Parameterized Suffix Arrays for Binary Strings

Satoshi Deguchi[1], Fumihito Higashijima[1],
Hideo Bannai[1], Shunsuke Inenaga[2], and Masayuki Takeda[1]

[1]Department of Informatics, Kyushu University
[2]Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishiku, Fukuoka 819-0395, Japan
{satoshi.deguchi,bannai,takeda}@i.kyushu-u.ac.jp
inenaga@c.csce.kyushu-u.ac.jp

Abstract. We consider the suffix array for parameterized binary strings that consist of only two types of parameter symbols. We show that the parameterized suffix array, as well as its longest common prefix (LCP) array of such strings can be constructed in linear time. The construction is direct, in that it does not require the construction of a parameterized suffix tree. Although parameterized pattern matching of binary strings can be done by either searching for a pattern and its inverse on a standard suffix array, or constructing two independent standard suffix arrays for the text and its inverse, our approach only needs a single p-suffix array and a single search.

1 Introduction

1.1 Parameterized Pattern Matching

Consider strings over $\Pi \cup \Sigma$, where Π is the set of *parameter symbols* and Σ is the set of *constant symbols*. These strings are called *parameterized strings* (*p-strings*). Baker [7] introduced the notion of *parameterized pattern matching*, where two p-strings of the same length are said to parameterized match (p-match) if one string can be transformed into the other by using a bijection on $\Sigma \cup \Pi$. The bijection should be the identity on the constant symbols of Σ, namely, it maps any $a \in \Sigma$ to a itself, while symbols of Π can be interchanged. Examples of applications of parameterized pattern matching are software maintenance [7,8], plagiarism detection [12], and RNA structural matching [25].

Similar to standard string matching, preprocessing for the text strings is efficient for p-string matching. In [8], Baker proposed the *parameterized suffix tree* (*p-suffix tree*) structure to locate all positions of the text string where a given pattern string p-matches. She presented an $O(n(\pi + \log(\pi + \sigma)))$ time algorithm to construct the p-suffix tree for a given text string of length n. The algorithm uses $O(n)$ space, where $\pi = |\Pi|$ and $\sigma = |\Sigma|$. Kosaraju [22] proposed an improved algorithm for constructing p-suffix trees in $O(n(\log \pi + \log \sigma))$ time. Both algorithms are based on McCreight's algorithm that builds standard suffix trees [24]. Shibuya [25] developed an on-line construction algorithm working in $O(n(\log \pi + \log \sigma))$ time, which is based on Ukkonen's construction algorithm for standard suffix trees [26]. Given a pattern p of length m, we can compute the set $Pocc$ of all positions of t where the corresponding substring of t p-matches pattern p in $O(m \log(\pi + \sigma) + |Pocc|)$ time, by using the p-suffix tree of a text string t.

In this paper, we consider *parameterized suffix arrays* (*p-suffix arrays*), whose relation to p-suffix trees is analogous to the relation between standard suffix arrays [23] and standard suffix trees [27]. As is the case with suffix trees and suffix arrays, the

Satoshi Deguchi, Fumihito Higashijima, Hideo Bannai, Shunsuke Inenaga, Masayuki Takeda : Parameterized Suffix Arrays for Binary Strings, pp. 84–94.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

array representation is superior in terms of memory usage and memory access locality. Also, most operations on a p-suffix tree can be efficiently simulated with the p-suffix array and an array containing the lengths of longest common prefixes of the p-suffixes, which we shall call the PLCP array. For instance, using p-suffix and PLCP arrays, the parameterized pattern matching problem can be solved in $O(m \log n + |Pocc|)$ time with a simple binary search, or $O(m + \log n + |Pocc|))$ with a binary search utilizing PLCP information, or $O(m \log(\pi + \sigma) + |Pocc|)$ time if we consider an *enhanced* p-suffix array [1,18].

P-suffix arrays and PLCP arrays can be obtained from a simple linear time traversal of the corresponding p-suffix trees. However, unlike the case of standard suffix arrays [16,19,21], linear time algorithms for *direct* construction of parameterized suffix arrays are not known so far.

In this paper, we take a first step in this problem, and show that for any text p-string t over binary parameter alphabet Π, p-suffix arrays and PLCP arrays can be constructed directly in $O(n)$ time. Our construction algorithm does not need the p-suffix tree of t as an intermediate structure.

There is a naïve solution to the p-matching problem for a binary alphabet using two standard suffix arrays. Given a text t and pattern p over $\Pi = \{a, b\}$, compute another text t' by exchanging a and b in t. Then, compute two suffix arrays for t and t', and search the arrays for pattern p. Or alternatively, compute another pattern p' by exchanging a and b in p, and search for the array of t for p and p'. On the other hand, our approach only needs a *single* p-suffix array and a *single* search for p-matching, and thus is both space- and time-efficient. We also performed some experiments to show the efficiency of our p-suffix arrays.

Parameterized strings on binary parameter alphabet were investigated in literature. Apostolico and Giancarlo [6] pointed out that parameterized strings over a binary parameter alphabet behave in a similar way to standard strings with respect to periodicity and repetitions, but the case with larger parameter alphabet remains open. Our result on the direct linear-time construction of p-suffix arrays for a binary alphabet is yet another one showing the similarity of parameterized strings on a binary alphabet to standard strings.

1.2 Related Work

Another approach for solving the parameterized pattern matching is to preprocess patterns. Idury and Schäffer [15] proposed a variation of the Aho-Corasick automaton [2], which can be constructed in $O(m \log(\pi + \sigma))$ time for a single pattern, and scanning the text takes $O(n \log(\pi + \sigma))$ time. Amir et al. [4] presented a KMP algorithm [20] based approach with $O(m \log(\mathbf{min}\{m, \pi\}))$ preprocessing time and $O(n \log(\mathbf{min}\{m, \pi\}))$ scanning time. They also stated that it suffices to consider strings over Π rather than $\Pi \cup \Sigma$ for the p-matching problem, showing that the problem with $\Pi \cup \Sigma$ can be reduced in linear time to that with Π.

Parameterized pattern matching has been extended to two dimensional parameterized pattern matching [3,14] and approximate parameterized pattern matching [13,5]. Parameterized edit distance was considered in [9].

2 Preliminaries

Let Σ and Π be two disjoint finite sets of *constant symbols* and *parameter symbols*, respectively. An element of $(\Sigma \cup \Pi)^*$ is called a *p-string*. The length of any p-string

s is the total number of constant and parameter symbols in s and is denoted by $|s|$. For any p-string s of length n, the i-th symbol is denoted by $s[i]$ for each $1 \leq i \leq n$, and the *substring* starting at position i and ending at position j is denoted by $s[i:j]$ for $1 \leq i \leq j \leq n$. In particular, $s[1:j]$ and $s[i:n]$ denote the *prefix* of length j and the *suffix* of length $n-i+1$ of s, respectively. For any two p-strings s and t, $lcp(s,t)$ denotes the length of the *longest common prefix* of s and t.

Definition 1 (Parameterized Matching). *Any two p-strings s and t of the same length m are said to* parameterized match *(*p-match*) iff one of the following conditions hold for every $1 \leq i \leq m$:*

1. $s[i] = t[i] \in \Sigma$,
2. $s[i], t[i] \in \Pi$, $s[i] \neq s[j]$ *and* $t[i] \neq t[j]$ *for any* $1 \leq j < i$,
3. $s[i], t[i] \in \Pi$, $s[i] = s[i-k]$ *for any* $1 \leq k < i$ *iff* $t[i] = t[i-k]$.

We write $s \simeq t$ when s and t p-match.

For example, let $\Pi = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\Sigma = \{\mathtt{X}, \mathtt{Y}\}$, $s = \mathtt{abaXabY}$ and $t = \mathtt{bcbXbcY}$. Observe that $s \simeq t$.

Let $\mathcal{N}$ be the set of non-negative integers. For any non-negative integers $i \leq j \in \mathcal{N}$, let $[i,j] = \{i, i+1, \ldots, j\} \subset \mathcal{N}$.

Definition 2. *We define $pv : (\Sigma \cup \Pi)^* \to (\Sigma \cup \mathcal{N})^*$ to be the function such that for any p-string s of length n, $pv(s) = u$ where, for $1 \leq i \leq n$,*

$$u[i] = \begin{cases} s[i] & \text{if } s[i] \in \Sigma, \\ 0 & \text{if } s[i] \in \Pi \text{ and } s[i] \neq s[j] \text{ for any } 1 \leq j < i, \\ i-k & \text{if } s[i] \in \Pi \text{ and } k = \boldsymbol{max}\{j \mid s[i] = s[j], 1 \leq j < i\}. \end{cases}$$

In the running example, $pv(s) = 002\mathtt{X}24\mathtt{Y}$ with $s = \mathtt{abaXabY}$.

The following proposition is clear from Definition 2.

Proposition 3. *For any p-string s of length n, it holds for any $1 \leq i \leq j \leq n$ that*

$$pv(s[i:j]) = v[1:j-i+1],$$

where $v = pv(s[i:n])$.

The pv function is useful for p-matching, because:

Proposition 4 ([8]). *For any two p-strings s and t of the same length, $s \simeq t$ iff $pv(s) = pv(t)$.*

In the running example, we then have $s \simeq t$ and $pv(s) = pv(t) = 002\mathtt{X}24\mathtt{Y}$.

We also define the dual of the pv function, as follows:

Definition 5. *We define $fw : (\Sigma \cup \Pi)^* \to (\Sigma \cup \mathcal{N} \cup \{\infty\})^*$ to be the function such that for any p-string s of length n, $fw(s) = w$ where, for $1 \leq i \leq n$,*

$$w[i] = \begin{cases} s[i] & \text{if } s[i] \in \Sigma, \\ \infty & \text{if } s[i] \in \Pi \text{ and } s[i] \neq s[j] \text{ for any } i < j \leq n, \\ k-i & \text{if } s[i] \in \Pi \text{ and } k = \boldsymbol{min}\{j \mid s[i] = s[j], i < j \leq n\}. \end{cases}$$

Here, ∞ denotes a value for which $i < \infty$ for any $i \in \mathcal{N}$. [1]

[1] In practice, n can be used in place of ∞ as long as we are considering a single p-string of length n, and its substrings.

In the running example, $fw(s) = 242\texttt{X}\infty\infty\texttt{Y}$ with $s = \texttt{abaXabY}$.

The following proposition is clear from Definition 5.

Proposition 6. *For any p-string s of length n, it holds for any $1 \leq i \leq n$ that*

$$fw(s[i:n]) = w[i:n],$$

where $w = fw(s)$.

For any p-string s of length n, $pv(s)$ and $fw(s)$ can be computed in $O(n)$ time with extra $O(\pi)$ space, using a table of size π recording the last position of each parameter symbol in the left-to-right (resp. right-to-left) scanning of s [8].

3 P-matching Problem and P-suffix Arrays

In this section we introduce a new data structure *p-suffix arrays* that are useful to solve the p-matching problem given below.

3.1 Problem

The problem considered in this paper is the following:

Problem 7 (P-matching problem). Given any two p-strings t and p of length n and m respectively, $n \geq m$, compute $Pocc(t,p) = \{i \mid t[i:i+m-1] \simeq p\}$.

It directly follows from Proposition 4 that

$$Pocc(t,p) = \{i \mid pv(p) = pv(t[i:i+m-1])\}.$$

Therefore, from Proposition 3, we are able to compute $Pocc(t,p)$ efficiently, by indexing all elements of the set $\{pv(t[i:n]) \mid 1 \leq i \leq n\}$. The corresponding indexing structure is introduced in the next subsection.

Amir et al. [4] showed that actually we have only to consider p-strings from Π to solve Problem 7, as follows.

Lemma 8 ([4]). *Problem 7 on alphabet $\Sigma \cup \Pi$ is reducible in linear time to Problem 7 on alphabet Π.*

Hence, in the remainder of the paper, we consider only p-strings in Π^*. Then, note that for any p-string s of length n, $pv(s) \in \{[0, n-1]\}^n$ and $fw(s) \in \{[1, n-1] \cup \{\infty\}\}^n$. We also see that if $pv(s)[i] > 0$ then $fw(s)[i - pv(s)[i]] = pv(s)[i]$. Similarly, if $fw(s)[i] < n$ then $pv(s)[i + fw(s)[i]] = fw(s)[i]$.

3.2 P-suffix Arrays

In this section we introduce our data structure p-suffix arrays. Let us begin with normal suffix arrays [23] that have widely been used for standard pattern matching.

Let $\preceq_+$ and $\preceq_-$ denote the total order and its reverse on integers, i.e., for integers $x, y \in \mathcal{N} \cup \{\infty\}$, $x \preceq_+ y \iff x \leq y$ and $x \preceq_- y \iff x \geq y$. The lexicographic ordering on strings of an integer alphabet $[i,j] \cup \{\infty\}$ with respect to a total order $\preceq$ on integers can be defined as follows. For $x, y \in ([i,j] \cup \{\infty\})^*$,

$$x \preceq y \iff: \begin{cases} x \text{ is a prefix of } y, \text{ or} \\ \exists \alpha, u, v \in ([i,j] \cup \{\infty\})^*, a, b \in [i,j] \cup \{\infty\}, \\ \qquad \text{such that } a \prec b,\ x = \alpha a u,\ y = \alpha b v. \end{cases}$$

We define a variation of suffix arrays on the integer alphabet $[1, n-1] \cup \{\infty\}$ and the lexicographical ordering of suffixes w.r.t. $\preceq_-$.

Definition 9 (Suffix Arrays). *For any string $w \in ([1, n-1] \cup \{\infty\})^n$ of length n, its* suffix array SA_w *is an array of length n such that $SA_w[i] = j$, where $w[j : n]$ is the lexicographically i-th suffix of w w.r.t. $\preceq_-$.*

We abbreviate SA_w as SA when clear from the context.

Definition 10 (LCP Arrays). *For any string $w \in ([1, n-1] \cup \{\infty\})^n$ of length n, its* LCP array LCP_w *is an array of length n such that*

$$LCP_w[i] = \begin{cases} -1 & \text{if } i = 1, \\ lcp(w[SA[i-1] : n], w[SA[i] : n]) & \text{if } 2 \le i \le n. \end{cases}$$

We abbreviate LCP_w as LCP when clear from the context.

Remark 11. We emphasize that suffix array $SA_{fw(t)}$ of p-string text $t \in \Pi^*$ does *not* solve Problem 7 directly. Let $p \in \Pi^*$ be a p-string pattern. A careful consideration reveals that an ordinary binary search on $SA_{fw(t)}$ for $fw(p)$ does not work, in that we may miss some occurrences of $fw(p)$ in $fw(t)$. The suffix tree for $fw(t)$ is not useful either, since a node of the tree can have $O(n)$ children and searching the tree for $fw(p)$ requires $O(n^m)$ time, where $n = |t|$ and $m = |p|$. Thus the combination of $SA_{fw(t)}$ and $LCP_{fw(t)}$ does not efficiently solve Problem 7, either. Interestingly, however, $SA_{fw(t)}$ and $LCP_{fw(t)}$ are very helpful to construct the following data structures which provide us with efficient solutions to the problem, to be shown in Section 4.

In order to solve Problem 7 efficiently, we use the two following data structures corresponding to $pv(s)$.

Definition 12 (P-suffix Arrays). *For any p-string $s \in \Pi^n$ of length n, its* p-suffix array PSA_s *is an array of length n such that $PSA_s[i] = j$, where $pv(s[j : n])$ is the lexicographically i-th element of $\{pv(s[i : n]) \mid 1 \le i \le n\}$ w.r.t. $\preceq_+$.*

We abbreviate PSA_s as PSA when clear from the context. Note that PSA_s is not necessarily equal to SA_s, since $pv(s[i : n])$ may not always be a suffix of $pv(s)$.

The following function is useful for p-matching with PSA.

Definition 13. *We define $short : \mathcal{N}^* \to \mathcal{N}^*$ to be the function such that for any string $x \in \mathcal{N}^n$ of length n, $short(x) = y$ where, for $1 \le i \le n$,*

$$y[i] = \begin{cases} x[i] & \text{if } x[i] < i, \\ 0 & \text{if } x[i] \ge i. \end{cases}$$

Lemma 14 ([15]). *For any p-string $s \in \Pi^n$ of length n, let $v = pv(s)[i : n]$ for any $1 \le i \le n$. Then, $short(v) = pv(s[i : n])$.*

Lemma 14 implies that, when using PSA_s, we do not have to store $pv(s[i : n])$ for all $1 \le i \le n$; only $pv(s)$ is sufficient.

Theorem 15. *Problem 7 can be solved in $O(m \log n + |Pocc(t, p)|)$ time by using PSA_t and $pv(t)$.*

Proof. By Proposition 3 and Lemma 14, we can compute $Pocc(t, p)$ by a binary search on PSA_t and $pv(t)$, which takes $O(m \log n + |Pocc(t, p)|)$ time in total. □

The following auxiliary array enables us to solve Problem 7 more efficiently.

Definition 16 (PLCP Arrays). *For any p-string $s \in \Pi^n$ of length n, its* PLCP array $PLCP_s$ *is an array of length n such that*

$$PLCP_s[i] = \begin{cases} -1 & \text{if } i = 1, \\ lcp(pv(s[PSA[i-1] : n]), pv([s[PSA[i] : n])) & \text{if } 2 \le i \le n. \end{cases}$$

We abbreviate $PLCP_s$ as $PLCP$ when clear from the context.

Using $PLCP$, we can achieve an improved solution, as follows:

Theorem 17. *Problem 7 can be solved in $O(m + \log n + |Pocc(t, p)|)$ time by using PSA_t, $PLCP_t$ and $pv(t)$.*

Proof. The time complexity can be improved to $O(m+\log n+|Pocc(t, p)|)$ by a similar manner to [23] for standard suffix and LCP arrays. □

Considering the *enhanced* [1] p-suffix array, we obtain the following bound:

Theorem 18. *Problem 7 can be solved in $O(m \log \pi + |Pocc(t, p)|)$ time.*

Proof. For any p-string $t \in \Pi^*$, the number of children of any internal node of the p-suffix tree for $pv(t)$ is at most π [8]. Hence the enhanced p-suffix array enables us to solve Problem 7 in $O(m + \log \pi + |Pocc(t, p)|)$ time (see [18] for more details). □

In the next section, we present our algorithm to construct PSA and $PLCP$ arrays for binary strings, that is, for the case where $\pi = 2$. Our algorithm runs in linear time, and uses SA and LCP arrays.

4 P-Suffix and PLCP Arrays of Binary P-strings

In this section, we will show that for any binary p-string s, its p-suffix array PSA_s and PLCP array $PLCP_s$ can be computed in linear time, directly from s without the use of p-suffix trees.

We first show that the p-suffix array PSA_s of a binary p-string s is equivalent to the suffix array $SA_{fw(s)}$ of $fw(s)$. Then, we show the relationship between $PLCP_s$ and $LCP_{fw(s)}$, so that $PLCP_s$ can be calculated from $fw(s)$ and $LCP_{fw(s)}$.

Figure 1 shows PSA_s, $PLCP_s$ for $s =$ `abaabaaaabba` and corresponding suffixes of $fw(s)$ and $LCP_{fw(s)}$.

Lemma 19. *For any p-string s and $1 \le i \le n$, $|\{j \mid i = j - pv(s)[j]\}| \le 1$ and $|\{j \mid i = j + fw(s)[j]\}| \le 1$.*

Proof. Let $i = x - pv(s)[x] = y - pv(s)[y]$ for some $i < x < y$. Then, by definition, $s[i] = s[x] = s[y]$ and $y - i = pv(s)[y] = y - \mathbf{max}\{j \mid s[y] = s[j], 1 \le j \le y\} \le y - x$, which is a contradiction. Similar arguments hold for $fw(s)$.

Lemma 20. *For any p-strings $s, t \in \Pi^*$ with $\pi = 2$, $pv(s) \preceq_+ pv(t)$ if and only if $fw(s) \preceq_- fw(t)$.*

i	$PSA[i]$	$s[PSA[i]:n]$	$pv(s[PSA[i]:n])$	$PLCP_s[i]$	$fw(s[PSA[i]:n])$	$LCP_{fw(s)}[i]$
1	12	a	0	-1	∞	-1
2	11	ba	0 0	1	$\infty\infty$	1
3	5	baaaabba	0 0 1 1 1 $\underline{5}$ 1 3	2	$\underline{5}$ 1 1 1 3 1 $\infty\infty$	0
4	9	abba	0 0 1 $\underline{3}$	3	$\underline{3}$ 1 $\infty\infty$	0
5	2	baabaaaabba	0 0 1 $\underline{3}$ 2 1 1 1 5 1 3	4	$\underline{3}$ 1 2 5 1 1 1 3 1 $\infty\infty$	2
6	4	abaaaabba	0 0 $\underline{2}$ 1 1 1 5 1 3	2	$\underline{2}$ 5 1 1 1 3 1 $\infty\infty$	0
7	1	abaabaaaabba	0 0 $\underline{2}$ 1 3 2 1 1 1 5 1 3	4	$\underline{2}$ 3 1 2 5 1 1 1 3 1 $\infty\infty$	1
8	10	bba	0 $\underline{1}$ 0	1	$\underline{1}$ $\infty\infty$	0
9	8	aabba	0 $\underline{1}$ 0 1 3	3	$\underline{1}$ 3 1 $\infty\infty$	1
10	3	aabaaaabba	0 $\underline{1}$ 0 2 1 1 1 5 1 3	3	$\underline{1}$ 2 5 1 1 1 3 1 $\infty\infty$	1
11	7	aaabba	0 $\underline{1}$ 1 0 1 3	2	$\underline{1}$ 1 3 1 $\infty\infty$	1
12	6	aaaabba	0 $\underline{1}$ 1 1 0 1 3	3	$\underline{1}$ 1 1 3 1 $\infty\infty$	2

Figure 1. The p-suffix array PSA of $s = \texttt{abaabaaaabba}$ and corresponding suffixes of $fw(s)$, as well as the PLCP and LCP values. Here, $n = 12$.

Proof. It is clear that $pv(s) = pv(t)$ iff $fw(s) = fw(t)$. Let us now consider the other case.
(=:) Assume $pv(s) \prec_+ pv(t)$. If $pv(s)$ is a prefix of $pv(t)$, then $fw(s) \prec_- fw(t)$ since

$$fw(s)[i] = fw(t[1:|s|])[i] = \begin{cases} fw(t)[i] & i + fw(t)[i] \leq |s| \\ \infty & \text{otherwise,} \end{cases}$$

for all $1 \leq i \leq |s|$. Next, assume that $pv(s)$ is not a prefix of $pv(t)$, and let $i = \mathbf{min}\{j \mid pv(s)[j] \prec_+ pv(t)[j]\}$, $\ell = pv(t)[i]$ and $r = pv(s)[i]$. Then, we have $t[i-\ell] = t[i] \neq t[k]$ for any $i - \ell < k < i$. Therefore, we get $fw(t)[i-\ell] = \ell$. On the other hand, we have $s[i-\ell] \neq s[i] = s[k]$ for any $i - \ell < k < i$, since $\pi = 2$. Hence $fw(s)[i-\ell] > \ell$, which implies that $fw(s)[i-\ell] > fw(t)[i-\ell]$. Thus if $i - \ell = 1$, then clearly $fw(s) \prec_- fw(t)$. Now we consider the case where $i - \ell > 1$. For any $1 \leq h < i - \ell$, we have

$$fw(s)[h] \leq \begin{cases} i - \ell - h & \text{if } fw(s)[h] = fw(s)[i-\ell] \text{ or } r = 0, \\ i - \ell + 1 - h & \text{otherwise,} \end{cases}$$

where the second case comes from the fact that $\pi = 2$. Note that the same inequality stands for t. This implies that there exists $1 \leq p, q \leq i-1$ such that $h = p - pv(s)[p] = q - pv(t)[q]$. From the assumption, $pv(s)[1:i-1] = pv(t)[1:i-1]$ and Lemma 19, we have $p = q$ and hence, $fw(s)[h] = fw(t)[h]$. Therefore, $fw(s)[1:i-\ell-1] = fw(t)[1:i-\ell-1]$, and consequently $fw(s) \prec_- fw(t)$.
($\Longleftarrow$) Assume $fw(s) \prec_- fw(t)$. If $fw(s)$ is a prefix of $fw(t)$, $fw(s) = fw(t)[1:|s|] = fw(t[1:|s|)$. Then $pv(s) = pv(t[1:|s|]) = pv(t)[1:|s|]$, and therefore $pv(s) \prec_+ pv(t)$. Next, assume $fw(s)$ is not a prefix of $fw(t)$, and let $l = lcp(fw(s), fw(t)) < \mathbf{min}\{|s|, |t|\}$. Then, since $fw(s)[1:l] = fw(t)[1:l]$, we have $fw(s[1:l]) = fw(t[1:l])$, and $pv(s[1:l]) = pv(t[1:l])$, and finally $pv(s)[1:l] = pv(t)[1:l]$. Furthermore, $pv(s)[l+1] = pv(t)[l+1]$ holds, since either there exists $1 \leq j \leq l$ such that $j + fw(s)[j] = j + fw(t)[j] = l + 1$ in which case $pv(s)[l+1] = pv(t)[l+1]$, or there doesn't, in which case $pv(s)[l+1] = pv(t)[l+1] = 0$. Therefore, assume $|s| \geq l + 2$ since otherwise the proof is finished.

Let $p = fw(s)[l+1]$ and $q = fw(t)[l+1]$. From the assumption, $\infty \geq p > q \geq 1$. Since $\pi = 2$, $t[l+1] = t[l+1+q] \neq t[k]$ for any $l+1 < k < l+1+q$, and $s[l+1] \neq s[k]$ for

any $l+1 < k \leq \mathbf{min}\{|s|, l+1+q\}$. If $q = 1$, then $pv(s)[l+2] = 0$ since by Lemma 19, there cannot exist $1 \leq j \leq l$ such that $j + fw(s)[j] = j + fw(t)[j] = l + 1$. If $q \geq 2$, this gives us $pv(s)[l+2] = pv(t)[l+2]$, $pv(t)[k] = 1$ for any $l+2 < k < l+1+q$, and $pv(s)[k] = 1$ for any $l+2 < k \leq \mathbf{min}\{|s|, l+1+q\}$, while $pv(t)[l+1+q] = fw(t)[l+1] = q \geq 2$. Either way, we have $pv(s) \prec_+ pv(t)$. □

The next lemma is a direct consequence of Lemma 20.

Lemma 21. *For any p-string $s \in \Pi^*$ with $\pi = 2$, $PSA_s = SA_{fw(s)}$.*

It is well known that the suffix array can be constructed directly from the string in linear time.

Theorem 22 ([16,19,21]). *For any string $w \in ([1, n])^n$ of length n, SA_w can be directly constructed in $O(n)$ time.*

The next theorem follows from Lemma 21 and Theorem 22:

Theorem 23. *For any p-string $s \in \Pi^n$ of length n with $\pi = 2$, PSA_s can be constructed directly in $O(n)$ time by constructing $SA_{fw(s)}$.*

From now on let us consider construction of $PLCP_s$.

Lemma 24. *For any p-strings $s, t \in \Pi^*$ with $\pi = 2$,*

$$lcp(pv(s), pv(t)) = \begin{cases} l & l = k, \\ \boldsymbol{min}\{k, l + \boldsymbol{min}\{fw(s)[l+1], fw(t)[l+1]\}\} & otherwise, \end{cases}$$

where $l = lcp(fw(s), fw(t))$, and $k = \boldsymbol{min}\{|s|, |t|\}$.

Proof. By similar arguments as in ($\Longleftarrow$) of Lemma 20. □

It is well known that the LCP array for strings can be constructed efficiently from its corresponding suffix array.

Theorem 25 ([17]). *For any string s of length n, LCP_s can be constructed in $O(n)$ time, given s and its suffix array SA_s.*

Due to Lemma 24 and Theorem 25, we have:

Theorem 26. *For any p-string $s \in \Pi^n$ of length n with $\pi = 2$, $PLCP_s$ can be constructed in $O(n)$ time from $PSA_s = SA_{fw(s)}$ and $fw(s)$.*

5 Computational Experiments

Here, we consider parameterized pattern matching on binary strings. We compare the method using parameterized suffix arrays and two naïve methods that either uses two patterns or two suffix arrays. We run our algorithm for various pattern and text lengths for random binary strings.

Figure 2 shows the time for p-matching not including the p-suffix array construction for random texts of length 100 and 1000. The average of 1000 p-matchings for the same text and pattern strings is further averaged by 100 runs for different random strings. We can see that our approach is the fastest for short patterns. However, the overhead for creating $pv(p)$ for pattern p seems to take over, when p becomes longer.

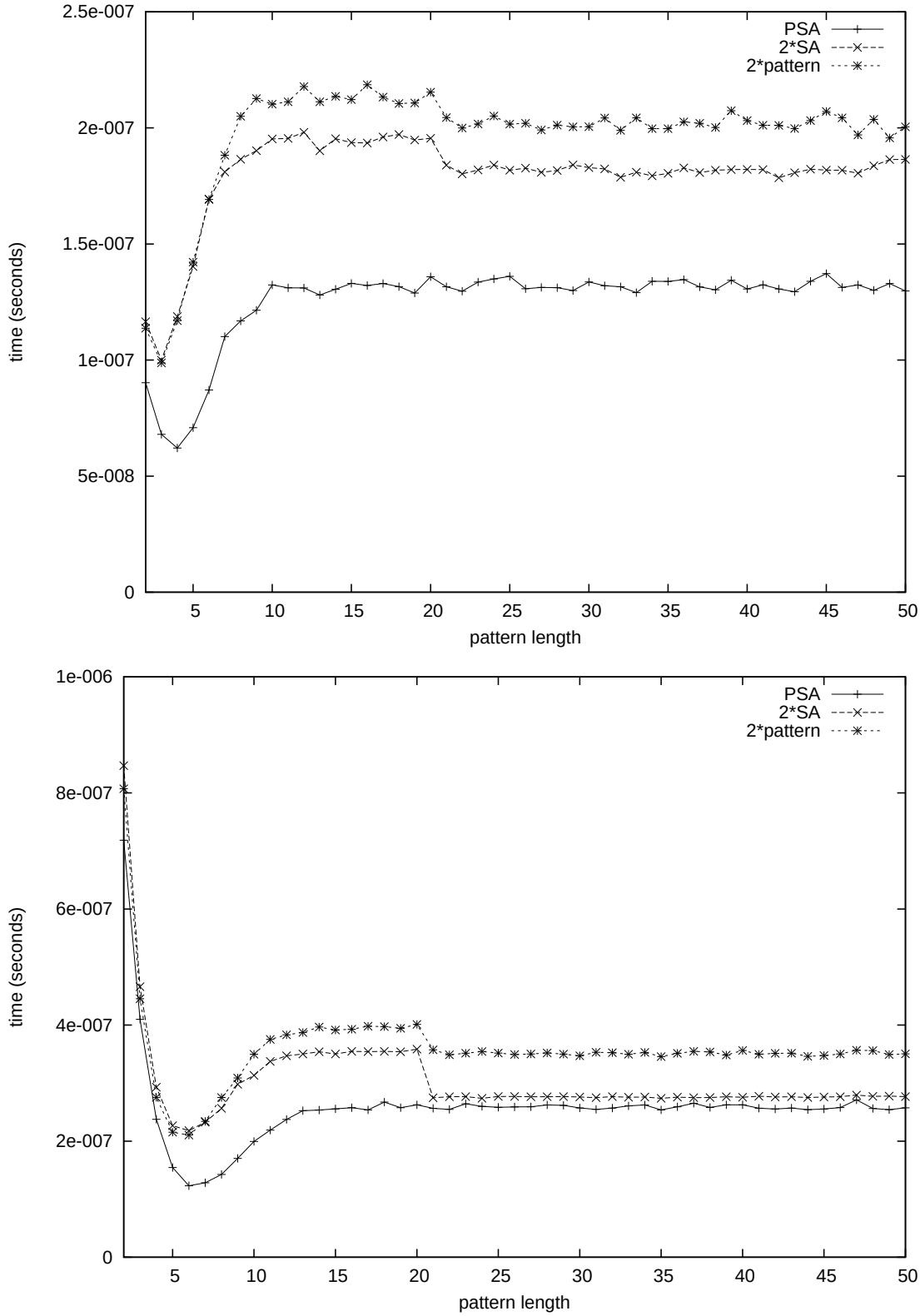

Figure 2. Comparison of running times for p-matching on random binary strings. The length of the text is 100 (upper) and 1000 (lower). The increase in time for short patterns is due to the increase of $|Pocc|$.

6 Conclusion and Future Work

We showed that p-suffix arrays and PLCP arrays for binary strings can be constructed in linear time. It is an open problem whether or not the parameterized suffix array and PLCP array for larger alphabets can be constructed directly in linear time. It is difficult to apply standard suffix array algorithms or LCP calculation algorithms, since an important property does not hold for p-strings. Namely, a suffix $pv(s)[i:n]$ of $pv(s)$ is not necessarily equal to the $pv(s[i:n])$ of the suffix $s[i:n]$. As an important consequence, for any p-strings s, t with $lcp(pv(s), pv(t)) > 0$, $pv(s) \preceq_+ pv(t)$ does not necessarily imply $pv(s[2:|s|]) \preceq_+ pv(t[2:|t|])$ which is essential in the standard case.

For similar reasons, the reverse problem of finding a p-string whose p-suffix array is equal to a given array of integers also does not seem to be as simple as in the case for standard suffix arrays [11,10], and is another open problem.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* Journal of Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
3. A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat: *Function matching: Algorithms, applications, and a lower bound*, in Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP'03), vol. 2719 of Lecture Notes in Computer Science, 2003, pp. 929–942.
4. A. Amir, M. Farach, and S. Muthukrishnan: *Alphabet dependence in parameterized matching.* Information Processing Letters, 49(3) 1994, pp. 111–115.
5. A. Apostolico, P. L. Erdös, and M. Lewenstein: *Parameterized matching with mismatches.* Journal of Discrete Algorithms, 5(1) 2007, pp. 135–140.
6. A. Apostolico and R. Giancarlo: *Periodicity and repetitions in parameterized strings.* Discrete Applied Mathematics, 156(9) 2008, pp. 1389–1398.
7. B. S. Baker: *A program for identifying duplicated code.* Computing Science and Statistics, 24 1992, pp. 49–57.
8. B. S. Baker: *Parameterized pattern matching: Algorithms and applications.* Journal of Computer and System Sciences, 52(1) 1996, pp. 28–42.
9. B. S. Baker: *Parameterized diff*, in Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), 1999, pp. 854–855.
10. H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda: *Inferring strings from graphs and arrays*, in Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003), vol. 2747 of Lecture Notes in Computer Science, 2003, pp. 208–217.
11. J.-P. Duval and A. Lefebvre: *Words over an ordered alphabet and suffix permutations.* Theoretical Informatics and Applications, 36 2002, pp. 249–259.
12. K. Fredriksson and M. Mozgovoy: *Efficient parameterized string matching.* Information Processing Letters, 100(3) 2006, pp. 91–96.
13. C. Hazay, M. Lewenstein, and D. Sokol: *Approximate parameterized matching.* ACM Transactions on Algorithms, 3(3) 2007, Article No. 29.
14. C. Hazay, M. Lewenstein, and D. Tsur: *Two dimensional parameterized matching*, in Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05), vol. 3537 of Lecture Notes in Computer Science, 2005, pp. 266–279.
15. R. M. Idury and A. A. Schäffer: *Multiple matching of parameterized patterns.* Theoretical Computer Science, 154(2) 1996, pp. 203–224.
16. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP'03), vol. 2719 of Lecture Notes in Computer Science, 2003, pp. 943–955.

17. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*, in Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01), vol. 2089 of Lecture Notes in Computer Science, 2001, pp. 181–192.
18. D. K. Kim, J. E. Jeon, and H. Park: *An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size*, in Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE'04), vol. 3246 of Lecture Notes in Computer Science, 2004, pp. 138–149.
19. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Linear-time construction of suffix arrays*, in Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03), vol. 2676 of Lecture Notes in Computer Science, 2003, pp. 186–199.
20. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(2) 1977, pp. 323–350.
21. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03), vol. 2676 of Lecture Notes in Computer Science, 2003, pp. 200–210.
22. S. Kosaraju: *Faster algorithms for the construction of parameterized suffix trees*, in Proc. 36th Annual Symposium on Foundations of Computer Science (FOCS'95), 1995, pp. 631–637.
23. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches.* SIAM J. Computing, 22(5) 1993, pp. 935–948.
24. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23(2) 1976, pp. 262–272.
25. T. Shibuya: *Generalization of a suffix tree for RNA structural pattern matching.* Algorithmica, 39(1) 2004, pp. 1–19.
26. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
27. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.