Algorithmica (2008) 52: 350–377
DOI 10.1007/s00453-007-9061-2

Linearized Suffix Tree: an Efficient Index Data Structure with the Capabilities of Suffix Trees and Suffix Arrays

Dong Kyue Kim · Minhwan Kim · Heejin Park

Received: 7 January 2006 / Accepted: 19 March 2007 / Published online: 24 October 2007

Abstract Suffix trees and suffix arrays are fundamental full-text index data structures to solve problems occurring in string processing. Since suffix trees and suffix arrays have different capabilities, some problems are solved more efficiently using suffix trees and others are solved more efficiently using suffix arrays. We consider efficient index data structures with the capabilities of both suffix trees and suffix arrays without requiring much space. When the size of an alphabet is small, enhanced suffix arrays are such index data structures. However, when the size of an alphabet is large, enhanced suffix arrays lose the power of suffix trees. Pattern searching in an enhanced suffix array takes $O(m|\Sigma|)$ time while pattern searching in a suffix tree takes $O(m \log |\Sigma|)$ time where m is the length of a pattern and Σ is an alphabet.

In this paper, we present *linearized suffix trees* which are efficient index data structures with the capabilities of both suffix trees and suffix arrays even when the size of an alphabet is large. A linearized suffix tree has all the functionalities of the enhanced suffix array and supports the pattern search in $O(m \log |\Sigma|)$ time. In a different point of view, it can be considered a practical implementation of the suffix tree supporting $O(m \log |\Sigma|)$-time pattern search.

In addition, we also present two efficient algorithms for computing suffix links on the enhanced suffix array and the linearized suffix tree. These are the first algorithms that run in $O(n)$ time without using the range minima query. Our experimental results show that our algorithms are faster than the previous algorithms.

D.K. Kim
Division of Electronics and Computer Engineering, Hanyang University, Seoul 133-791, South Korea

M. Kim
Division of Computer Engineering, Pusan National University, Busan 609-735, South Korea

H. Park (✉)
College of Information and Communications, Hanyang University, Seoul 133-791, South Korea
e-mail: hjpark@hanyang.ac.kr

Keywords Suffix trees · Suffix arrays · Index data structures · String algorithms

1 Introduction

1.1 Backgrounds

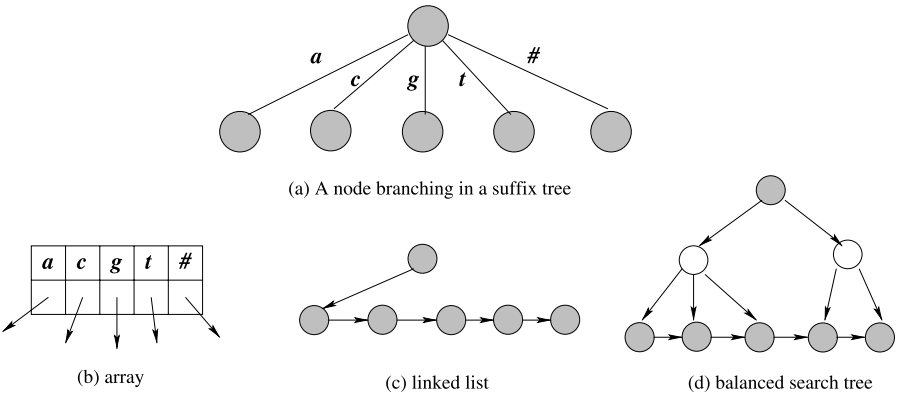
The full-text index data structure for a text incorporates the indices for all the suffixes of the text. It is used in numerous applications [18], which are exact string matching, computing matching statistics, finding maximal repeats, finding longest common substrings, and so on. Two fundamental full-text index data structures are the suffix tree and the suffix array.

The suffix tree of a text T due to McCreight [31] is a compacted trie of all suffixes of T. It was designed as a simplified version of Weiner's position tree [36]. If the size of an alphabet is small, the suffix tree for text T of length n, consumes $O(n)$ space and can be constructed in $O(n)$ time [5, 11, 12, 15, 27, 31, 35]. In addition, a pattern P of length m can be found in $O(m)$ time in the suffix tree. If the size of an alphabet is large, the size of the suffix tree and the search time in the suffix tree are affected by the data structure for node branching. There are several kinds of data structures for node branching: arrays, linked lists, balanced search trees, and hashing. Figure 1 shows arrays, linked lists and balanced search trees. If the data structure is an array, the size of the suffix tree is $O(n|\Sigma|)$ where Σ is an alphabet and searching the pattern P takes $O(m)$ time. If it is a linked list, the size of the suffix tree is $O(n)$ and searching the pattern P takes $O(m|\Sigma|)$ time. If the data structure is a balanced search tree, the size of the suffix tree is $O(n)$ and searching the pattern P takes $O(m \log|\Sigma|)$ time. In addition, the suffix tree supporting $O(m \log|\Sigma|)$-time search can be obtained by binary encoding of characters [6, 16]. However, suffix trees supporting $O(m \log|\Sigma|)$-time are not easy to implement and thus are not widely used in practice.

The suffix array of T due to Manber and Myers [29] and independently due to Gonnet et al. [16] is basically a sorted list of all the suffixes of T. The suffix array consumes only $O(n)$ space even though the size of an alphabet is large. However, it takes $O(n \log n)$ time to construct the suffix array (without using the suffix tree of T as an intermediate data structure) and $O(m + \log n)$ time for pattern search even with the lcp (longest common prefix) information when suffix arrays were introduced. Recently, almost at the same time, three different algorithms have been developed to directly construct the suffix array in $O(n)$ time by Kim et al. [25], Ko and Aluru [26], and Kärkkäinen and Sanders [21]. In addition, practically fast algorithms for suffix array construction have been developed [4, 9, 10, 23, 28, 30, 34].

Although suffix arrays are becoming more efficient, suffix trees still have merits because some problems such as matching statistics can be solved in a simple and efficient manner using suffix trees. Thus, there were efforts to augment suffix arrays so that they have some capabilities of suffix trees [7, 13, 16, 20]. However, they do not have all the capabilities of suffix trees.

Thus, there has been an effort to develop a full-text index data structure that has the capabilities of both suffix trees and suffix arrays without requiring much space.



(a) A node branching in a suffix tree

(b) array

(c) linked list

(d) balanced search tree

	Arrays		Balanced search trees		Linked lists
Branching time	$O(1)$	$<$	$O(\log \lvert\Sigma\rvert)$	$<$	$O(\lvert\Sigma\rvert)$
Search time	$O(m)$	$<$	$O(m \log \lvert\Sigma\rvert)$	$<$	$O(m\lvert\Sigma\rvert)$
Space	$O(n\lvert\Sigma\rvert)$	$>$	$O(n)$	$>^*$	$O(n)$

Fig. 1 Data structures for node branching. *Although both a balanced search tree and a linked list need $O(n)$ space, the linked list implementation has a smaller constant than the balanced search tree implementation

The enhanced suffix array of T developed by Abouelhoda et al. [1, 2] is such an index data structure when the size of an alphabet is small. It consists of a `pos` array, an `lcp` array, and a child table. The child table stores the parent-child relationship between the nodes in the suffix tree whose data structure for node branching is *a linked list*. Thus, on enhanced suffix arrays, every algorithm using the top-down or the bottom-up traversal of suffix trees can be run with a small and systematic modification. The top-down traversal can be done with the help of all the three arrays and the bottom-up traversal can be done using only the `pos` and `lcp` arrays. Since the child table is an array of only n elements and it is constructed very fast, the enhanced suffix array is time/space-efficient. Moreover, Abouelhoda et al. [1] defined suffix links on suffix arrays and developed algorithms for computing suffix links. Hence, even the algorithms using suffix links of the suffix tree can be run on enhanced suffix arrays.

However, when the size of an alphabet is large, enhanced suffix arrays lose the power of suffix trees. The pattern search in the enhanced suffix array of T takes $O(m|\Sigma|)$ time, while the pattern search in the suffix tree of T takes $O(m \log |\Sigma|)$ time. This is because the child table stores the information about the suffix tree whose data structure for node branching is a linked list.

1.2 Our Contribution

Our contribution is twofold. The major contribution is presenting the *linearized suffix tree* which is an efficient index data structure with the capabilities of both the suffix tree and the suffix array even when the size of the alphabet is large. The linearized suffix tree is a modification of the enhanced suffix array such that it supports

$O(m \log |\Sigma|)$-time pattern search. The other contribution is presenting two efficient algorithms for constructing suffix links on suffix arrays that are faster than previous algorithms.

1.2.1 The Linearized Suffix Tree

The linearized suffix tree consists of the `pos` array, the `lcp` array, and a new child table. This child table differs from the previous child table of the enhanced suffix array in that it stores the parent-child relationship between the nodes in the suffix tree whose data structure for node branching is *a complete binary tree*. With this new child table, one can search the pattern P in $O(m \log |\Sigma|)$ time. In a different point of view, it can be considered the a practical implementation of the suffix tree supporting $O(m \log |\Sigma|)$-time pattern search. The performance of linearized suffix tree is compared with those of the enhanced suffix array and the suffix trees in Fig. 2.

The main difficulty in developing the linearized suffix tree is that we cannot easily obtain the structural information of the complete binary tree directly from the `lcp` array. We overcome the difficulty by developing an *lcp extension* technique that extends the lcp to reflect the structure of the complete binary tree. This lcp extension technique requires the right-to-left scan of the bit representations of n integers to find the rightmost 1 in the bit representations. It seems to take $O(n \log n)$ time at first glance, however, it can be shown that it takes $O(n)$ time by resorting to amortized analysis.

1.2.2 Computing Suffix Links

We present two efficient methods for constructing suffix links in suffix arrays. Until now, the fastest running time for constructing suffix links has been either $O(n \log n)$ or $O(n)$ using the range minima query [1] where n is the length of the string. Both of our methods runs in $O(n)$ time without range minima query: One is for constant alphabet and the other is for integer alphabet. The experimental results show that our methods are faster than the previous methods.

1.3 Outline

We introduce some notations and definitions in Sect. 2. In Sect. 3, we introduce the enhanced suffix array. In Sect. 4, we describe the linearized suffix tree and its construction algorithm. In Sect. 5, we describe two efficient algorithms for constructing suffix links in suffix arrays. In Sect. 6, we present some experimental results. We conclude in Sect. 7.

	LST	ESA	STL	STB
Search time	$O(m \log \|\Sigma\|)$	$O(m \|\Sigma\|)$	$O(m \|\Sigma\|)$	$O(m \log \|\Sigma\|)$

Fig. 2 Comparison of time complexity of the linearized suffix tree with those of other full-text index data structures. The linearized suffix tree is denoted by LST and the enhanced suffix array by ESA. The suffix tree whose data structure for node branching is a linked list is denoted by STL and the suffix tree whose data structure for node branching is a balanced search tree by STB

2 Preliminaries

We first introduce some basic notations, then suffix arrays and suffix trees, and lcp-interval trees. Consider a string S of length n over an alphabet Σ. Let $S[i]$ for $1 \leq i \leq n$ denote the ith symbol of S. We assume that $S[n]$ is a special symbol # which is lexicographically larger than any other symbol in Σ. We denote by $S[i..j]$ the substring of S that starts at the ith symbol and ends at the jth symbol. Accordingly, a prefix (resp. suffix) of S is denoted by $S[1..k]$ (resp. $S[k..n]$) for some $1 \leq k \leq n$. The suffix starting at the ith symbol, i.e., $S[i..n]$, is called the ith *suffix* of S.

2.1 Suffix Arrays and Suffix Trees

The *suffix array* of S consists of $\texttt{pos}[1..n]$ and $\texttt{lcp}[1..n]$ arrays. The $\texttt{pos}[1..n]$ array is basically a sorted list of all the suffixes of S. That is, $\texttt{pos}[i]$ for $1 \leq i \leq n$ stores the start position of the ith smallest nonempty suffix of S. The $\texttt{lcp}[1..n]$ array stores the lengths of the longest common prefix of two adjacent suffixes in the $\texttt{pos}$ array. Specifically, we store in $\texttt{lcp}[i]$, $2 \leq i \leq n$, the length of the longest common prefix of $\texttt{pos}[i-1]$ and $\texttt{pos}[i]$, and we store -1 in $\texttt{lcp}[1]$ to indicate $\texttt{lcp}[1]$ is undefined. The suffix array of $caggtcagtcacggtatca\#$ is shown in Fig. 3(b).

The *suffix tree* of S is the compacted trie on the set of all suffixes of S. It has n leaves, each of which corresponds to a suffix of S. We denote the leaf corresponding to the ith suffix by leaf i. Figure 3(c) shows the suffix tree of $caggtcagtcacggtatca\#$. Each edge in the suffix tree of S has a label that is a substring of S, which is normally represented by the start position and the end position of the substring in S. Each node x also has a label, denoted by $label(x)$, that is the concatenation of the labels of the edges in the path from the root to the node. A *suffix link* is defined on every internal node of the suffix tree. The suffix link of the node whose label is $a\alpha$ is another internal node whose label is α where a and α are a symbol and a string, respectively. In Fig. 3(c), the suffix link of the node whose label is tca is the node whose label is ca.

2.2 Lcp-Interval Trees

The *lcp-interval tree* of S is a modification of the suffix tree of S such that each node x is replaced by the interval $[i..j]$ such that $\texttt{pos}[i..j]$ stores the suffixes in the subtree rooted at x. (Note that the suffixes in the subtree rooted at a node x are stored consecutively in the $\texttt{pos}$ array because all of them have the same prefix.)

Example 1 The lcp-interval tree of $caggtcagtcacggtatca\#$ is shown in Fig. 4. Consider the node whose label is ca in Fig. 3(c). The suffixes in the subtree of the node are stored in $\texttt{pos}[6..9]$ and thus the node is replaced by interval [6..9] in Fig. 4.

The intervals in lcp-interval trees are called *lcp-intervals*. The *first index* of an lcp-interval is the smallest index in the lcp-interval and the *last index* of an lcp-interval is the largest index in the lcp-interval. For example, the first index of $[i..j]$ is i and the last index of $[i..j]$ is j. We say that an lcp-interval *starts* at its first index and *ends* at its last index. For example, an lcp-interval $[i..j]$ starts at i and ends at j. The *length*

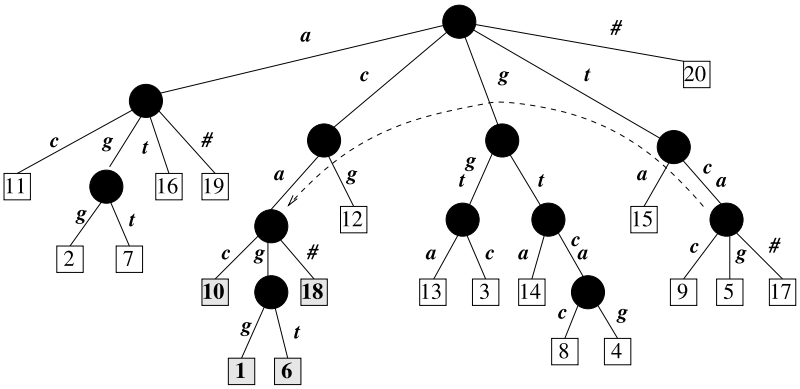
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
c	*a*	*g*	*g*	*t*	*c*	*a*	*g*	*t*	*c*	*a*	*c*	*g*	*g*	*t*	*a*	*t*	*c*	*a*	#

(a) The string

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
pos	11	2	7	16	19	**10**	**1**	**6**	**18**	12	13	3	14	8	4	15	9	5	17	20
lcp	−1	1	2	1	1	0	2	3	2	1	0	3	1	2	4	0	1	3	3	0

corresponding suffixes																				
	a	*a*	*a*	*a*	*a*	*c*	*c*	*c*	*c*	*c*	*g*	*g*	*g*	*g*	*g*	*t*	*t*	*t*	*t*	#
	c	*g*	*g*	*t*	#	*a*	*a*	*a*	*a*	*g*	*g*	*g*	*t*	*t*	*t*	*a*	*c*	*c*	*c*	
	g	g	*t*	*c*		*c*	*g*	*g*	#	*g*	*t*	*t*	*a*	*c*	*c*	*t*	*a*	*a*	*a*	
	g	*t*	*c*	*a*			*g*	*t*		*t*	*a*	*c*	*t*	*a*	*a*	*c*	*c*	*g*	#	
	t						*t*	*c*		*a*	*t*	*a*	*c*	*c*	*g*	*a*				

(b) The suffix array

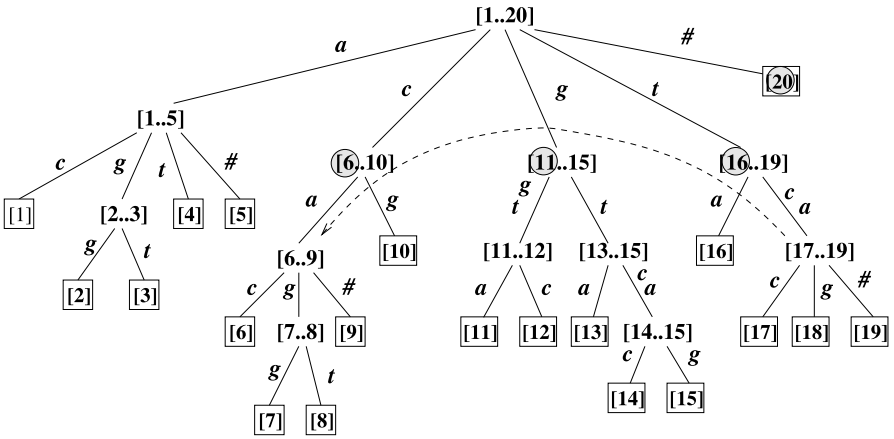


(c) The suffix tree

Fig. 3 The suffix array and the suffix tree of *caggtcagtcacggtatca*#

of an lcp-interval $[i..j]$ is defined as the number of indices in the lcp-interval, i.e., $j - i + 1$. For example, the length of [6..10] is $10 - 6 + 1 = 5$. *Suffix links* in lcp-interval trees are defined in the same way they are defined in suffix trees. In Fig. 4, the suffix link of the lcp-interval [17..19] whose label is tca is the lcp-interval [6..9] whose label is ca.

We present some useful properties of lcp-interval trees that are used later. We first show that the children of an internal lcp-interval $[i..j]$, $i < j$, form a partition of $[i..j]$. Let k denote the number of children of $[i..j]$ and x denote the corresponding node of $[i..j]$ in the suffix tree. Since each child interval of $[i..j]$ corresponds to a child node of x, the children of $[i..j]$ form a partition of $[i..j]$. That is, if we denote



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
–1	1	2	1	1	**0**	2	3	2	1	**0**	3	1	2	4	**0**	1	3	3	**0**

Fig. 4 The lcp-interval tree and the lcp array

the first index of the rth child of $[i..j]$ by c_r, $1 \le r \le k$, the children of $[i..j]$ can be represented as $[c_1(=i)..c_2-1]$, $[c_2..c_3-1]$, $\ldots$, $[c_k..j]$.

Example 2 Consider the root interval [1..20] in Fig. 4. The children of [1..20] are [1..5], [6..10], [11..15], [16..19], and [20], which form a partition of [1..20].

Since the children of $[i..j]$ form a partition of $[i..j]$, we have only to store $c_2, c_3, \ldots, c_k$ to enumerate all the children of $[i..j]$. Thus, we get the following lemma.

Lemma 1 *Given an lcp-interval* $[i..j]$, *we can enumerate all the children of an internal lcp-interval* $[i..j]$ *if we store the first indices of the children of* $[i..j]$ *except the first child.*

We show that the first indices of the children of $[i..j]$ except the first child (i.e., $c_2, c_3, \ldots, c_k$) correspond to the indices of the smallest value in the $\mathtt{lcp}[i+1..j]$ array. Let l denote the length of $label(x)$. Since $label(x)$ is the longest common prefix of the suffixes in $\mathtt{pos}[i..j]$, $\mathtt{lcp}[c_2] = \mathtt{lcp}[c_3] = \cdots = \mathtt{lcp}[c_k] = l$ and $\mathtt{lcp}[q] > l$ for all $i+1 \le q \ne c_2, c_3, \ldots, c_k \le j$. Hence, $c_2, c_3, \ldots, c_k$ correspond to the indices of the smallest value in the $\mathtt{lcp}[i+1..j]$ array and we get the following lemma.

Lemma 2 *The first indices of the children of* $[i..j]$ *except the first child correspond to the indices of the smallest value in* $\mathtt{lcp}[i+1..j]$.

Example 3 Consider the lcp-interval [1..20] in Fig. 4. In `lcp`[2..20], the smallest value 0 is stored in `lcp`[6], `lcp`[11], `lcp`[16], and `lcp`[20]. The indices 6, 11, 16, and 20 correspond to the first indices of the children [6..10], [11..15], [16..19] and [20], respectively.

3 The Enhanced Suffix Array

The enhanced suffix array of S due to Abouelhoda et al. [1, 2] consists of the `pos`[1..n] array, the `lcp`[1..n] array, and the child table `cldtab`[1..$n-1$] of S. The `pos` and `lcp` arrays were introduced in the previous section and it is known that they can be constructed in linear time [21, 22, 25, 26]. Thus, we only describe the child table in this section. When the child table was introduced [1, 2], it was explained using `up`, `down`, and `nextlindex` arrays. In this paper, we explain the child table in a different way which is, we believe, more intuitive. We define the child table in Sect. 3.1 and show how to construct the child table in Sect. 3.2.

3.1 Definition of the Child Table

The child table `cldtab`[1..$n-1$] of S stores the parent-child relationship between each lcp-interval and its children in the lcp-interval tree of S. We first give a rough description of the child table. For each internal lcp-interval $[i..j]$, we store the first indices of its children except the first child in the child table. (From these values, we can enumerate all the children of $[i..j]$ by Lemma 1.) We denote the first indices by $c_2 < c_3 < \cdots < c_k$. The child table stores c_i's in a linked list manner: Either `cldtab`[i] or `cldtab`[j] stores c_2, `cldtab`[c_2] stores c_3, `cldtab`[c_3] stores c_4, and so on.

Example 4 The enhanced suffix array and the lcp-interval tree of *caggtcagtca cggtatca#* are shown in Fig. 5. In the lcp-interval tree, for the root [1..20], `cldtab`[1] stores 6, `cldtab`[6] stores 11, `cldtab`[11] stores 16, and `cldtab`[16] stores 20.

We introduce the formal definition of the child table. Let next(i, j) denote the first index of the next sibling of an lcp-interval $[i..j]$ and child(i, j) denote the first index of the second child of $[i..j]$. In Fig. 5, next(11, 15) = 16 and child(11, 15) = 13. The child table stores next(i, j) and child(i, j) for each lcp-interval $[i..j]$. To store next(i, j) and child(i, j), `cldtab`[i] and `cldtab`[j] are used in the following way:

- *If* $[i..j]$ *is neither the last child nor the root*: We store next(i, j) in `cldtab`[i] and child(i, j) in `cldtab`[j]. The next(i, j) is not defined if $[i..j]$ is the first child and child(i, j) is not defined if $[i..j]$ is a leaf.
- *If* $[i..j]$ *is the last child or the root*: We store child(i, j) in `cldtab`[i] if $[i..j]$ is not a leaf. The next(i, j) is not defined.

Example 5 In Fig. 5, consider the lcp-intervals [11..15], [6], and [13..15]. Since the lcp-interval [11..15] is neither the last child nor the root and it is neither the

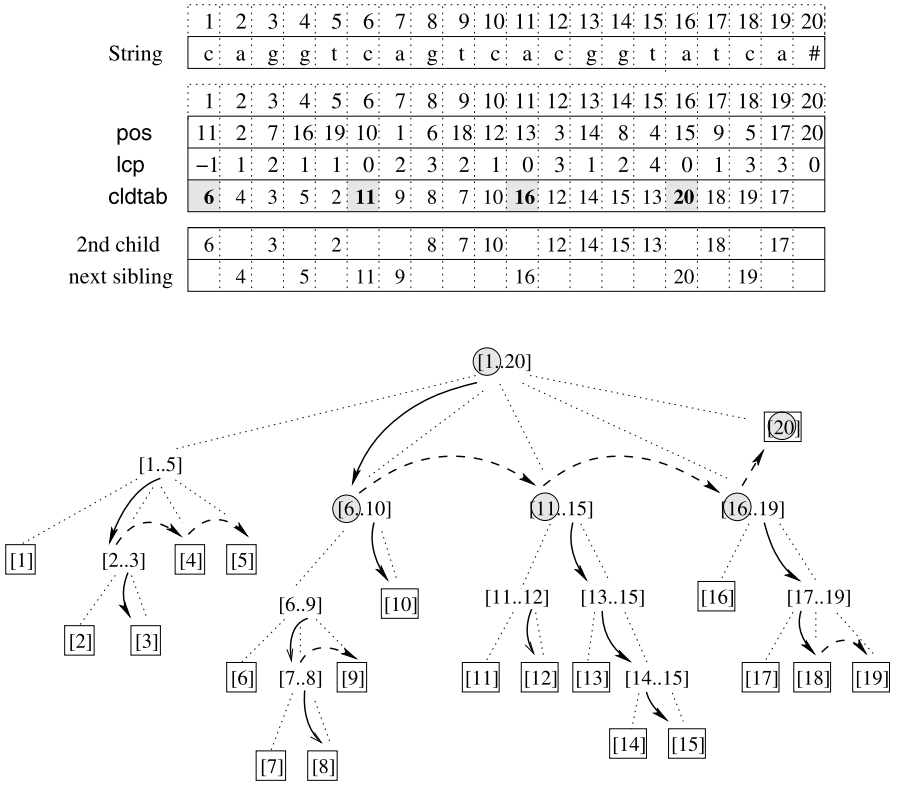


Fig. 5 The enhanced suffix array and the lcp-interval tree of *caggtcagtcacggtatca*#. In the lcp-interval tree, the first indices of the next siblings are pointed to by *dashed arrows* and the first indices of the second children are pointed to by *solid arrows*

first child nor the leaf, both the next sibling and the second child are defined. The next$(11, 15)$ $(= 16)$ is stored in `cldtab`[11] and the child$(11, 15)$ $(= 13)$ is stored in `cldtab`[15]. Since the lcp-interval [6] is the first child and it is a leaf, none of the next sibling and the second child is defined. Since the lcp-interval [13..15] is the last child and it is not a leaf, only the second child is defined and child$(13, 15)$ $(= 14)$ is stored in `cldtab`[13].

3.2 Construction of the Child Table

The child table can be constructed directly from the `lcp` array in $O(n)$ time without constructing the lcp-interval tree explicitly. Procedures CHILD and NEXT in Fig. 6 due to Abouelhoda et al. [1, 2] construct the child table. Let $[\alpha..i]$ denote the *longest lcp-interval ending at* i and $[i..\beta]$ denote the *longest lcp-interval starting at* i. For example, $[\alpha..15]$ is [11..15] and $[6..\beta]$ is [6..10] in Fig. 5. Due to Abouelhoda et al. [1, 2], each entry `cldtab`$[i]$, $1 \leq i \leq n-1$, stores either child$(\alpha..i)$, child$(i..\beta)$, or next$(i..\beta)$. Procedure CHILD computes every entry `cldtab`$[i]$,

```
Procedure CHILD
1:   lastIndex := −1;
2:   push(1);
3:   for i := 2 to n + 1 do
4:     while lcp[i] < lcp[top()] do
5:       lastIndex := pop();
6:       if lcp[i] < lcp[top()] and lcp[top()] ≠ lcp[lastIndex] then
7:         cldtab[top()] := lastIndex
8:     if lastIndex ≠ −1 then
9:       cldtab[i − 1] := lastIndex;
10:      lastIndex := −1;
11:    push(i)
end

Procedure NEXT
1:   push(1);
2:   for i := 2 to n do
3:     while lcp[i] < lcp[top()] do
4:       pop();
5:     if lcp[i] = lcp[top()] then
6:       lastIndex := pop();
7:       cldtab[lastIndex] := i
8:     push(i)
end
```

Fig. 6 Procedures CHILD and NEXT. Procedure CHILD assumes a sentinel value -2 is stored in $\texttt{lcp}[n+1]$

$1 \le i \le n-1$, that stores either child$(\alpha..i)$ or child$(i..\beta)$ in $O(n)$ time, and procedure NEXT computes every entry $\texttt{cldtab}[i]$, $1 \le i \le n-1$, that stores next$(i..\beta)$ in $O(n)$ time. The procedure CHILD uses the fact that the child(i, j) for each lcp-interval $[i..j]$ is the index of the leftmost smallest value in $\texttt{lcp}[i+1..j]$, which can be easily deduced from Lemma 2. The procedure NEXT uses the fact that $\texttt{lcp}[i] = \texttt{lcp}[\text{next}(i, \beta)]$. The analyses of running time of the procedures, in addition to the proof of their correctness, are given in [1, 2].

4 The New Child Table

The linearized suffix tree of S consists of the $\texttt{pos}[1..n]$ array, the $\texttt{lcp}[1..n]$ array and the new child table $\texttt{cldtab}[1..n-1]$ of S. We define the new child table in Sect. 4.1, show that the new child table is well-defined in Sect. 4.2, and describe how to construct the new child table in Sect. 4.3. In Sect. 4.4, we show how to search a pattern on the new child table. In this section, we will just say 'child table' instead of saying 'new child table.'

4.1 Definition

The child table cldtab[1..$n-1$] stores the parent-child relationship between the lcp-intervals in the *modified* lcp-interval tree whose data structure for node branching is the *complete binary tree*. We first describe the shape of the complete binary tree for the children of an lcp-interval $[a..b]$. In this complete binary tree, the interval $[a..b]$ is the root, all the children of $[a..b]$ are leaves, and the other internal nodes are appropriately generated to form the complete binary tree. Let k denote the number of children of $[a..b]$. Let d and k' be integers such that $k = 2^d + k'$ for some $1 \leq k' \leq 2^d$. We generate a complete binary tree of depth $d+1$ having $2k'$ leaves in level 0. Each ith $(1 \leq i \leq 2k')$ child of $[a..b]$ is the ith leftmost leaf in level 0 and each ith $(2k'+1 \leq i \leq 2^d + k')$ child of $[a..b]$ is the $(i-k')$th leftmost node in level 1.

Example 6 Consider the lcp-interval [1..20] in Fig. 7(a). The lcp-interval [1..20] has five children [1..5], [6..10], [11..15], [16..19], and [20]. Since $5 = 2^2 + 1$, we construct the complete binary tree of depth 3 (Fig. 7(b)). The first two children [1..5] and [6..10] are the leaves in level 0. The other three children [11..15], [16..19], and [20] are the leaves in level 1. Three internal nodes [1..10], [1..15], and [16..20] are newly created.

We show how to store a modified lcp-interval tree in a child table. For each internal lcp-interval $[i..j]$, we store the first index of its second child in the child table. (Note that every internal lcp-interval in a modified lcp-interval tree has two children and we can enumerate the children of $[i..j]$ by storing the first index of its second child.) Let child(i, j) denote the first index of the second child of $[i..j]$. The child table stores child(i, j) in the following way:

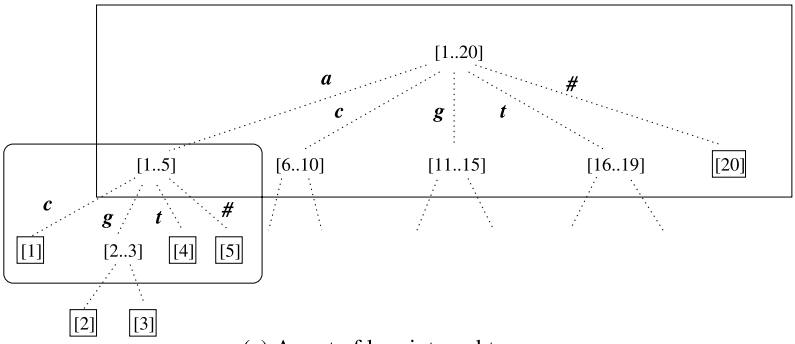
- If $[i..j]$ is the first child of its parent, we store child(i, j) in cldtab$[j]$.
- Otherwise (if $[i..j]$ is the second child or the root), we store child(i, j) in cldtab$[i]$.

In Fig. 7, [1..15] is the first child and thus child$(1, 15)$ $(= 11)$ is stored in cldtab[15]. The lcp-interval [1..20] is the root and thus child$(1, 20)$ $(= 16)$ is stored in cldtab[1]. The lcp-interval [16..20] is the second child and thus child$(16, 20)$ $(= 20)$ is stored in cldtab[16].

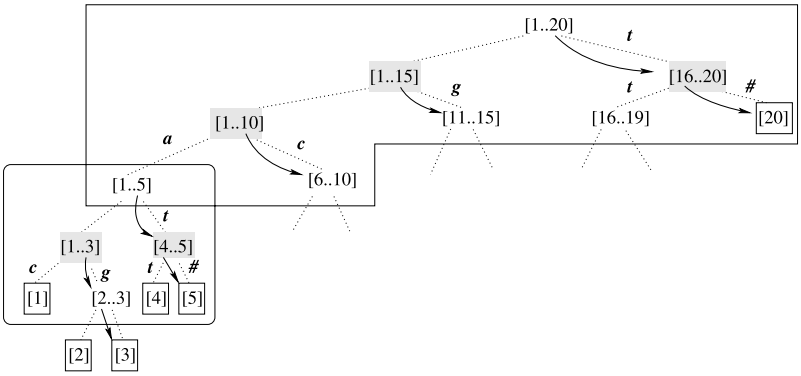
Consider the number of indices to be stored in the child table. Since the number of leaves in the modified lcp-interval tree is n and the modified lcp-interval tree is a binary tree, the number of internal lcp-intervals in the modified lcp-interval tree is $n-1$. Since we store child(i, j) for every internal lcp-interval $[i..j]$, we store $n-1$ indices in cldtab[1..$n-1$]. We show we store the indices in cldtab[1..$n-1$] without conflicts in the next subsection.

4.2 Justification of Avoiding Conflicts in the Child Table

We first show that cldtab$[i]$ is not used by any lcp-intervals except the longest lcp-interval starting at i and the longest lcp-interval ending at i in Lemma 4 and then show only the longest of them uses cldtab$[i]$ in Lemmas 5 and 6. Recall that



(a) A part of lcp−interval tree



(b) A part of modified lcp−interval tree

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
String	c	a	g	g	t	c	a	g	t	c	a	c	g	g	t	a	t	c	a	#

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
pos	11	2	7	16	19	10	1	6	18	12	13	3	14	8	4	15	9	5	17	20
lcp	−1	1	2	1	1	0	2	3	2	1	0	3	1	2	4	0	1	3	3	0
cldtab	16	3	2	5	4	10	8	7	9	6	13	12	14	15	11	20	19	18	17	

(c) The linearized suffix tree

Fig. 7 A part of the lcp-interval tree, its corresponding part of the modified lcp-interval tree that uses the complete binary tree for node branching, and the linearized suffix tree of *caggtcagtcacggtatca#*

the longest lcp-interval ending at i is denoted by $[\alpha..i]$ and the longest lcp-interval starting at i is denoted by $[i..\beta]$.

We first consider the inclusion between two lcp-intervals $[i..j]$ and $[k..l]$ in an lcp-interval tree. Since $[i..j]$ and $[k..l]$ are lcp-intervals, either:

(i) $[i..j]$ is an ancestor of $[k..l]$;
(ii) $[i..j]$ is a descendant of $[k..l]$; or
(iii) $[i..j]$ is neither an ancestor nor a descendant of $[k..l]$.

In case of (i), $[i..j]$ includes $[k..l]$. In case of (ii), $[k..l]$ includes $[i..j]$. In case of (iii), $[i..j]$ and $[k..l]$ are disjoint. Thus, we get the following lemma.

Lemma 3 *Given two lcp-intervals in an lcp-interval tree, either one includes the other or they are disjoint. If one includes the other, the one including the other is an ancestor of the other.*

We show that `cldtab`$[i]$ is not used by any lcp-intervals except $[\alpha..i]$ and $[i, \beta]$ in the next lemma.

Lemma 4 *The* `cldtab`$[i]$, $1 \leq i \leq n-1$, *is not used by any lcp-intervals except the longest lcp-interval starting at* i ($[i..\beta]$) *and the longest lcp-interval ending at* i ($[\alpha..i]$).

Proof By definition of the child table, `cldtab`$[i]$ is not used by any lcp-intervals that neither start nor end at i. Thus, we have only to show that every lcp-interval starting at i except $[i..\beta]$ does not use `cldtab`$[i]$ and that every lcp-interval ending at i except $[\alpha..i]$ does not use `cldtab`$[i]$. We only prove the former case by showing that an lcp-interval $[i..j]$ ($\neq [i..\beta]$) is the first child of its parent, which does not use `cldtab`$[i]$ by definition of the child table. (Proving the latter is symmetric. In that case, we show that an lcp-interval $[h..i]$ ($\neq [\alpha..i]$) is the second child of its parent, which does not use `cldtab`$[i]$.)

We show that $[i..j]$ is the first child by contradiction. Suppose that $[i..j]$ is not the first child of its parent. Then, $[i..j]$ is either the root or the second child of its parent. Since $[i..j]$ is not $[i..\beta]$, it cannot be the root and thus it is the second child. Let $[h..i-1]$ denote the first child of the parent of $[i..j]$. Since $[i..\beta]$ includes $[i..j]$, $[i..\beta]$ is an ancestor of $[i..j]$ by Lemma 3. Since $[h..i-1]$ and $[i..j]$ have the same parent, $[i..\beta]$ is also an ancestor of $[h..i-1]$. Since $[i..\beta]$ is an ancestor of $[h..i-1]$, $[i..\beta]$ includes $[h..i-1]$ by Lemma 3, which a contradiction. Hence, $[i..j]$ is the first child. □

Lemma 5 *If the length of* $[i..\beta]$ *is longer than that of* $[\alpha..i]$, `cldtab`$[i]$ *stores* child$(i..\beta)$. *If the length of* $[\alpha..i]$ *is longer than that of* $[i..\beta]$, `cldtab`$[i]$ *stores* child$(\alpha..i)$.

Proof We only prove the first case. Proving the second case is symmetric. We first show $[i..\beta]$ stores child$(i..\beta)$ in `cldtab`$[i]$ and then $[\alpha..i]$ stores nothing in `cldtab`$[i]$. Since $[i..\beta]$ is the longest lcp-interval starting at i, it is not the first child. (If it is the first child, its parent becomes a longer lcp-interval starting at i than $[i..\beta]$.) Since $[i..\beta]$ is longer than $[\alpha..i]$, it is not a leaf. Since $[i..\beta]$ is not the first child and not a leaf, it stores child$(i..\beta)$ in `cldtab`$[i]$ by definition of the child table. We show $[\alpha..i]$ stores nothing in `cldtab`$[i]$ by showing that $[\alpha..i]$ is leaf $[i]$. Since $[\alpha..i]$ and $[i..\beta]$ are not disjoint, one of them should includes the other by Lemma 3.

Since $[i..\beta]$ is not a leaf, $[\alpha..i]$ cannot includes $[i..\beta]$ and thus $[i..\beta]$ should include $[\alpha..i]$, which implies $[\alpha..i]$ is leaf $[i]$. Hence, $[\alpha..i]$ stores nothing in $\texttt{cldtab}[i]$. □

Lemma 6 *If* $[\alpha..i]$ *and* $[i..\beta]$ *have the same length, both of them are leaf* $[i]$ *that stores nothing in* $\texttt{cldtab}[i]$.

Proof Since $[\alpha..i]$ and $[i..\beta]$ are not disjoint, one of them includes the other by Lemma 3. Since the lengths of $[\alpha..i]$ and $[i..\beta]$ are the same, both of them should be leaf $[i]$. □

By Lemmas 4, 5, and 6, we get the following theorem.

Theorem 1 *Each* $\texttt{cldtab}[i]$, $1 \le i \le n-1$, *stores only one of* child$(\alpha..i)$ *and* child$(i..\beta)$ *where* $[\alpha..i]$ *is the longest lcp-interval ending at* i *and* $[i..\beta]$ *is the longest lcp-interval starting at* i.

4.3 Construction

We extend the `lcp` array using the `depth` array, which is a temporary array used to construct the child table. The $\texttt{depth}[1..n-1]$ array is defined as follows. Consider the complete binary tree rooted at an interval $[a..b]$. Let $c_1, c_2, \ldots, c_k$ denote the first indices of the children of $[a..b]$. Then, we store in $\texttt{depth}[c_i]$ $(2 \le i \le k)$ the depth of the lowest common ancestor of two adjacent child intervals $[c_{i-1}..c_i - 1]$ and $[c_i..c_{i+1} - 1]$ in the complete binary tree.

Example 7 In Fig. 8, the lowest common ancestor of [2..3] and [4] is [1..5] whose depth is 0 and thus we store 0 in $\texttt{depth}[4]$. Similarly, we store 1 in $\texttt{depth}[2]$ and $\texttt{depth}[5]$.

Each element of the `depth` array can be stored in $\lceil \log\log|\Sigma| \rceil$ bits because the largest value that $\texttt{depth}[i]$, $1 \le i \le n$, may store is $\lceil \log|\Sigma| \rceil - 1$ due to the fact that a complete binary tree may have at most $|\Sigma|$ leaves.

The extended `lcp` array (i.e., the `lcp` array and the `depth` array) reflects the structure of the lcp-interval tree that uses the complete binary tree for node branching. The child(i, j) of $[i..j]$ is the index k, $i+1 \le k \le j$, minimizing $\texttt{lcp}[k]\|\texttt{depth}[k]$ where $\texttt{lcp}[k]\|\texttt{depth}[k]$ is the bit concatenation of $\texttt{lcp}[k]$ and $\texttt{depth}[k]$. We will denote $\texttt{lcp}[k]\|\texttt{depth}[k]$ by $\texttt{extlcp}[k]$. In Fig. 8, child$(1, 5)$ is 4 and 4 is the index that minimizes $\texttt{extlcp}[i]$ because $\texttt{extlcp}[4] = 4$ and $\texttt{extlcp}[2] = \texttt{extlcp}[5] = 5$.

Thus, we can compute the child table of the linearized suffix tree in $O(n)$ time by running the procedure CHILD in Fig. 6 with a small modification that each `lcp` is replaced by `extlcp`. For example, '**while** $\texttt{lcp}[i] < \texttt{lcp}[top()]$ **do**' in line 4 of the procedure CHILD is replaced by '**while** $\texttt{extlcp}[i] < \texttt{extlcp}[top()]$ **do**.'

We only have to show how to compute the `depth` array in $O(n)$ time. The computation of the `depth` array consists of the following two steps:

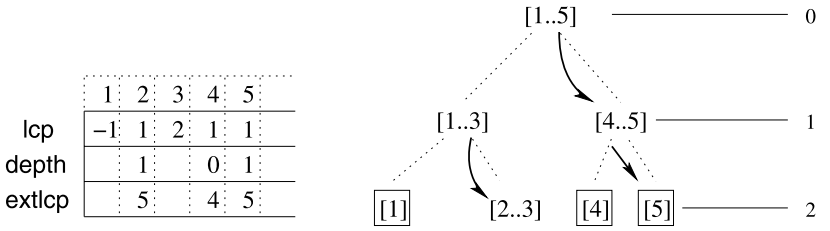


Fig. 8 The extended `lcp` array when $|\Sigma| = 16$

(1) *Compute the number of children for every lcp-interval*: We can do this in $O(n)$ time by running the procedure NEXT given in Fig. 6 on the `lcp` array.
(2) *For each lcp-interval* $[a..b]$*, we compute* $\texttt{depth}[c_i]$ *for* $2 \leq i \leq k$ *where* c_i *is the first index of the* i*th child of* $[a..b]$: We describe how to compute $\texttt{depth}[c_i]$ when $k = 2^d$ for some d. (Computation of $\texttt{depth}[c_i]$ when $k \neq 2^d$ for any d is slightly different.) Recall that $\texttt{depth}[c_i]$ $(2 \leq i \leq k)$ stores the depth of the lowest common ancestor of two adjacent child intervals $[c_{i-1}..c_i - 1]$ and $[c_i..c_{i+1} - 1]$. Since the depth of the lowest common ancestor is $d - L_i - 1$ where L_i is the level of the lowest common ancestor, we describe how to compute L_i. The L_i corresponds to the number of tailing 0's in the bit representation of $i - 1$. For example, every odd numbered child has no tailing 0's and thus the level of it is 0. We consider the running time of computing $\texttt{depth}[c_i]$ for $2 \leq i \leq k$. To compute them, we have to scan the bit representation from the right until we reach the rightmost 1 for all integers $2, \ldots, k$. One can show this takes $O(k)$ time overall by resorting to the amortized analysis which is very similar to the one used to count the bit-flip operations of a binary counter [8]. Overall, this step takes $O(n)$ time.

Lemma 7 *The* $\texttt{depth}[1..n-1]$ *array can be constructed in* $O(n)$ *time.*

Theorem 2 *The new child table* `cldtab` *can be constructed in* $O(n)$ *time.*

We consider the space consumption of the new child table and the linearized suffix tree. The new child table has $n - 1$ integer entries that is the same number of entries the original child table in the enhanced suffix array requires. Thus, the linearized suffix tree requires the same space as the enhanced suffix array. It should be noted that the additional `depth` array of n entries is not included in the linearized suffix tree because it is discarded after the new child table is constructed.

4.4 Pattern Searching

Searching a pattern on a linearized suffix tree is essentially the same as searching a pattern on a modified lcp-interval tree. We traverse down the lcp-interval tree with the pattern until we reach the first node whose label includes the pattern as its prefix. Then, the leaves in the subtree rooted at the node store the positions in the text where the pattern occurs. Consider searching a pattern `ag` on the modified lcp-interval tree

in Fig. 7(b). We start at the root [1..20]. We find the leaf in the complete binary tree rooted at [1..20] whose incoming edge label starts with a. Since the incoming edge label of [1..5] is a, we move to [1..5]. Then, we find the leaf in the complete binary tree rooted at [1..5] whose incoming edge label starts with g. Since the incoming edge label of [2..3] is g, we move to [2..3]. In this way, we find the lcp-interval [2..3] whose label is ag.

It remains to show how to find the leaf whose first symbol of its incoming edge label is the same as a symbol of a pattern. Consider finding the child of $[a..b]$ whose incoming edge label starts with the ith symbol of the pattern which is denoted by $p[i]$. Without loss of generality, we assume that child(a, b) is stored in $\texttt{cldtab}[a]$. First, we compare $p[i]$ with $S[\texttt{pos}[\texttt{cldtab}[a]]+i-1]$. If $p[i]$ is smaller, we move to the left child of $[a..b]$. Otherwise, we move to the right child of $[a..b]$. In this way, we traverse down the complete binary tree rooted at $[a..b]$ until we reach a leaf node.

Example 8 Consider finding the child of [1..5] whose incoming edge label starts with g, when we search a pattern ag on the linearized suffix tree in Fig. 7(b). First, we compare g with $S[\texttt{pos}[4]+1]$ that is t. Since g is smaller than t, we move to [1..3]. Then, we compare g with $S[\texttt{pos}[2]+1]$ that is g. Since g is the same as $S[\texttt{pos}[2]+1]$, we move to [2..3].

It should be noted that we do not know the depth of the complete binary tree during the pattern search because the depth array is not available after the child table is constructed. Instead, we can determine whether we have arrived at a leaf using the lcp array. Let $[i..j]$ be an lcp-interval in the complete binary tree. If $\texttt{lcp}[i] = \texttt{lcp}[\texttt{cldtab}[i]]$ in case of $[i..j]$ being a right child (if $\texttt{lcp}[i] = \texttt{lcp}[\texttt{cldtab}[j]]$ in case of $[i..j]$ being a left child), $[i..j]$ is not a leaf and it is a leaf, otherwise. Since the complete binary tree is of depth $\log|\Sigma|$ in the worst case, we can find the child in $O(\log|\Sigma|)$ time and thus we get the following theorem.

Theorem 3 *The linearized suffix tree supports $O(m\log|\Sigma|)$-time pattern search.*

5 Computing Suffix Links

Recall that the suffix link of an lcp-interval whose label is $a\alpha$, is the lcp-interval whose label is α where a and α is a symbol and a string, respectively. For example, in Fig. 9, the suffix link of [2..3] whose prefix is ag, is [11..15] whose prefix is g. The suffix links are stored in $\texttt{suflink}[1..n]$. Let the suffix link of an lcp-interval $[i..j]$ be the lcp-interval $[p..q]$. When we store the suffix link of lcp-interval $[i..j]$, we only store the first and the last indices of the lcp-interval $[p..q]$, i.e., p and q. We store them in $\texttt{suflink}[x]$ where x is child(i, j). Since the index child(i, j) is unique for every lcp-interval $[i..j]$, every lcp-interval can store its suffix link in a unique place in $\texttt{suflink}[1..n]$. (Note that the index child(i, j) is not the same when we use the enhanced suffix array and when we use the linearized suffix tree.)

Example 9 In Fig. 9, we store the suffix link of an lcp-interval [6..9], which is [1..5]. If we use the enhanced suffix array, child(6, 9) is 7 (Fig. 5) and thus we store 1 and 5

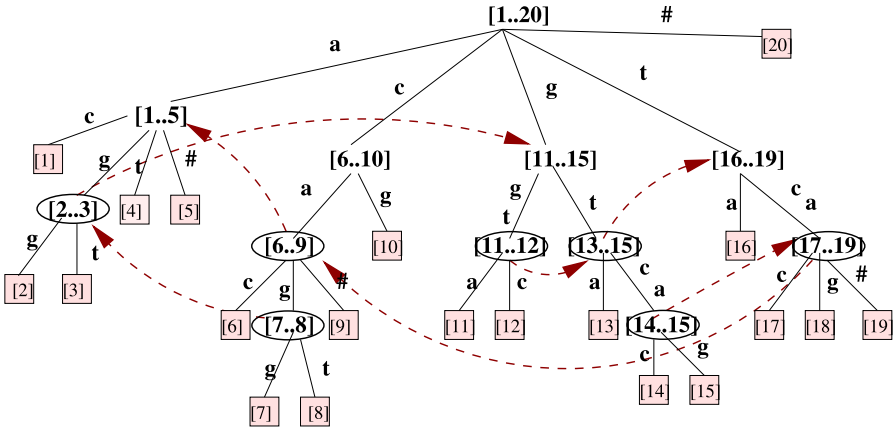


Fig. 9 Suffix links in the lcp-interval tree

in `suflink`[7]. If we use the linearized suffix tree, child(6, 9) is 9 (Fig. 7) and thus we store 1 and 5 in `suflink`[9].

When the subtree rooted at the suffix link $[p..q]$ of $[i..j]$ is visited, one has to know whether $[p..q]$ is the left or right child of its parent. Abouelhoda et al. [1, 2] showed that one can find it by checking whether (1) $p < \texttt{cldtab}[p] \leq q$ or (2) $p < \texttt{cldtab}[q] \leq q$: if $p < \texttt{cldtab}[p] \leq q$, $[p..q]$ is the right child and if $p < \texttt{cldtab}[q] \leq q$, it is the left child. We describe the essential idea of the proof that only the inequality (1) is satisfied when $[p..q]$ is the right child. (Similarly, one can show only the inequality (2) is satisfied when $[p..q]$ is the left child.) If $[p..q]$ is the right child $\texttt{cldtab}[p]$ stores child(p, q) and thus $p < \texttt{cldtab}[p] \leq q$. Now, we show that $\texttt{cldtab}[q] \leq p$. Since $[p..q]$ is the right child, $\texttt{cldtab}[q]$ is used by the lowest ancestor of $[p..q]$ that is the left child of its parent and thus $\texttt{cldtab}[q] \leq p$. Hence, only the inequality (1) is satisfied.

Previously, Abouelhoda et al. [2] developed two algorithms for constructing `suflink`$[1..n]$. One runs in $O(n \log n)$ time and the other runs in $O(n)$ with the help of the range minima query. We improve the previous algorithms to construct `suflink`$[1..n]$ in $O(n)$ time without range minima query. We present two algorithms for suffix link computation, which are about 3-4 times faster than the previous algorithms. We first describe an algorithm for constant alphabets and then an algorithm for integer alphabets.

5.1 Constant Alphabets

If the size of an alphabet is constant, we can compute the suffix links by performing a preorder depth-first traversal on the lcp-interval tree using the linearized suffix tree. For example, in the lcp-interval tree in Fig. 9, suffix links of the nodes in the leftmost subtree of the root ([1..5] and [2..3]) are computed first, then the nodes in the second leftmost subtree ([6..10], [6..9], and [7..8]), and so on. This is similar to McCreight's,

Ukkonen's, and Giegerich and Kurtz's computation of suffix links [14, 31, 35]. During the traversal, every time we encounter an lcp-interval $[i..j]$, we compute its suffix link as follows:

(1) *Find the parent of* $[i..j]$: Let $[u..v]$ denote the parent of $[i..j]$. Since we are performing preorder traversal of the lcp-interval tree, we can find $[u..v]$ in $O(1)$ time.
(2) *Find the suffix link of* $[u..v]$: We can find the suffix link in $O(1)$ time because we compute the suffix links by performing the preorder traversal and thus the suffix link of $[u..v]$ was already computed.
(3) *Find the suffix link of* $[i..j]$: We traverse down the lcp-interval tree from the suffix link of $[u..v]$ using the incoming edge label of $[i..j]$ to find the suffix link of $[i..j]$.

Example 10 We consider the computation of the suffix link of [7..8] in Fig. 9. We first find the parent of [7..8], which is [6..9], and then find the suffix link of [6..9], which is [1..5]. Since the incoming edge label of [7..8] is g, we traverse down from [1..5] with g. In this way, we can find [2..3] that is the suffix link of [1..5].

The next lemma shows that this algorithm runs in linear time.

Lemma 8 *The above algorithm computes all the suffix links in linear time for constant alphabet.*

Proof The overall cost of steps (1) and (2) for all nodes is $O(n)$ because each step takes $O(1)$ time per node. We show the overall cost of steps (3) for all nodes is $O(n|\Sigma|\log|\Sigma|)$. We show this by showing that the amortized cost of steps (3) for all nodes in the subtree rooted at a child of the root (i.e., all nodes whose labels start with the same symbol) is $O(n\log|\Sigma|)$, which implies the overall cost of steps (3) is $O(n|\Sigma|\log|\Sigma|)$ because the number of children of the root is the size of alphabet. Consider the amortized cost of steps (3) for all nodes in the subtree rooted at a child of the root. Note that during the step (3) for a node, we only traverse down the suffix tree from the suffix link of its parent to its suffix link, and that no two nodes in the subtree rooted at a child of the root have the same suffix link. Thus, we do not traverse down any part of the lcp-interval tree more than once and the amortized cost corresponds to the cost of depth-first search in the lcp-interval tree, which is $O(n\log|\Sigma|)$. Hence, the overall cost of steps (3) for all nodes is $O(n|\Sigma|\log|\Sigma|)$. Since the size of alphabet is constant, the above algorithm runs in $O(n)$. □

However, if we are to compute the suffix links for integer alphabets, this approach takes $O(n^2\log n)$ time.

5.2 Integer Alphabets

We describe how to compute suffix links for integer alphabets in $O(n)$ time without using the range minima query. Let a *k-interval* denote an internal lcp-interval the length of whose label is k. Since the suffix links of 1-intervals is trivially the 0-interval

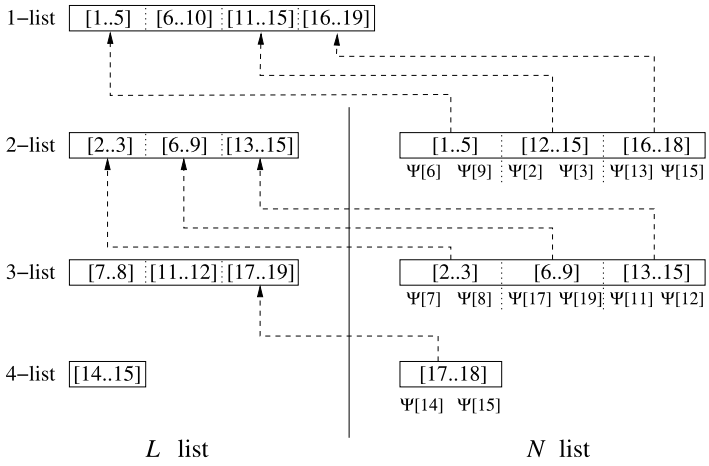


Fig. 10 Computing suffix links for integer alphabets

$[1..n]$, we only consider the suffix links of k-intervals, $k \geq 2$. We start with presenting a key property that enables us to compute the suffix links in $O(n)$ time. Let $[u..v]$ be a k-interval whose label is $a\alpha$. Consider the interval $[\Psi[u]..\Psi[v]]$ where $\Psi[i] = \mathtt{pos}^{-1}[\mathtt{pos}[i]+1]$. From the definition of Ψ, all the suffixes in $\mathtt{pos}[\Psi[u]..\Psi[v]]$ have the prefix α. We call this interval the Ψ-*interval* of $[u..v]$. (The Ψ-interval of $[u..v]$ may not be an lcp-interval but it does not matter.) From the fact that all the suffixes in $\mathtt{pos}[\Psi[u]..\Psi[v]]$ has the prefix α and all the suffixes whose prefixes are α should be stored in $\mathtt{pos}[p..q]$ where $[p..q]$ is the suffix link of $[u..v]$, we get the following lemma.

Lemma 9 *The suffix link* $[p..q]$ *of an lcp-interval* $[u..v]$ *contains the* Ψ*-interval of* $[u..v]$, *i.e.*, $p \leq \Psi[u] < \Psi[v] \leq q$.

By Lemma 9, finding the suffix link of a k-interval $[u..v]$ is finding the $(k-1)$-interval $[i..j]$ containing the Ψ-interval of $[u..v]$. (Note that only one $(k-1)$-interval contains the Ψ-interval of $[u..v]$ because no two $(k-1)$-intervals overlap.)

Computing the suffix links consists of three steps. In these steps, we maintain two multiple lists $L[i]$ and $N[i]$, $1 \leq i < n$. We use $L[i]$ to store i-intervals and $N[i]$ to store the Ψ-intervals of i-intervals.

Step 1. *Compute* L: We perform an inorder traversal on the lcp-interval tree. If we encounter a k-interval $[u..v]$, we insert the k-interval $[u..v]$ into $L[k]$. Note that the k-intervals in $L[k]$ is sorted in the increasing order of the first indices of the k-intervals.

Step 2. *Compute* N: For each lcp-interval $[u..v]$ in $L[k]$, $2 \leq k \leq n-1$, we compute the Ψ-interval of $[u..v]$ and insert it into $N[k]$. We sort the Ψ-intervals in $N[k]$ in the increasing order of the first indices of the intervals for every $2 \leq k \leq n-1$. This can be done in $O(n)$ time by bucket sorting of size $n-1$ using backward pointers [3, 24].

Step 3. *Compute* `suflink`: For all k-intervals in $L[k]$, $2 \le k \le n-1$, we compute the suffix links as follows. For every Ψ-interval $[\Psi[u]..\Psi[v]]$ in $N[k]$, we find the $(k-1)$-interval $[p..q]$ in $L[k-1]$ such that $p \le \Psi[u] < \Psi[v] \le q$. Then, we set the suffix link of the k-interval $[u..v]$ as $(k-1)$-interval $[p..q]$ by Lemma 9. For example, consider the lcp-interval [2..3] in $L[2]$. The $[\Psi[2]..\Psi[3]]$ is [12..15] in $N[2]$. Since [12..15] is contained in [11..15] in $L[1]$, the suffix link of [2..3] is [11..15].
We show that step 3 can be performed in $O(n)$ time. Since the intervals in $N[k]$ and $L[k-1]$ are sorted in the increasing order of the first index of the intervals, we can find all the suffix links of k-intervals in $O(c(k)+c(k-1))$ time where $c(i)$ is the number of i-intervals. Overall, step 3 takes $O(n)$ time.

Since all steps are performed in $O(n)$ time, we get the following lemma.

Lemma 10 *We can compute all the suffix links in $O(n)$ time for integer alphabet without range minima query.*

We consider the space requirement of the suffix links. We need two additional lists L and N and additional space for backward pointers during construction. However, they are all temporary data structures and thus the resulting space requirement after construction is $2n$ integer entries.

6 Experimental Results

In this section, we measure and compare the running time of our algorithms with that of previous algorithms. We measured the running times on a PC equipped with a 3.0 Ghz Pentium IV processor and 4 GB DDR main memory. The PC is operated by Linux (Fedora core) and we used gcc/g++ for compilation. We made experiments on random strings and real world strings. The random strings differ in lengths (1 M, 10 M, 30 M, and 50 M) and in the sizes of alphabets (4, 20, 64, and 128). The real world strings are DNA sequences, protein sequences, and other real data strings, which are listed in the table in Fig. 11.

6.1 Construction Time

We measure the construction time of enhanced suffix arrays (ESAs) and linearized suffix trees (LSTs) in seconds (Figs. 12 and 13). The construction time for an ESA is the sum of construction times for a `pos` array, an `lcp` array, and a `cldtab` table. The construction time for an LST is the sum of construction times for a `pos` array, an `lcp` array, and a new `cldtab` table. To construct the `pos` array, we used Larsson and Sadakane's [28] (LS) algorithm, Manzini and Ferragina's [30] (MF) algorithm, and Schürmann and Stoye's [34] (SS) algorithm. The codes for LS, MF, and SS algorithms were obtained from the authors' web sites. They are http://www.larsson.dongmanet/research.html, http://www.mfn.unipmn.it/~manzini/lightweight/, and http://bibserv.techfak.uni-bielefeld.de/bpr/. Sometimes MF algorithm is the fastest and sometimes SS algorithm is the fastest. The fastest

Real strings	Text length	$\|\Sigma\|$	Real strings	Text length	$\|\Sigma\|$
DNA					
ecoli.nt	4, 662, 239	4	igseqnt.ftptemp	35, 653, 634	15
mito.nt	3, 164, 247	8	month.est_human	16, 895, 674	5
month.est_mouse	31, 031, 233	5	month.nt	118, 391, 923	15
vector.nt	3, 613, 945	6	yeast.nt	12, 155, 026	4
Protein					
drosoph.aa	7, 178, 856	20	ecoli.aa	1, 358, 990	21
igseqprot.ftptemp	4, 332, 606	23	month.aa	45, 921, 783	24
pataa	50, 066, 949	23	pdbaa	4, 026, 733	21
yeast.aa	2, 974, 038	20			
Real-world					
chr22.dna	34, 537, 472	5	etext00	105, 267, 200	146
gcc-3.0.tar	86, 630, 400	150	howto	39, 419, 904	197
jdk13c	69, 713, 920	113	linux-2.4.5.tar	116, 244, 480	256
rctail96	114, 704, 384	93	rfc	116, 408, 320	120
sprot34.dat	109, 608, 960	65	w3c2	102, 629, 376	256

Fig. 11 Input data

Length	`pos` (LS)	`pos` (MF)	`pos` (SS)	`lcp`	`cldtab`	new `cldtab`	%
	$\|\Sigma\| = 4$						
1 M	0.47	**0.19**	0.25	0.33	0.05	0.13	114
10 M	6.49	3.77	**3.62**	3.80	0.51	1.29	110
30 M	21.65	13.01	**12.33**	13.59	1.52	3.89	109
50 M	42.68	**19.22**	26.76	24.75	2.52	6.44	108
	$\|\Sigma\| = 20$						
1 M	0.45	**0.19**	0.57	0.34	0.04	0.11	112
10 M	6.01	**3.60**	6.34	3.94	0.38	1.15	110
30 M	23.08	**13.23**	19.98	14.01	1.08	3.36	108
50 M	42.74	**19.25**	26.76	25.69	1.86	5.87	109
	$\|\Sigma\| = 64$						
1 M	0.47	**0.17**	0.40	0.34	0.03	0.10	113
10 M	6.17	**2.84**	4.57	3.96	0.32	1.10	111
30 M	20.67	**10.16**	15.67	14.14	1.06	3.39	109
50 M	42.70	**19.18**	26.77	25.85	1.69	5.33	108
	$\|\Sigma\| = 128$						
1 M	0.43	**0.19**	0.56	0.34	0.03	0.11	114
10 M	7.04	**3.16**	6.24	3.97	0.31	1.00	109
30 M	22.79	**10.83**	19.70	14.12	0.78	2.95	108
50 M	42.70	**19.16**	26.78	25.88	1.28	5.02	108

Fig. 12 The experimental results for construction for random data

Strings	pos (LS)	pos (MF)	pos (SS)	lcp	cldtab	new cldtab	%
	DNA						
ecoli.nt	2.72	1.51	**1.49**	1.68	0.26	0.49	107
igseqnt.ftptemp	50.51	**20.81**	23.34	3.53	2.06	3.53	106
mito.nt	2.12	**0.89**	1.71	1.08	0.18	0.32	107
month.est_human	14.39	7.17	**6.91**	5.02	0.98	1.68	105
month.est_mouse	29.15	**14.12**	14.91	12.98	1.83	3.17	105
month.nt	149.66	**65.99**	84.53	60.57	6.86	12.13	104
vector.nt	5.18	3.13	**1.60**	1.03	0.20	0.34	105
yeast.nt	8.31	4.64	**4.27**	4.67	0.67	1.23	106
	Protein						
drosoph.aa	4.55	**2.57**	4.23	2.62	0.33	0.73	107
ecoli.aa	0.65	**0.29**	0.80	0.47	0.06	0.14	110
igseqprot.ftptemp	3.35	**1.41**	1.95	1.24	0.25	0.47	108
month.aa	40.17	**22.71**	26.76	22.64	2.21	4.72	105
pataa	51.57	**27.52**	27.63	21.84	2.68	5.25	105
pdbaa	3.43	**1.44**	2.54	1.36	0.22	0.43	107
yeast.aa	1.62	**0.85**	1.77	1.07	0.14	0.30	108
	Real-world						
chr22.dna	25.78	**14.59**	16.46	14.75	1.92	3.53	105
etext00	145.35	**71.82**	92.09	49.88	5.62	12.53	105
gcc-3.0.tar	96.06	69.23	**57.19**	26.81	4.55	9.52	106
howto	38.90	**17.27**	22.40	14.13	2.10	4.62	108
jdk13c	103.54	65.32	**62.79**	21.36	3.67	6.87	104
linux-2.4.5.tar	127.24	**53.22**	65.75	37.64	6.23	13.36	107
rctail96	180.22	**123.90**	127.17	46.53	6.26	12.04	103
rfc	152.43	**61.37**	91.63	44.52	6.50	12.20	105
sprot34.dat	142.59	**67.16**	97.15	46.06	6.03	11.46	105
w3c2	188.29	104.37	**78.08**	31.95	4.59	10.39	105

Fig. 13 The experimental results for construction for real-world data

construction times are represented by boldface numbers and they are used to compute the construction times of LSTs and ESAs. To construct the `lcp` array, we used Kasai et al.'s [22] algorithm.

We computed the percentage of the construction times of LSTs over those of ESAs. The construction time of an LST is at most 14% slower than that of an ESA on random strings and at most 8% slower on real-world data. Actually, the construction time of a new `cldtab` table is about twice of the construction time of a `cldtab` table. However, the construction time for a (new) child table is only a small part of the construction times of an LST and an ESA, and thus an LST can be constructed just a little slower than an ESA.

6.2 Pattern Searching

We compared the pattern search time in our index data structure with that in the suffix tree (ST), that in the suffix array using the Manber and Myers' algorithm (MM), and that in the enhanced suffix array (ESA). Figure 14 shows the experimental results. The code for ST was obtained from http://www.tigr.org/software/mummer/ and the code for MM was obtained from Myers. However, when $|\Sigma| \geq 128$, the code from Myers was not working and thus we slightly modified the code to generate data for such cases. The data generated by the modified code are denoted by asterisks. The code for ESA and LST was implemented by us. We measured the running time in seconds for performing existential queries for 1 million patterns of lengths between 300 and 400 drawn from the text. Note that existential queries and enumeration queries take almost the same time on suffix arrays, ESAs, and LSTs because we find an interval of the `pos` array as the results of enumeration queries. However, enumeration queries take more time than the existential queries on suffix trees.

It shows that the pattern search in an LST is faster than in an ESA when the size of alphabet is larger than 20 regardless of the length of a string. Moreover, the ratio of the pattern search time in an ESA to that in an LST becomes larger as the size of alphabet becomes large. These experimental results are consistent with the time complexity analysis of the pattern search. The pattern search in an LST is quite fast so it can be comparable to that in a suffix array using Manber and Myers' pattern search.

6.3 Computing Suffix Links

We measure the running time of our algorithms for computing suffix links and compare them with two previous algorithms. The previous one running in $O(n \log n)$ time is denoted by 'BS' and the other running in $O(n)$ time with the help of the range minima query is denoted by 'RMQ' in Figs. 15 and 16. Our algorithm for constant alphabets is denoted by 'Our1' and the other for integer alphabets is denoted by 'Our2.' We compare construction times of the BS and RMQ algorithms with those of Our1 and Our2 algorithms for random strings and real world data. Figure 15 shows the results on random strings and Fig. 16 shows the results on real world data. When the size of alphabet is 4, the Our1 algorithm is about 2 times faster than the BS algorithm, and when the size of alphabet is 20 or more, the Our2 algorithm is about 3-4 times faster than the RMQ algorithm.

6.4 Application

Among various string problems, we selected matching statistics as an application of the linearized suffix tree because it is an essential subproblem of the sequence alignment problem, which are widely used in bioinformatics. *Matching statistics* $MS_{A,B}(i)$ for two strings A and B, which is also defined by Gusfield [18], is the length of the longest prefix of the ith suffix of A that occurs in B for $1 \leq i \leq n$. To compute matching statistics $MS_{A,B}(i)$ for all $1 \leq i \leq n$, we first generate the en-

Length	ST	MM	ESA	LST	ST	MM	ESA	LST
	$\|\Sigma\| = 4$				$\|\Sigma\| = 20$			
1 M	4.50	4.59	3.88	4.41	5.43	4.52	5.84	4.42
10 M	6.53	6.49	5.89	6.54	8.72	6.37	10.05	6.56
30 M	7.63	7.59	7.41	8.19	9.95	7.44	12.33	8.25
50 M	8.20	8.16	8.10	9.01	10.91	7.99	13.96	9.05
	$\|\Sigma\| = 64$				$\|\Sigma\| = 128$			
1 M	9.19	4.49	10.83	4.52	10.32	4.36(*)	18.78	4.55
10 M	13.61	6.38	19.31	6.78	22.79	6.38(*)	27.09	6.80
30 M	19.57	7.55	25.13	8.45	24.34	7.47(*)	35.26	8.58
50 M	21.04	8.11	28.47	9.35	26.16	7.99(*)	41.88	9.40

Strings	ST	MM	ESA	LST
	DNA			
ecoli.nt	6.02	5.92	5.22	6.01
igseqnt.ftptemp	18.29	8.26	11.12	14.03
mito.nt	6.06	5.63	4.96	6.07
month.est_human	9.02	7.26	7.31	8.88
month.est_mouse	10.76	7.88	8.35	10.32
month.nt	12.36	10.63(*)	10.96	13.12
vector.nt	7.57	7.40	5.61	6.72
yeast.nt	7.28	6.75	6.25	7.14
	Protein			
drosoph.aa	8.54	6.14	8.99	6.43
ecoli.aa	6.03	4.74	6.27	4.92
igseqprot.ftptemp	21.84	6.00	13.00	11.09
month.aa	11.85	7.96	14.02	9.70
pataa	12.91	8.90(*)	14.42	10.09
pdbaa	8.93	5.84	8.21	6.55
yeast.aa	7.29	5.98(*)	7.58	5.61
	Real-world			
chr22.dna	10.69	8.83(*)	8.82	9.84
etext00	37.33	9.84(*)	34.49	16.30
howto	37.11	8.70(*)	33.84	14.39
jdk13c	31.64	7.78(*)	28.43	16.41
rctail96	32.61	13.47(*)	30.88	16.45
sprot34.dat	26.94	9.86	26.32	16.00
w3c2	46.11	6.40(*)	42.16	17.90

Fig. 14 The experimental results for pattern search

hanced suffix array or the linearized suffix tree of A and then search for the suffixes of B. (Computing matching statistics on enhanced suffix arrays was already given in [1, 2].) With the help of suffix links, one can do this in linear time. We measure the

Length	$\|\Sigma\| = 2$		$\|\Sigma\| = 4$		$\|\Sigma\| = 8$		$\|\Sigma\| = 16$	
	BS	Our1	BS	Our1	BS	Our1	BS	Our1
1 M	0.48	0.17	0.35	0.18	0.29	0.20	0.25	0.23
10 M	5.53	1.73	4.08	1.89	3.36	2.07	2.85	2.34
30 M	18.31	5.62	13.40	6.18	11.12	6.85	9.94	7.91
Length	$\|\Sigma\| = 20$		$\|\Sigma\| = 32$		$\|\Sigma\| = 64$		$\|\Sigma\| = 128$	
	RMQ	Our2	RMQ	Our2	RMQ	Our2	RMQ	Our2
1 M	0.91	0.23	0.91	0.25	0.94	0.25	0.84	0.22
10 M	10.07	2.76	9.13	2.46	9.11	2.61	9.79	2.62
30 M	30.83	7.96	30.95	8.38	33.29	9.19	27.17	6.65

Fig. 15 The experimental results for computing suffix links for random strings. We generated different kinds of random strings which differ in lengths (1 M, 10 M, and 30 M). These strings are drawn from the alphabet size of 2, 4, 8 and 16 for the algorithms for constant alphabets and 20, 32, 64, and 128 for the algorithms for integer alphabets

Strings	$\|\Sigma\|$	BS	Our1	RMQ	Our2
ecoli.nt	4	1.93	0.91	5.30	1.41
igseqnt.ftptemp	15	26.29	7.75	58.71	19.02
mito.nt	8	1.40	0.61	3.79	1.05
month.est_human	5	11.18	3.71	25.98	8.13
month.est_mouse	5	19.11	6.96	45.88	13.57
vector.nt	6	2.35	0.57	5.99	1.75
yeast.nt	4	5.33	2.36	14.16	3.81
drosoph.aa	20	2.40	1.95	7.66	2.22
ecoli.aa	21	0.37	0.33	1.32	0.35
igseqprot.ftptemp	23	2.17	0.85	5.63	1.87
month.gss	9	16.09	6.30	40.24	11.17
pdbaa	21	1.97	1.17	5.19	1.65
yeast.aa	20	0.93	0.76	3.06	0.85

Fig. 16 The experimental results for computing suffix links for real-world data

running time to compute matching statistics (Fig. 17). When the size of alphabet is 20 or more, using the linearized suffix tree is faster than using the enhanced suffix array.

7 Concluding Remarks

We presented an index data structure with the capabilities of the suffix tree and the suffix array even when the size of the alphabet is not small by improving the enhanced suffix array. We improved the enhanced suffix array such that our index data structure supports the pattern search in $O(m \log |\Sigma|)$ time without sacrificing the space. In a different point of view, it can be considered a practical one facilitating the capabili-

Length	$\|\Sigma\| = 4$		$\|\Sigma\| = 20$		$\|\Sigma\| = 64$		$\|\Sigma\| = 128$	
	ESA	LST	ESA	LST	ESA	LST	ESA	LST
1 M	0.82	1.06	1.88	1.71	4.04	2.24	13.10	3.10
10 M	11.99	13.74	24.13	21.54	79.32	35.78	79.00	32.54
30 M	31.83	36.74	84.28	77.75	222.21	110.52	323.26	125.27
50 M	73.85	85.43	159.25	144.58	324.14	172.73	734.67	235.29

Strings	$\|\Sigma\|$	ESA	LST
DNA			
ecoli.nt(E. Coli K12) vs. E. Coli O157:H7	4	6.55	7.20
Human chromosome 22 vs. Human chromosome 21	5	47.90	54.97
month.est_human vs. month.est_mouse	5	40.84	47.12
Protein			
pdbaa vs. pataa	23	117.01	101.75
yeast.aa vs. pataa	23	107.30	94.06
yeast.aa vs. pdbaa	21	8.90	7.96
Real-world			
gcc-3.0.tar vs. etext99	150	408.19	264.51
jdk13c vs. rfc	120	252.98	193.06

Fig. 17 The experimental results for computing matching statistics for random strings and real-world data

ties of suffix trees when the size of the alphabet is not small because the suffix tree supporting $O(m \log |\Sigma|)$-time pattern search is not easy to implement and thus it is rarely used in practice. We also presented two efficient algorithms for computing suffix links in the enhanced suffix array and our data structure. Our algorithms are the first algorithms running in $O(n)$ time without range minima query.

There's another line of research on developing an index data structure occupying $O(n)$-bit with the help of succinct representation. Munro et al. [32] suggested space efficient suffix trees, Grossi and Vitter [17] suggested the compressed suffix array, Ferragina and Manzini [13] suggested opportunistic data structures under the name of FM-index, and Sadakane [33] introduced the compressed suffix tree with full functionality. They tried to reduce the space consumption by sacrificing the construction and search time. Recently, Hon et al. [19] developed a linear time construction using $O(n)$-bit extra space. It will be interesting to develop a succinct representation of the linearized suffix tree and compare it with the previous index data structures occupying $O(n)$-bit.

Acknowledgements The preliminary version of this paper was presented in SPIRE 2004. This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement) (IITA-2006-(C1090-0602-0011)), and also supported by the Program for the Training of Graduate Students in Regional Innovation which was conducted by the Ministry of Commerce, Industry and Energy of the Korean Government. This work, corresponded by Heejin Park, was supported by the research fund of Hanyang University (HY-2004-S).

We thank Kunsoo Park for valuable suggestions including the simpler definition of the child table than that in the preliminary version. Jeong Eun Jeon helped us set the experimental environment. We thank the reviewers for their valuable comments that helped improve the presentation of this paper.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**, 53–86 (2004)
2. Abouelhoda, M., Ohlebusch, E., Kurtz, S.: Optimal exact string matching based on suffix arrays. In: Symposium on String Processing and Information Retrieval, pp. 31–43 (2002)
3. Aho, A., Hopcroft, J., Ullman, J.: Data Structures and Algorithms. Addison-Wesley, Reading (1983)
4. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Symposium on Combinatorial Pattern Matching, pp. 55–69 (2003)
5. Chen, M.T., Seiferas, J.: Efficient and elegant subword tree construction. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO ASI Series F: Computer and System Sciences, pp. 97–107. Springer, Berlin (1985)
6. Clark, D., Munro, I.: Efficient suffix trees on secondary storage. In: SODA, pp. 383–391 (1996)
7. Colussi, L., Col, A.: A time and space efficient data structure for string searching on large texts. IPL **58**(5), 217–222 (1996)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
9. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica **32**, 1–35 (2002)
10. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. In: Workshop on Algorithm Engineering and Experiments (2005)
11. Farach, M.: Optimal suffix tree construction with large alphabets. In: IEEE Symposium on Foundations of Computer Science, pp. 137–143 (1997)
12. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. Assoc. Comput. Mach. **47**, 987–1011 (2000)
13. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2001)
14. Giegerich, R., Kurtz, S.: A comparison of imperative and purely functional suffix tree construction. Sci. Comput. Program. **25**, 187–218 (1995)
15. Giegerich, R., Kurtz, S.: From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. Algorithmica **19**, 331–353 (1997)
16. Gonnet, G., Baeza-Yates, R., Snider, T.: New indices for text: Pat trees and pat arrays. In: Frakes, W.B., Baeza-Yates, R.A. (eds.) Information Retrieval: Data Structures & Algorithms, pp. 66–82. Prentice-Hall, Englewood Cliffs (1992)
17. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: ACM Symposium on Theory of Computing, pp. 397–406 (2000)
18. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, Cambridge (1997)
19. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. In: IEEE Symposium on Foundations of Computer Science, pp. 251–260 (2003)
20. Kärkkäinen, J.: Suffix cactus: a cross between suffix tree and suffix array. In: Symposium on Combinatorial Pattern Matching, pp. 191–204 (1995)
21. Kärkkäinen, J., Sanders, P.: Simpler linear work suffix array construction. In: International Colloqium on Automata Languages and Programming, pp. 943–955 (2003)
22. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Symposium on Combinatorial Pattern Matching, pp. 181–192 (2001)
23. Kim, D.K., Jo, J., Park, H.: A fast algorithm for constructing suffix arrays for fixed-size alphabets. In: Workshop on Efficient and Experimental Algorithms, pp. 301–314 (2004)
24. Kim, D.K., Park, K.: Linear-time construction of two-dimensional suffix trees. In: International Colloqium on Automata Languages and Programming, pp. 463–472 (1999)
25. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Symposium on Combinatorial Pattern Matching, pp. 186–199 (2003)

26. Ko, P., Aluru, S.: Space-efficient linear time construction of suffix arrays. In: Symposium on Combinatorial Pattern Matching, pp. 200–210 (2003)
27. Kurtz, S.: Reducing the space requirement of suffix trees. Softw. Pract. Experience **29**, 1149–1171 (1999)
28. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Technical report No. LU-CS-TR:99-214, Department of Computer Science, Lund University, Sweden (1999)
29. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**, 935–938 (1993)
30. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**, 33–50 (2004)
31. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. Assoc. Comput. Mach. **23**, 262–272 (1976)
32. Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. J. Algorithms **39**, 205–222 (2001)
33. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. (2007, in press)
34. Schürmann, K., Stoye, J.: An incomplex algorithm for fast suffix array construction. Softw. Pract. Exp. **37**(3), 309–329 (2007)
35. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**, 249–260 (1995)
36. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)