SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0003 $01.00/0

FINDING ALL SPANNING TREES OF DIRECTED AND UNDIRECTED GRAPHS*

HAROLD N. GABOW† AND EUGENE W. MYERS†

Abstract. An algorithm for finding all spanning trees (arborescences) of a directed graph is presented. It uses backtracking and a method for detecting bridges based on depth-first search. The time required is $O(V+E+EN)$ and the space is $O(V+E)$, where V, E, and N represent the number of vertices, edges, and spanning trees, respectively. If the graph is undirected, the time decreases to $O(V+E+VN)$, which is optimal to within a constant factor. The previously best-known algorithm for undirected graphs requires time $O(V+E+EN)$.

Key words. spanning tree, arborescence, bridge, depth-first search

1. Introduction. The problem of finding all spanning trees of directed and undirected graphs arises in the solution of electrical networks [7, pp. 252–364]. Algorithms of varying efficiency have been proposed [4], [5], [6], [8], [9], [10], [11], [13]. For undirected graphs, the best algorithm seems to be that of Minty, Read and Tarjan. It uses $O(V+E+EN)$ time and $O(V+E)$ space, where the graph has V vertices, E edges, and N spanning trees. We refine their approach and present an algorithm that uses $O(V+E+VN)$ time and $O(V+E)$ space. In terms of worst-case asymptotic bounds, this algorithm is optimal. The algorithm also applies to directed graphs. Here it uses $O(V+E+EN)$ time and $O(V+E)$ space. A previous algorithm [11] uses exponential time per tree (in the worst case).

We first review some terms for undirected graphs, and generalize them to directed graphs. In a connected undirected graph G, a *spanning tree* is a subgraph having a unique simple path between any two vertices of G. A *bridge* is an edge e where $G-e$ is not connected. Equivalently, e is in every spanning tree of G.

In a directed graph G, a *spanning tree (rooted at r)* is a subgraph having a unique (directed) path from r to any vertex of G. If such a tree exists, G is *rooted at r*. A *bridge (for r)* is an edge e where $G-e$ is not rooted at r. Equivalently, e is in every spanning tree rooted at r. ("Spanning arborescence" is often used for "spanning tree" of a directed graph. There appears to be no standard term for what we call a "bridge" of a directed graph.)

The problem we consider is to find all spanning trees of a graph. This means that for a given graph, a list is to be printed that contains each spanning tree exactly once. Section 2 presents our results. Section 3 discusses some open problems.

2. Algorithm for all spanning trees. This section begins with an algorithm for all spanning trees rooted at a given vertex r in a directed graph. This algorithm is used to find all spanning trees, first in a directed graph and then in an undirected graph.

For all spanning trees rooted at r, the approach is to find all spanning trees containing a given subtree T rooted at r. To do this, first choose an edge e_1 directed from T to a vertex not in T; find all spanning trees containing $T \cup e_1$; then delete e_1 from the graph. Next choose an edge e_2 from T to a vertex not in T; find all spanning trees (in the modified graph) containing $T \cup e_2$; then delete e_2. To continue, repeatedly choose an edge e_i from T to a vertex not in T; find all spanning trees (in the modified graph) containing $T \cup e_i$; then delete e_i. Stop when the edge e_k that has just been processed is a bridge of the modified graph. At this point each spanning tree

* Received by the editors January 21, 1977. This work was partially supported by the National Science Foundation under Grant GJ36461.

† Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309.

containing T has been found (exactly once). For if a spanning tree does not contain any e_j, $j<k$, it must contain the bridge e_k.

This basic approach needs an efficient method for discovering when an edge e is a bridge. This can be done in a variety of ways; set merging techniques and edge exchanges are two possibilities [3]. Below we describe a method based on depth-first search.

We choose edges e so the tree T grows depth-first. More precisely, we always add the edge e to T that originates at the greatest depth possible. Now suppose all spanning trees containing $T \cup e$ have been found, and we want to check if e is a bridge. Let L be the last spanning tree found that contains $T \cup e$, and let $e=(u, v)$. Intuitively, in L, vertex v has the fewest descendents possible (among all spanning trees containing $T \cup e$). Equivalently, no edge goes from a nondescendent of v to a proper descendent of v. (This is proved below in Lemma 3.) So e is a bridge when no edge (besides e) goes from a nondescendent of v to v. This observation gives an efficient bridge test.

To grow T depth-first requires some care. The algorithm below uses F, a list of all edges directed from vertices in T to vertices not in T. F uses stack operations: to enlarge T, an edge e is popped from the front of F and added to T; new edges for $T \cup e$ are pushed onto the front of F. In addition when e is added to T, some edges are removed from F; when e is removed from T, these edges are restored in F. It is crucial that the remove and restore operations leave the order of edges unchanged in F. Otherwise, T will not grow depth-first.

Besides F, the algorithm uses lists FF. Each recursive invocation has a local FF list. It is used to reconstruct the original F list. It is managed as a stack.

The algorithm also uses data structures for T, the current tree, and L, the last spanning tree output thus far. The algorithm is given below in ALGOL-like notation.

```
                    procedure S; comment S finds all spanning trees rooted
                      at r in a directed graph G rooted at r; begin
                      procedure GROW; comment GROW finds all spanning
                        trees rooted at r containing T; begin
 1.                     if T has V vertices then begin L ← T; output (L) end
 2.                     else begin make FF an empty list, local to GROW;
 3.                       repeat
 4. new tree edge:          pop an edge e from F; let e go from T to vertex v,
                              v ∉ T;
 5.                         add e to T;
 6. update F:               push each edge (v, w), w ∉ T, onto F;
 7.                         remove each edge (w, v), w ∈ T, from F;
 8. recurse:                GROW;
 9. restore F:              pop each edge (v, w), w ∉ T, from F;
10.                         restore each edge (w, v), w ∈ T, in F;
11. delete e:               remove e from T and from G; add e to FF;
12. bridge test:            if there is an edge (w, v), where w is not a
                              descendent of v in L then b ← false else b ← true;
13.
14. reconstruct G:        until b;
                          pop each edge e from FF, push e onto F, and add e to G;
                        end end GROW;
```

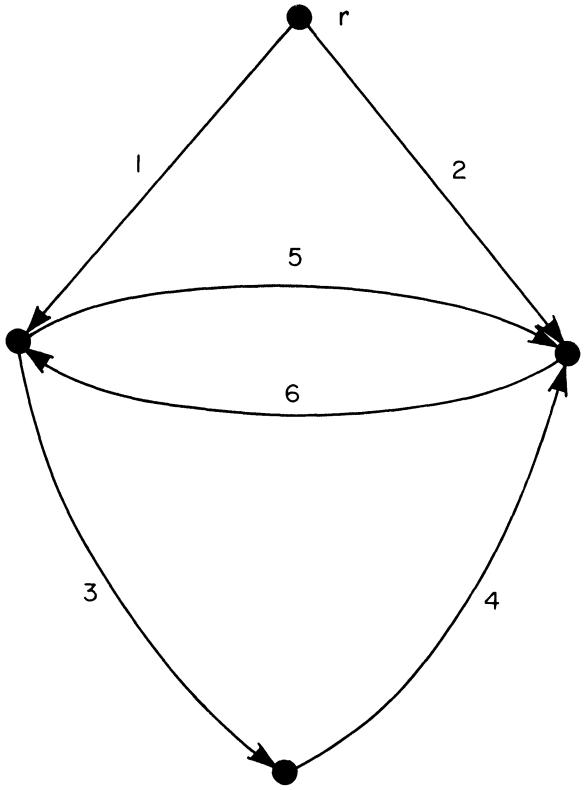

FIG. 1. *Example graph.*

15. *start*: initialize T to contain the vertex r; initialize F to contain all edges (r, v) from r;
16. GROW;
end S;

Figure 1 shows a graph with four spanning trees rooted at r. Figure 2 shows a computation tree indicating how S finds these trees T_i, $1 \leqq i \leqq 4$. In the computation tree, a node represents a call to GROW; the arcs directed down from the node correspond to the edges e added to T in line 5. For example, the root node first adds edge 1, then deletes 1 and adds 2. Since 2 is a bridge in the modified graph, no other edges are added.

Note the importance of restoring edges in correct order in line 10. When edge 4 is added to get T_1, edges 5 and 2 are removed from F. If they are restored in opposite order (i.e., 2, 5), T_3 is found, and then T_2; then edge 1 is mistakenly declared a bridge, and T_4 is not found.

Now we prove procedure S is correct. Note the original graph G is modified by GROW, by deleting and replacing edges (lines 11,14). In the discussion below the *current graph* refers to the edges in the graph at a specified point in the computation.

We first show the tree T grows depth-first. This amounts to showing F simulates the stack of active vertices in a normal depth-first search.

LEMMA 1. *Let* GROW *be called with* F *containing the sequence of edges* (v_i, w_i), $i = 1, 2, \cdots, |F|$. *Then*

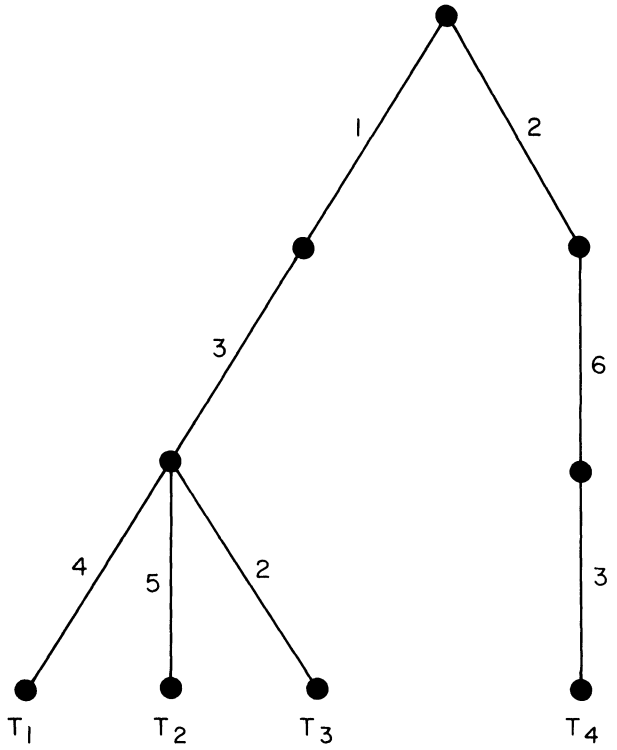

FIG. 2. *Computation tree.*

(i) *F contains all edges joining T to the rest of the graph, i.e.,* $\{(v, w)|v \in T, w \notin T, (v, w)$ *is in the current graph*$\} = \{(v_i, w_i)|1 \leqq i \leqq |F|\}$.

(ii) *F contains edges in a "depth-first order", i.e., if $j \geqq i$ then v_j is a descendent*[1] *of v_i in T.*

Proof. First note that on exit from GROW, F is identical to what it was on entry. This follows by observing that the changes to F in lines 4, 6 and 7 are undone by lines 14, 9 and 10, respectively.

Clauses (i)–(ii) of the Lemma hold for the first call to GROW (line 16), by the initialization step (line 15). In general, if clauses (i)–(ii) hold when GROW is called, they hold for the calls made in line 8, by inspection of lines 3–13 and by the preliminary remark. So by induction, clauses (i)–(ii) hold for all calls to GROW. □

COROLLARY 1. *Let e_j, $1 \leqq j \leqq |T|$, be the edges in T, indexed in the order they are added to T. Let e_j be directed to vertex v_j. Then the descendents of any v_j in T are vertices v_k, $j \leqq k \leqq J$, for some J.*

Proof. It suffices to show the descendents of a given vertex v are added to T consecutively. We do this as follows. Imagine modifying G, by adding an edge $(v, 0)$ leading to a dummy vertex 0; in GROW, when v is added to T and edges (v, w) are pushed onto F (line 6), push $(v, 0)$ first. Now Lemma 1(ii) shows as long as $(v, 0)$ is in F, the edges added to T (in line 4) join descendents of v. When $(v, 0)$ is removed from F, Lemma 1(i) shows there are no edges from descendents of v to vertices not in T. Thus no other vertices become descendents of v. The corollary follows. □

Now we show the bridge test of line 12 is correct.

[1] A vertex is considered a descendent, but not a proper descendent, of itself.

LEMMA 2. *The bridge test sets b to* **true** *exactly when edge e is a bridge of the current graph.*

Proof. Let $e=(u, v)$, and let D denote the descendents of vertex v in the latest spanning tree L. Below we show that when e's bridge test is executed, the current graph has no edge (w, x), where $w \notin D$, $x \in D-v$. This suffices to prove the lemma. For e is *not* a bridge if and only if some path P not containing e goes from r to v. When there are no edges (w, x) as above. P must end in an edge (w, v), $w \notin D$; further, P exists if and only if such an edge (w, v) exists. The bridge test checks if (w, v) exists. Hence it is correct.

So we must show the current graph has no edge (w, x), $w \notin D$, $x \in D-v$. Let the edges in L be e_j, $j=1, \cdots, V-1$, indexed in the order they are added; let $e=e_i$. The bridge test is executed on edges e_j, $j=V-1, \cdots, i+1$, and then on $e_i=e$. For $j>i$, b is set **true**. (Otherwise, another spanning tree would be output after L.)

Now consider any vertex $x \in D-v$. By Corollary 1, the edge in L directed to x is some e_h, $h>i$. So b is **true** for e_h. Thus no edge (w, x), $w \notin D$, exists when e_h's bridge test is executed.

So if (w, x) is in the current graph, it is added in an execution of line 14 following some e_k's bridge test, where $i<k \leqq h$. Corollary 1 shows e_k joins descendents of v. The edges added following e_k's bridge test precede e_k in list F. Lemma 1 (ii) shows these edges originate from D. Thus no edge (w, x), $w \notin D$, is added. We conclude no edge (w, x) is in the current graph. □

Now we can show S is correct.

LEMMA 3. *Procedure S finds all spanning trees rooted at r of a directed graph G rooted at r.*

Proof. Suppose GROW is called, with T a tree rooted at r. Let C be the current graph (when GROW is called). It suffices to show GROW finds all spanning trees (rooted at r) of C containing T. For in the initial call (line 16), T contains only the vertex r, and $C=G$.

The proof is by induction, with the calls to GROW ordered so the size of T is nonincreasing. For the base case, T contains V vertices; this is handled correctly by line 1. For the inductive step, suppose when GROW is called T contains less than V vertices. Let F contain edges e_i, $i=1, \cdots, |F|$. Define

$$\mathcal{T}_i = \{R \mid R \text{ is a spanning tree rooted at } r \text{ and } T \cup e_i \subseteq R \subseteq C-\{e_j \mid 1 \leqq j<i\}\}.$$

By induction, it is easy to see GROW finds the trees $\cup_{i=1}^{k} \mathcal{T}_i$, where e_k is the first edge for which b (in line 12) is **true**. Sets $\mathcal{T}_i$ are disjoint, by definition. Lemma 2 shows e_k is a bridge in the graph $C-\{e_j \mid 1 \leqq j<k\}$. Thus any spanning tree R of C that contains T contains some e_j, $1 \leqq j \leqq k$, i.e., $R \in \mathcal{T}_j$. So GROW finds the desired spanning trees. □

To estimate the efficiency of S, we must give some implementation details. First we discuss how F is managed, and in particular, how it is restored to its original state in line 10. F is a doubly linked list of edges. Line 7 traverses the list of edges directed to v, from beginning to end. Each edge directed from T is removed from F; however, the values of its links are *not* destroyed. Line 10 traverses the list of edges directed to v in the reverse direction, from end to beginning. Each edge directed from T is inserted in F, at the position given by its link values. This way, each edge is restored in its original position.

Next we discuss the implementation of the bridge test. To detect descendents efficiently, the vertices of L are numbered in preorder [1, pp. 54–55]: For a vertex v, $P(v)$ is v's preorder number, and $H(v)$ is the highest preorder number of a descendent of v. So w is a descendent of v if and only if $P(v) \leqq P(w) \leqq H(v)$. This test is used in

line 12. In line 1, when L is formed, the values $P(v)$ and $H(v)$ are computed and stored in the data structure for L.

Now we derive the resource bounds for S.

LEMMA 4. *Procedure S uses $O(EN)$ time and $O(E)$ space on a directed graph rooted at r.*

Proof. First consider time. One execution of the body of the **repeat** loop (lines 4–12), excluding the recursive call (line 8), takes time proportional to the number of edges directed to and from vertex v. Here v is the vertex added to T. In the process of generating a spanning tree, v ranges over all vertices (except r). So the total time in the loop body for one tree is $O(E)$. This dominates the run time of S, which thus is $O(EN)$.

Next consider the space. The graph $\dot{G}$ is stored as a collection of doubly linked lists of edges directed to and from each vertex. This uses $O(E)$ space. At any point in the computation, an edge e may be on the F list, or on at most one FF list. So F and FF use $O(E)$ space. In addition, $O(V)$ space is needed for T, P, and H. Thus the space is $O(E)$. □

Now consider the problem of finding all spanning trees of a directed graph. The possible root vertices r form a strongly connected component that precedes all others. A strong connectivity algorithm can be used to find these roots in time $O(V+E)$[12]. Then procedure S can be applied to each root. So we have the following result.

THEOREM 1. *All spanning trees of a directed graph can be found in time $O(V+E+EN)$ and space $O(V+E)$.*

Next consider the problem of finding all spanning trees of an undirected graph. If the graph is made directed (by giving each edge both directions), and root r is chosen arbitrarily, then procedure S finds all spanning trees of the undirected graph. The time can be estimated more precisely, as follows:

THEOREM 2. *All spanning trees of an undirected graph can be found in time $O(V+E+VN)$ and space $O(V+E)$.*

Proof. We need only show the time bound. Line 1 does a preorder traversal and outputs each spanning tree; the time is clearly $O(V)$ per tree, or $O(VN)$ total. Now we analyze the time spent in lines 4–12, when edge e is added. Ignoring the recursive call (line 8), the time is proportional to the number of edges incident to v. Now we consider two cases, and show in each case the total time in lines 4–12 is $O(VN)$.

First suppose e is a bridge. Each edge f incident to v is in some spanning tree R containing $T \cup e$. Charge the time spent on f, $O(1)$, to R. Then each spanning tree gets charged $O(V)$. So a total time $O(VN)$ is spent on bridges in lines 4–12.

Now suppose e is a nonbridge. The time spent on e in lines 4–12 is $O(V)$. Now we show there are exactly $N-1$ nonbridges, so the total time spent on nonbridges is $O(VN)$: Let the nonbridge e correspond to the tree L used in e's bridge test. Since e fails the test, it gets deleted, and another tree is grown before the next bridge test. So a given tree L corresponds to at most one nonbridge. If L is any spanning tree but the last one found, it is used in the bridge test for some nonbridge. So it corresponds to precisely one nonbridge. Thus there are exactly $N-1$ nonbridges. □

Procedure S can be sped up in a number of ways. The preorder labeling of trees can be done as trees are grown. Several trees can be grown at once (e.g., each edge (w, v) in line 7 gives a spanning tree. Algorithms using this "factoring" approach are [2, pp. 20–25], [8]). However, if each tree is output as a list of edges, $O(VN)$ time is required for the output step. So on undirected graphs, the algorithm is optimal, to within a constant factor.

We have programmed S and other spanning tree algorithms in FORTRAN on

TABLE 1.
Time for graphs with $V = 10$.

N	50	105	310	680	839	1415
t(msec)	100	180	456	831	1048	1634

the CDC6400. Compared to the Minty, Read and Tarjan algorithm, S is over 3 times faster for $6 \leqq V \leqq 10$, $8 \leqq E \leqq 14$; the difference increases with denser graphs. Table 1 shows that the time for S, with V fixed at 10, is approximately proportional to N, as predicted by the $O(VN)$ time bound.

3. Open problems. This section briefly discusses two problems related to this work. The first is to improve the $O(V+E+EN)$ time bound for spanning trees of a directed graph. To illustrate the difficulty here, consider the family of graphs illustrated in Fig. 3. The top part consists of N paths of length 2 from r to s; the bottom part is a directed path of length N from s to t, plus all possible back edges. The

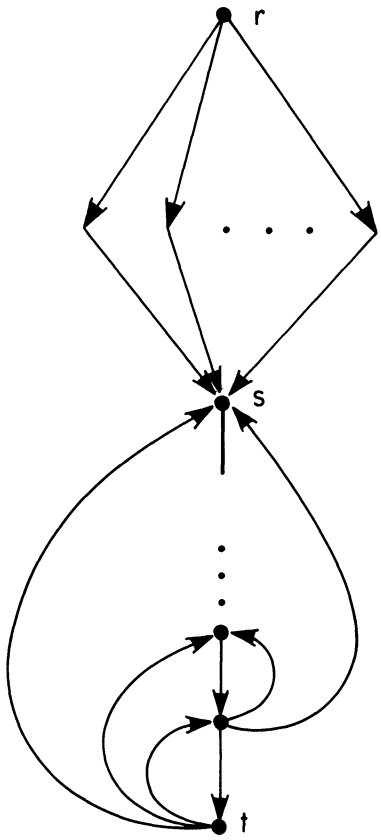

FIG. 3. *Difficult graph.*

algorithm of Theorem 1 uses $O(EN)$ time on these graphs. The time spent repeatedly scanning back edges is "wasted." (As a point of interest, note the algorithm of [11] can use exponential time per tree on these graphs.)

The second problem is, can the computation tree of S (Fig. 2) be represented in less than $O(VN)$ space? Note some computation trees have $O(VN)$ nodes. For

instance, the tree for an undirected cycle has $V(V-1)/2 = O(VN)$ nodes. There are two reasons why a more compact form is desirable.

First, the claim S is optimal for undirected graphs is based on a lower bound for outputting the spanning trees. If a computation tree is acceptable output, it may be possible to lower this bound and speed up the algorithm.

Second, consider the problem of listing all spanning trees in order of increasing weight in a weighted undirected graph. (In a *weighted* graph, each edge has a numerical weight; a tree's weight is the sum of all its edge weights.) One approach is to find all spanning trees, and then sort them. The sort takes time $O(N \log N)$, which is $O(\min(V \log V, E)N)$, since $N \leqq \min(2^E, V^{V-2})$. This dominates the run time of the algorithm. The space is $O(VN)$, since the spanning trees must be saved until the sort is done. A previous algorithm [3] uses $O(EN)$ time and $O(E+N)$ space. So our approach is no slower, sometimes faster, but uses more space. Thus a "reduced" computation tree is desirable.

Acknowledgments. The authors thank the referees for their suggestions.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. M. CHASE, *Analysis of algorithms for finding all spanning trees of a graph*, RC3190, IBM T. J. Watson Research Center, Yorktown Heights, NY, Dec. 1970.

[3] H. N. GABOW, *Two algorithms for generating weighted spanning trees in order*, this Journal, 6 (1977), pp. 139–150.

[4] S. L. HAKIMI AND D. G. GREEN, *Generation and realization of trees and k-trees*, IEEE Trans. on Circuit Theory, CT-11 (1964), pp. 247–255.

[5] F. J. MACWILLIAMS, *Topological network analysis as a computer program*, IRE Trans., CT-5 (1958), pp. 228–229.

[6] W. MAYEDA AND S. SEHU, *Generation of trees without duplications*, IEEE Trans. on Circuit Theory, CT-12 (1965), pp. 181–185.

[7] W. MAYEDA, *Graph Theory*, John Wiley, New York, 1972.

[8] M. D. MCILROY, *Generation of spanning trees (Algorithm 354)*, Comm. ACM, 12 (1969), p. 511.

[9] G. J. MINTY, *A simply algorithm for listing all the trees of a graph*, IEEE Trans. on Circuit Theory, CT-12 (1965), p. 120.

[10] R. C. READ AND R. E. TARJAN, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), pp. 237–252.

[11] S. SHINODA, *Finding all possible directed trees of a directed graph*, Electron. Commun. Japan, 51-A (1968), pp. 45–46.

[12] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[13] H. WATANABE, *A computational method for network topology*, IRE Trans., CT-7 (1960), pp. 296–392.