EDGE-DISJOINT SPANNING TREES, DOMINATORS, AND DEPTH-FIRST SEARCH

by
Robert E. Tarjan

STAN-CS-74-455
SEPTEMBER 1974

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

EDGE-DISJOINT SPANNING TREES, DOMINATORS, AND DEPTH-FIRST SEARCH

by

Robert Endre Tarjan
Department of Electrical Engineering and Computer Sciences
Computer Science Division
University of California, Berkeley, California 94720
and
Computer Science Department
Stanford University, Stanford, California 94306

Abstract

This paper presents an algorithm for finding two edge-disjoint spanning trees rooted at a fixed vertex of a directed graph. The algorithm uses depth-first search, an efficient method for computing disjoint set unions, and an efficient method for computing dominators. It requires $O(V \log V + E)$ time and $O(V+E)$ space to analyze a graph with V vertices and E edges.

Keywords: branching, bridge, complexity, connectivity, depth-first search, directed graph, dominators, matroid, set union, spanning tree, tree.

This work was partially supported by the National Science Foundation, Grant GJ-35604X, and by a Miller Research Fellowship, at the University of California, Berkeley; and by the Office of Naval Research contract NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

EDGE-DISJOINT SPANNING TREES, DOMINATORS, AND DEPTH-FIRST SEARCH

by

Robert Endre Tarjan

Definitions

A graph $G = (V, E)$ is an ordered pair consisting of a set of vertices V and a multiset of edges E. Let V be the number of vertices and E be the number of edges in G. In an undirected graph, each edge is an unordered pair (v,w) of distinct vertices; in a directed graph, each edge is an ordered pair (v,w) of distinct vertices. (This definition allows multiple edges but not loops in graphs.) An edge (v,w) is incident to v and w. A directed edge (v,w) leaves v and enters w. If $G_1 = (V_1, E_1)$ is a graph and $V_1 \subseteq V$, $E_1 \subseteq E$, then G_1 is a subgraph of G. We define $G-G_1 = G-E_1 = (V, E-E_1)$. If $V_2 \subseteq V$ and $E_2 = \{(i,j) \mid (i,j) \in E \text{ and } i,j \in V_2\}$ (E_2 is a multiset), then $G_2 = (V_2, E_2)$ is the subgraph of G induced by the vertices V_2.

A sequence of edges $(v_1,v_2),(v_2,v_3), \ldots, (v_{n-1},v_n)$ in G is a path from v_1 to v_n. This path contains vertices $v_1,\ldots,v_n$ and avoids all other vertices. There is a path of no edges from every vertex to itself. A path is simple if all its vertices are distinct except possibly v_1 and v_n. A cycle is a path such that $v_1 = v_n$. A cycle must contain at least two edges. Vertex w is reachable from vertex v if there is a path from v to w. A directed graph is strongly connected if every vertex is reachable from every other. A flow graph (G, r) is a graph with a distinguished vertex r such

that every vertex in G is reachable from r . Vertex v dominates vertex w in flow graph (G,r) if $v \neq w$ and every path from r to w contains v . An edge (v,w) is a bridge of a flow graph if every path from r to w contains (v,w) .

A tree T is a graph with a vertex r such that there is a unique simple path from r to every vertex in T . If T is directed, r is unique and is called the root of T ; if T is undirected, r can be any vertex of T . If T_1 is a tree and T_1 is a subgraph of T , T_1 is called a subtree of T . If T is a subgraph of a graph G and T contains all the vertices of G , then T is a spanning tree of G . If T is a directed tree, the notation $v \rightarrow w$ means (v,w) is an edge of T ; in this case v is the father of w and w is a son of v . The notation $v \overset{*}{\rightarrow} w$ means there is path from v to w in T; v is an ancestor of w (proper if $v \neq w$) and w is a descendant of v (proper if $v \neq w$) . Using these conventions, every vertex is a (non-proper) ancestor and descendant of itself.

History

Let G be an undirected graph. Suppose we wish to find (i) a maximum number of spanning trees in G which are pairwise edge-disjoint, or (ii) a minimum number of spanning trees whose union contains all the edges of G , or (iii) a set of k spanning trees such-that the fewest possible edges are outside the union of the trees (for some fixed constant k). Problem (iii) for $k = 2$ has applications in the solution of Shannon switching games and in the "mixed" analysis of electrical networks. Many researchers, including Tutte[26], Edmonds [4,5], Nash-Williams [16,17], and others [3,9,10,15,18] have

studied one or more of these problems and have given efficient algorithms for solving them. The best algorithm known has a time bound of $O(E^2)$ for problems (i) and (ii) and a time bound of $O(k^2V^2)$ for problem (iii) [25].

Less is known about analogous problems in directed graphs. Edmonds has considered the problem of finding k mutually edge-disjoint spanning trees rooted at a fixed vertex r . He has shown that there exist k disjoint spanning trees rooted at r if and only if there exist at least k edge-disjoint paths from r to any other vertex v [7]. Based on this result, one can use a network flow algorithm to find k disjoint spanning trees, if they exist, in $O(k^2E^2)$ time [24].

In this paper we consider faster ways of finding exactly two directed spanning trees with fewest common edges.

Lemma 1: Let (G,r) be a flow graph. Each bridge in G is in every spanning tree rooted at r . There exist two spanning trees with only the bridges in common.

We can prove Lemma 1 using the algorithm below, which finds two spanning trees of a directed graph with only the bridges in common.

(1) find a spanning tree T_1 rooted at r ;

find a tree T_2 rooted at r in $G-T_1$ with as many vertices as possible;

while T_2 is not a spanning tree do begin

a: find an edge $v \rightarrow w$ in T_1 such that $v \in T_2$, $w \notin T_2$, and no descendants x, y of w in T_1 satisfy $x \rightarrow y$ in T_1, $x \in T_2$, and $y \notin T_2$;

b: if w is not reachable from r in $G-T_2-\{(v,w)\}$ then

duplicate (v,w) ;

comment (v,w) must be a bridge;

replace T_1 by a spanning tree rooted at r in $G-T_2-\{(v,w)\}$;

c: find a tree T_2 rooted at r in $G-T_1$ with as many vertices as possible;

end;

Lemma 2: Step (1) finds two spanning trees rooted at r which have only bridges of G in common.

Proof; At least one vertex (w) gets added to T_2 during each execution of the while loop in step (1), so the while loop can be executed at most V-1 times. Thus the algorithm terminates. Clearly the algorithm works correctly if the test in statement b fails whenever (v,w) is not a bridge. Suppose (v,w) is not a bridge and the test in statement b is performed for some T_2 . There is a path $p = (r,v_2)(v_2,v_3),\ldots,(v_{n-1},w)$ in

$G-\{(v,w)\}$. Let (v_i,v_{i+1}) be the last edge on this path such that $v_i \in T_2$. Then $(v_i,v_{i+1}) \in T_1$; otherwise (v_i,v_{i+1}) would have been added to T_2 during the last execution of statement a. Since $v_{i+1} \notin T_2$, v_i is not a descendant of w in T_1 by the condition in statement a. Then w must be reachable from r in $G-T_2-\{(v,w)\}$ by a path of edges from r to v_i in T_1 followed by the path $(v_i,v_{i+1}),\ldots(v_{n-1},w)$. Thus the test in statement b fails. It follows that step(1) computes two spanning trees with only bridges in common.

Q.E.D.

Lemma 2 implies the second half of Lemma 1; the first half of Lemma 1 is obvious. Lemma 1 also follows from Edmond's more general result [7].

Statements a, b, and c clearly require O(E) execution time if a set of adjacency lists is used to represent the graph, so the whole algorithm requires $O(VE)$ time and O(V+E) space. We can improve the method's time bound to $O(V^2)$ by first finding a set of edges partitionable into two disjoint spanning trees and then applying step (1). However, depth-first search gives an even faster algorithm.

Depth-First Search

If T is a directed tree rooted at r , a preorder numbering [11] of the vertices of T is any numbering which can be generated by the following algorithm:

```
begin
   procedure PREORDER(v); begin
      number v greater than any previously numbered vertex;
      comment  if v = r,  v may be numbered arbitrarily;
      for w such that v → w do PREORDER(w);
   end;
   PREORDER(r);
end;
```

Lemma 3: Let ND(v) denote the number of descendants of a vertex v in a directed tree T . If T has V vertices numbered from 1 to V in preorder and vertices are identified by number, then $v \overset{*}{\rightarrow} w$ in T **iff** $v \leq w < v + ND(v)$.

Proof: See [21].

Let (G,r) be a flow graph, and let T be a spanning tree of G rooted at r which has a preorder numbering. T is a depth-first spanning tree (DFS tree) if the edges in G-T can be partitioned into three sets:

(i) a set of edges (v,w) with $w \overset{*}{\rightarrow} v$ in T , called cycle arcs;

(ii) a set of edges (v,w) with $v \overset{*}{\rightarrow} w$ in T , called forward arcs;

(iii) a set of edges (v,w) with neither $v \overset{*}{\rightarrow} w$ nor $w \overset{*}{\rightarrow} v$, and $w < v$, called cross arcs.

A DFS tree is so named because it can be generated by starting at r and carrying out a depth-first search of G , numbering the vertices in increasing order as they are reached during the search. A properly

implemented algorithm [19,21] requires O(V+E) time to execute step (2) below.

(2) Carry out a depth-first search of G , finding a DFS tree, numbering the vertices in preorder, calculating ND(v) , and finding sets of cycle arcs, forward arcs, and cross arcs.

Henceforth assume that step (2) has been applied to flow graph (G,r) , that T is the resulting DFS tree, and that vertices are identified by number. An s-order numbering s(v) of the vertices of T is a preorder numbering such that $w \rightarrow w_1$, $w \rightarrow w_2$, and $w_1 < w_2$ imply $s(w_1) > s(w_2)$. An s-order numbering of T can be calculated during step (2).

Lemma 4: Let s(v) be an s-order numbering of T . Then $s(v) < s(w)$ if (v,w) is a tree arc, forward arc, or cross arc, and $s(v) > s(w)$ if (v,w) is a cycle arc.

Proof: See [21].

If G is acyclic, s(v) defines a topological sorting of the vertices (an ordering such that all arcs run from smaller numbered to larger numbered vertices). . By examining the vertices of G in s-order, from largest to smallest, we can compute the strong components [19], the period [13], or the weak components [23] of G , each in O(V+E) time. By examining the vertices of G in preorder from largest to smallest we can compute the dominators and bridges of G in O(V log V+ E) time, as discussed in the next section. A third systematic method of exploring a DFS tree allows us to find pairs of disjoint spanning trees efficiently.

Let S be a set of vertices in G and let $v \notin S$. By collapsing S into v we mean forming a new graph G' by deleting all vertices in S and all edges incident to vertices in S , adding a new edge (v,x) for each deleted edge (w,x) with $x \notin S \cup \{v\}$, and adding a new edge (x,v) for each deleted edge (x,w) with $x \notin S \cup \{v\}$. Each edge of G' corresponds to an edge of G , and each edge of G either disappears or corresponds to an edge of G' .

For any vertex w , let $C(w) = \{v \mid (v,w)$ is a cycle arc$\}$ and let $P(w) = (v \mid w \overset{*}{\rightarrow} v$ and $\exists z \in C(w)$ such that there is a path from v to z which contains only proper descendants of w$\}$. Let w be the largest vertex of G such that $C(w) \neq \emptyset$. Let G' be formed by collapsing P(w) into w . Let T' be the subgraph of G' whose edges correspond to the edges of T .

Lemma5: The subgraph of G induced by the vertices $P(w) \cup \{w\}$ is strongly connected.

Proof: Obvious.

Lemma 6: T' , with numbering the same as that of T , is a DFS tree of G' with root r . Cycle arcs of G' correspond to cycle arcs of G, forward arcs of G' correspond to forward arcs or cross arcs of G, and cross arcs of G' correspond to cross arcs of G .

Proof: See [22].

Suppose we calculate P(V) in G and collapse P(V) into V to create a new graph G' , calculate P(V-1) in G' and collapse P(V-1) into V-1 , and so on, until we reach vertex 1 . Eventually

we collapse G into an acyclic graph whose vertices correspond to the maximal strongly connected subgraphs of G . This idea gives a way to test the reducibility of G efficiently [22], and to efficiently find a pair of edge-disjoint spanning trees (as we shall see).

Dominators

Lemma 7: Let (G,r) be a flow graph with G = (V,E) and let T be a DFS tree of G with root r . Edge (v,w) is a bridge of G iff (v,w) is a tree arc, w has no entering forward arcs or cross arcs, and there is no cycle arc (x,w) such that w does not dominate x .

Proof: If (v,w) is not a tree arc, or w has an entering forward arc or cross arc, or there is a cycle arc (x,w) such that w does not dominate x , then there is a path from r to w which avoids (v,w) , and (v,w) is not a bridge. If (v,w) is not a bridge, there must be a simple path from r to w which avoids (v,w) . If the last edge on this path is a tree arc, (v,w) is not a tree arc, if it is a forward arc or a cross arc then w has an entering forward arc or cross arc, and if it is a cycle arc (x,w) then w does not dominate x .

Q.E.D.

If v dominates w and no vertex larger than v dominates w , then v is called the immediate dominator of w , denoted v = d(w) . By convention d(1) = 0 .

Lemma 8: The edges $\{(d(w),w) \mid w \in V-\{1\}\}$ form a tree, called the dominator tree of G, such that v dominates w if and only if $v \overset{*}{\rightarrow} w$ in the dominator tree.

Proof: See [2].

If we calculate $d(w)$ for all vertices w, then we can use Lemmas 7 and 8 to find the bridges of G. Here is an $O(V \log V+E)$ algorithm for calculating $d(w)$ values. The method is a greatly simplified and improved version of [21]. We calculate $d(w)$ by processing the vertices in preorder from largest to smallest. Let $G_k = (V,\{(v,w) \mid (v,w) \in E \text{ and } w \geq k\})$. $G_1 = \cdots = G_{V+1} = (V,\emptyset)$. Let $d_k(w) = \min\{v \mid w \text{ is reachable from } v \text{ in } G_k\}$. Clearly $d_k(w) \leq k$ for all w, and $d_k(w) < k$ if $k \leq w$ and $w > 1$.

Lemma 9: $d_k(k) = \min(\{v \mid (v,k) \text{ is a forward arc or tree arc}\} \cup \{d_{k+1}(v) \mid (v,k) \text{ is a cross arc or cycle arc}\})$ if $k > 1$.

Proof: Obvious.

Lemma 10: Suppose $w \neq k$. If $k \overset{*}{\rightarrow} w$ and $d_{k+1}(w) > d_k(k)$, then $d_k(w) = d_k(k)$. Otherwise $d_k(w) = d_{k+1}(w)$.

Proof: If $w>k$, then any path from k to w must contain a common ancestor of w and k. This result is proved in [21]. Thus w is reachable from k in G_k iff $k \overset{*}{\rightarrow} w$. Hence $d_k(w) = \min(d_{k+1}(w),d_k(k))$ if $k \overset{*}{\rightarrow} w$, $d_k(w) = d_{k+1}(w)$ otherwise.

Q.E.D.

Lemmall: Suppose $w \neq k$ and $d(w) \leq k$. If $d_{k+1}(w) = k$ then $d(w) = k$. Otherwise $d(w) < k$.

Proof: If k does not dominate w , there is some path from 1 to w which avoids k . Let (x,y) be the last edge on this path with $x<k$. Then $d_{k+1}(w) \leq x < k$. Thus if $d_{k+1}(w) = k$, w dominates k , and since $d(w) < k$, $d(w) = k$. If $d_{k+1}(w) \neq k$, then $d_{k+1}(w) < k$, and w is reachable from 1 by a path of tree arcs, to $d_{k+1}(w)$ followed by a path in G_{k+1} . Since this path avoids k , k doesn't dominate w , and since $d(w) \leq k$, $d(w) < k$.

Q.E.D.

We use Lemmas 9, 10, and 11 to calculate dominators, working from $k = V$ to $k = 1$. The algorithm appears below in an Algal-like notation. At the end of an execution of for loop d below, each vertex $w \geq k$ will be contained in a unique set. All vertices w in the same set will have the same value of $d_k(w)$. Each set will have a distinguishing name and a priority whose value is $d_k(w)$ for all elements w of the set. In addition, given a set, either all its vertices have known immediate dominators or none have known immediate dominators. Associated with each vertex $w \geq k$ such that $v \rightarrow w$ implies $v < k$ will be a priority queue named w containing all sets which have descendants of w as elements.

We use the following set operations:

FIND(w) returns the value (x,p) where x is the name and p is the priority of the set containing w as an element;

UNION(x,y) adds the elements in set x to set y (destroying x). The new set y remains in the same priority queue with the same priority as the old set y.

We use the following priority queue operations:

HIGH(q) returns the value (x,p) where x is the name and p the priority of a set in queue q with highest priority (by convention HIGH(q) returns (0,0) if queue q is empty);

DELETE(x,q) deletes the set named x from queue q;

QUNION(q,r) adds the sets in queue q to queue r (destroying queue q).

```
(3)  d: for k:=V step -1 until 1 do   update: begin
        d(k) :=0;
     e: p :=min((v | (v,k) is a forward arc or a tree arc)
           U (p' | ∃x,y such that (y,k) is a cross arc or cycle arc
              and (x,p') = FIND(y)} U {k-1});
        comment p = d_k(k) if k ≠ 1, p = 0 if k = 1;
        create a set {k} with name 2k-1 and priority p;
        create a set ∅ with name 2k and priority p;
        create a queue named k containing the sets named 2k-1 and 2k;
        for w such that k → w do QUNION(w,k)
        (x,p') :=HIGH(k);
     f: while p' > p do begin
     g:       ≥ k if p' all vertices w in set x have d(w) = 0 then
              for each vertex w in set x do d(w) :=k;
           DELETE(x,k);
           if all vertices w in set x have d(w) = 0 then UNION(x,2k-1)
           else UNION(x,2k);
           (x,p') :=HIGH(k);
        end;
        if the set named 2k is empty then DELETE(2k,k);
     end;
```

Steps (2) and (3) will compute d(w) for every vertex w . Statement e implements Lemma 9, statement f (minus statement g) implements Lemma 10, and statement g implements Lemma 11. The total time required by steps (2) and (3) is O(V+E) plus time for O(V) set unions, O(E) FIND's, and O(V) priority queue operations. The set operations require

$O(V \log V + E)$ time using a method given in [8,20]. The priority queue operations require $O(V \log V)$ time using Crane's method [12]. The total time is thus $O(V \log V + E)$. The storage space required is $O(V+E)$. (See [21] for further details.)

If the graph has no cross arcs, the priority queues are unnecessary and dominators can be calculated faster, using only disjoint set union operations. (See [21] for details.)

Disjoint Branchings

The dominators algorithm above forms an important part of an efficient algorithm for finding two spanning trees having only bridges in common. We use the dominators algorithm to find all the bridges of G . We duplicate the bridges and discard all but the edges which will form the spanning trees. The following lemma forms the basis for this calculation.

Lemma 12: Let $w \neq 1$ be a vertex of G . Suppose the tree arc entering w is not a bridge. There must exist a non-tree arc (x,w) with $d_{w+1}(x) = d_w(w)$. Form G' by deleting all edges entering w except the entering tree arc and (x,w) . Let $d'_k(v) = \min\{x \mid v \text{ is reachable from } x \text{ in } G'_k\}$ where $G'_k = (V, \{(x,y) \mid (x,y) \text{ is an edge of } G' \text{ and } y \geq k\})$. Then $d'_k(v) = d_k(v)$ for all v and for all k .

Proof: Clearly $d'_k(v) \geq d_k(v)$ for all k and $d'_k(v) = d_k(v)$ for all $k > w$. Suppose there is some $k \leq w$ and some v such that $d'_k(v) > d_k(v)$. Then there is a simple path containing w from $d_k(v)$ to v in G_k . Let p be the part of this path from $d_k(v)$

to w . If there is a vertex y on p with $y < w$, $y \neq d_k(v)$ and $y \neq w$, then some common ancestor z of y and w lies on p by Lemma 8 of [21]. But the path of tree arcs from z to w is in G'_k , and there is a path from $d_k(v)$ to v in G'_k , a contradiction.

If every vertex y on p other than $d_k(v)$ and w has $y \geq w$, then p is a path in G_w , and $d_k(v) = d_k(w) = d_w(w)$. But clearly $d_w(w) = d'_w(w)$, and thus there is a path from $d_k(v) = d;(w)$ to w to v in G'_k , a contradiction. Q.E.D.

To find two spanning trees with fewest common edges, we execute step (2), which carries out a depth-first search of the problem graph G . Next we execute step (4) below, which uses statement "update" of step (3).

```
(4)   for k := V step -1 until 2 do begin
        m := min{x|(x,k) is a forward arc} U
             U {p' |∃x,y such that (y,k) is a cross arc or a cycle arc
                and (x,p') = FIND(y)} U {k};
        comment m = min{d_{k+1}(x)|(x,k) is a non-tree arc] U [k);
        ifm = k then begin
          comment the tree arc entering k is a bridge;
          duplicate the tree arc entering k;
          delete all other edges entering k;
        end else begin
          let (x,k) be a non-tree arc with d_{k+1}(x) = m;
          delete all edges entering k except the tree arc and (x,k);
          calculate d_k values from d_{k+1} values using statement "update"
            in step (3);
          comment  statement "update" may actually be simplified somewhat
            since dominators are not needed, only d_k values;
```

end end;

delete all edges entering vertex 1;

It is easy to prove by induction using Lemmas 7 and 12 that during the k-th iteration of the for loop in step (4), $m = k$ if the tree arc entering k is a bridge, $m = d_k(k)$ otherwise; and that after step (4) is completed, the graph remaining is a bridgeless graph with exactly $2(V-1)$ edges, containing two copies of each bridge of the original graph. Execution of step (4) requires $O(V \log V + E)$ time. Henceforth assume that G is a bridgeless flow graph with $2(V-1)$ edges.

The idea of the remaining part of the disjoint spanning trees algorithm is to collapse strongly connected regions of G until we create a bridgeless acyclic graph. We can easily find two disjoint spanning trees in the resulting graph. Then we expand the collapsed regions, modifying the spanning trees accordingly, to produce two disjoint spanning trees of the original graph.

Let $G^{(V+1)} = G$. For $2 \leq k \leq V$, let $G^{(k)}$ be formed from $G^{(k+1)}$ by computing $P(k)$ in $G^{(k+1)}$ and collapsing $P(k)$ into k . For $k = 2,3,\ldots,V+1$, $G^{(k)}$ has a DFS tree $T^{(k)}$ corresponding to the DFS tree T of G . $G^{(2)}$ is acyclic. The following lemmas show that d_k values are preserved during this collapsing process, and that $G^{(V)}, G^{(V-1)},\ldots,G^{(2)}$ have no bridges.

Lemma 13: Let G be a bridgeless flow graph and let w be the highest vertex of G with entering cycle arcs. Let G' be formed from G by collapsing $P(w)$ into w . Suppose $d'_k(v)$ is defined in G' . Then $d'_k(v) = d_k(v)$ for all v in G' and for all $k \leq w$.

Proof: If $k \leq w$, every path in G'_k (possibly containing w') corresponds to a path in G_k (possibly containing one or more vertices of $P(w) \cup \{w\}$) and vice-versa. It follows that $d'_k(v) = d_k(v)$.

Q.E.D.

Lemma 14: Let G be a bridgeless flow graph. Let $G^{(V+1)}, G^{(V)}, \ldots, G^{(2)}$ be defined as above. Then for $k = 2,3,\ldots,V+1$, $G^{(k)}$ has no bridges.

Proof: $G^{(V+1)}$ has no bridges by assumption. Suppose for some $k \geq 2$, $G^{(k+1)}$ has no bridges. We show that $G^{(k)}$ has no bridges. The lemma then follows by induction.

From Lemma 12 we have $d_k^{(k+1)}(x) = d_k^{(k)}(x)$ for all x. Let (v,w) be a tree arc of $G^{(k)}$. By hypothesis the tree arc entering w in $G^{(k+1)}$ is not a bridge; we wish to show that (v,w) is not a bridge in $G^{(k)}$. Two cases arise from Lemma 7.

(i) Vertex w has an entering forward or cross arc in $G^{(k+1)}$. Then w has an entering cross or forward arc in $G^{(k)}$, and (v,w) is not a bridge by Lemma 7.

(ii) Vertex w has an entering cycle arc (y,w) with y not dominated by w in $G^{(k+1)}$. Then $w \leq k$.

a) If $w = k$, let $(1,v_2),(v_2,v_3),\ldots(v_{n-1},y)$ be a path from 1 to y in $G^{(k+1)}$ which doesn't contain k. Let (v_i,v_{i+1}) be the last edge on this path with v_i not a descendant of k in the DFST of $G^{(k+1)}$. Then $v_{i+1} \in P(k)$, so (v_i,w) is a forward or cross arc of $G^{(k)}$, and (v,w) is not a bridge.

b) If $w < k$ and $y \notin P(k)$, $d^{(k)}_{w+1}(y) = d^{(k+1)}_{w+1}(y) < w$ by Lemma 12 and the fact that w doesn't dominate y in $G^{(k+1)}$. Thus w doesn't dominate y in $G^{(k)}$ and (v,w) is not a bridge.

c) If $w < k$ and $y \in P(k)$, (k,w) is a cycle arc of $G^{(k)}$, and $d^{(k)}_{w+1}(k) = d^{(k+1)}_{w+1}(k) = d^{(k+1)}_{w+1}(y) < w$, by Lemma 12, the fact that $P(k) \cup \{k\}$ induces a strongly connected subgraph of $G^{(k+1)}$, and the fact that w doesn't dominate y in $G^{(k+1)}$. It follows that (v,w) is not a bridge.

Thus $G^{(k)}$ contains no bridges.

Q.E.D.

Now we have a systematic way to collapse the bridgeless flow graph $G = G^{(V+1)}$ into an acyclic bridgeless flow graph $G^{(2)}$. We need to find two disjoint spanning trees of $G^{(2)}$ and to systematically expand them to give two disjoint spanning trees of G.

For any edge (v,w) in $G^{(2)}$ let $h(v,w) = 0$. For any edge (v,w) in $G^{(k+1)}$, let $h(v,w) = k$ if $v \in P(k) \cup \{k\}$ and $w \in P(k) \cup \{k\}$. Otherwise let $h(v,w) = h(v',w')$, where (v',w') is the edge in $G^{(k)}$ corresponding to (v,w). According to this inductive definition, $h(v,w)$ is the largest vertex into which both v and w are collapsed when forming $G^{(V+1)}, G^{(V)}, \ldots, G^{(2)}$; if v and w are never collapsed together, $h(v,w) = 0$. The value $h(v,w)$ is defined for all edges (v,w) in all graphs $G^{(k)}$, $k = 2,3,\ldots,V+1$.

Since $G^{(2)}$ has no bridges, each vertex except 1 in $G^{(2)}$ has at least two entering edges. Let $T^{(2)}_1 = T^{(2)}$ ($T^{(2)}$ is the DFS tree of $G^{(2)}$) and let $T^{(2)}_2$ be any subgraph of $G^{(2)} - T^{(2)}_1$ containing exactly one arc entering each vertex except vertex 1.

For some k, $2 \leq k \leq V$, suppose that $T_1^{(k)}$ and $T_2^{(k)}$ have been defined. $T_1^{(k)}$ and $T_2^{(k)}$ will be edge-disjoint subgraphs of $G^{(k)}$ which together contain all the $T^{(k)}$-arcs of $G^{(k)}$. Without loss of generality, suppose $T_1^{(k)}$ contains the $T^{(k)}$-arc entering k.

Let $(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},k)$ be a simple path in $G^{(k+1)}$ such that

(i) $v_1 \notin P(k)$ and $v_j \in P(k)$ for $j = 2,3,\ldots,n-1$;

(ii) either (v_1,v_2) corresponds to an edge of $T_2^{(k)}$ or (v_1,v_2) is a non-$T^{(k+1)}$-arc of $G^{(k+1)}$ such that the $T_2^{(k)}$-arc entering $h(v_1,v_2)$ is not a cycle arc; and,

(iii) for all $j = 3,4,\ldots,n-1$, there is a non-$T^{(k+1)}$-arc (x,v_j) of $G^{(k+1)}$ such that either $x \in P(k) \cup \{k\}$ or the $T_1^{(k)}$ arc entering $h(x,v_j)$ is not a cycle arc.

There must be such a path since there is an edge (x,y) in $G^{(k+1)}$ with $x \notin P(k) \cup \{k\}$, $y \in P(k) \cup \{k\}$, corresponding to the $T_2^{(k)}$-arc entering k; and there is a simple path from y to k in $G^{(k+1)}$ which contains only vertices in $P(k) \cup \{k\}$. Some final part of this path plus some initial edge (v_1,v_2) must satisfy (i), (ii), and (iii).

Let $T_1^{(k+1)}$ and $T_2^{(k+1)}$ be defined as follows:

For $i = 1,2$ let $T_i^{(k+1)}$ contain all arcs in $G^{(k+1)}$ corresponding to arcs in $T_i^{(k)}$.

Let $(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},k)$ be in $T_2^{(k+1)}$. (If (v_1,v_2) corresponds to an arc of $T_2^{(k)}$, it is already in $T_2^{(k+1)}$.)

For each vertex w in P(k) with an entering arc (x,w) in $T_2^{(k+1)}$, let (y,w) be another entering arc, a $T^{(k+1)}$-arc if possible, such that either $y \in P(k) \cup \{k\}$ or the $T_1(k)$-arc entering h(y,w) is not a cycle arc. Add (y,w) to $T_1^{(k+1)}$.

For each vertex w in P(k) which still has no entering arcs in $T_1^{(k+1)}$ or $T_2^{(k+1)}$, let (x,w) be the entering $T^{(k+1)}$-arc and let (y,w) be any other entering arc. If $y \notin P(k) \cup \{k\}$ and the $T_1^{(k)}$-arc entering h(y,w) is a cycle arc, then add (y,w) to $T_2^{(k+1)}$ and (x,w) to $T_1^{(k+1)}$. Otherwise, add- (x,w) to $T_2^{(k+1)}$ and (y,w) to $T_1^{(k+1)}$.

We need to show that, for all $2 \leq k \leq V+1$, $T_1^{(k)}$ and $T_2^{(k)}$ are edge-disjoint spanning trees of $G^{(k)}$. Clearly $T_1^{(k)}$ and $T_2^{(k)}$ are edge-disjoint subgraphs of $G^{(k)}$. It is easy to show by induction that $T_1^{(k)}$ and $T_2^{(k)}$ each contain exactly one edge entering every vertex of $G^{(k)}$ except vertex 1, and that $T_1^{(k)}$ and $T_2^{(k)}$ together contain all the $T^{(k)}$-arcs of $G^{(k)}$.

Because of the way the $G^{(k)}$'s are constructed, if (v,j) is a cycle arc of $G^{(j)}$, then for all $k > j$ there is a corresponding cycle arc (w,j) of $G^{(k)}$.

Consider-:

(A) Let (x,y) be an edge of $G^{(k)}$ which corresponds to an edge (x',y') of $T_1^{(j+1)} \cup T_2^{(j+1)}$ for some $j < k$ but not to any edge of $T_1^{(j)} \cup T_2^{(j)}$, and such that $x' \notin P(j) \cup \{j\}$. If h(x,y) has an entering cycle arc in $T_i^{(k)}$ for i = 1 or 2, then (x,y) is not in $T_i^{(k)}$.

Lemma 15: For $k = 2,3,\ldots,V+1$, $T_1^{(k)}$ and $T_2^{(k)}$ satisfy property (A) above and are edge-disjoint spanning trees of $G^{(k)}$.

Proof: The lemma is clearly true for $k = 2$ since $G^{(2)}$ is acyclic. Suppose the lemma holds for integers from 2 to k. We prove the lemma for $k+1$. To prove that (A) holds, let (x,y) be an edge of $G^{(k+1)}$ which corresponds to an edge of $T_1^{(j+1)} \cup T_2^{(j+1)}$ for some $j < k+1$ but not to any edge of $T_1^{(j)} \cup T_2^{(j)}$, and such that $x' \notin P(j) \cup \{j\}$. If $j = k$, (A) holds for (x,y) because of the way $T_1^{(k+1)}$ and $T_2^{(k+1)}$ are constructed. If $j < k$, let (x',y') in $G^{(k)}$ correspond to (x,y) in $G^{(k+1)}$. Then $h(x',y') = h(x,y) < k$ and any cycle arc in $T_1^{(k+1)} \cup T_2^{(k+1)}$ entering $h(y,z)$ corresponds to a cycle arc in $T_1^{(k)} \cup T_2^{(k)}$ entering $h(y',z')$, so (A) holds for (x,y) by the induction hypothesis and the way $T_1^{(k+1)}$ and $T_2^{(k+1)}$ are constructed.

Now we must show that $T_1^{(k+1)}$ and $T_2^{(k+1)}$ are spanning trees; that is, that neither $T_1^{(k+1)}$ nor $T_2^{(k+1)}$ contains a cycle. Suppose to the contrary that for some $i \in \{1,2\}$, $T_i^{(k+1)}$ contains a cycle. This cycle must contain sane vertex of $P(k) \cup \{k\}$, since $T_i^{(k)}$ contains no cycles.

Suppose the cycle contains only vertices in $P(k) \cup \{k\}$. Then the cycle must contain a cycle arc entering k, which means the cycle is in $T_2^{(k+1)}$. But every vertex of $G^{(k+1)}$ has only one edge of $T_2^{(k+1)}$ entering it, and there is a path of $T_2^{(k+1)}$-arcs from outside $P(k) \cup \{k\}$ to k. This is impossible, so no $T_i^{(k+1)}$ cycle containing only vertices in $P(k) \cup \{k\}$ can exist.

Suppose the cycle contains one or more vertices outside $P(k) \cup \{k\}$. The cycle must contain a cycle arc (v,w) such that all vertices on the cycle are descendants of w in $T^{(k+1)}$. (This follows from Lemma8 in [21].) Let (x,y) be any edge of the cycle. We will show that in $G^{(w+1)}$ either x and y are collapsed together or there is an edge in $T_i^{(w+1)}$ corresponding to (x,y). Clearly, $h(x,y) \geq w$, since x and y are descendants of w (in $T^{(k+1)}$ and in $T^{(w+1)}$), there is a path from x to y to w in $G^{(k+1)}$ which contains only descendants of w, and some path in $G^{(w+1)}$ corresponds to this path. If x and y are not collapsed together in $G^{(w+1)}$, then $h(x,y) = w$. If in addition (x, y) corresponds to no edge in $T_i^{(w+1)}$, then for some $w+1 \leq j \leq k$, (x, y) must correspond to an edge in $T_i^{(j+1)}$ with $x' \notin P(j) \cup \{j\}$, and to no edge in $T_i^{(j)}$. But this is impossible, since then property (A) would imply that (x,y) is not an edge of $T_i^{(k+1)}$, since a cycle arc entering w is in $T_i^{(k+1)}$.

Thus the cycle of $T_i^{(k+1)}$-arcs corresponds to a cycle of $T_i^{(w+1)}$ arcs, since v and w are not collapsed together in $G^{(w+1)}$. $T_i^{(w+1)}$ has no cycles. Thus $T_i^{(k+1)}$ can have no cycles, and $T_1^{(k+1)}$ and $T_2^{(k+1)}$ are spanning trees. The lemma follows by induction.

Q.E.D.

We now have a very delicate but direct way to construct two edge-disjoint spanning trees in a bridgeless flow graph. We must still find a way to implement this construction so that it is efficient. There are two steps to be implemented. First, we must collapse the graph, calculating $P(V)$, $P(V-1), \ldots, P(2)$ and successively forming $G^{(V+1)}, G^{(V)}, \ldots, G^{(2)}$. During this process we gather enough information about each P(k) to

enable us to later construct the paths necessary to give the spanning trees. Then we must expand the graph, constructing spanning trees for $G^{(2)},G^{(3)},\ldots,G^{(V+1)}$ from the previously gathered information.

The algorithm needs several arrays and other data structures. For each edge (v,w), $h(v,w)$ is the first vertex into which both v and w are collapsed, as defined previously. If v is a vertex, s(v) is the s-number of v, as defined in the section on depth-first search. With each vertex v is associated a p-set with name v, containing all those vertices currently collapsed into v. We use the following operations on p-sets:

PFIND(w) returns the name of the p-set containing vertex w;

PUNION(x,y) adds the elements in p-set x to p-set y, temporarily destroying p-set y;

SPLIT(x,y) undoes the operation PUNION(x,y), if PUNION(x,y) is the most recent PUNION not yet undone.

SPLIT(x,y) is necessary when we begin expanding the graph; we must undo each collapsing operation. The Appendix to this paper describes a way to implement PFIND, PUNION, and SPLIT so that each PFIND requires O(log V) time and each PUNION or SPLIT requires constant time independent of v.

With each vertex v is also associated an s-queue with name v. This s-queue is a priority queue containing each original edge (x,y) corresponding to an edge entering v in the currently collapsed graph. The priority of edge (x,y) in the queue is s(x). We use the following operations on s-queues:

SHIGH(q) returns an edge (x,y) with highest priority in s-queue q ;

SDELETE((x,y),q) deletes edge (x,y) from s-queue q ;

SUNION(q,r) adds all elements in queue q to queue r , destroying queue q .

We order s-queues by s-number for the following reason: Suppose $k \overset{*}{\rightarrow} v$. Then all edges (x,y) such that $k \overset{*}{\rightarrow} x$ will be deleted from s-queue v before edges (x,y) such that $\neg(k \overset{*}{\rightarrow} x)$. This fact facilitates determining the P(k) 's and makes the algorithm's running time linear except for set and priority queue operations.

Each vertex v can be in at most one P(k) . The array p is computed so that p(v) = k iff $v \in P(k)$ .If v is in no P(k) , p(v) = 0 . If $v \neq 1$, T(v) is the $T^{(v+1)}$ -arc entering v . If $v \in P(k)$ for some k, N(v) is an arc in $G^{(V+1)}$ corresponding to a non- $T^{(k+1)}$ -arc entering v in $G^{(k+1)}$. If $v \notin P(k)$ for any k , N(v) is an arc in $G^{(V+1)}$ corresponding to a non- $T^{(2)}$ '-arc entering v in $G^{(2)}$.

Suppose $v \in P(k)$. Then there is some path from v to k through vertices in P(k) . E(v) will be a $G^{(V+1)}$ -edge corresponding to the , first $G^{(k+1)}$ -edge on some such path. That is, some path $(v,v_2),(v_2,v_3),\ldots,(v_{n-1},k)$ in $G^{(k+1)}$ through vertices in P(k) will correspond to edges $E(v),E(v_2),\ldots,E(v_{n-1})$ in $G^{(V+1)}$. E(v) is necessary to calculate the-paths used in constructing the edge-disjoint spanning trees.

Step (5), appearing below in Algal-like notation, collapses $G = G^{(V+1)}$ into $G^{(V)}$, $G^{(V-1)}$, ..., $G^{(2)}$ .It calculates the sets P(k) , in addition to various data items described above. It uses as a procedure SEARCH, which is a recursively programmed depth-first search for exploring any particular P(k) .

```
(5)  begin
       procedure SEARCH(k,v); begin
          add v to P(k);
          p(v) :=k;
          (x,y) := SHIGH(v) ;
          if (N(v) = 0) and. (T(v) ≠ (x,y)) then N(v) := (x,y);
          while k →* x do begin
             SDELETE( (x, y) , v) ;
             h(x,y) :=k;
             w := PFIND(x)
             comment  if w has not been reached before, search from w;
             if (p(w) ≠ k) and (w ≠ k) then begin
                E(w) := (x,y);
                SEARCH(k,w);
             end;
             (x,y) := SHIGH(v);
             if (N(v) = 0) and (T(v) ≠ (x,y)) then N(v) := (x,y);
       end end;
       comment  initialization;
       for v := 1 until V do begin
          create a p-set {v} with name v;
          if v ≠ 1 then let T(v) be the tree arc entering v;
          create an s-queue named v containing all arcs (u,v)
             entering v, each with priority s(u);
          N(v) :=0;
          P(v) :=∅;
          p(v) = 0;
       end;
```

```
comment collapsing;
for k:=V step -1 until 2 do begin
   (x,y) := SHIGH(k);
   comment k has at most one entering cycle arc;
   if k ≠* x do begin
      SDELETE((x,y),k);
      h(x,y) :=k;
      comment find P(k);
      SEARCH(k,PFIND(x)) ;
      comment collapse P(k) into k;
      for v ∈ P(k) do begin
         SUNION(v,k);
         PUNION(v,k);
end end end end;
```

Step (6) below takes the information calculated by step (5) and Uses it to construct edge-disjoint spanning trees of $G^{(2)}, G^{(3)} \ldots G^{(V+1)}$. In the Process it undoes the PUNION operations performed in step (5), using operation SPLIT. The list "path" is a list of edges used to find a path from outside P(k) through P(k) to k of the type necessary for the spanning tree construction.

```
(6)  comment compute edge-disjoint spanning trees for G(2)
     for k :=2 until V do if p(k) = 0 then begin
        T1(k) :=T(k)
        (x,y) := SHIGH(k);
```

```
    if (x,y) = T(k) then begin
        SDELETE ( (x, y) , k) ;
        (x,y) := SHIGH(k);
    end;
    T2(k) := (x,y) ;
end;
comment compute edge-disjoint spanning trees for
        G(3),G(4),...,G(V+1) = G;
for k = 2 until V do if P(k) ≠ ∅ then begin
    for v ∈ P(k) do SPLIT(v,k);
    if T1(k) = T(k) then i := 2 else i := 1;
    (x,y) := Ti(k);
    w := PFIND(y);
    Ti(w) := (x,y);
    T3-i(w) := T(w);
    path := ∅;
    while w ≠ k do begin
        add (x,y) to front of path;
        (x,y) := E(w);
        w := PFIND(y);
    end;
    Ti(k) := (x,y);
    let (x,y) be first edge on path;
    delete (x,y) from path;
    while (p(PFIND(x)) = k) and ((h(N(PFIND(y)) = k) or
            (T3-i(h(N(PFIND(y)))) is not a cycle arc)) do begin
        Ti(PFIND(y)) := (x,y);
```

```
        if (x,y) ≠ T(PFIND(y)) then T_{3-i}(PFIND(y)) := T(PFIND(y))
           else T_{3-i}(PFIND(y)) := N(PFIND(y));
        let (x,y) be first edge on path;
        delete (x,y) from path;
     end;
     T_{3-i}(PFIND(y)) := T(PFIND(y));
     if p(PFIND(x)) ≠ k then T_i(PFIND(y)) := (x,y)
        else T_i(PFIND(y)) := N(PFIND(y));
     for v ∈ P(k) do if T_1(v) is undefined then begin
        g (h(N(v)) ≠ k) and T_{3-i}(h(N(v)) is a cycle arc) then begin
           T_i(v) := N(v) ;
           T_{3-i}(v) := T(v);
        end else begin
           T_i(v) := T(v);
           T_{3-i}(v) := N(v);
  end end end;
  comment T_1 and T_2 now give two edge-disjoint spanning trees of G;
```

It is an elementary if tedious exercise to verify that steps (5) and (6) correctly construct two edge-disjoint spanning trees of any bridgeless flow graph with exactly two edges entering each vertex except vertex 1 . It is also easy to show that the algorithm requires O(V) time, plus time for O(V) set operations and O(V) priority queue operations. The set operations require O(V log V) time using the method described in the Appendix and the priority queue operations require O(V log V) time using Crane's method [12].

The total time required to execute steps (2), (4), (5), and (6), which together construct two spanning trees containing only the bridges of an arbitrary flow graph G , is thus O(V log V + E) . The total space required is O(V+E) . Figures 1-6 illustrate the application of this algorithm to a flow graph.

Conclusions

This paper has presented a very simple O(VE) algorithm and a much more sophisticated O(V log V + E) algorithm for finding two spanning trees with fewest common edges in a directed graph. The latter method applied depth-first search, a highly simplified and streamlined version of an efficient dominators algorithm (presented for the first time here), and a systematic cycle-shrinking method. The data structures necessary, disjoint sets and priority queues, are sophisticated but quite easy to implement. The O(V log V + E) algorithm, although more complicated than the O(VE) algorithm, is theoretically better by a factor of V/log v . Computational experience with similar algorithms suggests that the O(V log V + E) algorithm will be competitive with or superior to the O(VE) algorithm for practical problems. Both algorithms can be generalized to find two minimally intersecting spanning trees with possibly different roots.

The depth-first search technique and the data manipulation methods used here are applicable to a variety of other graph problems. An interesting open problem is whether the methods used here (or other methods) can be combined to give an $\sim O(E)$ algorithm for finding two spanning trees with fewest common edges in an undirected graph. Such an algorithm could be used to efficiently solve Shannon switching games and to do "mixed" analysis of electrical networks.

Appendix: Implementation of Reversible Set Unions

Suppose we are initially given n disjoint sets, each a singleton and each with its own name. We wish to implement sequences of operations of three types:

FIND(z) returns the name of the set containing z as an element;

UNION(x,y) adds the elements in set x to set y , temporarily destroying set x ; and,

SPLIT(x,y) splits set y into two parts, one part corresponding to the old set x and the other corresponding to the old set y .

Any **SPLIT(x,y)** operation must follow a **UNION(x,y)** operation and be separated from it only by **FIND's** and paired UNION and SPLIT operations.

To implement these operations, we represent each set as a directed tree. Each vertex in a tree corresponds to an element in a set; a vertex contains the name of the corresponding element, a pointer to its father (if any) in the tree, and a count of its descendants in the tree. In addition, the root of a tree contains the name of the set corresponding to the entire tree.

To carry out FIND(z) , we locate the vertex corresponding to z and follow father pointers to the root of the tree, there finding the name of the set containing z .

To carry out **UNION(x,y)** , we locate the roots corresponding to x and y . If set x has more elements than set y , we combine the trees by making the root corresponding to y a son of the root corresponding to x . Otherwise, we make the root corresponding to y a son of the root corresponding to x . We update the number of descendants

of the new root and change the name in the root to y if necessary. The new edge created corresponds to the UNION operation.

To carry out **SPLIT(x,y)** , we break the edge corresponding to the **UNION(x,y)** operation which precedes **SPLIT(x,y)** . We update the names and numbers of descendants of the new roots as necessary.

Clearly each UNION and each SPLIT operation requires constant time. It is easy to prove by induction that any path in a tree with k vertices created by this algorithm has length $\leq \log k$. (See [20].) Thus each FIND operation requires $O(\log n)$ time.

In the application of this algorithm considered in the text, all the SPLIT operations follow all the UNION operations. In this special case it is possible to devise a slightly faster but much more complicated set union method, based on results in [8]. However, the method presented here is simple and is efficient enough for our purposes.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On finding lowest common ancestors in trees," Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (1973), 253-265.

[2] A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation and Compiling, Vol. II: Compiling, Prentice-Hall, Englewood Cliffs, N. J., 1972.

[3] S. M. Chase, "An implemented graph algorithm for winning Shannon switching games," Comm. ACM 15 (1972), 253-256.

[4] J. Edmonds, "Minimum partition of a matroid into independent subsets," Journal of Research of the Nat. Bur. of Standards, 69B (1965), 67-72.

[5] J. Edmonds, "On Lehman's switching game and a theorem of Tutte and Nash-Williams," Journal of Research of the Nat. Bur. of Standards, 69B (1965), 73-77.

[6] J. Edmonds, "Submodular functions, matroids, and certain polyhedra," Calgary Int. Conf. on Combinatorial Structures and their Applications (1969), Gordon and Breach, New York, 69-87.

[7] J. Edmonds, "Edge-disjoint branchings," Combinatorial Algorithms, R. Rustin (ed.), Algorithmics Press, New York, N. Y. (1972), 91-96.

[8] J. Hopcroft and J. Ullman, "Set merging algorithms," SIAM J. on Comput. 2 (1973), 294-303.

[9] T. Kameda and S. Toida, "Efficient algorithms for determining an extremal tree of a graph," Proc. 14th Annual IEEE Symp. on Switching and Automata Theory (1973).

[10] G. Kishi and Y. Kajitani, "Maximally distant trees and principal partition of a linear graph," IEEE Trans. on Circuit Theory, CT-16 (1969), 323-330.

[11] D. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1968), 315-346.

[12] D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass. (1973), 150-152.

[13] D. Knuth, private communication, 1973.

[14] E. L. Lawler, "Matroid intersection algorithms," Memorandum No. ERL-M333, Electronics Research Laboratory, University of California, Berkeley, California (1971).

[15] A. Lehman, "A solution to the Shannon switching game," SIAM J. Appl. Math. 12 (1964), 687-725.

[16] C. St. J. A. Nash-Williams, "Edge-disjoint spanning trees of finite graphs," J. London Math. Soc. 36 (1961), 445-450.

[17] C. St. J. A. Nash-Williams, "Decomposition of finite graphs into forests," J. London Math. Soc. 39 (1964), 12.

[18] T. Ohtsuki, Y. Ishizaki, and H. Watanabe, "Topological degrees of freedom and mixed analysis of electrical networks," IEEE Trans. on Circuit Theory, CT-17 (1970), 491-499.

[19] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. on Comput. 1 (1972), 146-160.

[20] R. Tarjan, "Efficiency of a good but not linear set union algorithm," J. ACM, to appear.

[21] R. Tarjan, "Finding dominators in directed graphs," SIAM J. on Comput. 3 (1974), 62-89.

[22] R. Tarjan, "Testing flow graph reducibility," JCSS, to appear.

[23] R. Tarjan, "A new algorithm for finding weak components," Information Processing Letters, to appear.

[24] R. Tarjan, "A good algorithm for edge-disjoint branchings," Information Processing Letters, to appear.

[25] R. Tarjan, unpublished notes, Computer Science Division, University of California, Berkeley, California (1974).

[26] W. T. Tutte, "On the problem of decomposing a graph into n connected factors," J. London Math. Soc. 3 (1961), 221-230.

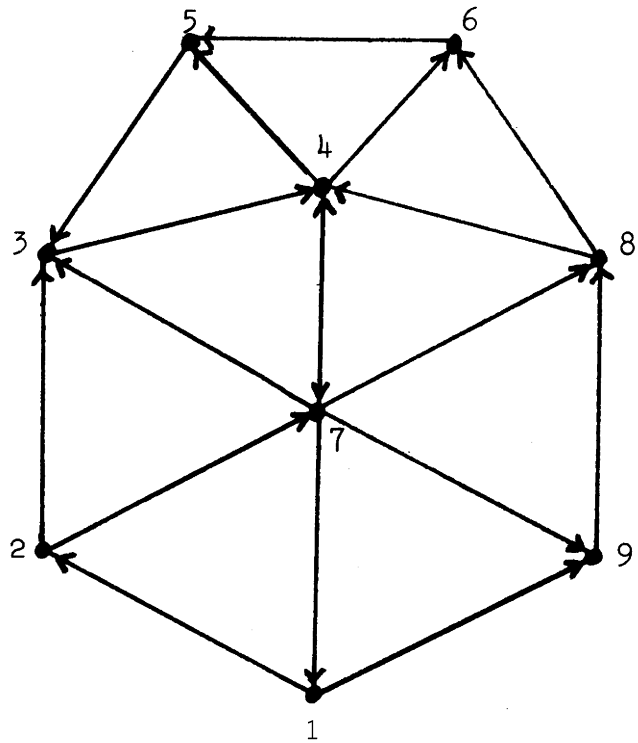


Figure 1: A flow graph, with start vertex 1 . Edge (1,2) is a bridge.

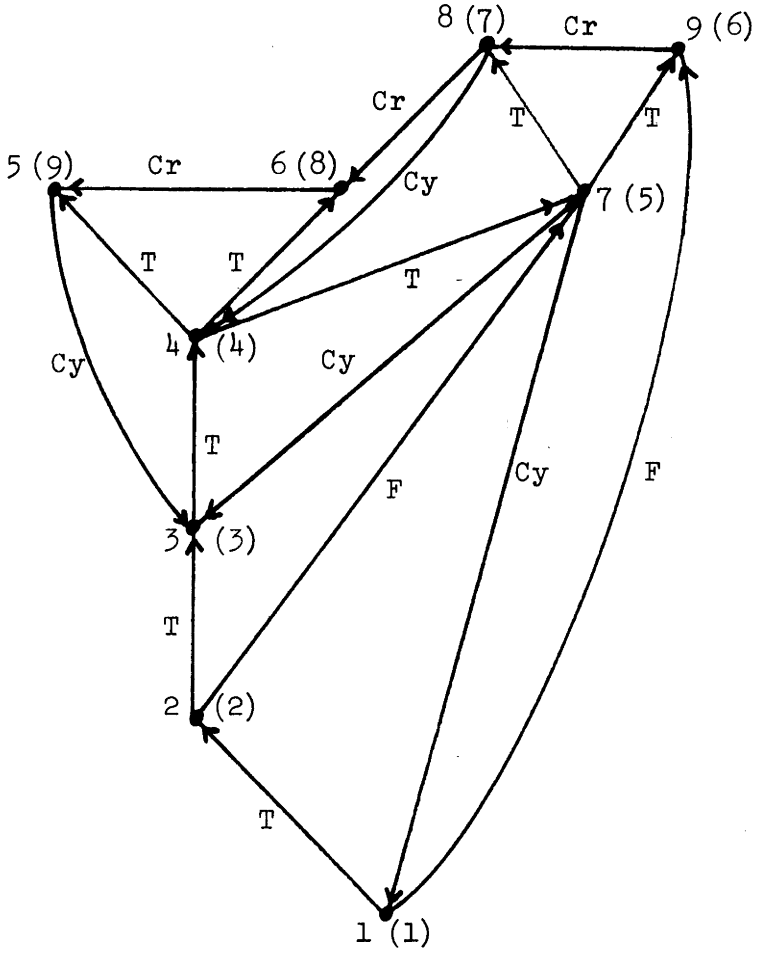
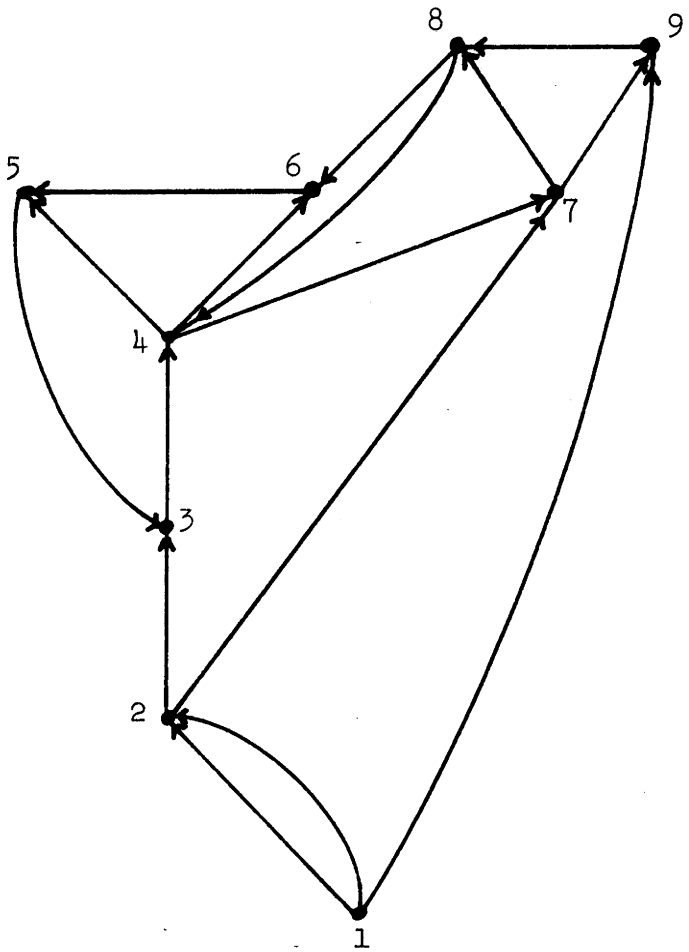


Figure 2: Depth-first search of graph in Figure 1. Tree arcs are marked T , forward arcs F , cycle arcs Cy , and cross arcs Cr . Vertices are numbered in preorder; numbers in parentheses give an s-order numbering.



-Figure 3: Graph after step (2**) applied. Bridge has been duplicated; two cycle arcs remain.

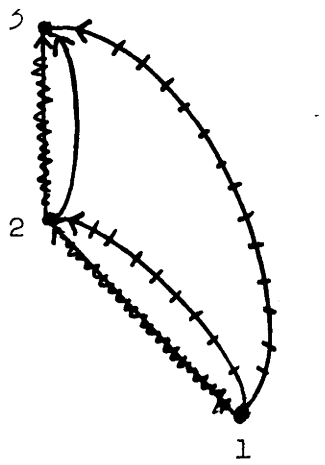


Figure 4: Completely collapsed graph $G^{(3)} = G^{(2)}$ with two edge-disjoint spanning trees, marked by ~~~ and !!!!.

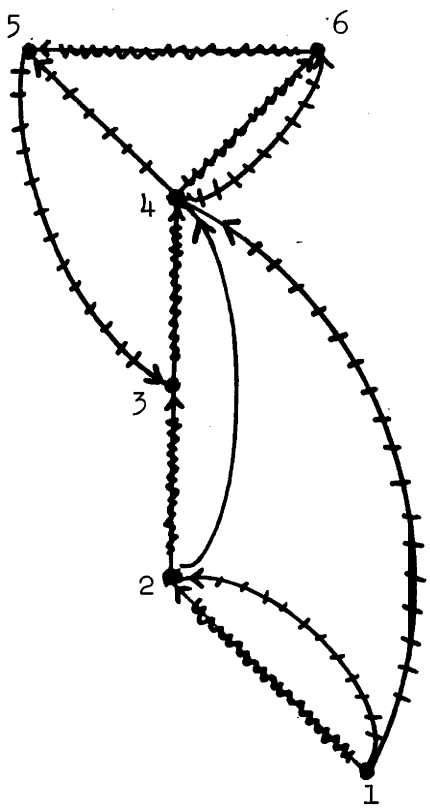


Figure 5: Partially expanded graph $G^{(4)}$ with two edge-disjoint spanning trees.

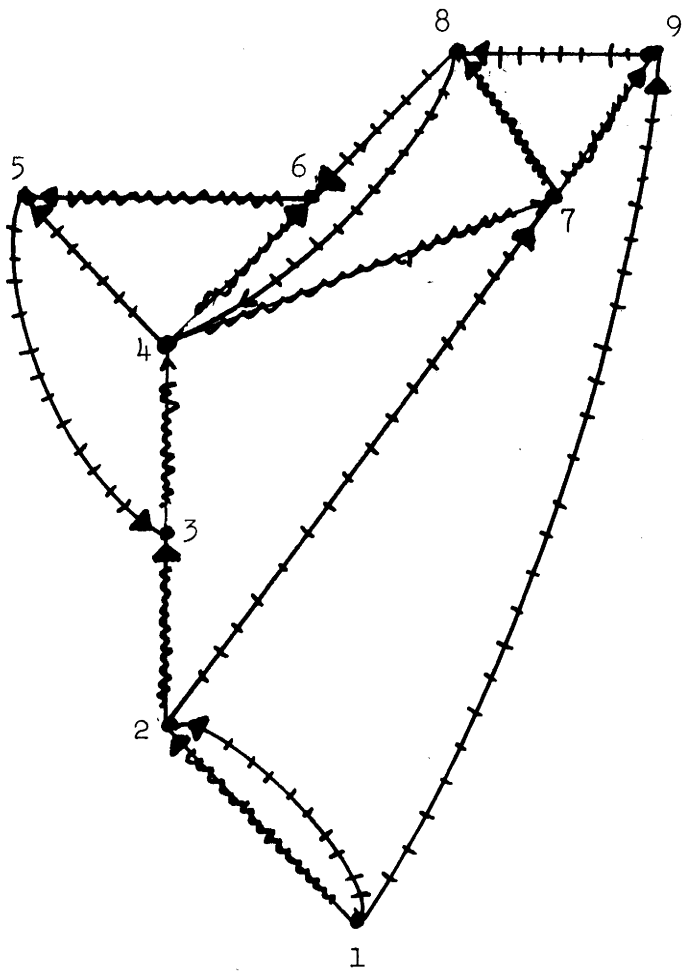


Figure 6: Completely expanded graph $G^{(5)} = G$ with two edge-disjoint spanning trees.