ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221313707

Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications

Conference Paper *in* Lecture Notes in Computer Science · June 2001

DOI: 10.1007/3-540-48194-X_17 · Source: DBLP

CITATIONS
427

READS
1,391

5 authors, including:

Gunho Lee
University of California, Berkeley
10 PUBLICATIONS **14,553** CITATIONS

SEE PROFILE

Hiroki Arimura
Hokkaido University
188 PUBLICATIONS **3,561** CITATIONS

SEE PROFILE

Arikawa Setsuo
Kyushu University
181 PUBLICATIONS **3,162** CITATIONS

SEE PROFILE

Kunsoo Park
Seoul National University
173 PUBLICATIONS **3,328** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project text mining View project

All content following this page was uploaded by Gunho Lee on 16 May 2014.

The user has requested enhancement of the downloaded file.

Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications

Toru Kasai[1], Gunho Lee[2], Hiroki Arimura[1,3], Setsuo Arikawa[1], and Kunsoo Park[2⋆]

[1] Department of Informatics, Kyushu University
Fukuoka 812-8581, Japan
{arim,arikawa}@i.kyushu-u.ac.jp
[2] School of Computer Science and Engineering
Seoul National University, Seoul 151-742, Korea
{ghlee,kpark}@theory.snu.ac.kr
[3] PRESTO, Japan Science and Technology Corporation, Japan

Abstract. We present a linear-time algorithm to compute the longest common prefix information in suffix arrays. As two applications of our algorithm, we show that our algorithm is crucial to the effective use of block-sorting compression, and we present a linear-time algorithm to simulate the bottom-up traversal of a suffix tree with a suffix array combined with the longest common prefix information.

1 Introduction

The suffix array [16] is a space-efficient data structure that allows efficient searching of a text for any given pattern. The suffix array is basically a sorted array *Pos* of all the suffixes of a text. A suffix array for a text of length n can be built in $O(n \log n)$ time, and searching the text for a pattern of length m can be done in $O(m \log n)$ time by a binary search. When a suffix array is coupled with information about the longest common prefixes (lcps) of some elements in the suffix array, string searches can be speeded up to $O(m + \log n)$ time. The *lcp* information is usually computed during the construction of suffix arrays [15,11]. In some cases, however, the *lcp* information may not be readily available.

In this paper we consider the *lcp problem in suffix arrays* that is to compute the lcp information from a text and its *Pos* array, and present a linear-time algorithm for the problem. We also describe two applications of our algorithm, i.e., block-sorting compression and the substring traversal problem.

The block-sorting algorithm [4] is a text compression method with good balance of compression ratio and speed. The original text can be decoded in linear time from block-sorting compression. An advantage of block-sorting compression is that the suffix array of the original text can also be obtained in the process of

⋆ This work was supported by the Brain Korea 21 Project.

A. Amir and G.M. Landau (Eds.): CPM 2001, LNCS 2089, pp. 181–192, 2001.
© Springer-Verlag Berlin Heidelberg 2001

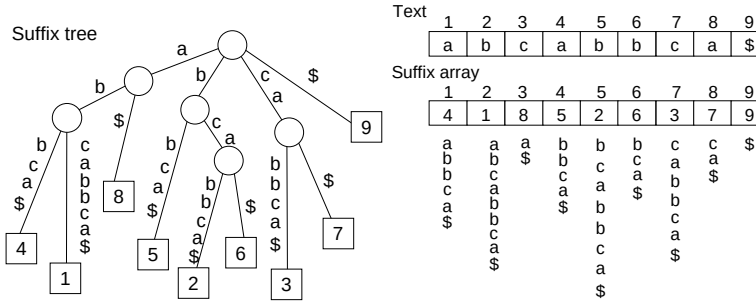

Fig. 1. An example of the suffix tree and the suffix array.

decoding. This means that we can compress a text and its suffix array together by simply using the block-sorting algorithm. This fact can be used for storing and transferring large full-text databases. However, the *lcp* information that is necessary for efficient searching is not obtained during the decoding of block-sorting compression. With our algorithm, block-sorting compression can be used more effectively to store a text and its suffix array.

The *substring traversal problem* is to enumerate all branching substrings appearing in a given text. Although the problem is easily solvable by a bottom-up traversal of the suffix tree, recent large scale applications in bioinformatics and data mining require a more practical and scalable solution for the problem [2].

We present a simple linear-time algorithm that simulates the bottom-up traversal of a suffix tree with a suffix array combined with the *lcp* information. Our algorithm is space-efficient and I/O-efficient, i.e., it requires only $7n$ bytes including the text while the suffix tree requires at least $15n$ bytes, and it has a good I/O complexity of $5n/B$ blocks. Furthermore, the algorithm can be modified to solve a class of problems based on the occurrence count of each branching substring, which include the longest common substring problem [12], the square/tandem repeat problem [22], and the frequent/optimal substring problem [2,3,9]. Experiments on English text data show that our proposed algorithms run efficiently in practice.

2 Preliminaries

Let $A = a_1 a_2 \cdots a_{n-1}\$$ be a text of length $n \geq 1$. In what follows, we assume that A ends with a special end marker \$ that does not appear in other positions. Let A_i denote the suffix of A that starts at position i. For a substring S of A, we denote by $Occ(S, A)$ the set of all occurrences of S in A. Let $\equiv_A$ be an equivalence relation on substrings defined as follows: For any substrings S, S', the relation $S \equiv_A S'$ holds if and only if $Occ(S, A) = Occ(S', A)$. A substring S of A is *branching* if S is the longest common prefix of distinct suffixes A_i and A_j $(i \neq j)$.

The *suffix array* of a text A [16] is a sorted array $Pos[1..n]$ of all the suffixes of A, i.e., $Pos[k] = i$ if A_i is lexicographically the k-th suffix. The *suffix tree* [17] is a data structure for storing all branching substrings of A, which is the compacted trie ST for all suffixes of A. The suffix tree has at most $2n-1$ nodes and can be stored in $O(n)$ space. The suffix tree ST of a text of length n can be constructed in $O(n)$ time [17,23,5]. The suffix array Pos of A coincides with the list of the leaves of ST ordered from left to right. In Fig. 1, we show the suffix tree and the suffix array of string $A =$ `abcabbca`\$. We denote by $str(v)$ the substring of A obtained by concatenating the labels on the path from the root to v. The following lemma is well known [17].

Lemma 1. *Let S be any substring of A. Then, the following 1–3 are equivalent.*

1. *S is branching.*
2. *S is the unique longest member of the equivalence class of S w.r.t. $\equiv_A$.*
3. *$S = str(v)$ for some internal node v of the suffix tree of A.*

We denote by $lcp(A, B)$ the length of the longest common prefix between strings A and B. The *lcp*s between suffixes that are adjacent in the sorted Pos array are denoted by an array $Height$: $Height[k] = lcp(A_{pos[k-1]}, A_{pos[k]})$ for $2 \leq k \leq n$. All the necessary *lcp*s for $O(m + \log n)$ search (called arrays *Llcp* and *Rlcp* in [16]) can be computed easily in $O(n)$ time from array $Height$ [11,15]. Therefore, we define the *lcp* problem as follows.

Definition 1. *The* lcp problem in suffix arrays *is to compute the $Height$ array from a text A and its suffix array Pos.*

For any substring S of A, the suffix array Pos gives a compact representation of all occurrences of S. The set of all occurrences of S occupy a contiguous interval $[L, R] \subseteq \{1, \ldots, n\}$, namely, $Occ(S, A) = \{Pos[k] : L \leq k \leq R\}$. We call the pair (L, R) the *rank interval* of S. Then, the *triple* for S is the triple (L, R, H) of integers, where (L, R) is the rank interval of S and $H = |S|$ is the length of S. If necessary, the substring can be immediately obtained by $S = A[Pos[L]..Pos[L] + H - 1]$.

A *bottom-up traversal* of the suffix tree is any list L of its nodes such that each node appears exactly once in L, and a node appears in L only after all of its children appear. The *post-order traversal* [1] is an example of bottom-up traversals. A *bottom-up substring traversal* of A is a list L of the triples (L, R, H) for all branching substrings of A which is generated by a bottom-up traversal of the suffix tree of A. Then, the substring traversal problem is stated as follows.

Definition 2. *The* substring traversal problem *is to compute the substring traversal L for a text A.*

This problem is linear time solvable by a post-order traversal of the suffix tree ST. Unfortunately, it is difficult to solve this problem with the suffix array Pos alone because Pos has lost the information on tree topology. The array $Height$ has the information on the tree topology which is lost in the suffix array Pos.

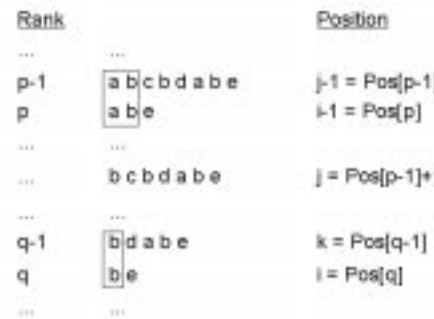

Fig. 2. An example of sorted suffixes and *lcp*s.

Lemma 2. *A substring S of a text A is branching if and only if there exists some rank $1 \leq k \leq n$ such that S is the longest common prefix of the adjacent suffixes $A_{Pos[k-1]}$ and $A_{Pos[k]}$.*

From Lemma 2, we can compute the list of all branching substrings associated with the *in-order traversal* of ST simply by reporting $A[Pos[k]..Pos[k] + Height[k] - 1]$ for every rank $1 \leq k \leq n$. Unfortunately, the obtained list may contain duplicates since ST is not a binary tree. Furthermore, there is no obvious way to compute either the associated rank intervals (L, R) or the post-order traversal.

3 Linear-Time lcp Computation

In the *lcp* computation, we will use an intermediate array $Rank$. The array $Rank$ is defined as the inverse function of Pos, and it can be obtained immediately when the Pos array is given: If $Pos[k] = i$, then $Rank[i] = k$.

3.1 Properties of lcp

The *lcp* between two suffixes is the minimum of the *lcp*s of all pairs of adjacent suffixes between them on the Pos array [16]. That is,

$$lcp(A_{Pos[x]}, A_{Pos[z]}) = \min_{x<y\leq z} \{lcp(A_{Pos[y-1]}, A_{Pos[y]})\}.$$

This implies that the *lcp* of a pair of adjacent suffixes on Pos is greater than or equal to the *lcp* of a pair of suffixes that surround them.

Fact 1. $lcp(A_{Pos[y-1]}, A_{Pos[y]}) \geq lcp(A_{Pos[x]}, A_{Pos[z]}), \quad x < y \leq z.$

When the *lcp* between a pair of adjacent suffixes on Pos is greater than 1, the lexicographical order of the suffixes is preserved when the first character of each suffix is deleted.

Fact 2. If $lcp(A_{Pos[x-1]}, A_{Pos[x]}) > 1$, then

$$Rank[Pos[x-1]+1] < Rank[Pos[x]+1].$$

In this case, the *lcp* between $A_{Pos[x-1]+1}$ and $A_{Pos[x]+1}$ is one less than the *lcp* between $A_{Pos[x-1]}$ and $A_{Pos[x]}$.

Fact 3. If $lcp(A_{Pos[x-1]}, A_{Pos[x]}) > 1$, then

$$lcp(A_{Pos[x-1]+1}, A_{Pos[x]+1}) = lcp(A_{Pos[x-1]}, A_{Pos[x]}) - 1.$$

Now we consider the following problem: compute the *lcp* between a suffix A_i and its adjacent suffix on Pos when the *lcp* between A_{i-1} and its adjacent suffix is known. For notational convenience, let $p = Rank[i-1]$ and $q = Rank[i]$. Also let $j-1 = Pos[p-1]$ and $k = Pos[q-1]$. See Fig. 2. That is, we want to compute $Height[q]$ when $Height[p]$ is given.

Lemma 3. If $lcp(A_{j-1}, A_{i-1}) > 1$ then $lcp(A_k, A_i) \geq lcp(A_j, A_i)$.

Proof. Since $lcp(A_{j-1}, A_{i-1}) > 1$, we have $Rank[j] < Rank[i]$ by Fact 2. Since $Rank[j] \leq Rank[k] = Rank[i] - 1$, we get $lcp(A_k, A_i) \geq lcp(A_j, A_i)$ by Fact 1.

Theorem 1. If $Height[p] = lcp(A_{j-1}, A_{i-1}) > 1$ then

$$Height[q] = lcp(A_k, A_i) \geq Height[p] - 1.$$

Proof.

$$\begin{aligned} lcp(A_k, A_i) &\geq lcp(A_j, A_i) && \text{(by Lemma 3)} \\ &= lcp(A_{j-1}, A_{i-1}) - 1. && \text{(by Fact 3)} \end{aligned}$$

By Theorem 1, when the *lcp* between suffix A_{i-1} and its adjacent suffix is h, suffix A_i and its adjacent suffix on Pos has a common prefix of length at least $h-1$. Therefore, it suffices to compare from the h-th characters for computing the *lcp* between suffix A_i and its adjacent suffix. If h is less than or equal to 1, we will compare from the first characters.

3.2 Algorithm and Analysis

We now present the algorithm *GetHeight* that solves the lcp problem in suffix arrays. By Theorem 1, we do not need to compare all characters when we compute the *lcp* between a suffix and its adjacent suffix on Pos. To compute all the *lcp*s of adjacent suffixes on Pos efficiently, we examine the suffixes from A_1 to A_n in order.

Theorem 2. *Algorithm GetHeight computes array Height in $O(n)$.*

```
Algorithm GetHeight
input: A text A and its suffix array Pos
   1  for i:=1 to n do
   2     Rank[Pos[i]] := i
   3  od
   4  h:=0
   5  for i:=1 to n do
   6     if Rank[i] > 1 then
   7        k := Pos[Rank[i]-1]
   8        while A[i+h] = A[j+h] do
   9           h := h+1
   10       od
   11       Height[Rank[i]] := h
   12       if h > 0 then h := h-1 fi
   13    fi
   14 od
```

Fig. 3. The linear-time algorithm for the lcp problem.

Proof. The correctness of $GetHeight$ follows from previous discussions. The execution time of the algorithm is proportional to the number of times line 9 is executed, since line 9 is the innermost loop of $GetHeight$. The value of h increases one by one in line 9, and it is always less than n due to the end marker \$. Since the initial value of h is 0 and it decreases at most n times in line 12, h increases at most $2n$ times. Therefore, the time complexity of Algorithm $GetHeight$ is $O(n)$.

4 Application to Block-Sorting Compression

4.1 Block-Sorting Compression

The block-sorting algorithm is a text compression method with good balance of compression ratio and speed [4,8]. It achieves speed comparable to dictionary compressors, but obtains compression close to the best statistical compressor. The block-sorting algorithm is used in *bzip*2 [21].

The encoder of block-sorting consists of three processes: the Burrows-Wheeler transformation, move-to-front encoding and entropy coding. The Burrows-Wheeler transformation (BWT) is the most time-consuming process. It transforms a string A of length n by forming the n rotations (cyclic shifts) of A, sorting them lexicographically, and extracting the last character of each of the rotations. A string L is formed from these characters, where the i-th character of L is the last character of the i-th sorted rotation. In addition to L, the BWT computes the index I of the original string A in the sorted list of rotations. Fig. 4 is an example of BWT where A='abraca'. A move-to-front encoding encodes an instance of a character ch by the count of distinct characters between itself and the previous occurrence of ch.

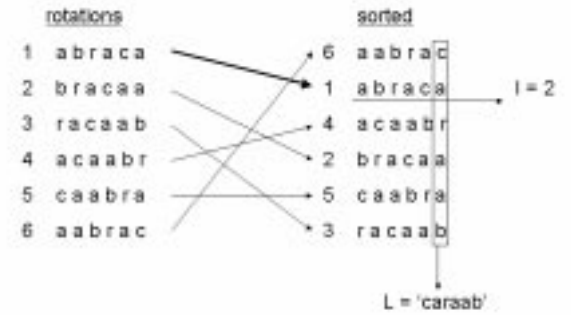

Fig. 4. An example of the Burrows-Wheeler transformation.

As a result of BWT, the locality of characters of L goes higher than that of A [4]. So, when applied to the string L, the output of a move-to-front encoder will be dominated by low numbers, which can be effectively encoded with Huffman coding or run-length coding.

The decoder of block-sorting is the reverse of the encoder. Decoding speed of an entropy code depends on the used method, but Huffman coding or run-length coding, which is generally used for encoding, can be reversed in linear time. A move-to-front code can be reversed in $O(n)$ time, and the original string A, the reverse of the BWT, can be recovered from L and I in $O(n)$ time. Therefore, the block-sorting decompression takes linear time in general.

4.2 Block-Sorting and Suffix Arrays

The first step of block-sorting, the BWT, is similar to the construction process of a suffix array. The BWT takes much time for sorting the suffixes. However, its reverse transformation from L and I to A is quickly computed in linear time by a radix-sort-like procedure. Moreover, the suffix array *Pos* of A can be computed immediately when the compressed text is decoded.

To search for a pattern using the suffix array more efficiently, the *lcp* information (*Llcp* and *Rlcp*) is required. The *lcp* information can be computed in $O(n \log n)$ time when the suffix array is constructed from the original text A. With our algorithm, the *lcp* information can be computed in $O(n)$ time from the original text A and its *Pos* array. Therefore, suffix arrays can be stored and used efficiently by the block-sorting compression.

Since block-sorting has the effect of storing the compressed text and its suffix array, it can be used for storing and transferring large data. Sadakane and Imai presented a cooperative distributed text database management method unifying search and compression based on BWT [18]. Sadakane also presented a modified BWT for case-insensitive search with the suffix array [19]. Recently, Sadakane proposed a compressed text database system [20] based on the compressed suffix array [10]. Ferragina and Manzini [7,6] proposed a data structure that supports search operations without uncompressing the block-sorting compression.

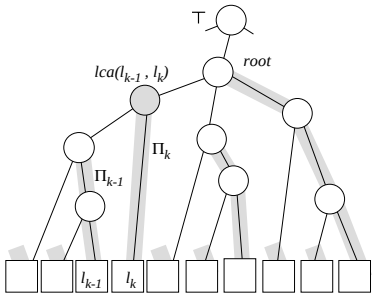

Fig. 5. An example of the rightmost branch decompositions.

```
Algorithm BottomUpTraverse;
input: An ordered and compacted tree T
with n ≥ 0 leaves ℓ1,...,ℓn.
   1  S:={⊤} /* Initialize the stack S */
   2  for k:=1 to n+1 do /* k-th stage */
   3     v := lca(ℓk−1, ℓk);
   4     while (depth(top(S)) > depth(v)) do
   5        v := Pop(S) and report v; od;
   6     if (depth(top(S)) < depth(v)) then
   7        Push(v,S); fi;
   8     Push(ℓk, S);      /* Set Sk = S */
   9  od /* for-loop */
```

Fig. 6. The algorithm to compute the post-order traversal of an ordered tree.

5 Bottom-Up Traversal of Suffix Trees

5.1 Properties of the Post-Order Traversal

An ordered tree T is *compacted* if every internal node of T has at least two children. Let T be an ordered and compacted tree with $n \geq 0$ leaves $\ell_1, \ldots, \ell_n$. In what follows, a *path* in T is always written in the upward direction. That is, a *path* (or upward path) is a sequence $\pi = (v_0, v_1, \ldots, v_m)$ $(m \geq 0)$ of nodes in T such that v_i is the parent of v_{i-1} for every $1 \leq i \leq m$. The length of π is $|\pi| = m$. A path π from the k-th leaf $(1 \leq k \leq n)$ to the root is called the k-th *branch* of T and denoted by $\pi(\ell_k)$. A *node-depth* of a node v, denoted by $depth(v)$, is the length of the path from v to the root. We write $u \preceq v$ $(u \prec v)$ if a node u is an ancestor (proper ancestor) of node v. We denote by $lca(u, v)$ the *lowest common ancestor* of nodes u and v and by $\pi(\ell)$ the branch starting at a leaf ℓ. Let ℓ be any leaf. A *rightmost branch* (RM branch, for short) starting with ℓ, denoted by $\Pi(\ell)$, is the longest branch $\pi = (v_0 = \ell, v_1, \ldots, v_m)$ $(m \geq 0)$ starting at ℓ that consists of only *rightmost edges*, that is, v_{i-1} is the rightmost child of v_i for every $1 \leq i \leq m$.

$\Pi(\ell_k)$ is called the *k-th RM branch*. Since the set $\{\Pi(\ell_1), \ldots, \Pi(\ell_n)\}$ of all RM branches of T is called the *RM branch decomposition* of T since it is a partition of T. Fig. 5 shows an example of the RM branch decompositions, where each shadowed line indicates an RM branch (See below for the special node $\top$).

Lemma 4. *The post-order traversal of an ordered tree T equals the concatenation $\Pi(\ell_1) \cdots \Pi(\ell_n)$ of the RM branches of T from left to right.*

5.2 Algorithm for Bottom-Up Substring Traversal

From now on, we consider a method to compute the post-order traversal of an ordered compacted tree T with $n \geq 0$ leaves $\ell_1, \ldots, \ell_n$ when the lowest common

ancestors of adjacent leaves and the depth of a node are available. Fig. 6 shows the algorithm *BottomUpTraverse* for the problem. In the algorithm, we assume a special *top* node$\top$ such that $\top \prec v$ for every v in T and special leaves ℓ_0 and ℓ_{n+1} such that $lca(\ell_0, \ell_1) = lca(\ell_n, \ell_{n+1}) = \top$ (See Fig. 5).

Scanning the height array $Height$ from left to right, the algorithm enumerates the nodes of T without duplicates by a sequence of push/pop operations to a stack S as follows. During the scan, a leaf node, say ℓ_k, is pushed into the stack S when it is first encountered at stage k and popped immediately at stage $k+1$. The case for internal nodes is more complicated (See Fig. 5). Conceptually, a node v is pusded when it is visited from below at the first time and popped when it is visited at the last time in the depth-first search of T.

An internal node v is pushed into the stack when the leftmost leaf of the second child of v, say ℓ_k, is encountered at the first time in the scan, i.e., $v = lca(\ell_{k-1}, \ell_k)$. Then, v is popped from the stack S when the leftmost leaf of the next right sibling of v, is encountered in the scan. Then, $p = lca(\ell_{k-1}, \ell_k)$ is the parent of v. Since the tree is compacted, the second leftmost leaf always exists for every internal node. Thus from Lemma 2, the algorithm *BottomUpTraverse* enumerates all nodes without duplicates by a scan of $Height$.

To see that the algorithm correctly computes the post-order traversal of T, we need to know the precise contents of the stack during the scan. A key observation is that if an internal node v is $lca(\ell_{k-1}, \ell_k)$ for some k then v is on the k-th branch from ℓ_k to the root and all nodes of $\Pi(\ell_{k-1})$ are proper descendants of v. We gives the following lemma without proof due to the space limitation (See [14] for the complete proof).

Lemma 5. *Let us consider the algorithm BottomUpTraverse of Fig. 6. For any stage $1 \leq k \leq n+1$, the contents of the stack S at the beginning of the k-th stage is the subsequence $S_k = (v_{j_0}, \ldots, v_{j_k})$ of the k-th branch $\pi_k = (v_0 = \ell_k, v_1, \ldots, v_m = \top)$ $(m \geq 0)$ such that for every $0 \leq j \leq m$, $v_j \in S_k$ if and only if the following* inclusion condition *holds at position j: either (i) $j = 0$ or (ii) v_{j-1} is not the leftmost child of v_j.*

From Lemma 5, we see that in the end of every stage k, the k-th RM branch $\Pi(\ell_{k-1})$ is stored on the top of the stack S. Then, $\Pi(\ell_{k-1})$ is deleted from the stack S when ℓ_k is encountered in the scan. By repeating this process, the algorithm finally outputs all RM branches $\Pi(\ell_1), \ldots, \Pi(\ell_n)$ of T from left to right. Hence, the next lemma immediately follows from Lemma 4.

Lemma 6. *The algorithm BottomUpTraverse of Fig. 6 computes the post-order traversal of an ordered compacted tree with n leaves in $O(n)$ time when the node-depth for a node and the lowest common ancestor of adjacent leaves are constant time computable.*

Now we present a linear time algorithm for the substring traversal problem when the height array and the suffix array of A is given. Fig. 7 shows the algorithm *TraverseWithArray* to compute the list of triples for text A generated by the post-order traversal of a suffix tree. In the algorithm, we encode a node v

```
Algorithm TraverseWithArray;
input: The height array Height and the suffix array Pos for a text A;
   1  S:= (-1, -1); n:=|T| /* Initialize the stack S */
   2  for k:=1 to n+1 do   /* k-th stage */
   3     (Llca, Hlca) := (k-1, Height[k]);
   4     (L, H) := top(S);
   5     while (H > Hlca) do
   6        (L, H) := pop(S), R := k-1; Then, report triple (L, R, H);
   7        Llca := L;     /* Update the left boundary */
   8        (L, H) := top(S);
   9     od
   10    if (H < Hlca) then
   11       Push((Llca,Hlca),S); fi;
   12    Push((k, n - Pos[k] + 1), S);    /* Set Sk = S */
   13 od /* for-loop */
```

Fig. 7. A linear time algorithm for the substring traversal problem.

by any pair (L, H) such that L and H are the any occurrence and the length of the substring $str(v)$, respectively. The top node is encoded by $(-1, -1)$.

Recall that there were only two types of nodes processed in the algorithm *BottomUpTraverse*, a leaf and the lca of adjacent leaves. Thus for any rank $1 \leq k \leq n+1$, we encode v by (L, H) as follows: (i) if v is the leaf ℓ_k then $(L, H) = (k, |A_{Pos[k]}|) = (k, n - Pos[k] + 1)$ and (ii) if v is the lca node $lca(\ell_{k-1}, \ell_k)$ then $(L, H) = (k-1, Height[k])$. The depth of the node v is obviously given by H. From Lemma 6, we know that the algorithm correctly simulates *BottomUpTraverse*.

We then consider the computation of the rank intervals. Suppose a pair (L, H) is popped from the stack S at stage k and it represents a node v. By induction on the number of nodes below v on the $(k-1)$-th path, we can show that L is the rank of the leftmost leaf of v, where the value of L is kept at the variable `Llca` at Line 7 of the algorithm. Since v is on the $(k-1)$-th RM branch, $R = k-1$ is obviously the rank of the rightmost leaf of v. Therefore, (L, R, H) is the triple of v, and the next theorem follows from Lemma 6.

Theorem 3. *The algorithm TraverseWithArray of Fig. 7 computes in $O(n)$ time the list of all triples generated by the post-order traversal of the suffix tree of a text A of length n when the height array and the suffix array of A is given.*

Hence, the substring traversal problem is solvable in linear time when the height array $Height$ of a text A is given. Since the algorithm *TraverseWithArray* makes only sequential I/Os and does not access the text A, we can also see that the algorithm is I/O efficient in the external I/O model of [24] (See [14]).

6 Experimental Results

We run experiments on a real dataset. For the height array construction, we implemented the naive $O(n^2)$ time algorithm (Abbreviated as NaiveHeight)

and the linear time algorithm *GetHeight* (GetHeight). For the bottom-up substring traversal in Section 5, we implemented the algorithm with the suffix tree (TravTree), the naive algorithm with binary search on the suffix array (TravBinary), and the algorithm *TraverseWithArray* (TravHeight).

Table 1. Comparison of the computation time on English texts.

	Height array construction		Substring traversal		
Algorithm	NaiveHeight	GetHeight	TravTree	TravBinary	TravHeight
Time (sec)	17.59	7.81	2.07	13.62	1.94

In Table 1, we show the running time of the algorithms on an English text of 5.3MB [13] and a workstation (Sun UltraSPARC 300MHz, 256MB, g++ on Solaris 2.6). In the substring traversal, the preprocessing time for building the height array is not included. For the height array construction, we see from this table that GetHeight is faster than NaiveHeight more than twice on this test data. For the substring traversal, TravHeight is as fast as TravTree when the height array is precomputed, and faster than TravBinary even when the computation time of the height array is included.

References

1. A. V. Aho, J. E. Hopcroft and U. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.
2. H. Arimura, S. Arikawa and S. Shimozono, Efficient discovery of optimal word-association patterns in large text databases, *New Generation Comput.*, 18, 49–60, 2000.
3. H. Arimura, H. Asaka, H. Sakamoto and S. Arikawa, Efficient discovery of proximity patterns with suffix arrays, In *Proc. CPM 2001*, Poster paper, LNCS, Springer-Verlag, 2001. (In this volumn).
4. M. Burrows and D. J. Wheeler, A block-sorting lossless data compression algorithm, *Digital Systems Research Center Research Report 124*, 1994.
5. M. Farach-Colton, P. Ferragina and S. Muthukrishnan, On the sorting-complexity of suffix tree construction, *Journal of the ACM*, Vol.47, No.6, 987–1011, 2000.
6. P. Ferragina and G. Manzini, Opportunistic data structures with applications, In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, 390–398 2000.
7. P. Ferragina and G. Manzini, An experimental study of an opportunistic index, In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, 269–278 2001.
8. P. Fenwick, Block sorting text compression, In *Proc. Australian Computer Science Communications*, 18(1), 193–202, 1996.
9. R. Fujino, H. Arimura and S. Arikawa, Discovering unordered and ordered phrase association patterns for text mining, In *Proc. PAKDD2000*, LNAI 1805, 281–293, 2000.
10. R. Grossi and J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, In *Proc. 32nd ACM Symposium on Theory of Computing*, 397–406, 2000.

11. D. Gusfield, An increment-by-one approach to suffix arrays and trees, *Technical Report CSE-90-39*, UC Davis, Dept. Computer Science, 1990.
12. D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, 1997.
13. R. Harris, Abstract Index, Monash Univ (1998).
14. T. Kasai, H. Arimura and S. Arikawa, Efficient substring traversal with suffix arrays, DOI-TR 185, Feb. 2001. (First appeared as T. Kasai, Fast algorithms for the subword statistics problems with suffix arrays, *Mc. Thesis*, Dept. Informatics, Kyushu Univ.,1999, In Japanese.)
15. S. E. Lee and K. Park, A new algorithm for constructing suffix arrays, *Journal of Korea Information Science Society (A)*, 24(7), 697–704, 1997.
16. U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM J. Computing*, 22(5), 935–948 (1993).
17. E. M. McCreight, A space-economical suffix tree construction algorithm, *Journal of the ACM*, 23(2), 262–272, 1976.
18. K. Sadakane and H. Imai, A cooperative distributed text database management method unifying search and compression based on the Burrows-Wheeler transformation, In *Proc. International Workshop on New Database Technologies for Collaborative Work Support and Spatio-Temporal Data Management*, 434–445, 1998.
19. K. Sadakane, A modified Burrows-Wheeler transformation for case-insensitive search with application to suffix array compression, In *Proc. Data Compression Conference*, p.548, 1999.
20. K. Sadakane, Compressed text databases with efficient query algorithms based on the compressed suffix array, In *Proc. 11th Annual International Symposium on Algorithms and Computation*, 410–421, 2000.
21. J. Seward, `http://sources.redhat.com/bzip2\/`
22. J. Stoye and D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, In *Proc. CPM'98*, LNCS, 140–152, 1998.
23. E. Ukkonen, On-line construction of suffix trees, *Algorithmica* 14, 249–260, 1995.
24. J. S. Vitter, External memory algorithms, In *Proc. PODS'98*, 119–128 (1998).