SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

FINDING MINIMUM SPANNING TREES*

DAVID CHERITON† AND ROBERT ENDRE TARJAN‡

Abstract. This paper studies methods for finding minimum spanning trees in graphs. Results include

1. several algorithms with $O(m \log\log n)$ worst-case running times, where n is the number of vertices and m is the number of edges in the problem graph;
2. an $O(m)$ worst-case algorithm for dense graphs (those for which m is $\Omega(n^{1+\varepsilon})$ for some positive constant ε);
3. an $O(n)$ worst-case algorithm for planar graphs;
4. relationships with other problems which might lead to a general lower bound for the complexity of the minimum spanning tree problem.

Key words. equivalence algorithm, graph algorithm, minimum spanning tree, priority queue

1. Introduction.

Let $G=(V, E)$ be a connected undirected graph with $|V|=n$ vertices and $|E|=m$ 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 [3, p. 179], [5], [10], [12], [13], [20]. All of these algorithms are based on the following lemma.

LEMMA A. *Let $X\subseteq V$. Let $\{v, w\}\in E$ be such that $v\in X$, $w\in V-X$, and $c(v, w)=\min\{c(x, y)|\{x, y\}\in E,\ x\in X,$ and $y\in V-X\}$. Then some minimum spanning tree contains $\{v, w\}$.*

This paper presents a very general minimum spanning tree algorithm and studies its running time for various implementations. The paper also gives relationships between the minimum spanning tree problem and certain data manipulation problems which may lead to a general lower bound on the complexity of the problem. Section 2 describes the general algorithm, the data manipulation methods needed to implement it, and two "classical" minimum spanning tree algorithms. Section 3 discusses possible implementations for a crucial part of the algorithm. Section 4 proves an $O(m \log\log n)$ worst-case running time for several variations of the algorithm. Section 5 proves an $O(n)$ worst-case running time for planar graphs and an $O(m)$ worst-case running time for dense graphs. Section 6 gives relationships with other problems. Section 7 presents conclusions and open problems.

2. A general algorithm.

To find a minimum spanning tree in a connected graph G, we use the following general method.

First step. Pick some vertex. Choose the smallest edge incident to the vertex. This is the first edge of the minimum spanning tree.

* Received by the editors June 10, 1975, and in revised form November 11, 1975.

† Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. This research was supported by the National Research Council of Canada.

‡ Computer Science Department, Stanford University, Stanford, California 94305. This research was supported by the National Science Foundation under Grant GJ-35604X1 and a Miller Research Fellowship at University of California, Berkeley; and by the National Science Foundation under Grants DCR72-03752 A02 and MCS75-22870 at Stanford University.

General step. The edges so far selected define a forest F (a graph consisting of a set of trees) which is a subgraph of G. (Isolated vertices, not incident to any edges selected so far, are regarded as trees with one vertex.) Pick a tree T in the forest. Select the smallest unselected edge $\{v, w\}$ incident to a vertex in T and to a vertex in a *different* tree T'. Delete all edges smaller than $\{v, w\}$ which are incident to T (these edges form cycles with edges in T). Add $\{v, w\}$ to the minimum spanning tree (updating the forest by combining T and T'). Repeat the general step until all vertices are connected.

Cleanup step. (This optional step may be executed after any general step. It simplifies future calculations by deleting superfluous edges.) Delete all unselected edges with both endpoints in the same tree of the forest. For each pair of trees connected by an unselected edge, delete all but the minimum such edge connecting the pair of trees.

It is immediate from Lemma A that this general method correctly finds a minimum spanning tree. The method requires a mechanism for choosing the tree T to examine and certain bookkeeping mechanisms for keeping track of the trees in the forest. To keep track of the vertices in each tree of the forest, we use a simple, efficient disjoint set union algorithm described in [8] and [15]. Given a collection of disjoint sets (in this case, sets of the vertices in each tree), the set union algorithm implements three operations:

FIND(x): returns the name of the set containing vertex x;

UNION(i, j): combines sets i and j, naming the new set i;

INIT(i, L): initializes set i to contain all vertices in list L.

The worst-case time required for $O(m)$ FINDs and $O(n)$ UNIONs is $O(m\alpha(m, n))$, where α is a functional inverse of Ackermann's function [15]; this time is bounded by $O(n \log^* n + m)$, where

$$\log^* n = \min \{i \mid \overbrace{\log \log \cdots \log}^{i \text{ times}} n \leqq 1\}$$

To keep track of the edges incident to each tree of the forest, we use a mechanism of priority queues. We maintain queues of all edges incident to each tree of the forest. We need three operations on these queues.

QUNION(i, j): combines queues i and j, naming the new queue i;

MIN(i): returns the smallest edge in queue i which has an endpoint outside tree i. MIN(i) also deletes this edge and all smaller edges from queue i. (Note: MIN(i) must use the FIND operation to test whether a given edge has an endpoint outside tree i.)

QINIT(i, L): initializes queue i to contain all edges in the list L.

Implementation of these priority queue operations is discussed in § 3.

An implementation of the general algorithm using these operations as primitives appears below. We assume that the graph G has vertex set $V = \{1, 2, \cdots, n\}$ and that for each vertex v, $I(v) = \{(v, w) \mid \{v, w\} \in E\}$ is a list of the edges incident to v. (Note that (v, w) is an *ordered* pair; the undirected edge $\{v, w\}$ is represented twice; once as (v, w) in $I(v)$ and once as (w, v) in $I(w)$.) Each tree i of the forest is represented by a set i of its vertices and by a queue i of its incident edges. Initially each vertex v is represented by the set $\{v\}$ and by a queue consisting of the edges in $I(v)$. At all times during execution of the algorithm, every edge (v, w) in queue k satisfies FIND$(v) = k$.

```
algorithm MINSPAN;
    begin
        for $i := 1$ until $n$ do
            begin
                INIT($i, \{i\}$);
                QINIT($i, I(i)$);
            end;
        while more than one tree do
            begin
                execute cleanup step if desired;
                pick some tree $k$;
                $(i, j)$ := MIN($k$);
                add edge $(i, j)$ to minimum spanning tree;
                $x$ := FIND($j$);
                UNION($x, k$);
                QUNION($x, k$)
            end
    end MINSPAN;
```

There are two "classical" minimum spanning tree algorithms. One, due to Kruskal [12], works as follows: initially all edges are sorted by value. Then the edges are examined in order, smallest to largest. If an edge, when examined, connects two different trees in the forest of selected edges, it is selected and added to the forest. Otherwise it is deleted. The initial edge sorting eliminates the need for a priority queue mechanism; the running time of Kruskal's algorithm is $O(m \log n)$ for the sorting plus $O(m\alpha(m, n))$ for the necessary set union operations. Thus the total running time is $O(m \log n)$, and if the edges are initially given in sorted order, the running time is $O(m\alpha(m, n))$.

Another algorithm, discovered by Prim [13] and independently by Dijkstra [5], is a version of our general algorithm in which the same tree is considered at each execution of the general step. One priority queue is needed for this tree (all other trees of the forest F consist of single vertices). In the original implementation, this queue is represented as a single array, with one (possibly filled) position for each vertex not in the tree, giving the size of the smallest edge from the tree to that vertex. A general step requires examination of the entire array and thus takes $O(n)$ time. After each general step, a cleanup step is executed; this cleanup step requires only $O(n)$ time because only a single vertex at a time is added to the unique, growing tree. Thus the original implementation of the Prim–Dijkstra algorithm runs in $O(n^2)$ time. If implemented differently, the algorithm runs in $O(m \log n)$ time [10].

To beat $O(m \log n)$, we must use a good priority queue implementation plus a careful selection of the tree to be considered in the general step. Two selection rules lead to efficient algorithms. One, based on a minimum spanning tree algorithm of Sollin [3, p. 189], is to process the trees uniformly. The uniform selection rule works as follows. Initially all of the n trees (each a single vertex) are placed on a queue. At the general step, the first tree T at the front of the queue is examined. When this tree T is connected to another tree T', both T and T' are deleted from the queue and the new combined tree is placed at the rear of the queue. It is easy to implement this selection rule so that the total overhead for selection is $O(n)$.

An alternative selection rule is the *smallest tree rule*: always examine the tree with the smallest number of vertices. This rule can also be implemented so that the total overhead for selection is $O(n)$ (though the constant of proportionality is higher than for the uniform selection rule). This is done by keeping track of the number of vertices in each tree, and by using an array A of size n. The array element $A(i)$ is a list of all trees with i vertices. Since the number of vertices in every tree is between 1 and n, and the number of vertices in a tree never decreases, we can use a pointer which marches along array A to find the tree with the smallest number of vertices. It is easy to update the array if each tree has a pointer to its position in the array.

3. Implementation of priority queues. We consider three implementations of priority queues, ranging from simplest to most sophisticated. Time bounds for the various implementations are given in Table 1.

TABLE 1

Time bounds for priority queue operations.
$m(i)$ = *number of edges in queue* i (*and list* L).
$s(i)$ = *number of sets in queue* i (*implementation* (b)).
$l(i)$ = *number of edges deleted from queue* i *by* MIN(i).
$h(i)$ = *number of delayed merges into queue* i (*implementation* (c)).

	INIT(i, L)	MIN(i)	QUNION(i)
(a) Unordered sets	$O(m(i))$	$O(m(i))+m(i)$ FINDs	$O(1)$
(b) Ordered subsets of size k	$O(m(i)\log k)$	$O(s(i)+l(i))$ $+s(i)+l(i)-1$ FINDs	$O(1)$
(c) Leftist trees with delayed merge	$O(m(i))$	$O\left((l(i)+h(i))\log\left(\frac{m(i)}{l(i)+h(i)}+1\right)\right)$ $+O(l(i)+h(i))$ FINDs	$O(1)$

(a) *Unordered sets.* The simplest representation of a priority queue i is as an unordered list of items. Then MIN(i) requires $O(m(i))$ time plus time for $m(i)$ FINDs, QUNION(i, j) requires $O(1)$ time, and INIT(i, L) requires $O(m(i))$ time, where $m(i)$ is the size of queue i (and of L).

(b) *Ordered subsets of size k.* Another possibility is to represent a queue i as a list of sets, each set initially of size k, plus possibly some "special" sets, each initially of size less than k. Each set is in sorted order, but there is no ordering relationship among the sets. We carry out INIT(i, L) by dividing L into $\lfloor |L|/k \rfloor$ sets of size k plus at most one set of size less than k. We then sort these sets. This process requires $O(|L|\log k)$ time. We carry out QUNION(i, j) by merging the list of sets in queue i and the list of sets in queue j. This requires $O(1)$ time. We carry out MIN(i) by inspecting the edges in each set of queue i in order, discarding those that don't connect two different trees. We then compare the smallest edge from each set and select the overall minimum edge. MIN(i) requires $O(s(i)+l(i))$ time plus time for $s(i)+l(i)-1$ FINDs, where $s(i)$ is the number of sorted sets in queue i and $l(i)$ is the number of edges deleted from queue i by MIN(i).

A similar priority queue method was used by Yao [20] in the first $O(m \log\log n)$ time minimum spanning tree algorithm. However, initialization of Yao's data structure requires the use of a linear-time median-finding algorithm such as [4], which is complicated to implement and requires more comparisons than the sorting used here for initialization.

(c) *Leftist trees with delayed merge.* This representation is an extension of one discovered by Crane [11]. In Crane's implementation, each queue is represented by a leftist tree. (A leftist tree is a binary tree such that, given any node v, there is a shortest path from v to a node with a missing left or right son, such that this path contains the right son of v.) Each node in such a tree represents an edge in the corresponding queue. The nodes are heap-ordered (ordered so that the node with smallest value is at the root of the tree and any node 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 node, merging the two paths into a single path on which the nodes are sorted by value, attaching the remaining subtrees of each original tree to the appropriate nodes on the combined path, and switching left sons and right sons of nodes along the path (if necessary) to make the new tree leftist. To implement this operation, we must store four parameters with each node: its value, pointers to its left and right sons, and the length of the shortest path from the node to a missing node. See [11] for implementation details. Figure 1 illustrates such a MERGE operation. Since a leftist binary tree with $m(i)$ nodes has a rightist path of at most $\log(m(i)+1)$ nodes, the time required for MERGE(i, j) is $O(\log(m(i))+\log(m(j))+1)$. (All logarithms in this paper are base two.)

Here we extend Crane's idea. We represent each priority queue by a binary tree. Some of the nodes in the tree (called *queue nodes*) correspond to edges in the graph. The rest of the nodes (called *dummy nodes*) correspond to previous QUNION operations. Each queue node is the root of a subtree which consists only of queue nodes and is leftist and heap-ordered. The dummy nodes define a subtree rooted at the root of the entire tree.

Each node v requires five associated parameters:

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

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

$c(v)$: the value of the edge associated with v (only defined if v is a queue node);

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

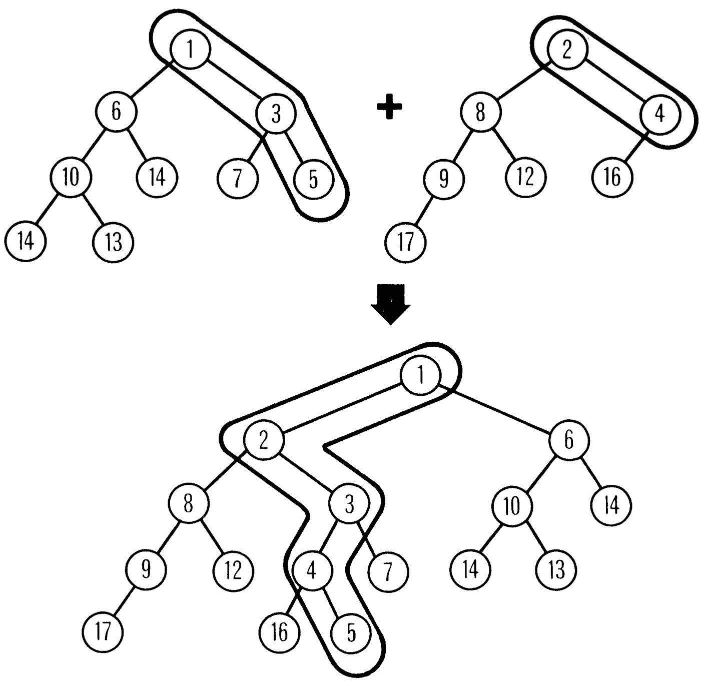

FIG. 1. *Merging two leftist binary trees*

To carry out QUNION(i, j) using this data structure, we create a new dummy node, make the roots of the trees for queues i and j the left and right sons of the new node, and mark the new node as the root of the new tree for queue i. QUNION clearly requires $O(1)$ time. We call this operation *delayed merge* of queue j into queue i.

We carry out MIN(i) in three steps. Suppose there are d dummy nodes in queue i. First, we explore the binary tree for queue i, starting from the root and working through its descendants, stopping at queue nodes which correspond to edges connecting two different trees in the forest. That is, we locate the set of queue nodes $\{v \mid v$ is in queue i, v represents an edge connecting two different trees, and no ancestor of v represents an edge connecting two different trees$\}$. We discard all ancestors of such minimal elements. The number of nodes discarded is $d+l-1$, where l is the total number of edges which will be removed from queue i by the MIN(i) instruction.

We are left with $d+l$ or fewer leftist binary trees, each rooted at one of the minimal nodes. We place these trees in a queue, merge the first two trees in the queue using MERGE, insert the resultant leftist tree at the rear of the queue, and repeat until only one tree is left. The root of this leftist, heap-ordered tree represents the desired edge of minimum value connecting two different trees. We select this edge and convert the root of the leftist tree to a dummy node. (This step has the effect of discarding one more element from queue i.)

The overall effect of the MIN(i) operation is to discard from tree i all dummy nodes and all queue nodes up to and including the one of minimum value connecting two different trees and to combine the remaining nodes into a single leftist tree with a single dummy node, namely the root. An implementation for MIN(i) is presented below, in ALGOL-like notation. The implementation uses a recursively programmed procedure SEARCH, which explores a binary tree to find the minimal queue nodes connecting two different trees.

```
**procedure** MIN($i$);
    **begin**
        **Boolean procedure** $p(x)$;
            **begin**
                let $(v, w)$ be the edge corresponding to queue node $x$;
                $p := (i \neq \text{FIND}(w))$;
            **end**;
        **procedure** SEARCH($x$);
            **comment** we assume that the next **if** statement is implemented
                so that $p(x)$ is evaluated only if $q(x) = \textbf{true}$;
            **if** $q(x) \wedge p(x)$ **then** add tree rooted at $x$ to $L$;
            **else**
                **begin**
                    **if** leftson($x$) $\neq 0$ **then** SEARCH(leftson($x$));
                    **if** rightson($x$) $\neq 0$ **then** SEARCH(rightson($x$));
                **end**;
        $L$ := the empty list;
        let $r$ be the root of tree $i$;
        SEARCH($r$);
        **while** $|L| > 1$ **do**
            **begin**
                delete first two trees $j$ and $k$ from $L$;
                MERGE($j$, $k$);
                insert new tree $j$ at rear of $l$;
            **end**;
        tree remaining on $L$ is new tree $i$;
        let $r$ be the root of this tree;
        MIN := edge associated with $r$;
        $q(r)$ := **false**;
    **end** MIN;
```

Suppose MIN is applied to queue i, initially containing $m(i)$ queue nodes and d dummies, and that l queue nodes are deleted by MIN. The running time of MIN is $O(l+d)$ plus the time required for $2l+d-1$ FIND operations plus the time required to merge $l+d$ or fewer trees formed from $m(i)-l+1 \leqq m(i)$ nodes.

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

$$O\left(\sum_{j=0}^{b} \max \left\{ \sum_{k=1}^{2^{b-j}} \log (n_k+1) \,\middle|\, \sum_{k=1}^{2^{b-j}} n_k \leqq m(i),\, n_k \geqq 0 \text{ for all } k \right\}\right).$$

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

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

Note that d, the initial number of dummy nodes in tree i, is at most one plus twice the number of queues merged into queue i between MIN(i) 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, $d \leqq 2h+1$, where h is the number of queues merged into queue i by delayed merge. Hence the total time for the MIN(i) instruction is

$$O\left((l+2h+1)\log\left(\frac{m(i)}{l+2h+1}+1\right)\right)=O\left((l+h)\log\left(\frac{m(i)}{l+h}+1\right)\right)$$

plus time for $2l+2h$ FINDs.

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

It is worth noting that the delayed merge idea can be used with other priority queue data structures, such as 3-2 trees [1] and S_n trees with a heap ordering [19], to achieve the same time bounds as given above. It is unclear which data structure achieves the best constant factor in the running time.

By using an idea of Aho, Hopcroft and Ullman [2], we can implement the following additional operation on queues:

ADD(i, x): adds a constant of x to the value of every queue node in queue i. To implement this operation, we do not use the value $c(v)$ to encode the true value of queue node v, but instead maintain the values $c(v)$ so that the true value of queue node v is $\sum_{j=1}^{k} c(v_j)$, where $r=v_1, v_2, \cdots, v_k=v$ is the sequence of (dummy and queue) nodes on the path from the root r to v in the leftist tree for queue i. To carry out ADD(i, x), we add x to $c(r)$. Thus ADD(i, x) requires $O(1)$ time. INIT, MIN and QUNION must be modified slightly (this does not change their time bounds). The ADD operation is not useful for finding minimum spanning trees, but it is useful for finding optimum branchings [17].

4. Implementations without cleanup. In this section we analyze the running time of MINSPAN, implemented without cleanups, for various combinations of selection rules and priority queue implementations. Table 2 summarizes the time bounds derived in this section.

MINSPAN is implemented so that after MIN(i) is executed, queue i is merged into some other queue. MIN(i) is executed exactly $n-1$ times, each time for a different value of i. Without loss of generality, assume that the vertices of the graph are numbered so that the ith execution of MIN is MIN(i). Let $m(i)$ be the number of edges in queue i when MIN(i) is executed. For each i, $1 \leqq i \leqq n-1$, let T_i be the tree in F containing vertex i when MIN(i) is executed. Let T_n be the minimum spanning tree constructed by MINSPAN.

TABLE 2

Time bounds for implementations without cleanup

	Uniform selection	Smallest tree selection
(a) Unordered sets	$O(m \log n)$	$O(m \log n)$
(b) Ordered subsets of size 1, $\log \log n$, $\log n$	$O(m \log \log n)$ ($2m \log \log n$ $+O(m \log \log \log n)$ comparisons)	$O(m \log \log n)$ ($2m \log \log n + O(m \log \log \log n)$ comparisons)
(c) Leftist trees with delayed merge	$O(m \log \log n)$	$O(m \log \log n)$

Suppose the uniform selection rule is used. For each tree T which ever occurs in F, we define a *stage number* stage(T) for T by stage$(T) = 0$ if T contains a single vertex, stage(T)= min {stage(T'), stage(T'')} + 1 if T is formed by connecting trees T' and T'' by an edge. It is easy to prove by induction that if stage$(T) = j$, then T has at least 2^j vertices, so there are at most $(\log n)+1$ stages.

LEMMA 1. *With uniform selection, two different trees T_i and T_j with the same state number are vertex disjoint.*

Proof. Suppose trees T_i and T_j share a vertex. Then either $T_i \subseteq T_j$ or $T_j \subseteq T_i$. Without loss of generality, assume $T_i \subseteq T_j$. The trees T initially on the queue used to implement the uniform selection rule have nondecreasing stage numbers. Furthermore, the uniform selection rule preserves this property as the algorithm proceeds. Just before MIN(i) is executed, T_i has a stage number as small as any tree on the queue. Thus if $T_i \neq T_j$, all trees T on the queue which are subtrees of T_j have stage$(T) \geqq$ stage(T_i), which means that stage$(T_j) \geqq$ stage$(T_i) + 1$. Hence $T_i \neq T_j$ implies stage$(T_i) \neq$ stage(T_j). □

COROLLARY 1. *With uniform selection,*

$$\sum_{\text{stage}(T_i)=k} m(i) \leqq 2m \quad \textit{for any constant } k,$$

$$\sum_{i=1}^{n-1} m(i) \leqq 2m \log n.$$

Proof. The proof is immediate from Lemma 1, since each edge is represented twice in the queues, and $0 \leqq \text{stage}(T_i) \leqq \log n - 1$ if $1 \leqq i \leqq n-1$ by the proof of Lemma 1 and the fact that $|T_i| \geqq 2^{\text{stage}(T_i)}$. □

For the smallest tree selection rule, we have similar results, but we must define stage(T) somewhat differently. For smallest tree selection, let stage$(T) = \lfloor \log |T| \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer not greater than x. Clearly there are at most $(\log n)+1$ stages.

LEMMA 2. *With smallest tree selection, two different trees T_i and T_j with the same stage number are vertex disjoint.*

Proof. Suppose T_i and T_j share a vertex. Then $T_i \subseteq T_j$ or $T_j \subseteq T_i$, and without loss of generality we may assume $T_i \subseteq T_j$. After MIN(i) is executed, T_i is connected by an edge to some tree T having $|T| \geqq |T_i|$. Thus if $T_i \neq T_j$, $|T_j| \geqq 2|T|$ and stage$(T_j) \geqq$ stage$(T_i)+1$. □

COROLLARY 2. *With smallest tree selection,*

$$\sum_{\text{stage}(T_i)=k} m(i) \leqq 2m \quad \text{for any constant } k,$$

$$\sum_{i=1}^{n-1} m(i) \leqq 2m \log n.$$

For either selection rule, define the *i-th stage* of the minimum spanning tree algorithm to be the time during which the algorithm performs MIN operations on trees with stage number i. (The ith stage includes time spent combining these trees via UNIONs to form trees with stage number $\geqq i+1$.)

With these preliminaries, we can now bound the running time for the algorithm, assuming no cleanups are performed, for various priority queue implementations. Suppose implementation (a) (unordered lists) is used. Since only $n-1$ UNION operations are ever carried out, and since $\sum_{i=1}^{n-1} m(i) \leqq 2m \log n$, the time for priority queue and other operations is $O(m \log n)$ plus time for $O(m \log n)$ FINDs and $O(n)$ UNIONs. The time for these operations is $O(m \log n)$ [15]. Thus the total time is $O(m \log n)$ (for either the uniform or the smallest tree selection rule).

Suppose implementation (b) (ordered subsets of size k) is used. Suppose we initialize the queues at the beginning of the pth stage of the algorithm and that we use this queue implementation until the qth stage of the algorithm. Since there are no more than $n/2^p$ queues at the beginning of the pth stage, after initialization there are no more than $2m/k + n/2^p$ subsets in all the queues. The queue initialization time is $O(m \log k)$. The time required to execute any stage j is proportional to $m/k + n/2^p + L(j)$, where $L(j)$ is the number of edges deleted from queues during stage j (see Table 1). Thus the total time required to initialize and execute from stage p to stage q is $O([m/k + n/2^p](q-p) + m)$.

Now consider the following implementation of MINSPAN:

Step 1. Initialize all queues as unordered sets.

Step 2. Execute MINSPAN until stage $\log \log \log n$.

Step 3. Re-initialize all queues as lists of ordered subsets of size $k_1 = \log \log n$.

Step 4. Execute MINSPAN until stage $\log \log n$.

Step 5. Re-initialize all queues as lists of ordered subsets of size $k_2 = \log n$.

Step 6. Execute MINSPAN to completion.

The total time required for this process is $O(m + m \log \log \log n + m \log \log n)$ for initialization and re-initialization of queues plus $O(m \log \log \log n + m + m)$ for stage execution (including all UNIONs, FINDs, MINs, and QUNIONs). Thus the total time is $O(m \log \log n)$ (for either selection rule). Furthermore, the running time is asymptotically dominated by the time required for sorting sets of size $\log n$ in Step 5; if all sorting is done using a fast sorting method [11], then this version of MINSPAN requires $2m \log \log n + O(m \log \log \log n)$ comparisons. Yao's $O(m \log \log n)$ minimum spanning tree algorithm requires at least $6m \log \log n$ comparisons if the best median-finding algorithm known is used; Yao's algorithm is also more complicated to implement than ours. The running time of our algorithm is still $O(m \log \log n)$ (though the constant factor may grow) if Steps 2 and 3 are dropped.

Now suppose queue implementation (c) (leftist trees with delayed merge) is used. For each i, let $l(i)$ be the number of edges deleted from queue i during execution of MIN(i), and let $h(i)$ be the number of queues merged into queue i by delayed merge. Clearly $\sum_{i=1}^{n-1} l(i) \leqq 2m$ and $\sum_{i=1}^{n-1} h(i) \leqq n-1$. To estimate the running time of MINSPAN, we must bound

$$\sum_{i=1}^{n-1} (l(i)+h(i)) \log\left(\frac{m(i)}{l(i)+h(i)}+1\right).$$

This will give a bound on the time required for all MIN operations (see Table 1).

We can break this sum into two parts: a sum over i such that $l(i)+h(i) \leqq m(i)/(\log n)^2$ and a sum over i such that $l(i)+h(i) > m(i)/(\log n)^2$. The sum is then bounded by

$$\sum_{i=1}^{n-1} \frac{m(i)}{\log n} + \sum_{i=1}^{n-1} (l(i)+h(i)) \log((\log n)^2+1)$$
$$\leqq 2m + (2m+n-1)2 \log(\log n + 1) = O(m \log\log n).$$

The time for all other operations is also $O(m \log\log n)$. Thus queue implementation (c) (with either selection rule) gives an $O(m \log\log n)$ algorithm.

The time bounds computed in this section are summarized in Table 2. Though queue implementations (b) and (c) give better asymptotic running times than queue implementation (a), other factors, such as ease of implementation, constant factors, and lower order terms in the running time, may govern the best choice of implementation in practice. Experimental tests of these algorithms have yet to be done.

5. Implementations with cleanup.

The cleanup step is useful for graphs in which MINSPAN generates many redundant edges when it combines trees. Two such cases are (i) planar graphs and (ii) dense graphs (graphs for which m/n is large). This section analyzes several versions of MINSPAN with cleanup. Table 3 summarizes the time bounds derived here.

TABLE 3

Time bounds for implementation with cleanup

	Uniform selection	Smallest tree selection
(a) Unordered sets, cleanup every stage	$O(\min(m \log n, n^2))$; $O(n)$ if planar.	$O(\min(m \log n, n^2))$; $O(n)$ if planar.
(b) Ordered subsets		
(c) Leftist trees with delayed merge, cleanup every $[\log a(j)]$th stage		$O\left(m \log\left(\frac{\log n}{\log\left(\frac{m}{n \log n}\right)}\right)\right)$ if $m > n(\log n)^3$; $O(m \log\log n)$ otherwise.

To execute a cleanup, we assign the number i to each vertex in set i. Then we replace each edge $\{v, w\}$ by a corresponding edge whose endpoints are the numbers assigned to v and w. (For each new edge $\{v', w'\}$, we remember the corresponding edge $\{v, w\}$ in the original graph.) We sort the new edges $\{v', w'\}$ lexicographically, using a two-pass radix sort [1]. We delete all new edges $\{v', w'\}$ with $v' = w'$, and we replace multiple copies of an edge $\{v', w'\}$ with $v' \neq w'$ by a single copy whose value is the minimum of the values of all the copies. This entire process requires $O(m)$ time. We then re-initialize the queues and the sets representing the trees F.

The net effect of these operations is to shrink each tree to a single vertex and to delete loops and multiple edges. Each vertex in the resulting new graph corresponds to a tree in the original graph. Subsequently, when selecting an edge for the spanning tree, we use the corresponding original edge $\{v, w\}$ rather than the new edge $\{v', w'\}$.

Suppose queue implementation (a) is used and that a cleanup is performed after each stage. The time for a cleanup is $O(m)$ (including re-initialization time), so the total cleanup time is $O(m \log n)$ and the total running time is $O(m \log n)$, using the bound from § 4.

After stage j, there are at most $n/2^{j+1}$ trees. Thus there are $n/2^{j+1}$ new vertices and $(n/2^{j+1})^2$ edges remaining after the cleanup following stage j. If $m(i)$ is the number of edges in queue i when MIN(i) is executed, then $\sum_{i=1}^{n-1} m(i) \leqq \sum_{j=0}^{\infty} (n/2^j)^2 = O(n^2)$. Thus MINSPAN, with queue implementation (a) and a cleanup after every stage, runs in $O(\min(m \log n, n^2))$ time, and the running time is linear in m for graphs with $m \sim n^2$. This algorithm, with the uniform selection rule, is a version of a method proposed originally by Sollin [3, p. 179].

Suppose the problem graph is planar. Any planar graph has $m \leqq 3n-3$ ($m \leqq 3n-6$ if $n \geqq 3$). After a cleanup, the edges remaining must define a planar graph whose vertices correspond to the subtrees of the minimum spanning tree so far constructed. Thus, for planar graphs, $\sum_{i=1}^{n-1} m(i) \leqq \sum_{j=0}^{\infty} 3n/2^j \leqq 6n$, and MINSPAN with queue implementation (a) and a cleanup after every stage runs in $O(n)$ time on planar graphs. This observation is due to Yao [21].

Now suppose MINSPAN is implemented using queue implementation (c), smallest tree selection, and with a cleanup before the $[\log a(j)]$th stage for $j = 1, \cdots$, where $a(j)$ is recursively defined as follows:

$$a(1) = \left\lceil \max\left\{(\log n)^2, \frac{m}{n \log n}\right\}\right\rceil, \qquad a(j+1) = \left\lceil \frac{a(j)^2}{\log n} \right\rceil,$$

where $\lceil x \rceil$ denotes the smallest integer not less than x. (That is, the jth cleanup is performed when the smallest tree remaining contains at least $a(j)$ vertices, if we ignore the shrinking caused by the cleanups.)

We can bound $a(j)$ from below as follows:

$$a(j) \geqq (\log n)^{f(j)},$$

where $f(j+1) = 2f(j) - 1$, hence $f(j) - 1 = 2^{j-1}(f(1) - 1)$. Thus if C is the total number of cleanups, $C > 0$ implies

$$\frac{n}{2} \geqq (\log n)^{f(C)} = (\log n)^{2^{C-1}(f(1)-1)+1},$$

which means C is

$$O\left(\log\left(\frac{\log n}{(\log\log n)(f(1)-1)}\right)\right).$$

By definition $f(1)\geqq 2$, so C is $O(\log\log n)$. Also, if $(\log n)^2\leqq m/(n\log n)$, i.e., $n(\log n)^3\leqq m$, then

$$f(1)\geqq\frac{\log\left(\dfrac{m}{n\log n}\right)}{\log\log n}$$

and C is

$$O\left(\log\left(\frac{\log n}{\log\left(\dfrac{m}{n\log n}\right)}\right)\right).$$

The time required for all the cleanups is $O(mC)$ (including re-initialization time). The rest of the time required by MINSPAN is dominated by the time spent in MIN. The time spent in MIN is

$$O\left(\sum_{i=1}^{n-1}(l(i)+h(i))\log\left(\frac{m(i)}{l(i)+h(i)}+1\right)\right),$$

where $m(i)$ is the number of edges in queue i when MIN(i) is executed, $l(i)$ is the number of edges deleted from queue i by execution of MIN(i), and $h(i)$ is the number of queues merged into queue i after the last cleanup which occurs before execution of MIN(i). The total time spent in MIN is $O(m\log\log n)$ by § 4. If the graph is dense, we can get a better bound.

Suppose $m\geqq n(\log n)^3$. We bound $\sum_{i=1}^{n}(l(i)+h(i))$ by dividing this sum among the cleanups. Consider a tree T (in the original graph) corresponding to a queue i_0 such that MIN(i_0) is executed between the jth and $(j+1)$st cleanups, and such that T is part of no larger tree corresponding to a queue on which a MIN operation is performed between the jth and $(j+1)$st cleanups. We call such a tree T a *j-tree*. Suppose T is formed from b trees existing at the jth cleanup, using a sequence of $b-1$ MIN operations, say MIN$(i_1),\cdots,$ MIN(i_{b-1}).

Then $\sum_{k=0}^{b-1}l(i_k)\leqq b(b-1)+1$, since after the jth cleanup each of the b trees making up T has at most *one* edge incident with each of the other $b-1$ trees. Also, $\sum_{k=0}^{b-1}h(i_k)\leqq b$. Each of the b trees making up T has at least $a(j)$ vertices, and T contains fewer than $a(j+1)$ vertices, so $b\leqq|T|/(a(j))\leqq a(j+1)/(a(j))$.

Let $T_1,\cdots,T_d$ be all the j-trees. Then $\sum\{l(i)+h(i)|\text{MIN}(i)$ executed between jth, $(j+1)$st cleanups$\}$ is

$$O\left(\sum_{k=1}^{d}\frac{|T_k|^2}{a(j)^2}\right)=O\left(\frac{a(j+1)}{a(j)^2}\sum_{k=1}^{d}|T_k|\right)=O\left(\frac{n}{\log n}\right).$$

A similar argument shows that $\sum\{l(i)+h(i)|\text{MIN}(i)$ executed before first cleanup$\}$ is $O(a(1)\cdot n)=O(m/(\log n))$.

Thus $\sum_{i=1}^{n-1} [l(i)+h(i)] = O(m/(\log n))$ and the time spent in MIN is $O(m)$ if $m > n(\log n)^3$. The total time for this implementation is thus

$$O(mC) = O\left(m \log\left(\frac{\log n}{\log\left(\frac{m}{n \log n}\right)}\right)\right)$$

if $m > n(\log n)^3$. If $m \geqq cn^{1+\varepsilon}$ for some positive constants c and ε, this version of the general algorithm runs in $O(m \log(1/\varepsilon))$ time, and it always runs as fast as the version without cleanups (to within a constant factor).

6. Relationships with other problems. In this section we consider relationships between the minimum spanning tree problem and other problems, which might lead to a general lower bound. We show the following. 1. If the edges of the graph are presented in sorted order, the minimum spanning tree problem is equivalent (to within a constant factor) to a special type of disjoint set manipulation. 2. The worst case for finding a minimum spanning tree occurs when the problem graph is sparse, in particular when all vertices are degree three. 3. If the edges are not given in sorted order, finding a minimum spanning tree on certain graphs of m edges requires computing maximum elements for $\Omega(m)$ sets, each of size $\Omega(\log m)$. This leads to an $\Omega(m \log \log m)$ lower bound for a special class of minimum spanning tree algorithms. (Note: $\Omega(f(m))$ denotes a function which exceeds $cf(m)$ for some positive constant c and infinitely many m.)

Suppose the edges of the problem graph are presented in sorted order, smallest to largest. Consider a family of disjoint sets which partition the set $\{1, 2, \cdots, n\}$. Let the operation EQUIV(v, w) have the effect of combining the set containing v and the set containing w into a single set, and returning the value **true** if v and w were already in the same set, **false** otherwise.

Suppose $\{v_1, w_1\}, \cdots, \{v_m, w_m\}$ are the edges of the problem graph, in sorted order. If we begin with the singleton sets $\{1\}, \{2\}, \cdots, \{n\}$ and execute EQUIV(v_i, w_i) in order for each i, then the edges $\{v_i, w_i\}$ such that EQUIV(v_i, w_i) returns **false** give a minimum spanning tree (this is just an implementation of Kruskal's algorithm). Conversely, given a list of EQUIV(v_i, w_i) operations (each with a different ordered pair $\{v_i, w_i\}$), to be performed on the singleton sets $\{1\}$, $\{2\}, \cdots, \{n\}$, the (unique) minimum spanning tree of the graph with vertex set $\{1, 2, \cdots, n\}$ edge set $\{\{v_i, w_i\}\}$, and edge values $c(v_i, w_i) = i$ contains exactly the edges $\{v_i, w_i\}$ such that EQUIV(v_i, w_i) returns **false.**

Thus finding a minimum spanning tree if the edges are presented in order is equivalent to executing a list of EQUIV instructions. Using the algorithm described in [8] and [15], m EQUIV instructions can be executed in $O(m\alpha(m, n))$ time, where $\alpha(m, n)$ grows very slowly. A nonlinear lower bound on the time to execute a list of EQUIV instructions would give a nonlinear lower bound on the data-manipulation cost (as opposed to the comparison cost) of the minimum spanning tree problem.

To study the comparison cost of the minimum spanning tree problem, we need a simple definition of a comparison algorithm. 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 $c(m)$ required to determine a minimum spanning tree for the worst-case graph G of no more than m edges.

We first show that, ignoring constant factors in running time, there are worst-case graphs which are sparse, in particular regular of degree three. Let G be any graph. Consider applying the following (nondeterministic) procedure to G.

```
procedure REGULARIZE($G$);
    begin
        while $G$ has a vertex $v$ of degree $\leqq 2$ do
            if degree $(v) = 0$ then
                delete $v$;
            else if $G$ has a vertex $v$ of degree 1 do
                delete $v$ and its incident edge;
            else if $G$ 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 $\geqq 4$ do
                begin
                    create a new vertex $w$;
                  for half of the edges $\{u, v\}$ in $G$ do
                        convert $\{u, v\}$ to an edge $\{u, w\}$;
                    create a new edge $\{v, w\}$ of value less than that of all other
                    edges;
                end;
    end REGULARIZE;
```

Let G' be a graph produced when REGULARIZE is applied to a connected graph G having m edges. G' is connected, regular of degree three and has at most $3m$ edges. REGULARIZE can be implemented to run in $O(m)$ time (e.g., see [9]).

A spanning tree for G is characterized by its set of edges. The edges of a minimum spanning tree for G can be found from the edges of a minimum spanning tree for G' by

(i) adding all edges deleted from G by REGULARIZE;
(ii) deleting all edges added to G by REGULARIZE; and
(iii) restoring the original endpoints of each edge converted by REGULARIZE.

This process requires $O(m)$ time.

Let $t(m)$ be the minimum time required to find a minimum spanning tree of any connected graph with no more than m edges. Let $t_3(m)$ be the minimum time required to find a minimum spanning tree of any graph, regular of degree three, with no more than m edges. We desire a bound on $t(m)$ as a function of $t_3(m)$. It follows from the preceding paragraphs that $t(m)$ is $O(t_3(3m)+m)$. But we also know that $k_1 m \leqq t_3(m) \leqq k_2\, m \log\log m$, so $t_3(3m)$ is $O(t_3(m))$, and $t(m)$ is

$O(t_3(m))$. Similarly $c(m)$ is $O(c_3(m))$, if $c_3(m)$ is the minimum number of *comparisons* required to find a minimum spanning tree in a graph regular of degree three with no more than m edges. (Note: $m-1 \leqq c_3(m)$ since finding a minimum spanning tree for a graph which is a cycle is equivalent to finding the maximum edge of the cycle.)

Now we derive a special case lower bound. We shall 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 3. *An edge $\{v, w\}$ becomes included exactly when there is a set $X \subseteq V$ such that $v \in X$, $w \in V-X$, the most recent comparison shows $\{v, w\}$ to have minimum value in $\partial(X) = \{\{x, y\} | \{x, y\} \in E, x \in X$ and $y \in V-X\}$, and all edges in $\partial(X)-\{\{v, w\}\}$ 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 $\partial(X)$. 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 distinct 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 $\partial(X)$, T can be modified to have smaller total value by deleting $\{v, w\}$ and adding an edge in $\partial(X)$. Furthermore, no edge in $\partial(X)-\{\{v, w\}\}$ can be among the included edges. □

LEMMA 4. *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. The proof is analogous to that of Lemma 3. □

We would like to prove a worst-case lower bound of $\Omega(m \log \log m)$ comparisons for finding a minimum spanning tree in a graph with m edges. 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 "demon" (sometimes called an oracle or adversary in such proofs by other authors) generates a bad case for such a comparison algorithm:

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

(ii) If an excluded and a nonexcluded edge are compared, the demon declares the excluded one as bigger.

(iii) If two nonexcluded edges are compared, the demon declares the one with more edges known to be smaller as bigger.

LEMMA 5. *Suppose the demon above determines the results of comparisons for a comparison algorithm obeying rule* Max. *Then at all times during the comparison process, any nonexcluded 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 demon and Lemma 4, no excluded edge is ever known to be smaller than any nonexcluded edge. Let $\{v, w\}$ be any nonexcluded edge. By rule Max and rule (iii) of the demon, any comparison which adds to the number of nonexcluded 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 5 to give the desired lower bound. The *girth* of a graph is the length of the shortest cycle in the graph.

LEMMA 6 (Tutte [18], p. 82]). *For all n, there is a graph of n vertices, with all vertices of degree three, having a girth which is $\Omega(\log n)$.*

THEOREM 1. *Any comparison algorithm obeying rule* Max *requires $\Omega(m \log\log m)$ comparisons for some graph regular of degree three having m edges.*

Proof. Suppose a comparison algorithm obeying rule Max is applied to one of the graphs given by Lemma 6. Any such graph has exactly $m = \frac{3}{2}n$ edges. Thus $\frac{1}{3}m + 1$ 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 nonexcluded edges (Lemma 4), but since any such cycle has length $\Omega(\log m)$, Lemma 5 implies that $\Omega(\log\log m)$ direct comparisons with the excluded edge are required. These $\Omega(\log\log m)$ comparisons are distinct for each excluded edge; thus a total of $\Omega(m \log\log m)$ comparisons are required. □

This proof of an $\Omega(m \log\log m)$ lower bound works only for a restricted class of algorithms. However, Lemma 4 and Lemma 6 together imply that any method which computes a minimum spanning tree for one of Tutte's graphs must compute the maximum element of $\Omega(m)$ sets, each of size $\Omega(\log m)$. Priority queue method (b) for implementing MINSPAN in fact works by computing minima over sets of size $\log m$. This step (of computing maxima or minima over sets of size $\log m$) seems to be the costly part of finding a minimum spanning tree, and if we could get a better understanding of the cycle structure in Tutte's graphs, we might be able to prove a general $\Omega(m \log\log m)$ lower bound for the minimum spanning tree problem.

7. Conclusions and conjectures.

We have presented a general minimum spanning tree algorithm and studied its worst-case running time for various implementations and various types of graphs. We have given $O(m \log\log n)$ implementations for arbitrary graphs, an $O(n)$ implementation for planar graphs, and an $O(m)$ implementation for dense graphs (those for which $m > cn^{1+\varepsilon}$ for some positive constants c and ε). We believe the algorithms we have presented are not hard to implement and have constants of proportionality small enough to make them competitive in practice with older algorithms. (We have not yet performed any experiments to verify this conjecture.)

We have shown connections between the minimum spanning tree problem and a set manipulation problem and an ordering problem which may lead to general nonlinear lower bounds. It is interesting to note that the problem of testing whether a given spanning tree is in fact minimum requires only $O(m\alpha(m, n))$ time [16]; thus testing a minimum spanning tree may be easier than finding a minimum spanning tree.

In addition to the problem of lower bounds, another area needs study; that of *average* running time. A "random" graph can be defined in several ways [6]; for any such definition, what is a minimum spanning tree algorithm with a good average running time? We can show that Kruskal's algorithm runs in $O(n+m)$ time on the average if the edges are presented in sorted order. We have devised an algorithm with a provable $O(n \log\log n + m)$ average running time (for edges not in sorted order) and a related algorithm which we believe has an $O(n+m)$ average running time. We also conjecture that MINSPAN, implemented without cleanups and with queues represented as leftist trees with delayed merge, has an $O(n+m)$ average running time. We hope to report on these ideas in a future paper.

Acknowledgment. The authors wish to thank Andrew Yao for providing stimulus and many helpful suggestions for this work.

REFERENCES

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

[2] ———, *On finding lowest common ancestors in trees*, Proc. 5th Ann. ACM Symp. on Theory of Computing, 1973, pp. 253–265.

[3] C. BERGE AND A. GHOUILA-HOURI, *Programming, Grames, and Transportation Networks*, John Wiley, New York, 1965.

[4] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST AND R. TARJAN, *Time bounds for selection*, J. Comput. Systems Sci., 7 (1973), pp. 448–461.

[5] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[6] P. ERDÖS AND A. RÉNYI, *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Közl, 5 (1960), pp. 17–61.

[7] J. HOPCROFT AND R. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[8] J. HOPCROFT AND J. ULLMAN, *Set-merging algorithms*, this Journal, 2 (1973), pp. 294–303.

[9] J. HOPCROFT AND J. WONG, *Linear time algorithm for isomorphism of planar graphs*, Proc. 6th Ann. ACM Symp. on Theory of Computing, 1974, pp. 172–184.

[10] A. KERSCHENBAUM AND R. VAN SLYKE, *Computing minimum spanning trees efficiently*, Proc. 25th Ann. Conf. of the ACM, 1972, pp. 518–527.

[11] D. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[12] J. B. KRUSKAL, JR., *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 48–50.

[13] R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

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

[15] ———, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[16] ———, *Applications of path compression on balanced trees*, STAN-CS-75-512, Computer Sci. Dept., Stanford Univ., Stanford, Calif., 1975.

[17] ———, *Finding optimum branchings*, Networks, to appear.

[18] W. T. TUTTE, *Connectivity in Graphs*, Toronto University Press, Toronto, 1967.

[19] J. VUILLEMIN, private communication, 1975.

[20] A. C. YAO, *An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees*, Information Processing Letters, 4 (1975), pp. 21–23.

[21] ———, private communication, 1975.