Lightweight Parameterized Suffix Array Construction

Tomohiro I[1], Satoshi Deguchi[1], Hideo Bannai[1], Shunsuke Inenaga[2], and Masayuki Takeda[1]

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

Abstract. We present a first algorithm for direct construction of parameterized suffix arrays and parameterized longest common prefix arrays for non-binary strings. Experimental results show that our algorithm is much faster than naïve methods.

1 Introduction

Parameterized pattern matching is a form of pattern matching first introduced by Baker [1], that allows for interchange in the alphabet. More formally, let Π be the set of *parameter symbols* and Σ be the set of *constant symbols*. Strings over $\Pi \cup \Sigma$ are called *parameterized strings* (*p-strings*). 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 $\Sigma \cup \Pi$, with the restriction that the bijection must be the identity on the constant symbols of Σ. In other words, the bijection 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 [1, 2], plagiarism detection [3], and RNA structural matching [4].

For the standard pattern matching problem, there exist several data structures that can be obtained by preprocessing the text string so that pattern matching can be performed efficiently. The most famous are the suffix tree [5] and suffix array [6]. These data structures can both be constructed directly in linear time [5, 7–12], independent of the alphabet size. Most operations on a suffix tree can be efficiently simulated with the suffix array and several other auxiliary arrays, including an array containing the lengths of longest common prefixes of the suffixes (LCP array), composing an *enhanced* suffix array [13, 14]. The array representation has become more preferable as it requires less memory, and is generally faster to construct and work on, due to memory access locality.

For p-string pattern matching, Baker [2] introduced the *parameterized suffix tree* (*p-suffix tree*), which is similar in concept to the suffix tree. Baker gave an $O(n(\pi + \log(\pi + \sigma)))$ time algorithm to construct the p-suffix tree for a given text string, where n is the text length, $\pi = |\Pi|$ and $\sigma = |\Sigma|$. Kosaraju [15] proposed an algorithm to construct p-suffix trees in $O(n(\log \pi + \log \sigma))$ time.

Both algorithms are based on McCreight's construction algorithm for standard suffix trees [7]. Shibuya [4] gave 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 [8]. 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, using the p-suffix tree of a text t.

Concerning the array representation, parameterized suffix arrays were considered by Deguchi *et al.* [16]. 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 *enhanced* p-suffix arrays. As with the case of standard suffix trees and arrays, the array representation is superior in memory usage and memory access locality. Deguchi *et al.* presented a linear time algorithm for direct construction of the parameterized suffix array and parameterized LCP (PLCP) array for binary strings. To the best of our knowledge, no efficient algorithm for direct construction of a parameterized suffix array and PLCP array for non-binary strings exist, and the best theoretical worst-case time bound is $O(n^2)$, using a standard radix sort on strings.

This paper presents a new algorithm for efficient construction of p-suffix arrays and PLCP arrays for non-binary strings. For p-suffix array construction, our algorithm combines any string sorting algorithm with linear time pre- and post processing. Though we are unable to reduce the theoretical worst case time bound, our algorithm considerably reduces the number of suffixes to be sorted using the string sorting algorithm, hence greatly reducing the running time. For the PLCP array, we modify the linear time LCP array construction algorithm of [17], so that it can be used for p-strings. However, due to properties of p-strings, it is still open if the theoretical time bound of our algorithm is linear. Computational experiments show both our algorithms are generally much faster than naïve approaches for various texts.

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 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 $1 \leq i \leq n$,*

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

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

Proposition 2 ([2]). *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 3. *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 $1 \leq i \leq n$,*

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

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

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

Proposition 3. *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 [2].

[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.

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

Proposition 2 implies that $Pocc(t, p) = \{i \mid pv(p) = pv(t[i : i+m-1])\}$.

Lemma 1 ([18]). *Problem 1 on alphabet $\Sigma \cup \Pi$ is reducible in linear time to Problem 1 on alphabet Π.*

Due to the above lemma, 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 $pv(s)[i + fw(s)[i]] = fw(s)[i]$.

Let $\preceq$ denote the standard lexicographic ordering on strings of an integer alphabet. To simplify discussions on the end of strings, we assume that for any p-string s, $pv(s)[i] = -1$ for any $i > |s|$.

In this paper, we will consider construction of the following data structures.

Definition 4 (P-suffix Array). *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 $\{pv(s[k : n]) \mid 1 \leq k \leq n\}$.*

Definition 5 (PLCP Array). *For any p-string $s \in \Pi^n$ of length n, its* PLCP array *$PLCP_s$ is an array of length n such that*

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

We abbreviate $PLCP_s$ as $PLCP$ when clear from the context. The following is a useful auxiliary array that we will use for the construction of $PLCP$.

Definition 6 (rank array). *For any p-string $s \in \Pi^n$ of length n, its* rank array *$rank_s$ is an array of length n such that $rank_s[PSA_s[i]] = i$, for any $1 \leq i \leq n$.*

We abbreviate $rank_s$ as $rank$ when clear from the context. Note that $rank_s[i]$ can be computed in linear time from PSA_s for all i where $1 \leq i \leq n$.

Table 1 shows an example of a p-suffix array, PLCP array and rank array for the string $s =$ `babbcacaabcb`.

The PSA, $PLCP$, and $rank$ arrays can naturally be used in similar ways as the suffix, LCP, and rank arrays for standard strings. PSA and $PLCP$ arrays can be constructed by a linear time traversal on the p-suffix tree. However, unlike standard suffix arrays and lcp arrays, direct linear time algorithms that do not construct the tree as an intermediate data structure are not known, except for the case of binary alphabets [16].

The main difficulty in developing efficient algorithms for constructing PSA and $PLCP$ is that for any p-string s, a suffix $pv(s)[i : n]$ of $pv(s)$ is not necessarily equal to $pv(s[i : n])$ 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 a property essential for efficient construction algorithms in the standard case.

Table 1. PSA_t and $PLCP_t$ for p-string $s =$ `babbcacaabcb`.

i	$PSA[i]$	$PLCP[i]$	rank[i]	1	2	3	4	5	6	7	8	9	10	11	12	$zlen[PSA[i]]$	type
1	12	-1	9	0												1	A
2	11	1	6	0	0											2	A
3	9	2	12	0	0	0	2									3	B
4	4	4	4	0	0	0	2	2	1	6	4	2				3	B
5	7	2	10	0	0	1	0	4	2							2	B
6	2	6	8	0	0	1	0	4	2	2	1	6	4	2		2	B
7	10	2	5	0	0	2										2	C
8	6	3	11	0	0	2	1	0	4	2						2	C
9	1	7	3	0	0	2	1	0	4	2	2	1	6	4	2	2	C
10	5	3	7	0	0	2	2	1	0	4	2					2	C
11	8	1	2	0	1	0	0	2								1	C
12	3	5	1	0	1	0	0	2	2	1	6	4	2			1	C

3 Algorithm

For our lightweight algorithm, we will use the number of contiguous zeroes in the prefix of each suffix $pv(s[i:n])$, to sort them coarsely.

Definition 7. *For any p-string s of length n, we define $zlen_s[i]$ as the length of contiguous zeroes in the prefix of $pv(s[i:n])$ for any $1 \leq i \leq n$. That is,*

$$zlen_s[i] = \max\{j \mid pv(s[i:i+j-1]) = 0^j, 1 \leq j \leq n-i+1\}.$$

We abbreviate $zlen_s$ as $zlen$ when clear from the context. Note that $zlen_s[i]$ can be computed in amortized linear time for all i where $1 \leq i \leq n$.

3.1 Constructing P-suffix Array

$zlen$ divides the set of suffixes $s[i:n] (1 \leq i \leq n)$ of p-string s into 3 types.

- Type A: those consisting of zeroes only, that is $zlen_s[i] = |s| - i + 1$.
- Type B: those with $zlen_s[i] < |s|-i+1$ and $zlen_s[i] > pv(s[i:n])[zlen_s[i]+1]$
- Type C: those with $zlen_s[i] < |s|-i+1$ and $zlen_s[i] = pv(s[i:n])[zlen_s[i]+1]$.

Our algorithm consists of the following steps.

1. Calculate $zlen_s[i]$ for all i in linear time and determine its type.
2. Determine the positions of type A suffixes using $zlen_s[i]$.
3. Coarsely sort all type B and C suffixes in linear time by radix sort using $zlen_s[i]$ and the first non-zero value $pv(s[i:n])[zlen_s[i]+1]$ in $pv(s[i:n])$.
4. Determine the positions of the type B suffixes using any sorting algorithm.
5. Determine the positions of the remaining type C suffixes in linear time.

Each step can be calculated in linear time except for Step 4 which depends on the underlying sort algorithm. We will describe the details of each step below.

First, we determine the positions of the type A suffixes (Step 2). Let A_s denote the number of type A suffixes of s. For the type A suffixes, it is obvious that $PSA_s[1 : A_s] = n, n-1, \ldots, n - A_s + 1$.

For the type B and C suffixes (Step 3), they are divided into *blocks* of suffixes that have the same *zlen* value and first non-zero value. We can determine the order of these blocks in linear time and space by radix sort, that is, bucket sort first in ascending order of the first non-zero value and then in descending order of the *zlen* value. For any suffix i, the first non-zero value is not greater than $zlen[i]$. Therefore, we can also do this operation by a single bucket sort in descending order of $\sum_{t=1}^{zlen[i]} t - pv(s[i:n])[zlen[i]+1] = \frac{zlen[i](zlen[i]+1)}{2} - pv(s[i:n])[zlen[i]+1]$. This alternative method works in linear time and space, as long as $\frac{z(z+1)}{2} = O(n)$, where $z = \max\{zlen_s[i] \mid 1 \le i \le n - A_s\}$.

Step 4 is then conducted using any string sorting algorithm within each type B block. Note that within a block, it suffices to see the order of $pv(s[i:n])[zlen_s[i]+2:n]$ since they will have a common prefix of length $zlen_s[i]+1$.

Let $\overline{PSA}_s$ denote the intermediate array obtained just after processing Step 4, that is, only the type C suffixes are not in position yet. The next lemmas describe the key properties for sorting these remaining suffixes in linear time (Step 5).

Lemma 2. *Let s be a p-string of length n. For any i, j $(1 \le i, j \le n-1)$, if $fw(s)[i] \ge fw(s)[j]$ and $pv(s[i+1:n]) \prec pv(s[j+1:n])$, then $pv(s[i:n]) \prec pv(s[j:n])$.*

Proof. Assume on the contrary that $pv(s[i:n]) \succ pv(s[j:n])$. Let l be $lcp(pv(s[i:n]), pv(s[j:n]))$. Since $pv(s[i:n]) \succ pv(s[j:n])$, then

$$pv(s[i:n])[1:l] = pv(s[j:n])[1:l],\ pv(s[i:n])[l+1] > pv(s[j:n])[l+1].$$

(i) $fw(s)[j] > l$. Since $fw(s)[i] \ge fw(s)[j]$, then $pv(s[i+1:n])[1:l-1] = pv(s[j+1:n])[1:l-1]$, $pv(s[i+1:n])[l] > pv(s[j+1:n])[l]$. We get $pv(s[i+1:n]) \succ pv(s[j+1:n])$, a contradiction.
(ii) $fw(s)[j] = l$. By assumption, $pv(s[i:n])[l+1] > pv(s[j:n])[l+1] = fw(s)[j] = l$. However, by Definition 2, $pv(s[i:n])[l+1] \le l$, a contradiction.
(iii) $fw(s)[j] < l$. Since $fw(s)[i] = fw(s)[j]$, then $pv(s[i+1:n])[1:l-1] = pv(s[j+1:n])[1:l-1]$, $pv(s[i+1:n])[l] > pv(s[j+1:n])[l]$. We get $pv(s[i+1:n]) \succ pv(s[j+1:n])$, a contradiction. □

Lemma 3. *Let s be a p-string of length n. For any i $(1 \le i \le n-1)$, if $pv(s[i:n])$ is a type C suffix, then $pv(s[i+1:n]) \prec pv(s[i:n])$.*

Proof. Assume on the contrary that $pv(s[i+1:n]) \succ pv(s[i:n])$. Since $pv(s[i:n])$ is a type C suffix, $zlen[i+1] \ge zlen[i]$. Then for some k $(zlen[i]+2 \le k \le n-i+1)$, $pv(s[i+1:n])[1:k-1] = pv(s[i:n])[1:k-1]$ and $pv(s[i+1:n])[k] > pv(s[i:n])[k]$. In addition, it follows from $pv(s[i:n])[zlen[i]+1] = zlen[i]$ that

```
// Initialize:
//   head[i] = k: min position in type C block with k = zlen[PSA[j]]
//   block[i] = if(PSA[i]-1 is type C) then zlen[PSA[i]-1] else 0
for (i = 1; i < n; i++) {
  j = block[psa[i]];
  if (j != 0) {                 // s[psa[i]-1:n] is type C
    psa[head[j]] = psa[i] - 1;  // determine its position in block j
    head[j]++;                  // increment head position of block
  }
}
```

Fig. 1. Algorithm for sorting type C suffixes in linear time. Note that the initialization of `head` and `block` arrays can be done in linear time.

$pv(s[i+1:n])[zlen[i]+1:n] = pv(s[i:n])[zlen[i]+2:n-1]$. Here,

$$\begin{aligned} pv(s[i:n])[k] &= pv(s[i+1:n])[k-1] = pv(s[i:n])[k-1] \\ &= pv(s[i+1:n])[k-2] = pv(s[i:n])[k-2] \\ &= \cdots = pv(s[i+1:n])[zlen[i]+1] = pv(s[i:n])[zlen[i]+1] = zlen[i]. \end{aligned}$$

Hence $pv(s[i+1:n])[1:k-1] = 0^{zlen[i]}zlen[i]^{k-1-zlen[i]}$. This implies that $pv(s[i+1:n])[k] \leq zlen[i] = pv(s[i:n])[k]$, a contradiction. □

Theorem 1. *Let s be a p-string of length n. If the positions of type A and B suffixes are determined, the positions of the remaining type C suffixes can be determined in linear time.*

Proof. From Lemma 2, if $pv(s[i+1:n]) \prec pv(s[j+1]:n)$ and $pv(s[i:n])$, $pv(s[j:n])$ are in the same type C block, we have $pv(s[i:n]) \prec pv(s[j:n])$ since fw values are equal. Therefore, we scan $\overline{PSA_s}$ in increasing order and determine the correct position of type C suffix $pv(s[i:n])$ if the position of suffix $pv(s[i+1:n])$ is already determined. This is guaranteed by Lemma 3, since the position of suffix $pv(s[i+1:n])$ precedes that of $pv(s[i:n])$. Hence, we can determine the positions of the type C suffixes by running through $\overline{PSA_s}$ once. □

Fig. 1 shows our linear time sorting algorithm for type C suffixes.

3.2 Constructing PLCP Array

This section considers the construction of PLCP arrays, given the p-suffix array. For standard LCP arrays, Kasai *et al.* [17] showed a linear time algorithm for its construction, given the suffix array. However, the same algorithm cannot be applied directly to PLCP arrays because of the difficulties mentioned at the end of Section 2. In the following, we show some characteristics of PLCP arrays and propose P-Kasai, a modified version of the algorithm of [17].

For the type A suffixes, it is obvious that $PLCP_s[1:A_s] = -1, 1, 2, \ldots, A_s-1$. Hence we mainly consider computing $PLCP_s[i]$ where $A_s < i \leq n$.

Lemma 4. *Let s be a p-string of length n and $l_i = PLCP_s[rank[i]]$. For any i $(1 \leq i < n)$, if $l_i > 0$ then*

$$\begin{cases} l_{i+1} \geq l_i - 1 & \text{if } pv(s[j+1:n]) \prec pv(s[i+1:n]), \\ l_{j+1} \geq l_i - 1 & \text{if } pv(s[j+1:n]) \succ pv(s[i+1:n]), \end{cases}$$

where $j = PSA[rank[i]-1]$.

Proof. Since $pv(s[j:j+l_i-1]) = pv(s[i:i+l_i-1])$, then $pv(s[j+1:j+l_i-1]) = pv(s[i+1:i+l_i-1])$. If $pv(s[j+1:n]) \prec pv(s[i+1:n])$, then for any k $(rank[j+1] \leq k \leq rank[i+1])$,

$$pv(s[j+1:j+l_i-1]) = pv(s[PSA[k]:PSA[k]+l_i-2]) = pv(s[i+1:i+l_i-1]).$$

Hence, $l_{i+1} \geq l_i - 1$. Similarly, if $pv(s[j+1:n]) \succ pv(s[i+1:n])$, we can get $l_{j+1} \geq l_i - 1$. □

Note that the case of $l_i > 0$ and $pv(s[j+1:n]) \succ pv(s[i+1:n])$ does not occur for LCP arrays, which is the key property for amortized linear time construction algorithm [17]. In the case for $PLCP$ arrays, for example, we can see in Table 1 that $pv(s[2:12]) \prec pv(s[10:12])$ but $pv(s[3:12]) \succ (s[11:12])$.

A necessary condition for $pv(s[j+1:n]) \succ pv(s[i+1:n])$ is given below:

Lemma 5. *Let s be a p-string of length n and $l_i = PLCP_s[rank[i]]$. For any i $(1 \leq i < n)$, if $pv(s[j+1:n]) \succ pv(s[i+1:n])$ then $pv(s[i:n])[l_i+1] = l_i$, where $j = PSA[rank[i]-1]$.*

Proof. First, $pv(s[j+1:n])[1:l_i-1] = pv(s[i+1:n])[1:l_i-1]$, and by Definition 2 $pv(s[i:n])[l_i+1] \leq l_i$. If we assume $pv(s[i:n])[l_i+1] < l_i$, then

$$pv(s[j+1:n])[l_i] \leq pv(s[j:n])[l_i+1] < pv(s[i:n])[l_i+1] = pv(s[i+1:n])[l_i].$$

This implies that $pv(s[j+1:n]) \prec pv(s[i+1:n])$, a contradiction. □

Focusing on the case of $pv(s[i+1:n]) \prec pv(s[i:n])$, we have:

Lemma 6. *Let s be a p-string of length n and $l_i = PLCP_s[rank[i]]$. For any i $(1 \leq i < n)$, if $pv(s[i:n])[l_i+1] = l_i$ and $pv(s[i+1:n]) \prec pv(s[i:n])$ then $zlen[i+1] \geq l_i$.*

Proof. Assume contrary that $zlen[i+1] < l_i$. Since $pv(s[i+1:n])[l_i] = 0$, there exists $m = \min\{k \mid 2 \leq k < l_i, pv(s[i+1:n])[k] \neq 0\}$, that is, $zlen[i+1] = m-1$, and then $zlen[i] = m$. This implies $pv(s[i:n]) \prec pv(s[i+1:n])$, a contradiction. □

Lemmas 5 and 6 lead to the next lemma.

Lemma 7. *Let s be a p-string of length n and $l_i = PLCP_s[rank[i]]$. For any i $(1 \leq i < A_s - 1)$, if $pv(s[j+1:n]) \succ pv(s[i+1:n])$ and $pv(s[i+1:n]) \prec pv(s[i:n])$ then*

$$\begin{cases} l_{i+1} = A_s & \text{if } rank[i+1] = A_s + 1 \text{ and } l_i > A_s, \\ l_{i+1} \geq l_i & \text{otherwise}, \end{cases}$$

where $j = PSA[rank[i]-1]$.

```
plcp[1] = -1;
for (b = 2; psa[b] == n - b; b++) {
  plcp[b] = b - 1; // PLCP for type A suffixes
}
// b = As + 1
k = 1;
for (i = 1; i <= n - b + 1; i++) {
  j = psa[rank[i]-1];
  if (plcp[rank[i]] > k)
    k = plcp[rank[i]];
  while (pv(s[i:n])[k+1] ==  pv(s[j:n])[k+1])
    k++;
  plcp[rank[i]] = k;
  if (rank[j+1] < rank[i+1])
    k--;                         // Kasai's algorithm up to here
  else {                         // below is modification
    if (plcp[rank[j+1]] < k - 1)
      plcp[rank[j+1]] = k - 1;
    if (rank[i+1] < rank[i]) {
      if (rank[i+1] == b && k > b - 1)
        k = b - 1;
    } else
      k = 1;
  }
}
```

Fig. 2. Algorithm for constructing PLCP array (P-Kasai).

Proof. It follows from Lemma 5 and 6 that $zlen[i+1] \geq l_i$. If $rank[i+1] = A_s+1$, it is obvious that $zlen[PSA[rank[i+1]-1]] = zlen[PSA[A_s]] = A_s$. In addition if $l_i > A_s$, then $pv(s[1 : A_s]) = 0^{A_s}$ is a prefix of $pv(s[i+1 : n])$. Hence $l_{i+1} = A_s$.

On the other hand, in the case of $rank[i+1] > A_s+1$, we have $zlen[PSA[rank[i+1]-1]] \geq l_i$, since type B and C suffixes are sorted in descending order of *zlen*. If $rank[i+1] = A_s+1$ but $l_i \leq A_s$, then $zlen[PSA[rank[i+1]-1]] = A_s \geq l_i$. Hence in either case, $zlen[PSA[rank[i+1]-1]] \geq l_i$, and then $l_{i+1} \geq l_i$. □

Lemma 7 helps to compute l_{i+1} in the case of $pv(s[j+1 : n]) \succ pv(s[i+1 : n])$. Fig. 2 shows our algorithm for constructing PLCP array.

4 Computational Experiments

We compare our algorithms and naive algorithms on randomly generated text, and some files taken from the The Canterbury Corpus[2], and those used in [19][3] (Lightweight). All experiments were conducted on an Apple Mac Pro (Early 2008) with 3.2GHZ dual core Xeons and 18GB of memory, running MacOSX 10.5 Leopard. Programs were written in the C language and compiled with the gcc compiler and -O3 option.

[2] `http://corpus.canterbury.ac.nz/`

[3] `http://web.unipmn.it/~manzini/lightweight/corpus/`

Table 2. Running times for random strings of length n and alphabet size = 255.

	p-suffix array			PLCP array				
n	Radix	Bucket	Qsort	P-Kasai	Naive	avg *lcp*	avg *zlen*	%C
1024	0.000099	0.000170	0.000710	0.000023	0.000207	23.6	19.4	7.58
2048	0.000241	0.000319	0.001672	0.000050	0.000374	25.5	19.6	7.57
4096	0.000689	0.000778	0.004032	0.000147	0.000595	27.1	19.7	7.66
8192	0.001759	0.001840	0.009391	0.000304	0.001116	28.1	19.6	7.73
16384	0.003840	0.003908	0.018881	0.000629	0.002363	29.0	19.7	7.71
32768	0.009518	0.009440	0.042851	0.001291	0.004697	29.6	19.6	7.73
65536	0.022920	0.022261	0.096403	0.002856	0.010423	30.3	19.7	7.72
131072	0.050810	0.048990	0.201429	0.006450	0.020761	31.0	19.7	7.73
262144	0.124608	0.120097	0.458788	0.015064	0.039021	31.9	19.7	7.72
524288	0.287857	0.276315	0.998540	0.062928	0.083262	33.0	19.7	7.72

Table 2 shows results on random data for various text lengths, and a fixed alphabet size of 255. Table 3 shows results on random data for various alphabet sizes and a fixed text length of 1,000,000. Table 4 shows results on various texts from several corpora. Radix and Bucket denote the two alternatives for the coarse sorting in Step 3. We use a standard quick sort on strings for the sorting algorithm of Step 4. Qsort denotes a naive algorithm using a standard quick sort on all suffixes. For the PLCP array, we compare the P-Kasai algorithm and a naive algorithm. %C denotes the percentage of type C suffixes. The running times were measured by user time, averaged over 100 and 10 iterations for random strings and files, respectively. In the tables, they are presented in second.

Our algorithms are clearly much faster than naive methods except for the PLCP computation when the average lcp is small.

5 Conclusion and Future Work

Using several characteristics of parameterized suffixes, we introduced techniques to speed up the direct construction of parameterized suffix arrays and PLCP arrays. The worst case time complexity of sorting the suffixes is $O(n^3)$ when using a standard Quicksort on strings. For example, considering $pv(\texttt{abbaabb}\ldots) = 0013131\ldots$, gives one type B block of size $n/2$ requiring $O(n^3)$ time. However, since the size of the blocks to be sorted is reduced considerably compared to n, the total time required for our algorithm is much faster than a naïve use of Quicksort.

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

Table 3. Running times for random strings of length 1,000,000 and alphabet size π.

	p-suffix array			PLCP array				
π	Radix	Bucket	Qsort	P-Kasai	Naive	avg *lcp*	avg *zlen*	%C
2	0.351	0.327	1.144	0.203	0.124	19.8	1.5	75.0
4	0.387	0.363	0.851	0.228	0.109	11.5	2.2	55.5
8	0.430	0.404	0.839	0.242	0.112	10.7	3.2	40.6
16	0.493	0.459	0.975	0.239	0.121	12.2	4.7	29.4
32	0.522	0.495	1.114	0.225	0.131	15.1	6.8	21.2
64	0.543	0.528	1.346	0.204	0.145	19.4	9.7	15.2
128	0.584	0.572	1.645	0.187	0.161	25.7	13.9	10.8
256	0.633	0.613	2.111	0.170	0.186	34.2	19.7	7.70
512	0.697	0.688	2.708	0.159	0.224	45.1	28.0	5.48
1024	0.754	0.754	3.570	0.147	0.264	61.1	39.8	3.89
2048	0.851	0.825	4.688	0.141	0.360	84.8	56.4	2.75
4096	0.940	0.921	6.309	0.132	0.454	118.6	79.8	1.95
8192	0.984	0.990	8.373	0.127	0.628	165.8	113.1	1.38
16384	1.039	1.068	11.570	0.124	0.864	230.3	160.0	0.978
32768	1.289	1.074	15.510	0.119	1.266	317.0	226.2	0.691
65536	0.967	1.070	21.972	0.115	1.896	430.8	320.0	0.489
131072	0.834	0.992	30.362	0.112	3.058	578.2	453.5	0.345
262144	0.667	0.946	42.425	0.111	5.780	766.4	640.4	0.245
524288	0.509	0.941	58.855	0.105	9.795	1019.7	907.2	0.173

References

1. Baker, B.S.: A program for identifying duplicated code. Computing Science and Statistics **24** (1992) 49–57
2. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences **52**(1) (1996) 28–42
3. Fredriksson, K., Mozgovoy, M.: Efficient parameterized string matching. Information Processing Letters **100**(3) (2006) 91–96
4. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. Algorithmica **39**(1) (2004) 1–19
5. Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory. (1973) 1–11
6. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing **22**(5) (1993) 935–948
7. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM **23**(2) (1976) 262–272
8. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3) (1995) 249–260
9. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. 38th Annual Symposium on Foundations of Computer Science. (1997) 137–143
10. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. ICALP'03. Volume 2719 of LNCS. (2003) 943–955
11. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proc. CPM'03. Volume 2676 of LNCS. (2003) 186–199
12. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proc. CPM'03. Volume 2676 of LNCS. (2003) 200–210

Table 4. Running times for files from several corpora.

	p-suffix array			PLCP array					
File	Radix	Bucket	Qsort	P-Kasai	Naive	length	avg *lcp*	avg *zlen*	%C
Artificial									
aaa.txt	0.0060	0.0044	175.3258	0.0023	17.0399	100000	49999.5	1.0	99.9
alphabet.txt	0.0060	0.0040	168.2780	0.0024	16.9446	100000	49999.5	26.0	99.9
random.txt	0.0350	0.0321	0.0947	0.0049	0.0102	100000	17.7	9.6	15.2
Canterbury									
alice29.txt	0.0492	0.0477	0.1174	0.0088	0.0163	152089	13.6	5.5	31.0
asyoulik.txt	0.0413	0.0379	0.0896	0.0070	0.0125	125179	13.4	5.9	27.5
cp.html	0.0078	0.0073	0.0175	0.0010	0.0036	24603	18.6	6.2	25.1
fields.c	0.0027	0.0025	0.0062	0.0004	0.0016	11150	18.6	5.2	30.0
grammar.lsp	0.0006	0.0006	0.0016	0.0001	0.0004	3721	13.5	5.0	30.2
lcet10.txt	0.1645	0.1544	0.4040	0.0534	0.0510	426754	15.8	5.6	30.5
plrabn12.txt	0.1862	0.1849	0.4506	0.0728	0.0537	481861	13.6	6.1	29.0
xargs.1	0.0008	0.0007	0.0021	0.0001	0.0004	4227	11.6	6.2	28.6
Large									
E.coli	2.9639	2.9823	7.0833	1.4862	1.0672	4638690	19.3	2.2	55.9
bible.txt	3.1321	3.0253	6.5019	1.1278	0.9816	4047392	18.0	5.5	31.6
world192.txt	2.0905	2.0194	3.9638	0.6075	0.7444	2473400	31.3	5.9	28.2
Misc.									
pi.txt	0.4567	0.4529	0.8871	0.2449	0.1174	1000000	11.0	3.6	36.6
Lightweight									
chr22.dna	31.8458	29.8698	4119.5402	12.6971	249.1254	34553758	1980.9	2.0	59.5
etext99	8.4958	7.8785	14.1768	1.9445	3.3963	6291456	94.6	5.6	29.7
howto	80.2950	77.1156	151.0006	13.6105	49.7969	39422105	273.8	4.8	34.9

13. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms **2**(1) (2004) 53–86
14. Kim, D.K., Jeon, J.E., Park, H.: An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In: Proc. SPIRE'04. Volume 3246 of LNCS. (2004) 138–149
15. Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In: Proc. FOCS'95. (1995) 631–637
16. Deguchi, S., Higashijima, F., Bannai, H., Inenaga, S., Takeda, M.: Parameterized suffix arrays for binary strings. In: Proc. PSC'08. (2008) 84–94
17. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. CPM'01. Volume 2089 of LNCS. (2001) 181–192
18. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Information Processing Letters **49**(3) (1994) 111–115
19. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40** (2004) 33–50