JOURNAL OF COMPUTER AND SYSTEM SCIENCES 16, 333–344 (1978)

# Algorithms for Updating Minimal Spanning Trees*

FRANCIS CHIN

*Department of Computing Science, University of Alberta, Alberta, Canada T6G 2H1*

AND

DAVID HOUCK

*Department of Mathematics, University of Maryland, Baltimore County, Baltimore, Maryland 21228*

Received February 24, 1976; revised May 15, 1977

The problem of finding the minimal spanning tree on an undirected weighted graph has been investigated by many people and $O(n^2)$ algorithms are well known. P. M. Spira and A. Pan (*Siam J. Computing* 4 (1975), 375–380) present an $O(n)$ algorithm for updating the minimal spanning tree if a new vertex is inserted into the graph. In this paper, we present another $O(n)$ algorithm simpler than that presented by Spira and Pan for the insertion of a vertex. Spira and Pan further show that the deletion of a vertex requires $O(n^2)$ steps. If all the vertices are considered, $O(n^3)$ steps may be used. The algorithm which we present here takes only $O(n^2)$ steps and labels the vertices of the graph in such a way that *any* vertex may be deleted from the graph and the minimal spanning tree can be updated in constant time. Similar results are obtained for the insertion and the deletion of an edge.

## 1. INTRODUCTION

Let $G = (V, E)$ be a finite, undirected graph with $| V | = n$. A *tree* is a connected graph containing no cycles. A *spanning tree* of $G$ is a subgraph of $G$ which is a tree and contains $n - 1$ edges. A vertex is called a *tip* of a tree if only one edge of the tree is incident to it. Let $c: E \to R$ be a cost function defined on the edges of $G$. A *minimal spanning tree* (m.s.t.) $T = (V, E_T)$ is a spanning tree which minimizes $c(T) = \sum_{e \in E_T} c(e)$. Another necessary and sufficient condition for $T$ to be m.s.t., is that each $e \in (E - E_T)$ is the largest edge in the cycle created by adding $e$ to $T$. Algorithms whose complexity is $O(n^2)$ to find a m.s.t. are well known [1]. Spira and Pan [2] present an $O(n)$ algorithm[1] for

* The work done by Francis Chin was supported in part by a Canadian National Research Council Grant.

[1] It is mentioned in [2] that another $O(n)$ algorithm was discovered by Donald Johnson and James Simon.



0022-0000/78/0163-0333$0.200/0

updating the m.s.t. if a new vertex is inserted into the graph. They also discuss the deletion of a vertex as well as the insertion and deletion of an edge. In this paper, we present another $O(n)$ algorithm simpler than the one presented in [2] for the insertion of a vertex. We also present an $O(n^2)$ algorithm which labels the vertices of the graph in such a way that *any* vertex may be deleted from $G$ and the m.s.t. can be updated in constant time. We also present algorithms to update the m.s.t. if an edge is inserted or deleted, or in other words, if its cost $c(e)$ is changed.

Before we proceed to the algorithms, a few trivial lemmas are helpful. Since the lemmas are almost obvious, only the key steps of their proofs are presented.

LEMMA 1. *Let $T = (V, E_T)$ be a* m.s.t. *of $G$. If a vertex and the edges incident to it are deleted from $G$ or just an edge is deleted from $E_T$ then each of the resulting components of $T$ is a* m.s.t. *on the graph induced by its vertices.*

*Proof.* Suppose some component $T_0 = (V_0, E_0)$ is not a m.s.t. Then there exists an edge in the induced subgraph of $V_0$ which is not the largest edge in the cycle it creates in $T_0$. This contradicts the minimality of $T$. ▮

Let $G' = (V', E')$ denote the graph $G$ after the deletion and let $E_D = E_T \cap E'$ be the tree edges in $G'$. Define a graph $G'' = (V'', E'')$ with vertices corresponding to the components of $T$ and with the edge between each pair of vertices in $V''$ being the smallest edge in $G'$ between the two corresponding components.

LEMMA 2. *If $T'' = (V'', E_T'')$ is a* m.s.t. *for $G''$ then $T' = (V', E_T')$ is a* m.s.t. *for $G'$ where $E_T' = E_D \cup E_T''$.*

*Proof.* Suppose $T'$ is not a m.s.t. Then there exists an $e^* \in (E' - E_T')$ which is not the largest edge in the cycle it creates in $T'$. By Lemma 1, $e^*$ must join vertices in different components of $T$. By minimality of $T''$ the largest edge in the cycle must join vertices in the same component of $T$. This contradicts the minimality of $T$ in $G$. ▮

LEMMA 3. *Let $T = (V, E_T)$ be a* m.s.t. *on $G = (V, E)$. If a new edge $e'$ is added to $G$, let us denote the new graph by $G' = (V, E \cup \{e'\})$. The* m.s.t. *$T'$ on $G'$ is obtained by adding $e'$ to $T$ and deleting the largest edge in the cycle created.*

*Proof.* Assume that $T'$ is not the m.s.t. on $G'$. Then there exists an edge $e^*$ which when added to $T'$ can create a cycle in which there is a larger edge, and this contradicts the minimality of $T$ on $G$. ▮

## 2. VERTEX AND EDGE INSERTION

We now present Algorithm I which updates a m.s.t. when a new vertex $x$ is added to $G$. Let $T = (V, E_T)$ be a m.s.t. represented by adjacency lists as defined in [3], i.e. $L(v)$, $v \in V$ is a list of all vertices incident to $v$ in $T$. Notice that $L(v)$ considers only edges in $T$.

The algorithm will construct a m.s.t. $T' = (V', E_T')$ on the new graph where $V' = V \cup \{z\}$. Algorithm I arbitrarily chooses a vertex as the root of $T$ and performs INSERT recursively by a depth-first search [3, 4]. It adds the edges $(z, v)$, $v \in V$ one at a time to $T$ and each time a cycle is created the largest edge in the cycle is deleted. By Lemma 3, $T'$ will be a m.s.t.

*Algorithm I. Insertion of a New Vertex to a* m.s.t.

*Input.* A m.s.t. $T = (V, E_T)$ on $G = (V, E)$ with a cost function $c(e)$, $e \in E$ and a new vertex $z$ with costs on the edges $(z, v)$ for $v \in V$. Note that $T$ is represented by adjacency lists $L(v)$ for $v \in V$, and $L(v)$ is only used for edges in $T$.

*Output.* The m.s.t. $T' = (V', E_T')$ for the new graph.

*Method.* Initially, all the vertices are marked "new", $E_T' = \phi$, and $t$ is a global variable. Choose any vertex $r \in V$ and execute INSERT($r$). This procedure stops when all vertices are marked "old." $T' = (V', E_T' \cup \{t\})$ is the m.s.t. for the new graph.

*Procedure* INSERT($r$)

*comment*: $t$ is a global variable and is the largest edge in the path between $w$ to $z$, whereas $m$ is the largest edge between $r$ and $z$.

```
1.  begin mark r "old"
2.      m ← (r, z)
3.      for each vertex w on L(r) do
4.          if w is marked "new" then
5.              begin INSERT(w)
6.                  k ← the larger of the edges t and (w, r)
7.                  h ← the smaller of the edges t and (w, r)
8.                  E_T' ← E_T' ∪ {h}
9.                  if c(k) < c(m) then m ← k
                end
10.     t ← m
    end
```

THEOREM 1. *Algorithm I correctly constructs the* m.s.t. *and takes* $O(n)$ *steps.*

*Proof.* The correctness is shown as follows. First, we have to show that after the execution of line 5, $m$ and $t$ are the largest edges in the paths from $r$ to $z$ and $w$ to $z$ respectively. If the vertices are numbered in the order that they complete their calls INSERT, then this can be proved by induction (see Fig. 1).

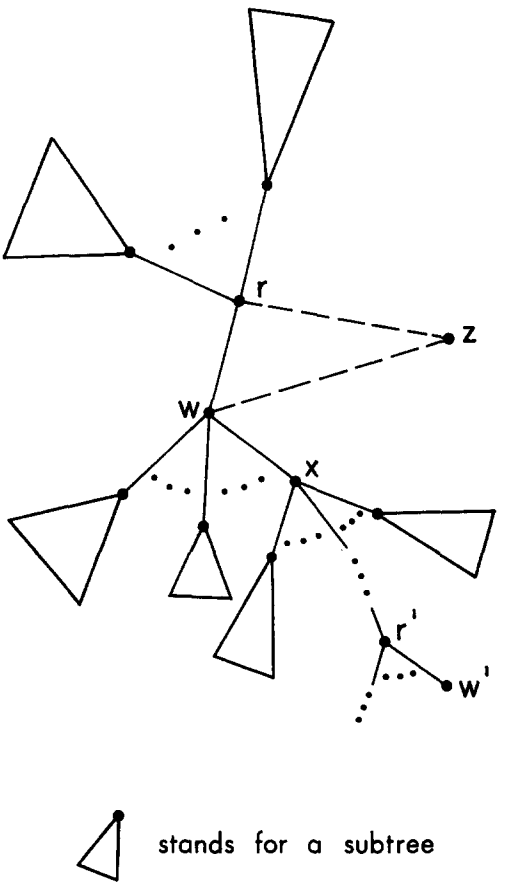


FIG. 1. Vertex insertion (Theorem 1).

*Basis.* The first vertex, say $w'$, which completes its call INSERT($w'$), must be a tip of $T'$. Thus, line 5 to line 9 will be skipped and $t$ will be assigned $(w', z)$ which is the only edge joining $w'$ and $z$. If $r'$ is the vertex incident to $w'$, it is easy to see that $m = (r', z)$ before and after the execution of INSERT($w'$).

*Induction steps.* Assume we are at line 5 in the procedure INSERT($r$) executing INSERT($w$). If $w$ is a tip, again lines 5 to 9 are skipped and $t = (w, z)$ which is the only edge joining $w$ and $z$. Otherwise, let $x$ be the vertex which is incident to $w$ and which is considered last in the procedure INSERT($w$). By the induction hypothesis, after the execution of INSERT($x$), $m$ and $t$ are the largest edges in the paths joining $w$ and $x$ to $z$ respectively. It can be shown easily that in all cases, $t$ will be the largest edge in the path joining $w$ to $z$. Similarly, we can also show that $m$ is the largest edge in the path joining $r$ to $z$.

In line 6 to 9, the largest edge in the cycle, i.e. the largest edge among $m$, $(w, r)$ and $t$, is

deleted and the m.s.t. $T'$ and $m$ are updated. By Lemma 3, since $n - 1$ edges are deleted, each of which was the largest in a cycle, $T'$ will be a m.s.t.

As far as time complexity is concerned, INSERT($r$) is called $n$ times (with $(n - 1)$ recursive calls at line 5) so the lines 1, 2, 6 to 9, and 10 are executed $n$ times. Lines 3 and 4 are executed $\sum_{v \in V} |L(v)|$ times. But this counts each tree edge twice so $\sum_{v \in V} |L(v)| = 2(n - 1)$. Therefore, Algorithm I has complexity $O(n)$. ▮

It is rather obvious that edge insertion follows directly from vertex insertion. To insert the edge $(v, w)$, we can simply add a new vertex $z$ with two edges $(v, z)$ and $(z, w)$. Let $c(v, z)$ be the same as $c(v, w)$ and let $c(z, w)$ be less than any other edge in $G$. The edge $(z, w)$ is always in the new m.s.t. and $(v, z)$ is in the m.s.t. iff $(v, w)$ should be in the m.s.t. The time bound follows directly from the time bounds for vertex insertion which is $O(n)$. But the following corollary covers a slightly more general situation, as it deals with the case of inserting more than one edge at one time. It is shown to be still $O(n)$ as long as all these edges are incident with a common vertex.

COROLLARY. *Let $T = (V, E_T)$ be* m.s.t. *on $G$. Assume that some of the edges incident to a vertex $w \in V$ have decreased in cost. (This case includes the insertions of one or more edges incident to $w$.) Then the* m.s.t. *can be updated in $O(n)$ steps.*

*Proof.* The m.s.t. can be obtained from the following algorithm.

*Step* 1. Create a new vertex $z$ and let $c((z, v)) = c((w, v))$ for $v \in V - \{w\}$ and $c((z, w)) = \min_{e \in E_T}\{c(e)\} - 1$.

*Step* 2. Execute Algorithm I with $T$ and $z$ as input.

*Step* 3. The new m.s.t. is obtained by the contraction of edge $(z, w)$, i.e., the deletion of vertex $z$ and the identification of edge $(w, v)$ and $(z, v)$, $v \in V - \{w\}$.

The correctness follows from the fact that the tree obtained in Step 2 always contains edge $(z, w)$. This makes the edges $(z, v)$ and $(w, v)$ equivalent topologically and the smaller of the two is always preferred. The time complexity is obvious since each of the three steps requires at most $O(n)$ steps. ▮

## 3. EDGE AND VERTEX DELETION

In this section, we present two $O(n^2)$ algorithms for updating the m.s.t. under two different kinds of modifications. One deals with the cases where an edge in $T$ is deleted or its cost is increased. (Obviously, similar modification of a non-tree edge has no effect on the m.s.t.). By Lemma 2, the new m.s.t. is found by deleting this edge from $T$ and inserting the shortest edge between the two resulting components. Spira and Pan [2] show that such an operation must take $O(n^2)$ steps. If all the tree edges are considered, $O(n^3)$ steps may be used by the brute-force method. The algorithm which we present here takes only $O(n^2)$ steps and it labels each tree edge with its replacement should it be

deleted. The difference in cost between these two edges is the amount that the cost of the tree edge can be increased without changing the m.s.t.

The second kind of modification deals with the deletion of an arbitrary vertex $z \in V$. By Lemma 2 the new m.s.t. can be found by adding $d_T(z) - 1$ edges to the remaining components of $T$, where $d_T(z)$ is the degree of vertex $z$ in the m.s.t. $T$. The algorithm to be presented will label each vertex with the $d_T(z) - 1$ edges. Notice that the total number of labels required will only be $\sum_{z \in V} [d_T(z) - 1] = 2(n - 1) - n = n - 2$. Analogous to the case of deleting an arc, this algorithm only takes $O(n^2)$ steps where the brute force method might use $O(n^3)$.

We should note that edge deletion can also follow directly from vertex deletion. Assume we have a labeled m.s.t. for vertex deletion, and if we want to delete an (arbitrary) edge, say $(u, v)$, we can obtain the new m.s.t. by first deleting the vertex $u$ and then adding a new $u'$ which has all of the edges incident with $u$ except for $(u', v)$. But in this case, since the insertion of a new vertex takes $O(n)$ steps, deletion of an arbitrary edge will take $O(n)$ steps with a labeled m.s.t. for vertex deletion. However, with a labeled m.s.t. which is specifically used for edge deletion as described in Algorithm DE, the deletion of an arbitrary edge only takes constant time.

Both of these algorithms start with a single vertex and build a subtree $T' = (V', E_T')$ of $T$. At each iteration a vertex $v$ adjacent to $V'$ in $T$ is added to $T'$ and the labels for $T'$ are updated. Both algorithms terminate when $T' = T$. At each iteration $T'$ is a m.s.t. on the subgraph generated by $V'$ and the labels would apply to the subgraph.

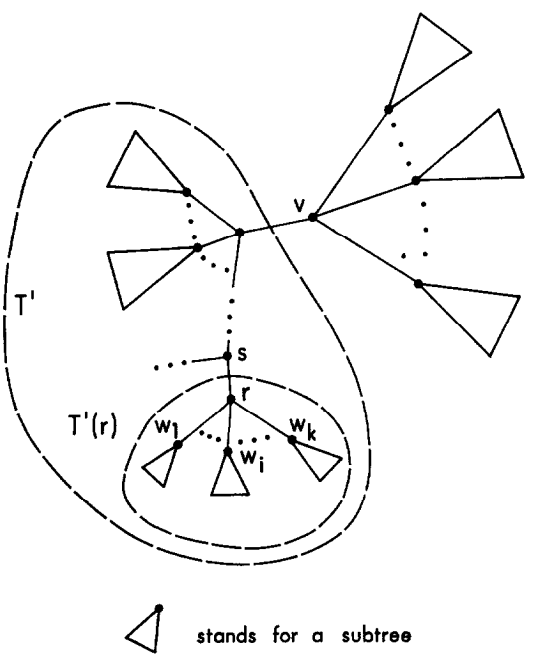


FIG. 2. Vertex and edge deletion (Algorithm DIST).

Before we proceed, the Algorithm DIST is necessary. DIST finds the smallest edge between the new vertex $v$ and each of the $| V' | - 1$ subtrees of $T'$ which are formed when considering $v$ as the root. More explicitly, for any $r \in V' - \{v\}$ let $(r, s)$ be the edge in the unique path between $r$ and $v$. Define $T'(r)$ to be the component of $T'$ containing $r$ formed by deleting edge $(r, s)$ and let $D(r) \in V'$ be the vertex such that $(D(r), v)$ is the smallest edge between $T'(r)$ and $v$ (see Fig. 2). Algorithm DIST can find $D(r)$ for all $r \in V' - \{v\}$ in time proportional to $| V' |$ by a depth-first search [3, 4].

*Algorithm* DIST: *To Find* $D(r)$ *for All* $r \in V' - \{v\}$

*Input.* A m.s.t. $T = (V, E_T)$ and $T' = (V', E_T')$ with $V' \subseteq V$ and $E_T' \subseteq E_T$. $T'$ is a m.s.t. on the graph generated by $V'$. All the vertices in $V'$ are marked "1" and "old" except $v$ which is marked "1" and "new," while all the vertices in $V - V'$ are marked "0."

*Output.* The array $D(r)$, $r \in V' - \{v\}$.

*Method.* Find an $r$ such that $(v, r) \in E_T'$, i.e., $r$ is incident to $v$ and is marked "1" and "old." Execute DIST$(r, v)$.

*Procedure* DIST$(r, v)$

*comment*: Two kinds of markings are needed. All the vertices in $V'$ are marked "1" and all the unvisited vertices are marked "old."

```
1. Begin mark r "new"
2.       m ← r
3.       for each vertex w on L(r) do
4.           if w is marked "1" and "old" then
5.             begin DIST(w, v)
6.                   if c((v, m)) > c((v, D(w))) then m ← D(w)
                  end
7.       D(r) ← m
   end
```

LEMMA 4. DIST *correctly finds* $D(r)$ *for* $r \in V' - \{v\}$ *and takes* $O(| V' |)$ *steps.*

*Proof.* The correctness can be proved by induction on the number of vertices in $T'(r)$ which is denoted by $| T'(r)|$. If $| T'(r)| = 1$ then vertex $r$ is a tip and $D(r) = r$ is correct. Suppose that the algorithm is correct for all trees up to $n$ vertices, and assume that for some vertex $s$, $| T'(s)| = n + 1$ and $w_1, w_2, \ldots, w_k$ are vertices in $T'(s)$ which are incident to $s$. By the induction hypothesis since $T'(w_1)$, $T'(w_2)$,..., $T'(w_k)$ have at most $n$ vertices in each of them, DIST$(w_i, v)$ returns correct values for $D(w_1)$,..., $D(w_k)$. Step 5 ensures that $m$ is the closest vertex to $v$ and $D(s)$ is then correct.

The time spent on DIST($r$, $v$) exclusive of recursive calls is proportional to $|L(r)|$. DIST is called once for each vertex and $\sum_{r \in V'} |L(r)| = 2(|V'| - 1)$. Thus, the total time is $O(|V'|)$. ∎

Algorithm DE handles the case where an edge is deleted. A label $l_e(e)$ for each $e \in E_T$ will indicate the new edge in the m.s.t. should $e$ be deleted. Vertices in $T'$ will be marked "1," all others "0." Each vertex is visited by a depth-first search and is added to $T'$, DIST finds the necessary smallest edges, and Procedure LE updates the labels. Again, there are two kinds of markings. All the vertices in $T'$ are marked "1" while all the vertices in $V - V'$ are marked "0." The markings "new" and "old" are for procedures DIST and LABELEDGE. Originally, all the vertices in $T'$ are marked "old." After the execution of DIST, all of them are changed to "new," but they are changed back to "old" again after LABELEDGE.

*Algorithm DE: Deletion of an Edge from a* m.s.t.

*Input.* A m.s.t. $T = (V, E_T)$ represented by adjacency lists $L(v)$ for $v \in V$.

*Output.* A label $l_e(e)$ for each $e \in E_T$.

*Method.* Let $T' = (V', E_T')$ be a connected subgraph of $T$, and $m$ be $\max_{e \in E}\{c(e)\}$.

(1) Initially, all the vertices in $V$ are marked "0" and "new" and $T'$ is empty. Later $T'$ will contain only those vertices marked "1."

(2) Choose any $r \in V$ and mark it "1" and "old," $r$ is the first vertex in $T'$.

(3) Execute LABELEDGE($r$).

*Procedure* LABELEDGE($r$)

*comment*: *$r$ is the new vertex in $T'$, and $w$ will be the next new vertex.*

1. *Begin for* every vertex $w$ on $L(r)$ *do*
2. *if* $w$ is marked "0" *then*
3. *begin* mark $w$ "1"

*comment*: the smallest edges from $w$ to each of the subtree in $T'$ are found in DIST, and they may be the new edges for replacement. Special attention is paid to $(r, w)$ so as to prevent DIST from returning $(r, w)$ as the smallest edge between $w$ and $T'(r)$.

4. temp $\leftarrow c((r, w))$
5. $c((r, w)) \leftarrow m + 1$
6. DIST($r$, $w$)
7. mark $w$ "old"
8. LE($r$, $w$, $w$)

```
9.              c((r, w)) ← temp
10.             LABELEDGE(w)
            end
    end
```

*Procedure* LE($r$, $s$, $w$)

*comment*: This procedure updates the label $l_e(r, s)$ in $T'$ with $w$ being the new vertex in $T'$, and this procedure is called recursively by a depth-first search.

```
11. begin mark r "old"
```

*comment*: except when $s = w$, $l_e(r, s)$ is either unchanged or a new edge joining $w$ to $T'(r)$.

```
12.         if s = w then l_e(r, s) = (w, D(r))
                    else l_e(r, s) = the smaller of (w, D(r)) and l_e(r, s)
13.         for each vertex v on L(r) do
14.             if v is marked "new" and "1" then LE(v, r, w)
    end
```

THEOREM 2. *Algorithm* DE *correctly labels each edge* $e \in E_T$ *and takes* $O(n^2)$ *steps.*

*Proof.* The correctness can be proved by induction on the size of $T'$. It is first necessary to observe that the inputs to DIST and LE are correct, i.e., prior to the execution of line 6 of LABELEDGE the vertices in $T'$ are marked "1" and "old" except for $w$ which is marked "1" and "new" and prior to the execution of line 8 the vertices of $T'$ are marked "1" and "new" except for $w$ which is marked "1" and "old."

It can be seen by enumeration that the algorithm is correct up to $| V' | = 3$. Assume that $l_e(e)$ is correct for $| V' | = k$. Let $w$ be the next vertex adjacent to $r$ to be considered in line 2. By Lemma 4, DIST($r$, $w$) returns the smallest edge joining $w$ and $T'(v)$ for $v \in V' - \{w\}$. Lines 4 and 5 guarantee that DIST will not return edge $(r, w)$. LE($r$, $w$, $w$) updates the labels in $T'$ properly since the new label on each edge is either the old label or the smallest edge between vertex $w$ and the subtree formed by deleting that edge. To prove that the complexity is $O(n^2)$, we must first look at procedure LE. Lines 11 and 12 are executed $| V' | - 1$ times while lines 13 and 14 are executed at most $2(n - 1)$ times. Thus, LE is $O(n)$. The initialization of Algorithm DE is $O(n)$ since markings have to be put on $n$ vertices. In the procedure LABELEDGE steps 1 and 2 are executed at most $2(n - 1)$ times while steps 3 — 10 are executed $n$ times. Since steps 6 and 10 are $O(n)$ the total time spent is at most $O(n^2)$. ∎

The final algorithm to be presented is Algorithm DV which deals with the deletion of a vertex from $G$. Algorithm DV labels each vertex with the edges needed to update the m.s.t. should that vertex be deleted. As in DE, a tree $T'$, indicated by vertices marked "1," is built up one vertex at a time. The procedure LV updates the labels of $T'$ at each iteration when the vertex $w$ is added. This is done by INSERT($w$) on $T''$ with vertices corresponding

to the components of $T'$ after the deletion of $r$. ($T''$ is defined similarly as in Lemma 2.) In order to call INSERT($w$) properly, the labels $l_v(r)$ on each internal vertex $r$ of $T'$ must indicate the tree structure of $T''$ as well as the actual edges being used. As analogous to Algorithm DE, the markings in Algorithm DV serve the same purpose as in DE, and the vertices are visited by a depth-first search.

*Algorithm* DV: *Deletion of a Vertex from a* m.s.t.

*Input.* A m.s.t. $T$ represented by adjacency lists $L(v)$ for $v \in V$.

*Output.* A label $l_v(r)$ for each internal vertex $r$ of $T$, which describes the tree structure to join the disjoint components of $T'$ if vertex $r$ is deleted.

*Method.* Let $T' = (V', E_T')$ be a connected subgraph of $T$. $T'$ will be indicated by those vertices marked "1." Initially all the vertices are marked "0" and "new". Choose any $r \in V$ and mark it "1" and "old." Execute LABELVERTEX($r$).

*Procedure* LABELVERTEX($r$)

*comment*: $r$ is the new vertex in $T'$ and $w$ will be the next new vertex.

```
1.  begin for each vertex w on L(r) do
2.          if w is marked "0" then
3.            begin mark w "1"
4.                  DIST(r, w)
5.                  mark w "old"
6.                  LV(r, w, w)
7.                  LABELVERTEX(w)
            end
    end
```

*Procedure* LV($r$, $s$, $w$)

*comment*: LV updates the label $l_v(r)$ in $T'$ with $w$ being the new vertex in $T'$, and LV is called recursively by a depth-first search.

```
8.  begin mark r "old"
9.    comment: flag is used to indicate whether r is a tip.
9.        flag ← 0
10.       for each vertex v on L(r) do
11.         if v is marked "new" and "1" then
12.           begin flag ← 1
13.                 LV(v, r, w)
              end
```

14. *if* flag = 1 *then*

15. *if* $l_v(r) = \phi$ *then* $l_v(r) = (w, D(x))$ where $(w, r) \in T', x \neq w$

*comment*: $x$ is unique in this case.

16. *else* using the tree $T'' = (V'', E'')$ defined by $l_v(r)$, and the new vertex $w$ with its costs (from array $D$) to each vertex in $V''$ as input, call INSERT($w$). Use the new m.s.t. to update $l_v(r)$.

*end*

THEOREM 3. *Algorithm* DV *correctly labels the vertices of* $T$ *in* $O(n^2)$ *steps.*

*Proof.* Procedure LABELVERTEX is very similar to procedure LABELEDGE. It adds one vertex at a time to $T'$, calls DIST to get the smallest edges between $w$ and the subtree of $T'$, and then calls LV to update the labels on the vertices. To prove correctness of DV we need only look at LV($r$, $s$, $w$) where $w$ is the new vertex inserted. A flag is used in LV to ensure that tips of $T'$ receive no label. Lines 15 and 16 are the key steps of this procedure. Line 15 will be executed only when vertex $r$ is a tip in the old $T'$, and obviously in this case one edge is sufficient for $l_v(r)$. Line 16 is executed when vertex $r$ is already labeled; i.e., when $r$ is an internal vertex in the old $T'$. Obviously, the updating of $l_v(r)$ can be achieved by considering the insertion of $w$ to the m.s.t. $T''$, denoted by $l_v(r)$, which does not contain $w$. Since LV is called for each $r \in V'$, all the labels are updated.

To show the complexity it is sufficient to show that LV requires at most $O(k)$ steps where $k = |V'|$. LV($r$) is called $k$ times and the steps other than the recursive call to itself are proportional to the vertices adjacent to $r$ (note that INSERT($w$) takes $O(|V''|)$ steps where $|V''| =$ the degree of $r$ in $T'$). Since $|E_T'| = k - 1$, LV($r$) takes no more than $O(k)$ steps. The remaining part of the proof is similar to the proof for Theorem 2 and so the total time spent is at most $O(n^2)$. ∎

## 4. CONCLUSION

We have presented algorithms for updating the m.s.t. under insertion or deletion of a vertex or edge in a graph. Algorithm I, which deals with the insertion of a new vertex, yields another $O(n^2)$ algorithm for constructing the m.s.t. when applied successively. Algorithm DV (DE) takes $O(n^2)$ steps to provide enough information to update the m.s.t. for the deletion of any vertex (edge) in the graph. Unfortunately, the deletion cannot be done sucessively, for an arbitrary set of vertices (edges) without repeating the deletion algorithm each time. However, the information provided by DE or DV can be updated in linear time if a new vertex is inserted.

One possible application of the DE algorithm is for the travelling salesman problem. In the approach taken by Held and Karp [5] and improved upon by Hansen and Krarup [6], minimal spanning trees play an important role. In their branch and bound algorithm, a

sub-problem consists of a minimal spanning tree with certain edges fixed as being included or excluded from every future m.s.t. to be generated by this sub-problem. The next sub-problem is generated by first finding for each unfixed edge, the increase in cost of the m.s.t. should that edge be excluded from the tree, and then excluding the edge with the smallest increase. Clearly, the DE algorithm can decide the branching selection used in the sub-program quite efficiently.

## ACKNOWLEDGMENT

The author would like to thank the referee for his constructive comments in improving this paper.

## REFERENCES

1. E. W. DIJKSTRA, A Note on two problems in connection with graphs, *Numer. Math.* 1 (1959), 269–271.
2. P. M. SPIRA AND A. PAN, On finding and updating spanning trees and shortest paths, *SIAM J. Computing* 4 (1975), 375–380.
3. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
4. R. TARJAN, Depth-first search and linear graph algorithms, *SIAM J. Computing* 1 (1972), 146–160.
5. M. HELD AND R. M. KARP, The travelling salesman and minimum spanning trees, II, *Math. Programming* 1 (1971), 6–25.
6. K. H. HANSEN AND J. KRARUP, Improvements of the Held–Karp algorithm for the symmetric Travelling salesman problem, *Math. Programming* 7 (1974), 87–96.