Constructing the Suffix Tree of a Tree with a Large Alphabet

Tetsuo SHIBUYA[†],

SUMMARY The problem of constructing the suffix tree of a 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 optimal linear time algorithm for integer alphabets. 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.
key words: *algorithm, suffix tree, common suffix tree, integer alphabet, tree pattern matching*

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 [18] 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 [5], [15], [17], with algorithms achieving an $O(n \log |\Sigma|)$ computing time (see also [11] for details). Recently, Farach [8] 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 [4] and tree pattern matching [14]. Kosaraju [14] 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 [4] 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 [18]. But this algorithm becomes $O(n \log n)$ when Σ is large. In this paper, we improve their bound by giving an optimal $O(n)$ algorithm for integer alphabets.

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], [10], [13], 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 an $O(n \log^3 n)$ time [6], these results mean that a Bsuffix tree is a very useful data structure.

The extended abstract of this paper has appeared in [16]. Note that we improved the computation time bound for constructing the suffix tree of a tree, which was $O(n \log \log n)$ in the abstract.

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:

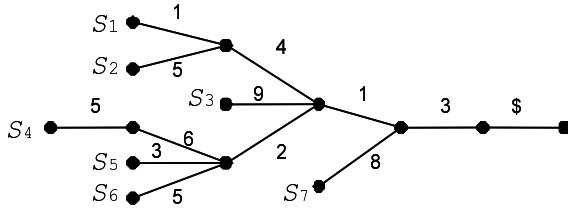**Theorem 1** (Farach [8]): The suffix tree of a string

Fig. 1 CS-tree of the strings S_1='1413\$', S_2='5413\$', S_3='913\$', S_4='56213\$', S_5='3213\$', S_6='5213\$', and S_7='83\$'

$S \in \{1, \ldots, n\}^n$ 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 [12]): 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 [14], 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 $\Theta(n^2)$. 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 [4] for this problem is based on Weiner's suffix tree construction algorithm [18]. We will improve it by giving a new algorithm based on Farach's linear-time suffix tree construction algorithm.

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 [8]. 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 $S[2i-1] \ldots S[n]\$$, and the even tree is a trie of suffixes $S[2i] \ldots S[n]\$$. 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

```
construct_suffix_tree(CS-tree U)
{
    if (U consists of only 0 or 1 node) {
        The suffix tree T of U is obvious;
        return(T);
    }
    Construct a tree U' (|U'| ≤ |U|/2);
    T_U' = construct_suffix_tree(U');
    Construct a partial suffix tree T_small using T_U';
    Construct a partial suffix tree T_large using T_small;
    Merge T_small and T_large to construct T;
    return(T);
}
```

Fig. 2 Algorithm outline for constructing the suffix tree of a CS-tree.

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. Figure 2 shows the algorithm outline in a pseudo-code.

In our algorithm, we use the following theorem by Dietz and several others [2], [3], [7]:

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 constant time after $O(n)$ preprocessing.

Let us now define several notations. Let $\{S_1, \ldots, S_k\}$ be the strings represented by a given CS-tree. Let n_i be the length of S_i and let $S_i = S_i[n_i] \ldots S_i[1]$. Note that the indices are arranged in reverse order. Above theorem 3 indicates that, for any i and j, we can access $S_i[j]$ in constant time after $O(n)$ preprocessing. Let $S_i(m)$ be S_i's suffix of length m, i.e., $S_i[m] \ldots S_i[1]$. 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., $parent_U(v_{i,j}) = v_{i,\max(0,j-1)}$, where $v_{i,j}$ denotes the ancestor of v_i whose depth is j. 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 $E_{small} = \{(v, parent_{U'} = parent_U(parent_U(v))) \mid v \in V_{small}, v \neq t\}$ 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 $v \in V_{small} \setminus t$ and remove duplicates, where $label(e)$ denotes the label of an edge e in the original CS-tree U. 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 entire 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' 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 [8]), and the length of the longest common prefix of adjacent leaves, we can reconstruct the trie [8], [9]. 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. We can obtain that node from its specified depth and its some descendant leaf in constant time according to Theorem 3. Hence the total time required by this procedure is $O(n)$.

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 processing on T_{small} (see Theorem 2). In this way, we can construct T_{large} from T_{small} in $O(n)$.

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 and we can also achieve $O(n)$ time in our case. The outline of the algorithm is as follows.

First, we merge the even and odd trees as following by treating one of two edge labels as 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 have 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 edges (v, v_1') and e_2. This merging requires only constant time because we can find the node of the CS-tree which corresponds to the first character of new edge (v_1', v_1) in a constant time. 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 also $O(n)$.

Next, we unmerge the edges with different labels because we have merged edges too far. Farach showed that we can unmerge correctly in a linear time [8] for the problem of strings, which is also the case for our problem. Consider a node u in the merged tree M, and let $v' \in T_{odd}$ and $w' \in T_{even}$ be some nodes that become u's descendants, v and w respectively, in M. Let $L(u)$ be the length of the label of u, and $\hat{L}(u)$ be the longest common prefix length of the labels of v and w. We can see that u is merged too far if $L(u) > \hat{L}(u)$. $L(u)$ can be consulted in T_{odd} or T_{even}, and $\hat{L}(u)$ can be computed for all the nodes in $O(n)$ time by the Farach's technique which uses a data structure called d-links. We call u a border node if $\hat{L}(u) < L(u)$ and $\hat{L}(p) = L(p)$ where p is the parent of u. All the border nodes can be found in a linear time. We can correct M by only

unmerging the border nodes. For each unmerged edge, we must find the node of the CS-tree that corresponds to the first character of its label, which requires only a constant time according to Theorem 3. Thus the total computing time for unmerging is $O(n)$.

Hence the step of our algorithm for merging trees takes a total of $O(n)$ time. Thus we obtain an equation $T(n) = T(n/2) + O(n)$, where $T(n)$ is the time needed to construct the suffix tree of a CS-tree of size n. Therefore, our algorithm achieves the optimal $T(n) = O(n)$ computing time for general CS-trees with integer alphabets.

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 $b = c_{2^i} \cdots c_{2^{i+1}-1}$ 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 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 Blabel 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 Blabels 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 4 shows an example of a Bsuffix tree. By using T, we can easily

query any completely balanced binary subtree of U.

Edge labels of T can be represented by the first node in U and the depths of the first and the last nodes in the corresponding subtree 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.

In a Bsuffix tree, to enable fast access to a node's outgoing edge whose first Bcharacter of its label is given, two simple preprocessing can be considered. One simple method is constructing a hash table for it, which enables linear time query in average. The data structure for it can be built in linear time. The other method is constructing a prefix tree to represent all of the first Bcharacters of edge labels, which enables deterministic $O(m \log |\Sigma|)$ query time for a query of size m, where $|\Sigma|$ denotes the size of the alphabet. This data structure can also be built in linear time.

4.2   Construction of the BSuffix Tree

In this subsection, we describe the $O(n)$ algorithm for constructing the Bsuffix tree T of U. Figure 3 shows the outline of the algorithm in a pseudo-code.

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 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 entire algorithm recursively. Figure 5 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 the Blabels of all the nodes in U and the length (*i.e.*, number of Bcharacters) of the longest common Bprefix of adjacent Blabels in this list, we can construct Bsuffix

```
construct_Bsuffix_tree(binary_forest U)
{
    if (U has no node with more than 1 child) {
        The Bsuffix tree T of U is obvious;
        return(T);
    }
    Construct a forest U' (|U'| ≤ |U|/2);
    T' = construct_Bsuffix_tree(U');
    Construct the Bsuffix tree T using T';
    return(T);
}
```

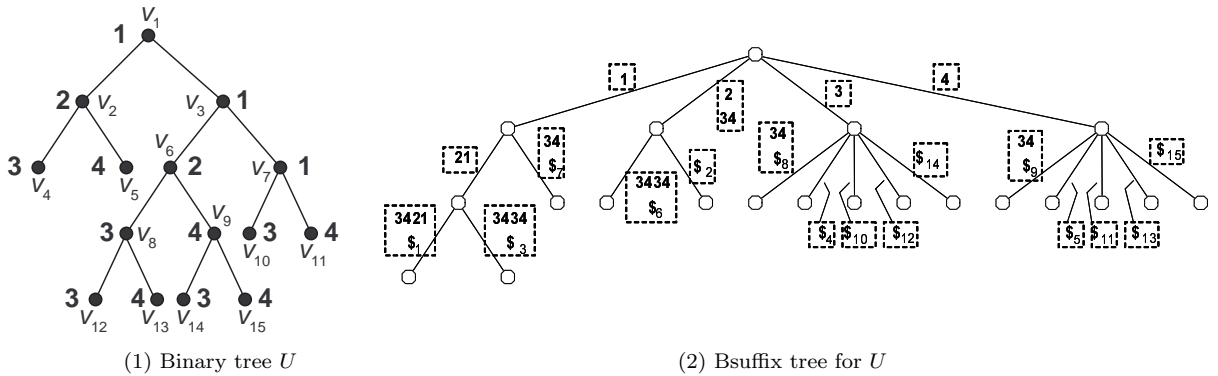**Fig. 3**   Algorithm outline for constructing the Bsuffix tree of a binary forest.

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

The longest common Bprefix length l of adjacent Blabels can also be obtained from T'. If the first characters of two adjacent Blabels are different, $l = 0$. Otherwise, if one of the adjacent Blabels 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 Blabels 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 but is same for all such nodes. 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.

(1) Binary tree U (2) Bsuffix tree for U
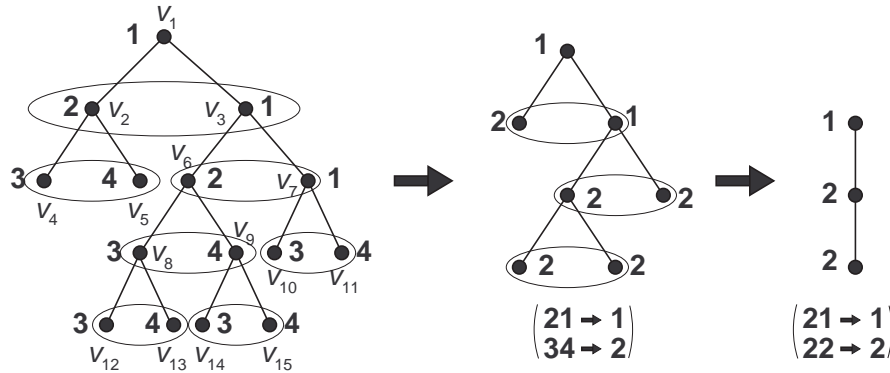
Fig. 4 An example of the Bsuffix tree.

Fig. 5 Recursive construction of new binary trees in computing Bsuffix tree

5. Concluding Remarks

We have described an optimal $O(n)$ algorithm for constructing the suffix tree of a common suffix tree (CS-tree). In addition, we proposed a new data structure called a Bsuffix tree, that enables efficient query for completely balanced subtrees. As for future work, we must seek for more useful suffix trees that allow querying more general and flexible patterns than paths or completely balanced trees. It is also an important problem to deal with dynamic trees.

References

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

[2] S. Alstrup and J. Holm, "Improved Algorithms for Finding Level Ancestors in Dynamic Trees," Proc. 27th International Colloquium on Automata, Languages, and Programming, LNCS 1853, pp. 73-84, 2000.

[3] O. Berkman and U. Vishkin, "Finding Level-Ancestors in Trees," J. Comp. Sys. Sci., Vol. 48, pp. 214-230, 1994.

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

[5] 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, pp. 97-107, 1985.

[6] 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 Algorithms, pp. 245-254, 1999.

[7] P. Dietz, "Finding level-ancestors in dynamic trees," Proc. 2nd Workshop on Algorithms and Data Structures, LNCS 1097, pp. 32-40, 1991.

[8] M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," Proc. 38th IEEE Symp. Foundations of Computer Science, pp. 137-143, 1997.

[9] M. Farach and S. Muthukrishnan, "Optimal Logarithmic Time Randomized Suffix Tree Construction," Proc. 23rd International Colloquium on Automata, Languages, and Programming, pp. 550-561, 1996.

[10] R. Giancarlo, "The Suffix Tree of a Square Matrix, with Applications," Proc. 4th Symposium on Discrete Algorithms, pp. 402-411, 1993.

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

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

[13] D. K. Kim and K. Park, "Linear Time Construction of 2-D Suffix Trees," Proc. 26th International Colloquium on Automata, Languages, and Programming, pp. 463-472, 1999.

[14] S. R. Kosaraju, "Efficient Tree Pattern Matching," Proc. 30th IEEE Symp. Foundations of Computer Science, pp. 178-183, 1989.

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

[16] T. Shibuya, "Constructing the suffix tree of a tree with a large alphabet," Proc. 10th Annual International Symposium on Algorithms and Computation, LNCS 1741, pp. 225–236, 1999.

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

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