Acta Informatica

Vol. 6 Fasc. 2 1976

Contents

Indexed in Current Contents

cta Informatica
NFA2 6(2) 109—210 (1976)
V. 1976

Springer-Verlag
Berlin · Heidelberg · New York

Acta Informatica 6, 171—185 (1976)

Edge-Disjoint Spanning Trees and Depth-First Search*

Robert Endre Tarjan

Received August 28, 1974

Summary. 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 and an efficient method for computing disjoint set unions. It requires $O(e\alpha(e, n))$ time and $O(e)$ space to analyze a graph with n vertices and e edges, where $\alpha(e, n)$ is a very slowly growing function related to a functional inverse of Ackermann's function.

Definitions

A *graph* $G=(V, E)$ is an ordered pair consisting of a set V of $n=|V|$ *vertices* and a multiset E of $e=|E|$ *edges*. Either each edge of G is an ordered pair (v, w) of distinct vertices (G is a *directed graph*), or each edge is an unordered pair of distinct vertices, also represented as (v, w) (G is an *undirected graph*). (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 $E_2=\{(i, j) \mid (i, j) \in E \text{ and } i, j \in V_2\}$ (all brackets $\{\,\}$ used in this paper denote multisets), 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 nonempty path such that $v_1=v_n$. A vertex w is *reachable* from a 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. An edge (v, w) is a *bridge* of a flow graph if every path from r to w contains (v, w). Figure 1 illustrates a flow graph with a bridge.

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 a tree 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 "$v \to w$ in T" means (v, w) is an edge of T; in this case v is the

* 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 National Science Foundation, Grant MCS 72-03752 A03, at Stanford University.

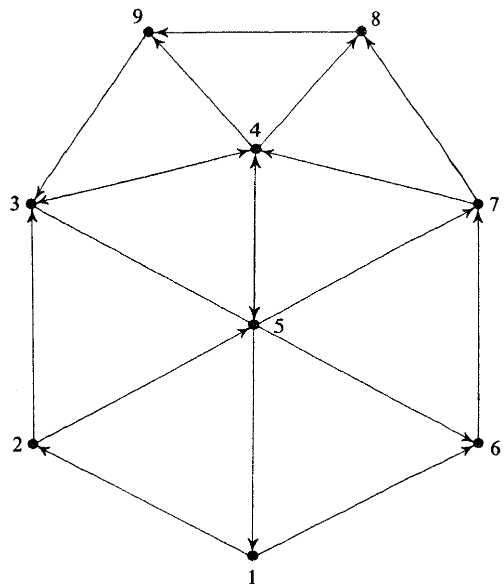

Fig. 1. A flow graph, with start vertex 1. Edge (1, 2) is a bridge

father of w and w is a *son* of v. The notation "$v \xrightarrow{*} w$ in T" means there is a 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 $O(k^2 n^2)$ 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 $O(k^2 e^2)$ 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 by considering the algorithm below, which finds two spanning trees of a directed graph with only the bridges in common.

algorithm *SPAN2*; **begin**
  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 $v \to w$ in T_1 such that $v \in T_2$, $w \notin T_2$, and no descendants x, y of w in T_1 satisfy $x \to y$ 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** add a new copy of edge (v, w) to G;
  comment after step b), w must be reachable from r in $G - T_2 - \{(v, w)\}$ (where only one of the two possible copies of (v, w) is deleted);
  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;
end *SPAN2*;

Lemma 2. Algorithm *SPAN2* finds two spanning trees rooted at r which have only bridges of G in common.

Proof. At least one vertex gets added to the vertex set of T_2 during each execution of the **while** loop in *SPAN2*, since an edge (v, w) can always be added to T_2 at step c). Thus the **while** loop can be executed at most $V-1$ times, and the algorithm terminates after producing two edge-disjoint spanning trees T_1 and T_2. Clearly the algorithm works correctly if the test in step b) fails whenever (v, w) is not a bridge. Suppose (v, w) is not a bridge and the test in step b) is performed on (v, w) 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 execution of step c) immediately preceding this execution of step b). Since $v_{i+1} \notin T_2$, v_i is not a descendant of w in T_1 by the condition in step 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 step b) fails. It follows that *SPAN2* computes two spanning trees with only bridges in common. □

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

One execution of the **while** loop in *SPAN2* clearly requires $O(e)$ time if a set of adjacency lists is used to represent the graph. Thus the whole algorithm requires $O(ne)$ time and $O(e)$ space. We can improve the method's time bound to $O(n^2)$ by first finding a subset of edges partitionable into two disjoint spanning trees and then applying *SPAN2* to the subgraph having only this subset of edges. 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 n vertices numbered from 1 to n in preorder and vertices are identified by number, then $v \overset{*}{\rightarrow} w$ in T iff $v \leqq w < v + ND(v)$.

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* edges;

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

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

Fig. 2 shows a DFS tree of the flow graph in Fig. 1.

A DFS tree is so named because it can be generated by starting at r and carrying out a depth-first search of G. A properly implemented algorithm [19, 21] requires $O(e)$ time to execute the following step.
DFS: Carry out a depth-first search of G, finding a DFS tree, numbering the vertices in preorder to satisfy (iii), calculating $ND(v)$, and finding sets of cycle edges, forward edges, and cross edges.

Henceforth assume that DFS has been applied to flow graph (G, r), that T is the resulting DFS tree, and that vertices are identified by number.

Lemma 4. If $v > w$, any path from v to w contains a common ancester of v and w.

Proof. See [19, 21]. □

If G is acyclic, the numbering defines a topological sorting of the vertices (an ordering such that all edges run from smaller numbered to larger numbered vertices). By examining the vertices of G in order, from largest to smallest, we can compute the strong components [19], the period [13], or the weak components [23] of G, each in $O(e)$ time. By using this ordering in combination with a systematic method for collapsing the graph G, we can find pairs of disjoint spanning trees efficiently.

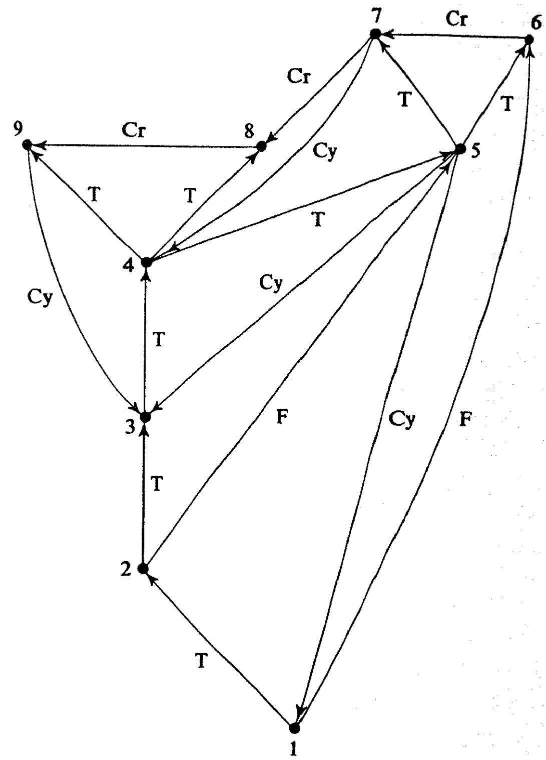

Fig. 2. Depth-first search of graph in Fig. 1. Tree edges are marked T, forward edges F, cycle edges Cy, and cross edges Cr. Vertices are numbered so that all edges but cycle edges run from lower to higher numbers

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

Lemma 5. The subgraph of G induced by the vertices $I(w)$ 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' corespond 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 $I(n)$ in $G=G(n)$ and collapse $I(n)-\{n\}$ into n to create a new graph $G(n-1)$, calculate $I(n-1)$ in $G(n-1)$ and collapse $I(n-1)-\{n-1\}$ into $n-1$ to create $G(n-2)$, and so on, until we reach vertex 1. Eventually we collapse G into an acyclic graph $G(0)$ 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.

Since we are interested in spanning trees rooted at vertex 1 of G, we will assume (without loss of generality) that vertex 1 has no entering edges. For $2 \leqq k \leqq n$, let $I(k)$ be defined in $G(k)$. Let $I(1)$ be the vertex set of $G(1)$ ($G(1)$ must be acyclic). The sets $I(k)-\{k\}$ partition the set $\{i \mid 2 \leqq i \leqq n\}$. We call the sets $I(k)$ *intervals* for G. For $2 \leqq i \leqq n$, let $h(i)=k$ if and only if $i \in I(k)-\{k\}$. The graph $T_I=\{\{1 \leqq i \leqq n\}, \{(h(i), i) \mid 2 \leqq i \leqq n\}\}$ is a tree, called the *interval tree* of G. For any pair of vertices v and w, let $l(v, w)$ be the largest vertex k such that there exist i and j for which $h^i(v)=h^j(w)=k$. That is, $l(v, w)$ is the largest vertex into with both v and w are collapsed when forming $G(n)$, $G(n-1), \ldots, G(1)$; if v and w are never collapsed together, $l(v, w)=1$. The key to finding two spanning trees having only bridges in common is the computation of the intervals of G. This computation will tell us the bridges of G and give us other information useful in constructing two spanning trees. We discuss this computation in the next section.

Computing Intervals and Bridges

To compute intervals, we use a modification of an algorithm for testing flow graphs for reducibility [22]. The idea is to systematically compute the sets $I(w)$ by using a backward search from the vertices in $C(w)$.

To represent the sets $I(w)$ and the collapsed graphs $G(n), G(n-1), \ldots, G(1)$ we use a disjoint set union method discussed in [8, 20]. Initially each vertex v is in a singleton set $\{v\}$ named $\{v\}$. As the algorithm proceeds, v is in a set whose name is the vertex into which v has been collapsed. We need two operations on disjoint sets.

(a) $FIND(v)$: returns the name of the set containing vertex v;

(b) $UNION(v, w)$: adds all vertices in the set named w to the set named v, destroying W.

We need a way to make sure that the backward searches from vertices in $C(w)$ do not extend beyond descendants of w. For any edge (v, w) let $LCA(v, w)$, the *least common ancestor* of v and w, be the vertex x such that $x \xrightarrow{*} v$, $x \xrightarrow{*} w$ in T, and any vertex y satisfying $y \xrightarrow{*} v$, $y \xrightarrow{*} w$ also satisfies $y \xrightarrow{*} x$. We can compute $LCA(v, w)$ for each edge (v, w) by using the algorithm in [1], which uses depth-first search and the set union method of [8, 20]. The LCA algorithm has an $O(e\,\alpha(e, n))$ running time [20], where $\alpha(e, n)$ is a very slowly growing function related to a functional inverse of Ackermann's function.

To restrict the backward searches, we only allow edge (u, v) (or a corresponding edge in the collapsed graph) to be traversed when computing $I(w)$ for $w \leqq LCA(u, v)$. The following algorithm computes $h(k)$ for all $k > 1$ and $I(i)$ for all i.

```
algorithm INTERVALS; begin
   procedure SEARCH(v); begin
      h(v) := i;
      I(i) := I(i) ∪ {v};
      UNION(i, v);
      for (w, v) ∈ E do
         if FIND(w) ≠ i and h(FIND(w)) = 1 then
            SEARCH(FIND(w));
   end SEARCH;
   for i := 1 until n do begin
      create a set {i} named i;
      h(i) := 1;
      I(i) := {i};
   end;

   delete all cross edges and forward edges from E;
   for i := n step -1 until 2 do begin
      for each cross edge or forward edge (v, w) with
         LCA(v, w) = i do add(v, FIND(w)) to E;
      for each cycle edge (v, i) do
         if h(FIND(v)) = 1 then SEARCH(FIND(v));
   end;

   for i := 2 until n do
      if h(i) := 1 then
         I(1) := I(1) ∪ {i};
end INTERVALS;
```

It is easy to see that this procedure correctly computes $h(k)$ and $I(i)$, assuming that the LCA values are given. SEARCH is a recursively programmed depth-first search which carries out the backward searching. The algorithm requires $O(e)$ time plus time for $O(n)$ UNIONs and $O(e)$ FINDs. The set union operations require $O(e\,\alpha(e, n))$ time [20]. Thus the total time for INTERVALS, including time to compute LCA values, is $O(e\,\alpha(e, n))$. The storage space required is $O(e)$.

We find the bridges of G by using the next lemma. For $2 \leqq k \leqq n$, let $(x(k), y(k))$ be a non-tree edge of G with minimum $x(k) < k$ such that for some i, $h^i(y(k)) = k$. If no such non-tree edge exists, let $x(k) = k$.

Lemma 7. Edge (i, k) of G is a bridge if and only if (i, k) is a tree edge and $x(k) = k$.

Proof. Since there is a path of tree edges from vertex 1 to every vertex, every bridge is a tree edge. Let (i, k) be a tree edge such that $x(k) < k$. There is a path from $x(k)$ to k containing only descendants of k. The path of tree edges from vertex 1 to $x(k)$ avoids edge (i, k), since $x(k) < k$ implies $x(k)$ is not a descendant of k. It follows that there is a path from 1 to k which avoids (i, k) and (i, k) is not a bridge. Conversely, if (i, k) is not a bridge, consider any path from 1 to k which avoids (i, k) and let (x, y) be the last edge on this path with $x<k$. Then (x, y) is a non-tree edge, and all vertices following x on the path are descendants of k. It follows that for some i, $h^i(y)=k$. Hence $x(k) \leqq x < k$. □

The following algorithm computes an edge $(x(k), y(k))$ for each $k>1$. Each vertex k with $x(k)=k$ has an entering tree edge which is a bridge. The algorithm duplicates each such bridge and sets $(x(k), y(k))$ equal to the new copy of the bridge. The algorithm also computes, for all edges (v, w), the value of $l(v, w)$ and the edge (v^*, w^*) which corresponds to edge (v, w) in $G(l(v, w))$.

```
algorithm BRIDGES; begin
   for i := 1 until n do begin
      create a set {i} named i;
      x(i) := i;
      list(i) := ∅;
   end;

   for i := n step−1 until 1 do begin
      for (v, w) ∈ E with LCA (v, w) = i do
         add (v, w) to list(FIND (w));
      for (v, i) a non-tree edge do
         if v < x(i) then (x(i), y(i)) := (v, i);
      for u ∈ I(i) − {i} do begin
         for (v, w) ∈ list(u) do (v*, w*) = (FIND (v), FIND (w));
         if x(u) < x(i) then (x(i), y(i)) := (x(u), y(u));
      end;

      for u ∈ I(i) − {i} do UNION(i, u);
      if x(i) = i then begin
         let (v, i) be tree edge entering i;
         comment (v, i) is a bridge;
         add a new copy of (v, i) to G;
         (x(i), y(i)) := (v, i);
end end end BRIDGES;
```

This algorithm requires $O(e\,\alpha(e, n))$ time and $O(e)$ space to compute the desired parameters. Henceforth we assume that G has no bridges and that all the parameters in this section have been computed. In the next section we show how to use these parameters to find two edge-disjoint spanning trees of G. Fig. 3 shows the graph of Fig. 2 with the bridge duplicated and the cycle edge entering vertex 1 deleted.

Computing Two Disjoint Spanning Trees

We need one more set of parameters to construct the edge disjoint spanning trees. For $2 \leqq k \leqq n$, let $(v(k), w(k))$ be a non-tree edge of G such that:

(i) $(v(k), w(k))$ corresponds to an edge $(v'(k), k)$ of $G(k-1)$;

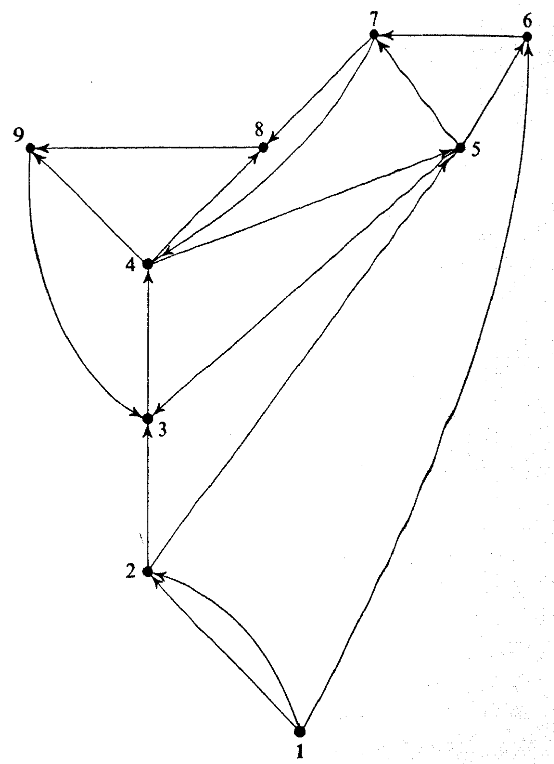

Fig. 3. Graph in Fig. 2 with bridge duplicated, cycle edge entering 1 discarded

(ii) $(v(k), w(k))$ corresponds to an edge $(v'(k), w'(k))$ of $G(k)$ such that $(v(k), w(k)) = (v(w'(k)), w(w'(k)))$; and

(iii) if $w'(k) \neq k$, then in $G(k)$ there is a simple path from $w'(k)$ to k containing only vertices in $I(k)$ and containing only tree edges of $G(k)$, a cycle edge entering k, and edges $(v'(i), i)$ for vertices $i \in I(k)$.

For each vertex v except k on a path from $w'(k)$ to k of the type described in (iii), let $path(v) = k$; for each vertex v not on such a path (except as a last vertex), let $path(v) = 0$. Each vertex can only be on one such path, except as a last vertex. For each k with a non-trivial path of type (iii), let $cycle(k)$ be the edge of G (a cycle edge) corresponding to the last edge on the path.

We use the parameters given in the last section to compute a set of edges $(v(k), w(k))$ and corresponding $path(v)$ and $cycle(k)$ values. Notice that the subgraph of $G(k)$ induced by the vertices $I(k)$ contains exactly the edges (v^*, w^*) such that $l(v, w) = k$. By Lemma 5 each of these induced subgraphs is strongly connected. The following algorithm computes the edges $(v(k), w(k))$.

```
algorithm PATHS; begin
   for i := 2 until n do begin v(i) := cycle(i) := path(i) := 0; end;
   for i∈I(1) do begin
      let (x*, y*) be any non-tree edge in G(l) with y* = i;
      (v(i), w(i)) := (x, y);
   end;
   for k := 2 until n do if I(k) ≠ {k} then begin
   a) if w(k) ≠ k then let w'(k) be such that for some j, h^j(w(k)) = w'(k), and
         h(w'(k)) = k;
      else w'(k) = k;
      (v(w'(k)), w(w'(k))) := (v(k), w(k));
      if w'(k) ≠ k then begin
      b) find a path (x1*, x2*), ..., (xj*, x*j+1) in the subgraph of
         G(k) induced by I(k) such that x1* = w'(k), x*j+1 = k;
         cycle(k) := (xj, xj+1);
         path(x1*) := k;
         for i := 2 until j do begin
            path(xi*) := k;
            if (xi-1, xi) is not a tree edge then
               (v(i), w(i)) := (xi-1, xi);
      end end;
      for i∈I(k) do if v(i) = 0 then
         (v(i), w(i)) := (x(i), y(i));
end end PATHS;
```

This algorithm clearly selects a set of edges $(v(k), w(k))$ satisfying (i), (ii), (iii). PATHS requires $O(e)$ time not counting time spent in steps a) and b). Step a) is executed at most once for each k. Such an execution requires $O(1)$ time for each vertex in $I(k)-\{k\}$, if the test of Lemma 3 is used to determine whether $\exists j$ for which $h^j(w(k)) = w'(k)$; i.e., whether $w'(k) \xrightarrow{*} w(k)$ in T_I. Thus the total time spent in step a) is $O(n)$. Execution of step b) for some k requires time proportional to the number of vertices and edges in the subgraph of $G(k)$ induced by $I(k)$ [19]. Thus the total time spent in step b) is $O(e)$. Combining, PATHS requires $O(e)$ total time (and space).

The following algorithm uses the previously calculated parameters to find two edge-disjoint spanning trees of G. The algorithm works as follows. Step a) selects edges to define two edge-disjoint spanning trees of $G(1)$. Execution of loop b) for a particular value of k expands the previously found edge-disjoint spanning trees of $G(k-1)$ into edge-disjoint spanning trees of $G(k)$. The rather delicate construction in loop b) guarantees that no cycles are introduced.

```
algorithm FASTSPAN2; begin
      T1 := T2 := ∅;
   a) for i∈I(1)-{1} do begin
         add the tree edge entering i to T1;
         add (v(i), w(i)) to T2;
      end;
```

b) **for** $k := 2$ **until** n **do if** $I(k) \neq \{k\}$ **then begin**
if the tree edge entering k is in T_1 **then** $j := 1$ **else** $j := 2$;
if $cycle(k) \neq 0$ **then** add $cycle(k)$ to T_{3-j};
for $i \in I(k) - \{k\}$ **do**
if $(v(i), w(i)) \in T_{3-j}$ **then**
add the tree edge entering i to T_j
else if a cycle edge in T_j enters $l(v(i), w(i))$ **or**
$(v(i)^*, w(i)^*)$ has PATH $(v(i)^*) = k$ **then begin**
add the tree edge entering i to T_j;
add $(v(i), w(i))$ to T_{3-j};
end
else begin
add the tree edge entering i to T_{3-j};
add $(v(i), w(i))$ to T_j;
end end end FASTSPAN2;

This algorithm runs in $O(n)$ time. To prove that the edge sets T_1 and T_2 constructed by *FASTSPAN*2 define edge-disjoint spanning trees of G, let $T_j(1)$ for $j \in \{1, 2\}$ be the subgraph of $G(1)$ whose edges correspond to edges added to T_j by step a). For $k = 2, 3, \ldots, n$ let $T_j(k)$ for $j \in \{1, 2\}$ be the subgraph of $G(k)$ whose edges correspond to edges added to T_j by step a) and the first $k-1$ iterations of loop b. Fig. 4–6 illustrate the construction for the graph of Fig. 3.

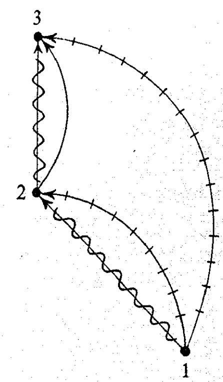

Fig. 4. Completely collapsed graph $G^{(2)} = G^{(1)}$ with two edge-disjoint spanning trees, marked by ~~~ and | | | |

Theorem 1. For $k = 1, 2, \ldots, n$, $T_1(k)$ and $T_2(k)$ are edge disjoint spanning trees of $G(k)$.

Proof. The lemma is clearly true for $k = 1$ since $G(1)$ is acyclic. Suppose the lemma holds for integers from 2 to $k-1$. We prove the lemma for k. By the construction $T_1(k)$ and $T_2(k)$ each have a unique edge entering each vertex $v \neq 1$ of $G(k)$. We must show that neither $T_1(k)$ nor $T_2(k)$ contains a cycle. Suppose to the contrary that for some $j \in \{1, 2\}$, $T_j(k)$ contains a cycle. This cycle must contain some vertex of $I(k)$, since $T_j(k-1)$ contains no cycles.

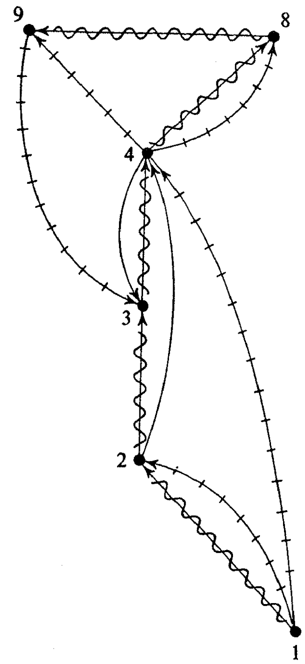

Fig. 5. Partially expanded graph $G^{(3)}$ with two edge-disjoint spanning trees

Suppose the cycle contains only vertices in $I(k)$. Consider the path in $G(k)$ from $w'(k)$ to k containing only vertices of $I(k)$ and containing only tree edges of $G(k)$, a cycle edge entering k, and edges $(v'(i),i)$ for vertices $i \in I(k)$. If all of this path is in $T_j(k)$, then there is a path in $T_j(k)$ from outside $I(k)$ to k. If not all of this path is in $T_j(k)$, let (x,y) be the last edge on the path not in $T_j(k)$.

If (x,y) is not in $T_j(k)$, then by the construction in loop b) (x,y) must be a tree edge, $l(v(y),w(y))$ must be less than k, i.e., $v^*(y) \notin I(k)$, and $(v(y),w(y))$ must be in $T_j(k)$. But this also implies there is a path in $T_j(k)$ from outside $I(k)$ to k. Each vertex of $I(k)$ has only one entering edge in $T_j(k)$. Thus $T_j(k)$ cannot contain a cycle which contains only vertices in $I(k)$, for such a cycle would contain a cycle edge entering k, and some vertex on the cycle would have to contain two entering edges in $T_j(k)$.

Suppose the cycle contains one or more vertices outside $I(k)$. The cycle must contain a cycle edge (v,w) such that all vertices on the cycle are descendants of w in $T(k)$. This follows from Lemma 4. Let (x,y) be any edge of the cycle. We shall show that in $G(w)$ either x and y are collapsed together or there is an edge in $T_j(w)$ corresponding to (x,y). Clearly $l(x,y) \geqq w$, since x and y are descendants of w (in $T(k)$ and in $T(w)$), there is a path from x to y to w in $G(k)$ which contains only descendants of w, and some path in $G(w)$ corresponds to this path.

If x and y are not collapsed together in $G(w)$, then $l(x,y)=w$. If in addition (x,y) corresponds to no edge in $T_j(w)$, then for some $w < i \leqq k$, (x,y) must correspond to an edge (x',y') in $T_j(i)$ with $x' \notin I(i)$, and to no edge in $T_j(i-1)$. This

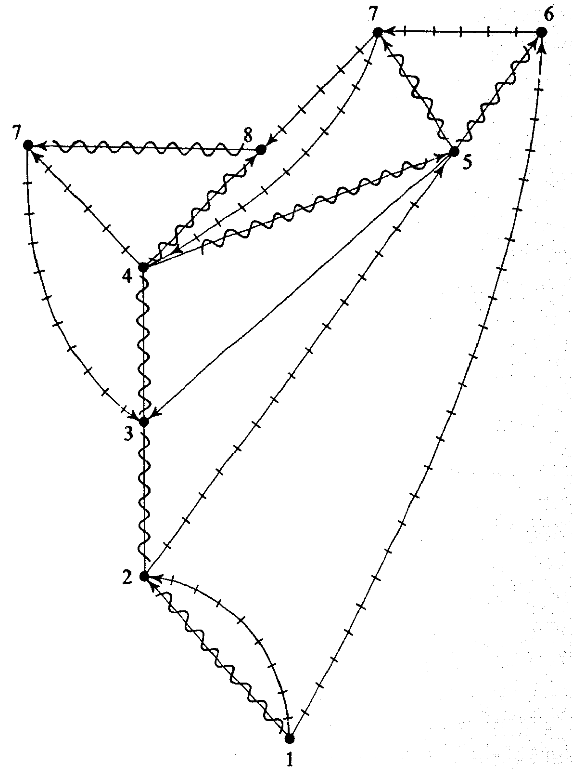

Fig. 6. Completely expanded graph $G^{(4)} = G$ with two edge-disjoint spanning trees

means (x,y) corresponds to an edge added to T_j during the $i-1$ $-$st iteration of loop b. But the fact that the cycle edge entering $l(x,y)=w$ is in T_j guarantees that this edge cannot be added to T_j if the conditions in loop b) are obeyed. Thus (x,y) must correspond to an edge in $T_j(w)$.

Thus the cycle of $T_j(k)$ edges corresponds to a non-empty cycle of $T_j(w)$ edges (v and w are not collapsed together in $G(w)$). But $T_j(w)$ has no cycles. Thus $T_j(k)$ can have no cycles, and $T_1(k)$ and $T_2(k)$ are spanning trees of $G(k)$. □

This completes the description of the edge-disjoint spanning tree algorithm. We summarize the algorithm below.

Step 1: Perform a depth-first search of the problem graph. Determine $LCA(v,w)$ for all edges (v,w).
Time: $O(e\,\alpha(e, n))$.

Step 2: Apply INTERVALS and BRIDGES to compute intervals, bridges, and other necessary parameters. Duplicate all bridges.
Time: $O(e\,\alpha(e, n))$.

Step 3: Apply PATHS to find paths needed for spanning tree construction.
Time: $O(e)$.

Step 4: Build spanning trees using *FASTSPAN2*.
Time: $O(n)$.

The method requires $O(e\,\alpha(e, n))$ total time and $O(e)$ storage space. We have presented PATHS and *FASTSPAN2* separately to clarify the proof of correctness; if the algorithm were to actually be programmed, PATHS and *FASTSPAN2* could be combined, with a corresponding savings of computing time and storage space. It is also possible to combine the INTERVALS and BRIDGES computations. The result is a reasonably clean and simple three-step algorithm which builds a DFS tree of the problem graph, computes certain parameters working from the leaves of the tree toward the root, and then re-examines the tree, in an order dependent on the cycle edges, to compute two edge-disjoint spanning trees.

Conclusions

This paper has presented a simple $O(ne)$ algorithm and a more sophisticated $O(e\,\alpha(e, n))$ algorithm for finding two spanning trees with fewest common edges in a directed graph. Though the $O(e\,\alpha(e,n))$ algorithm uses some powerful techniques, it would be quite easy to program. Computational experience with similar algorithms suggests that the $O(e\,\alpha(e, n))$ algorithm would be competitive with the simple algorithm for small-to-medium-size problems (10—100 vertices) and much faster for large problems (100—1000 vertices). Both algorithms can be generalized to find two minimally intersecting spanning trees with possibly different roots.

The depth-first search technique and the set union algorithm used here are applicable to a variety of other graph problems. Interesting open questions related to this work include:

(1) Do the methods extend to give fast algorithms for finding $k > 2$ edge-disjoint spanning trees in a directed graph?

(2) Do the methods extend to give fast algorithms for finding k edge-disjoint spanning trees in an undirected graph?

A fast algorithm for problem (2) with $k = 2$ could be used to efficiently solve Shannon switching games and do "mixed" analysis of electrical networks.

References

1. Aho, A. V., Hopcroft, J. E., Ullman, J. D.: On finding lowest common ancestors in trees. Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, Austin (Tex.) 1973, p. 253–256
2. Aho, A. V., Ullman, J. D.: The theory of parsing, translation and compiling, Vol. II: Compiling. Englewood Cliffs (N. J.): Prentrice Hall 1972
3. Chase, S. M.: An implemented graph algorithm for winning Shannon switching games. Comm. ACM **15**, 253–256 (1972)
4. Edmonds, J.: Minimum partition of a matroid into independent subsets. J. of Research of the Nat. Bur. of Standards, **69B** 67–72 (1965)
5. Edmonds, J.: On Lehman's switching game and a theorem of Tutte and Nash-Williams. J. of Research of the Nat. Bur. of Standarts, **69B** 73–77 (1965)
6. Edmonds, J.: Submodular functions, matroids, and certain polyhedra. Calgary Int. Conf. on Combinatorial Structures and their Applications 1969. New York: Gordon and Breach 1969, p. 69–87

7. Edmonds, J.: Edge-disjoint branchings. Combinatorial algorithms. In: Rustin, R. (ed.): Combinatorial algorithms. New York (N.Y.): Algorithmics Press 1972, p. 91–96
8. Hopcroft, J., Ullman, J.: Set merging algorithms. SIAM J. Computing **2**, 294–303 (1973)
9. Kameda, T., Toida, S.: Efficient algorithms for determining an extremal tree of a graph. Proc. 14th Annual IEEE Symp. on Switching and Automata Theory 1973
10. Kishi, G., Kajitani, Y.: Maximally distant trees and principal partition of a linear graph. IEEE Trans. on Circuit Theory, CT-16 323–330 (1969)
11. Knuth, D.: The art of computer programming, Vol. 1: Fundamental Algorithms. Reading (Mass.): Addison-Wesley 1968, p. 315–346
12. Knuth, D.: The art of computer programming, Vol. 3: Sorting and Searching. Reading (Mass.): Addison-Wesley 1973, p. 150–152
13. Knuth, D.: Private communication, 1973
14. Lawler, E. L.: Matroid intersection algorithms. Electronics Research Laboratory, University of California, Berkeley, California Memorandum No. ERL—M333, 1971
15. Lehman, A.: A solution to the Shannon switching game. SIAM J. Appl. Math. **12**, 687–725 (1964)
16. Nash-Williams, C. St. J. A.: Edge-disjoint spanning trees of finite graphs. J. London Math. Soc. **36**, 445–450 (1961)
17. Nash-Williams, C. St. J. A.: Decomposition of finite graphs into forests. J. London Math. Soc. **39**, 12 (1964)
18. Ohtsuki, T., Ishizaki, Y., Watanabe, H.: Topological degrees of freedom and mixed analysis of electrical networks. IEEE Trans. on Circuit Theory, CT-17 491–499 (1970)
19. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Computing **1**, 146–160 (1972)
20. Tarjan, R.: Efficiency of a good but not linear set union algorithm. J. ACM **22**, 215–225 (1975)
21. Tarjan, R.: Finding dominators in directed graphs. SIAM J. Computing **3**, 62–89 (1974)
22. Tarjan, R.: Testing flow graph reducibility. J. Computer and System **9**, 355–365 (1974)
23. Tarjan, R.: A new algorithm for finding weak components. Information Processing Letters **3**, 13–15 (1974)
24. Tarjan, R.: A good algorithm for edge-disjoint branchings. Information Processing Letters **3**, 51–53 (1974)
25. Tarjan, R.: Unpublished notes. Computer Science Division, University of California, Berkeley, California, 1974
26. Tutte, W. T.: On the problem of decomposing a graph into n connected factors. J. London Math. Soc. **3**, 221–230 (1961)

Robert Endre Tarjan
Computer Science Department
Stanford University
Stanford, California 94305
U.S.A.