Recognizing Unordered Depth-First Search Trees of an Undirected Graph in Parallel

Chen-Hsing Peng, *Member*, *IEEE*, Biing-Feng Wang, *Member*, *IEEE*, and Jia-Shung Wang

Abstract—Let G be an undirected graph and T be a spanning tree of G. In this paper, an efficient parallel algorithm is proposed for determining whether T is an unordered depth-first search tree of G. The proposed algorithm runs in $O(m/p + \log m)$ time using p processors on the EREW PRAM, where m is the number of edges contained in G. It is cost-optimal and achieves linear speedup.

Index Terms—Depth-first search trees, spanning trees, parallel algorithms, PRAM, the Euler-tour technique.

1 Introduction

Depth-first search, abbreviated as *DFS*, is well-known to be an important technique for designing sequential algorithms on graphs [13]. One might expect that if the DFS technique can be parallelized efficiently, a lot of sequential graph algorithms can be done as well. Unfortunately, Reif [10] proved that it seems very hard to check efficiently in parallel whether a given order of vertices is equal to the visiting sequence obtained by performing an ordered DFS on a graph, and concluded that ordered DFS is inherently sequential. By "ordered" we mean that for each vertex u in the DFS tree its children are visited in the same order as they appear on the adjacency list of u. Reif's result is pessimistic. Therefore, many researchers turned their attention to other related topics. When the ordered restriction is removed, some positive results can be derived. Hagerup [5] proposed an $O(\log n)$ time parallel unordered DFS algorithm on planar undirected graphs. Aggarwal et al. [1] proposed a randomized NC algorithm for performing unordered DFSs on general directed graphs. Opposite to the construction of a DFS tree, Schevon and Vitter [11] considered the problem of determining whether a given spanning tree of a directed graph is an unordered DFS tree of the graph. Schevon and Vitter showed that the problem can be solved in $O(\log^2 n)$ time.

In this paper, we study the problem of determining whether a given spanning tree of an undirected graph is an unordered DFS tree of the graph. We show that for an undirected graph containing n vertices and $m(\geq n-1)$ edges, its unordered DFS trees can be recognized in $O(m/p + \log m)$ time using p processors on the EREW PRAM.

The problem of verifying whether a given spanning tree satisfies some specific properties is of theoretical interest. Thus, the problem of recognizing various spanning trees had been extensively studied in literatures. For example, besides DFS trees, Manber [8] studied the problem of recognizing breadth-first search trees, Tarjan [14] and Chazelle [2] studied the problem of recognizing minimum spanning trees, and Peng et al. [9] studied the problem of recognizing shortest path trees.

An efficient algorithm for recognizing DFS trees has several applications [7], [11]. For example, in [11], it was mentioned that an efficient algorithm for recognizing DFS trees can be used as a subroutine for an algorithm that constructs a DFS tree by successively generating candidates until a valid one is obtained. In [7], Korach and Ostfeld gave two examples. Consider an undirected graph G in which no two edges have the same weight. The first example in [7] was to answer the following question: Is the unique minimum spanning tree of G obtained by performing a DFS in G? The second example, described in [7], is to solve a certain task scheduling problem. The description of the application is long and thus omitted here.

Besides being of theoretical interest, the recognition problem of DFS trees is also of practical importance. In the real world, a computation environment is not always reliable. Thus, it is necessary to verify the outputs of a DFS tree construction algorithm or to check the validness of a DFS tree inputted into a procedure.

Consider that $G = (V, E)$ is an undirected graph composed of $|V| = n$ *vertices and* $|E| = m(\geq n-1)$ edges. An *unordered depth-first search tree*, or abbreviated as *unordered DFS tree*, of G is a rooted spanning tree of G output by performing the following nondeterministic DFS algorithm.

```
Algorithm 1. (Unordered depth-first search)
Input: An undirected connected graph $G$.
Output: An unordered DFS tree $T$ of $G$.
    a. Select a vertex $r$ as the starting point.
    b. call DFS($r$).
Procedure DFS($v$)
    1. Mark $v$ as visited.
    2. for each vertex $w$ adjacent to $v$ do
    3. if $w$ is not visited then mark $(v, w)$ as an edge of $T$
       and call DFS($w$).
```

• C.H. Peng is with the Computer Center, TaiChung Veteran General Hospital, TaiChung, Taiwan 40717, Republic of China.
• B.-F. Wang and J.-S. Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan 30043, Republic of China. E-mail: bfwang@cs.nthu.edu.tw.

Manuscript received 27 June 1997; revised 26 June 1998; accepted 19 Jan. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105298.

The starting point selected in Step **a** is treated as the root of the output DFS tree. Since Steps **a** and **2** are nondeterministic, there may be more than one unordered DFS tree. To *recognize an unordered DFS tree* is to determine whether a given spanning tree is a possible output of the above unordered depth first search algorithm, and to decide the visiting order. In fact, if T is known to be an unordered DFS tree of graph G, the visiting order can be derived by performing a preorder traversal on T using the algorithm proposed by Chen, Das, and Akl in [3].

The nondeterminism of Steps **a** and **2** makes the recognition problem more complicated. Suppose that these two steps are deterministic, i.e., a specific vertex r is designated as the root of T, and for each vertex v we traverse the adjacent vertices of v following the order of the prescribed adjacency list of it. Then, to check whether T is a DFS tree or not can be simply done in linear time using the common depth first search algorithm. Note that in case the two steps are deterministic, the obtained DFS tree is ordered. If only step **2** is nondeterministic and a specific vertex r is designated as the root of T, we can verify easily by using the famous property of DFS trees: T is a DFS tree if and only if T has no cross edge [11]. In the case that both Steps **a** and **2** are nondeterministic, a linear time sequential algorithm for recognizing unordered DFS trees was proposed by Korach and Ostfeld [7].

The recognition problem for the case of directed graphs can be defined similarly. The recognition problem on directed graphs are harder than that on undirected graphs, because an undirected graph can be easily converted into a directed one by replacing each undirected edge (v, w) with two directed edges (v, w) and (w, v), and then can be solved by using the algorithms for directed graphs. Schevon and Vitter [11] showed that the recognition of unordered DFS trees for directed graphs can be done in $O(\log^2 n)$ time using $O(n^{2.376})$ processors on a CREW PRAM. In the directed case, there is only one vertex of in-degree 0 in a directed spanning tree T, and thus the root is always designated. In this paper, we show that the recognition for undirected graphs without a designated root can be done in $O(m/p + \log m)$ parallel time using p processors on the EREW PRAM. The major technique utilized in our algorithms is the Euler-tour technique [12], [15], [16], which is well-known to be a good paradigm for designing efficient parallel algorithms on trees.

The remainder of this paper is organized as follows: In Section 2, a necessary and sufficient condition for recognizing unordered DFS trees is given. In Section 3, we present a linear time sequential recognition algorithm. In Section 4, by parallelizing the sequential algorithm proposed in Section 3, an efficient parallel solution is presented on the EREW PRAM. Finally, in Section 5, we conclude this paper.

2 A Necessary and Sufficient Condition for Recognizing DFS Trees

Let $G = (V, E)$ be an undirected graph composed of $|V| = n$ vertices and $|E| = m$ edges, and T be a spanning tree of G. The edges of T are called *tree edges*, and the edges of $E - T$ are called *nontree edges*. A *tree path* is a simple path going along only tree edges. Since the tree path connecting any two vertices v and w is unique, it can be unambiguously denoted as $treepath(v, w)$. Note that the given spanning tree T is a free tree, that is, no root is designated. When a root r is assigned for T, the spanning tree will be denoted as $T_{(r)}$ instead.

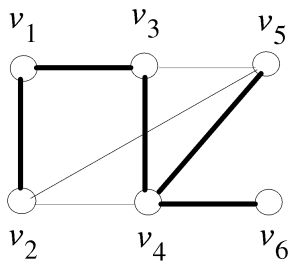


Fig. 1. A spanning tree T of a graph G (bold/fine edges: tree/nontree edges).

If a rooted spanning tree $T_{(r)}$ is given, the recognition problem will become simple. In such a case, all nontree edges can be classified into two classes: cross edges and back edges. A nontree edge (v, w) is called a *cross edge* if v and w are not ancestors of each other; otherwise, it is called a *back edge*. It is well-known that $T_{(r)}$ is a DFS tree of G if and only if there is no cross edge, or equivalently, all nontree edges are back edges [11]. For example, consider the tree T and the graph G depicted in Fig. 1. If we select v_1 as the root, $T_{(v_1)}$ is not a DFS tree, since it has two cross edges: (v_2, v_4) and (v_2, v_5). On the other hand, if we select v_2 as the root, $T_{(v_2)}$ is a DFS tree, since it has no cross edge. Thus, to recognize a DFS tree $T_{(r)}$ can be done with the following procedure.

1. Use Chen, Das, and Akl's algorithm in [3] to determine the preorder number $pre(v)$ and the postorder number $post(v)$ for every vertex $v \in T_{(r)}$ in $O(n/p + \log n)$ time using p processors on the EREW PRAM.
2. Check whether all nontree edges are back edges. Given two vertices v and w in $T_{(r)}$, to check whether v is an ancestor of w is equivalent to verifying whether $pre(v) < pre(w)$ and $post(v) > post(w)$ [6], which can be done in $O(1)$ time using a single processor. Thus, this step can be easily implemented in $O(m/p + \log m)$ time using p processors on the EREW PRAM.

The total time complexity of the above procedure is $O(m/p + \log m)$.

In our problem, the root of the given spanning tree T is not designated. Since any vertex v of T could be the root such that $T_{(v)}$ is a DFS tree, it is necessary to check for every vertex v whether all nontree edges are back edges with respect to $T_{(v)}$. Such a vertex v is called a *candidate root* of T. The tree T is a DFS tree if and only if there exists a candidate root. Thus, the main job of our recognition problem is to determine if there exists a candidate root.

We should be aware of that the terms "cross edge" and "back edge" are meaningful only for a rooted tree. A nontree edge may be a cross edge with respect to some root, but a back edge with respect to another root. For example, in Fig. 1, the nontree edge (v_2, v_5) is a cross edge with respect to $T_{(v_1)}$, but a back edge with respect to $T_{(v_2)}$.

For each nontree edge $e \in E - T$, we define $CROSS(e)$ as the set of vertices v in V such that e is a cross edge with respect to $T_{(v)}$. Similarly, for each nontree edge $e \in E - T$, we define $BACK(e)$ as the set of vertices v in V such that e is a back edge with respect to $T_{(v)}$. Note that for each nontree edge $e \in E - T$, $CROSS(e) \cap BACK(e) = \emptyset$ and $CROSS(e) \cup BACK(e) = V$.

We have the following theorem.

Theorem 1. *T is an unordered DFT tree of G if and only if $V - \bigcup_{e \in E-T} CROSS(e)$ is not an empty set.*

Proof. Suppose $V - \bigcup_{e \in E-T} CROSS(e)$ is not empty. Let s be a vertex in $V - \bigcup_{e \in E-T} CROSS(e)$. For each nontree edge $e \in E - T$, s is not in $CROSS(e)$. Therefore, by definition, all nontree edges in $E - T$ are back edges with respect to $T_{(s)}$. Consequently, $T_{(s)}$ is a DFS tree, and thus, T is a DFS tree of G. The if-part of this theorem is established.

Now, suppose that T is a DFS tree of G. Recall that T is a DFS tree of G if and only if there is a vertex $v \in V$ such that $T_{(v)}$ has no cross edge. Let s be a vertex in V such that $T_{(s)}$ has no cross edge. Since $T_{(s)}$ has no cross edge, s is not in $CROSS(e)$ for every nontree edge $e \in E - T$. Thus, s is in $V - \bigcup_{e \in E-T} CROSS(e)$ Therefore, $V - \bigcup_{e \in E-T} CROSS(e)$ is not empty. The only-if part of this theorem is established. $\square$

Theorem 1 is a necessary and sufficient condition for recognizing DFS trees. According to it, our recognition problem becomes the problem of determining whether $V - \bigcup_{e \in E-T} CROSS(e)$ is empty. In case $V - \bigcup_{e \in E-T} CROSS(e)$ is not empty, T is a DFS tree of G and each vertex v in $V - \bigcup_{e \in E-T} CROSS(e)$ is a candidate root of T.

The above discussion provides the base of our recognition algorithms. Later, in this section, we will give some properties which are helpful for determining $CROSS(e)$ for a given nontree edge e. To speedup the computation of $V - \bigcup_{e \in E-T} CROSS(e)$, we need an efficient way to represent the set V such that we can remove each set $CROSS(e)$ easily. In our algorithms, we will use an Euler tour of T to represent the set V. Using this representation, as we will show in Section 3, each $CROSS(e)$ can be removed from V by performing a constant number of weight assignments together with a prefix sum computation. Furthermore, the removal of $\bigcup_{e \in E-T} CROSS(e)$ can be done efficiently by performing $O(n)$ weight assignments and a prefix sum computation. Section 3 will define Euler tours and give a sequential implementation of the above idea in detail. Most steps of the sequential algorithm proposed in Section 3 are easily parallelized. However, a straightforward parallel implementation will result in write conflicts. Section 4 will provide some noble tricks to avoid write conflicts and describe our parallel implementation.

First, in the remainder of this section, we give some properties which are helpful for determining $CROSS(e)$ and $BACK(e)$ for any nontree edge e.

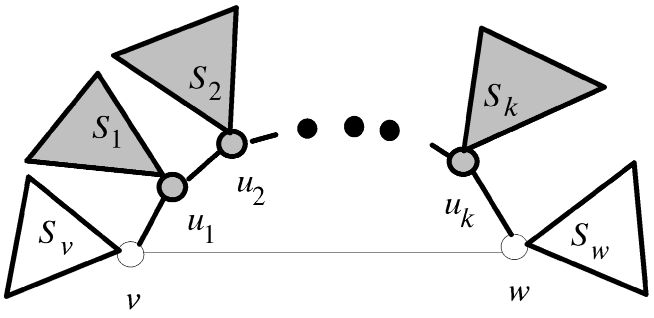


Fig. 2. The subtrees $S_v, S_1, S_2, \ldots, S_k$, and S_w for a nontree edge $e = (v, w)$. Subtrees containing vertices in $CROSS(e)$ are shadowed.

Lemma 1. *Let $e = (v, w)$ be a nontree edge in $E - T$ and $treepath(v, w) = (v, u_1, \ldots, u_k, w), k > 0$. Then, $CROSS(e) = V - (S_v \cup S_w)$ and $BACK(e) = S_v \cup S_w$, where S_v and S_w denote, respectively, the subtree of T attached to u_1 via the edge (v, u_1) and the subtree of T attached to u_k via the edge (w, u_k).*

Proof. Removing all edges in $treepath(v, w)$ from T, $k + 2$ disjoint subtrees can be obtained. Denote $S_v, S_1, S_2, \ldots, S_k$, and S_w as the obtained subtrees containing $v, u_1, u_2, \ldots, u_k$, and w, respectively, as shown in Fig. 2. Consider any vertex x in $S_i, 1 \le i \le k$. If x is the root of T, the edge (v, w) is a cross edge. Thus, all vertices in $S_i, 1 \le i \le k$, are in $CROSS(e)$. On the other hand, consider any vertex x in S_v and S_w. If x is the root of T, the edge (v, w) is a back edge. Thus, $CROSS(e) = \bigcup_{1 \le i \le k} S_i$ and $BACK(e) = S_v \cup S_w$. Since $\bigcup_{1 \le i \le k} S_i = V - (S_v \cup S_w)$, the lemma holds. $\square$

In the algorithms we shall propose in the next two sections, we will choose an arbitrary vertex $r \in V$ and then orient T into a rooted tree $T_{(r)}$. In the rooted tree $T_{(r)}$, a nontree edge $e = (v, w)$ is either a cross edge or a back edge, as shown in Fig. 3a and 3b, respectively. Suppose e is a cross edge in $T_{(r)}$, as shown in Fig. 3a. In this case, $treepath(v, w)$ is the conjunction of the tree path from v up to $lca(v, w)$ and the tree path from w up to $lca(v, w)$, where $lca(v, w)$ denotes the lowest common ancestor of v and w. By comparing Fig. 3a with Fig. 2, we obtain the following corollary from Lemma 1 immediately .

Corollary 1. *Let $e = (v, w)$ be a cross edge with respect to $T_{(r)}$. Then, $CROSS(e) = V - (X_v \cup X_w)$ and*

$$BACK(e) = X_v \cup X_w,$$

where X_v and X_w denote the two subtrees in $T_{(r)}$ rooted at v and w, respectively.

On the other hand, suppose e is a back edge in $T_{(r)}$, as shown in Fig. 3b. Without loss of generality, assume v is an ancestor of w. In this case, $treepath(v, w)$ can be traced by starting from w up to v directly. We define $child(v, w)$ as the child of v on $treepath(v, w)$. By comparing Fig. 3b with Fig. 2, we obtain the following corollary from Lemma 1 immediately.

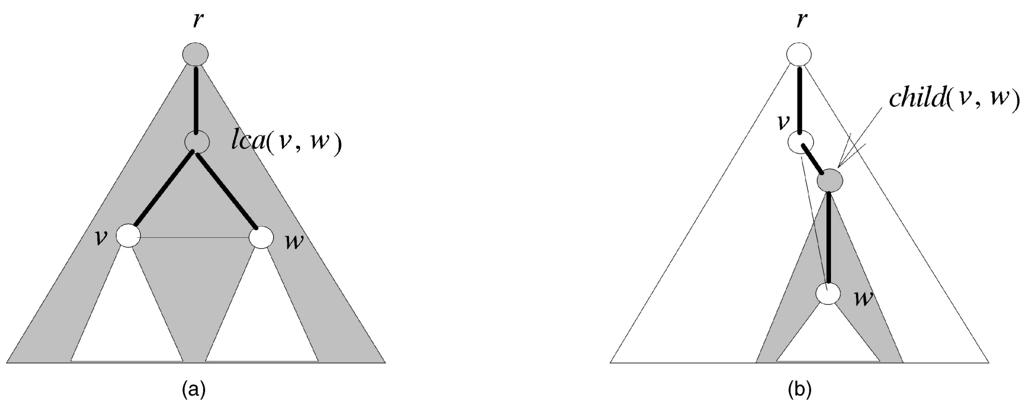


Fig. 3. Two cases for a nontree edge $e = (v, w)$ in $T_{(r)}$. Vertices in $CROSS(e)$ are shadowed. (a) e is a cross edge. (b) e is a back edge.

Corollary 2. *Let $e = (v, w)$ be a back edge with respect to $T_{(r)}$ and v be an ancestor of w. Then, $CROSS(e) = Xchild(v, w) - X_w$ and $BACK(e) = V - (Xchild(v, w) - X_w)$, where $Xchild(v, w)$ and X_w denote the two subtrees in $T_{(r)}$ rooted at $child(v, w)$ and w, respectively.*

3 A Sequential Recognition Algorithm

In this section, we shall explain the motivation why we apply the Euler-tour technique to our problem, and give a sequential algorithm to validate the correctness of our strategy. Most steps in our sequential algorithm can be easily parallelized.

Let $G = (V, E)$ be an undirected graph and T be an spanning tree of G. Based upon Theorem 1 and Corollaries 1 and 2, we can recognize whether T is a DFS tree as follows. First, we choose an arbitrary vertex r in V and orient T into a rooted tree $T_{(r)}$. Second, we compute the set $U = V - \bigcup_{e \in E-T} CROSS(e)$ by initially setting $U = V$ and then pruning away from U the vertices in $CROSS(e)$ for every nontree edge $e \in E - T$. During the computation of U, each set $CROSS(e)$ is determined by Corollaries 1 and 2. Finally, by Theorem 1, we determine whether T is a DFS tree of G by checking whether U is not empty. Clearly, the correctness of the above recognition procedure is ensured by Theorem 1 and Corollaries 1 and 2.

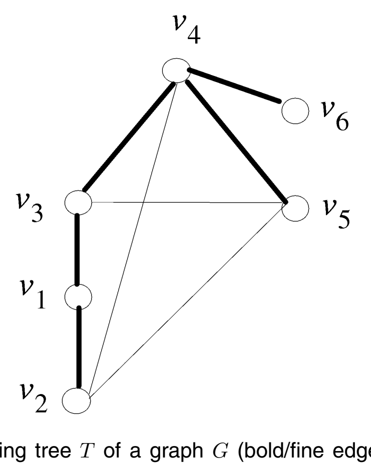


Fig. 4. A spanning tree T of a graph G (bold/fine edges: tree/nontree edges).

For example, consider the tree T and the graph G depicted in Fig. 4. Assume that v_4 is chosen as the root of T. Initially, set $U = \{v_1, v_2, \ldots, v_6\}$. There are three nontree edges, i.e., (v_2, v_4), (v_2, v_5), and (v_3, v_5). According to Corollary 2, vertices v_1 and v_3 are pruned away from U for the nontree edge (v_2, v_4), since they are in the subtree rooted at $v_3 (= child(v_4, v_2))$ but not in the subtree rooted at v_2. According to Corollary 1, vertices v_1, v_3, v_4, and v_6 are pruned away from U for the nontree edge (v_2, v_5), since they are not in the two subtrees rooted at v_2 and v_5, respectively. Similarly, vertices v_4 and v_6 are pruned away from U for the nontree edge (v_3, v_5), since they are not in the subtrees rooted at v_3 and v_5, respectively. After the pruning, only two vertices v_2 and v_5 remain in U. Thus, $U = V - \bigcup_{e \in E-T} CROSS(e) = v_2, v_5$. Since U is not empty, by Theorem 1 T is a DFS tree of G, and v_2 and v_5 are candidate roots.

It is obvious that to perform Corollaries 1 and 2 efficiently, we should be able to check whether a vertex is inside the subtree rooted at another vertex. By virtue of this, Euler tours will be very helpful.

For a rooted tree $T_{(r)}$, an *Euler tour* is a directed path starting and ending at r and traversing each tree edge forward and backward exactly once. An Euler tour for the rooted tree $T_{(v_4)}$ depicted in Fig. 4 is

$$v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_2 \rightarrow v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_4 \rightarrow v_6 \rightarrow v_4.$$

To construct such an Euler tour, we simply apply a depth-first search on the given tree $T_{(r)}$, and record the sequence of visited edges. An Euler tour is usually represented by a sequence of vertices interlaced with arrow signs.

Because of the definition of an Euler tour, the information about all subtrees of $T_{(r)}$ is stored in its Euler tour. Consider the above example again. We can see the following, for instance.

1. The entire sequence corresponds to the whole tree $T_{(v_4)}$.
2. The subtree rooted at v_3 corresponds to the subsequence enclosed by the two occurrences of v_3, which is

$$v_3 \rightarrow v_1 \rightarrow v_2 \rightarrow v_1 \rightarrow v_3.$$

In general, the subtree rooted at a vertex in $T_{(r)}$ corresponds to the subsequence enclosed by the first and the last occurrences of the vertex in the Euler tour. If a vertex is a leaf in $T_{(r)}$, then it occurs in the Euler tour exactly once and its subtree is this vertex itself.

With the above property in mind, Corollaries 1 and 2 can be easily implemented by using Euler tours.

From the above discussion, whether T is a DFS tree of G can be recognized using the following steps.

0. Select an arbitrary vertex r and orient T into a rooted tree $T_{(r)}$.
1. Construct an Euler tour U of $T_{(r)}$. Note that the set of vertices in U is equal to V.
2. For every cross edge $e = (v, w)$ with respect to $T_{(r)}$, prune away from U the vertices outside the two subsequences corresponding to the two subtrees rooted at v and w, respectively.
3. For every back edge $e = (v, w)$ with respect to $T_{(r)}$, assuming v is an ancestor of w, we first find $child(v, w)$, and then prune away from U the vertices that correspond to the subtree rooted at $child(v, w)$, except the vertices that correspond to the subtree rooted at w.
4. Output the vertices remaining in U as candidate roots.

Since there may be $O(n)$ vertices to be pruned for every nontree edge and there are $O(m)$ nontree edges, a straightforward implementation will take $O(mn)$ time. To reduce the time complexity, the vertex-pruning procedure will be implemented by a prefix sum computation along the Euler tour.

Consider that a subsequence of vertices from v_i to v_j should be pruned away. We assign "+1" to the directed edge before v_i, "-1" to the directed edge after v_j, and 0 to all other directed edges and vertices. After a prefix sum operation is performed, the prefix sums of the vertices in this subsequence are 1, and the prefix sums of other vertices are 0, which is illustrated as follows.

$$\begin{array}{ccccccccc}\cdots \rightarrow & v_{i-1} \overset{+1}{\rightarrow} & v_i & \rightarrow v_{i+1} & \rightarrow \cdots \rightarrow & v_{j-1} \rightarrow & v_j \overset{-1}{\rightarrow} & v_{j+1} & \rightarrow \cdots \\ \cdots & 0 & 1 & 1 & \cdots & 1 & 1 & 0 & \cdots . \end{array}$$

Apparently, every vertex with a prefix sum greater than 0 should be pruned away. If there are two or more subsequences to be pruned away, we can also assign "+1" and "-1", respectively, to the beginning and end of each subsequence. For instance, see the following diagram. If the subsequence of vertices from v_i to v_k should be pruned, we assign "+1" to the directed edge before v_i and "-1" to the one after v_k. If there is another subsequence of vertices from v_j to v_s to be pruned, "+1" and "-1" will be assigned to the directed edges before v_j and after v_s, respectively. For all the other directed edges and vertices, we assign "0". After a prefix sum operation is performed, the vertices with prefix sums greater than 0 are pruned away.

$$\begin{array}{ccccccccccc}\cdots\rightarrow & v_{i-1}\overset{+1}{\rightarrow} & v_i & \rightarrow\cdots\rightarrow & v_{j-1}\overset{+1}{\rightarrow} & v_j\cdots\rightarrow & v_k\overset{-1}{\rightarrow} & v_{k+1} & \rightarrow\cdots\rightarrow & v_s\overset{-1}{\rightarrow} & v_{s+1}\rightarrow\cdots \\ \cdots & 0 & 1 & \cdots & 1 & 2\cdots & 2 & 1 & \cdots & 1 & 0 \quad \cdots . \end{array}$$

We know that there may be $O(n)$ vertices to be pruned for a nontree edge. However, according to Corollaries 1 and 2, the vertices to be pruned will be contained in only two or three subsequences in an Euler tour. Since two nonzero weight assignments are sufficient for each subsequence, the total number of nonzero weight assignments is $O(m)$. The prefix sum operation is performed just one time after all weights have been assigned. Therefore, we can reduce the total time to $O(m + n)$.

In our problem, no root is designated for the given spanning tree T in advance. However, in order to apply Corollaries 1 and 2 and the Euler-tour technique, we will arbitrarily select a vertex r as the root of T. After r is selected, all nontree edges will be classified as cross edges and back edges with respect to $T_{(r)}$. Note that a nontree edge may be classified as a different type if a different root is selected. However, the final result will be the same no matter what vertex r is chosen.

Let U be an Euler tour of $T_{(r)}$. If we want to prune away from U a subsequence starting from a vertex s, we need to assign "+1" to the directed edge before s. In case that s is the first vertex in U, there is no directed edge before s. Notice that the first vertex in U is an occurrence of r. To avoid such ambiguity, in the remainder of this section, we assume that there is a *virtual directed edge* $(^* \rightarrow r)$ preceding the first vertex in U.

Our algorithm is composed of two stages. At stage one, for all nontree edges, we add weights to the corresponding directed edges on the Euler tour. According to Corollaries 1 and 2, the weights will be assigned by two different rules for the cross edges and the back edges, respectively. At stage two, we compute the prefix sums of these assigned weights along the Euler tour. Finally, all vertices associated with nonzero prefix sums are pruned away. The remaining vertices are candidate roots.

Let us consider the case of cross edges with respect to $T_{(r)}$. This is illustrated in Fig. 5. Suppose that (v, w) is a cross edge with respect to $T_{(r)}$. Recall that by Corollary 1, all vertices outside the two subtrees rooted at v and w, respectively, should be pruned away. Let $p(v)$ and $p(w)$ denote the parent vertices of v and w, respectively. We shall assign weights to the corresponding directed edges on the Euler tour by the following rule.

Rule 1. Weight assignments for a cross edge (v, w).

a. Assign -1 to the edges $(p(v) \rightarrow v)$ and $(p(w) \rightarrow w)$, i.e., the edges where the Euler tour enters into the subtrees rooted at v and w, respectively.
b. Assign $+1$ to the edges $v \rightarrow p(v))$ and $(w \rightarrow p(w))$, i.e., the edges where the Euler tour exits from the subtrees rooted at v and w, respectively.
c. Assign $+1$ to the edge $(^* \rightarrow r)$.
d. Assign 0 to all other vertices and directed edges.

Lemma 2. *Let $e = (v, w)$ be a cross edge with respect to $T_{(r)}$. Let s be a vertex in the Euler tour. After the weights are assigned according to Rule 1 and a prefix sum operation is performed, the prefix sum of s is 1 if $s \in CROSS(e)$, and is 0 otherwise.*

Proof. Let X_v and X_w denote the two subtrees of $T_{(r)}$ rooted at v and w, respectively. Let S_v and S_w denote the

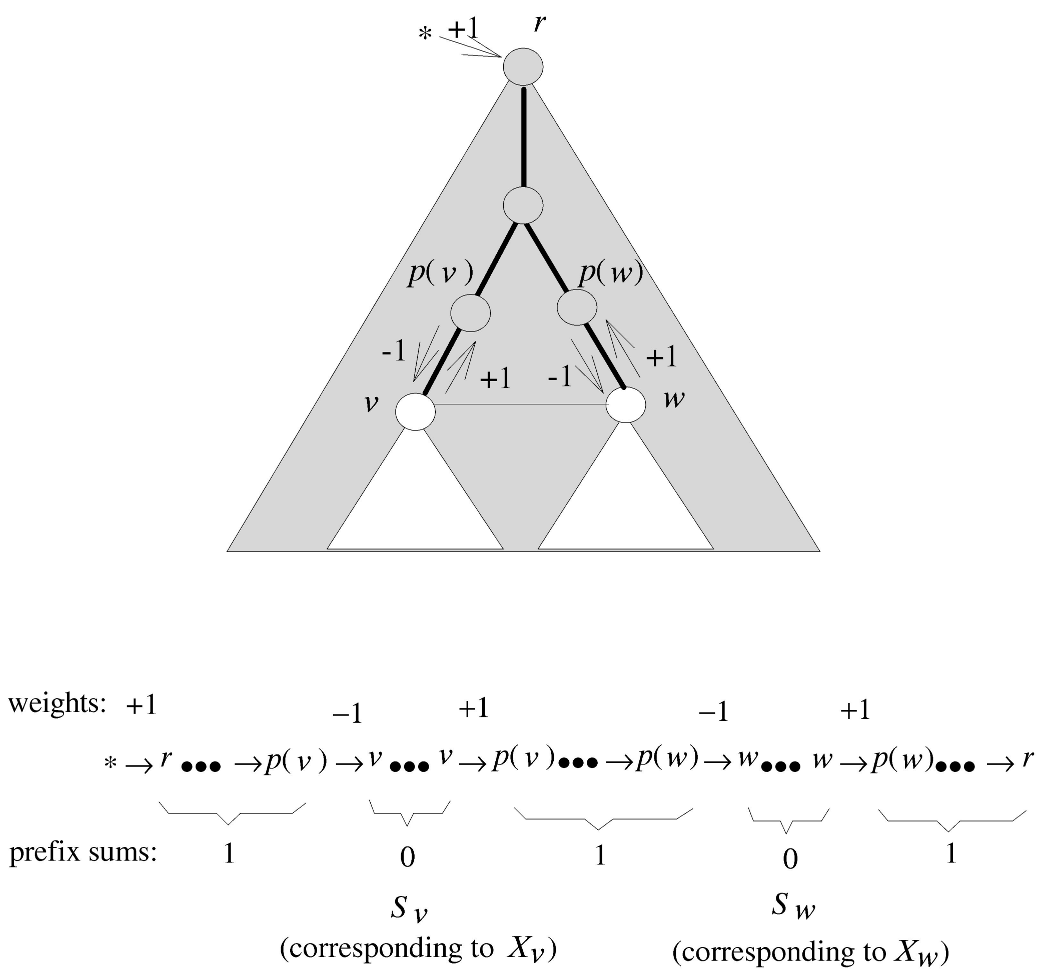


Fig. 5. The weight assignment for a cross edge (v, w).

subsequences in the Euler tour corresponding to X_v and X_w, respectively. From Fig. 5, it can be observed that after assigning weights according to Rule 1 the prefix sum of s is 0 if s is in S_v or S_w, and is 1 otherwise. The whole Euler tour, S_v, and S_w correspond to $T_{(r)}$, X_v, and X_w, respectively. Thus, we can easily conclude that after the weight assignments the prefix sum of s is 0 if $s \in (X_v \cup X_w)$, and is 1 if $s \in (V - (X_v \cup X_w))$. Since by Corollary 1 $CROSS(e) = V - (X_v \cup X_w)$, the lemma holds. □

For all back edges with respect to $T_{(r)}$, we shall assign weights properly such that a vertex is to be pruned away if its prefix sum is greater than 0. Suppose that (v, w) is a back edge with respect to $T_{(r)}$. Without loss of generality, we assume that v is an ancestor of w. This is illustrated in Fig. 6. Recall that by Corollary 2, all vertices in the subtree rooted at $child(v, w)$ but not in the subtree rooted at w should be pruned away. Let $p(w)$ denote the parent vertex of w. We shall assign weights to the corresponding directed edges on the Euler tour by the following rule.

Rule 2. Weight assignments for a back edge (v, w), where v is an ancestor of w.

a. The edge $(v \rightarrow child(v, w))$, where the Euler tour enters into the subtree rooted at $child(v, w)$, is assigned $+1$.

b. The edge $(child(v_1, w) \rightarrow v_1)$, where the Euler tour exits from the subtree rooted at $child(v_1, w)$, is assigned -1.

c. The edge $(p(w) \rightarrow w)$, where the Euler tour enters into the subtree rooted at w, is assigned -1.

d. The edge $(w \rightarrow p(w))$, where the Euler tour exits from the subtree rooted at w, is assigned $+1$.

e. Assign 0 to all other vertices and directed edges.

Lemma 3. *Let $e = (v, w)$ be a back edge with respect to $T_{(r)}$. Let s be a vertex in the Euler tour. After the weights are assigned according to Rule 2 and a prefix sum operation is performed, the prefix sum of s is 1 if $s \in CROSS(e)$, and is 0 otherwise.*

Proof. Similar to Lemma 2. □

For each nontree edge e, to prune away $CROSS(e)$, we need to apply either Rule 1 or Rule 2 once and then to perform a prefix sum operation. There are $O(m)$ nontree edges. Thus, if we perform a prefix sum operation for each of them, the resulting time complexity will be $O(mn)$. To save time, we can prune away $CROSS(e)$ for every nontree edge e by performing only one prefix sum operation as follows. First, we apply Rules 1 and 2 for all nontree edges simultaneously and accumulate the assigned weights for each directed edge and vertex in the Euler tour. Then, a prefix sum operation along the Euler tour is performed.

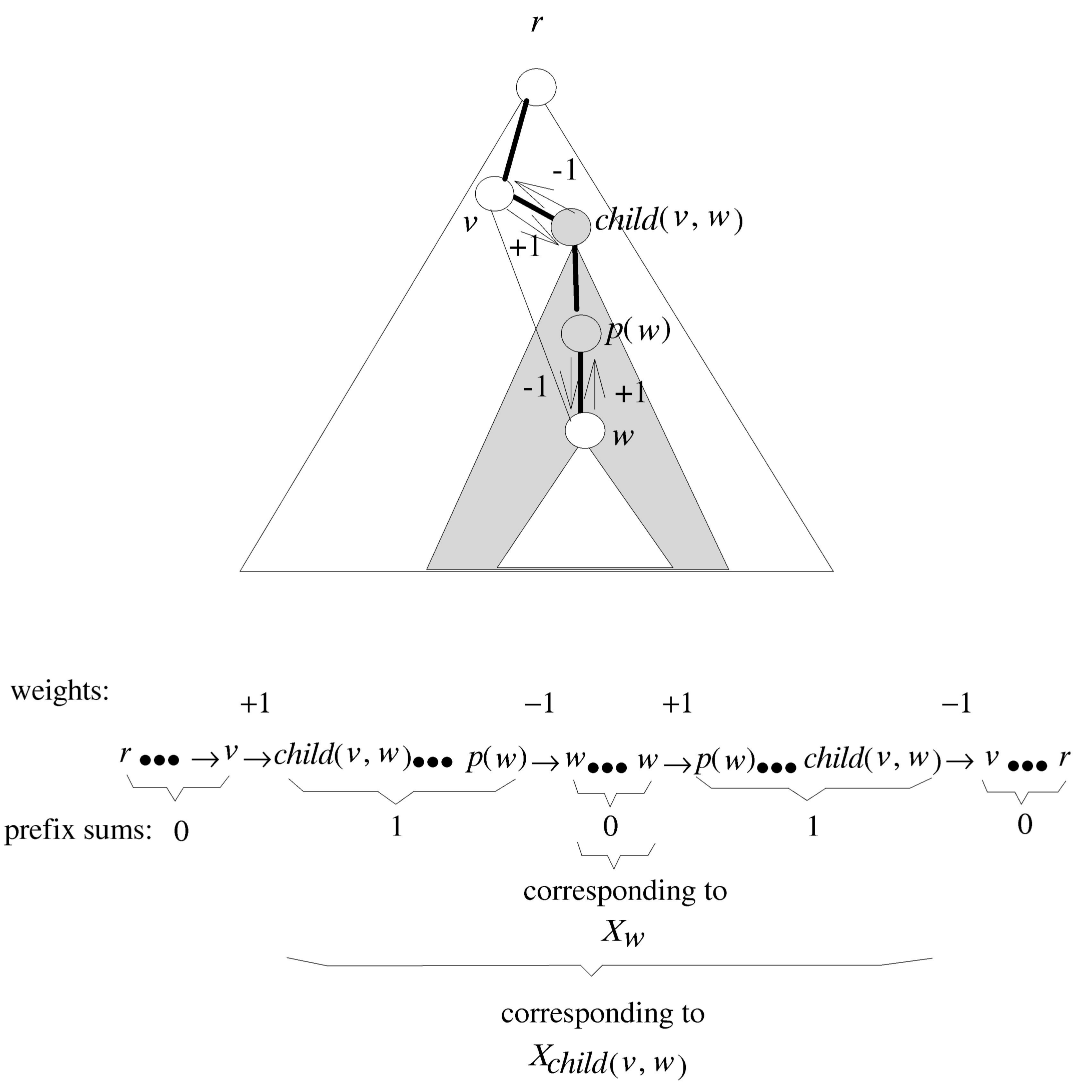

r
v
child(v, w)
p(w)
w
-1
+1
-1
+1
weights:
+1
−1
+1
−1
r ••• → v → child(v, w) ••• p(w) → w ••• w → p(w) ••• child(v, w) → v ••• r
prefix sums: 0
1
0
1
0
corresponding to X_w
corresponding to $X_{child(v,w)}$

Fig. 6. The weight assignment for a back edge (v, w).

For each vertex s, we define the *pruning level* of it, denoted by $pl(s)$, as the number of nontree edges (v, w) that are cross edges with respect to $T_{(s)}$. By definition, a vertex s is a candidate root if and only if its pruning level is 0; in other words, a vertex s should be pruned away if and only if $pl(s) > 0$. We have the following lemma.

Lemma 4. *For each vertex s in the Euler tour, its pruning level $pl(s)$ is equal to its prefix sum after applying Rules 1 and 2 for all nontree edges and then performing a prefix sum operation.*

Proof. Let s be a vertex in the Euler tour. The resulting prefix sum of s is equal to

$$\begin{aligned}
&\Sigma_{d \text{ is a directed edge before } s}\ (the\ total\ weight\ assigned\ to\ d)\\
=&\Sigma_{d \text{ is a directed edge before } s}(\Sigma_{e \text{ is a nontree edge}}\\
&\quad (the\ weight\ assigned\ to\ d\ for\ e))\\
=&\Sigma_{e \text{ is a nontree edge}}(\Sigma_{d \text{ is a directed edge before } s}\\
&\quad (the\ weight\ assigned\ to\ d\ for\ e))\\
=&\Sigma_{e \text{ is a nontree edge}}(the\ prefix\ sum\ of\ s\ after\ weight\\
&\quad assignments\ for\ e),
\end{aligned}$$

which, by Lemmas 2 and 3, is equal to

$$\begin{aligned}
&\Sigma_{e\in E-T \text{ and } s\, \in\, CROSS_{(e)}} 1 + \Sigma_{e\in E-T \text{ and } s\, \notin\, CROSS_{(e)}} 0\\
&= |\{e | e \in E - T \text{ and } e \text{ is a cross edge in } T(s)\}|.
\end{aligned}$$

The lemma holds. □

Now, we are ready to discuss our sequential recognition algorithm. At first, an Euler tour of T is built by using a depth first search on $T_{(r)}$. We use an array $W[\,]$ to store the accumulated weights for all directed edges of this Euler tour. Until all weights have been assigned, a prefix sum operation is performed. Finally, all vertices to be pruned are marked, and the remaining vertices are outputted as candidate roots. (Note that $p(v)$ denotes the parent vertex of v.)

Algorithm 2 (Recognizing depth-first search trees in sequential)

Input: An undirected graph G and a spanning tree T of G.

Output: All candidate roots, if T is an unordered DFS tree of G; otherwise, return none.

0. Arbitrarily select a vertex, say r, as the root of T.

```
1. Construct an Euler tour of $T$ by using a depth first
   search on $T_{(r)}$. Then, add a new edge $({}^*\rightarrow r)$ to the
   beginning of the Euler tour.
2. Compute the following values for each vertex $v$:
     $pre(v) =$ the preorder number of $v$,
     $post(v) =$ the postorder number of $v$, and
     $level(v) =$ the level of $v$ in $T_{(r)}$.
3. /* Stage 1 : Weight assignments for nontree edges. */
   (a) Set $W[(x \rightarrow y)]$ as 0, for all directed edges
       $(x \rightarrow y)$s of the Euler tour.
   (b) for each cross edge $(v, w)$ do
           Add $-1$ to $W[(p(v) \rightarrow v_1)]$ and $W[(p(w) \rightarrow w)]$
                       /* Rule 1a */
           Add $+1$ to $W[(v \rightarrow p(v))]$ and $W[(w \rightarrow p(w))]$
                       /* Rule 1b */
           Add $+1$ to $W[({}^*\rightarrow r)]$
                       /* Rule 1c */
       endfor
   (c) for each back edge $(v, w)$ do /* Let $v$ be an
       ancestor of $w$. */
           Add $+1$ to $W[(v \rightarrow child(v, w))]$  /* Rule 2a */
           Add -1 to $W[(child(v, w) \rightarrow v)]$  /* Rule 2b */
           Add $-1$ to $W[(p(w) \rightarrow w)]$         /* Rule 2c */
           Add $+1$ to $W[(w \rightarrow p(w))]$         /* Rule 2d */
       endfor
4. /* Stage 2 : Perform a prefix sum operation and prune
                away vertices with prefix sums larger
                than 0 */
   $PS \leftarrow 0$
   for each edge $(x \rightarrow s)$ along the Euler tour do
       $PS \leftarrow PS + W[(x \rightarrow s)]$
       if $PS > 0$ then mark $s$
       /* Note that since the total weights assigned to a
       vertex $s$ is 0, the prefix sum of $s$ is equal to that of
       $(x \rightarrow s)$.*/
   endfor
5. for each vertex $s$ do
       if $s$ is not marked then output $s$ as a candidate
       root.
   endfor
```

Note that in Step 3(c), the values of $child(v, w)$s are used. Clearly, these values can be easily determined in $O(n)$ time by scanning $T_{(r)}$ in depth-first order maintaining the path from the root r to the current vertex in a vector.

Theorem 2. *Let T be a given spanning tree of an undirected graph G. Algorithm 2 reports all candidate roots of T, if T is an unordered DFS tree of G; otherwise, no candidate root is reported.*

Proof. According to Lemma 4, Algorithm 2 indeed counts the times that each vertex is pruned for nontree edges. That is, Algorithm 2 computes the pruning level $pl(s)$ for every vertex s. According to the definition of pruning levels, a vertex s is a candidate root of T if and only if $pl(s) = 0$. Thus, a vertex s is not marked in Step 4 if and only if it is a candidate root. This ensures the correctness of Algorithm 2. The theorem holds. □

4 A Parallel Recognition Algorithm

In this section, we shall present a parallel algorithm to recognize unordered depth-first search trees. As mentioned before, our algorithm is based on the Euler-tour technique [12], [15], [16]. The *Euler-tour technique* works as follows:

1. Convert all of the tree edges in the given tree T into an Euler tour.
2. Assign proper weights to tree edges according to the problem to be solved.
3. Apply a list-ranking algorithm to evaluate the prefix sums of these weights along the Euler tour.

The Euler-tour technique [6] is a good paradigm of designing parallel algorithms for trees. The most complex step in the Euler-tour technique is to perform a list-ranking algorithm. Fortunately, Cole and Vishkin [4] showed that the list-ranking problem can be solved optimally in $O(n/p + \log n)$ time using p processors on the EREW PRAM.

Most steps in Algorithm 2 can be easily parallelized on the EREW PRAM. Recall that Chen, Das, and Akl had showed that the values of $pre(v)$, $post(v)$, and $level(v)$ can be computed in $O(n/p + \log n)$ time [3]. The only exception is the weight assignments performed in Step 3. The weights are assigned according to Rules 1 and 2. The difficulty for parallelizing Step 3 is that the concurrent execution of Rules 1 and 2 for all nontree edges may introduce write conflicts on edges of the Euler tour. In order to avoid the write conflicts, we scan the nontree edges adjacent to a vertex to determine the total weights to be assigned, and then write the total weights to the corresponding edges of the Euler tour. Under such a scenario, we rewrite the statements of Step 3 of Algorithm 2 as follows:

```
Procedure STEP3 (An alternative way for weight
    assignments.)

(a) Set $W[(x \rightarrow y)] \leftarrow 0$ for all directed edges $(x \rightarrow y)$s of
    the Euler tour.
(b) /* Weight assignments for cross edges. */
    (1) for each vertex $w \neq r$ do
            $k_w \leftarrow$ the number of cross edges adjacent to $w$.
            Add $-k_w$ to $W[(p(w) \rightarrow w)]$.     /* Rule 1a */
            Add $+k_w$ to $W[(w \rightarrow p(w))]$.     /* Rule 1b */
        endfor
    (2) $k \leftarrow$ the number of cross edges with respect to $T_{(r)}$.
        Add $+k$ to $W[({}^*\rightarrow r)]$.            /* Rule 1c */
(c) /* Weight assignments for back edges. */
    (1) for each vertex $u \neq r$ do
            $k_u \leftarrow$ the number of back edges $(p(u), w)$ such
                that $p(u)$ is an ancestor of $w$.
            Add $+k_u$ to $W[(p(u) \rightarrow u)]$      /* Rule 2a */
            Add -$k_u$ to $W[(u \rightarrow p(u))]$      /* Rule 2b */
        endfor
    (2) for each vertex $w$ do
            $k_w \leftarrow$ the number of back edges $(v, w)$ such
                that ($v$ is an ancestor of $w$. Add $-k_w$ to
                $W[(p(w) \rightarrow w)]$.               /* Rule 2c */
            Add $+k_w$ to $W[(w \rightarrow p(w))]$.     /* Rule 2d */
        endfor
```

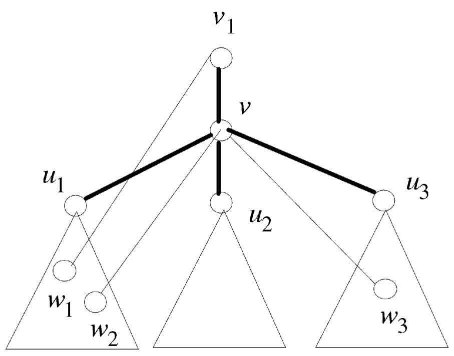


Fig. 7. An illustrative example.

We assume that for each nontree edge (v, w), both the adjacent lists of v and w have its own data about (v, w), so that all adjacent lists are disjoint and the generalized list-ranking algorithm [4] can be applied to count the cross edges or back edges adjacent to each vertex. Thus, Steps (b)(1) and (c)(2) of the above procedure can be implemented in $O(m/p + \log m)$ time. Clearly, Step (b)(2) of Procedure STEP3 takes $O(m/p + \log m)$ time. Therefore, except Step (c)(1), all steps of Procedure STEP3 can be performed efficiently in $O(m/p + \log m)$ time.

In the following, we discuss the implementation of Step (c)(1). We note that the purpose of Step (c)(1) is to assign weights for back edges according to Rules 2a and 2b.

For the convenience of usage, we define the following terms. A back edge (v, w) is said to *pass over* a tree edge $(p(u), u)$, if $(p(u), u)$ is on $treepath(v, w)$. Let $po[u]$ be the number of back edges (v, w) passing over $(p(u), u)$, and $po_1[u]$ be the number of back edges $(p(u), w)$ passing over $(p(u), u)$. Note that the back edges counted in $po_1[u]$ should connect to the parent vertex of u. Furthermore, by definition, we have that $po_1[u] \leq po[u]$.

Consider the example in Fig. 7. The back edge (v_1, w_1) passes over $(v_1, v), (v, u_1)$, and so on. There are two back edges (v_1, w_1) and (v, w_2) passing over (v, u_1), but only (v, w_2) connects to $p(u_1)$. Thus, we have $po[u_1] = 2$ and $po_1[u_1] = 1$. Similarly, we have $po[u_2] = po_1[u_2] = 0$ and $po[u_3] = po_1[u_3] = 1$.

Lemma 5. *The total weight assigned to $(p(u) \rightarrow u)$ according to Rule 2a is $+po_1[u]$; and, the total weight assigned to $(u \rightarrow p(u))$ according to Rule 2b is $-po_1[u]$.*

Proof. By the definition of $po_1[u]$, this lemma holds if we can show that for each back edge $(p(u) \rightarrow w)$ passing over $(p(u) \rightarrow u), (p(u) \rightarrow u)$ is assigned $+1$ according to Rule 2a, and $(u \rightarrow p(u))$ is assigned -1 according to Rule 2b.

Let v be the parent vertex of u, i.e., $p(u)$. For a back edge $(v \rightarrow w)$ passing over the tree edge $(v \rightarrow u)$, w must be a descendant of u and $child(v, w) = u$. Thus, $(v \rightarrow child(v_1, w)) = (p(u) \rightarrow u)$ is assigned $+1$ according to Rule 2a, and $(child(v, w) \rightarrow v) = (u \rightarrow p(u))$ is assigned -1 according to Rule 2b. □

Using Lemma 5, the weight assignments in Step (c)(1) of Procedure STEP3 can be rewritten as the follows.

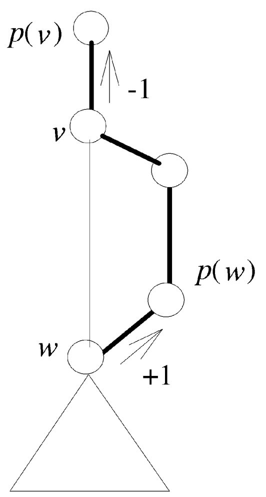


Fig. 8. Weight assignments for computing $po[u]$s.

```
(c) (1) for each vertex $u \neq r$ do
            $k_u \rightarrow po_1[u]$.
            Add $+k_u$ to $W[(p(u) \rightarrow u)]$        /* Rule 2a */
            Add $-k_u$ to $W[(u \rightarrow p(u))]$        /* Rule 2b */
        endfor
```

Now, the remaining problem is to compute $po_1[u]$ for every vertex u. Let f_v be the number of back edges (v, w) such that v is an ancestor of w. Clearly, we have the following lemma.

Lemma 6. *Let v be a vertex in $T_{(r)}$ and $u_1, u_2, \ldots, u_d$ be the child vertices of v. The value of f_v is equal to the sum of $po_1[u_i]$s, $i = 1, 2, \ldots, d$.*

In our algorithm, the value of $po_1[u]$ is estimated by $po[u]$, which can be computed much more easily than $po_1[u]$. We need the following computations.

1. Compute the values of f_vs, which are the number of back edges (v, w)s such that v is an ancestor of w.

   As mentioned above, using Cole and Vishkin's list-ranking algorithm, the number of back edges adjacent to each vertex v can be easily computed. Thus, the values of f_vs can be computed in $O(m/p + \log m)$ time.
2. Compute the values of $po[u]$s. Using the Euler-tour technique, the values of $po[u]$s can be computed in $O(m/p + \log m)$ time as follows. As shown in Fig. 8, for each back edge (v, w) such that v is an ancestor of w, we assign -1 to $(v \rightarrow p(v))$ and $+1$ to $(w \rightarrow p(w))$. And then, prefix sums of these weights along the Euler tour are computed. For any tree edge $(p(u), u)$ on $treepath(v, w)$, the weight assignments for the back edge (v, w) make the prefix sums of its corresponding directed edges $(p(u) \rightarrow u)$ and $(u \rightarrow p(u))$ differ by 1. For other tree edges, which are not on $treepath(v, w)$, the weight assignments for the back edge (v, w) do not have any effect on the difference between the prefix sums of their corresponding directed edges. Thus, the difference

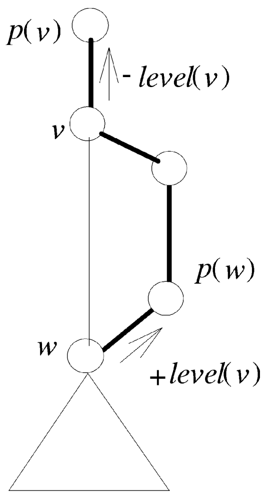


Fig. 9. Weight assignments for determining whether $po[u]$ is equal to $po_1[u]$.

between the prefix sums of $(p(u) \rightarrow u)$ and $(u \rightarrow p(u))$ are equal to the number of back edges passing over the edge $(p(u), u)$. Therefore, after the prefix computation, we can compute $po[u]$ as the difference between the prefix sums of $(p(u) \rightarrow u)$ and $(u \rightarrow p(u))$.

3. Determine whether $po[u]$ is equal to $po_1[u]$. By definition, we have $po_1[u] \leq po[u]$, and the equality holds if and only if all back edges passing over $(p(u) \rightarrow u)$ are adjacent to $p(u)$. Using the Euler-tour technique, we can determine whether $po[u] = po_1[u]$ for each vertex u in $O(m/p + \log m)$ time as follows. First, as shown in Fig. 9, we assign $-level(v)$ to $(v \rightarrow p(v))$ and $+level(v)$ to $(w \rightarrow p(w))$, for each back edge (v, w), where v is an ancestor of w. Then, prefix sums of these weights along the Euler tour are computed. And then, we compute the difference between the prefix sums of $(p(u) \rightarrow u)$ and $(u \rightarrow p(u))$ for each vertex u. Denote the difference as $differ(u)$. Recall that $po[u] = po_1[u]$ if and only if all back edges passing over $(p(u), u)$ connect to $p(u)$. Thus, according to the weight assignments, if $po[u] = po_1[u]$, we have $differ(u) = po[u] \times level(p(u))$. On the other hand, when $po_1[u] < po[u]$, there are back edges passing over $(p(u), u)$, but connecting to higher vertices, and thus, the value $differ(u)$ should be smaller than $po[u] \times level(p(u))$, since they are assigned smaller $level(v)$. Therefore, after the prefix computation, we can easily determine whether $po[u]$ is equal to $po_1[u]$ by determining whether $differ(u)$ is equal to $po[u] \times level(p(u))$.

Now, we are ready to determine the values of $po_1[u]$s. Assume that the three computations mentioned above have already been done. Consider a vertex v and its child vertices $u_1, u_2, \ldots,$ and u_d. In the following, we show how to determine the values of $po_1[u_i]$s. Let q_v be the number of u_is with $po_1[u_i] < po[u_i]$. Note that since whether $po[u_i] = po_1[u_i]$ has been determined for each i, we can compute the value of q_v easily. Three cases are to be discussed: 1) $q_v = 0$, 2) $q_v = 1$, and 3) $q_v > 1$. Suppose that case 1 is true. In this case, we have $po[u_i] = po_1[u_i]$ for all $i = 1, 2, \ldots, d$. Since the values of $po[u_i]$s have been computed, each $po_1[u_i]$ can be determined in $O(1)$ sequential time easily. Suppose that case 2 is true. Let u_k be the unique vertex with $po_1[u_k] < po[u_k]$. In this case, we can compute $po_1[u_i]$ as $po[u_i]$ for each $i \neq k$. And then, by Lemma 6, we can compute $po_1[u_k]$ as $f_v - \Sigma_{1 \leq i \leq d \text{ and } i \neq k} po_1[u_i]$. Clearly, the above computation takes $O(m/p + \log m)$ time.

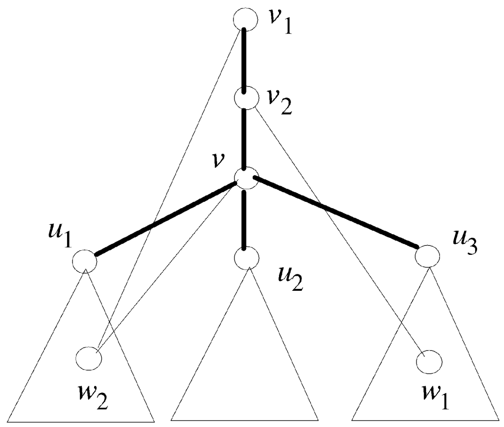


Fig. 10. All subtrees will be pruned away, if there are two or more child vertices u_is with $po_1[u_i] < po[u_i]$.

Next, let us suppose that case 3 is true. So far, for this case, the authors have not found an efficient way to determine the values of $po_1[u_i]$s. But, fortunately, as we shall show in the following, in this case ($q_v > 1$), the exact values of $po_1[u_i]$s are unnecessary in our recognition algorithm. Let u_k be a vertex with $po_1[u_k] < po[u_k]$. Since $po_1[u_k] < po[u_k]$, there must be a back edge (v', w') passing over (v, u_k), where $v' \neq v$ and v' is an ancestor of v. According to Corollary 2, all vertices in the subtrees rooted at u_is, where $1 \leq i \leq k$ and $i \neq k$, should be pruned away for the back edge (v', w'). Let u_l be another vertex with $po_1[u_l] < po[u_l], l \neq k$. Similarly, since $po_1[u_l] < po[u_l]$, all vertices in the subtrees rooted at u_is, where $1 \leq i \leq k$ and $i \neq l$, should be pruned away for some back edge (v'', w''). Consequently, in this case, we can conclude that all vertices in the subtrees rooted at u_is, $i = 1, 2, \ldots, d$, should be pruned away. For example, consider Fig. 10. In this example, since $po_1[u_1] < po[u_1]$ and $po_1[u_3] < po[u_3]$, we have $q_v = 2$. By Corollary 2, the back edge (v_1, w_2) will prune away the two subtrees rooted at u_2 and u_3, respectively, and the back edge (v_2, w_1) will prune away the two subtrees rooted at u_1 and u_2, respectively. Therefore, all the subtrees rooted at u_1, u_2, and u_3 will be pruned away for the two back edges (v_1, w_2) and (v_2, w_1).

According to Corollary 2, all the vertices pruned away for back edges (v, w) such that v is an ancestor of w are contained in the subtrees rooted at u_is. Thus, in case 3, ignoring the weight assignments for all back edges (v, w) will not produce an incorrect set of candidate roots, because all the subtrees rooted at u_is should be pruned away by other back edges.

On the basis of the above discussion, we modify Step (c) of Procedure STEP3 as follows.

```
(c) /* Weight assignments for back edges. */
    (0) for each vertex $v$ do
            $f_v \leftarrow$ the number of back edges $(v, w)$ such that
                $v$ is an ancestor of $w$
            $po[v] \leftarrow$ the number of back edges passing over
                the tree edge $(v, p(v))$
            Determine whether $po_1[v]$ is equal to $po[v]$.
            $q_v \leftarrow$ the number of $v$s children with
                $po[\,] \neq po_1[\,]$
            if $q_v > 1$ then mark all back edges $(v, w)$ such
                that $v$ is an ancestor
                    of $w$ with a tag ignored.
        end
    (1) for each vertex $u$ do
            if $q_{p(u)} \leq 1$ then
                $k_u \leftarrow po_1[u]$
                Add $+k_u$ to $W[(p(u) \rightarrow u)]$
                    /* Rule 2a */
                Add $-k_u$ to $W[(u \rightarrow p(u))]$
                    /* Rule 2b */
            endif
        endfor
    (2) for each vertex $w$ do
            $k_w \leftarrow$ the number of back edges $(v, w)$ such
                that $v$ is an ancestor of $w$ and $(v, w)$ is not
                with a tag ignored.
            Add $-k_w$ to $W[(p(w) \rightarrow w)]$.    /* Rule 2c */
            Add $+k_w$ to $W[(w \rightarrow p(w))]$.    /* Rule 2d */
        endfor
```

Note that in the above weight assignments, all back edges (r, w) will not be marked as ignored, because we have $po_1[u] = po[u]$ for every child u of r. This guarantees that neglecting the weight assignments of all the nontree edges with a tag ignored will not trap into incorrect circumstances.

The above parallel implementation of Step 3(c) of Algorithm 2 takes $O(m/p + \log m)$ time using p processors on the EREW PRAM. Recall that at the beginning of this section we showed that all the other steps of Algorithm 2 can be implemented, as well in $O(m/p + \log m)$ time using p processors on the EREW PRAM. We obtain the following theorem.

Theorem 3. *Let T be a given spanning tree of an undirected graph G that contains n vertices and m edges. We can determine whether T is an unordered depth-first search tree of G in $O(m/p + \log m)$ time using p processors on the EREW PRAM.*

5 Concluding Remarks

In this paper, a linear time sequential algorithm was first proposed for recognizing unordered depth-first search trees of an undirected graph. Then, by parallelizing it, an efficient parallel solution was obtained on the EREW PRAM. The resulting parallel algorithm performs in $O(m/p + \log m)$ time using p processors. When $p \leq m/\log m$, it is cost-optimal and achieves linear speedup.

As mentioned at the end of Section 3, our sequential algorithm can compute the pruning level of every vertex. However, since some back edges are ignored, our parallel algorithm can not compute the exact pruning levels. The problem of efficiently computing the pruning level of each vertex in parallel requires further studies.

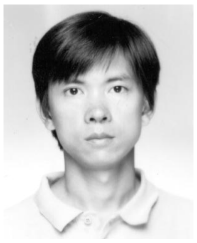
Acknowledgments

The authors express their gratitude to the anonymous referees for their valuable comments which improved the presentation of this paper a lot, and to Prof. Michael A. Palis for his patience in handling this paper. This research is supposed by the National Science Council of the Republic of China under grant NSC-84-2221-E007-002.

References

[1] A. Aggarwal, R.J. Anerson, and M.-Y. Kao, "Parallel Depth-First Search in General Directed Graphs," *SIAM J. Computing,* vol. 19, no. 2, pp. 397-409, 1990.
[2] B. Chazelle, "Computing on a Free Tree Via Complexity-Preserving Mapping," *Algorithmica,* vol. 3, pp. 337-361, 1987.
[3] C.C.-Y. Chen, S.K. Das, and S.G. Akl, "A Unified Approach to Parallel Depth-First Traversals of General Trees," *Information Processing Letters,* vol. 38, pp. 49-55, 1991.
[4] R. Cole and U. Vishkin, "Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time," *SIAM J. Computing,* vol. 17, no. 1, pp. 128-142, 1988.
[5] T. Hagerup, "Planar Depth-First Search in $O(\log n)$ Parallel Time," *SIAM J. Computing,* vol 19, no. 4, pp. 678-704, 1990.
[6] J. JaJa, *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.
[7] E. Korach and Z. Ostfeld, "DFS Tree Construction: Algorithms and Characterizations," *Proc. 14th Int'l Workshop Graph-Theoretic Concepts in Computer Science (WG-88),* 1988.
[8] U. Manber, "Recognizing Breadth-First Search Trees in Linear Time," *Information Processing Letters,* vol. 34, pp. 167-171, 1990.
[9] C.-H. Peng, J.-S. Wang, and R.C.-T. Lee, "Recognizing Shortest-Path Trees in Linear Time," *Information Processing Letters,* vol. 57, pp. 78-85, 1994.
[10] J.H. Reif, "Depth-First Search Is Inherently Sequential," *Information Processing Letters,* vol. 20, pp. 229-234, 1985.
[11] C.A. Schevon and J.S. Vitter, "A Parallel Algorithm for Recognizing Unordered Depth-First Search," *Information Processing Letters,* vol. 28, pp. 105-110, 1988.
[12] B. Schieber and U. Vishkin, "On Finding Lowest Common Ancestors: Simplification and Parallelization," *SIAM J. Computing,* vol. 17, pp. 1,253-1,262, 1988.
[13] R.E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Computing,* vol. 1, pp. 814-830, 1972.
[14] R.E. Tarjan, "Applications of Path Compression on Balanced Trees," *J. ACM,* vol. 26, pp. 690-715, 1979.
[15] R.E. Tarjan and U. Vishkin, "An Efficient Parallel Biconnectivity Algorithm," *SIAM J. Computing,* vol. 14, pp. 862-874, 1985.
[16] U. Vishkin, "On Efficient Parallel Strong Orientation," *Information Processing Letters,* vol. 20, pp. 235-240, 1988.

Chen-Hsing Peng received the BS degree in electrical engineering from National Taiwan University, Taiwan, in 1984, and the MS and PhD degree in computer science from National Tsing Hua University, Taiwan, in 1986 and 1994, respectively. Currently, he is the technical chief of the Computer and Communication Center at Taichung Veterans General Hospital, Taiwan. He is also an associate professor in the Department of Computer Science at Tunghai University, Taiwan. His research interests include parallel and distributed systems, algorithm design and analysis, and medical informatics. He is a member of the IEEE and the IEEE Computer Society.



Biing-Feng Wang received the BS degree in computer science from National Chiao Tung University, Taiwan, in 1988 and the PhD degree in computer science from National Taiwan University in 1991. In 1993, he joined the faculty of National Tsing Hua University, where he now is a professor in the Department of Computer Science. His current research interests include design and analysis of algorithms and parallel computation.

Jia-Shung Wang received the BS degree in mathematics from National Taiwan University in 1978, and the MS and PhD degrees in computer science from National Tsing Hua University in 1983 and 1986, respectively. He joined the Department of Computer Science, National Tsing Hua University, in 1986 as an associate professor and became a professor in 1995. His current research interests cover several aspects of image/video coding and transmission, VLSI design, neural networks, and parallel processing.

He received the Research Award from National Science Council of R.O.C. every year since 1987. He and the staff at his laboratory have developed several advanced systems, including a parallel processing system with 256 processors, a Java accelerator, and a video-on-demand system. Accordingly, he won the Dr. Sun-Yet-Sen Technology Award in 1989 and the Winbond H.261 Competition Award in 1996. He was also also the champion of the first Taiwan Java Competition Cup, in 1998.