Yale University
Department of Computer Science

Linear Algorithms for Analysis of Minimum Spanning and Shortest Path Trees in Planar Graphs

Heather Booth[1] *Jeffery Westbrook*[2]

YALEU/DCS/TR-763
February 1990

Research partially supported by Office of Naval Research Grant N00014-87-K-0467 and National Science Foundation Grant CCR-8610181.

[1]Department of Computer Science, Princeton University, Princeton, NJ 08544.

[2]Department of Computer Science, Yale University. This research was partially done while the author was at the Department of Computer Science, Stanford University, Stanford, CA 94305.

Abstract

We give a linear time and space algorithm for analyzing trees in planar graphs. The algorithm can be used to analyze the sensitivity of a minimum spanning tree, to changes in edge costs, find its replacement edges, and verify its minimality. It can also be used to analyze the sensitivity of a single-source shortest path tree to changes in edge costs, and to analyze the sensitivity of a minimum cost network flow. The algorithm is simple and practical. It uses the properties of a planar embedding, combined with a heap-ordered queue data structure.

Let $G = (V, E)$ be a planar graph, either directed or undirected, with n vertices and $m = O(n)$ edges. Each edge $e \in E$ has a real-valued cost $cost(e)$. A minimum spanning tree of a connected, undirected planar graph G is a spanning tree of minimum total edge cost. If G is directed and r is a vertex from which all other vertices are reachable, then a shortest path tree from r is a spanning tree that contains a minimum cost path from r to every other vertex.

We consider the following problems:

- Finding the replacement edges of a minimum spanning tree, and verifying its minimality.
- Performing sensitivity analysis of a minimum spanning tree.
- Performing sensitivity analysis of a shortest path tree.
- Performing sensitivity analysis of a minimum cost network flow.

Sensitivity analysis measures the robustness of a minimum spanning tree or shortest path tree by determining how much the cost of each individual edge can be perturbed before the tree is no longer minimal [14, 17].

Let e be some edge in a minimum spanning tree of G. The replacement edge for e is the non-tree edge that replaces e in the minimum spanning tree of $G' = (V, E - e)$. Finding replacement edges is an important subproblem of determining the k smallest spanning trees of a graph [4, 6]. Given the set of replacement edges, we may verify the minimality of a spanning tree.

Sensitivity analysis of shortest paths and network flows has been studied by Shier and Witzgall [14] and Gusfield [10]. The fastest known algorithms for all these problems are due to Tarjan [17, 16] and run in time and space $O(m\alpha(m, n))$, where α is the functional inverse of Ackermann's function. Gabow [7] also achieves these bounds.

Here we show that in the special case of planar graphs, these problems can be solved in $O(n)$ time and space.

Our result also remedies a lacuna in Fredrickson's proof of Theorem 9, reference [6], which is incorrect without an $O(n)$ algorithm for finding replacement edges in a planar graph.

The above problems can all be solved by an algorithm for what we call the *critical edge problem.* We are given an undirected planar graph G containing a spanning tree T, rooted at vertex r. We allow G to have multiple edges and loop edges, but for convenience we will continue to call G a graph, rather than multigraph or pseudograph. For each vertex v we wish to determine the minimum-cost non-tree edge with exactly one endpoint a descendant of v. We call this edge the *critical edge* for vertex v. In this paper we first give a critical edge algorithm and then describe its application to the problems listed above.

1 Preliminaries

We assume we are given an embedding of $G = (V, E)$ in the plane in which r, the root of T, is on the outer face. Such an embedding always exists (see [11, page 105]) and can be generated in $O(n)$ time using the algorithms of Hopcroft and Tarjan [12] or Booth and Leuker [1] (see Chiba et al. [3]). The embedding of G specifies the order in which edges incident to $v \in V$ are encountered as we walk around v in the counterclockwise direction (this ordering defines the embedding). We assume a standard representation of the embedding in which the counterclockwise successor of an edge around a vertex can be found in constant time.

For convenience of exposition, we define the directions "up", "down", "left" and "right" in the plane so that r is the uppermost vertex. Since a loop edge cannot be a critical edge, we assume that any loops in G have been removed in an $O(n)$ preprocessing stage. If u and v are vertices, $\{u, v\}$ denotes the undirected edge with endpoints u and v, (u, v) denotes a directed edge from u to v and $p(v)$ denotes the parent of v in T.

We assign preorder and postorder numbers to the vertices of T according to a *topological depth-first search.* For each vertex v, a linear edge list $\{e_0, e_1, \ldots, e_d\}$ is constructed, where d is the degree of v. Edge e_0 is the edge from v to its parent in T and the remaining edges $e_1, \ldots, e_d$ are listed in the order they are encountered by walking counterclockwise around v ¿from e_0. At the tree root r, e_0 is the edge such that e_0 and its counter-clockwise predecessor e_d both lie on the outer face. The depth-first search visits and numbers each child of v according to the order it appears in the edge list. (Note that the edge list contains both tree and non-tree edges.) Figure 1 gives an example.

We denote the preorder and postorder numbers of v by $pre(v)$ and $post(v)$, respectively. It is well-known (see e.g. [15]) that for any pair u and v of vertices, v is an ancestor of u if and only if $pre(v) \leq pre(u)$ and $post(u) \leq post(v)$.

Let f be a non-tree edge $\{u, v\}$. We denote by $nca(f)$ the nearest common ancestor of u and v. Edge f is a potential critical edge for every tree edge on the tree path connecting u and v. The cycle induced by f together with the tree path between u and v separates the plane into an interior and an exterior region. We denote the interior region by $R(f)$. Given a tree edge $e = \{v, p(v)\}$ in the boundary cycle of $R(f)$, we

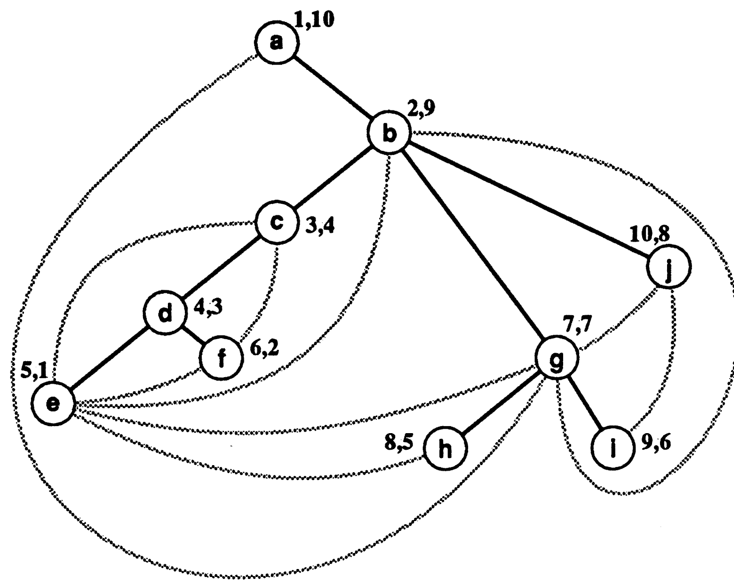

Figure 1: A planar graph with spanning tree rooted at a. Solid edges are tree edges. Vertices have been labeled by the preorder and postorder numbers given by a topological depth-first search. Edge $\{e, g\}$ is a right edge of e and a left edge of g. Edge $\{g, b\}$ is right edge of g and a dead edge of b.

say $R(f)$ is a left or right region of e if $R(f)$ lies to the left or right, respectively, of e as we look along e from v to $p(v)$.

The edges incident to a given vertex v are partitioned into three classes. Edge f is called a *dead edge* of v if $v = nca(f)$ or if $f = \{v, p(v)\}$. Note that every tree edge is a dead edge. If f is not a dead edge of v, it is called a *left edge* of v if $R(f)$ is a left region of the tree edge $\{v, p(v)\}$, and a *right edge* of v otherwise. Thus a non-tree edge f is a left edge of one endpoint and a right edge of the other, unless the endpoints are related, in which case f is a dead edge of the ancestor endpoint and a left or right edge of the descendent endpoint. Note that if f is a left edge of v, then $R(f)$ is a left region of all vertices on $P(v, f)$. The analogous property holds for right edges.

We can determine whether an edge $f = \{u, v\}$ is a dead, left, or right edge of v by using the topological preorder and postorder numbering in the following manner. If v is an ancestor of u or if $u = p(v)$, then f is dead. If u and v are unrelated then f is a left edge if $pre(u) < pre(v)$ and a right edge otherwise. If u is an ancestor (but not parent) of v, then let $e = \{u, x\}$ be the first tree edge following f in the edge list of u. If there is no such tree edge, or if $pre(x) > pre(v)$, then f is a right edge of v; otherwise f is a left edge. As mentioned above, ancestor queries can be answered in constant time using the preorder and postorder numbering.

Lemma 1 *Let f be a left edge of u and let g be a non-tree edge that precedes f in the edge list of u, i.e., is encountered before the tree edge $\{u, p(u)\}$ in clockwise order around u from f. Then g is also a left edge, and $nca(f)$ is a (not necessarily proper) ancestor of $nca(g)$.*

Proof. Since g lies between f and $\{u, p(u)\}$ in clockwise order, g must be contained wholly within the region $R(f)$, by the assumption of planarity. This implies that $R(g)$ is a left region of $\{u, p(u)\}$. Let $g = \{u, w\}$. Either w is on the boundary of $R(f)$ or lies within $R(f)$; i.e., is a proper descendent of a node on the boundary. Since all boundary nodes are descendents of $nca(f)$ by definition, $nca(g)$ must also be a descendent of $nca(f)$. □

The analogous result holds for right edges.

Let v be a leaf of T with edge list $\{e_0, f_1, f_2, \ldots, f_d\}$, where $e_0 = \{v, p(v)\}$ and $f_1 \ldots f_d$ are non-tree edges. Lemma 1 implies that there is an index ℓ, $0 \le \ell \le d$, such that the edges in $\{f_1, \ldots, f_\ell\}$ are left edges and the edges in $\{f_{\ell+1}, \ldots, f_d\}$ are right edges. Furthermore, let f_i and f_j be edges such that $1 \le i < j \le d$. Then $nca(f_j)$ is an ancestor of $nca(f_i)$ for $i, j \le \ell$ (the left edges) while $nca(f_i)$ is an ancestor of $nca(f_j)$ for $i, j > \ell$ (the right edges).

Let $e = \{u, v\}$ be a tree edge, with $v = p(u)$. A *contraction* of e, shrinks u up into v leaving only the single vertex v. The new edge list of v is constructed by removing e from the list of u and v and inserting the edge list of u into the edge list of v at the position formerly occupied by e. Edge contraction preserves planarity [13, Lemma 1].

and the edge list produced by the contraction specifies a valid embedding. It may, however, produce new loop and multiple edges.

2 Critical Edge Algorithm

The algorithm is based on the approach of Shier and Witzgall [14].

If $v \in T$ is a leaf its critical edge is simply the minimum cost edge in its edge list, excluding the tree edge from v to its parent, which we will ignore from now on.

To compute the critical edges for the remaining vertices, we construct a series of graphs $G_0, G_1, \ldots, G_j$ with corresponding spanning trees $T_0, T_1, \ldots, T_j$, where G_0 is G minus its loop edges and j is the number of non-leaf vertices in G_0. Graph G_i is constructed from G_{i-1} by the following procedure:

1. Choose any vertex v in G_{i-1} all whose children in T_{i-1} are leaves.
2. Contract the edges from v to its children and delete any resultant loop edges. This produces graph G_i and tree T_i in which v is a leaf.
3. Set *critical(v)* to be the minimum weight non-tree edge incident to v in G_i.

The correctness of the algorithm is proved in [14] and is easily seen. For a vertex $v \in G$, let the *relevant* edges denoted $rel(v)$, be the set of non-tree edges with exactly one endpoint in the subtree of T rooted at v. By definition, $critical(v)$ is the minimum weight edge in $rel(v)$. A simple induction on i shows that when v becomes a leaf in stage i, its edge list contains exactly its relevant edges.

So far our algorithm does not particularly depend upon the planarity of G; this algorithm solves the critical edge problem in any general graph, and can be implemented in $O(m \log n)$ time using a mergeable heap data structure to store the edges at each vertex [10]. To further improve the running time of the algorithm to $O(n)$ in the planar case, we take advantages of the properties discussed in Section 1.

Given G and T rooted at r, we first perform a topological depth-first search on T as described in Section 1, computing and storing preorder and postorder numbers and constructing the edge lists for each vertex. As part of this preprocessing, we determine whether each edge is dead, left, or right with respect to each of its endpoints. This requires two scans of the edge lists, one to determine for each non-tree edge the first subsequent tree edge in its edge list, and one to classify each edge using the constant-time test described in Section 1. These scans can be combined with the topological depth-first search. Loop edges can be removed at the same time.

Let v be a leaf with edge list $\{e_0, f_1, f_2, \ldots, f_d\}$. Lemma 1 implies that there is an index ℓ such that all edges f_1 to f_ℓ are left edges of v and all edges $f_{\ell+1}$ to f_d are right edges of v. Using ℓ we split the edge list into two lists $L_v = f_1, f_2, \ldots, f_\ell$ and $R_v = f_d, f_{d-1}, \ldots, f_{\ell+1}$. By Lemma 1 the edges in L_v and R_v are *nca-ordered*, i.e., if

edges a and b belong to the same list and edge a precedes edge b, then $nca(a)$ is a descendant of $nca(b)$.

At each stage of the processing, our algorithm explicitly maintains the two nca-ordered lists L_v and R_v for each leaf node v. The lists for each leaf of the initial tree T are constructed during the preprocessing. The lists are maintained in a *heap-ordered concatenable queue* data structure that supports the following operations:

1. *make queue*(x): Create and return a new queue containing the single element x.
2. *pop*(q) : Delete and return the first item from queue q.
3. *concatenate*(q_1, q_2): Return the queue formed by concatenating q_2 to the back of q_1.
4. *find min*(q): Return the item of minimum weight in q without removing it from q. If q is empty return null.
5. *first*(q): Returns the first element in q without removing it. If q is empty return null.

Let v be a non-leaf vertex of G processed in the ith stage. Any child u of v must be a leaf, with left and right lists L_u and R_u, respectively. In Step 2 of the above procedure, the edges from v to its children are contracted, and two lists L_v and R_v are constructed from the non-tree edges incident to v and from the left and right lists of the children of v. The union of L_v and R_v is exactly the set of relevant edges for v. Step 3 is simply performed by finding the minimum of *find min*(L_v) and *find min*(R_v). Thus the bulk of the work occurs in Step 2.

Let $\{e_0, e_1, \ldots, e_d\}$ be the edge list of v. To begin Step 2, with each edge e_i, $i > 0$, we associate two lists L_i and R_i. If e_i is a tree edge $\{u, v\}$, where u is a child of v, then $L_i = L_u$ and $R_i = R_u$. If e_i is a left edge then $L_i =$ *make queue*(e_i) and $R_i =$ *make queue*($\emptyset$). If e_i is a right edge then then $R_i =$ *make queue*(e_i) and $L_i =$ *make queue*($\emptyset$). If e_i is a dead non-tree edge, i.e., if both endpoints of e_i are descendants of v, then both L_i and R_i are empty queues.

Next we delete the loop edges formed by the contractions. The loop edges are those edges f with $nca(f) = v$. Since each left or right list is nca-ordered, all such loop edges are grouped at the front of the lists. Let $f = \{x, y\}$ be a loop edge formed by the contractions. Edge f is not a relevant edge of v, and hence in the original tree $T_0 = T$ both x and y must be descendents of v. This can be tested in $O(1)$ time using the preorder and postorder numbers, as described in Section 1. Thus to delete loop edges, we simply examine each list and pop edges off until reaching the first edge whose nearest common ancestor is not v, i.e., the first that is a relevant edge of v.

After deleting loop edges, the collection of left and right lists contains only relevant edges. We form L_v by concatenating the left lists from left to right and form

R_v by concatenating the right queues ¿from right to left. That is, we form L_v by initializing L_v to the empty queue and then performing $L_v = \mathit{concatenate}(L_v, L_i)$ for $i = 1, 2, \ldots, d$. The same is done for R_v, except the index runs from d to 1. Code for the general processing step is given in Figure 2.

The edge list of v in G_i, after performing the concatenations, contains the edges of L_v in order followed by the edges of R_v in reverse order. Since G_i is planar and v is a leaf, Lemma 1 implies that L_v and R_v are nca-ordered and, further, that there is an index ℓ such that for $i < \ell$, R_i is empty and for $i > \ell$, L_i is empty.

Let d be the degree of v. Excluding the work involved in popping loop edges, the processing of vertex v requires $O(d)$ queue operations plus $O(d)$ additional work. The total number of pops is $O(n)$, since each edge is inserted into at most two queues and so can be popped at most twice. The preprocessing phases requires $O(n)$ time. If each queue operation takes $O(1)$ amortized time, then the total running time of the algorithm is $O(n)$.

We now describe the heap-ordered concatenable queue data structure. The data structure we present is a simple extension of ideas presented by Gajewska and Tarjan [9]. The items in queue q are stored in a linked list, with additional pointers to the front and back items. In order to answer the *find min* queries a second list r is maintained, consisting of the *rightward minima* of q. The first rightward minima is the minimum element of the entire list q. The ith rightward minima is the minimum element occurring after the $(i-1)^{st}$. We store r as a doubly-linked list, with pointers to the first and last elements. To make a new queue containing item x, we initialize both lists to contain x. The operations *first*(q) and *find min*(q) are implemented in $O(1)$ worst-case time by using the pointer to the front of the appropriate linked list. The operations *pop* and *concatenate* are implemented as follows:

- *pop*(q) : Delete the first element from q and if it is also the first element of r, delete it from r.

- *concatenate*(q_1, q_2): Link the list of q_2 to the back of q_1. Let y be the first element of r_2. Remove the elements of r_1 from the last element forward, until reaching an element x with $\mathit{cost}(x) < \mathit{cost}(y)$. Link y to x, concatenating r_2 to the back of the modified r_1. Reset all pointers to first and last elements appropriately.

The worst-case time required for *pop* is $O(1)$. The time to perform a concatenation is $O(1)$ plus the number of items removed from the rightward minima list of the first queue. The removal of an element x is charged to the *make queue* that created x. Once x is removed from the minima list it can never be put back on it; x is removed because there is some other item y of lesser key following it in the queue, and y cannot be popped before x. Thus the amortized time per *make queue* and *concatenate* is $O(1)$. This in turn implies that the critical edge algorithm runs in time $O(n)$. The space required is also $O(n)$, since at any time each edge appears in at most two lists.

```
ProcessVertex(v : vertex) begin
/* Let v have edge list {e_0, f_1, f_2, ..., f_d} */

/* Delete loop edges */
   for i = 1 to d do begin
      if f_i is a tree edge
         while nca(first(L_i)) = v and L_i ≠ ∅
            pop(L_i);
         while nca(first(R_i)) = v and R_i ≠ ∅
            pop(R_i);
   end
/* Initialize lists */
   for i = 1 to d do begin
      if f_i is a non-tree edge begin
           L_i = make queue(∅);
           R_i = make queue(∅);
           if f_i is a left edge
              L_i = concatenate(L_i, make queue(f_i));
           if f_i is a right edge
              R_i = concatenate(R_i, make queue(f_i));
      end
   end
/* Compute L_v and R_v */
   L_v = make queue(∅);
   R_v = make queue(∅);
   for i = 1, 2, ..., d do L_v = concatenate(L_v, L_i)
   for i = d, d − 1, ..., 1 do R_v = concatenate(R_v, R_i)
/* Compute critical(v) */
   set critical(v) = min{find min(L_v), find min(R_v)}
end
```

Figure 2: Algorithm for processing one vertex.

3 Minimum Spanning Tree Analysis

In this section we use the critical edge algorithm to solve the following problems in linear time: finding all replacement edges of a minimum spanning tree, verifying the minimality of a spanning tree, and performing sensitivity analysis on the edges of a minimum spanning tree. The following lemma will be useful.

Lemma 2 [6, 16] *Tree T is a minimum spanning tree if and only if for each non-tree edge $f = \{u, v\}$, the cost of f is greater than or equal to the cost of each edge on the path from u to v.*

Let T be the minimum spanning tree of $G = (V, E)$. As defined in Section 1, the replacement edge of edge $e \in T$ is the non-tree edge f that replaces e in the minimum spanning tree of $G' = (V, E - e)$. The removal of edge e breaks T into two fragments T' and T''. It is well-known ([16]) that the replacement edge f is that edge with minimum weight edge in the cut induced by (T', T''). We arbitrarily root T at some vertex r. Then for each tree edge $e = \{v, p(v)\}$, f is the minimum cost edge with one endpoint in the subtree rooted at v. Thus to find replacement edges we run the critical edge algorithm and set the replacement edge of e to be $critical(v)$.

If the cost of a tree edge is greater than the cost of its replacement edge, then a spanning tree of smaller total cost can be constructed by replacing the tree edge with its replacement edge. If T is a minimum spanning tree, however, this situation cannot occur; thus the replacement edges can be used to to verify the minimality of T. (An alternative method is to run the $O(n)$-time algorithm for computing minimum spanning trees of Cheriton and Tarjan [2].)

To analyze the sensitivity of a minimum spanning tree T we determine for each edge e how much its cost can be perturbed before T is no longer minimal. We compute lower and upper bounds $[a, b]$ such that T remains minimal as long as $a \leq cost(e) \leq b$. If e is an edge in T, then the lower bound is $-\infty$. The above discussion implies that the upper limit is the cost of the replacement edge of e. Now consider a non-tree edge $f = \{u, v\}$. The upper limit for the cost of f is $+\infty$. By Lemma 2, the lower limit for $cost(f)$ is the cost of the maximum cost edge on the path from u to v. To compute the non-tree edge lower bounds in linear time we find critical edges in the *dual graph* $G^* = (V^*, E^*)$ of G.

Let S denote the planar subdivision given by the embedding of G. For each face in S there is a corresponding vertex in G^* and for each edge e in G there is a dual edge e^* connecting the dual vertices representing the two faces adjacent to e in S. Thus G^* is dependent on the embedding of G. The dual graph is clearly planar; an embedding S^* is given by placing each dual vertex inside the face it represents and placing each dual edge so that it crosses only the primal edge corresponding to it. We set $cost(e^*) = cost(e)$. The planar graph representation scheme of [?] simultaneously maintains both primal and dual graphs; in any case, given the embedding of G the

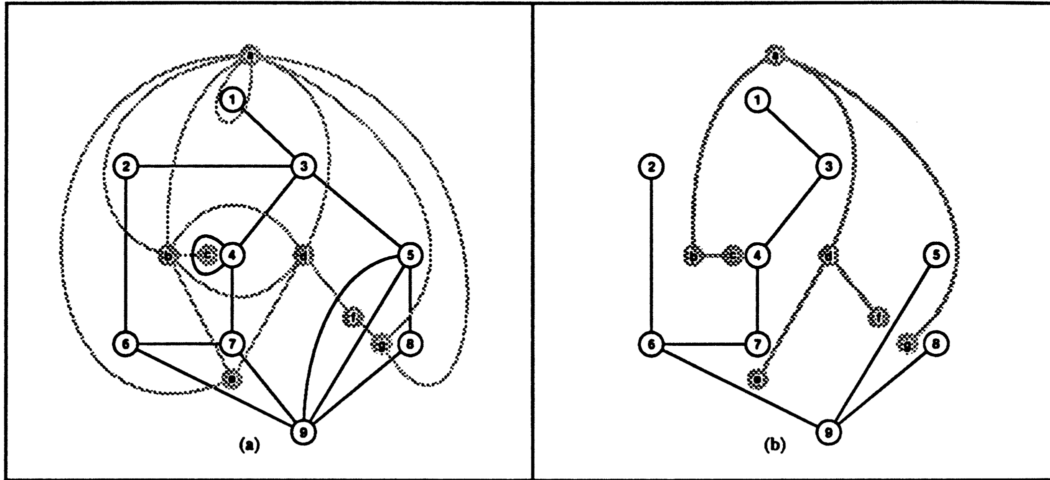

Figure 3: (a) A planar graph (bold) and its dual (shaded). (b) Primal and dual spanning trees for the graph of (a).

dual can easily be computed in $O(n)$ time. Further discussion of dual graphs can be found in Harary [11]. An example is given in Figure 3.

Lemma 3 [5, 8] *Given a spanning tree T in G, let T^* be the set of dual edges $\{e^*|e$ is not in $T\}$. The set T^* is a spanning tree for G^* and, furthermore, T is a minimum spanning tree for G if and only if T^* is a maximum spanning tree for G^*.*

Lemma 4 *Let $f = \{u, v\}$ be a non-tree edge in G; hence its dual f^* is a tree edge in T^*. The replacement edge for f^* in G^* is exactly the dual of the maximum cost edge on the path between u and v in T.*

Proof. As in Section 1, let $R(f)$ denote the interior region of the plane bounded by f and the path in T from u to v. Removal of f^* breaks T^* into two fragments, $T^{*\prime}$ and $T^{*\prime\prime}$. All the vertices of one of these fragments (which are faces in the embedding of G) lie inside $R(f)$. Thus the edges in the cut $(T^{*\prime}, T^{*\prime\prime})$ are exactly the duals of the boundary edges of $R(f)$. The replacement edge for f^* is the minimum weight edge in this cut. □

By Lemma 4 the lower limits for the non-tree edges can be computed by finding replacement edges in the dual graph and setting the lower bound for each non-tree edge to be the cost of its dual replacement edge. The result of this section are summarized in the following theorem.

Theorem 1 *The problems of computing replacement edges and determining the sensitivity of a minimum spanning tree of a planar graph can be solved in $O(n)$ time and space.*

4 Shortest Path Tree Analysis

Let G be a directed graph whose edges each have an associated cost and let T be a single-source shortest path tree ¿from source vertex s. In shortest path sensitivity analysis we are interested in finding bounds $[a, b]$ on the cost of each directed edge e such that, in the absence of other changes, T remains a shortest path tree for $a \leq cost(e) \leq b$.

Let $d(v)$ denote the distance from s to v, which is the sum of the costs of the edges on the path from s to v in T. The following lemma is well-known.

Lemma 5 [15, 18] *A spanning tree T in G is a shortest path tree if and only if for all non-tree edges $e = (u, v)$, $d(u) + cost(e) \geq d(v)$.*

Let $e = (u, v)$ be a non-tree edge. By Lemma 5, T remains a shortest path tree for $(d(v) - d(u)) \leq cost(e) \leq +\infty$. Now consider a tree edge $e = (p(v), v)$. Changing $cost(e)$ by Δ changes the distances $d(v)$ for all nodes x in T_v, the subtree rooted at v. Lemma 5 implies that for T to remain a shortest path tree, Δ must satisfy the following constraints:

1. for each non-tree edge $f = (x, y)$ such that $x \in T_v$ and $y \notin T_v$ (the *outgoing* edges), $\Delta \geq d(y) - d(x) - cost(f)$
2. for each non-tree edge $f = (x, y)$ such that $y \in T_v$ and $x \notin T_v$ (the *incoming* edges), $\Delta \leq d(x) - d(y) + cost(f)$.

For each non-tree edge $f = (x, y)$ we compute a transformed cost $cost'(f) = cost(f) + d(x) - d(y)$. Then the lower bound on the cost of tree edge $e = (p(v), v)$ is $cost(e) - cost'(f_O)$, where f_O is the edge going out from T_v of minimum transformed cost. The upper bound for e is $cost(e) + cost'(f_I)$, where f_I is the edge coming in to T_v of minimum transformed cost. To compute f_O for each vertex v we initialize the edge lists of each vertex to contain only outgoing edges and run the critical edge algorithm, setting $f_O = critical(v)$. To compute f_I we initialize the edge lists to contain only incoming edges and again run the critical edge algorithm, setting $f_I = critical(v)$. (Note that these initializations preserve the planarity of G.)

Theorem 2 *Sensitivity analysis of a single-source shortest path tree in a planar graph can be performed in $O(n)$ time and space.*

5 Minimum Cost Network Flow

We consider the network flow problem in which each edge e of the network G has upper and lower capacity bounds $[l, u]$ and a cost per unit flow across e, and each vertex has a demand $D(v)$. If $D(v) > 0$, v is a *source*; if $D(v) < 0$, v is a *sink*. A

minimum cost flow in G assigns flow values $x(e)$ to the directed edges of G that satisfy the flow constraints and minimize the sum over all edges of $x(e)cost(e)$. Sensitivity analysis determines how much the edge costs can be perturbed without changing the optimality of the flow.

If there is any feasible flow there is an optimal flow with *basis* T; T is a spanning tree of G such that any non-tree edge has flow $x(e) = l(e)$ or $x(e) = u(e)$. There exists a price function on the nodes π such that the transformed cost of $e = (x, y)$, $cost'(e) = cost(e) + \pi(x) - \pi(y)$, is zero for all tree edges. The flow is optimal if and only if for all non-tree edges f, $cost'(f) \geq 0$ if $x(e) = l(e)$ and $cost'(f) \leq 0$ if $x(e) = u(e)$ [14].

If we root T at some vertex r, we can regard T as a shortest path tree from r, with $\pi(v)$ the distance from r in the tree, by replacing any edge e pointing up the tree by a reversed edge e' with cost $-cost(e)$. Then the sensitivity of the flow can be found by computing the sensitivity of the shortest path tree, reversing upper and lower bounds for reversed edges. Further details can be found in [14].

Theorem 3 *Sensitivity analysis of a minimum cost network flow in a planar network can be performed in $O(n)$ time and space.*

6 Acknowledgements

We would like that thank Robert E. Tarjan for his insightful comments.

References

[1] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.*, 13:335–379, 1976.

[2] D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742, 1976.

[3] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. Comput. System Sci.*, 30:54–76, 1985.

[4] D. Eppstein. Finding the k smallest spanning trees. Manuscript, 1989.

[5] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, to appear. Submitted to SODA special issue of *Journal of Algorithms.*