Journal of Korea Multimedia Society Vol. 13, No. 12, December 2010(pp. 1760-1766)

A Generalization of the Linearized Suffix Tree to Square Matrices

Joong Chae Na[†], Sunho Lee[††], Dong Kyue Kim[†††]

ABSTRACT

The linearized suffix tree (LST) is an array data structure supporting traversals on suffix trees. We apply this LST to two dimensional (2D) suffix trees and obtain a space-efficient substitution of 2D suffix trees. Given an $n \times n$ text matrix and an $m \times m$ pattern matrix over an alphabet Σ, our 2D-LST provides pattern matching in $O(m^2 \log|\Sigma|)$ time and $O(n^2)$ space.

Key words: Linearized Suffix trees, Two-Dimensional Suffix trees

1. INTRODUCTION

The Linearized Suffix Tree (LST) is an array representation of a suffix tree based on longest common prefix (lcp) information [1,2]. The suffix tree plays a crucial role in various string algorithms [3,4], but its weakness is the size of tree representation [1]. For a space-efficient substitution, a suffix array was proposed [5], which is a simple array of lexicographically sorted suffixes. Only on the suffix array, the LST simulates traversals on suffix trees.

The LST provides two kind of traverses that solve important problems in string processing: a bottom-up traverse and a top-down traverse [1]. The bottom-up traverse is a post-order traversal on suffix trees in $O(n)$ time by using lcp information. The top-down traverse is a branching of child with labeled with a character. This branching takes an additional table of $O(n)$ space and $O(|\Sigma|)$ time, and it is improved to $O(\log|\Sigma|)$ time by Kim et al [2]. Recently, more space-efficient suffix trees with rich functions were proposed [6,7,8], however these succinct suffix trees depends on heavy operations of compressed suffix arrays and auxiliary succinct data structures. Therefore, the LST has its own advantage in practice.

In this paper, we apply the LST to Two Dimensional (2D) matrices. Given $n \times n$ square matrix over alphabet Σ, 2D suffix trees were defined [9] and a linear time, i.e., $O(n^2)$ time, construction algorithm was proposed [10]. This tree representation has also the weakness of space complexity as in one dimensional suffix trees. Moreover, the 2D suffix trees define Lstrings or Istrings as comparison units, which are n-length substrings of rows and columns of a matrix. To compare these n-length substrings, the 2D suffix trees build extra data structures such as tries for branching substrings and suffix trees for all rows and columns. Using the linearized 2D suffix trees, we replace these complex data structures with only arrays of sorted suffixes and lcp-lengths.

※ Corresponding Author: Dong Kyue Kim, Address: (133-791) Department of Electronic Engineering, Hanyang University 17 Haengdang-Dong, Seongdong-Gu, Seoul, 133-791, Korea, TEL: +82-02-2220-2312, FAX: +82-2-2220-4926, E-mail: dqkim@hanyang.ac.kr
Receipt date: Nov. 24, 2010, Revision date: Dec. 20, 2010
Approval date: Dec. 28, 2010
[†] Department of Computer Sciences and Engineering, Sejong University
(E-mail: jcna@sejong.ac.kr)
[††] Daum Communications
(E-mail: sunholeee@gmail.com)
[†††] Department of Electronic Engineering, Hanyang University
※ This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2006-311-D00762).

Our contribution is a generalization of the LST for two dimensional square matrices. By introducing refined lcp-lengths that represent lengths of prefixes by units of characters on Σ, we obtain the generalization to two dimensional suffix trees. Our 2D-LST simulates traversals on 2D suffix trees so provides pattern matching for $m \times m$ matrices in $O(m^2 \log|\Sigma|)$ time and $O(n^2)$ space.

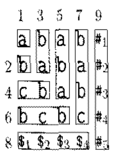
2. PRELIMINARIES

2.1 Istring for Square Matrices

We first introduce basic notations for two dimensional suffix trees and suffix arrays. Let A be an $n \times n$ square matrix over an alphabet Σ. This matrix A is assumed to have distinct and unique characters in its last column and row. We transform the matrix A into a linear string IA as in Figure 1. This transform divides A into I-shaped strings called *Icharacters* [9,10]. An Icharacter from the k-th column is its prefix of length k and one from the k-th row is its prefix of length $k-1$.

By interleaving these Icharacters from rows and columns, we obtain the linear string IA called an *Istring*. The Istring IA is a string of $2n-1$ Icharacters; we use notations $IA[i]$ for the i-th Icharacter and $IA[i..j]$ for the substring from $IA[i]$ to $IA[j]$. A general substring $IA[i..j]$ is called an *Isubstring*, and $IA[1..j]$ is an *Iprefix*. We also regard $IA[i]$ as a string, so $IA[i][k]$ is the k-th character of $IA[i]$ and $IA[i][k..l]$ is the substring from $IA[i][k]$ to $IA[i][l]$.

Given two Istrings IX and IY, the Longest Common Prefixes (*lcps*) are defined on Icharacters and on characters. The lcp on Icharacters called *Ilcp* gives the length of common Iprefixes, i.e., an Ilcp-length l means $IX[l][1..l] = IY[l][1..l]$. The refined lcp on characters called *Elcp* specifies the common prefixes between final Icharacters, i.e., an Elcp-length of a pair (l,h) means $IX[l][1..h] = IY[l][1..h]$ and $IX[1..l-1] = IY[1..l-1]$. Any Elcp-length (l,h) for $0 \le h < \left\lceil \frac{l}{2} \right\rceil$ gives the same Ilcp-length $l-1$. The Elcp-length $(l, \left\lceil \frac{l}{2} \right\rceil)$ is equivalent to $(l+1, 0)$, because $IX[l]$ is a string of $\left\lceil \frac{l}{2} \right\rceil$ characters.



$IA =$ a b ba cb aba bcb babc $\$_1\$_2\$_3\$_4$ $\#_1\#_2\#_3\#_4\#_5$
$IA(1..5) =$ a b ba cb aba
$IA(7) =$ babc
$IA(7)[1..2] =$ ba

Fig. 1. Transform from matrix A to string IA.

2.2 Isuffix Tree and Isuffix Array

Given the $n \times n$ matrix A, a suffix $A_{i,j}$ is defined as the largest square submatrix of A starting at position (i,j). $A_{i,j}$ is an $l \times l$ matrix such that $l = n - \max(i,j)$. Let $IA_{i,j}$ denote the Istring of $A_{i,j}$ and $IA_{i,j}$ is called an *Isuffix*.

Then, an Isuffix tree of A is a compacted patrica trie for all Isuffixes of A [9,10]. A leaf node corresponds to an Isuffix $IA_{i,j}$. An internal node has at least two children and each child is labeled with an Isubstring of an Isuffix, and the path from the root to an internal node represents a common Iprefix between Isuffixes. See Figure 2.

An Isuffix array of A is a lexicographically sorted array of Isuffixes of A [11]. For any two Isuffixes of A, one cannot be the other's proper prefix because of the distinct characters in the last row and column. Hence, an Isuffix $IA_{i,j}$ has its own *rank* in the Isuffix array.

The Isuffix array consists of two components: *POS* and *LCP*. The array *POS* keeps positions of Isuffixes in lexicographical order such that *POS*$[k]$ is position (i,j) of the Isuffix $IA(i,j)$ with rank k. The array *LCP* contains lcp-lengths such that *LCP*$[k]$ is the lcp-length between Isuffixes at *POS*$[k-1]$ and *POS*$[k]$. For the lcp-lengths, we will use refined lcp-lengths, i.e., Elcp-lengths. See Figure 3.

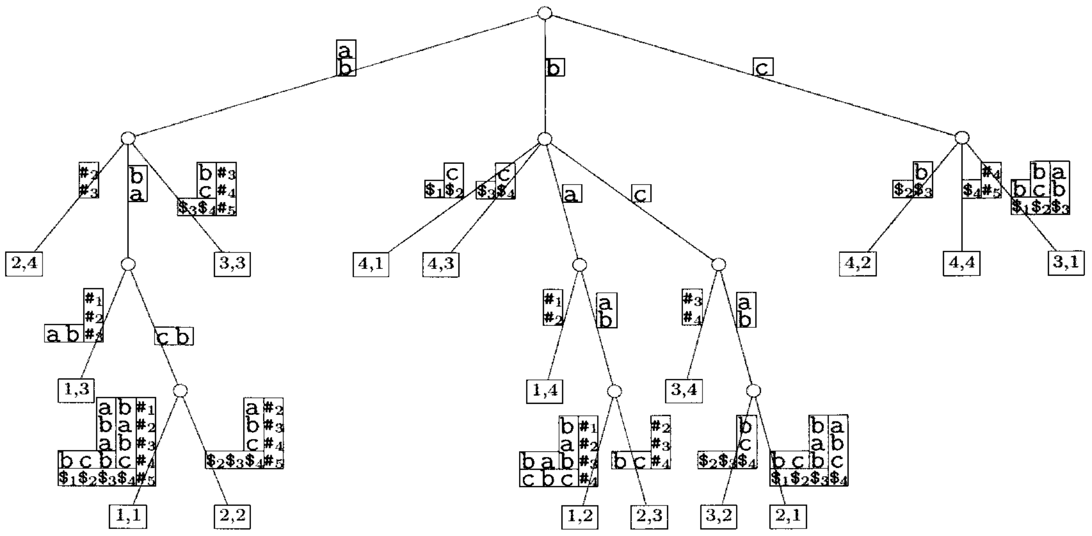


Fig. 2. Isuffix trees of square matrix *A*.

Index	POS	LCP	Isuffixes
1	(2, 4)		a b $\#_2\#_3$
2	(1, 3)	(2, 0)	a b ba ab $\#_1\#_2\#_3$
3	(1, 1)	(3, 0)	a b ba cb aba bcb babc
4	(2, 2)	(4, 2)	a b ba cb abc $\$_2\$_3\$_4$ $\$_1\$_2\$_3\$_4$
5	(3, 3)	(2, 1)	a b bc $\$_3\$_4$ $\#_3\#_4\#_5$
6	(4, 1)	(0, 0)	b $\$_1$ c$\$_2$
7	(4, 3)	(1, 0)	b $\$_3$ c$\$_4$
8	(1, 4)	(1, 0)	b a $\#_1\#_2$
9	(1, 2)	(2, 0)	b a ab ba bab cbc $\#_1\#_2\#_3\#_4$
10	(2, 3)	(3, 1)	b a ab bc $\#_2\#_3\#_4$
11	(3, 4)	(1, 0)	b c $\#_3\#_4$
12	(3, 2)	(2, 0)	b c ab $\$_2\$_3$ bc$\$_4$
13	(2, 1)	(3, 0)	b c ab bc bab $\$_1\$_2\$_3$ abc$\$_4$
14	(4, 2)	(0, 0)	c $\$_2$ b$\$_3$
15	(4, 4)	(1, 0)	c $\$_4$ $\#_4\#_5$
16	(3, 1)	(1, 0)	c b bc $\$_1\$_2$ ab$\$_3$

Fig. 3. Isuffix arrays of square matrix *A*.

3. GENERALIZATION OF LINEARIZED SUFFIX TREES

In this section, we generalize the Linearized Suffix Trees (LST) to two dimensional Isuffix arrays. The LST represents a node of suffix trees by an interval on *LCP* array. By visiting an interval in a certain order, the LST simulates two kind of traversals on suffix trees: bottom-up traversal and top-down traversal [1].

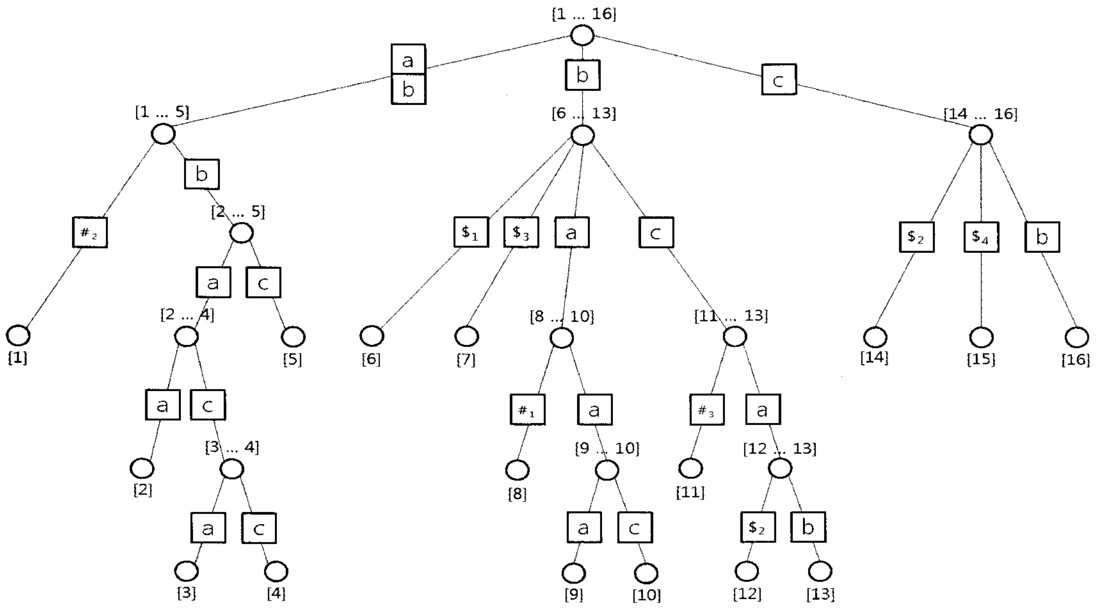
We show that these two traversals are still possible on Isuffix arrays. The bottom-up traverse visits intervals of *LCP* array in post-order. In fact, this bottom-up traversal can be applied to a general compacted trie, so we directly apply the traverse to our Isuffix arrays.

The top-down traversal is based on a branching function, $\chi ld(v,\alpha)$, that returns the child of v labeled with Icharacter α. Using the techniques in [2], we process $\chi ld(v,\alpha)$ in $O(|\alpha|\log|\Sigma|)$ time and $O(n^2)$ space. Hence, given a $m\times m$ pattern matrix, we find the occurrences of the pattern in $O(m^2\log|\Sigma|)$ time.

A problem is that a branching Icharacter is an n-length substring of rows or columns, not a character in Σ. One solution was an internal trie for branching Icharacters [9], however our 2D-LST supports a simple branching function based on tables.

3.1 Lcp-Interval Tree

An lcp-interval tree is a conceptual tree defined by parent-children relationship of lcp-intervals. Figure 4 shows an example of lcp interval trees for strings. We define general lcp-intervals on lex-



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	(2, 0)	(3, 0)	(4, 2)	(2, 1)	(0, 0)	(1, 0)	(1, 0)	(2, 0)	(3, 1)	(1, 0)	(2, 0)	(3, 0)	(0, 0)	(1, 0)	(1, 0)

Fig. 4. The lcp-interval tree and LCP array of square matrix A.

icographically sorted strings and show an one-to-one mapping between lcp-interval trees and compacted tries. Thus, we obtain conceptual Isuffix trees from Isuffix arrays and lcp arrays.

Given lexicographically sorted strings, $S_1, S_2, \cdots, S_n$, let L be an array whose element $L[i]$ is the length of lcp between S_{i-1} and S_i. An lcp-interval $[i..j]$ is defined as follows.

Definition 1 *An interval* $[i..j]$ *is an lcp-interval with length* l *if* $L[i] < l$, $L[j+1] < l$, *and* $\min L[i+1..j] = l$. *A singleton* $[i]$ *is an lcp-interval with the length of* S_i.

An lcp-interval $[i..j]$ has disjoint sub-intervals, which are defined by a position c_i such that $L[c_i] = l$. Let $c_1, c_2, \dots, c_k$ are the positions such that $L[c_i] = l$ for $1 \le i \le k$. Then, all disjoint sub-intervals $[i..c_1-1], [c_1, c_2-1], \dots, [c_k, j]$ become lcp-intervals with length $l' > l$. We regard these sub intervals as children of an lcp-interval $[i..j]$. Hence, the lcp-intervals form a tree.

This lcp-interval tree corresponds to a compacted trie representing all strings S_i's. So, all traversals on the lcp-interval tree simulate those on the compacted trie.

Lemma 1 *Given lexicographically sorted strings,* $S_1, S_2, \cdots, S_n$, *all nodes of the lcp-interval tree is one-to-one mapped to the compacted trie representing all* S_i *for* $1 \le i \le n$

Proof. We show an one-to-one map between the lcp-interval tree and the compacted trie by an induction. First, we define one-to-one map between the lcp-interval $[1..n]$ and the root node of compacted trie T_S. Next, we show one-to-one maps for children of a parent node, which has been already mapped.

Let v be a node of compacted trie and $[i..j]$ is the corresponding lcp-interval with length l. Note that $S_i, S_{i+1}, \cdots, S_j$ have the longest common prefix of length l.

i) Trie nodes to lcp-intervals: If v' be a child node of v with label α, then there exists strings of $S_{i'}, S_{i'+1}, \ldots, S_{j'}$ with the longest common prefix of length $l' = l + |\alpha|$. The interval $[i'..j']$ is an lcp-interval with length l', because $L[i'] < l'$, $L[j'+1] < l'$, and $\min L[i'..j'] = l'$. Since $l' > l$, $[i'..j']$ is a sub-interval of $[i..j]$.

ii) Lcp-intervals to trie nodes: If $[i'..j']$ is an lcp-interval with length l', then strings $S_{i'}, S_{i'+1}, \ldots, S_{j'}$ have the longest common prefix of length l'. This prefix should make a node in the trie T_S. □

The bottom-up traverse algorithm [1] visits conceptual lcp-interval trees in a post-order by comparing height information in L array. The algorithm returns a sequence of lcp-intervals in the post-order visiting. We refer to [1] for details. By Lemma 1, the traverse algorithm now finds a post-order visiting for a general compacted trie.

Note that there are two levels of lcp information in our Isuffix arrays. If we use only Ilcp-lengths, then we obtain the bottom-up traverse of Isuffix trees. A Refined Elcp-lengths are introduced to provide a top-down traverse, i.e., a node branching.

3.2 Child Table

For the top-down traverse [1,2] in lcp-interval trees, we need a branching function $\chi ld([i..j], \alpha)$ that returns the child lcp-interval of $[i..j]$ labeled with Icharacter α. The problem is that an Icharacter is actually an n-length string on Σ. A solution was an explicit internal trie in Isuffix trees by Giancarlo [9], however the lcp-interval tree on *LCP* array is a simple representation of the trie as we shown in Lemma 1. By using child tables of $O(n^2)$ space, we provide an $O(|\alpha| \log|\Sigma|)$ time branching function. For a complete description, we start with a brief of the child tables. Figure 5 shows an example of LST (including child tables) for strings.

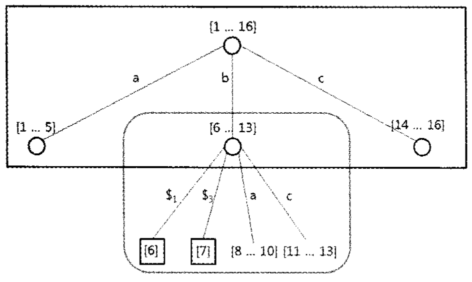
Given an lcp-interval $[i..j]$ and its $(k+1)$ disjoint sub-intervals, let c_i be the starting index of the $(i+1)$-th sub-interval, i.e., $[i..j]$ has $k+1$ sub-intervals $[i..c_1-1], [c_1..c_2-1], \ldots, [c_k..j]$. There are two methods organizing the child table *CLD*: linked lists [1] and binary trees [2].

The linked lists representation [1] make $CLD[i]$ to point the next sibling $j+1$ and $CLD[j]$ to point second child c_1. For each interval $[c_i..c_{i+1}-1]$, $CLD[c_i]$ recursively points its next sibling c_{i+1} and $CLD[c_{i+1}-1]$ points its second child. The exception is the first interval $[i..c_1-1]$ and the last interval $[c_k..j]$. The first interval has only child pointer $CLD[c_1-1]$, and the last interval keeps child pointer at $CLD[c_k-1]$. Because the interval $[i..j]$ occupies only entries of *CLD* at i, j, and $c_1-1, c_1, \ldots, c_k-1, c_k$, we can recursively avoid conflicts to sub-intervals in the child table.

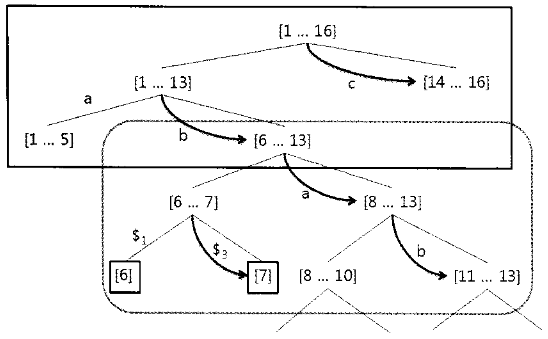
The binary trees representation [2] is a refinement to find a child in logarithmic time. For an interval $[i..j]$ and sub-interval indices c_i's, we choose a middle index c_m. We store c_m at $CLD[i]$ if $[i..j]$ is a right-interval, or store at $CLD[j]$ if $[i..j]$ is a left-interval. Then, the sub-interval $[i..c_m-1]$ is a left-interval of $[i..j]$, so $CLD[c_m-1]$ points a new middle index from c_1 to c_{m-1}. The right sub-interval $[c_m..j]$ make $CLD[c_m]$ to keep the next middle index from c_{m+1} to c_{k-1}. Finally, all sub-intervals $[c_i..c_{i+1}-1]$'s occupies the same entries of *CLD* as the linked list representation, so there is no conflict in *CLD* [2].

Remark that the binary tree representation insert some intermediate intervals, which is an union of sub-intervals, without conflicts in table *CLD*. Given an lcp-interval and its disjoint sub-intervals, the entries used in *CLD* are fixed by the sub-intervals. Therefore, we can insert to table *CLD* any intermediate lcp-interval that is an union of sub-intervals.

Lemma 2 [1,2] *Any intermediate lcp-interval*



(a) A part of lcp-interval tree



(b) A part of modified lcp-interval tree

Fig. 5. 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.

tree with k leaf intervals occupies its own O(k) entries in table CLD.

The branching function $\chi ld([i..j], \alpha)$ is straightforward by Lemma 1 and 2. Let us consider an intermediate lcp-interval tree made from refined lcp-values (l, h) with the same Ilcp-value l. Since this tree is made from the lcp-values h's of the $(l+1)$-th Icharacter, the tree corresponds to a compacted trie representing all $(l+1)$-th Icharacters by Lemma 1. The tree has a root interval with Iprefix of length l and leaf intervals with Iprefix of length $l+1$. By Lemma 2, it is embedded in table *CLD* without conflicts to the lcp-interval tree on Ilcp-values, i.e., the Isuffix tree. There are n^2 singleton intervals, so the size of *CLD* is $O(n^2)$. Then, we follow the tree by comparing the h-th character, where h is the lcp-value given by an intermediate interval [2], so $\chi ld([i..j], \alpha)$ takes $O(|\alpha| \log |\Sigma|)$ time.

Lemma 3 *The branching function* $\chi ld([i..j], \alpha)$ *takes* $O(|\alpha| \log |\Sigma|)$ *time in space of* $O(n^2)$.

4. CONCLUSION

In this paper we proposed a 2D-LST based on 2D suffix arrays with refined lcp information. Our lcp arrays require a refined lcp-value, which is full description of the lcp between suffixes of square matrices. We showed that an lcp-interval tree is generalized to 2D suffix trees and that the traversal algorithms on the lcp-interval tree simulates traversals on 2D suffix trees. So, we provide $O(m^2 \log |\Sigma|)$ time pattern matching by top-down traversal of 2D-LST. Our results made a step toward space-efficient substitutions of complex 2D suffix trees.

REFERENCE

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," Journal of Discrete Algorithms, Vol.2, No.1, pp. 53-86, 2004.

[2] D.K. Kim, M. Kim, and H. Park, "Linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays," Algorithmica, Vol.52, No.3, pp. 350-377, 2008.

[3] E.M. McCreight, "A space-economical suffix tree construction algorithm," Journal of the ACM, Vol.23, No.2, pp. 262-272, 1976.

[4] E. Ukkonen, "On-line construction of suffix trees," Algorithmica, Vol.14, No.3, pp. 249-260, 1995.

[5] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," SIAM Journal on Computing, Vol.22, No.5, pp. 935-948, 1993.

[6] K. Sadakane, "Compressed suffix trees with full functionality," Theory of Computing Systems, Vol.41, No.4, pp. 589-607, 2007.

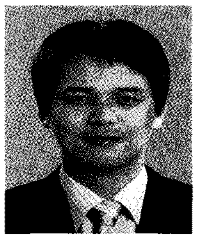
[7] L. Russo, G. Navarro, and A. Oliveira, "Fully-Compressed suffix trees," In LATIN, pp. 362-373, 2008.

[8] J. Fischer, V. Maakinen, and G. Navarro, "An(other) Entropy-Bounded compressed suffix tree," In CPM, pp. 152-165, 2008.

[9] R. Giancarlo, "A generalization of the suffix tree to square matrices, with applications," SIAM Journal on Computing, Vol.24, No.3, pp. 520-562, 1995.

[10] D.K. Kim, J.C. Na, J.S. Sim, and K. Park, "Linear-time construction of two-dimensional suffix trees," Algorithmica, To Appear.

[11] D.K. Kim, Y.A. Kim, and K. Park, "Generalizations of suffix arrays to multi-dimensional matrices," Theoretical Computer Science, Vol.302, No.1-3, pp. 223-238, 2003.



Joong Chae Na

He received a B.S., an M.S., and a Ph.D. in Computer Science and Engineering from Seoul National University in 1998, 2000, and 2005, respectively. He worked as a visiting postdoctoral researcher in the Department of Computer Science at the University of Helsinki in 2006. He is currently an assistant professor in Department of Computer Science and Engineering at Sejong University. His research interests include design and analysis of algorithms, and bioinformatics.

Sunho Lee

He received a B.S. and a Ph.D. in Computer Science and Engineering from Seoul National University in 2002 and 2009, respectively. He is currently working at Daum Communications. His research interests include design and analysis of algorithms, and data analyses.

Dong Kyue Kim

He received the BS, MS and PhD degrees in Computer Engineering from Seoul National University in 1992, 1994, and 1999, respectively. From 1999 to 2005, he was an assistant professor in the Division of Computer Science and Engineering at Pusan National University. He is currently an associate professor in the Division of Electronics and Computer Engineering at Hanyang University, Korea. His research interests are in the areas of embedded security systems, crypto-coprocessors, and information security, and hardware implementation for cryptographic devices.