Two Algorithms for Generating Weighted Spanning Trees in Order

Harold N. Gabow*

CU-CS-078-75 August 1975

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was partially supported by the National Science Foundation under Grant GJ 36461

1. Introduction

In many practical situations, one wants to generate the spanning trees of a graph in order of increasing weight. For example, consider this electrical wiring problem: n pins must be wired together with as little wire as possible; further, the wiring must satisfy complicated and diverse constraints, e.g., at most k wires can meet any pin, no two wires can be too close, etc. One way to solve the problem is to generate spanning trees in order of increasing length (weight) until a tree satisfying the constraints is found.

This paper presents an algorithm for generating the K smallest spanning trees of a connected graph, in order of increasing weight. K may be known in advance or specified as trees are generated. The algorithm requires time $O(KE\alpha(E,V)+E \log E)$ and space $O(K+E)$. Here V is the number of vertices, E is the number of edges, and α is Tarjan's inverse of Ackermann's function and is very slow-growing. A previously known algorithm for this problem [2] has slower run time, $O(K \min(N^2,E(\log E)^2)+ K \log K)$. The algorithm works by using a minimum weight spanning tree as a "reference tree" [6], and exchanging edges to find other trees. It uses fast set merging algorithms [10], and a bounding technique for partitioning the solution space [5].

The algorithm is modified for the problem of generating all spanning trees of a connected graph, in order of increasing weight. The run time is $O(NE)$ and the space is $O(N+E)$, where N is the number of spanning trees of the graph. Efficient algorithms for generating all spanning trees of a graph without weights [7,8] can be applied to this problem.

The best known algorithm gives time bound O(NE) and space bound O(NV+E). Since N can be large, storage may be a limiting factor. So the improvement in space is significant.

Section 2 gives definitions from graph theory. Section 3 gives a result on edge exchanges that is the basis of the algorithm. Section 4 presents the algorithm, and analyzes time and space requirements. Section 5 describes the modified algorithm. Section 6 summarizes computational experience.

2. Definitions

A graph G consists of a finite set of V vertices and a finite set of E edges. An edge is an (unordered) set of two distinct vertices. The edge containing vertices v and w is denoted (v,w); it joins v and w.

A subgraph of G is a graph whose vertices and edges are in G. If H is a collection of edges on the vertices of G, graph G-H consists of the vertices of G and all edges in G except H; graph $G \cup H$ consists of all vertices of G and all edges in G or H.

A path from v_1 to v_n is a sequence of edges $(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},v_n)$. If all vertices v_i, $1 \le i \le n$, are distinct, the path is simple. A graph is connected if there is a path between any two distinct vertices. A bridge is an edge (v,w) that is in every path from v to w.

A tree is a connected graph, where any two distinct vertices are joined by a unique simple path. A spanning tree of G is a tree that is a subgraph containing all vertices of G.

A rooted tree is a tree T with one vertex r chosen as the root. Let v be a vertex in T, and let $v = v_0, v_1, \ldots, v_n = r$ be the sequence of vertices in the simple path from v to r. Any vertex v_i, for $0 \le i \le n$, is an ancestor of v; v_1 is the father of v; v is a son of v_1; the depth of v is n. The _first common ancestor_ of two vertices v and w is the ancestor of both v and w that has greatest depth. Two distinct sons of the same vertex are _brothers_. In an _oriented tree_, the sons of a vertex are ordered from left to right.

In a _weighted graph_, every edge e has a number, _w(e)_, called its _weight_. If H is a set of edges (such as a spanning tree), the _weight of H_ is $\sum_{e \in H} w(e)$.

3. T-exchanges

This section describes how spanning trees can be derived by exchanging edges. T-exchanges are defined and a useful property is proved.

Let T be a spanning tree of graph G. A T-exchange is a pair of edges e,f where $e \in T$, $f \notin T$, and $T-e \cup f$ is a spanning tree. The weight of exchange e,f is w(f)-w(e). So the weight of tree $T-e \cup f$ is the weight of tree T plus the weight of exchange e,f.

A T-exchange can be used to derive one minimum weight spanning tree from another, as shown below.

Lemma 1: A spanning tree T has minimum weight if and only if no T-exchange has negative weight.

Proof: The necessity of this condition is obvious. Sufficiency is proved in [4]. ∎

Theorem 2: Let T be a minimum weight spanning tree of G, and let e be an edge in T. Let e,f be a T-exchange having the smallest weight of all T-exchanges e,f'. Then $T-e \cup f$ is a minimum weight spanning tree of graph G-e.

Proof: Let $S = T-e \cup f$. Suppose S does not have minimum weight. By Lemma 1, there is an S-exchange g,h having negative weight. We derive a contradiction below.

We first show edges e,f,g,h are situated as in Figure 1. Let T-e consist of the two trees U,V. Edge e joins U and V. Edge f also joins U and V, since e,f is a T-exchange.

Edge h also joins U and V. For if not, assume without loss of generality that h joins two vertices in U. Since g,h is an S-exchange, g is in tree U. Thus g,h is a T-exchange. But g,h has negative weight, and T is a minimum weight spanning tree. This contradiction proves h joins U and V.

Now we show edge $g \in U \cup V$. Since e,h is a T-exchange, its weight is no smaller than that of e,f. Thus

(1) $$w(f) \leq w(h) < w(g).$$

So $g \neq f$, and $g \in S-f = U \cup V$.

Assume without loss of generality that $g \in U$. This gives Figure 1.

Now let U-g consist of the two trees W,X. Edge e is incident to one of these trees, say W. Since $T-e-g \cup f \cup h$ is a spanning tree, either f or h is incident to X. So either g,f or g,h is a T-exchange. But (1) implies both of these exchanges have negative weight, a contradiction. This contradiction shows the original assumption is false. Thus S has minimum weight. ∎

Now let T be a minimum weight spanning tree. The Theorem shows a spanning tree with the next smallest weight is $T-e \cup f$, where e,f is a minimum weight T-exchange. This observation is the basis of the algorithm.

4. Algorithm for K Spanning Trees

The algorithm for generating K spanning trees consists of procedures EX,GEN, and a main procedure GENK. This section describes these procedures in turn.

EX finds a minimum weight T-exchange e,f subject to certain constraints, IN,OUT. We describe EX, first assuming there are no constraints and then incorporating the constraints.

For each edge $g \in T$, EX finds a minimum weight T-exchange g,h; it sets e,f to the smallest of these exchanges. During the execution of EX, we call an edge $g \in T$ <u>eligible</u> if a minimum weight exchange g,h has not been found.

EX works as follows. Originally all edges in T are eligible. A list L contains all edges $h_1, h_2, \ldots, h_E$ of G, sorted in order of increasing weight. EX begins by finding every T-exchange g, h_1. This is a minimum exchange for g, since h_1 has the smallest weight possible. Now all edges g become ineligible.

Edge h_2 is processed similarly. EX finds every T-exchange g, h_2, where g is eligible. This is a minimum exchange for g. The edges g become ineligible.

The procedure is repeated for every edge h_k in L. The smallest exchange found is the minimum T-exchange e,f.

The tree T is represented by a father array F. Vertex 1 is the root; for any vertex $v \neq 1$, F(v) is the father of v. Call a vertex $v \neq 1$ <u>eligible</u> if edge (v,F(v)) is eligible, i.e., a minimum exchange for (v,F(v)) has not been found; call vertex 1 eligible, too.

Now suppose EX examines the edge $h_k = (x,y)$, as shown in Figure 2. Vertex <u>a</u> is the first common ancestor of x and y that is eligible; vertices x_i, for $1 \le i \le n$, are the other eligible vertices on the path from x to <u>a</u>; vertices y_j, for $1 \le j \le m$, are defined similarly. Edge h_k gives a minimum exchange for each edge $(x_i, F(x_i))$, $(y_j, F(y_j))$, and no others.

EX finds the vertices x_i, y_j, by using set merging techniques. A family of sets partitions the vertices of G. Each set contains a unique eligible vertex, which is used as the name of the set. Vertex w is in the set named v when v is the first eligible ancestor of w. (Since 1 is eligible, v exists for any w.) EX uses two procedures to manipulate these sets: FIND (v) computes the name of the set containing vertex v; UNION (v,w,x) combines sets named v and w into a new set named x (destroying sets v and w).

EX computes the vertices x_i using the equations x_1 = FIND(x), x_{i+1} = FIND(x_i). Vertices y_j are computed similarly. Vertex a is the first common value, a = x_{n+1} = y_{m+1}. In this manner, EX finds all minimum exchanges for edge h_k.

Now we discuss the constraint lists IN and OUT. These lists prevent trees from being generated twice; their construction is described fully below. IN is a list of edges in T that must remain in T; OUT is a list of edges out of T that must remain out of T. Thus EX must find a T-exchange e,f, such that $e \varepsilon T$-IN, $f \varepsilon G$-OUT, and e,f has the smallest weight possible.

To do this, EX begins by making all IN edges (x,y) ineligible. This is done by placing x and y in the same set (since they have the same first eligible ancestor). Also, EX marks all OUT edges in the list L, so they are not considered for exchanges. These edges are later unmarked, for subsequent calls to EX. In this manner, the constraints are handled.

The final array used is W, which specifies the weight W(e) of each edge e.

The procedure below sets global variables e,f and r, so e,f is a minimum weight exchange subject to constraints IN,OUT, and r is the weight of e,f.

```
                      procedure EX(F,IN,OUT); begin
1.  initialize:       r ← ∞; make each vertex v the only element in a set
                      named v;
2.  IN edges:         for edge (x,y) in IN do begin wlog assume F (x) = y;
3.                      y ← FIND (y); UNION (x,y,y);
                        end;
```

```
 4. OUT edges:        for edge (x,y) in OUT do mark (x,y) in L;
 5. L edges:          for edge (x,y) in L do comment edges in L are in
                      increasing weight order;
 6.                     if (x,y) is marked then unmark (x,y)
 7.                     else if F(x) ≠ y and F (y) ≠ x then begin
 8. find exchanges:   let a be the first eligible common ancestor of x and y;
 9.                   for v1 ← x,y do begin v ← FIND(v1);
10.                     while v ≠ a do begin
11. check exchange:           r1 ← W(x,y)-W(v,F(v)); comment r1 is the
                              exchange's weight;
12.                           if r1 < r then begin r ← r1; e ← (v,F(v));
                              f ← (x,y), end;
13. advance:                  u ← FIND(F(v)); UNION(v,u,u); v ← u;
                      end end end end EX;
```

Table I illustrates EX. Input parameters describe T_1 for the graph of Figure 3. The exchanges found by EX in step 8 are listed, along with the output values describing T_2.

Lemma 3: Let T be a minimum weight spanning tree subject to constraints $IN \subseteq T \subseteq G-OUT$. Then EX, called with a father array for T, sets e,f so $T-e \cup f$ is the next smallest spanning tree subject to the same constraints.

Proof: Theorem 2 shows $T-e \cup f$ has the desired property if e,f is a minimum weight T-exchange subject to constraints $e \notin IN$, $f \notin OUT$. The remarks preceding EX show EX finds such an exchange. ∎

Lemma 4: EX runs in time $O(E\alpha(E,V))$.

Proof: First we bound the time for set operations. Lines 3 and 13 together do at most one UNION for each vertex. So they do O(V) UNIONs, and also O(V) FINDs. Line 9 does at most two FINDs for each edge in G-T. Thus, a total of O(V) UNIONs and O(E) FINDs are done. If fast set merging algorithms are used, the total time is $O(E\alpha(E,V))$ [10].

Line 4 can be done in time O(E). For OUT contains at most E edges e; e can be marked (in L) in time O(1), if OUT contains a pointer to e in L.

Next consider line 8. Vertex a can actually be computed as exchanges for vertices v are found (lines 9-13). To do this, the paths from x to a and y to a are processed simultaneously. Referring to Figure 2, suppose x_i and y_j have been found, for some $i \leq n$, $j \leq m$; if $x_i \neq y_j$, the vertex with greater depth precedes a, and so is x_{i+1} or y_{j+1}. In this manner, the exchanges for vertices v, and the vertex a, are found.

To do this efficiently, we use an array D, that gives the depth D(v) of vertex v in T. D can be computed from F in time O(V). Thus, line 8 requires only O(V) extra time.

The remaining lines in EX require time O(E+V). This gives the desired time bound. ▮

Procedures GEN and GENK call EX to generate spanning trees in correct order. A technique resembling branch and bound, described by Lawler [5], is used.

Let T_i, for $1 \leq i \leq N$, denote the spanning trees of G, in order of increasing weight. Suppose the algorithm has output the first j-1 trees, j>1. The remaining trees have been partitioned into j-1 sets of the form

$$P_i^{j-1} = \{T_k \mid k > j-1;\ e_1, e_2, \ldots, e_r \in T_k;\ e_{r+1}, \ldots, e_s \notin T_k\},$$

for $1 \leq i < j$ (Here r and s vary over sets). Tree T_i satisfies all conditions of this set except the first, $k > j-1$. The smallest tree in this set is known; it is the result of a T_i-exchange e,f.

Procedure GEN finds the next smallest tree T_j from among the j-1 smallest trees in these sets. Suppose T_j is in the above set P_i^{j-1}, and $T_j = T_i - e \cup f$. GEN computes T_j and outputs it. Then a new partition P_k^j, $1 \leq k \leq j$, is formed, by subdividing P_i^{j-1}. For all $k \neq i$ in $1 \leq k < j$, set $P_k^j = P_k^{j-1}$; further,

$$P_i^j = \{T_k \mid k > j;\ e_1, \ldots, e_r, e \in T_k;\ e_{r+1}, \ldots, e_s \notin T_k\},$$
$$P_j^j = \{T_k \mid k > j;\ e_1, \ldots, e_r \in T_k;\ e_{r+1}, \ldots, e_s, e \notin T_k\}.$$

T_i satisfies all conditions of P_i^j except the first. GEN finds the smallest tree in P_i^j by executing EX on T_i, with edges $e_1, \ldots, e_r, e$ in IN, and $e_{r+1}, \ldots, e_s$ in OUT. (Lemma 3 shows EX works correctly.) P_j^j is processed similarly, using tree T_j. In this way, the partition is updated. Now T_{j+1} can be found.

The algorithm uses a list P to represent the partition. A set P_i^j is represented by a tuple (t,e,f,F,IN,OUT) in P. Here t is the weight of the smallest tree in P_i^j; F is the father array for T_i; the smallest tree in P_i^j results from the T_i-exchange e,f; IN is the list of edges that are in all trees of P_i^j; OUT is the list of edges that are out of all trees of P_i^j.

Procedure GEN outputs the next smallest tree T_j from set P_i^{j-1}, and puts the new sets P_i^j, P_j^j in the partition.

```
                 procedure GEN; begin
1.  find $P_i^{j-1}$:  remove the tuple (t,e,f,F,IN,OUT) with smallest weight t
                 from P;
2.               if $t = \infty$ then stop; comment all spanning trees have been output;
```

```
3. output $T_j$:  Fj ← F; modify Fj so edge f replaces e; output (Fj);
4.                ti ← t-W(f)+W(e); comment ti is the weight of $T_i$;
5.                form list INi by adding e to IN; form list OUTj by adding e
                  to OUT;
6. form $P_i^j$:  EX(F,INi,OUT); add (ti+r,e,f,F,INi,OUT) to P; comment EX
                  sets e,f,r for the minimum weight exchange;
7. form $P_j^j$:  EX(Fj,IN,OUTj); add (t+r,e,f,Fj,IN,OUTj) to P;
                  end GEN;
```

Table II illustrates GEN, by showing the partition after tree T_2 for Figure 3 is output.

Lemma 5: Let GEN be called, with P containing tuples for the sets P_k^{j-1}, $1 \leq k < j$. Then GEN outputs T_j, and changes P to tuples for the sets P_k^j, $1 \leq k \leq j$.

Proof: The Lemma follows from the remarks preceding GEN. ■

Lemma 6: GEN runs in time $O(E\alpha(E,V))$.

Proof: Lines 1, 6 and 7 remove and add tuples to P. These operations can be done in time O(E). For this, we use a heap [1]. The tuples in P are numbered, by assigning i to P_i^j. An entry in the heap is a number i; entries are ordered on the weight t of the corresponding tuple. The heap contains at most K entries. So an entry can be removed or added in time O(log K). Since $K<2^E$, this is O(E).

Line 3 requires time O(V) to derive Fj. Referring to Figure 2, let $e = (x_i, F(x_i))$, $f = (x,y)$. To derive Fj, array F need only be changed on the path from x to x_i.

The rest of the time is dominated by the two calls to EX. This gives the desired bound. ▮

The main procedure GENK finds a minimum spanning tree, and calls GEN for the next (K-1) trees.

```
                        procedure GENK(K); begin
1.                      make L a list of the edges of G, sorted in order of increasing
                        weight;
2.  find $T_1$:         make F a father array for a minimum weight spanning tree,
                        having weight t; output(F);
3.  form $P_1^1$:       EX $(F,\phi,\phi)$; make $(t+r,e,f,F,\phi,\phi)$ the only tuple in P;
4.  generate $T_j$:     for $j \leftarrow 2$ to K do GEN;
                        end GENK;
```

Table III shows the partition after all minimum weight spanning trees for Figure 3 are output.

Lemma 7: GENK generates the K smallest weight spanning trees in order, in time $O(KE\alpha(E,V) + E \log E)$.

Proof: The correctness of GENK follows by induction, using Lemma 5. Now we analyze the time. Line 1 requires time $O(E \log E)$ to sort E edges. Line 2 can be done in time $O(E \log E)$, using Kruskal's algorithm [1]. The rest of the time is dominated by $O(K)$ calls to GEN, requiring time $O(KE\alpha(E,V))$ by Lemma 6. ▮

When K is small, the term $E \log E$ in the run time is significant. It can be reduced to $E \log \log E$. This is done by using a faster minimum spanning tree algorithm [3,12], and changing EX so the edges of G need not be sorted. For the latter, the techniques of [11] are used.

Now we examine the space requirements. The dominating requirement is for P. Since a tuple can contain O(E) words, P can use O(KE) words. Below we modify the data structure to reduce this bound, without changing the run time.

Lemma 8: GENK uses O(K+E) space.

Proof: We show the structures F,IN,OUT in a tuple can be replaced by 6 words, and still be computed in time O(E). The extra time is spent in line 1 of GEN, and does not change the algorithm's time bound; P is reduced to O(K) words. So this suffices to prove the Lemma.

Suppose the first k trees T_j, $1 \le j \le k$, have been output. Make these trees the vertices of an oriented tree $\mathcal{T}$, as follows. Make T_1 the root of $\mathcal{T}$. For $j>1$, make T_j a son of T_i if GEN derives T_j from a T_i-exchange. Let e_j,f_j denote this T_i-exchange. Arrange the sons T_j of T_i so j increases from left to right.

For a set P_j^k, the structures F, IN, and OUT can be derived from $\mathcal{T}$, as follows. Let A be the path in $\mathcal{T}$ from T_j to the root T_1. Consider these sets:

$$O_1 = \{e_i \mid T_i \text{ is in A and } i>1\},$$

$$O_2 = \{f_i \mid T_i \text{ is in A and } i>1\},$$

$$T = (T_1 \cup O_2)-O_1,$$

$$I_1 = \{e_i \mid T_i \text{ is a left brother of some } T_\ell \text{ in A}\},$$

$$I_2 = \{e_i \mid T_i \text{ is a son of } T_j\}.$$

T contains the edges of T_j; the father array F for T_j is easily derived. $I_1 \cup I_2$ contains the edges of IN, and O_1 contains the edges of OUT.

Now for each set P_j^k, replace F,IN,OUT by the 6 words j,i,ej,fj,s,b. Here j is the index in P_j^k; i is the index of the father T_i of T_j; ej,fj is the exchange that derives T_j from T_i; s is the index of the rightmost son (if any) of T_j; b is the index of the brother (if any) immediately to the left of T_j.

These 6 words are easily computed in GEN. Let the tuple found in line 1 be (t,e,f,i,il,ei,fi,s,b). Then line 6 adds the tuple (ti+r, e,f,i,il,ei,fi,j,b) to P; line 4 saves e,f as ej,fj, and line 8 adds the tuple (t+r,e,f,j,i,ej,fj,0,s) to P.

Finally, it is easy to see the five sets above, and F,IN,OUT, can be constructed (in line 1 of GEN) in time O(E).

Table IV shows how the partition in Table III is changed using this scheme.

5. Algorithm for All Spanning Trees

This section describes a modification of the algorithm that generates all spanning trees of a connected graph, in order of increasing weight. The run time is O(NE), a slight improvement over GENK. The three main changes are described below.

The first change eliminates extra calls to EX, by modifying the tuples for sets P_i^j in P. The tuple's minimum T_i-exchange e,f is replaced by X, a list of T_i-exchanges. For each edge $g \varepsilon T_i$-IN, X contains g,h, the smallest exchange with $h \varepsilon G$-OUT. So e,f is the smallest exchange in X.

EX is modified to generate X. X is initialized to an empty list (in line 1), and a line is added after 12:

```
12.1.    add exchange (v,F(v)),(x,y) to X;
```

GEN is modified to use X. In line 1, the minimum exchange e,f is found by examining X. In line 6, the call to EX is eliminated; the tuple for P_i^j is formed by removing e,f from X, and computing the new minimum weight t, using X.

The second change, in the set merging algorithms, speeds up FIND at the expense of UNION. Sets are represented by an array VSET, where VSET(w) = v when v is the name of the set containing w. With this approach, FIND takes time O(1) and UNION takes time O(V).

EX uses these algorithms to manipulate the sets of vertices. Also, lines 2-3 are modified to avoid calls to UNION:

```
2'.  IN edges:  sort the edges (x,y) of IN, so F(x) = y and the depth
                of x increases;
3'.             for edge (x,y) in IN do VSET(x) ← VSET(y);
```

When (x,y) is processed in line 3', VSET(y) already has its final value. So lines 2'-3' initialize the sets correctly.

The third change involves the main procedure. A new main procedure is used to insure all trees are generated:

```
        procedure GENA; begin GENK(∞) end;
```

Note the algorithm halts in line 2 of GEN.

Further, GENK is modified to reduce the set-up time in lines 1-2. Line 1 is changed:

```
1'.  make B a list of the bridges of G; make L a list of the edges of G-B,
     sorted in order of increasing weight;
```

In line 2, the bridges B are placed in the spanning tree, before the minimum spanning tree algorithm is called.

The remaining changes in the algorithm are typographical.

Lemma 9: GENA generates all spanning trees in order, in time $O(NE)$ and space $O(N + E)$, where N is the number of spanning trees.

Proof: The correctness of GENA follows from Lemma 7 and the discussion above. Now we analyze the time, beginning with GENK.

Let E' be the number of edges that are not bridges. Then lines 1'-2 of GENK use time $O(E + E' \log E')$. For in line 1', the bridges B can be found in time $O(E)$ [9]; L can be sorted in time $O(E' \log E')$. Line 2, using Kruskal's algorithm with the bridges already in the tree, uses time $O(E' \log E')$.

Now we show $E' \log E'$ is $O(NE)$. It suffices to show $E' < 2N$. Let T be a spanning tree. We can list T-exchanges e,f, so each edge that is not a bridge occurs in the list. Since each exchange represents a distinct tree, $E' < 2N$. Thus, lines 1'-2 use time $O(NE)$.

The rest of GENK is dominated by $O(N)$ calls to GEN. As before, the heap operations in GEN (lines 1,6,7) require time $O(NE)$; the rest of GEN is dominated by $O(N)$ calls to EX (line 7).

In EX, line 2' can be done once in time $O(V)$. First the depth of each vertex in the tree is computed, using F. Then the edges of IN are sorted on depth, using a bucket sort [1] with one bucket for each depth. Thus, line 2' uses total time $O(NV)$.

Line 13 of EX, using the modified FIND and UNION procedures, is done once in time $O(V)$. It is executed N times, once for each spanning tree. So the total time is $O(NV)$.

The rest of EX uses time $O(NE)$. This gives the desired time bound for GENA.

Now we examine the space. The X lists in P require a total of O(N) words, since each exchange in an X list represents a distinct spanning tree. Using the modification of Lemma 8, the rest of P uses O(N) words. So O(N + E) space is used. ∎

Minty [7] and Read and Tarjan [8] give an algorithm that generates all spanning trees of a graph without weights. It requires time O(NE) and space O(E), assuming each tree is output as soon as it is generated. We can generate all spanning trees in a weighted graph, by first applying this algorithm, and then sorting the trees in order of increasing weight. This method requires time O(NE). The space is O(NV + E), since each tree must be stored until the sort is done. GENA has the same time bound but uses less space. Since N can be large ($N = V^{V-2}$ in a complete graph), space is likely to be a limiting factor, so this improvement is significant.

6. Computational Experience

A version of GENK has been programmed in PASCAL and run on the CDC 6400. The program stores tuples in a hybrid form (using the notation defined above, a tuple is (t,X,i,fj,F,IN)). The time bound for the program is $O(KE\alpha(E,V) + E \log E)$, and the space is O(KV + E).

Experiments were conducted on random connected graphs with random integer edge weights between 50 and 10000. Tables V-VI show results. The run time is specified by $\overline{T}$, the average time in milliseconds to generate one spanning tree. (Experiments show for a given graph, the time per spanning tree is approximately constant, as expected.) Table V shows $\overline{T}$ for graphs that are almost complete; $\overline{T}$ is computed from K = 15. Table VI shows $\overline{T}$ for graphs with 60 vertices; again K = 15. When $\overline{T}$ is plotted against E, the data is almost linear. Least square fits for the two tables have slopes .72 and .68, respectively. Almost linear performance is expected from the asymptotic time bound, since the factor $\alpha(E,V)$ is constant in this relatively small range.

7. Acknowledgments

The author thanks Professor Donald Johnson of Pennsylvania State University for his inspiring discussions, Mr. Robert Palasek for programming GENK, and the referee for indicating how to improve GENK for small K.

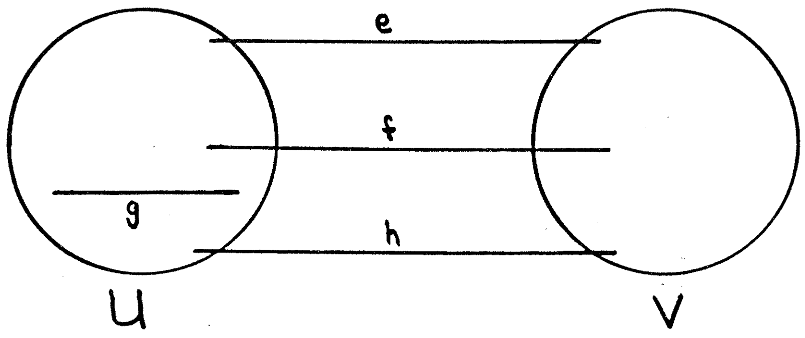

Fig. 1 Edges in Theorem 1

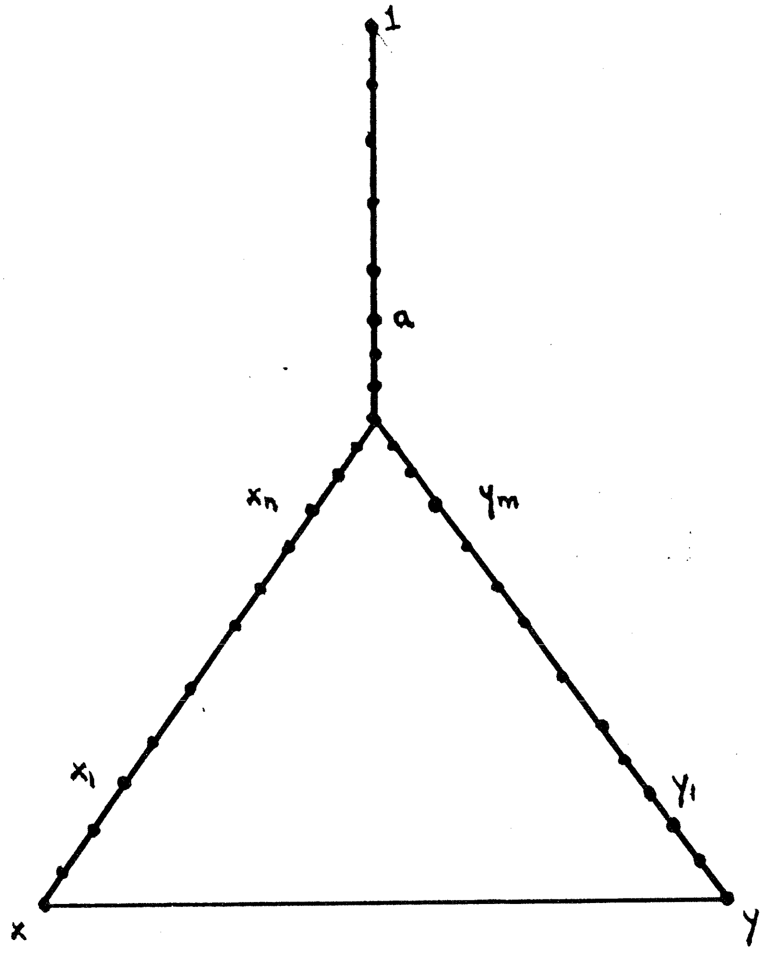

Fig. 2 Exchanging (x,y) into T

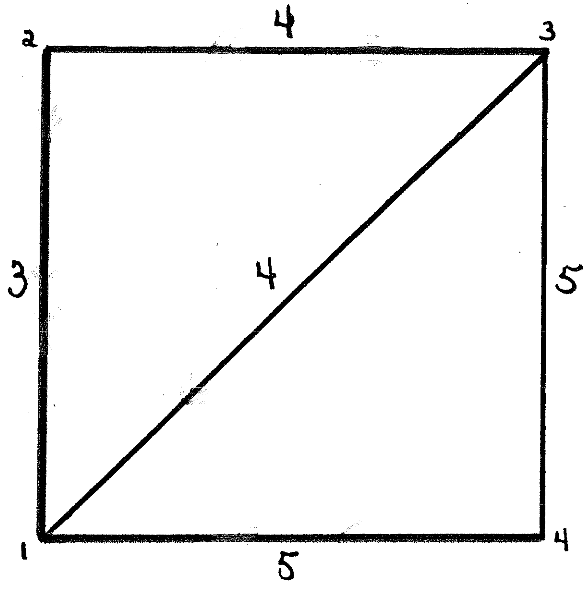

Fig. 3 Example graph

t	F(2)	F(3)	F(4)	IN	OUT		r	e	f
12	1	2	3	-	-	(2,1),(3,1);(3,2),(3,1);(4,3),(4,1)	0	(3,2)	(3,1)
input						exchanges	output		

Table I.

EX processes T_1

	t	e	f	F(2)	F(3)	F(4)	IN	OUT
P_1^2	12	(4,3)	(4,1)	1	2	3	(3,2)	-
P_2^2	12	(4,3)	(4,1)	1	1	3	-	(3,2)

Table II.

Partition P_i^2

	t	e	f	F(2)	F(3)	F(4)	IN	OUT
P_1^4	13	(2,1)	(3,1)	1	2	3	(3,2),(4,3)	-
P_2^4	13	(3,1)	(4,1)	1	1	3	(4,3)	(3,2)
P_3^4	13	(2,1)	(3,1)	1	2	1	(3,2)	(4,3)
P_4^4	∞	-	-	1	1	1	-	(3,2),(4,3)

Table III.

Partition P_i^4

	j	i	ej	fj	s	b
P_1^4	1	-	-	-	3	-
P_2^4	2	1	(3,2)	(3,1)	4	-
P_3^4	3	1	(4,3)	(4,1)	-	2
P_4^4	4	2	(4,3)	(4,1)	-	-

Table IV

Changes in P_i^4 using storage reduction scheme

V	10	15	25	35	45	55	60
E	44	104	300	595	985	1476	1762
$\overline{T}$	12	27	74	143	236	362	422

Table V

Time for generating one tree

E	159	336	513	690	867	1044	1221	1398	1575	1757
$\overline{T}$	48	86	124	162	199	237	279	324	365	413

Table VI

Time for generating one tree, V = 60

References

[1] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[2] Camerini, P. M., L. Fratta, and F. Maffioli, "The K shortest spanning trees of a graph", Int. Rep. 73-10, IEE-LCE Politecnico di Milano, Italy.

[3] Cheriton, D. and R. E. Tarjan, "Finding minimum spanning trees", SIAM J. Comput., to appear.

[4] Deo, N., Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[5] Lawler, E. L., "A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem", Management Sci. 18, 7 (Mar. 1972), 401-405.

[6] Mayeda, W. and S. Seshu, "Generation of trees without duplications", IEEE Trans. on Circuit Theory CT-12, 2 (June 1965), 181-185.

[7] Minty, G. J., "A simple algorithm for listing all the trees of a graph", IEEE Trans. on Circuit Theory CT-12, 1 (Mar. 1965), 120.

[8] Read, R. C. and R. E. Tarjan, "Bounds on backtrack algorithms for listing cycles, paths, and spanning trees", preprint (Dec. 1973).

[9] Tarjan, R. E., "A note on finding the bridges of a graph", Information Processing Letters 2, 6 (Apr. 1974), 160-161.

[10] Tarjan, R. E., "Efficiency of a good but not linear set union algorithm", J. ACM 22, 2 (Apr. 1975), 215-225.

[11] Tarjan, R. E., "Applications of path compression on balanced trees", Tech. Rep. STAN-CS-75-512, Comp. Sci. Dept., Stanford Univ., Stanford, Calif. (Aug. 1975).

[12] Yao, A. C., "An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees", Inf. Proc. Letters 4, 1 (Sept. 1975), 21-23.