정수 문자집합상의 접미사트리 구축을 위한 새로운 합병알고리즘

(A New Merging Algorithm for Constructing Suffix Trees for Integer Alphabets)

김 동 규 † 심 정 섭 †† 박 근 수 †††

(Dong Kyue Kim) (Jeong Seop Sim) (Kunsoo Park)

요 약 주어진 스트링 S의 접미사트리 T_S를 구축하기 위하여, 먼저 홀수위치들에 대한 접미사트리 T_o를 재귀적으로 구축하고, 짝수위치들에 대한 접미사트리 T_e를 T_o로 부터 구축한 다음, T_o와 T_e를 합병하여 T_S를 구축하는 새로운 방식이 사용되고 있다. 인덱스 자료구조에 관련된 문제들 중 정수 문자집합상의 접미사트리를 선형시간에 구축하는 문제는 오랫동안 미해결문제로 남아있었다. Farach은 이 방식을 적용하여 처음으로 선형시간이 소요되는 알고리즘을 제시하였다. 이 알고리즘 중 가장 어려운 곳은 합병하는 부분이다. 본 논문에서는 BFS(breadth-first search)에 기반한 새로운 합병알고리즘을 제안한다. 제안된 합병알고리즘은 Farach의 DFS(depth-first search) 방식보다 개념적으로 단순하게 동작하므로, 다른 응용으로 쉽게 확장될 수 있다.

키워드 : 접미사트리, 인덱스자료구조, 정수문자집합, 연결 BFS 합병

Abstract A new approach of constructing a suffix tree T_S for the given string S is to construct recursively a suffix tree T_o for odd positions, construct a suffix tree T_e for even positions from T_o, and then merge T_o and T_e into T_S. To construct suffix trees for integer alphabets in linear time had been a major open problem on index data structures. Farach used this approach and gave the first linear-time algorithm for integer alphabets. The hardest part of Farach's algorithm is the merging step. In this paper we present a new and simpler merging algorithm based on a coupled BFS (breadth-first search). Our merging algorithm is more intuitive than Farach's coupled DFS (depth-first search) merging, and thus it can be easily extended to other applications.

Key words : suffix trees, index data structures, integer alphabet, coupled BFS merging

1. Introduction

The suffix tree T_S of a string S is a compacted trie that represents all suffixes of S. It was designed as a space-efficient alternative[1] to Weiner's position tree[2]. The suffix tree has been a fundamental data structure in the area of string processing algorithms. When the alphabet Σ of the given string S is of constant size, the suffix tree can be constructed in $O(n)$ time[1,3,4], where n is the length of S.

The nature of alphabets affects the construction time of suffix trees. Other than constant-sized alphabets, there are two cases: general alphabets and integer alphabets. For the case of general alphabets in which the only operations on the input are symbol comparisons, the suffix tree construction has time bound of $\Theta(n \log n)$ because the known algorithms[1,4] take $O(n \log n)$ time and there is a lower bound of $\Omega(n \log n)$ by a reduction from sorting.

† 종신회원 : 부산대학교 전자전기정보컴퓨터공학부 교수
dkkim1@pusan.ac.kr
†† 학생회원 : 서울대학교 컴퓨터공학부
jssim@theory.snu.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
kpark@theory.snu.ac.kr
논문접수 : 2001년 6월 18일
심사완료 : 2001년 12월 17일

In the question of determining the exact dependence on the alphabet size[5], most interesting is the case of integer alphabets, i.e., input symbols are integers in the range $[0, n^c]$ for a constant c. Since sorting does not provide a super-linear lower bound for integer alphabets, there has been a gap between the linear lower bound and the upper bound of $O(n \log n)$. Kosaraju and Delcher[6] claimed a linear-time algorithm for this problem, but it turned out to be faulty[7]. Farach and Muthukrishnan[8] proposed a randomized linear-time algorithm. Recently, Farach[9] gave a deterministic linear-time algorithm for constructing suffix trees for integer alphabets, which solves a major open problem on index data structures [5].

In this paper we present a new and simpler algorithm for constructing suffix trees for integer alphabets in linear time. The traditional approach in constructing suffix trees[1,3,4] is to scan the given string S either left-to-right or right-to-left and construct intermediate trees incrementally until the suffix tree T_S is completed. A new approach in recent parallel and sequential algorithms [8,9,10,11] is to construct recursively the suffix tree T_o for the set of odd positions, construct the suffix tree T_e for the set of even positions from T_o, and then merge T_o and T_e into T_S. The hardest part of this approach is the merging step and our contribution is a new merging algorithm based on a coupled breadth-first search. A subproblem arising in the merging step is the following.

Prefix decision problem: Given two substrings u and v of string S, decide whether one of u and v is a prefix of the other or not.

Farach and Muthukrishnan[8] gave a randomized constant-time solution for this problem that uses fingerprints in [12]. Farach[9] does not solve the prefix decision problem, and thus the merged tree of his algorithm is structurally different from T_S. Hence, his algorithm needs to *unmerge* some parts of the merged tree so as to obtain T_S. We present a deterministic constant-time solution for the prefix decision problem after linear-time preprocessing, and by using it we can construct a merged tree that is structurally isomorphic to the suffix tree T_S. Our solution for the prefix decision problem uses *cross suffix links* between the two trees T_o and T_e.

2. Preliminaries

The n integers of the given string S in the range $[0, n^c]$ can be mapped into integers in the range $[1, n]$ by linear-time sorting. Hence we assume that $\Sigma = \{1, 2, \ldots, n\}$. The *suffix tree* T_S is the compacted trie of all suffixes of $S\#$, where $S \in \Sigma^n$ and $\# \notin \Sigma$. Fig. 1 shows the suffix tree of a string 12221123212311#. Since # is not in the alphabet, all suffixes of S are distinct and each of them is associated with a leaf of T_S. Let S_j be the j th suffix of the given string S, and $S[i]$ be the i th symbol of S.

Fig. 1 The suffix tree of a string 12221123212311#

The label of an edge (u, v) in T_S is denoted by $label(u, v)$, which is a nonempty substring of S and it is represented by the starting and ending positions of an occurrence of the substring. For a node u in T_S, let $L(u)$ be the string which is made by concatenating all the labels of edges on the path from the root to u. The leaf node associated with S_j is denoted by l_j.

It is well known[1,2] that if there is a node u in T_S such that $L(u) = a\alpha$ for $a \in \Sigma$ and $\alpha \in \Sigma^*$, there is a node v such that $L(v) = \alpha$. Each internal node u with $L(u) = a\alpha$ has a *suffix link* $sl(u)$ pointing to the node v such that $L(v) = \alpha$, i.e., $sl(u) = v$ (if α is empty, v is the root of T_S).

These suffix links form a tree, called the *sl*-tree, rooted at the root of the suffix tree T_S. For an internal node u, $|L(u)|$ is the depth of u in the *sl*-tree by definition of suffix links.

The length of the *longest common prefix* of two strings α, β is denoted by $\mathrm{lcp}(\alpha, \beta)$. The *least common ancestor* of two nodes u, v is denoted by $\mathrm{lca}(u, v)$. Then the following property is satisfied between **lcp** and **lca**: $\mathrm{lcp}(L(u), L(v)) = |L(\mathrm{lca}(u, v))|$ for all nodes u, v in T_S. By the results of [13,14] the computation of **lca** of two nodes can be done in constant time after linear-time preprocessing on a tree.

3. Construction algorithm

Let the odd tree T_o be the suffix tree of all suffixes beginning at odd positions, and the even tree T_e be the suffix tree of all suffixes beginning at even positions. The construction algorithm consists of the following four main steps.

1. Construct the odd tree T_o recursively.
2. Construct the even tree T_e from T_o.
3. Compute *cross suffix links* between T_o and T_e.
4. Merge T_o and T_e into T_S.

The merging step is the hardest part of the algorithm and it will be explained in Section 4.

3.1 The odd tree

Construction of the odd tree T_o is based on recursion and it takes linear time besides recursion.

1. Encode the given string S into a string of a half size. We make pairs $(S[2i-1], S[2i])$ for every $1 \le i \le \lceil n/2 \rceil$. Radix-sort all the pairs in linear time, and map the pairs into integers in the range $[1, \lceil n/2 \rceil]$. If we convert the pairs in the given string into corresponding integers, we get a new string of length $\lceil n/2 \rceil$, which is denoted by S'.
2. Recursively construct the suffix tree $T_{S'}$ of S'.
3. Construct T_o from $T_{S'}$. First, we replace the indices of all leaves and the lengths of edge labels in $T_{S'}$ by those of T_o. Since two symbols in S are encoded into one symbol in S', different symbols in S' may have the same first symbol in S. This can be done in linear time since it requires local adjustments in $T_{S'}$.

3.2 The even tree

The even tree T_e is constructed from T_o in linear time. The following fact is used in this construction: If we have the lexicographically sorted order of all the suffixes and the **lcp**'s of adjacent suffixes in the sorted order, then we can construct the suffix tree in linear time and vice versa[3,4]. Construction of T_e consists of the following two steps.

1. Make a sorted order of the even suffixes. The lexicographically sorted order of the odd suffixes can be computed from T_o. An even suffix is one symbol followed by an odd suffix. We make tuples for even suffixes: the first element of a tuple is $S[2i]$ and the second element is suffix S_{2i+1}. Initially, the tuples are sorted by the second elements. Then we stably sort the tuples by the first elements.

2. Compute **lcp**'s of adjacent even suffixes. Consider two even suffixes S_{2i} and S_{2j}. If $S[2i]$ and $S[2j]$ match, **lcp** of S_{2i} and S_{2j} is $\mathrm{lcp}(S_{2i+1}, S_{2j+1})+1$. If they do not match, $\mathrm{lcp}(S_{2i}, S_{2j})$ is 0. After linear-time preprocessing on T_o, $\mathrm{lca}(l_{2i+1}, l_{2j+1})$ can be computed in constant time[13,14]. Then $\mathrm{lcp}(S_{2i+1}, S_{2j+1})$ can be computed by the property between **lcp** and **lca** mentioned in Section 2.

Fig. 2 Cross suffix links between an odd tree and an even tree

3.3 Cross suffix links

We will compute suffix links from the odd tree T_o to the even tree T_e and suffix links from T_e to T_o which will be used to solve the prefix decision problem later. Such suffix links can be defined by the following lemma. (See Fig.2.)

Lemma 1 *Let u_o and u_e be nodes (except the*

root nodes) in T_o *and* T_e, *respectively. If* $L(u_o)=a\alpha$, *then there exists a node* v_e *in* T_e *such that* $L(v_e)=a$. *If* $L(u_e)=b\beta$, *then there exists a node* v_o *in* T_o *such that* $L(v_o)=\beta$.

Definition 1 *Let* u_o *and* u_e *be nodes in* T_o *and* T_e, *respectively, such that* $L(u_o)=a\alpha$ *and* $L(u_e)=b\beta$. *The* cross suffix link $csl(u_o)$ *of* u_o *points to the node* v_e *in* T_e *such that* $L(v_e)=a$. *Similarly,* $csl(u_e)$ *points to the node* v_o *in* T_o *such that* $L(v_o)=\beta$.

We now describe how to find the cross suffix links of all nodes in T_o and T_e. First, we preprocess T_o and T_e in linear time to perform **lca** operations in constant time[13,14]. For an internal node u_o in T_o, let l_i, l_j be two leaves in the subtree rooted at u_o. The cross suffix link $\mathrm{csl}(u_o)$ is the internal node $v_e=\mathrm{lca}(l_{i+1}, l_{j+1})$ in T_e. Finding $\mathrm{csl}(u_e)$ of an internal node u_e in T_e is similar. The cross suffix link of a leaf l_i is simply l_{i+1}. The cross suffix links for all nodes in T_o and T_e can be computed in $O(n)$ time.

4. Merging odd and even trees

In this section we describe how to merge the odd tree T_o and the even tree T_e into the final suffix tree T_S in $O(n)$ time. We will construct a *structurally merged tree* (*SM*-tree) that is structurally isomorphic to T_S, and then compute T_S from the *SM*-tree.

4.1 Coupled BFS

The uncompacted trie $Trie_S$ of string S is defined as follows: every edge in $Trie_S$ is labeled by a single symbol, and the concatenated string from the root node to a leaf node in $Trie_S$ is a suffix of S associated with the leaf node. Suppose that we merge uncompacted tries $Trie_o$ and $Trie_e$. Farach and Muthukrishnan[8,9] merged two tries by a coupled DFS(depth-first search) of the two tries. That is, a coupled DFS algorithm first merges the roots of both trees, and then simultaneously takes edges in both trees and recursively merges the two subtrees. However, in a coupled DFS algorithm it is hard to solve the prefix decision problem that arises in merging T_o and T_e.

To solve the prefix decision problem in constant time, our algorithm basically follows a coupled BFS(breadth-first search) of two tries. Since T_o and T_e are compacted tries, our coupled BFS traverses not by depths of nodes but by string lengths from the root. To visit all internal nodes u by ascending order of $|L(u)|$, we maintain multiple queues. Let $Q[k]$, $1\le k\le n$, be an array of queues. Each queue $Q[k]$ has *to-be-merged pairs*, which are defined below, as its elements.

Let u be a merged node of T_o and T_e. Let v and w be children of u such that one of v and w is in T_o, the other is in T_e, and $label(u, v)$ and $label(u, w)$ begin with the same symbol. Then (u, v) and (u, w) are called a *to-be-merged* pair. A to-be-merged pair (u, v) and (u, w) will be denoted by a tuple $\langle u, v, w\rangle$. (See Fig. 3.)

Suppose that two internal nodes u_o in T_o and u_e in T_e are merged into a node u in T_S. Let o_i, $1\le i\le I$, be the i th child node of u_o and e_j, $1\le j\le J$, be the j th child node of u_e, where I and J are the numbers of children of u_o and u_e, respectively. Let a_i and b_j be the first symbols of $label(u_o, o_i)$ and $label(u_e, e_j)$, respectively. When u is created in T_S, we need to find to-be-merged pairs among the edges (u_o, o_i)'s and (u_e, e_j)'s. To achieve $O(I+J)$ time, we perform the following procedure $Pair(u_o, u_e)$ that finds to-be-merged pairs and insert the pairs into queues. Fig. 3 shows an example of Procedure $Pair(u_o, u_e)$.

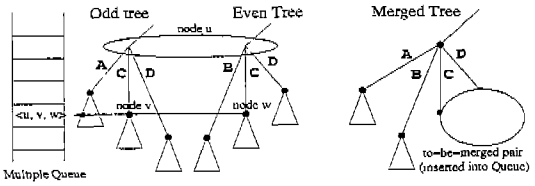

Fig. 3 An example of Procedure $Pair(u_o, u_e)$: to-be-merged pairs $\langle u, v, w\rangle$ should be inserted into the mutiple queue Q.

Procedure $Pair(u_o, u_e)$
merge u_o and u_e into u in T_S
$i \leftarrow 1$ and $j \leftarrow 1$
while $i \leq I$ and $j \leq J$ **do**
if $a_i = b_j$ **then**
$k \leftarrow \min\{|L(o_i)|, |L(e_j)|\}$
insert $\langle u, o_i, e_j \rangle$ into $Q[k]$
$i \leftarrow i+1$, $j \leftarrow j+1$
else if $a_i < b_j$ **then** $i \leftarrow i+1$
else $j \leftarrow j+1$ **fi**
od
end

We now describe our coupled BFS algorithm. The output of a coupled BFS algorithm is a structurally merged tree ($SM-tree$). Initially, we perform $Pair(r_o, r_e)$ where r_o and r_e are the root nodes of T_o and T_e, respectively. At the beginning of stage k of our coupled BFS algorithm, $Q[k]$ has every to-be-merged pair $\langle u, v, w \rangle$ such that $k = \min\{|L(v)|, |L(w)|\}$. In stage k, we extract a to-be-merged pair $\langle u, v, w \rangle$ from $Q[k]$ and do the following procedure until no elements are left in $Q[k]$. Assume without loss of generality that $k = |L(v)| \leq |L(w)|$. There are two cases depending on whether or not $label(u, v)$ is a prefix of $label(u, w)$. See Fig. 4.

Case 1. $label(u, v)$ is not a prefix of $label(u, w)$.

Since $\langle u, v, w \rangle$ is a to-be-merged pair, there exists a common prefix α of $label(u, v)$ and $label(u, w)$ such that $1 \leq |\alpha| \leq k - |L(u)|$. We make a new node u' (called a *refinement* node) and new edges (u', v) and (u', w). Since there are no more to-be-merged pairs in two subtrees rooted at v and w, we insert no tuples into queues. Notice that we do not know the length of $label(u, u')$ nor the order of $label(u', v)$ and $label(u', w)$ since the prefix decision problem simply returns 'No' in this case. We will handle these problems later in Section 4.3.

Case 2. $label(u, v)$ is a prefix of $label(u, w)$.

If $label(u, v) = label(u, w)$ then there can be new to-be-merged pairs between children of v and children of w. Thus we perform $Pair(v, w)$.

Otherwise (i.e., $|label(u, v)| < |label(u, w)|$), create a new node w' (called an *extra* node) between u and w such that $label(u, w') = label(u, v)$ and $label(w', w) = label(u, w) - label(u, v)$. (That is, the concatenation of $label(u, v)$ and $label(w', w)$ is $label(u, w)$.) Perform $Pair(v, w')$ in which there is at most one to-be-merged pair.

In Case 1 the merging at the subtrees rooted at v and w finishes, while in Case 2 the merging continues. Therefore, without solving the prefix decision problem (as in [9]) the merging process cannot produce a merged tree that is structurally isomorphic to the suffix tree T_S.

4.2 The prefix decision problem

We redefine the prefix decision problem: Given a to-be-merged pair $\langle u, v, w \rangle$, decide whether one of $label(u, v)$ and $label(u, w)$ is a prefix of the other or not. Difficulties in solving this problem are the following.

- We can use **lca** operations in the odd tree T_o or in the even tree T_e, but v and w are in different trees (so are $csl(v)$ and $csl(w)$). See Fig. 5.
- The final suffix tree T_S will have a correct structure between v and w, but we are in the process of merging and thus we cannot use **lca** operations in the merged tree.

The main idea in our solution is to find two nodes in one of T_o and T_e through cross suffix

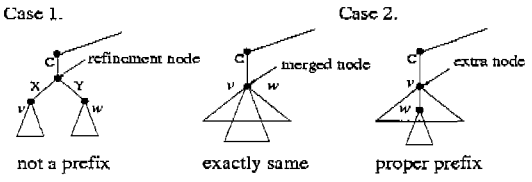

Fig. 4 Two cases when merging to-be-merged pairs.

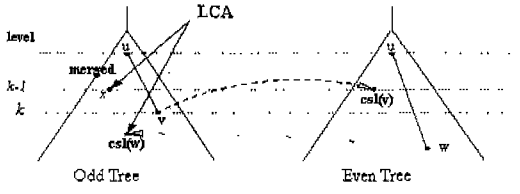

Fig. 5 A solution for the prefix decision problem.

links and the merged part in the coupled BFS such that an **lca** operation on the two nodes gives an answer to the prefix decision problem.

Assume without loss of generality that $k = |L(v)| \le |L(w)|$ and w is in T_e. Since v and w are children of merged node u, $label(u, v)$ is a prefix of $label(u, w)$ if and only if $L(v)$ is a prefix of $L(w)$. We solve the prefix decision problem in constant time as follows. We claim that $L(v)$ is a prefix of $L(w)$ if and only if $L(\mathrm{csl}(v))$ is a prefix of $L(\mathrm{csl}(w))$, which is true if the first symbols of $L(v)$ and $L(w)$ are the same. If u is the root, the first symbols of $L(v)$ and $L(w)$ are the same by definition of to-be-merged pairs (i.e., $label(u, v)$ and $label(u, w)$ begin with the same symbol). If u is not the root, the first symbols of $L(v)$ and $L(w)$ are the same because $L(u)$ is a prefix of both $L(v)$ and $L(w)$.

Since $|L(\mathrm{csl}(v))| = k-1$, $\mathrm{csl}(v)$ has been processed in our coupled BFS. There are two cases depending on whether $\mathrm{csl}(v)$ was merged with a node of T_o. See Fig. 5.

(i) $\mathrm{csl}(v)$ was not merged with a node of T_o : It means that there exists no node y in T_o such that $\mathbf{lcp}(L(\mathrm{csl}(v)), L(y)) \ge k-1$. Hence, $L(\mathrm{csl}(v))$ cannot be a prefix of $L(\mathrm{csl}(w))$.

(ii) $\mathrm{csl}(v)$ was merged with a node x of T_o : Since the node x may be an extra node, let z be the nearest descendant of x that is a node in the original T_o (z is unique because an extra node has only one child). Since we are at stage k, z is at most a grandchild of x. Since both z and $\mathrm{csl}(w)$ are nodes in T_o, we can use an **lca** operation on z and $\mathrm{csl}(w)$ in the original T_o. $L(\mathrm{csl}(v))$ is a prefix of $L(\mathrm{csl}(w))$ if and only if $|(L(\mathrm{lca}(z, \mathrm{csl}(w))))| \ge k-1$.

4.3 From *SM*-tree to suffix tree

We now construct the suffix tree T_S from the *SM*-tree that is structurally isomorphic to T_S. What remains to do is to compute $L(t)$ for every refinement node t (which has two children), and then determine the lexicographic order of the two children. This procedure is the same as that of Farach[9].

1. Construct the **sl**-tree of the *SM*-tree. We first preprocess the *SM*-tree, and then compute all the suffix links using **lca** operations. This can be done in $O(n)$ time.

2. Determine the depths of all internal nodes in the **sl**-tree by traversing the **sl**-tree. Then, we can get $|L(u)|$ for every internal node u in the *SM*-tree from the depth of u in **sl**-trees by definition. Thus, for every refinement node t, we can determine the length of $label(u, t)$, i.e., $|label(u, t)| = |L(u)| - |L(t)|$. The lexicographic order of two children c_1 and c_2 of a refinement node t is the order of the first symbols of $label(t, c_1)$ and $label(t, c_2)$.

5 Concluding Remarks

In this paper, we have presented a new and simpler merging algorithm based on a coupled BFS for constructing suffix trees for integer alphabets. Our merging algorithm is more intuitive than Farach's coupled DFS merging. Hence our algorithm can be easily extended to other construction algorithms. For example, consider the case when we apply this paradigm to a generalization of the suffix tree to square matrices. In general, the algorithms for constructing two-dimensional suffix trees are very complicated to design. However, our proposed merging algorithm could be easily applied to the linear-time construction algorithm for two-dimensional suffix trees[15] because of its simplicity.

References

[1] E.M. McCreight, A space-economical suffix tree construction algorithms, *J. ACM* 23 (1976), 262-272.

[2] P. Weiner, Linear pattern matching algorithms, *Proc. 14th IEEE Symp. Switching and Automata Theory* (1973), 1-11.

[3] M.T. Chen and J. Seiferas, Efficient and elegant subword tree construction, In A. Apostolico and Z.Galil, editors, Combinatorial Algorithms on

Words, NATO ASI Series F: Computer and System Sciences (1985).
[4] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995), 249-260.
[5] Z. Galil, Open problems in stringology, In A. Apostolico and Z. Galil, editors, Combinatorial Algorithms on Words, NATO ASI Series F: Computer and System Sciences(1985).
[6] S. Kosaraju and A. Delcher, Large-scale assembly of dna strings and space-efficient construction of suffix trees, *ACM Symp. Theory of Computing* (1995), 169-177.
[7] S. Kosaraju and A. Delcher, Large-scale assembly of dna strings and space-efficient construction of suffix trees (corrections), *ACM Symp. Theory of Computing* (1996).
[8] M. Farach and S. Muthhukrishnan, Optimal logarithmic time randomized suffix tree construction, *Int. Colloq. Automata Languages and Programming* (1996), 550-561.
[9] M. Farach, Optimal suffix tree construction with large alphabets, *IEEE Symp. Found. Computer Science* (1997), 137-143.
[10] R. Hariharan, Optimal parallel suffix tree construction, *IEEE Symp. Found. Computer Science* (1994), 290-299.
[11] S.C. Sahinalp and U. Vishkin, Symmetry breaking for suffix tree construction, *IEEE Symp Found. Computer Science.*(1994), 300-309.
[12] R.M. Karp and M.O. Rabin, Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development* 31 (1987), 249-260.
[13] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* 13(1984), 338-355.
[14] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. Comput.* 17, (1988), 1253-1262.
[15] D.K. Kim and K. Park, Linear-time construction of two-dimensional suffix trees, *Int. Colloq. Automata Languages and programming* (1999), 463-472.

김 동 규

1992년 서울대학교 컴퓨터공학과 학사. 1994년 서울대학교 컴퓨터공학과 석사. 1999년 서울대학교 컴퓨터공학과 박사. 1999년 ~ 현재 : 부산대학교 전자전기정보컴퓨터공학부 조교수. 관심분야는 컴퓨터이론, 컴퓨터보안, Bioinformatics.

심 정 섭

1995년 서울대학교 컴퓨터공학과 학사. 1997년 서울대학교 컴퓨터공학과 석사. 2002년 서울대학교 컴퓨터공학부 박사. 관심분야는 컴퓨터이론, Bioinformatics.

박 근 수

1983년 서울대학교 컴퓨터공학과 학사. 1985년 서울대학교 컴퓨터공학과 석사. 1991년 미국 Columbia University 전산학 박사. 1991년 ~ 1993년 영국 University of London, King's College 조교수. 1993년 ~ 현재 서울대학교 컴퓨터공학부 부교수. 관심분야는 컴퓨터이론, 암호학, 병렬계산.