Efficient Substring Traversal with Suffix Arrays

Kasai, Toru
Department of Informatics, Kyushu University

Arimura, Hiroki
Department of Informatics, Kyushu University

Arikawa, Setsuo
Department of Informatics, Kyushu University

出版情報：DOI Technical Report. 185, 2001-02. Department of Informatics, Kyushu University
バージョン：
権利関係：

Efficient Substring Traversal with Suffix Arrays

Toru Kasai[†] Hiroki Arimura[†*] Setsuo Arikawa[†]

[†]Department of Informatics, Kyushu University
Fukuoka 812-8581, Japan
[*]PRESTO, Japan Science Corporation
TEL: +81-92-642-2688, FAX: +81-92-642-2698
E-mail:arim@i.kyushu-u.ac.jp

Abstract

The *substring traversal problem* is the problem of enumerating all branching substrings appearing in a given text. Although this problem is easily solvable with the suffix tree of McCreight (1976), a space efficient and practically fast solution is important. We devise a simple and efficient algorithm that simulates the traversal of the suffix tree for a given text with the *suffix array* of Manber and Meyers (1993) and Gonnet, Baeza-Yates, Snider (1992). The algorithm runs in $O(n)$ time and $\frac{5n}{B}$ bulk I/O with the suffix array and an additional structure called the *height array*, while the naive algorithm using binary search on the suffix array requires $O(n^2)$ time in the worst case. The space requirement $7N$ bytes of our algorithm is smaller than $15N$ bytes of the traversal algorithm with the suffix tree. A linear time algorthm for computing the height array from the suffix and the height arrays is also presented. Computer experiments on real datasets showed that our traversal algorithm with the suffix array is an order of magnitude faster than the naive simulation method and comparable to the traversal algorithm with the suffix tree.

1 Introduction

In this paper, we investigate the *substring traversal problem*, which is the problem of enumerating all branching substrings appearing in a given text. Although it is well-known that the substring traversal problem is easily solvable by the traversal of the *suffix tree* of McCreight (1976), recent large scale applications in bioinformatics and data mining require a more practical and scalable algorithm for the problem [13, 2].

The *suffix array* is a space efficient data structure proposed by [10], and independently by [5], that still allows efficient and advanced pattern matching as the suffix tree. It is widely believed that most problems solvable with the suffix tree are also solvable with binary search on the suffix array by a slowdown of $\log n$ factor. However, we can easily observe that a naive algorithm for the substring traversal problem requires $O(n^2)$ time and $O(n \log n)$ time without and with the constant-time lcp computation [11].

We present a simple and efficient algorithm that simulates the traversal of the suffix tree for a given text in $O(n)$ time with the suffix array combined with an additional information called the *height array* [10]. This algorithm traverses a *virtual* suffix tree using the left to right scan of the height array by using a stack, and thus has a good I/O complexity with $\frac{5n}{B}$ blocks. Furthermore, we show that the algorithm can be modified to solve a class of problems based on the occurrence count of each branching substring. We also present a linear time algorithm for computing the height array Hgt from a text T and its suffix tree SA, which is faster than the naive $O(n^2)$ time algorithm that requires in the worst case. Finally, we run computer experiments on a real dataset and showed that the traversal with the suffix array is an order of magnitude faster than the simulated suffix tree traverse with array approach.

Our approach have the following advantages over the *suffix tree traversal* approach and the *simulated suffix tree traversal on array* approach: (i) *Fast*: Our traversal algorithm runs in $O(n)$ time, while the simulated tree traversal on array algorithm runs in $O(n^2)$ time without and $O(n \log n)$ time with the constant-time lcp information. (ii) *Space efficient*: Our data structure requires only $7n$ bytes bytes, while the suffix tree requires at least $15n$ bytes. (iii) *I/O efficient*: In the external memory environment, most external I/O made by our traversal algorithm is sequential and the algorithm makes a small number of random accesses to the main memory. This maximally utilizes the characteristics of the external I/O devices when traversing a huge suffix array on external memory. Furthermore, the algorithm does not need indirect accesses to the text via suffix pointers.

The rest of the paper is organized as follows. In Section 2, we prepare basic notions and definitions. Then in Section 3, we present linear time algorithms for the substring traversal problem and the related problems. In Section 4, we show that the height array is linear time computable from a text and the suffix array. In Section 5, we show experimental results and in Section 6, we conclude the results.

2 Preliminaries

2.1 Strings

Let Σ be an alphabet of letters. We denote by Σ^* and $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ the set of all strings and the set of all nonempty strings over Σ. We assume a total order $<$ on letters of Σ such that $a < \$$ for any $a \in \Sigma$. Let $S = a_1 \dots a_n$ be a string over Σ of length $n \geq 0$. We define the length of S by $|S| = n$. For $1 \leq i, j \leq n$, $i \leq j$, we denote by $S[i] = a_i$ the i-th letter of S and by $S[i..j] = a_i \cdots a_j$ the *substring* starting at i and ending at j. We define $S[i..j] = \varepsilon$ if $i > j$. For any $1 \leq i \leq j$, the i-th *prefix* and the i-th *suffix* of S are $S[1..i]$ and $S[i..n]$, respectively. For strings $S, T \in \Sigma^*$, we denote by $LCP(S,T)$ the *longest common prefix* (*lcp*, for short) of S and T, and by $lcp(S,T) = |LCP(S,T)|$ the *lcp-length* of S and T.

A *text* of length n is a string $T = a_1 a_2 \cdots a_{n-1} \$ \in \Sigma^* \times \{\$\}$, where $a_i \in \Sigma$ $(i = 1, 2, \dots, n-1)$ and $\$$ is a special end marker such that $\$ \notin \Sigma$. We suppose that $n > 1$. For any $1 \leq i \leq n$, we denote the i-th suffix of T by $T_i = T[i..n]$ if no ambiguity arises. A string S *occurs in* T *at position* $1 \leq i \leq n$ if there exist an integer such that $S = T[i..i + |S| - 1]$, where i is called the *occurrence* of S in T. We denote by $Occ(S,T)$ the set of all occurrences of S in T. We denote by $suffix(T)$ and $substr(T)$ the sets of all suffixes and all substrings of text T, respectively. Let $P \subseteq \{1, \dots, n\}$ be a set of positions in T. Then, we say that a string P *has the occurrence in* P *w.r.t.* T if there exists some position $p \in P$ that is an occurrence of P in T.

2.2 Ordered trees and their traversals

In this paper, we consider ordered and compacted trees. An ordered tree OT is *compacted* if every internal node of OT has at least two children. A suffix tree is ordered and compacted.

Let OT be an ordered and compacted tree. Let $root(OT)$ denote the root of OT. If v is a node in OT then we write $v \in OT$. Suppose that OT has $n \geq 0$ leaves $\ell_1, \dots, \ell_n$ ordered from left to right. For $1 \leq i \leq n$, we say ℓ_i is the i-th leaf of OT. An *upward path* (path, for short) is a sequence of nodes in OT $\pi = (v_0, v_1, \dots, v_m)$ $(m \geq 0)$ such that v_i is the parent of v_{i-1} for every $1 \leq i \leq m$, where we the length of π is $|\pi| = m$. An edge e *appears* in π if there is some $1 \leq i \leq m$ such that $e = (v_{i-1}, v_i)$. A *branch* is any upward path that starts with a leaf.

For each leaf ℓ, we denote by $\pi(\ell)$ the branch that start with ℓ and ends with the root. Let $v \in OT$ be a node. We denote by OT_v the subtree of OT whose root is v, and define the *depth* of v to be the length $depth(v)$ of the path from v to $root(OT)$.

A node w is an *ancestor* of a node v, denoted by $w \preceq v$, if there exists some path from v to u. If $w \preceq v$ and $w \neq v$ then we denote $w \prec v$ For nodes u and v, we denote by $lca(u,v)$ the *lowest common ancestor* of u and v.

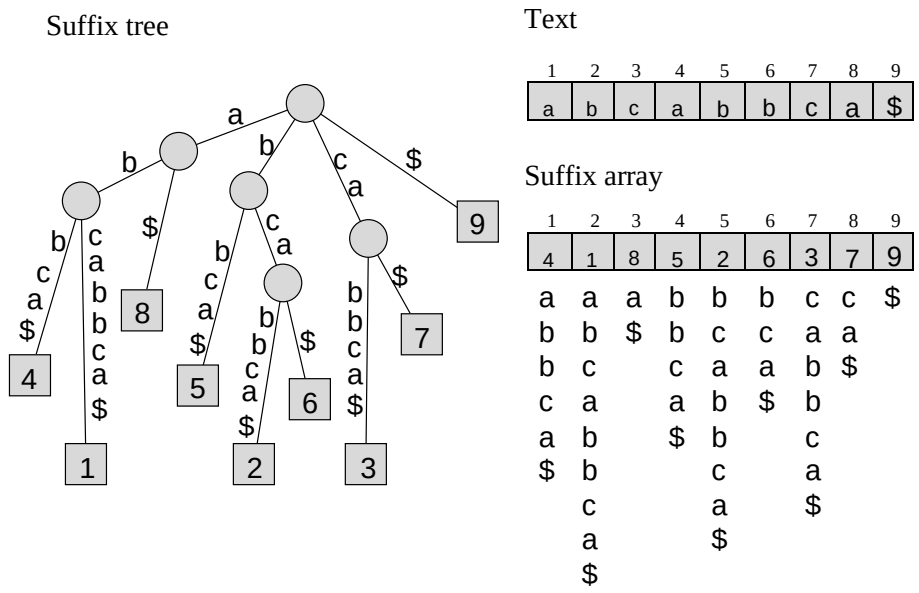

Figure 1: The suffix tree and the suffix array

For lists of nodes π_1, π_2, we denote by $\pi_1 \cdot \pi_2$ the concatenation of π_1 and π_2. A *traversal* of an ordered tree OT is any list $\pi = (v_1, \ldots, v_m)$ consisting of the nodes of OT such that every node of OT appears exactly once in π, and is *bottom-up* if for every $1 \leq i \leq m$, all the descendants of v_i appear in $v_1, \ldots, v_{i-1}$. The post-order traversal defined below is a special class of bottom-up traversals.

Definition. The *post-order traversal* of an ordered tree OT with the root *root*, denoted by $POT(OT)$, is a list of nodes of OT defined recursively as follows. (i) If OT consists of a single node v then $POT(OT) = (v)$. (ii) Let $v_1, \ldots, v_m$ be the children of *root* in the order from left to right and $OT_1, \ldots, OT_m$ be the corresponding subtrees of OT. Then, $POT(OT) = POT(OT_1) \cdots POT(OT_m)$.

2.3 Suffix trees and suffix arrays

In what follows, let $T \in \Sigma^* \times \{\$\}$ be a text of length n that ends with the end marker $\$ \notin \Sigma$. The *suffix tree* for text T, denoted by $ST(T)$, is a compacted trie for all nonempty suffixes of T (Fig. 1). Formally, the suffix tree for T is an ordered tree ST defined as follows. (i) Each edge has a nonempty substring $\alpha \in \Sigma^*$ as its label. A label is encoded by a pair of its starting position in T and its length. (ii) Each internal node has at least two out-going edges. Furthermore, the labels of distinct edges from the same node have to start with distinct letters. Out-going edges are ordered from left to right in the order of the first letter of their labels. (iii) Each node v represents the string, denoted by $str(v)$, obtained by concatenating the labels on the path from the root to v. (iv) ST has exactly n leaves that represent the nonempty suffixes of T. For every $1 \leq i \leq n$, the i-th leaf represents the i-th suffix and labeled by i. For a node v of ST, we define the string-depth of v by $strdepth(v) \stackrel{\text{def}}{=} |str(v)|$.

In Fig. 1, we show the suffix tree for string $T = \texttt{abcabbca\$}$.

The suffix tree $ST(T)$ has n leaves and at most $n-1$ internal nodes, and can be stored in $O(n)$ space. McCreight [9] gives an elegant algorithm for building $SA(T)$ in time $O(n)$ for constant size alphabet and in time $O(n \log n)$ for large alphabet.

For strings S, S', we define $S \leq_{\text{lex}} S'$ iff S is lexicographically smaller than or equal to S'. The *suffix array* for text $T[1..n]$ is an array $SA = [p_1, p_2, \ldots, p_n]$ consisting of the starting positions of the lexicographically ordered suffixes $T_{SA[p_1]} \leq_{\text{lex}} T_{SA[p_2]} \leq_{\text{lex}} \cdots \leq_{\text{lex}} T_{SA[p_n]}$ of T. By definition, for any $1 \leq i \leq n$, $SA[i]$ is the starting position of the suffix of rank i in T. The suffix array is equivalent to the array obtained by storing the leaves of the suffix tree ST for T from left to right. For example, if $T = \texttt{abcabbca\$}$ then $SA = [\texttt{4,1,8,5,2,6,3,7,9}]$ is the suffix array for T (Fig. 1).

For the suffix array $SA[1..n]$, we define the *inverse suffix array* $SA^{-1}[1..n]$ that represents the inverse mapping of $SA : \{1,\ldots,n\} \to \{1,\ldots,n\}$ by $SA^{-1}[SA[i]] = i$ for every $i = 1,\ldots,n$. The *height array* $Hgt[1..n]$ is the array defined by $Hgt[i] = lcp(T_{SA[i-1]}, T_{SA[i]})$ for every rank $1 \leq i \leq n$, namely, the array consisting of the lcp-length of lexicographically adjacent suffixes. We define $Hgt[i] = -1$ if $i = 1$ or $i = n+1$.

The suffix array SA and the height array Hgt for a text of length n occupy $4n$ bytes and $2n$, respectively, to store. Although we can build the suffix array from a given text in linear time by a linear time suffix tree construction algorithm, this construction requires as much space as for building a suffix tree. A simple and space efficient way to build a suffix array from a given text T is to use any sorting algorithm for strings. Manber-Myers' algorithm [10] and Sadakane's algorithm [8] are the fastest algorithms with $O(n \log n)$ time complexity even with a large alphabet. The ternary quick sort [3] is a practical choice for building a suffix array.

2.4 Substring traversal problem

First, we define a canonical class of substrings, called branching substrings. Let T be a text of length n and $P \subseteq \{1,\ldots,n\}$ be a set of positions. Define a binary relation $\subseteq_T^P$ between substrings as for every substrings S, S' of T, the $S \subseteq_T^P S'$ holds iff for every position $p \in P$, if S appears at p in T then S' also appears at p in T. We define the euivalence relation $\equiv^T$ by $S \equiv_T^P S'$ iff both $S \subseteq_T^P S'$ and $S' \subseteq_T^P S$ hold. If P is the whole set of positions, then we will omit the superscript P.

Definition. A substring S of T is *branching w.r.t.* P if there exists a pair of distinct suffixes $\alpha_1, \alpha_2 \in suffix_T(P)$ such that $S = LCP(\alpha_1, \alpha_2)$.

The following lemmas are well-known and easily proved using the suffix tree [9].

Lemma 1 *Let $SA[1..n]$ be the suffix array for T. A substring S of T is branching iff there exists a pair of ranks $1 \leq i, j \leq n$ such that $S = LCP(T_{SA[i]}, T_{SA[j]})$.*

Lemma 2 *A substring S of T is branching iff S is the unique longest member of the set $\{ S' \in substr(T) \mid S \equiv^T S' \}$, namely, the equivalence class for S w.r.t. $\equiv_T$.*

For a finite set of objects $\mathcal{O}$, a *traversal of* $\mathcal{O}$ is any sequence of elements of $\mathcal{O}$ in which every element of $\mathcal{O}$ appears exactly once. In our problem, $\mathcal{O}$ is the set of all branching substrings. We encode each substring P by a representation called an *info* that is a pair (i, ℓ) of the rank i and the length ℓ of P such that $T[p..p+\ell-1]$, where $p = SA[i]$. Now, we state our problem as follows.

Definition. The *substring traversal problem* is the problem of printing a traversal L of all branching substrings of a given text $T \in \Sigma^* \times \{\$\}$, where each branching substring is encoded by its info.

3 Substring Traversal with Suffix Arrays

In this section, we present an efficient algorithm that solves the substring traversal problem in linear time by simulating the traversal of a suffix tree with arrays SA and Hgt. First, we present a linear time algorithm for building the bottom-up traversal of an ordered tree by using three restricted class of operations: *scanning of leaves*, the *lowest common ancestors*, and the *ancestor relation decision*. Then, by modifying this algorithm, we present a linear time algorithm for the substring traversal with SA and Hgt. As an application, we show that the subword statistics problem can be solvable in linear time. Some proofs are omitted in this section. For more details, please consult [7].

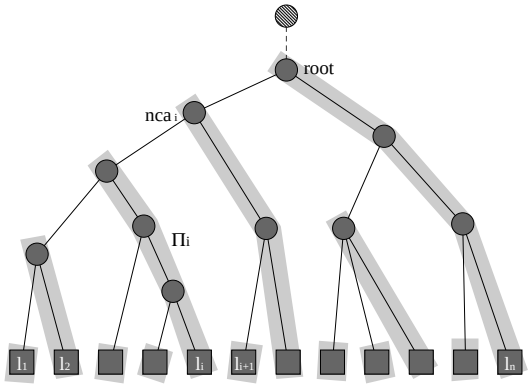

Figure 2: The rightmost branch decomposition of an ordered and compacted tree. Each shadowed line going upward from a leaf to the root indicates a rightmost branch. $\top$ denotes the special top node.

Algorithm Bottom_up_Traversal;
input: An ordered and compacted tree OT with $n \geq 0$ leaves $\ell_1, \ldots, \ell_n$.
output: The post-order traversal of OT.
method:
1 Let $S = \{\top\}$ be the empty stack with $\top$ as the bottom marker;
2 **foreach** $i = 1, \ldots, n+1$ **do begin**
i-th stage:
3 $lca_i := lca(\ell_{i-1}, \ell_i)$;
4 **while** $(top(S) \succ lca_i)$ **do**
5 $v := Pop(S)$ and report v;
6 **if** $(top(S) \prec lca_i)$ **then**
7 $Push(lca_i, S)$;
8 $Push(\ell_i, S)$; /* Set $S_i = S$ */
9 **end** /* for-loop */

Figure 3: The algorithm for traversing an ordered and compacted tree, where $\top$ is the special *top* node and $v \prec w$ means that node v is an ancestor of node w.

3.1 Bottom-up traversal of ordered compacted trees

Let OT be an ordered and compacted tree with $n \geq 0$ leaves $\ell_1, \ldots, \ell_n$. For convention, we assume a special *top* node$\top$ such that $\top \prec v$ for every $v \in OT$. We also assume special leaves ℓ_0 and ℓ_{n+1} such that $lca(\ell_0, \ell_1) = lca(\ell_n, \ell_{n+1}) = \top$. These assumptions means that $\top$ is the parent of the root of OT and the root is neither the first child nor the last child of $\top$.

In Fig. 3, we show the algorithm Bottom_up_Traversal for computing the post-order traversal using a stack and the following three operations: left-to-right scanning of leaves; the computation of the lowest common ancestors; decision of the ancestor relation between nodes. We write the contents of a stack from the top to the bottom as $S = (w_0, w_1, \ldots, w_m = \top)$, where w_0 is the top and w_m is the bottom of S. In the algorithm, $\top$ is used as a sentinel.

Before showing the correctness of the algorithm, we start with preparing some notations. For $1 \leq i \leq n+1$, we refer to the i-th execution of the for-loop from line 2 to line 9 of Fig. 3 as the i-th *stage*. For every stage $1 \leq i \leq n+1$, let ℓ_i be the i-th leaf, $\pi_i \stackrel{\text{def}}{=} \pi(\ell_i)$ be the branch from ℓ_i to $\top$ (not the root), and let $lca_i \stackrel{\text{def}}{=} lca(\ell_{i-1}, \ell_i)$ be the lca of the current and the previous leaves. We call ℓ_i, π_i and lca_i the *current leaf*, the *current branch*, and the *current lca*. Recall that any path is written in the upward direction. A branch $\pi = (v_0, v_1, \ldots, v_m)$ is a *rightmost branch* if it consists of only right most edges, that is, v_{i-1} is the rightmost child of v_i for every $1 \leq i \leq m$.

Definition. For every $1 \leq i \leq n$, the *i-th rightmost branch* is the longest rightmost branch starting with ℓ_i and denoted by $\Pi(\ell_i)$.

We identify an ordered tree OT with the set of its nodes and a path Π with the set of all nodes appearing in π. A *decomposion* of OT is any set $\{\Pi_1, \ldots, \Pi_m\}$ of paths in OT such that $\cup_{1 \leq i \leq m} \Pi_i = OT$ and $\Pi_i \cap \Pi_j = \emptyset$ for every $i \neq j$.

Lemma 3 *For any ordered and compacted tree OT with n leaves, the set $\{\Pi(\ell_1), \ldots, \Pi(\ell_n)\}$ is a decomposition of OT.*

By Lemma 3, we call the decomposition $\{\Pi(\ell_1), \ldots, \Pi(\ell_n)\}$ the *rightmost branch decomposition* of OT. The next lemma characterizes the post-order traversal of an ordered and compacted tree with its rightmost branch decomposition.

Lemma 4 *Let OT be an ordered and compacted tree of n leaves $\ell_1, \ldots, \ell_n$. Then, $POT(OT) = \Pi(\ell_1) \cdots \Pi(\ell_n)$ holds, that is, the post-order traversal of OT is equivalent to the concatenation of the rightmost branches of OT from left to right.*

Proof: It is easy to show the lemma by induction on the number $n \geq 1$ of the leaves of OT. ■

Let S_i be the contents of the stack S after executing the steps at line 3 to line 8 in the i-th stage of the algorithm Bottom_up_Traversal of Fig. 3. A node v is the *non-leftmost child* of node w if v is a child of w that is not the leftmost one. By the assumption that OT is compacted, every node has at least one non-leftmost child. The following lemma is central to the correctness of the algorithm. Below, we refer to the condition on v_j "$j > 0$ and v_{j-1} is the leftmost child of v_j" as the *exclusion condition* on v_j.

Lemma 5 *Let $1 \leq i \leq n+1$ and $\pi_i = (v_0 = \ell_i, v_1, \ldots, v_m = \top)$ $(m \geq 0)$ be the i-th branch. In the i-th stage, S_i is the subsequence $(v_{j_0}, \ldots, v_{j_k})$ of π_i $(0 \leq k \leq m, j_0 < \ldots < j_k)$ such that for every $1 \leq j \leq m$, $v_j \in S_i$ iff the exclusion condition is falsified.*

Proof: We show the claim by induction on stages $i = 1, \ldots, n+1$. First, suppose that $i = 1$. Since $lca_1 = lca(\ell_0, \ell_1) = \top$ and $top(S) = \top$, we can see the conditions $top(S) \succ lca_i$ and $top(S) \prec lca_i$ of the while-clause and the if-clause from line 4 to line 7 are not satisfied. Thus, the algorithm never executes these clauses. Then, the algorithm pushes the current leaf ℓ_1 into the stack S. This results $S_1 = (\ell_1, \top)$. Since ℓ_1 is the first leaf, the current branch $\pi(\ell_1) = (v_0, \ldots, v_m)$ consists of only the leftmost edges, and thus the claim immediately follows.

Next, suppose that $i > 1$ and the claim holds for the $i-1$ stage. Let ℓ_i be the current leaf and π_i be the current branch, and $\pi_{i-1} = (w_0, w_1, \ldots, w_m)$ $(m \geq 0)$ be the previous branch. Now, assume that the algorithm is executing line 3 of Fig. 3. By the induction hypothesis, the contents of the stack $S_{i-1} = S$ is a subsequence of π_{i-1}. Thus, we put $S_{i-1} = (w_{i_0}, \ldots, w_{i_k})$, where $k \geq 0$ and $i_0 < \cdots < i_k$. Since lca_i is a common ancestor of ℓ_{i-1} and ℓ_i, there is some breaking point $1 \leq \mu \leq k$ such that $w_{i_0} \prec \cdots \prec w_{i_{\mu-1}} \prec lca_i \preceq w_{i_\mu} \prec \cdots \prec w_{i_k}$. Then, the while-loop from line 4 to line 5 removes from the stack S all nodes $w_{i_0} \prec \cdots \prec w_{i_{\mu-1}}$ located below lca_i. Successively, if-clause of line 6 to line 7 inserts lca_i to S if it is not inserted before, and line 8 puts ℓ_i on the top of S. Therefore, at the end of the i-th stage, the stack contains $S_i = (\ell_i \cdot lca_i \cdot w_{i_\rho}, \ldots, w_{i_k})$, where $\rho = \mu + 1$ if $lca_i = w_{i_\mu}$ and $\rho = \mu$ otherwise.

Now, for every node v_j $(1 \leq j \leq m)$ in the current path π_i satisfies the claim. Since $lca_i = lca(\ell_{i-1}, \ell_i)$ is the lowest common ancestor of ℓ_{i-1} and ℓ_i, we assume without loss of generality that the current path is $\pi_i = (u_0 = \ell_i, \ldots, u_\sigma = lca_i, w_{i_\rho}, w_{i_\rho+1}, \ldots, w_m)$ for some $\sigma \geq 0$. Clearly, $u_{\sigma-1}$ is non-leftmost child of u_σ since $u_\sigma = lca(\ell_{i-1}, \ell_i)$ is a branching node. By definition of lca_i, if $u_\sigma = lca(\ell_{i-1}, \ell_i)$ then ℓ_i is the leftmost leaf in the subtree $T_{u_{\sigma-1}}$ of OT rooted with $u_{\sigma-1}$. Therefore, we see that the path $(u_0 = \ell_i, u_1, \ldots, u_{\sigma-1})$ consists only of leftmost edges. Thus, nodes $u_1, \ldots, u_{\sigma-1}$ falsify the exclusion condition and actually they appear in S_i. Again, since $u_{\sigma-1}$ is a non-leftmost child of u_σ, u_σ falsifies the exclusion condition and actually appears in S_i. Finally, if $u_\sigma = lca(\ell_{i-1}, \ell_i)$ then the suffix $(w_{i_\rho}, w_{i_\rho+1}, \ldots, w_m)$ of π_i following u_σ is also a suffix of π_{i-1}. By induction hypothesis, the claim immediately follows. This completes the proof. ■

Corollary 6 *For every $1 \leq i \leq n+1$, if $S_i = (v_0, v_1, \ldots, v_m)$ then $v_0 = \ell_i \succ v_1 \succ \cdots \succ v_m$.*

Proof: Immediate from Lemma 5. ■

Lemma 7 *For every $1 \leq i \leq n$, the i-th rightmost branch $\Pi(\ell_i)$ is stored on the top of the stack S_i in the i-th stage, i.e., $\Pi(\ell_i)$ is a prefix of S_i. Every node in $\Pi(\ell_i)$ is strictly below lca_{i+1}.*

Algorithm Traversal_with_Array;
input: The suffix tree SA and the height array Hgt for a text T of length n;
output: The post-order traversal of all branching substrings of T.
method:
1 Let $S = \{\top\}$ be the empty stack with $(0, -1)$ as the bottom marker;
2 **foreach** $i = 1, \ldots, n+1$ **do begin**
  i-th stage:
3 $(L_i, H_i) \coloneqq (i, Hgt[i])$;
4 **while** $(H > H_i)$ **do begin**
5 $(L, H) \coloneqq pop(S)$; report (L, H);
6 $(L, H) \coloneqq top(S)$;
7 **end**
8 **if** $(H < H_i)$ **then**
9 $Push((L_i, H_i), S)$;
10 $Push((i, |A_{SA[i]}|), S)$; /* Set $S_i = S$ */
11 **end** /* for-loop */

Figure 4: A linear time algorithm for the substring traversal problem.

Algorithm Substring_Statistics;
input: The suffix tree SA for a text T of length n;
given: Aggregation operator $(A, \oplus)$ and an initial assignment $I : \{1, \ldots, |T|\} \to A$;
output: The substring statistics of each branching substrings of T.
method:
1 Compute the height array Hgt for SA;
2 Let $S = \{(0, -1; \varepsilon)\}$ be the stack;
3 **foreach** $i = 1, \ldots, n+1$ **do begin** /* i-th stage */
4 $(L_i, H_i; C_i) \coloneqq (i, Hgt[i]; \varepsilon)$;
5 **while** $(H > H_i)$ **do begin**
6 $(L, H; C) \coloneqq pop(S)$; report $(L, H; C \oplus C_i)$;
7 $C_i \coloneqq C \oplus C_i$; $(L, H; C) \coloneqq top(S)$;
8 **end**
9 **if** $(H < H_i)$ **then** $Push((L_i, H_i; C_i), S)$;
11 **else** /* $H = H_i$ */
11 $(L, H) \coloneqq pop(S)$; $Push((L, H; C \oplus C_i), S)$;
13 $Push((i, |A_{SA[i]}|), I(i))$; /* Set $S_i = S$ */
14 **end** /* for-loop */

Figure 5: A linear time algorithm for computing the substring statistics for all branching substrings.

Proof: For every $1 \le i \le n+1$, we can see that the i-th branch is written by $\pi_i = (u_0 = \ell_i, u_1, \ldots, u_{\rho-1}, u_\rho = lca_{i+1}, \ldots, u_m)$ for some $0 \le \rho \le m$ and some nodes $u_1, \ldots, u_m$. By the definition of lca, we can show that ℓ_i is the rightmost leaf of the subtree $T_{u_{\rho-1}}$ of OT rooted with $u_{\rho-1}$. Thus, it immediately follows that the path $(u_0 = \ell_i, u_1, \ldots, u_{\rho-1})$ consists only of rightmost edges. On the other hand, we see that the next leaf ℓ_{i+1} belongs to the subtree of OT rooted with the child of $u_\rho = lca_{i+1}$ next to $u_{\rho-1}$. Therefore, since lca_{i+1} is a common ancestor of ℓ_i and ℓ_{i+1}, the edge $(u_{\rho-1}, u_\rho = lca_{i+1})$ cannot be the rightmost edge below u_ρ. Combining these arguments, we know that the prefix $\Pi = (u_0 = \ell_i, u_1, \ldots, u_{\rho-1})$ coincides to the longest rightmost branch $\Pi(\ell_i)$, and thus, the claim follows from Lemma 5. ■

Theorem 8 *Given OT be any ordered and compacted tree with n leaves, the algorithm Bottom_up_Traversal of Fig. 3 outputs the post-order traversal $POT(OT)$ of OT in $O(n)$ total time and constant-time per node, provided the following operations are computable in constant time: left-to-right scanning of leaves $\ell_1, \ldots, \ell_n$, the computation of $lca(\ell_{i-1}, \ell_i)$, and the decision of the ancestor relation $\prec$.*

Proof: From Lemma 7, we can show that for every stage $1 \le i \le n$, the i-th rightmost branch $\Pi(\ell_i)$ is stacked on the top of $S = S_i$ in the end of the i-th stage, and popped up from the stack in the the beginning of the next stage (in the while-loop from line 4 to line 5) because every node in $\Pi(\ell_i)$ is descendants of $lca_{i+1} = lca(\ell_i, \ell_{i+1})$. Repeating this process for $i = 1, \ldots, n$, the algorithm outputs $\Pi(\ell_1)$ in the i-th execution of the for-loop, and finally, all nodes in the rightmost decomposition $\{\Pi(\ell_1), \ldots, \Pi(\ell_n)\}$ are reported without duplicates. By Lemma 4, this sequence $\Pi(\ell_1) \cdots \Pi(\ell_n)$ coincides to the post-order traversal of OT. ■

3.2 Linear time traversal with suffix array

Fig. 4 shows the algorithm Traversal_with_Array that simulates the algorithm Bottom_up_Traversal using scanning of the suffix array SA and Hgt. Let $T \in \Sigma^* \times \{\$\}$ be a text of length $n \ge 0$ and ST be the suffix tree for T. In the algorithm, we represent a node of the suffix tree ST by a pair of integers (L, H). For a node v, we denote by $Rank_L(v)$ and $Rank_R(v)$ the ranks of the leftmost and the rightmost leaves in the subtree ST_v rooted by v.

Definition. For a node $v \in ST$, an info (L, H) *represents* v if the info satisfies: (i) L is the rank of a leaf in the subtree ST_v rooted with v, that is, $Rank_L(v) \leq L \leq Rank_R(v)$; (ii) H is the length of the string represented by v, that is, $H = |str(v)|$. An info (L, H) is *admissible* for ST if the pair represents some node of ST.

Lemma 9 *If a pair (L, H) is admissible for ST then the node represented by (L, H) is uniquely determined.*

Proof: Suppose a pair (L, H) represents some node v. Let ℓ_L be the L-th leaf of ST corresponding to the pair. By the condition (i), we know that v is on the path π_L from ℓ_L to $root(ST)$. Since every node on the path π_L have mutually different depth, the depth H uniquely determines the node v. ■

By Lemma 9, we denote by $\nu(L, H)$ the node represented by a pair (L, H) if such a node exists. In the algorithm Bottom_up_Traversal, only two types of nodes ℓ_i and lca_i are used $(1 \leq i \leq n + 1)$. Then, they can be represented by two arrays SA and Hgt as follows.

Lemma 10 *For each $1 \leq i \leq n + 1$, the following properties hold:*

- *The leaf ℓ_i is represented by $\ell_i = \nu(i, |A_{SA[i]}|)$;*
- *The lca node lca_i is represented by $lca_i = \nu(i, Hgt[i])$.*

Proof: The proof is immediate from the definitions of SA and Hgt. ■

In general, constant-time decision of the relation $\prec$ requires the constant time lca computation [11]. In our special case, however, it is decidable by simply examining the string depth of the nodes.

Lemma 11 *In the i-th stage of Traversal_with_Array of Fig. 4, if $(L, H) = top(S)$ and $(L_i, H_i) = (i, Hgt[i])$ be the pair representing lca_i, then: $\nu(L, H) \prec (=, \succ, resp)\nu(L_i, H_i)$ iff $H < (=, > resp)Hgt[i]$.*

Proof: Suppose that the algorithm has just entered te i-th stage. By induction on stage i, we can show the following observations. At steps from the beginning of the stage to line 8, we can show from Lemma 6 that the contents of the stack S are nodes on the path π_{i-1} from ℓ_{i-1} to $\top$. If lca_i is an ancestor of ℓ_{i-1}, then lca_i is also on the path π_{i-1}. Then, the following claim holds: *If nodes $u, v \in ST$ are on the same path then $v \prec w$ iff $strdepth(v) < strdepth(w)$.* Using this claim, we can show that the statements (i)–(iii) hold up to line 8. For the steps after line 8 to the end of this stage, the contents of the stack S are on the path π_i from ℓ_i to $\top$ by Lemma 6. Furhtermore, lca_i is also on the path. Thus, it follows from the above claim that the statements (i)–(iii) hold. This completes the proof. ■

Since the pairs $(i, |A_{SA[i]}|)$ and $(i, Hgt[i])$ can be obtained from SA and Hgt in constant time, the following theorem immediately follows from Theorem 8, Lemma 9, Lemma 10, and Lemma 11.

Theorem 12 *Let $T \in \Sigma \times \{\$\}$ be a text of length n. The algorithm Traversal_with_Array of Fig. 4 computes, given the height array Hgt for T, the post-order traversal of all branching substrings of T in $O(n)$ time, where each branching substring is represented by its info.*

Proof: Let ST be the suffix tree for the text T. Clearly, ST is an ordered, compacted tree of n leaves. By Theorem 8, Lemma 9, Lemma 10, and Lemma 11, we can see that the algorithm visits all nodes of ST in the post-order. In the execution, the access to each leaf, the push and the pop operations can be done in constant time. The representation $(i, |A_{SA[i]}|)$ and $(i, Hgt[i])$ can be obtained in constant time using the arrays SA and Hgt. Thus, the computation for reporting one node takes constant time per node and thus the total computation time is $O(n)$. ■

Next, we see that the algorithm Traversal_with_Array is also efficient in the external I/O model of [12]. Suppose a computer with a fast main memory of constant size and a $10^3 \sim 10^6$

times slower external memory of unlimited size. The performance of an external memory algorithm is measured by the number I/Os in blocks of fixed size B, which is counted by the number of elements for simplicity. Roughly speaking, the goal is to develop an external memory algorithm with I/O complexity $\frac{io(n)}{B}$ if a main memory algorithm has I/O complexity $io(n)$ [12]. Furthermore, a block I/O is *bulk* [4] if it is consecutive to the previous block I/O and *random* otherwise.

Theorem 13 *Let B be the I/O block size and $T \in \Sigma \times \{\$\}$ be a text of length n. The algorithm Traversal_with_Array of Fig. 4 can be implemented to run in $\frac{5n}{B}$ (block) I/O, and furthermore, all but constant number of I/O are bulk I/O. When the maximum depth of the stack does not exceed B, the I/O complexity is reduced to $\frac{n}{B} + O(1)$ (block).*

Proof: Since the arrays SA and Hgt are given as input, the external I/O are only required for scanning of Hgt and the push/pop operations on the stack S. For each $1 \leq i \leq n$, we can see that at most one read from Hgt are made for each leaf ℓ_i and a pair of read/write to S are made for (internal or leaf) node. Since all read/write to Hgt and S are consecutive, we have the I/O complexities claimed above. If the maximum stack length is less than B then constant number of memory blocks are sufficient to keep the stack. ■

3.3 An application to the substring statistics problem

A number of string algorithms use the accumulation through the bottom-up traversal of the suffix trees. We show that the substring traversal algorithm of Section 3.2 can be modified to solve such problems. Let $\mathcal{A} = (A, \oplus, \varepsilon)$ be a monoid, where A is a domain of elements, $\oplus : A \times A \to A$ is an associative binary operator, and $\varepsilon \in A$ be the unit such that $x \oplus \varepsilon = \varepsilon \oplus x = x$ for every $x \in A$. An *initial assignment* to text T is an assignment $I : \{1, \ldots, |T|\} \to A$ of elements of A to text positions on T. Then, for each substring α of T, the *substring statistics* of α on I w.r.t. $\mathcal{A}$ is the sum $Stat(\alpha) \stackrel{\text{def}}{=} \oplus_{p \in Occ(\alpha,T)} I(p)$ of the initial elements over all occurrences of α.

Definition. Given text T and an initial assignment I w.r.t. $\mathcal{A}$, the *substring statistics problem on $\mathcal{A}$* is to report the substring statistics $Stat(\alpha)$ for each branching substring $\alpha \in substr(T)$.

For example, given a text T and a set M of some *marked* positions in T, the problem of counting the marked positions for each branching substring is an instance of this scheme. In this case, we use the monoid $\mathcal{A} = (\mathbf{N}, +, \varepsilon)$, and assignment $I(i) = 1$ if $i \in M$, and $I(i) = 0$ if $i \notin M$. A naive algorithm for the substring statistics problem takes $O(n^2)$ time. In Fig. 5, we show the algorithm Substring_Statistics for solving the substring statistics problem in linear time.

Corollary 14 *For every $(A, \oplus, \varepsilon)$, there is a linear time algorithm that computes the substring statistics for all branching substrings in $O(n)$ when $\oplus$ is constant time computable.*

Proof: By Theorem 13, this algorithm correctly simulates the post-order traversal of the suffix tree ST for T. Since the post-order traversal is one of the bottom-up traversals, when a node v is visited by the algorithm then it is ensured that all of its children have been already visited, and thus, the statistics $Stat(str(v))$ is reported. ■

4 Linear-time Construction of the Height Array

The algorithm in the previous section uses the height array Hgt to compute the traversal of all branching substrings. In this section, we present a linear-time algorithm that computes Hgt from a text T and its suffix array SA.

By definition, Hgt can be obtained by computing $Hgt[i] = lca(A_{SA[i-1]}, A_{SA[i]})$ for every $1 \leq i \leq n$. This naive method, however, takes $O(n^2)$ time in the worst case, e.g., in the case for the text $T = \texttt{aaaaaa} \cdots \texttt{a}$. Although Hgt can be obtained as a by-product of the

Algorithm Fast_Hgt;
input: A text T of length n and the suffix tree SA for T;
output: The height array Hgt for SA;
method:

```
1  Compute the inverse array SA^{-1} of SA;
2  H := 0;
3  foreach position p = 1, ..., n + 1 do begin
   i-th stage:
4      Let i := SA^{-1}[p] and q := SA[i - 1];      /* SA[i] = p and SA[i - 1] = q */
5      if (i = 1) then Hgt[i] := -1;                /* Exception */
6      if (H = 0) then Hgt[i] := lcp(T_p, T_q);
7      else Hgt[i] := H - 1 + lcp(T_{p+h-1}, T_{q+h-1});   /* Skip H - 1 letters */
8      H := Hgt[i];
9  end /* for-loop */
```

Figure 6: A linear time algorithm for computing the height array Hgt.

construction of the suffix array SA, the linear time computation of Hgt from SA is useful in many applications.

In Fig. 6, we show the algorithm Fast_Hgt for computing the height array Hgt in linear time. The following lemma is essential to the correctness of the algorithm. Recall that the p-th suffix and the suffix of rank i, respectively, refer to $T_p = T[p..n]$ and $T_{SA[i]} = T[SA[i]..n]$. The proof of the next lemma is omitted by the limitation of space.

Lemma 15 *Let $1 \leq p \leq n$ be any position on text T. Let i and j be the ranks of the p-th and the $p+1$-th suffixes such that $SA[i] = p$ and $SA[j] = p+1$. Then, $Hgt[j] \geq Hgt[i] - 1$ holds.*

Proof: First, we note that since T ends with the end marker \$, all suffixes of T are mutually distinct. Let $1 \leq i \leq n$ be any rank in SA and let $p = SA[i]$ and $q = SA[i-1]$. Clearly, $T_{SA[i-1]} <_{\text{lex}} T_{SA[i]}$ are adjacent. Let p and q be the positions such that $T_p = T_{SA[i]}$ and $T_q = T_{SA[i-1]}$. Then, if these adjacent suffixes start with different letters, then their lcp-length $Hgt[i] = lcp(T_{SA[i]}, T_{SA[i-1]}) = lcp(T_p, T_q)$ is zero, and thus the claim immediately follows since $Hgt[j] \geq 0$.

Next, we suppose that the adjacent suffixes $T_p = T_{SA[i]}$ and $T_q = T_{SA[i-1]}$ start with the same letter $T[p] = T[q]$. Then, it immediately follows that $T[p+1]$ and $T[q+1]$ at least have a common prefix of length $Hgt[i]-1 = lcp(T_{SA[i]}, T_{SA[i-1]})-1 = lcp(T_p, T_q)-1 = lcp(T_{p+1}, T_{q+1})$ (Eq. 1). Let $1 < j \leq n$ be the integer such that $SA[j] = p+1$. Then, $T_{SA[j-1]}$ is obviously the lexicographically largest suffix such that $T_{SA[j-1]} <_{\text{lex}} T_{SA[j]} = T_{p+1}$. On the other hand, since $T[q] = T[p]$, $T_q <_{\text{lex}} T_p$ implies that $T_{q+1} <_{\text{lex}} T_{p+1}$. Thus, it immediately follows that $T_{q+1} <_{\text{lex}} T_{SA[j-1]} <_{\text{lex}} T_{SA[j]}$. By the definition of the lexicographic order, if $T_{q+1} <_{\text{lex}} T_{SA[j-1]} <_{\text{lex}} T_{SA[j]} = T_{p+1}$ then $lcp(T_{p+1}, T_{q+1}) \leq lcp(T_{SA[j]}, T_{SA[j-1]})$ (Eq. 2). Combining Eq. 1 and Eq. 2, we have $Hgt[j] = lcp(T_{SA[j]}, T_{SA[j-1]}) \geq lcp(T_{p+1}, T_{q+1}) \geq Hgt[i] - 1$. ■

From the above lemma, we know that if the lcp-length, say H, of two adjacent suffixes is non-zero then we can skip at least $H-1$ letter comparisons in computation of the next lcp-length. A letter comparison is called *redundant* if the text position is already examined, where we only mark one of the two letters consistently. Then, by a similar discussion to [10], we see that at each stage, at most one redundant comparison is made. Hence, we have the following theorem.

Theorem 16 *Given a text T and the suffix array SA for T, the algorithm Fast_Hgt of Fig. 6 computes the height array Hgt for SA in $O(n)$ time.*

Since the access to SA and SA^{-1} are random, unfortunately, this algorithm makes $2n + \frac{2n}{B}$ I/O but not $O(\frac{n}{B})$ in the external memory model with block size B, thus not I/O efficient.

5 Experimental Results

We implemented the algorithm Traversal_with_Array and Linear_time_Hgt of Section 3 Section 4 on workstations (Sun UltraSPARC 300MHz, 256MB, g++ on Solaris 2.6). By experiments on the English text data of 5.3MB [6], we compare the algorithm to a traversal algorithm with the suffix array by binary search and an algorithm with the suffix tree. In Table 1, we show the computation time of these algorithms. In the first, the second and the third columns, we show the computation times for naive traversal algorithm (Binary_Trav), our algorithm Traversal_with_Array (Trav_Array) and the algorithm with the suffix tree (Trav_Tree). The preprocessing time for building Hgt is not included. The fourth and the fifth columns show the computation times of the preprocess of Hgt by naive algorithm Naive_Hgt and the fast algorithm Fast_Hgt of Section 4. The height and the branching of the suffix tree were 9.15 and 2.91 (edges) in average, respectively.

Table 1: Comparison of the computation time on an English text collection of 5.3MB.

	Substring traversal			Height array build	
Algorithm	Binary_Trav	Trav_Array	Trav_Tree	Naive_Hgt	Fast_Hgt
Time (sec)	13.62	1.94	2.07	17.59	7.81

From Table 1, we observe that our algorithms are significantly faster than the naive algorithms on both problems. For the substring traversal problem, the space occupancies are $7N$ bytes and $15N$ bytes per letter including text, and the construction time are about 13.2 sec and 11.2 sec, respectively, for the suffix array and the suffix tree. The performance of our suffix array-based method is similar to that of the suffix tree based method, but easy to scale in space.

6 Conclusion

In this paper, we presented a simple linear-time algorithm for simulating the traversal of the suffix tree using the suffix array and the height array, and also showed that the height array is computable from a text and its suffix array in linear time. In [1, 2], we have developed efficient algorithms for a text mining problem called the *optimal pattern discovery* based on the techniques presented here where the traverse algorithm with suffix array is used for storing the sparse suffixes allowing quick reconstruction.

Acknowledgments

We would like to thank to Masayuki Takeda and Ayumi Shinohara for discussions and comments on this issue.

References

[1] H. Arimura, J. Abe, R. Fujino, H. Sakamoto, S. Shimozono, S. Arikawa, Text data mining: discovery of important keywords in the cyberspace, In *Proc. IEEE Kyoto ICDL*, 2001.

[2] H. Arimura, S. Arikawa, S. Shimozono, Efficient discovery of optimal word-association patterns in large text databases, *New Generation Computing*, 18, 49–60, 2000.

[3] J. L. Bentley, R. Sedgewick, Fast algorithms for sorting and searching strings, In *Proc. SODA'97*, 360–369 (1997).

[4] A. Crauser and P. Ferragina, On constructing suffix arrays in external memory, *Algorithmica* (2000). Also appeared in Proc. the 7th ESA, LNCS 1643, 224–235 (1999).

[5] G. Gonnet, R. Baeza-Yates and T. Snider, New indices for text: Pat trees and pat arrays. *Information Retrieval:Data Structures and Algorithms*, pages 66–82, 1992.

[6] R. Harris, Abstract Index, Monash Univ (1998).

[7] T. Kasai, H. Arimura, S. Arikawa, Linear-Time Substring Traversal with Suffix Arrays, DOI-TR 185, Feb. 2001.

[8] N. J. Larsson, K. Sadakane, Faster suffix sorting, *Technical Report*, LU-CS-TR-99-214, Department of Computer Science, Lund University, Sweden (1999).

[9] E. M. McCreight, A space-economical suffix tree construction algorithm, *JACM*, 23(2):262-272, 1976.

[10] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM J. Computing*, 22(5), 935–948 (1993).

[11] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplifications an parallelization, *SIAM J. Computing*, 17, 1253–1262, 1988.

[12] J. S. Vitter, External Memory Algorithms, In *Proc. ACM PODS'98*, 119–128 (1998).

[13] J. T. L. Wang, G. W. Chirn, T. G. Marr, B. Shapiro, D. Shasha and K. Zhang, In *Proc. SIGMOD'94*, 115–125, 1994.