Copyright © 1975, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

FINDING MINIMUM SPANNING TREES

by

R. Endre Tarjan

Memorandum No. ERL-M501

February 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

FINDING MINIMUM SPANNING TREES[†]

R. Endre Tarjan

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley

February 1975

Abstract

This paper gives a method for finding a minimum spanning tree in an undirected graph. If the problem graph has n vertices and e edges, the algorithm runs in $O(e \log \log n)$ time. This time bound is the same as that of a new algorithm by Yao, but Yao's method seems more complicated to implement. A modification of the method improves the running time to $O(e)$, if e is $\Omega(n^{1+\epsilon})$ for some positive constant ϵ. Another algorithm finds a minimum spanning tree of a planar graph in $O(n)$ time. The paper also presents some results which suggest that _any_ method for finding a minimum spanning tree requires $\Omega(e \log \log n)$ comparisons in the worst case.

Keywords: graph algorithm, minimum spanning tree, optimum branching, priority queue

---

[†]Research sponsored by National Science Foundation Grant GJ-35604X1 and by a Miller Research Fellowship.

1. Introduction

Let $G = (V,E)$ be a connected undirected graph with $|V| = n$ vertices and $|E| = e$ edges. Given a value $c(v,w)$ for each edge $(v,w) \in E$, we wish to find a spanning tree $T = (V,E')$, $E' \subseteq E$, such that $\sum_{(v,w)\in E'} c(v,w)$ is minimum. Several efficient algorithms exist for solving this problem [2,4,8,10,11,15]. All of these algorithms are based on the following lemma.

Lemma A. Let $X \subseteq V$. Let $(v,w) \in E$ be such that

$$c(v,w) = \min\{c(x,y) \mid (x,y) \in E,\ x \in X \text{ and } y \in V-X\} \quad .$$

Then some minimum spanning tree contains (v,w).

The "classical" algorithms are those by Kruskal [10] and Prim [4,11]. Kruskal's algorithm has a running time of $O(e \log n)$, and Prim's algorithm has a running time of $O(n^2)$. Recently, Yao [15] has developed an $O(e \log\log n)$ algorithm. His algorithm requires a linear-time median-finding method (e.g. [3]) as a subroutine.

This paper gives a new minimum spanning tree algorithm. The algorithm has an $O(e \log\log n)$ running time. A modification improves the running time to $O(e)$ if e is $\Omega(n^{1+\epsilon})$ for some positive constant ϵ. The algorithm uses as subroutines methods for handling disjoint set unions [6,13] and priority queues [9]. Since the algorithm does not require a linear-time median-finding algorithm, it is simpler to implement than Yao's method. The paper also presents some results which suggest that any method to find minimum spanning trees must require $\Omega(e \log\log n)$ comparisons in the worst case, and gives an algorithm for finding a minimum spanning tree in a planar graph in $O(n)$ time.

2. An Algorithm for Minimum Spanning Trees

To find a minimum spanning tree in a graph G, we use the following general method, which is based on Lemma A.

> First Step: Pick some vertex. Choose the smallest edge incident to the vertex. This is the first edge of the minimum spanning tree.
>
> General Step: The edges so far selected define a forest (set of trees) which is a subgraph of G. Pick a tree in the forest. Pick the smallest unexamined edge incident to a vertex in the tree. If this edge connects two vertices in the same tree, discard the edge. If the edge connects two vertices in different trees, add the edge to the minimum spanning tree (updating the forest). Repeat the general step until all vertices are connected.

It is immediate from Lemma A that this general method correctly finds a minimum spanning tree. The method requires certain bookkeeping mechanisms. To keep track of the vertices in each tree of the forest, we use a disjoint set union algorithm described in [6,13]. Given a collection of disjoint sets (in this case sets of the vertices in each tree) the set union algorithm implements two operations:

(i) FIND(x) returns the name of the set containing element x;

(ii) UNION(A,B) adds the elements in set B to set A, destroying set B.

The time required for $O(e)$ FIND's and $O(n)$ UNION's is $O(n \log^* n + e)$ [13], where $\log^* n = \min\{i \mid \underbrace{\log \log \cdots \log}_{i \text{ times}} n \leq 1\}$.

To keep track of the edges incident to each tree of the forest, we use a mechanism based on a method of handling priority queues [9]. Given a collection of elements, each with a value, and a collection of disjoint sets (called queues) of elements, the mechanism implements three operations:

(iii) QUNION(C,D) adds the elements in queue D to queue C, destroying queue D.

Time required: $O(1)$.

(iv) MIN(C,p) returns the smallest element in queue C satisfying condition p, deleting this element and all smaller elements from queue C.

Time required: $O((m+\ell)\log(\frac{|C|}{m+\ell}+1))$ plus time for m tests of condition p, if m elements are deleted from queue C during this MIN operation and ℓ queues have been merged into C before this MIN operation.

(v) INIT(C,L) initializes a queue C to contain all elements in the list L.

Time required: $O(|L|)$

Implementation of priority queues to satisfy these time bounds is discussed in the appendix.

Our algorithm is the following special case of the general method: In the first step, we pick any vertex and choose its minimum incident edge. In the general step, we choose a tree with the smallest number of vertices, select the smallest unexamined edge connecting a tree vertex with a vertex outside the tree, and discard all incident edges smaller than the selected edge.

We need a mechanism to select the tree with the smallest number of vertices. The number of vertices in every tree is between 1 and n, and the number of vertices in a tree never decreases. We use an array tree of size n, where array element tree(i) is a list of pointers to all trees with i vertices. We use a pointer which marches along this array to find the tree with the smallest number of vertices.

An implementation of the algorithm, expressed in Algol-like notation, appears below. The algorithm assumes that the graph G has vertices $V = \{1,2,\ldots,n\}$ and that for each v, $I(v) = \{(v,w) \mid (v,w) \in E\}$ is a list of the edges incident to vertex v.

```
algorithm MINSPAN;
   begin
      Boolean procedure NEW(v);
         NEW := (k ≠ FIND(v));
      for i := 1 until n do
         begin
            INIT(i,I(i));
            tree(i) := the empty list;
            comment size(i) gives the number of vertices in tree i;
            size(i) := 1;
            add i to tree(1);
            initialize a set named i containing i as its only element;
         end;
      pointer := 1;
      while pointer ≤ n/2 do if tree(pointer) = the empty list
         then
            begin
   increase:   pointer := pointer + 1;
            end
         else
            begin
               delete some element k from tree(pointer);
               comment the next test is used to ignore out-of-date entries
                  in the tree array;
               if size(k) = pointer then
                  begin
                     (i,j) := MIN(k,NEW);
                     add edge (i,j) to minimum spanning tree;
                     x := FIND(i);
                     if x = k then x = FIND(j);
                     UNION(x,k);
                     QUNION(x,k);
                     size(x) := size(x) + size(k);
                     size(k) := 0;
                     add x to tree (size(x));
                  end
            end
   end MINSPAN;
```

3. A Worst-Case Time Bound

Algorithm MINSPAN consists of

(a) priority queue operations;

(b) set operations; and

(c) other operations.

It is clear that $O(n+e)$ time is required for type (c) operations. $O(n)$ UNION operations and $O(e)$ FIND operations (at most six per edge) are carried out by MINSPAN, so $O(n \log^* n + e)$ time is required for type (b) operations. Most of the time used by MINSPAN is in type (a) operations.

MINSPAN is implemented so that M(k,NEW) is executed once for each value of k (except one value) in the range $1 \le k \le n$. Let $\underline{number}(k)$ be the number of edges in queue k when MIN(k,NEW) is executed. Let $\underline{number}(k) = 0$ if MIN(k,NEW) is never executed. We need a bound on $\sum_{k=1}^{n} \underline{number}(k)$. Since the number of vertices in a tree at least doubles each time a MIN operation is performed on the corresponding queue, each edge is counted at most $\log n$ times in $\sum_{i=1}^{n} \underline{number}(k)$. Thus $\sum_{i=1}^{n} \underline{number}(k) \le e \log n$.

The time spent in INIT and QUNION is clearly $O(n+e)$. The time required for a call MIN(k,NEW) is $O((m(k)+\ell(k))\log(\frac{\underline{number}(k)}{m(k)+\ell(k)}+1))$, where $m(k)$ is the number of entries deleted from queue k in this call, and $\ell(k)$ is the number of queues merged into queue k before this call. The total time for all calls on MIN is

$$O\left(\sum_{k=1}^{n} (m(k)+\ell(k))\log\left(\frac{\underline{number}(k)}{m(k)+\ell(k)}+1\right)\right) .$$

We can break this sum into two parts: a sum over k such that $m(k)+\ell(k) \le \frac{\underline{number}(k)}{(\log n)^2}$ and a sum over k such that $m(k)+\ell(k) > \frac{\underline{number}(k)}{(\log n)^2}$.

The total is then

$$O\left(\sum_{k=1}^{n} \frac{\text{number}(k)}{\log n} + \sum_{k=1}^{n} \bigl(m(k) + \ell(k)\bigr) \log((\log n)^2 + 1)\right)$$

$$= O\left(\frac{e \log n}{\log n} + (2e + n - 1) 2 \log \log n\right)$$

$$= O(e \log \log n) \ .$$

Thus the total time required by MINSPAN is dominated by the time spent in MIN, which is $O(e \log \log n)$. MINSPAN requires $O(n+e)$ storage.

4. A Modification for Dense Graphs

By adding an extra step to the algorithm, we can get a time bound of $O(e)$ for dense graphs (i.e. graphs for which e is $\Omega(n^{1+\epsilon})$ for some positive constant ϵ). To accomplish this, we add an initialization statement

$$a := \left\lceil \max\{(\log n)^3, \left(\frac{e}{n \log n}\right)^{1/2}\}\right\rceil ;.$$

at the front of MINSPAN and we insert the following block of code after statement increase.

```
cleanup: if pointer = a then begin
            a := ⌈a^{3/2} / log n⌉;
            examine all edges remaining in priority queues;
            discard all edges joining vertices in the same tree;
            while two or more edges connect the same pair of
               trees do discard all but the minimum such edge;
            combine the remaining edges into new priority queues,
               one for each tree;
         end;
```

This cleanup step has the effect of eliminating useless edges. If cleanup is implemented using a search [5,12], one execution of cleanup requires $O(e)$ time.

Let x be the number of times cleanup is executed when the modified version of MINSPAN is applied to a graph G, assuming that $\log n \geq 2$. If $x > 0$, then

$$(\log n)^{3(\frac{3}{2})^x - x} \leq \frac{n}{2} ,$$

which means x is $O(\log \log n)$. Also, if $(\log n)^3 \leq \left(\frac{e}{n \log n}\right)^{1/2}$,

i.e. $n(\log n)^7 \leq e$, then

$$\frac{\left(\frac{e}{n \log n}\right)^{\frac{1}{2}\left(\frac{3}{2}\right)^x}}{(\log n)^x} \leq \frac{n}{2},$$

which means x is $O(\log[\frac{\log n}{\log(\frac{e}{n \log n})}])$.

The time required for all the cleanups is $O(ex)$. The rest of the time required by MINSPAN is dominated by the time spent in MIN. The time spent in MIN is $O\left(\sum_{k=1}^{n} \left(m(k) + \ell(k)\right) \log\left(\frac{\text{number}(k)}{m(k)+\ell(k)} + 1\right)\right)$, which is $O(e \log \log n)$ by the previous argument. If $e \geq n(\log n)^7$, we can get a better bound.

To bound $\sum_{k=1}^{n} \left(m(k)+\ell(k)\right) \log\left(\frac{\text{number}(k)}{m(k)+\ell(k)} + 1\right)$, we divide this sum among the cleanups. If $e \geq n(\log n)^7$,

$\Sigma\{(m(k)+\ell(k)) \mid$ MIN(k,NEW) executed before first clean-up$\}$

is $O\left(n\left(\frac{e}{n \log n}\right)\right) = O\left(\frac{e}{\log n}\right)$.

$\Sigma\{(m(k)+\ell(k)) \mid$ MIN(k,NEW) executed between i^{th}, $i+1^{st}$ clean-ups$\}$

is $O\left(\frac{n}{a} \frac{a}{(\log n)^2}\right) = O\left(\frac{n}{(\log n)^2}\right)$. Thus the time spent in MIN is $O\left(e + \frac{nx}{\log n}\right) = O(e)$ if $e \geq n(\log n)^7$.

It follows that the modified version of MINSPAN requires $O\left(e \log[\frac{\log n}{\log(\frac{e}{n \log n})}]\right)$ time if $e \geq n(\log n)^7$ and $O(e \log \log n)$ time if $e \leq n(\log n)^7$. If e is $\Omega(n^{1+\epsilon})$ for some positive constant ϵ, the modified algorithm runs in $O\left(e \log(\frac{1}{\epsilon})\right)$ time, and the modified algorithm always runs as fast as the unmodified algorithm (to within a constant factor).

5. Toward a Lower Bound

How fast is the best possible minimum spanning tree algorithm? To answer this question, we need a simple definition of an algorithm and a simple measure of complexity. For our purposes, an algorithm will be a comparison tree. Each vertex of the tree represents a comparison between the values of two edges in the problem graph. Depending upon the outcome of the comparison, the next comparison to be made is either the left son or the right son of the previous comparison. We allow a different comparison tree for each possible graph. Given this model, we want to know the minimum number of comparisons required to determine a minimum spanning tree in the worst case.

We first show that, to within a constant factor, the worst case occurs when the problem graph is sparse, in particular when all vertices are of degree three. Let G be any graph. Consider applying the following procedure to G:

```
procedure REGULARIZE(G);
   begin
      while G has a vertex v of degree 1 do
         delete v and its incident edge;
      while G has a vertex v of degree 2 do
         begin
            let (v,w) be the minimum value edge incident to v;
            delete (v,w) and collapse v and w into a single vertex;
         end;
      while G has a vertex v of degree ≥ 4 do
         begin
            create a new vertex w;
            for half of the edges (u,v) in G do
               convert (u,v) to an edge (u,w);
            add an edge (v,w) of value less than that of all other edges;
         end
   end REGULARIZE;
```

Let G' be the graph produced when REGULARIZE is applied to graph G. G' is regular of degree three. REGULARIZE can be implemented to run in $O(n+e)$ time, if G has n vertices and e edges (e.g. see [7]). Furthermore, any minimum spanning tree T' of G' corresponds to a minimum spanning tree T of G. T can be constructed from T' by adding all edges deleted from G by REGULARIZE, deleting all edges added to G by REGULARIZE, and restoring the original endpoints of each edge modified by REGULARIZE. This process also takes $O(n+e)$ time.

G' has at most $2(e-n)$ vertices and $3(e-n)$ vertices. Thus, if $f(e')$ is the time required to find minimum spanning trees of the connected components of any graph regular of degree three with e' edges, and $F(e,n)$ is the time required to find minimum spanning trees of the connected components of <u>any</u> graph with e edges and n vertices, then $F(e,n)$ is $O\left(f(3(e-n))+e\right)$. Obviously $f(x) \geq x$, f is increasing, and $f(x+y) \geq f(x)+f(y)$. Using these facts, we can show that $f(3e) \leq kf(e)$ for some constant factor k. For, given a problem graph with $3e$ edges, partition the edges into nine equal sets, consisting of the smallest ninth of edges, the next smallest ninth, and so on. This can be done by using a linear-time median finding algorithm. Now find the minimum spanning trees of the connected components of the subgraph defined by the first ninth of the edges. Shrink each component to a single vertex, add the second ninth of edges, and find minimum spanning trees. Repeat until all vertices are connected. The total time required is $O(9f(e))$. It follows that $F(e,n)$ is $O(f(e))$.

For the purpose of trying to get a lower bound, we will assume that the values of all the edges are distinct. This guarantees that the minimum spanning tree is unique. As a comparison-type algorithm proceeds, there will be certain edges known to be in the minimum spanning tree, called _included_ edges, certain edges known not to be in the minimum spanning tree, called _excluded_ edges, and other edges, called _unresolved_ edges. The next two lemmas (which extend Lemma A) characterize the moment when an edge becomes resolved.

Lemma 1. An edge (v,w) becomes included exactly when there is a set $X \subseteq V$ such that $v \in V$, $w \in X-V$, the most recent comparison shows (v,w) to have minimum value in $\delta(X) = \{(x,y) \mid (x,y) \in E,\ x \in V$ and $y \in X-V\}$, and all edges in $\delta(X) - \{(v,w)\}$ are unresolved or excluded (just after the most recent comparison).

Proof. Suppose (v,w) is an edge such that a set X exists satisfying the hypotheses of the lemma. Let T be any spanning tree not containing (v,w). Some cycle exists composed of edges of T and (v,w). Some edge on this cycle other than (v,w) is in $\delta(X)$. Deleting this edge from T and adding (v,w) produces a new spanning tree of smaller total value. Thus T is not minimum, and (v,w) must be in any minimum spanning tree.

Conversely, suppose that after some comparison, edge (v,w) becomes included. Choose edge values so that, subject to the constraints of the comparisons made so far, as many edges as possible have values smaller than $c(v,w)$. Let T be a minimum spanning tree in the resultant graph. Removal of (v,w) from T breaks T into two parts. Let X consist of the vertices in one of these parts. Then X must satisfy

the hypotheses of the lemma, since if the value of (v,w) is greater than the value of some edge in $\delta(X)$, T can be modified to have smaller total value by deleting (v,w) and adding an edge in $\delta(X)$. □

Lemma 2. An edge (v,w) becomes excluded exactly when there is a cycle containing (v,w) and edges unresolved or included (just after the most recent comparison) such that the most recent comparison shows that (v,w) is the maximum value edge on this cycle.

Proof. Analogous to Lemma 1. □

We would like to prove a worst-case lower bound of $\Omega(e \log \log n)$ comparisons for finding a minimum spanning tree. Here we show that this bound holds for a certain subclass of comparison algorithms. Consider only algorithms which obey the following rule:

Max: Any unresolved or included edge which is used in a comparison must be a possible maximum among unresolved and included edges.

The following oracle generates a bad case for such a comparison algorithm:

(i) If two excluded edges are compared, the oracle declares either as bigger.

(ii) If an excluded and a non-excluded edge are compared, the oracle declares the excluded one as bigger.

(iii) If two non-excluded edges are compared, the oracle declares the one with more edges known to be smaller as bigger.

Lemma 3. Suppose the oracle above determines the results of comparisons for a comparison algorithm obeying rule Max. Then at all times during the comparison process, any non-excluded edge known to be bigger than k edges must have been directly compared to at least $\log k$ such edges.

Proof. Because of rule Max, rule (ii) of the oracle, and Lemma 2, no excluded edge is ever known to be smaller than any non-excluded edge. Let (v,w) be any non-excluded edge. By rule Max and rule (iii) of the oracle, any comparison which adds to the number of non-excluded edges known to be smaller than (v,w) must be a direct comparison with (v,w) and can at most double the number of edges known to be less than (v,w). The lemma follows. □

By appealing to a result of Tutte, we can use Lemma 3 to give the desired lower bound. The girth of a graph is the length of the shortest cycle in the graph.

Lemma 4 (Tutte [14]). For all n, there is a graph of n vertices, with all vertices of degree three or more, having a girth which is $\Omega(\log n)$.

Lemma 5. Any comparison algorithm obeying rule Max requires $\Omega(e \log \log n)$ comparisons in the worst case.

Proof. Suppose a comparison algorithm obeying rule Max is applied to one of the graphs given by Lemma 4. Any such graph has at least $(3/2)n$ edges. Thus at least $(1/3)e$ edges must be excluded before the minimum spanning tree is determined. For an edge to be

excluded, it must be known to be a maximum over a cycle of non-excluded edges (Lemma 2), but since any such cycle has length $\Omega(n \log n)$, Lemma 3 implies that $\Omega(\log \log n)$ direct comparisons with the excluded edge are required. These $\Omega(\log \log n)$ comparisons are distinct for each excluded edge; thus a total of $\Omega(e \log \log n)$ comparisons are required. □

We can generalize the result in Lemma 5 a little. Suppose we consider an algorithm which uses an arbitrary number of copies of each edge. The algorithm makes comparisons between the values of the various copies, and only excludes an edge (from the spanning tree) when comparisons previously made with a <u>single copy</u> of the edge imply that it is the largest on some cycle. Comparisons are only allowed between possible maxima among the non-excluded edges. Then a proof like that of Lemma 5 shows that $\Omega(e \log \log n)$ comparisons are required in the worst case by such an algorithm.

One might suspect that it is possible to beat $O(e \log \log n)$ for some classes of graphs which contain enough small cycles. Planar graphs form such a class. Any planar graph without degree one or two vertices has a cycle of five edges or less, and we can find a minimum spanning tree of a planar graph in $O(n)$ time using the following algorithm.

```
algorithm PLANARSPAN
  begin
    construct a planar representation of G;
    until minimum spanning tree found do
      begin
        if G has a vertex v of degree one or two do
          begin
            find smallest edge incident to v;
            add edge to minimum spanning tree;
            collapse endpoints of edge into a single vertex;
          end
        else
          begin
            find a face of five edges or less;
            delete largest edge on face;
  end end end PLANARSPAN;
```

Finding a planar representation for G requires $O(e)$ time [5]. Using a suitable representation of G, each iteration of the until loop, including all necessary updating of the graph representation, can be carried out in $O(1)$ time (e.g. see [7]). Thus the total running time of PLANARSPAN is $O(e)$. Since any planar graph has $e \leq 3n-3$, this running time is $O(n)$. Other special classes of graphs may have similar algorithms, though it seems reasonable to conjecture that the $O(e \log \log n)$ bound is not improvable in the general case.

Another step toward a general non-linear lower bound would be to prove a dual to Lemma 5, by using Lemma 1 and an analogy to Lemma 4 for cuts, to show that any algorithm, which, among non-included edges, compares only minima, requires $\Omega(e \log \log n)$ comparisons in the worst case. Most of the algorithms known to be efficient operate by computing minima over cuts, but they sometimes compare non-minimal edges.

Conclusions and Conjectures

This paper has presented $O(e \log \log n)$-worst case algorithm for finding a minimum spanning tree in an undirected graph. A modification to the algorithm improves its running time to $O(e)$ for dense graphs. Another algorithm finds a minimum spanning tree of a planar graph in $O(n)$ time. The paper also shows that all algorithms obeying a certain rule require $\Omega(e \log \log n)$ comparisons in the worst case.

We close with two conjectures:

(i) The algorithm presented in Section 2 runs in $O(e)$ time on the average (thus the worst case is very rare), assuming any reasonable probability measure on graphs.

(ii) Every algorithm for finding minimum spanning trees requires $\Omega(e \log \log n)$ comparisons in the worst case.

Appendix: Implementation of Priority Queues

To implement the priority queue operations, we extend a method discovered by Crane [9]. In Crane's implementation, each queue is represented by a leftist binary tree. (A leftist binary tree is a tree such that, given any vertex v, there is a shortest path from v to a vertex with a missing left or right son, such that this path contains the right son of v.) Each vertex in such a tree represents an element in a queue. The vertices in the tree are heap-ordered (ordered so that the vertex with smallest value is at the root of the tree and any vertex has value smaller than the values of both its sons).

A basic operation on two leftist binary trees is:

MERGE(i,j) combines trees i and j into a single leftist binary heap-ordered tree named i.

The MERGE operation can be carried out by finding, in each tree, a shortest (rightist) path from the root to a missing vertex, merging the two paths into a single path on which the vertices are sorted by value, attaching the remaining subtrees of each original tree to the appropriate vertices on the combined path, and switching left sons and right sons of vertices along the path (if necessary) to make the new tree leftist. To implement this operation, we must store four parameters with each vertex: its value, pointers to its left and right sons, and the length of the shortest path from the vertex to a missing vertex. See [9] for implementation details. Figure 1 illustrates such a MERGE operation. Since a leftist binary tree with $n(i)$ vertices has a leftist path of length at most $\log(n(i)+1)$, the time required for MERGE(i,j) is $O(\log(n(i)) + \log(n(j)) + 1)$.

Here we extend Crane's idea. We represent each priority queue by a binary tree. Some of the vertices in the tree correspond to queue elements, and some of the vertices are dummy vertices which correspond to previous QUNION operations. Each vertex which corresponds to a queue element is the root of a subtree which consists only of queue vertices and is leftist, binary, and heap-ordered. The dummy vertices define a subtree rooted at the root of the entire tree.

Each vertex v requires five associated parameters:

leftson(v), rightson(v): pointers to the left and right son of v;

path(v): the length of the shortest path from v to a missing vertex (only necessary if v is a queue vertex);

c(v): the value of the queue element associated with v (only defined if v is a queue vertex);

q(v): a Boolean variable true if and only if v is a queue vertex.

Here is an implementation of QUNION, MIN, and INIT, using this data structure. To carry out QUNION(i,j), we create a new dummy vertex, make the roots of trees i and j the left and right sons of the new vertex, and mark the new vertex as the root of the new tree i. QUNION clearly requires $O(1)$ time.

We carry out MIN(i,p) in three steps. Suppose there are ℓ dummy vertices in tree i. First, we explore tree i, from the root up, stopping at queue vertices which satisfy condition p. That is, we locate the set of queue elements $\{v|$ v is in queue i, v satisfies p, and no ancestor of v is a queue vertex satisfying p$\}$. We discard all ancestors of such minimal elements. Let m be the number of discarded queue vertices (all the dummy vertices are also discarded).

We are left with $\ell+m+1$ or fewer leftist binary trees, each rooted at one of the minimal elements. We place these trees in a circular queue, merge the first two trees in the queue using MERGE, add the resultant tree to the end of the queue, and repeat until only one tree is left. The root of this tree is the desired element of minimum value satisfying p. We record this element and convert the root of the tree to a dummy vertex.

The overall effect of the MIN(i,p) operation is to discard from tree i all dummy vertices, and all queue vertices up to and including the one of minimum value satisfying p; and to combine the remaining elements into a single leftist binary tree with a single dummy vertex. An implementation for MIN(i,p) is presented below, in Algol-like notation. In the program, SEARCH is a recursively programmed procedure which explores a binary tree to find the minimal queue vertices satisfying property p.

```
procedure MIN(i,p);
   begin
      procedure SEARCH(x);
         if q(x) ∧ p(x) then add tree rooted at x to circ;
         else
            begin
               if leftson(x) ≠ 0 then SEARCH(leftson(x));
               if rightson(x) ≠ 0 then SEARCH(rightson(x));
            end;
      circ := the empty list;
      let r be the root of tree i;
      SEARCH(r);
      while |circ| > 1 do
         begin
            delete first two trees j and k from circ;
            MERGE(j,k);
            add new tree j to end of circ;
         end
      tree left on circ is new tree i;
      let r be the root of this tree;
      MIN := queue element associated with r;
      q(r) := false;
   end MIN;
```

Suppose MIN is applied to queue i, initially containing k vertices of which ℓ are dummies, and that m queue elements are deleted by MIN. The running time of MIN is $O(m+\ell+1)$ plus the time required for m tests of condition p plus the time required to merge $m+\ell+1$ or fewer trees formed from the remaining $k-\ell-m$ elements.

During the merging process, the original trees (together containing all the remaining elements) are merged in pairs, leaving no more than $\lceil \frac{m+\ell+1}{2} \rceil$ trees containing all the elements. These trees in turn are merged in pairs, and the process continues until one tree is left. Let $b = \lceil \log(m+\ell+1) \rceil$. A bound on the total merge time is

$$O\left(\sum_{i=0}^{b} \max\{ \sum_{j=1}^{2^{b-i}} \log(n_j+1) \;\middle|\; \sum_{j=1}^{2^{b-i}} n_j \le k-\ell,\ n_j \ge 0 \text{ for all } j\} \right) .$$

The maximum inside the outer sum is achieved when all the terms in the inner sum are equal, so the merge time is

$$O\left(\sum_{i=0}^{b} 2^{b-i} \log\left(\frac{k}{2^{b-i}} + 1\right) \right) = O\left((m+\ell)\log\left(\frac{k-\ell}{m+\ell} + 1\right)\right) .$$

Note that ℓ, the initial number of dummy vertices in tree i, is at most one plus twice the number of queues merged into queue i between MIN(i,p) instructions. (Here we count all queues merged into queue i through a sequence of QUNION instructions with no intervening MIN instruction.) In the use of priority queues in this paper, only one MIN instruction is performed on each queue, after which the queue is merged into another queue. Thus in this case ℓ is at most one plus twice the number of queues merged directly into queue i.

To carry out INIT(i,L), we interpret each element of L as a leftist binary tree consisting of a single vertex. We place these trees into a circular queue and merge them as in MIN. The bound above reduces to $O(|L|)$ in this case.

Another queue operation (not necessary to the minimum spanning tree algorithm) can be implemented with only a small change in the data structure.

ADD(a,i) adds a constant a to the value of every element in queue i.

Time required: $O(1)$.

To implement this operation, we use a trick described in [1]. We do not store the current value c(v) for each vertex v but instead use two parameters d(v) and f(v). The meaning of these parameters is as follows. Let $u \overset{*}{\to} v$ denote that u is an ancestor of v in a tree ($v \overset{*}{\to} v$ for all v by convention). Then at all times during the execution of the priority queue operations, $c(v) = f(v) + \sum_{u \overset{*}{\to} v} d(u)$ for each queue vertex v.

Using this modified data structure, we can carry out ADD(a,i) by adding a to d(r), if r is the root of tree i. ADD clearly requires $O(1)$ time. We modify QUNION to set f(v) := d(v) := 0 for the newly created dummy vertex v. We modify SEARCH and MERGE to reset f and d values, for each vertex v examined, as follows:

```
f(v) := f(v)+d(v);
d(leftson(v)) := d(leftson(v))+d(v);
d(rightson(v)) := d(rightson(v))+d(v);
d(v) := 0;
```

Thus, as SEARCH and MERGE work their way up through a tree, they adjust the parameters of the vertices v examined so that $d(v) = 0$. (This implies that every vertex examined has $c(v) = f(v)$.) Then the other parts of SEARCH and MERGE will preserve the desired relationship $c(v) = f(v) + \sum_{u \overset{*}{\rightarrow} v} d(u)$ even though they modify the tree. These steps only increase the running times of SEARCH and MERGE by a constant factor.

We modify INIT to set $f(v) := c(v)$, $d(v) := 0$, for each vertex v in the newly created tree. This increases the running time of INIT by only a constant factor.

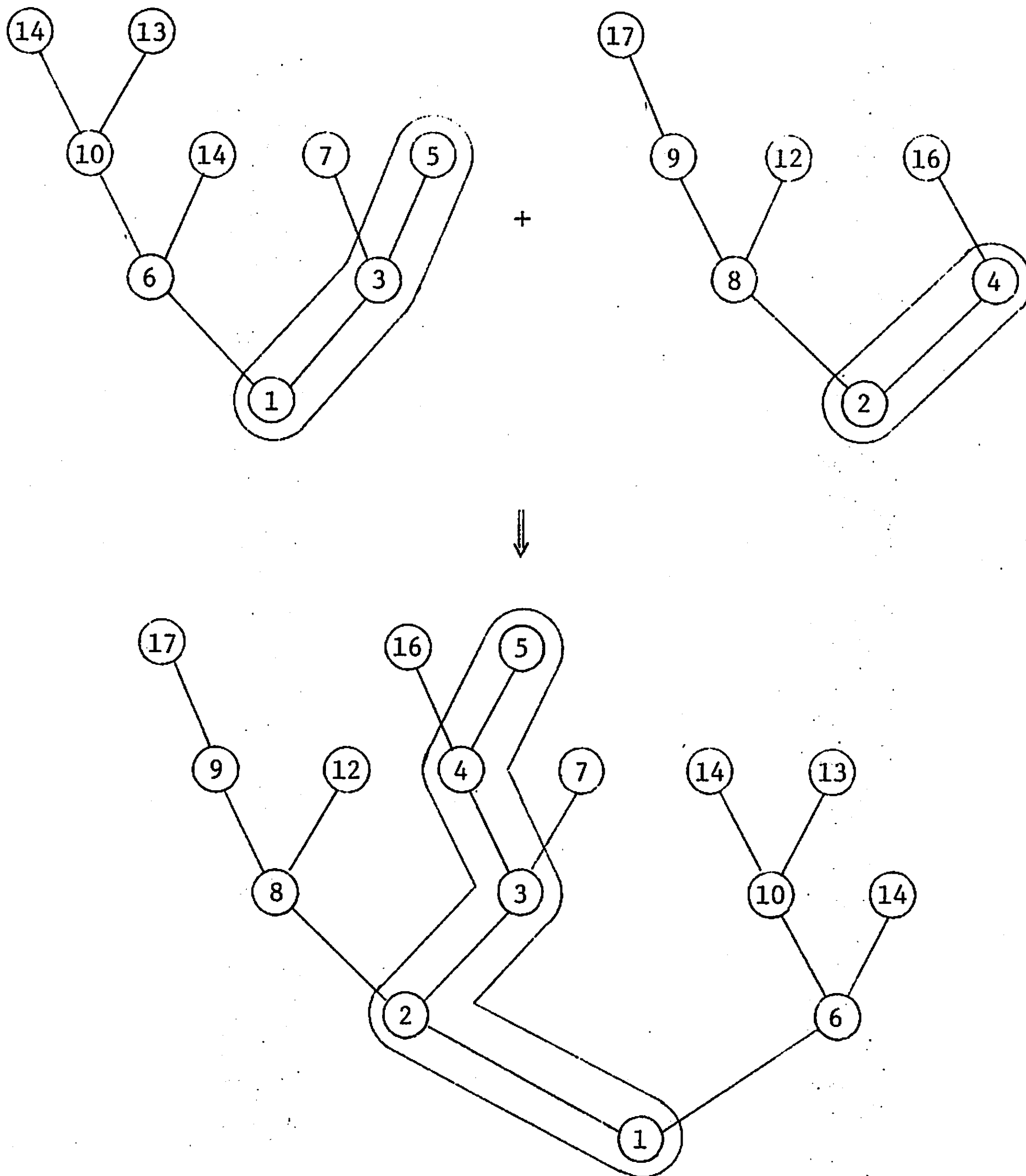

Figure 1: Merging two leftist binary trees.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, "On finding lowest common ancestors in trees," SIAM J. Comput., to appear.

[2] C. Berge and A. Ghouila-Houri, Programming, Games, and Transportation Networks, Wiley (1965), 179.

[3] M. Blum, R. Floyd, V. Pratt, R. Rivest and R. Tarjan, "Time bounds for selection," J. Computer and Sys. Sci., vol. 7, no. 4 (1973), 448-461.

[4] E.W. Dijkstra, "A note on two problems in connection with graphs," Numerische Mathematik, vol. 1 (1959), 269-271.

[5] J. Hopcroft and R. Tarjan, "Efficient planarity testing," J. ACM, vol. 21, no. 4 (1974), 549-568.

[6] J. Hopcroft and J. Ullman, "Set-merging algorithms," SIAM J. Comput., vol. 2, no. 4 (1973), 294-303.

[7] J. Hopcroft and J. Wong, "Linear time algorithm for isomorphism of planar graphs," Proceedings of Sixth Annual ACM Symposium on Theory of Computing (1974), 172-184.

[8] A. Kerschenbaum and R. Van Slyke, "Computing minimum spanning trees efficiently," Proceedings of the 25th Annual Conference of the ACM (1972), 518-527.

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

[10] J.B. Kruskal, Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," Proc. Amer. Math. Soc., vol. 7 (1956), 48-50.

[11] R.C. Prim, "Shortest connection networks and some generalizations," Bell System Tech. J. (1957), 1389-1401.

[12] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput., vol. 1, no. 2 (1972), 146-160.

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

[14] W.T. Tutte, Connectivity in Graphs, Toronto University Press, Toronto (1967), 82.

[15] A. Yao, "An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees," Inf. Proc. Letters, to appear.