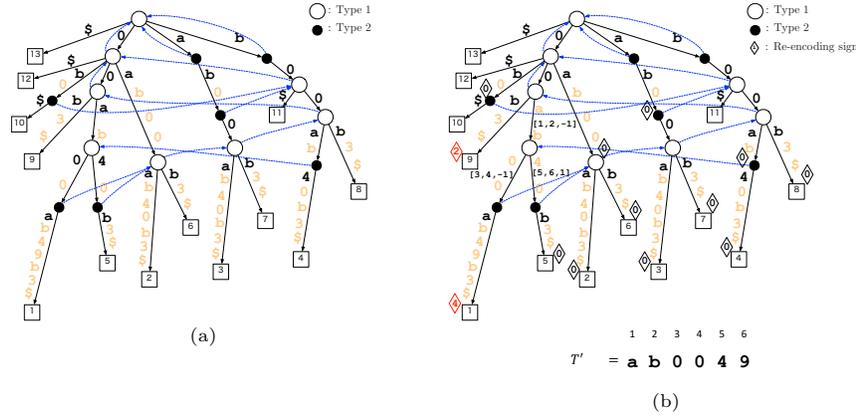


Fig. 3. (a) LST-like structure for $\text{PrevSub}(T)$ and (b) the PLST $\text{PLST}(T)$ for $T = \text{xyabzwbabzxbz}\$$ where $\Sigma = \{a, b, \$\}$ and $\Pi = \{x, y, z, w\}$. The triples of integers are reference to the text. White and black circles represent nodes of Type 1 and 2, respectively. The numbers in rhombus represent re-encoding signs. The PLST keeps only the first symbol (black) on each edge, while the succeeding symbols (orange) are discarded. The reference text $T' = \mathbf{ab0049}$ is shorter than the original T .

Observation 1. Any prev-encoded substring v of text T has at most one position i such that $v[i] = i - 1$. For such a position i , we have $\text{sl}(v)[i - 1] = 0$ and for any $j \in \{2, \dots, |v|\} \setminus \{i\}$, $\text{sl}(v)[j - 1] = v[j]$. Thus, such a position is unique in $\text{str}(u, v)$ for each edge (u, v) in $\text{PLST}(T)$.

For each edge (u, v) , we associate an integer named *re-encoding sign*, so that we can regain $\text{str}(u, v)$ from $\text{str}(\text{sl}(u), \text{sl}(v))$ as follows.

Definition 3 (Re-encoding sign). For each node $v \in V$, let u be the parent of v . We define re-encoding sign to v by

$$\text{Re}(v) = \begin{cases} i - |u| & \text{if there exists } i \text{ such that } v[i] = i - 1 \text{ and } |u| < i \leq |v|, \\ 0 & \text{otherwise.} \end{cases}$$

The re-encoding sign $\text{Re}(v)$ is well-defined by Observation 1. Figure 4 shows an example of re-encoding signs. The next lemma immediately follows from Observation 1 and Definition 3.

Lemma 1. Let (u, v) be an edge in $\text{PLST}(T)$ such that both u and v are good. Then for any $i \in \{1, \dots, |\text{str}(u, v)|\} \setminus \{\text{Re}(v)\}$, $\text{str}(u, v)[i] = \text{str}(\text{sl}(u), \text{sl}(v))[i]$. If $\text{Re}(v) \geq 1$, then $\text{str}(u, v)[\text{Re}(v)] = |u| + \text{Re}(v) - 1$ and $\text{str}(\text{sl}(u), \text{sl}(v))[\text{Re}(v)] = 0$.

Lemma 1 tells how to recover $\text{str}(u, v)$ from $\text{str}(\text{sl}(u), \text{sl}(v))$ using the re-encoding sign at v and the depth $|u|$ of u . Note that the depth $|u|$ is the depth of u in parameterized suffix tries, not the number of nodes from the root to u in PLSTs.

If either u or v is bad in an edge (u, v) with $|v| - |u| \geq 2$, we give up “reduction by suffix links” and simply label the edge with the reference to the corresponding

least extension v' of v such that $v' \in V$. In other words, p is a prefix of $\text{str}(u, v')$, where v' should be v itself if $v \in V$. Otherwise, it returns null.

For an input pair (p, u) , if $p = \varepsilon$, then P-MATCH returns u , as it is required. Otherwise, it first tries to regain $\text{str}(u, v)$ for the $p[1]$ -child v of u , if u has such a child. At first, suppose $|p| \geq |v| - |u| = l$. We would like to know whether $p[1 : l] = \text{str}(u, v)$. If $l = 1$, it means that we have already confirmed that $p[1 : l] = \text{str}(u, v)$. Then we just go down to v and recursively call P-MATCH($p[2 :], v$). If $l \geq 2$ and either u or v is bad, we refer to T' and check if $p[1 : l] = \text{str}(u, v)$ as with matching in a p-suffix tree. If $l \geq 2$ and both u and v are good, we cannot know from the edge (u, v) itself what $\text{str}(u, v)$ is except for its first symbol $\text{str}(u, v)[1] = p[1]$. To recover whole $\text{str}(u, v)$, we use the suffix link of u . Since u is good, $\text{SL}(u)$ is defined. If $\text{Re}(v) = 0$, we have $\text{str}(u, v) = \text{str}(\text{sl}(u), \text{sl}(v))$ by Lemma 1, and we simply call P-MATCH($p[1 : l], \text{SL}(u)$). Otherwise, we have $p[1 : l] = \text{str}(u, v)$ if and only if $p[\text{Re}(v)] = |u| + \text{Re}(v) - 1$ and $p'[1 : l] = \text{str}(\text{sl}(u), \text{sl}(v))$, where

$$p'[i] = \begin{cases} 0 & \text{if } i = \text{Re}(v), \\ p[i] & \text{otherwise} \end{cases}$$

for $i = 1, \dots, |p|$. Thus, the recursive call of P-MATCH($p'[1 : l], \text{SL}(u)$) returns null iff $p[1 : l] \neq \text{str}(u, v)$. If P-MATCH($p'[1 : l], \text{SL}(u)$) returns a node, then $p[1 : l] = \text{str}(u, v)$ and thus we continue matching by calling P-MATCH($p[l + 1 :], v$).

The above discussion is valid when $|p| \leq |v| - |u|$. If $\text{Re}(v) = 0$ or $\text{Re}(v) > |p|$, then p is a prefix of $\text{str}(u, v)$ iff p is a prefix of $\text{str}(\text{sl}(u), \text{sl}(v))$. Otherwise, p is a prefix of $\text{str}(u, v)$ iff $p[\text{Re}(v)] = |u| + \text{Re}(v) - 1$ and p' is a prefix of $\text{str}(\text{sl}(u), \text{sl}(v))$. Thus the recursion is justified. If P-MATCH($p'[1 : l], \text{SL}(u)$) returns a node, p is a prefix of $\text{str}(u, v)$ and we call P-MATCH(ε, v), which returns v .

Proposition 1. *We can decide whether T has a substring that p -matches P using Algorithm 1.*

The time complexity of Algorithm 1 is not linear as it is. Suppose that P-MATCH(p, u) is called. It can be the case $|v| - |u| \geq |p| \geq 2$ and either $\text{Re}(v) = 0$ or $\text{Re}(v) > l$ where $v = \text{child}(u, p[1])$. In this case, the algorithm simply calls P-MATCH($p, \text{SL}(u)$), where the first argument has not changed from the preceding call. Such recursion may be repeated, and amortized time complexity is not linear. The same difficulty and a solution have already been discussed by Crochemore et al. [4] for LSTs. Following them, we introduce *fast links* as follows, which allow us to skip recursions that always preserve the first argument.

Definition 4 (Fast link). *For each edge $(u, v) \in V \times V$ such that $|v| - |u| > 1$ and both u, v are good, the fast link for (u, v) is defined to be $\text{FL}(u, v) = \text{SL}^k(u)$ where $k \geq 1$ is the smallest integer satisfying either $|v_k| < |v| - k$ or $0 < \text{Re}(v_k)$, where $v_k = \text{child}(\text{SL}^k(u), a)$ for $a = \text{str}(u, v)[1]$.*

Algorithm 1 will run in linear time by replacing $\text{SL}(u)$ in Line 14 by $\text{FL}(u, v)$. If $|v_k| < |v| - k = |\text{sl}^k(v)|$, the node v_k occurs between $\text{sl}^k(u)$ and $\text{sl}^k(v)$. Then, P-MATCH($p, \text{SL}^k(u)$) will call P-MATCH($p[1 : |v_k| - |\text{SL}^k(u)|], \text{SL}^{k+1}(u)$). When

Algorithm 1: P-MATCH(p, u)

Input: A string p and a node u in PLST(T)
Output: The highest descendant v of u such that p is a prefix of $\text{str}(u, v)$

```

1 if  $p = \varepsilon$  then return  $u$ ;
2 else
3   if  $\text{child}(u, p[1])$  is undefined then return null;
4   else
5      $v \leftarrow \text{child}(u, p[1])$ ;
6      $l \leftarrow \min\{|p|, |v| - |u|\}$ ;
7     if  $l \geq 2$  and  $u$  or  $v$  is bad then
8       let  $\alpha$  be the label of the edge  $(u, v)$ ;
9       if  $p[:l] \neq \alpha[:l]$  then return null;
10    else if  $l \geq 2$  then
11      if  $1 \leq \text{Re}(v) \leq |p|$  then
12        if  $p[\text{Re}(v)] = |u| + \text{Re}(v) - 1$  then  $p[\text{Re}(v)] \leftarrow 0$ ;
13        else return null;
14      if P-MATCH( $p[1:l], \text{SL}(u)$ ) = null then return null;
15    return P-MATCH( $p[l+1:], v$ );

```

$0 < \text{Re}(v_k)$, we change the $\text{Re}(v_k)$ -th symbol of p , which must be a positive integer, to 0. Therefore, the number of fast links we follow is bounded by $2|p|$. Figure 5 shows how to p-match $\text{str}(u, v)$ and $p = \mathbf{a04b}$ using fast links. We know that $p[1] = \text{str}(u, v)[1] = \mathbf{a}$. After following the fast link (1), we check whether $p[\text{Re}(\text{sl}^2(v))] = 4$ and rewrite the value of $p[\text{Re}(\text{sl}^2(v))]$ to 0. After using (2), we check whether $p[3] = 0$. In this way, we can know that p matches $\text{str}(u, v)$.

Theorem 1. *Given PLST(T) and a pattern P of length m , we can decide whether T has a substring that p -matches P in $O(m)$ time.*

3.3 The size of PLSTs

We now show that the size of PLST(T) is linear with respect to the length n of a text T . First, we show a linear upper bound on the number of nodes of PLST(T). The nodes of Type 1 appear in the p-suffix tree, so they are at most $2n$ [3]. It is enough to show that the number of nodes of Type 2 is linearly bounded as well.

Lemma 2. *The number of Type 2 nodes in PLST(T) is smaller than $2n$.*

Proof. Let us consider an implicit suffix link chain in PSTrie(T) starting from $w = \text{prev}(T[:k])$ with $1 \leq k < n$, i.e., $(w, \text{sl}(w), \text{sl}^2(w), \dots, \text{sl}^{|w|}(w))$. PSTrie(T) has $n - 1$ such chains and each internal node of PLST(T) appears in at least one chain. If a chain has two distinct Type 2 nodes $\text{sl}^i(w)$ and $\text{sl}^j(w)$ with $i < j$, since $\text{sl}^{i+1}(w)$ is Type 1 by definition, there exists a Type 1 node between them.

Define a binary relation R between V_1 and V_2 by

$$R = \{ (u, v) \in V_1 \times V_2 \mid \text{there is } i \text{ s.t. } v = \text{sl}^i(u) \text{ and } \text{sl}^j(u) \notin V_2 \text{ for all } j < i \}$$

and let $R_2 = \{v \in V_2 \mid (u, v) \in R \text{ for some } u \in V_1\}$. Since R is a partial function from branching nodes to Type 2 nodes, we have $|R_2| \leq n$. By the above argument on a chain, each chain has at most one Type 2 node $v \in V_2$ such that $v \notin R_2$. Since there are $n - 1$ chains, we have $|V_2 \setminus R_2| < n$. All in all, $|V_2| = |R_2| + |V_2 \setminus R_2| < n + n = 2n$. \square

The number of edges and their labels, as well as the number of suffix links, depth and re-encoding sign for nodes, is asymptotically bounded above by the number of nodes in $\text{PLST}(T)$. T' is a subsequence of $\text{prev}(T)$, thus its length is $O(n)$. Therefore, the size of $\text{PLST}(T)$ is $O(n)$.

Theorem 2. *Given a p-string T of length n , the size of $\text{PLST}(T)$ is $O(n)$.*

4 Experiments

We performed comparative experiments on the number of nodes of PLSTs and p-suffix trees for four sorts of text strings changing their length. Text strings we used are random strings of length $n = 10, \dots, 10240$ over a constant alphabet Σ with $|\Sigma| = 2$ and those over a parameter alphabet Π with $|\Pi| = 2$, and the 11th through 22nd Fibonacci strings over Σ with $|\Sigma| = 2$ and those over Π with $|\Pi| = 2$. PLSTs for constant strings are of course identical to LSTs. For random strings, we measured the average number of nodes for 100 strings of each length. The results of our experiments are shown in Table 1. Recall that p-suffix trees consist of Type 1 nodes and $\text{prev}(T)$, while PLSTs have Type 1 and Type 2 nodes and T' . For random strings, we can see that the number of Type 2 nodes is close to the text length. Since a node requires much more memory size than a character, PLSTs will be bigger than p-suffix trees for random strings. On the other hand, for Fibonacci strings, PLSTs have few Type 2 nodes and $T' = \epsilon$. Thus, PLSTs will use less space than p-suffix trees for Fibonacci strings.

In addition to Fibonacci strings, we constructed PLSTs for other kinds of highly repetitive strings such as period-doubling and Thue-Morse strings [1]. We confirmed that those PLSTs have few Type 2 nodes and $T' = \epsilon$. PLSTs for such highly repetitive strings use less memory than p-suffix trees.

5 Conclusion and future work

In this paper, we presented an indexing structure called a PLST for the parameterized pattern matching problem. Given a p-string T of length n , the size of PLST for T is $O(n)$. We presented an algorithm that solves the problem in $O(m)$ time, where m is the length of the pattern. We experimentally showed that PLST is space-saving from p-suffix tree for indexing highly repetitive strings.

For PLSTs to be useful for various applications, like computing the longest common substrings, an efficient algorithm for constructing PLSTs is required like LSTs [11]. Furthermore, the ideas developed in this paper may be useful to generalize L-CDAWGs [14] to a data structure for parameterized strings.

Table 1. The numbers of nodes of PLSTs for different sorts of strings

length n	random strings			
	constant string		p-string	
	Type 1	Type 2	Type 1	Type 2
10	16.98	6.04	16.93	5.23
20	35.66	12.78	35.72	12.27
40	74.58	27.25	74.53	26.22
80	153.61	56.82	153.48	56.04
160	312.37	115.55	312.45	115.24
320	631.40	234.55	631.27	235.32
640	1270.34	477.29	1270.47	475.34
1280	2549.35	956.18	2549.39	957.03
2560	5108.37	1923.62	5108.48	1922.97
5120	10227.48	3845.35	10227.29	3853.97
10240	20466.49	7710.50	20466.14	7704.25

length n	Fibonacci strings			
	constant string		p-string	
	Type 1	Type 2	Type 1	Type 2
90	178	12	177	12
145	285	12	285	13
234	466	15	465	15
378	751	15	751	16
611	1220	18	1219	18
988	1971	18	1971	19
1598	3194	21	3193	21
2585	5165	21	5165	22
4182	8362	24	8361	24
6766	13527	24	13527	25
10947	21892	27	21918	27

References

1. Allouche, J.P., Shallit, J.: Automatic sequences: theory, applications, generalizations. Cambridge university press (2003)
2. Baker, B.S.: A program for identifying duplicated code. *Computing Science and Statistics* **24**, 49–57 (1992)
3. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences* **52**(1), 28–42 (1996)
4. Crochemore, M., Epifanio, C., Grossi, R., Mignosi, F.: Linear-size suffix tries. *Theoretical Computer Science* **638**, 171–178 (2016)
5. Crochemore, M., Rytter, W.: *Jewels of Stringology: Text Algorithms*. World Scientific (2003)
6. Deguchi, S., Higashijima, F., Bannai, H., Inenaga, S., Takeda, M.: Parameterized suffix arrays for binary strings. In: *Proceedings of the Prague Stringology Conference 2008*. pp. 84–94 (2008)
7. Diptarama, Katsura, T., Otomo, Y., Narisawa, K., Shinohara, A.: Position heaps for parameterized strings. In: *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*. pp. 8:1–8:13 (2017)
8. Fredriksson, K., Mozgovoy, M.: Efficient parameterized string matching. *Information Processing Letters* **100**(3), 91–96 (2006)
9. Fujisato, N., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Right-to-left online construction of parameterized position heaps. In: *Prague Stringology Conference 2018*. pp. 91–102 (2018)
10. Gusfield, D.: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press (1997)
11. Hendrian, D., Takagi, T., Inenaga, S.: Online Algorithms for Constructing Linear-Size Suffix Trie. In: *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. pp. 30:1–30:19 (2019)
12. I, T., Deguchi, S., Bannai, H., Inenaga, S., Takeda, M.: Lightweight parameterized suffix array construction. In: *Combinatorial Algorithms (IWOCA 2009)*. pp. 312–323 (2009)
13. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica* **39**(1), 1–19 (2004)
14. Takagi, T., Goto, K., Fujishige, Y., Inenaga, S., Arimura, H.: Linear-size cdawg: new repetition-aware indexing and grammar compression. In: *International Symposium on String Processing and Information Retrieval*. pp. 304–316 (2017)