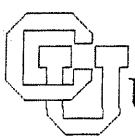
Finding All Spanning Trees of Undirected and Directed Graphs *

Harold N. Gabow

CU-CS-103-77



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was partially supported by the National Science Foundation under Grant GJ36461.

Finding All Spanning Trees of Undirected and Directed Graphs

by

Harold N. Gabow
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-103-77 January 1977

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 is actually 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.

This work was partially supported by the National Science Foundation under Grant GJ36461.

1. Introduction

The problem of finding all spanning trees of a connected, undirected graph arises in the solution of electrical networks [May]. Several algorithms of varying efficiency have been propsed [HG, Mac, MayS, Mc, Mi, RT,W]. We present an algorithm that uses O(VN) time and O(E) space, where the graph has V vertices, E edges, and $N \geq 1$ spanning trees. The previously best known algorithm of Minty-Read-Tarjan [Mi,RT] uses O(EN) time and O(E) space. Our algorithm may be viewed as a refinement of theirs. In terms of worst case asymptotic bounds, our algorithm is optimal.

The algorithm also applies to directed graphs. Here it uses O(EN) time and O(E) space. There appear to be no previous published algorithms for directed graphs.

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 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. Other papers use the term arborescence for spanning tree of a directed graph. There appears to be no standard term for what we call a "bridge" of a directed graph.

2. The Algorithm

First we describe an algorithm that finds all spanning trees rooted at r in a directed graph. Then we apply the algorithm to find all spanning trees in directed and undirected graphs.

The basic task is to find all spanning trees that contain a given subtree T that is 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 containing $T \cup e_2$, but not containing e_1; then delete e_2. To continue, repeatedly

choose an edge e_i from T to a vertex not in T; find all spanning trees containing $T \cup e_i$, but not containing any $e_j, j<i$; delete e_i. Stop when the edge e_k that has just been processed is a bridge of the modified graph. At this point all spanning trees containing T have been found. For if a spanning tree does not contain $e_j, j<k$, it must contain the bridge e_k.

To achieve our time bound, we need an efficient method for discovering when edge e is a bridge. This can be done in a variety of ways; set merging techniques and edge exchanges are two possibilities [G]. Here we describe an approach based on depth-first search.

Choose edges e so T grows depth-first. Consider L, the last spanning found that contains $T \cup e$. If e=(u,v), then in L, vertex v has the fewest descendents possible (among all spanning trees containg $T \cup e$). Equivalently, no edge goes from a non-descendent of v to a proper descendent of v. So e is a bridge when no edge other than e goes from a non-descendent of v to v. This observation gives an efficient bridge test.

The depth-first search must be implemented with some care. The algorithm uses F, a list of all edges directed from a vertex in T to a vertex not in T. F is managed as a stack: to enlarge T, an edge e is popped from the front of F; new edges for $T \cup e$ are pushed onto the front of F. In addition, when e is added to T, edges must be removed from F, and when e is removed from the tree, these edges must be restored in F. It is important that the remove and restore operations leave the order of edges unchanged in F. Otherwise, the search will not be depth-first.

Besides F, the algorithm uses lists $\overline{F}$, also managed as stacks. The algorithm is given below, in Algol-like notation.

```
                    procedure S;begin
                       procedure GROW;begin
 1.                    if T has V vertices then begin L←T;output (L) end
 2.                    else begin make F̄ an empty list, local to GROW;
 3.                       repeat
 4.  tree edge:              pop edge e from F; let e=(u,v), where u∈T,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 F̄;
12.  bridge test:            if there is an edge(w,v),where w≠u and w is not a
                                descendent of v in L then b←false else b←true;
13.                      until b;
14.  restore G:          pop each edge e from F̄, push e onto F and add e to G;
                         end GROW;
15.                 initialize T to contain vertex r; initialize F̄ to contain
                       all edges (r,s);
16.                 GROW;
                    end S;
```

Figure 1 shows a graph with four spanning trees rooted at r. Figure 2 shows a computation tree indicating how procedure S finds these trees T_i, $1 \le i \le 4$. In the computation tree, a node g represents a call to GROW; the arcs directed out of g correspond to the edges e_i added to T in this call. 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 (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 (2,5), T_3 is found before T_2, edge 1 is mistakenly declared a bridge, and T_4 is not found.

Lemma 1: Procedure S finds all spanning trees of a directed graph rooted at r.

Proof: It suffices to prove this inductive assertion: If GROW is called with T a tree, and F a list of the edges of G from T to V-T, then GROW finds all spanning trees in G that contain T.

To prove the assertion, let F contain edges $e_i, i \geq 1$. Define

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

Using the inductive assertion, it is easy to see GROW finds the trees in $\bigcup_{i=1}^{k} \mathcal{T}_i$, where e_k is the first edge for which b is true. Sets $\mathcal{T}_i$ are disjoint, by definition. Assuming e_k is a bridge, any spanning tree contains some $e_i, i \leq k$; so GROW finds all desired spanning trees.

Thus we need only prove bridges are discovered correctly in line 12. It suffices to show there are no edges (x,y), where y is a descendant of v in L, x is not, and $y \neq v$. For in this case, the existence of a path to v avoiding e is equivalent to the existence of the edge tested for in line 12. Thus e is a non-bridge exactly when b is false.

So suppose there is an edge (x,y), as described above. We derive a contradiction, as follows. Let f be the edge in L directed into y. Since edges are added depth-first to T, f is added after e. So the bridge test for f is executed before that for e. For f, b is false, because of edge (x,y). This implies L is not the last spanning tree found containing $T \cup e$, since GROW removes f and finds more spanning trees. This contradiction proves (x,y) does not exist. □

Lemma 2: Procedure S uses O(EN) time and O(E) space on a directed graph rooted at r.

Proof: First we give some implementation details for F and for the bridge test. 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.

Now consider the bridge test. We need an efficient way to detect descendents. Suppose the vertices of L are numbered in preorder. For a vertex v, P(v) is its preorder number; H(v) is the highest number assigned to a descendent of v. Then w is a descendent of v if and only if $P(v) \leq P(w) \leq H(v)$; 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.

Now we derive the time bound. 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 v. Here v is the vertex added to the tree. In the process of generating one 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 is O(EN).

Next consider the space. The graph 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 $\overline{F}$ list. So F and $\overline{F}$ use O(E) space. In addition, O(V) space is needed for T, P, and H. So the total 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. An efficient strong connectivity algorithm [T] can be used to find these roots in time O(V+E). 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 suppose the graph is undirected. Make it directed by giving each edge both directions. Choose r arbitrarily. Now procedure S finds all spanning trees. The time bound for S can be improved, 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 prove the time bound. Let C be the computation tree for S, as illustrated in Fig. 2. Each node in C represents a call to GROW. An arc labelled e in C corresponds to an execution of the repeat loop body where edge e is added to T. We make implicit use of this correspondence when we refer to arcs e and edges e.

The time in the leaves of C is O(VN), since each leaf uses time O(V) to do a preorder traversal of the tree and to output it (line 1). The remaining time is in the interior nodes of C. Now we apportion this time to the arcs e of C, so e is charged O(1) if edge e is a bridge, and O(V) otherwise.

An execution of the repeat loop body (lines 4-12) that adds edge e to T takes time proportional to the number of edges incident to v, or O(V). (We ignore the recursive call, line 8). If edge e is a non-bridge, we charge this O(V) to arc e. Otherwise e is a bridge. Every edge f incident to v is in some spanning tree containing T. So an arc f appears below e in C. We charge O(1), the time spent on edge f, to this arc f. (Note this charge to f is made only once, when v enters the tree.) This apportions the time to the arcs of C, as desired.

Now we show the total of all arc charges is O(VN). The number of bridges is at most the number of arcs of C; since C has height V-1, this is at most (V-1)N. So the charge to bridges is O(VN). The number of non-bridges is exactly the number of arcs minus the number of interior nodes (recall an interior node processes edges until it finds a bridge). Since C is a tree, this quantity is N-1. So the charge to non-bridges is O(VN). □

Procedure S can be sped up in a number of ways. The preorder labelling 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). However, if each tree is output as a list of arcs, O(VN) time is required for the output step. So the algorithm is optimal, to within a constant factor.

Procedure S and other spanning tree algorithms are currently being programmed. Preliminary indications are that in practice, S is simpler and faster than the previously best known algorithm of Minty-Read-Tarjan [Mi,RT].

3. Open Problems

We briefly discuss two problems related to this work. The first is, can we improve the O(EN) time bound for all spanning trees of a directed graph? To illustrate the difficulty here, take any graph that is "dense", but has a unique spanning tree (e.g., a directed path, plus all back edges except those entering the start vertex). Let r be the root. Add a new root r, several new vertices v, and new edges (r',v),(v,r). The algorithm of Theorem 1 uses O(EN) time on such graphs. The time spent repeatedly scanning back edges is "wasted".

Another problem is, can the computation tree be represented in less than O(VN) space? Note that for procedure S, some computation trees have O(VN) nodes, e.g., the tree for an undirected cycle has V(V-1)/2=O(VN) nodes. There are two reasons a more compact form is desirable.

First, our time optimality argument 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 $N \leq \min(2^E, V^{V-2})$. 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 [G] 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 author thanks Eugene Meyers for providing the initial stimulus for this work and for programming several versions of the algorithm.

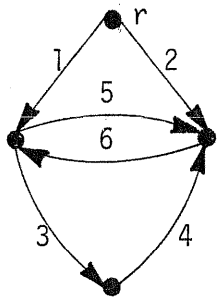


Fig. 1
Example graph

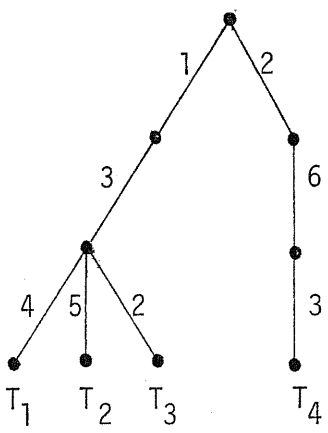


Fig. 2
Computation tree

References

[G] Gabow, H. N., "Two algorithms for generating weighted spanning trees in order", SIAM J. Computing 6 (1977), to appear.

[HG] Hakimi, S. L. and D. G. Green, "Generation and realization of trees and k-trees", IEEE Trans. on Circuit Theory, Vol. CT-11 (1964), pp. 247-255.

[Mac] MacWilliams, F. J., "Topological network analysis as a computer program", IRE Trans., Vol. CT-5 (1958), pp. 228-229.

[MayS] Mayeda, W. and S. Sehu, "Generation of trees without duplications", IEEE Trans. on Cirucit Theory, Vol. CT-12 (1965), pp. 181-185.

[May] Mayeda, W., Graph Theory, John Wiley and Sons, N.Y., 1972.

[Mc] McIlroy, M. D., "Generation of spanning trees (Algorithm 354)", Comm. ACM 12 (1969), p. 511.

[Mi] Minty, G. J., "A simple algorithm for listing all the trees of a graph", IEEE Trans. on Circuit Theory, Vol. CT-12 (1965), p. 120.

[RT] Read, R. C. and R. E. Tarjan, "Bounds on backtrack algorithms for listing cycles, paths, and spanning trees", Networks 5, pp. 237-252.

[T] Tarjan, R. E., "Depth-first search and linear graph algorithms", SIAM J. Computing 1 (1972), pp. 146-160.

[W] Watanabe, H., "A computational method for network topology", IRE Trans., Vol. CT-7 (1960), pp. 296-392.