アルゴリズム 69－4
（1999．9．2）

# アルファベットサイズの大きな木のSuffix Treeについて

渋谷 哲朗

日本アイ・ビー・エム（株） 東京基礎研究所

**概要:** 共通接尾木 (common suffix tree, CS-tree) の接尾木 (suffix tree) を構築する問題は通常の単語の接尾木を構築する問題の一般化であり、オートマトンの最小化や木のパタンマッチングなどに応用がある。共通接尾木のサイズを $n$、アルファベットのサイズを $|\Sigma|$ とした時、これまで Breslauer の $O(n \log |\Sigma|)$ 時間のアルゴリズムしか知られていなかったが、これは $|\Sigma|$ が大きいと $O(n \log n)$ かかる。本研究では、整数アルファベットに対し $O(n \log \log n)$ 時間のアルゴリズムを与える。また、sharrow k-ary tree という木に対しては同じく整数アルファベットに対し線形時間アルゴリズムを与える。また、本研究では $k$ 分木から完全平衡 $k$ 分木であるようなパタンを発見するのに有効な Bsuffix tree というものを提唱する。さらに、整数アルファベットの場合にこれを線形時間で構成するアルゴリズムを示す。

# Constructing the Suffix Tree of a Tree with a Large Alphabet

Tetsuo SHIBUYA

IBM Tokyo Research Laboratory

**Abstract:** The problem of constructing the suffix tree of a common suffix tree (CS-tree) is a generalization of the problem of constructing the suffix tree of a string. It has many applications, such as in minimizing the size of sequential transducers and in tree pattern matching. The best-known algorithm for this problem is Breslauer's $O(n \log |\Sigma|)$ time algorithm where $n$ is the size of the CS-tree and $|\Sigma|$ is the alphabet size, which requires $O(n \log n)$ time if $|\Sigma|$ is large. We improve this bound by giving an $O(n \log \log n)$ algorithm for integer alphabets. For trees called shallow $k$-ary trees, we give an optimal linear time algorithm for them. We also describe a new data structure, the Bsuffix tree, which enables efficient query for patterns of completely balanced $k$-ary trees from a $k$-ary tree or forest. We also propose an optimal $O(n)$ algorithm for constructing the Bsuffix tree for integer alphabets.

## 1 Introduction

The suffix tree of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S\$$ ($\$ \notin \Sigma$). This is a very fundamental and useful structure in combinatorial pattern matching. Weiner [13] introduced this structure and showed that it can be computed in $O(n|\Sigma|)$ time, where $|\Sigma|$ is the alphabet size. Since then, much work has been done on simplifying algorithms and improving bounds [3, 11, 12], with algorithms achieving an $O(n \log |\Sigma|)$ computing time (see also [8] for details). Recently, Farach [5] proposed a new algorithm that achieved a linear time (independent from the alphabet size) for integer alphabets.

A common suffix tree, or a CS-tree for short, is a data structure that represents a set of strings. This is also an important problem that appears in tasks such as minimizing sequential transducers of deterministic finite automata [2] and tree pattern matching [10]. Kosaraju [10] mentioned that the generalized suffix tree of all the suffixes of a set of strings represented by a CS-tree can be constructed in $O(n \log n)$ time where $n$ is the size of the CS-tree. Breslauer [2] improved this bound by giving an $O(n \log |\Sigma|)$ algorithm. Note that both of the algorithms were based on Weiner's suffix tree construction algorithm [13]. But this algorithm becomes $O(n \log n)$ when $\Sigma$ is large. In this paper, we improve their bound by giving an $O(n \log \log n)$ algorithm for integer alphabets. Shallow trees are trees such that their depths must be at most $c \log n$, where $n$ is the size of the tree and $c$ is some constant. We give an optimal $O(n)$ algorithm for trees called shallow $k$-ary trees, for constant $k$.

We also deal with a new data structure called a Bsuffix tree, which is a generalization of the suffix tree of a string. Using the suffix tree of a CS-tree, we can find a given path in a tree very efficiently. The Bsuffix tree is a data structure that enables us to query any given completely balanced $k$-ary tree pattern from a $k$-ary tree or forest very efficiently. Note that the concept of a Bsuffix tree is very similar to that of an Lsuffix tree [1, 7], which enables us to query any square submatrix of a square matrix efficiently. We will show that this data structure can be

built in $O(n)$ time for integer alphabets. Bsuffix trees have many useful features in common with ordinary suffix trees. For example, using this data structure, we can find a pattern (a completely balanced $k$-ary tree) in a text $k$-ary tree in $O(m \log m)$ time, where $m$ is the size of the pattern. Moreover, we can enumerate common completely balanced $k$-ary subtrees in a linear time. Considering that general tree pattern matching requires a $O(n \log^3 n)$ time [4], these results mean that a Bsuffix tree is a very useful data structure.

# 2 Preliminaries

## 2.1 The Suffix Tree

The suffix tree of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S\$$ ($\$ \notin \Sigma$). The tree has $n+1$ leaves and each internal node has more than one child. Each edge is labeled with a non-empty substring of $S\$$ and no two edges out of a node can have labels which start with the same character. Each node is labeled with the concatenated string of edge labels on the path from the root to the node, and each leaf has a label that is a different suffix of $S\$$. Because each edge label is represented by the first and the last indices of the corresponding substring in $S\$$, the data structure can be stored in $O(n)$ space. In this paper, we deal with only the suffix trees in which the edges going out from a node are sorted according to their labels. Notice that this property is very convenient for querying substrings.

For this powerful and useful data structure, we have the following theorems:

**Theorem 1 (Farach [5])** *The suffix tree of a string $S \in \{1, \ldots, n\}^n$ can be constructed in $O(n)$ time.*

Note that alphabet $\{1, \ldots, n\}$ is called an integer alphabet. In this paper, we will deal with only integer alphabets. Farach's suffix tree construction algorithm and our algorithms to be presented use the following theorem:

**Theorem 2 (Harel and Tarjan [9])** *For any tree with $n$ nodes, we can find the lowest common ancestor of any two nodes in a constant time after $O(n)$ preprocessing if the following values can be obtained in a constant time: bitwise AND, OR, and XOR of two binary numbers, and the positions of the leftmost and rightmost 1-bit in a binary number.*

This theorem indicates that the longest common prefix (LCP) of any two suffixes can be obtained from the suffix tree in a constant time after linear-time preprocessing.

## 2.2 The Suffix Tree of a Tree

A set of strings $\{S_1, \ldots, S_k\}$, such that no string is a suffix of another, can be represented by a common suffix tree (CS-tree for short), which is defined as follows:

**Definition 1 (CS-tree)** *In the CS-tree of a set of strings $\{S_1, \ldots, S_k\}$, each edge is labeled with a single character, and each node is labeled with the concatenated string of edge labels on the path from the node to the root. In the tree, no two edges out of a node can have the same label. Furthermore, the tree has k leaves, each of which has a different label that is one of the strings, $S_i$.*

Figure 1 shows an example of a CS-tree. The number of nodes in the CS-tree is equal to the number of different suffixes of strings. Thus, the size of a CS-tree is not larger than the sum of the lengths of the strings represented by the CS-tree. Note that the CS-tree can be constructed easily from strings in a time linear to the sum of the lengths of the strings.

The generalized suffix tree of a set of strings $\{S_1, \ldots, S_k\}$ is the compacted trie of all the suffixes of all the strings in the set. As mentioned in [10], the suffix tree of a CS-tree is the same as the generalized suffix tree of the strings represented by the CS-tree. Furthermore, the size of the generalized suffix tree is linear to that of the CS-tree, because the number of leaves of the suffix tree is equal to the number of edges in the CS-tree. Note that the edge labels of the suffix tree of a CS-tree corresponds to a path in the CS-tree, and they can be represented by the pointers to the first edge (nearest to the leaves) and the path length.

Let $n_i$ be the length of $S_i$, and let $N = \sum_i n_i$. Let $n$ be the number of nodes in the CS-tree of the strings. The generalized suffix tree can be obtained in $O(N)$ time in the case of integer alphabets (*i.e.*, $S_i \in \{1, \ldots, n\}^{n_i}$) as follows. First, we construct the suffix tree of a concatenated string of $S_1\$S_2\$\cdots\$S_k$ using Farach's suffix tree construction algorithm. Then, we obtain the generalized suffix tree by cutting away the unwanted edges and nodes. But $N$ is sometimes much larger than the size $n$ of the CS-tree: for example, there exists a tree for which $N$ is $O(n^2)$. This means that the $O(N)$-time suffix tree construction algorithm given above is not at all a linear time algorithm. The best-known $O(n \log |\Sigma|)$ algorithm [2] for this problem is based on Weiner's suffix tree construction algorithm [13]. We will improve it by giving a new algorithm based on Farach's linear-time suffix tree construction algorithm.

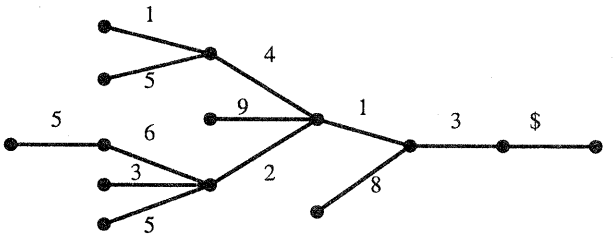



Figure 1: CS-tree of the strings 1413$, 5413$, 913$, 56213$, 3213$, 5213$, and 83$

# 3 New Algorithm for Constructing the Suffix Tree of a CS-Tree

## 3.1 Algorithm Outline

Our approach to constructing the suffix tree of a CS-tree is based on Farach's suffix tree construction algorithm [5]. Farach's algorithm has three steps. First, it constructs a tree called an odd tree recursively. Next, it constructs another tree called an even tree by using the odd tree. Finally it constructs the suffix tree by merging these two trees. Note that the odd tree is a trie of suffixes $S[2i-1]\dots S[n]\$$, and the even tree is a trie of suffixes $S[2i]\dots S[n]\$$. This algorithm achieves an $O(n)$ computation time for integer alphabets.

We later also define the odd and even trees for the suffix tree of a CS-tree, and our algorithm also has three following similar steps. First we build the odd tree or the even tree recursively, then we construct the even or odd tree by using the odd or even tree, respectively, and finally we merge them to construct the suffix tree.

In the algorithm, we use the following theorems:

**Theorem 3** *In any tree with $n$ nodes, for any node $v$ in the tree and any integer $d > 0$ that is smaller than the depth of $v$, we can find the ancestor of $v$ whose depth is $d$ in $O(\log\log n)$ time after $O(n)$ preprocessing, if the following values can be obtained in a constant time: OR, shift of any $i$ bits, the number of 1-bits in the binary number, and any $i$th-leftmost 1-bit in the binary number.*

**Theorem 4** *In any shallow $k$-ary binary tree with $n$ nodes where $k$ is constant, for any node $v$ in the tree and any integer $d > 0$ that is smaller than the depth of $v$, we can find the ancestor of $v$ whose depth is $d$ in a constant time after $O(n)$ preprocessing, if any $i$-bit shift of a binary number can be performed in a constant time.*

Proofs of these theorems are given in the appendix. See section 1 for the definition of a 'shallow tree'. Let $T_{lookup}(n)$ be the time needed to compute a node's ancestor of depth $d$ after $O(n)$ preprocessing.

Let us now define several notations. Let $\{S_1,\dots,S_k\}$ be the strings represented by a given CS-tree. Let $n_i$ be the length of $S_i$ and let $S_i = S_i[n_k]\dots S_i[1]$. Note that the indices are arranged in reverse order. Above theorems 3 and 4 indicates that, for any $i$ and $j$, we can access $S_i[j]$ in $T_{lookup}(n)$ time after $O(n)$ preprocessing. Let $S_i(m)$ be $S_i$'s suffix of length $m$, *i.e.*, $S_i[m]\dots S_i[1]$. Let $lcp(S,S')$ and $lcs(S,S')$ be the lengths of the longest common prefix and suffix of strings $S$ and $S'$, respectively. Let $parent_U(v)$ be the parent node of $v$ in the CS-tree $U$ if $v$ is not the root node $t$; otherwise, let it be $t$: *i.e.*, $parent_U(v_{i,j}) = v_{i,\max(0,j-1)}$. Let $label(e)$ be the label given to edge $e$ in the CS-tree. Let $T_U$ be the suffix tree of the CS-tree $U$.

## 3.2 Building a Half of the Suffix Tree Recursively

All nodes in the CS-tree $U=(V,E)$ have either odd or even label length. Let $V_{odd}$ and $V_{even}$ be the nodes with odd label lengths and those with even label lengths, respectively. If $|V_{odd}| \geq |V_{even}|$, let $V_{small} = V_{even}$ and $V_{large} = V_{odd}$; otherwise, let $V_{small} = V_{odd}$ and $V_{large} = V_{even}$. We can obtain $|V_{odd}|$ and $|V_{even}|$ in $O(n)$ time by the ordinary depth-first search on the CS-tree. Therefore, we can determine in a linear time which node set is $V_{small}$. In this subsection, we will recursively construct the compacted trie $T_{small}$ of all the labels of nodes in $V_{small}$. Note that the technique for constructing $T_{small}$ is very similar to that for constructing the odd tree in Farach's algorithm.

Consider a new CS-tree $U'=(V_{small},E_{small})$, where $E_{small} = \{(v, parent_{U'} = parent_U(parent_U(v)))|v \in V_{small}, v \neq t\}$ and the edge labels are determined as follows. Radix sort

the label pairs $pair(v) = (label((v, parent_U(v))), label((parent_U(v), parent_U(parent_U(v)))))$ for all $v \in V_{small} \backslash t$ and remove duplicates, where $label(e)$ denotes the label of an edge $e$ in the original CS-tree $U$ (let $label(t,t) = \phi \notin \{\Sigma,\}$). Let $rank(v)$ be the rank of $pair(v)$ in the sorted list, which belongs to an integer alphabet $[1, n/2]$ because the size of the new tree $U'$ is not larger than half of that of the original CS-tree $U$. Let $orig\_pair(i)$ be a label pair $pair(v)$ such that $rank(v) = i$. Let the label of an edge $(v, parent_{U'}(v)) \in E_{small}$ be $rank(v)$. Notice that all of these procedures can be performed in a linear time.

We then construct the suffix tree $T_{U'}$ of $U'$ by using our algorithm recursively. After that, we construct $T_{small}$ from $T_{U'}$ as follows. We can consider a tree $T'$ whose edge labels of $T_{U'}$ are modified to the original labels in $U$: for example, if the label of an edge in $T_{U'}$ is $ijk$, the label of the corresponding edge in $T'$ is $orig\_pair(i)$, $orig\_pair(j), orig\_pair(k)$. Notice that this modification can be performed by making only a minor modification of the edge label representation and that it takes only linear time.

We can construct $T_{small}$ from $T'$ very easily. $T'$ contains all the labels of nodes in $V_{small}$, but is not the compacted trie: the first characters of labels of outgoing edges from the same node may be the same. But the second character is different, and the edges are sorted lexicographically. Thus we can change $T'$ to $T_{small}$ by making only a minor adjustment: we merge such edges and make a node, and if all the first characters of all the labels of edges are the same, we delete the original node.

In this way, we can construct $T_{small}$ in a $T(n/2) + O(n)$ time, where $T(n)$ is the time our algorithm takes to build the suffix tree of a CS-tree of size $n$.

### 3.3 Building the Other Half of the Tree

In this section, we show how to construct the compacted trie $T_{large}$ of all the labels of nodes in $V_{large}$ from $T_{small}$ in a linear time. The technique is a slightly modified form of the second step of Farach's algorithm, which constructs the even tree from the odd tree.

If we are given an lexicographic traverse of the leaves of the compacted trie (which is called lex-ordering in [5]), and the length of the longest common prefix of adjacent leaves, we can reconstruct the trie [5, 6]. Note that it can be done in linear time in the case of constructing the suffix tree of a string. We will obtain these two parts of $T_{large}$ from $T_{small}$, and construct $T_{large}$ in the same way. But this method can obtain only the label length from the leaf or root for each node of the compacted trie. Recall that each label is represented by the first node and the label length in our case. Thus we must obtain that node from its specified depth and its some descendant leaf, which requires $T_{lookup}(n)$ time. Hence the total time required by this procedure is $O(nT_{lookup}(n))$.

Any leaf in $T_{large}$, except for those with labels of only one character, has a label consisting of a single character followed by the label of some corresponding leaf in $T_{small}$. We can obtain the lex-ordering of the labels of leaves in $T_{small}$ by an in-order traverse of $T_{small}$ which takes only a linear time. Thus we can obtain the lex-ordering of the labels of leaves $(S_i(m))$ in $T_{large}$ by using the radix sorting technique, because we have $S_i[m]$ and the lexicographically sorted list of $S_i(m-1)$.

The longest common prefix length of adjacent leaves of $T_{large}$ can also be obtained easily by using $T_{small}$. Let $S_i(m)$ and $S_j(n)$ be the labels of two adjacent leaves in $T_{large}$. If $S_i[m] \neq S_j[n]$, the longest common prefix length is 0. Otherwise, it is $1 + lcp(S_i(m-1), S_j(n-1))$ which can be obtained in a constant time from $T_{small}$ after linear-time preprocessing on $T_{small}$ (see Theorem 2). In this way, we can construct $T_{large}$ from $T_{small}$ in $O(nT_{lookup}(n))$. According to Theorems 3 and 4, it is $O(n \log\log n)$ for general CS-trees, and $O(n)$ for shallow $k$-ary CS-trees ($k$: constant).

### 3.4 Merging the Trees

Now we have two compacted tries $T_{odd}$ and $T_{even}$. In this subsection, we merge $T_{odd}$ and $T_{even}$ to construct the target suffix tree $T_U$. We call the compacted trie of odd/even-length suffixes of strings the generalized odd/even tree of the strings. The odd/even tree of a CS-tree is also the generalized odd/even tree of the strings represented by the CS-tree. Farach's algorithm merges the odd and even trees in a time linear to the sum of the sizes of odd and even trees. It can be directly applied also to our problem of merging generalized odd and even trees. Note that the merging can be done a linear time in Farach's algorithm, but requires $O(nT_{lookup}(n))$ time in our case. The outline of the algorithm is as follows.

First, we merge the even and odd trees as following by considering that one of two edge labels is a prefix of the other label if the first characters of labels of two edges are the same. Let edges $e_1 = (v, v_1)$ and $e_2 = (v, v_2)$ be the edges which starts from the same node $v$ and the same first

character. Let $l_1$ and $l_2$ be the label lengths of $e_1$ and $e_2$, respectively. Without loss of generality, we let $l_1 \geq l_2$. Then we construct a internal node $v_1'$ between $v$ and $v_1$ if $l_1 > l_2$, otherwise let $v_1'$ be $v_1$. In case that $l_1 > l_2$, let the label of edge $(v, v_1')$ be the first $l_2$ characters of the label of original edge $(v, v_1)$ and let the label of edge $(v_1', v_1)$ be the last $l_1 - l_2$ characters of the label of original edge $(v, v_1)$. Then we merge two edge $(v, v_1')$ and $e_2$. Note that this merging requires $T_{lookup}(n)$ time because we must find the node which corresponds to the first character of new edge $(v_1', v_1)$. We merge recursively all over the two trees by the normal coupled depth first search. Thus the total computing time required for the merging is $O(nT_{lookup}(n))$.

Next, we unmerge the edges with different labels because we have merged edges too far. In this unmerging stage, we first compute the longest common prefix length of any merged pair of node labels in the suffix tree. Farach [5] showed that all the required longest common prefix lengths for all the merged pairs can be obtained in $O(n)$ time by using a data structure called d-links; for details of d-links and the algorithm, see [5]. Using the common prefix length of merged nodes, we can easily determine how far to unmerge edges. For each unmerged edge, we must find the node of the CS-tree that corresponds to the first character of its label, which requires $T_{lookup}(n)$ time. Thus the total computing time for unmerging is also $O(nT_{lookup}(n))$.

Hence the step of our algorithm for merging trees takes a total of $O(nT_{lookup}(n))$ time. Thus we obtain an equation $T(n) = T(n/2) + O(nT_{lookup}(n))$, where $T(n)$ is the time needed to construct the suffix tree of a CS-tree of size $n$. Therefore, our algorithm achieves a $T(n) = O(n \log\log n)$ computing time for general CS-trees, and an optimal $T(n) = O(n)$ computing time for shallow $k$-ary CS-trees.

# 4 The BSuffix Tree

In this section, we propose a new data structure, the Bsuffix tree, which enables efficient queries of completely balanced binary trees from any binary forest (including a single tree). It can also be used for querying completely balanced $k$-ary subtrees from any $k$-ary forest ($k$ need not be constant in this case), but we will deal with binary trees at first. The Bsuffix tree is a data structure for matching of nodes, but it can be also used for matching of edges (see subsection 4.3).

## 4.1 Definition of the BSuffix Tree

Consider a completely balanced binary tree $P$ of height $h$. Let $p_1, p_2, \ldots, p_{2^h-1}$ be the nodes of $P$ in breadth-first order, and let $c_i \in \{1, \ldots, n\}$ be the alphabet given for node $p_i$. Note that $p_{\lfloor i/2 \rfloor}$ is the parent of $p_i$ in this order. We call $c_1 c_2 \cdots c_{2^h-1}$ the label of $P$. We call substring $c_{2^i} \cdots c_{2^{i+1}-1}$ of this label a Bcharacter. Furthermore, we call a string of Bcharacters a Bstring. For Bstring $b_1 b_2 \ldots b_n$, we call $b_1 b_2 \ldots b_m (m < n)$ a Bprefix of the Bstring. Note that $c_1 c_2 \cdots c_{2^h-1}$ is a Bstring of length $h$. For two Bcharacters $b_1$ and $b_2$, we let $b_1 > b_2$ if $b_1$ is lexicographically larger than $b_2$ in the normal string representation. Note that Bcharacter $b = c_{2^i} \cdots 2_{2^{i+1}-1}$ can be represented by node $p_{2^i} \in P$ and integer $i$.

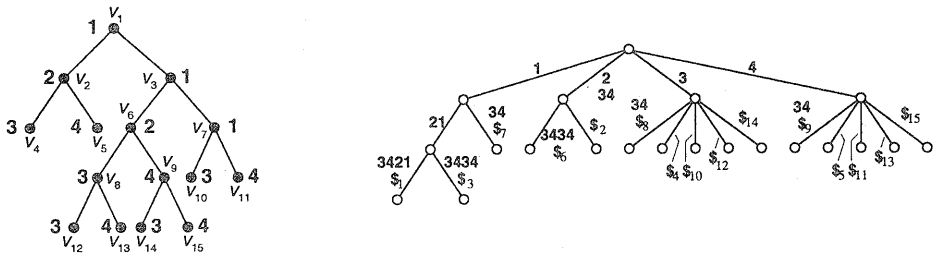
Consider a binary forest $U$ of size $n$ whose nodes are labeled with a character of an integer alphabet $\{1, \ldots, n\}$. Let $v_1, v_2, \ldots, v_n$ be the concatenated list of the breadth-first-ordered node lists of all the binary trees in forest $U$, and let $a_i \in \{1, \ldots, n\}$ be the label of node $v_i$. Let $L_i$ be the the label of the largest completely balanced binary subtree of $U$ whose root is node $v_i$. We call $L_i$ followed by $\$_i \notin \{1, \ldots, n\}$ ($\$_i \neq \$_j$) the label of node $v_i$. If the roots of two completely balanced binary subtrees $P_1$ and $P_2$ of $U$ are the same node and $P_1$ includes $P_2$, the label of $P_2$ is a Bprefix of the label of $P_1$. The Bsuffix tree of $U$ is the compacted trie $T$ of the labels of all the nodes in $U$ in the Bstring sense, *i.e.*, the outgoing edges from some node in the suffix tree have a label of different Bcharacter. Figure 2 shows an example of a Bsuffix tree. By using $T$, we can very easily query any completely balanced binary subtree of $U$.

Edge labels of $T$ can be represented by the first node in $T$ and the depths of the first and the last nodes in the pattern. Therefore $T$ can be stored in $O(n)$ space. Note that we can access any member of the edge label of $T$ in a constant time if we have both the breadth-first list and the depth-first list of the nodes of each tree in forest $U$.

## 4.2 Construction of the BSuffix Tree

In this subsection, we describe the $O(n)$ algorithm for constructing the Bsuffix tree $T$ of $U$.

If forest $U$ consists of only nodes with less than two children, it is obvious that we can construct the Bsuffix tree of $U$ in $O(n)$ time. Otherwise, we first construct a new binary forest $U'$ as follows: For every node $v_i$ with two children $v_j, v_{j+1}$, construct a node of $U'$ (let it be $w_i$). If $v_j$ and/or $v_{j+1}$ have two children, let $w_i$ be the



(1) Binary tree $U$ (2) Bsuffix tree for $U$

Figure 2: An example of the Bsuffix tree.

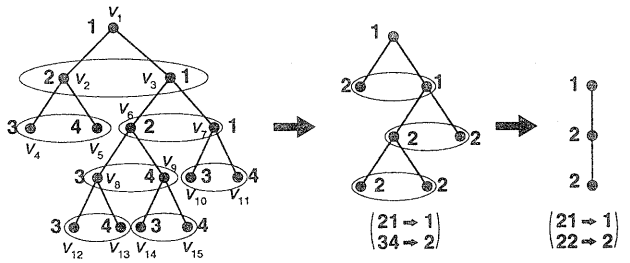


Figure 3: Recursive construction of new binary trees in computing Bsuffix tree

parent of $w_j$ and/or $w_{j+1}$ in forest $U'$. Radix sort the label pairs $(a_j, a_{j+1})$ and remove duplicates. Let the label $a'_i$ of $w_i$ be the rank of the label pair $(a_i, a_{i+1})$ in the sorted list. Notice that the number of nodes in $U'$ is not larger than $n/2$. We construct the Bsuffix tree $T'$ of $U'$ by using our algorithm recursively. Figure 3 shows an example of this recursive construction of new binary forests (trees in this case). Next, we construct $T$ from $T'$.

If we are given the lexicographically sorted list of all the node labels of $U$ and the length (*i.e.*, number of Bcharacters) of the longest common Bprefix of adjacent Bstring labels in this list, we can construct Bsuffix tree $T$ in a linear time. We obtain these two pieces of information from $T'$.

Notice that the in-order traverse of leaves of $T'$ is also a lexicographically sorted list of all the first-character-deleted labels of nodes that have two children in $T$. Thus we can obtain the lexicographically sorted list of all the node labels of $U$ by radix sorting the concatenated list of the in-order traverse of leaves of $T'$ and the labels of nodes with no or only one child.

The longest common Bprefix length $l$ of adjacent labels can also be obtained from $T'$. If the first characters of two adjacent labels are different, $l = 0$. Otherwise, if one of the adjacent labels consists of only one character, the depth is $l = 1$. Otherwise, we compute the depth as follows. Let $v_i$ and $v_j$ be the adjacent nodes. Notice that we can obtain the longest common Bprefix length $l'$ of labels of $w_i$ and $w_j$ in $U'$ in a constant time (see Theorem 2). Then it is clear that $l = l' + 1$.

In this way we can construct $T$ from $T'$ in a linear time. We obtain $T(n) = T(n/2) + O(n)$, where $T(n)$ denotes the time taken to compute the Bsuffix tree of a binary tree of size $n$. Therefore we conclude that our algorithm runs in $O(n)$ time.

### 4.3 Discussions on the Bsuffix Tree

Bsuffix trees are very similar to normal suffix trees. It enables $O(m \log m)$ query for a completely balanced binary tree pattern of size $m$. It can also be used for finding (largest) common completely balanced binary subtrees of two binary trees in linear time. We can also enumerate frequent patters of completely balanced binary trees in linear time by using this data structure.

The data structure and our algorithm assume that the labels are given to nodes, but they can very easily be modified to deal with edge-matching problems as follows: Let the label of

any node except for the root be the label of the incoming edge from its parent. Then $T'$ in the above algorithm can be used as the compacted trie for edge matching.

Bsuffix trees can also be used for querying completely balanced $k$-ary trees from any $k$-ary forest $U$. First, if a node has less than $k$ children, remove the edges between it and its children. Otherwise, we reconstruct each node that has $k$ children as a completely balanced binary tree of depth $\lceil \log_2 k \rceil$ and move each child to its leaf. For each inside node and leaf to which no node was mapped, give as its label a new character that is not in use. Notice that the size of the reconstructed forest is at most twice as that of the original one. Then construct the Bsuffix tree for this reconstructed binary tree. It can obviously used for querying completely balanced $k$-ary trees.

## 5 Concluding Remarks

We have described an $O(n \log \log n)$ algorithm for constructing the suffix tree of a common suffix tree (CS-tree). For trees called shallow $k$-ary trees, we also described an $O(n)$ algorithm. In addition, we proposed a new data structure called a Bsuffix tree, that enables efficient query for completely balanced subtrees.

The existence of a linear time algorithm for constructing the suffix tree of any trees for large alphabets remains as an open question, as does the existence of more useful suffix trees that allow querying more general and flexible patterns than paths or completely balanced trees.

## References


[1] A. Apostolico and Z. Galil, eds., "Pattern Matching Algorithms," *Oxford University Press, New York,* 1997.

[2] D. Breslauer, "The Suffix Tree of a Tree and Minimizing Sequential Transducers," *J. Theoretical Computer Science, Vol. 191,* 1998, pp. 131-144.

[3] M. T. Chen and J. Seiferas, "Efficient and Elegant Subword Tree Construction," A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words, Chapter 12,* NATO ASI Series F: Computer and System Sciences, 1985, pp. 97-107.

[4] R. Cole, R. Hariharan and P. Indyk, "Tree Pattern Matching and Subset Matching in Deterministic $O(n \log^3 n)$-time," *Proc. 4th Symposium on Discrete Mathematics (SODA '99),* 1999, pp. 245-254.

[5] M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," *Proc. 38th IEEE Symp. Foundations of Computer Science (FOCS '97),* 1997, pp. 137-143.

[6] M. Farach and S. Muthukrishnan, "Optimal Logarithmic Time Randomized Suffix Tree Construction," *Proc. 23rd International Colloquium on Automata Languages and Programming (ICALP '96),* 1996, pp. 550-561.

[7] R. Giancarlo, "The Suffix Tree of a Square Matrix, with Applications," *Proc. 4th Symposium on Discrete Mathematics (SODA '93),* 1993, pp. 402-411.

[8] D. Gusfield, "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology," *Cambridge University Press,* 1997.

[9] D. Harel and R. R. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM J. Computing, Vol. 13,* 1984, pp. 338-355.

[10] S. R. Kosaraju, "Efficient Tree Pattern Matching," *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS '89),* 1989, pp. 178-183.

[11] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM, Vol. 23,* 1976, pp. 262-272.

[12] E. Ukkonen, "On-Line Construction of Suffix-Trees," *Algorithmica, Vol. 14,* 1995, pp. 249-60.

[13] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. 14th Symposium on Switching and Automata Theory,* 1973, pp. 1-11.

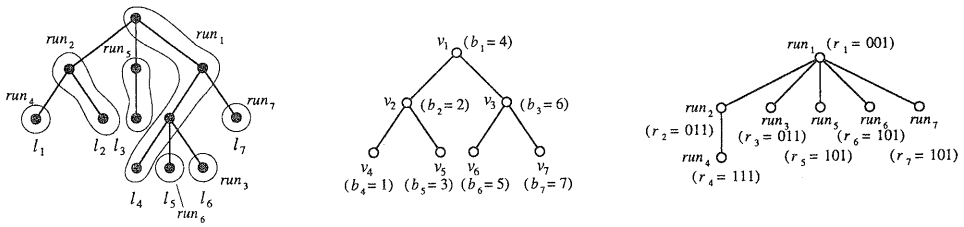

## Appendix: Proofs of Theorems 3 and 4

### Proof of Theorem 3

We achieve an $O(\log \log n)$ computing time by means of the following algorithm.

Let $m$ be the number of leaves in the target tree $T$. We call a path from some node to some leaf a 'run'. We first divide all nodes in the tree into $m$ runs. as follows:

1. Index the leaves by in-order traversing of $T$, and let them be $l_1, l_2, \ldots, l_m$.
2. Consider a completely balanced binary tree $B$ of height $\lceil \log_2(m+1) \rceil$. Let $v_1, v_2, \ldots, v_{m'}$ be the nodes of $B$ in the breadth-first order, where $m \leq m' = 2^{\lceil \log_2(m+1) \rceil} - 1 < 2m$. Map the leaves $l_1, l_2, \ldots, l_m$ to the in-order traverse of the nodes in $B$. Let $l_{b_i}$ be the leaf of $T$ to be mapped to $v_i$ in $B$.




(1) $T$ (2) $B$ (3) $R$

Figure 4: Example of $T$, $B$, and $R$.

3. For $i = 1, 2, \ldots, m'$, construct runs $run_1$, $run_2, \ldots, run_{m'}$ (note that $m' - m$ of these are empty runs) as follows:
   - If $l_{b_i}$ does not exist, let $run_i = \phi$ (an empty run) and continue.
   - Find the maximum run $run_i$ that does not contain any node of runs $\{run_j | j < i\}$ and ends at leaf $l_{b_i}$. Note that unless $run_i$ starts from the root of $T$, the parent node of the first node (nearest to the root) of $run_i$ must belongs to some run $run_{p_i}$. We call $run_{p_i}$ the parent run of $run_i$. Note that we can construct a tree $R$ of runs by using this parent-child relationship.
   - Let $r_1 = 1$. If $i > 1$, compute the following $r_i$: Let $r$ be a binary number with only one 1-bit that is at the same position as the leftmost 1-bit of $i$, *i.e.*, let $r = 2^{\lfloor \log_2 i \rfloor}$. Then let $r_i = r_{p_i} \vee r$, where $\vee$ denotes bitwise OR. Note that the depth of $v_i$ in $B$ is $1 + \lfloor \log_2 r_i \rfloor = 1 + \log_2 r = 1 + \lfloor \log_2 i \rfloor$.

Figure 4 shows an example of $T$, $B$, and $R$. Note that $r_i$ is displayed in a binary number in Figure 4 (3).

Note that the parent of node $v_i$ in $B$ is $v_{\lfloor i/2 \rfloor}$. Thus for any node of depth $d$ in $B$ and some integer $d'$ such that $0 < d' < d$, we can access the node's ancestor of depth $d$ in a constant time by simply right-shifting its index by $d - d'$ bits. For any run and some integer $d$, it is clear that the node of depth $d$ in the run can be accessed in a constant time if each run manages its nodes. Thus, once we find the run that contains the target ancestor, we can find it in a constant time. We now discuss how to find the run.

Consider node $w$ in $T$ and let $run_i$ be the run that contains $w$. It is obvious that any ancestor of $w$ in $T$ is contained by one of the ancestor runs of $run_i$ in $R$. Furthermore, if $run_j$ is an ancestor of $run_i$ in $R$, then $v_j$ is also an ancestor of $v_i$ in $B$. Let $run_{a_1}, run_{a_2}, \ldots, run_{a_k}$ $(a_1 = 1 < a_2 < \cdots < a_k = i)$ be the ancestor runs of $run_i$ (including itself). Note that the binary number $r_i$ contains $k$ 1-bits.

For any $j$ such that $0 < j \leq k$, we can access $run_{a_j}$ in a constant time by using the value of $r_i$ as follows: Let $r'$ be a binary number that has only one 1-bit whose position is the same as the $j$th-rightmost 1-bit of binary number $r_i$. Let $r''$ be a binary number that has only one 1-bit whose position is the same as the leftmost 1-bit of binary number $r_i$. Then $a_j = \lfloor i \cdot (r'/r'') \rfloor$ (right-shift by $\log_2 r'' - \log_2 r'$ bits). Using this constant-time access to the ancestor runs, we can search the target run in $O(\log k)$ time by checking the depths of the first nodes of ancestor runs. Hence we conclude that the time for finding the target node is $O(\log \log n)$, because $k <= 1 + \log_2 m' < 2 + \log_2 m$. Note that it is obvious that all of the data structures used above can be constructed in a total of $O(n)$ time.

### Proof of Theorem 4

This case is far simpler than that of Theorem 3. For a completely balanced binary tree, we can find the ancestor of depth $d$ in a constant time by indexing nodes in breadth-first order and shifting the bits of indices. The case of shallow binary trees is also obvious: We can consider a minimum complete balanced binary tree that contains a shallow binary tree of size $n$ as its subgraph. Its size is $O(n)$, and it can be built in $O(n)$ time. We can find the target ancestor in the new tree in a constant time.

In a general shallow $k$-ary tree where $k$ is some constant, every node can be mapped to an $O(n)$ binary shallow tree in such way that the depth of a mapped node is a constant times as the depth of the original node in the original tree. In this way, we conclude that such an ancestor can be found in a constant time after linear-time preprocessing.