Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph*

David Eppstein† *Giuseppe F. Italiano*‡ *Roberto Tamassia*§

Robert E. Tarjan¶ *Jeffery Westbrook*‖ *Moti Yung***

November 11, 1991

Abstract

We give an efficient algorithm for maintaining a minimum spanning forest of a plane graph subject to on-line modifications. The modifications supported include changes in the edge weights, and insertion

*Research supported in part by NSF grant CCR-88-14977, NSF grant DCR-86-05962, NSF grant CCR-90-09753, ONR Contract N00014-87-K-0467, DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) a National Science Foundation Science and Technology Center, grant NSF-STC88-09648, and Esprit II Basic Research Actions Program of the European Communities Contract No. 3075. A preliminary version of this article appeared in the Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, held in San Francisco, CA, January 1990.

†Department of Information and Computer Science, University of California, Irvine, CA 92715. This work was done while the author was at the Department of Computer Science, Columbia University, New York, NY 10027.

‡Department of Computer Science, Columbia University, New York, NY 10027 and Dipartimento di Informatica e Sistemistica, Università di Roma, Rome, Italy. Partially supported by an IBM Graduate Fellowship.

§Department of Computer Science, Brown University, Box 1910, Providence, RI 02912-1910.

¶Department of Computer Science, Princeton University, Princeton, NJ 08544, and NEC Research Institute, Princeton, NJ 08540.

‖Department of Computer Science, Yale University, New Haven, CT 06520. This work done while the author was at the Department of Computer Science, Princeton University.

**IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

and deletion of edges and vertices which are consistent with the given embedding. To implement the algorithms, we develop a data structure called an *edge-ordered dynamic tree*, which is a variant of the dynamic tree data structure of Sleator and Tarjan. Using this data structure, our algorithm runs in $O(\log n)$ time per operation and $O(n)$ space. The algorithm can be used to maintain the connected components of a dynamic planar graph in $O(\log n)$ time per operation. We also show that any algorithm will need $\Omega(\log n)$ amortized time per operation, given a set of machine operations that is fairly general.

1 Introduction

Let $G = (V, E)$ be an undirected plane graph (a planar graph embedded in the plane). Let $w(\mathbf{e})$ be a real-valued weight for each edge $\mathbf{e} \in E$. A minimum spanning forest for G is a spanning forest (a set of spanning trees, one for each connected component) that minimizes the sum of the weights of the edges in the forest. A maximum spanning forest is defined analogously. We consider the problem of maintaining a representation of a minimum or maximum spanning forest in G while processing on-line a sequence of update operations. An update operation may be a *change weight*$(\mathbf{e}, \Delta)$ operation, which adds real number Δ to the weight of the graph edge $\mathbf{e}$, or an operation that changes the structure of G, such as the insertion (or deletion) of an edge (or a vertex). A structural modification must be consistent with the current embedding; for example, edge $\mathbf{e} = \{u, v\}$ can be inserted only if u and v lie on a common face. Our representation will allow us to answer a number of queries about the minimum spanning forest, such as the current weight of each tree, whether an edge $\mathbf{e}$ is currently a spanning edge, and if so, which tree it belongs to.

Dynamic problems on graphs have been extensively studied. Several algorithms have been proposed for maintaining fundamental structural information about dynamic graphs, such as connectivity [9, 10, 15, 24, 26], transitive closure [17, 18, 19, 20, 21, 34, 23], and shortest paths [1, 8, 25, 28, 34]. Dynamic planar graphs arise in communication networks, graphics, and VLSI design, and they occur in algorithms that build planar subdivisions such as Voronoi diagrams. Algorithms have been proposed for maintaining the embedding of a planar graph [29] and for incremental planarity testing [2, 3]. The dynamic minimum spanning tree problem has been considered by Spira and Pan [28], Chin and Houck [7], Frederickson [10], and Gabow and Stallmann [11]. Frederickson gives an algorithm based on "topology trees" that runs in $O(\sqrt{m})$ time per operation on general graphs, and $O((\log n)^2)$ time on plane graphs. As Frederickson notes, the minimum spanning tree for a general graph being modified on-line by edge additions alone can be main-

tained in $O(\log n)$ amortized[1] or worst-case time per operation, using the dynamic tree data structure of Sleator and Tarjan [26]. Gabow and Stallmann [11] improve Frederickson's bound for planar graphs to $O(\log n)$ time per operation for the case of a fixed graph with changing edge weights. Their method also uses the dynamic tree data structure.

In this paper we present a data structure and an algorithm for maintaining a minimum spanning forest of an edge-weighted subdivision the plane subject to both edge weight changes and an extended set of modifications which permit the underlying structure to change dynamically. This is an important extension as it allows fast maintenance of a dynamically changing structure; a good example of such a structure is the map of Europe (which was, is, and probably will be subject to dynamic changes)— borders are redrawn in a manner that is consistent with the previous map. Our algorithm extends the approach of Gabow and Stallmann [11]. The subdivision is allowed to contain loop edges or multiple edges. Our algorithm runs in $O(m)$ space and $O(\log m)$ amortized time per operation, where m is the number of edges in the subdivision. We can maintain a minimum spanning forest of an n-vertex plane graph G in time $O(\log n)$ per update by using our subdivision algorithms. Our algorithm can also be used to maintain the connected components of a dynamic plane graph.

This paper addresses two questions: first, what is the correct framework to use in describing a dynamic plane graph, and second, how does one implement the desired operations? To describe and manipulate the dynamic plane graph, we use the subdivision representation scheme of Guibas and Stolfi [13], which we describe in more detail in Section 2. This scheme provides a pair of simple, powerful primitives from which more complicated operations such as the insertion or deletion of edges can be built. Our spanning tree algorithm is built on top of this framework.

For the sake of completeness we describe in detail the Gabow-Stallmann $O(\log m)$-time algorithm for the restricted setting of a fixed graph that is undergoing only edge weight changes on-line. To extend this scheme to dynamic subdivisions, each minimum spanning tree is maintained with a variant of the dynamic tree data structure of Sleator and Tarjan [26, 27] called an *edge-ordered dynamic tree*. This data structure is used to represent free trees in which for each vertex there is a total ordering of the incident edges. It can support much the same operations as Sleator-Tarjan dynamic trees, with the addition of operations to split and condense vertices while preserving the edge ordering. Depending upon the needs of the application, this repertoire of operations can be used to test membership of an edge in the spanning

[1]The amortized cost of an operation is the cost of a worst-case sequence of operations divided by the number of operations in the sequence. See [32] for a general discussion of amortization.

forest in $O(1)$ time, and to determine the spanning tree containing a given vertex, or find the edge of maximum or minimum weight on the tree path between two vertices, in $O(\log m)$ time. The edge-ordered tree also finds use in the on-line planarity testing algorithm of Di Battista and Tamassia [2, 3]. Thus our data structure is fairly general and powerful. The algorithms can be made to run in worst-case time $O(\log m)$ with the biased tree implementation of dynamic trees [26]. We also argue that in our machine model, any algorithm must spend $\Omega(\log n)$ amortized time per operation; we do this by reduction to sorting.

Our algorithms do not solve the dynamic minimum spanning tree problem when we allow the following dynamic operations on *planar* (rather than plane) graphs: insert a new vertex; delete a disconnected vertex; delete an edge; and insert an edge if the resultant graph remains planar. Our algorithms use properties of planar embeddings, and even if planar graph G_1 can be derived from another G_2 by a single edge addition, a non-constant number of modifications to the subdivision that embeds G_1 may be required to build a subdivision that embeds G_2. Currently no solution for this problem is known that achieves sub-linear time per dynamic operation.

2 Background

In this section we describe our graph representation and the basic algorithms and operations we use.

2.1 Planar Subdivisions and Their Representation

A plane graph or subdivision S of the plane is a connected set of vertices and edges that partition the plane into a collection of faces. S may have loop edges or multiple edges between vertices. We are interested only in the topology of S and do not consider the actual geometric positions of the vertices and edges. Let G be a planar graph. An embedding of G generates a collection of subdivisions, one for each connected component of the graph. If G is triconnected then the topological structure of its embedding is unique up to mirror image [14, pp. 105], but in general there are multiple embeddings possible for a given planar graph. Using the topological incidence relationship between edges and faces of S, we define the *dual graph* $G^* = (F, E^*)$ [14]. Each face of S gives rise to a vertex in F. Dual vertices f_1 and f_2 are connected by a dual edge $\mathbf{e}^*$ whenever primal edge $\mathbf{e}$ is adjacent to the faces of S corresponding to f_1 and f_2. The dual graph can be embedded in the plane by placing each dual vertex inside the corresponding face of S, and placing dual edges so that each one crosses only its corresponding primal

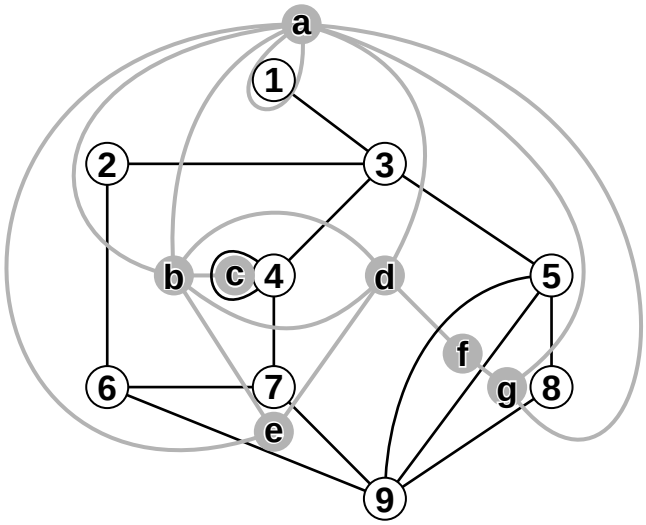


Figure 1: A subdivision (black) and its dual (grey).

edge. This embedding is called the dual subdivision S^*. Figure 1 gives an example of a subdivision and its dual.

Guibas and Stolfi [13] propose the following notation (and corresponding data structure) for describing a subdivision S. Each undirected edge $\mathbf{e} = \{u, v\}$ of the S can be directed in two ways. If e is the directed version of $\mathbf{e}$ originating in u and terminating in v, then $sym(e)$ is the version of $\mathbf{e}$ directed from v to u. Note that if $\mathbf{e}$ is a loop edge, u and v are identical, but we may still define e and $sym(e)$ as oppositely directed versions of the same undirected edge. The operator $orig(e)$ gives the vertex at which directed edge e originates. Thus we can use directed edges to specify vertices of G. Each directed edge has a left and right face as we look along its direction.

As in the primal subdivision, each undirected dual edge generates two directed edges of S^*; the *sym* and *orig* operators are extended to these dual directed edges. The operator $rot(e)$ gives the dual directed edge that originates in the right face of e and terminates in the left face, i.e., it is e rotated 90° counterclockwise. Similarly, $rot^{-1}(e)$ is the directed dual edge from the left face of e to the right face of e. For a given undirected edge $\mathbf{e}$ in the primal subdivision S, we denote the two pairs of primal and dual directed edges by e_0, e_1, e_2, e_3, where e_0 is a primal directed edge and $e_{i+1 \bmod 4} = rot(e_i)$, $0 \leq i \leq 3$.

The *edge ring* of a vertex v is a circular list of the directed edges originating at v, organized in counterclockwise order around v. By $next(e)$ we denote

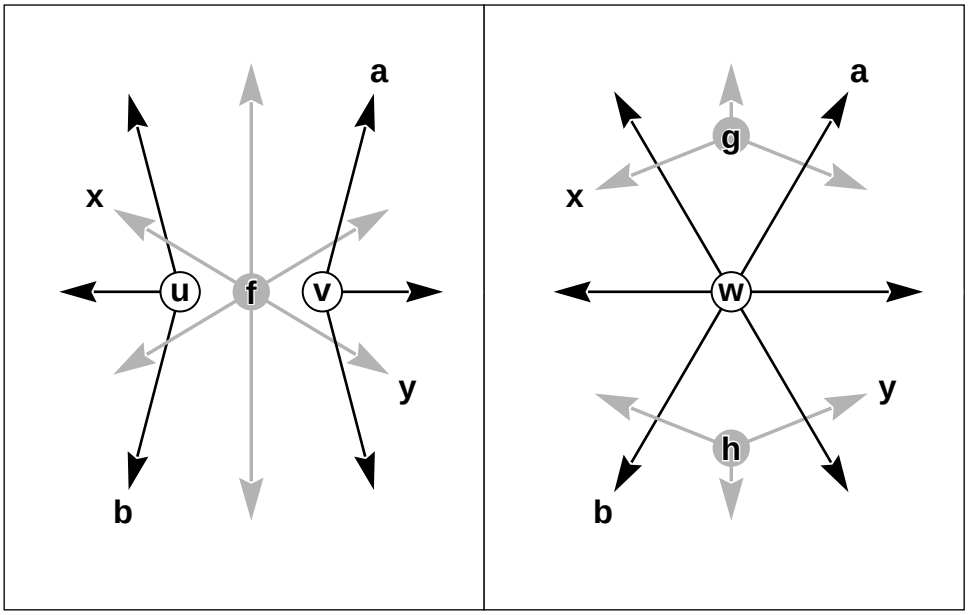
the directed edge following e in counterclockwise order around $orig(e)$, and by $next^{-1}(e)$ the edge preceding e in counterclockwise order. If v has only one incident undirected edge **e**, then its edge ring contains the single directed edge e originating at v, and $next(e)$ is e. On the other hand, for a loop edge **e** both e and $sym(e)$ belong to the edge ring of vertex $orig(e)$.

Next, we discuss the implementation of dynamic operations that affect the structure of planar subdivisions. Guibas and Stolfi [13] introduce two modification primitives, *make edge*, which increases the complexity of the structure by adding new unconnected vertices and edges, and *splice*, which changes the topology of the structure but does not increase its complexity. The primitives are very flexible and can be used to build more complicated dynamic operations, such as contraction along an edge. We later use these primitives in our main result.

In general, we maintain a collection of subdivisions and their duals. Each subdivision is thought of as lying in a distinct plane. The *make edge* primitive, which takes no parameter, creates two new vertices connected by a new single edge **e**. The edge and its endpoints form a new subdivision that is embedded along with its dual in a new plane. The *make edge* primitive returns the directed edge e_0. The inverse operation, *destroy edge*(e), takes as an argument an edge that is guaranteed to be disconnected. The edge is destroyed and the storage is released.

The second primitive is $splice(d, e)$, where d and e are directed edges of the primal subdivision. Splice operates on the vertices $orig(d)$ and $orig(e)$, and independently on the dual vertices corresponding to the left faces of d and e, which are given by $orig(rot^{-1}(d))$ and $orig(rot^{-1}(e))$. If the edges originate in the same vertex, then the splice operation splits that vertex in two, with the edges clockwise from d to e going to one of the halves, while the remaining edges go to the other. If the edges have different origins, then the two vertices are combined into one by inserting the edge ring of one vertex into the edge ring of the other. Figure 2 gives an example. Let $\delta = rot(next(d))$ and $\epsilon = rot(next(e))$. The splice simply exchanges the values of $next(d)$ and $next(e)$, while simultaneously exchanging the values of $next(\delta)$ and $next(\epsilon)$.

The values given by the $next$, $orig$, and rot operators determine incidence relations between the faces, edges, and vertices of S. In turn, these incidence relations determine the topology of the surface that S subdivides. Since $splice(d, e)$ changes the values of $next(d)$ and $next(e)$, the choice of d and e is restricted by the requirement that the result of the splice remain a subdivision of the plane. Any splice is allowed in which d and e have the same origin or left face, because the splitting of a vertex in either the primal or dual preserves planarity, and if one subdivision remains planar then its dual must also remain planar. If both the origins and the left faces differ, however,



(a) (b)

Figure 2: a) Example of edge rings. Primal vertices u and v lie on the boundary of face f. b) Edge rings and topology produced by executing $splice(a, b)$ (or equivalently, $splice(x, y)$) on edge rings of (a).

and the two edges are contained in the same subdivision, then the splice is disallowed. Such a splice increases by one the genus of the surface that S subdivides. On the other hand, if the edges lie in different subdivisions, i.e. different planes, the splice is allowed. In this case, the splice merges the two subdivisions so that they are contained in a single surface. Given S, it is always possible to draw a subdivision that is topologically equivalent to S but in which some specified edge or vertex is adjacent to the exterior face. Thus the splice of edges contained in different subdivisions can be thought of as redrawing the subdivisions to place the edges on the exteriors, and then plugging the subdivisions together at the origins of these edges. The validity of a *splice* or *destroy edge* operation can be tested using the data structure we present in the next section.

Let S be a subdivision containing m edges. Any undirected edge $\mathbf{e}$ can be deleted from S by taking one of its directed versions e and executing $splice(e, next^{-1}(e))$ and $splice(sym(e), next^{-1}(sym(e)))$, followed by *destroy edge*(e). Thus a sequence of $O(m)$ *splices* and *destroy edges* reduces S to the null subdivision. Since splice is reversible (in fact, splice is its own inverse), we may conclude that the operations *make edge* and *splice* are sufficient to generate any planar subdivision not consisting of a single isolated vertex. Furthermore, we see how to use *make edge* and *splice* to implement more complicated dynamic operations. For example, the operation *insert edge*(d, e), which inserts an edge between $orig(d)$ and $orig(e)$, dividing the face to the left of d and e, can be implemented by $x =$*make edge* followed by $splice(d, x)$ and $splice(e, sym(x))$. We can similarly implement other standard operations such as *delete edge*, *expand*, and *contract* (see [29]).

Let $\mathbf{G}$ denote the planar multigraph induced by the vertices and edges of a collection of subdivisions. Each subdivision induces a connected component of $\mathbf{G}$. We may use *make edge* and *splice* to generate any multigraph $\mathbf{G}$ not containing isolated vertices. (New vertices are always created by *make edge* in pairs, connected by the new edge. If one wishes to allow isolated vertices, they can very easily be handled.)

2.2 Changing Edge Weights Only

For the sake of completeness we first consider the restricted problem in which the topology of S is fixed and the only modification permitted is *change weight*$(\mathbf{e}, \Delta)$. The approach used and the result obtained are due to Gabow and Stallmann [11, Corollary 3.1]. We present them here using our notation and in a more detailed fashion; this will help in the description of our own results.

The algorithm uses both S and its dual S^*. For each dual edge we define $w(\mathbf{e}^*) = w(\mathbf{e})$. The following lemma (theorem XI.6 of [33]) is the basis for

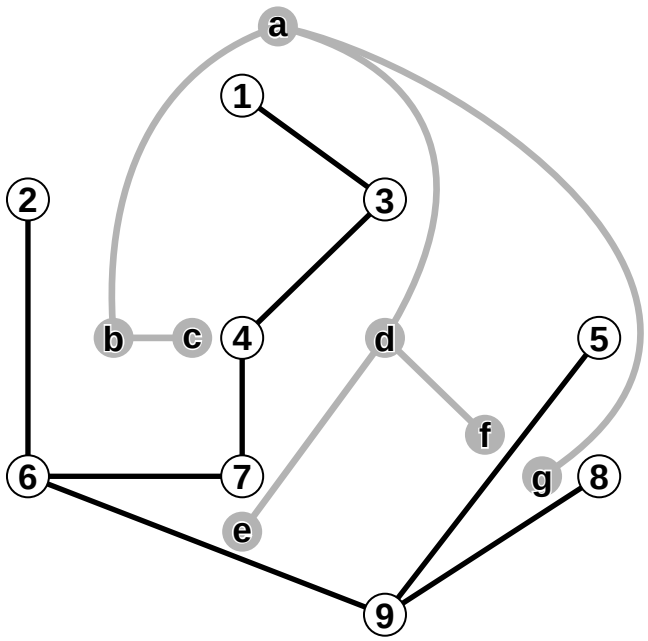


Figure 3: Primal and dual spanning trees for the subdivision of Figure 1

the algorithm.

Lemma 1 [33, pp. 289] *Given a spanning tree T for S, let T^* be the set of dual edges $\{\mathbf{e}^* \mid \mathbf{e}$ is not in $T\}$. Then T^* is a spanning tree for S^*.*

If $w(T)$ is the sum of the weights of the edges in T, and W is the sum of the weights of all edges in S, we have that $W = w(T) + w(T^*)$. Thus T is a minimum spanning tree for S if and only if T^* is a maximum spanning tree for S^*.

Figure 3 gives an example of primal and dual spanning trees for the subdivision of Figure 1.

The algorithm maintains T and T^* in tandem. Lemma 1 implies that after a change in edge weight, correct updating of the primal spanning tree automatically results in correct updating of the dual, and vice versa. To perform the updates efficiently, the dynamic tree data structure of Sleator and Tarjan [26, 27] is used. Dynamic trees are designed to represent a forest of rooted trees, each node of which has a real-valued cost, under the following operations:

make node: Make a new tree node with no incident edges and an initial cost of $-\infty$.

find cost(v): Return the cost of node v.

find root(v): Return the root of the tree containing node v.

find min(v) (*find max*(v)): Return the node of minimum (maximum) cost on the path from v to r, the root of the tree containing v.

add cost(v, Δ): Add real number Δ to the cost of all nodes on the path from v to r, the root of the tree containing v.

link(v, w): Add an edge from v to w, thereby making v a child of w in the forest. This operation assumes that v is the root of one tree and w is in another.

cut(v): Delete the edge from v to its parent, thereby dividing the tree containing v into two trees.

evert(v): Make v the root of its tree by reversing the path from v to the original root.

find parent(v): Return the parent of v, or null if v is the root of its tree.

find lca(u, v): Return the least common ancestor of nodes u and v.

All the above operations can be performed in $O(\log n)$ amortized time per operation and $O(n)$ space, where n is the number of nodes in the tree or trees to which the operation applies.

The vertices of S are represented by dynamic tree nodes with cost $-\infty$. Similarly, the vertices of S^* are represented by dynamic tree nodes with cost $+\infty$. In T the operation *find max* is used, while in T^* the operation *find min* is used. The roots of T and T^* are chosen arbitrarily. For every edge $\mathbf{e}$ there is a dynamic tree node $\hat{e}$ of cost $w(\mathbf{e})$. If $\mathbf{e}$ is a spanning edge of T then there is an edge between the tree node representing the vertex $orig(e_0)$ and $\hat{e}$, and an edge between $\hat{e}$ and the tree node for $orig(e_2)$. Conversely, if $\mathbf{e}^*$ is a spanning edge of T^*, then tree edges join $orig(e_1)$ to $\hat{e}$, and $\hat{e}$ to $orig(e_3)$. Thus $\mathbf{e}$ is represented by two edges connected through the degree-two node $\hat{e}$. This representation allows *find max* and *find min* on T and T^* respectively to return edges rather than vertices.

For each edge $\mathbf{e}$, the five values of $\hat{e}$ and $orig(e_i), 0 \leq i \leq 3$, are stored in the form of pointers to the corresponding dynamic tree nodes. If the subdivision has $O(m)$ edges the number of vertices and faces is also $O(m)$, and so the total space required for the trees is $O(m)$.

To process *change weight*($\mathbf{e}, \Delta$), we first update the edge weight by executing *evert*($\hat{e}$) and *add cost*($\hat{e}, \Delta$). Four cases can occur:

1. **e** is in T and Δ is negative.

2. **e** is not in T (**e*** is in T^*) and Δ is positive.

3. **e** is not in T and Δ is negative.

4. **e** is in T and Δ is positive.

Clearly, Cases 1 and 2 have no effect on the spanning trees. Now consider Case 3. It is well-known (e.g. see [10, 30]) that in this case T is no longer minimum if the weight of **e** is less than the weight of the maximum-cost edge **d** in the cycle formed by adding **e** to T. Edge **d** is found by executing $evert(orig(e_0))$ followed by *find max*$(orig(e_2))$. In the special case where the *find max* operation returns $-\infty$, processing terminates immediately. This case occurs when $orig(e_0) = orig(e_2)$; that is, **e** is a loop edge that can never be a spanning edge, while the dual edge **e*** is a bridge of G^* that must always be a spanning edge.

In any case, if $w(\mathbf{e}) \geq w(\mathbf{d})$, no action need be taken. If not, however, then the new minimum spanning tree T is given by deleting edge **d** and inserting edge **e**. Simultaneously, the new maximum spanning tree T^* is given by deleting edge **e*** and inserting edge **d***. This is done by executing the following operations:

$evert(orig(d_0))$; $cut(\hat{d})$; $cut(orig(d_2))$;
$evert(orig(e_1))$; $cut(\hat{e})$; $cut(orig(e_3))$;

followed by:

$evert(orig(e_0))$; $link(\hat{e}, orig(e_0))$; $link(orig(e_2), \hat{e})$;
$evert(orig(d_1))$; $link(\hat{d}, orig(d_1))$; $link(orig(d_3), \hat{d})$;

Since only a constant number of links, cuts and everts are required, the amortized time for the *change weight* operation is $O(\log m)$.

Now consider Case 4. Let (V_1, V_2) be a partition of the vertices of G. The cut induced by (V_1, V_2) is the set of edges of G with one endpoint in V_1 and the other in V_2. Again, it is well-known that, in Case 4, T is no longer minimum if the weight of **e** is greater than the weight of the minimum-cost edge in the cut induced by the partition (V_1, V_2), where V_1 and V_2 are the vertex sets of the connected components of T created by the removal of edge **e**. Given only the primal tree, this cut edge is hard to find. The utility of the dual spanning tree becomes clear, however, when it is observed that Case 4 is the equivalent in the dual tree of Case 3 in the primal tree. A dual edge not in T^* has increased in cost, and may therefore force a dual edge out of T^*. The same processing as in Case 3 can be applied, interchanging the role

of dual and primal tree, and using *find min* rather than *find max*. Thus Case 4 can also be handled in amortized time $O(\log m)$.

Thus, the result from [11] is that when given S, a subdivision of the plane undergoing on-line changes in edge weight, a minimum spanning tree of S can be maintained in $O(\log m)$ amortized time per operation and $O(m)$ space, where m is the number of edges.

The above time bound can be made worst-case with the biased tree implementation of the dynamic tree data structure [26].

Let G be a plane graph of n vertices (and hence $O(n)$ edges) undergoing changes in edge weight. Note that the planar embedding can be generated in $O(n)$ time using one of the algorithms of Hopcroft and Tarjan [16] or Booth and Lueker [4] (see Chiba, Nishizeki, Abe, and Ozawa [6]). Each connected component gives rise to a planar subdivision. The initial spanning trees can be found in $O(n)$ time with the algorithm of Cheriton and Tarjan [5]. Thus, given $O(n)$ preprocessing time, one can maintain the minimum spanning forest of G in $O(\log n)$ amortized time per operation and $O(n)$ space.

3 Edge-ordered Trees and a Fully Dynamic Algorithm

In this section we present our main result, the fully dynamic algorithm. We first develop the *edge-ordered dynamic tree*, a data structure designed to handle splices and the resultant cutting and linking of edge rings efficiently. An edge-ordered tree is a general rooted tree in which a total order is imposed on the edges adjacent to each given node (including the parent edge). The ordered set of edges adjacent to node v is called the *edge list* for v. For example, in our application we will use the counterclockwise ordering of the edges around the vertex in the current graph embedding, with an arbitrary edge first. Each node v in the tree has a real-valued cost, $cost(v)$. The edge-ordered tree supports the following collection of operations (we use capitals to distinguish them from the corresponding dynamic tree operations):

$Link(v, w)$: Add an edge e from v to w, thereby making v a child of w in the forest (v is assumed to be a root). The new edge is inserted at the end of the edge list of v and at the front of the edge list of w. Return e.

$Split(v, e)$: Split node v into two nodes v', v''. If $\alpha e \beta$ is the ordered list of edges adjacent to v then αe becomes the ordered list of edges adjacent to v', while β becomes the ordered list adjacent to v''. Nodes v' and v'' have the same cost as v.

$Merge(u, v)$: Merge nodes u and v into a single node w. If α is the ordered

list of edges for u and β is the ordered list of edges for v then $\alpha\beta$ is the ordered list of edges for w. Nodes u and v must have the same initial cost. Return w.

Cycle(v, e): Cyclically permute the order of edges adjacent to v so that e is the last edge in the order. The initial ordered list $\alpha e \beta$ becomes $\beta \alpha e$.

Add cost(v, x): Add real value x to *cost*(v). Note that this differs from the definition of *add cost* in [26, 27], since only node v is affected by the operation.

The edge-ordered tree data structure also supports *Evert*(v), *Cut*(v), *Find cost*(v), *Find root*(v), *Find min*(v) (*Find max*(v)), *Find parent*(v), and *Find lca*(u, v). These operations have the same definitions as the analogous (lower-case) operations that were defined in Subsection 2.2.

To implement the edge-ordered tree we do not create a completely new data structure; rather, we show how to transform any given tree T into a new tree T'. Each node v of T is expanded into a collection of subnodes called a *node path*. Each subnode s has a cost that is always set equal to *cost*(v). There is one subnode in the node path v for every edge e in the edge list of v. The subnode for e is connected by tree edges to the subnodes of its predecessor and successor in the edge list. The subnodes for the first and last edges in the list are connected only to their successor and predecessor respectively. For each vertex v there is an auxiliary block of storage that contains pointers to the first and last subnodes, denoted v_{first} and v_{last}. We assume the existence of routines *Make node* and *Destroy node*(v) that create and destroy this auxiliary storage. A node is referenced by a pointer to this storage block. Whenever an edge e connects nodes u and v in T, there is an edge in T' between the two subnodes s_u and s_v generated by e in the node paths of u and v. Edge e is referenced by one of its endpoints $\{s_u, s_v\}$ as appropriate. Thus, to split node v at edge e, we execute *Split*(v, s_v).

If T has n nodes and hence $n - 1$ edges, then T' has $2n - 2$ nodes. Note that every node in T' has degree at most three. A similar idea has been used by Goldberg, Grigoriadis and Tarjan [12] in a different extension of dynamic trees that supports computing minima and maxima over subtrees. (Our extension requires some additional ideas.) Figure 4 gives an example of an edge-ordered tree.

The transformed tree T' is maintained with a standard Sleator-Tarjan dynamic tree. The node path for node v has the property that if *evert*(v_{last}) is performed, then the ordered sequence of nodes on the tree path between v_{first} and v_{last} corresponds exactly to the ordered sequence of edges in the edge list from first to last. This property allows the processing of all the edge-ordered tree operations with only a constant number of dynamic tree operations. If

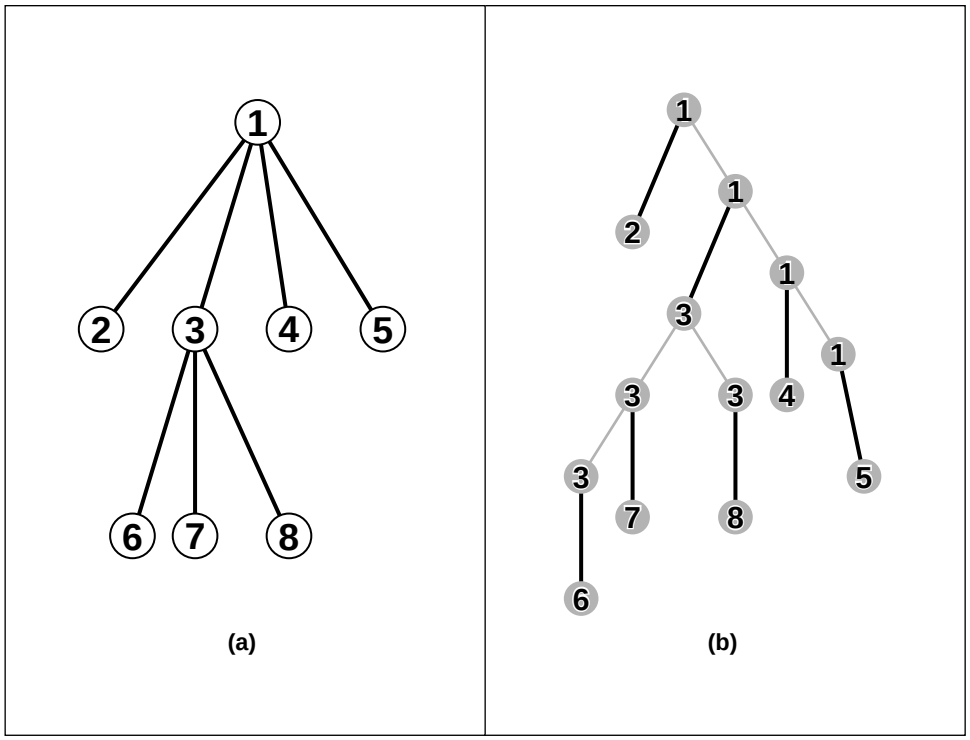


Figure 4: (a) Edge-ordered tree – the actual tree. Edge list for vertex 1 is: (1,2),(1,3),(1,4),(1,5). Edge list for vertex 3 is: (3,6),(3,7),(3,1),(3,8). (b) Tree of (a) transformed into node path representation. Dark edges correspond to true tree edges.

we only need to perform the operations *Link* through *Find cost*, the dynamic tree suffices. To perform *Cut*, the node paths must also be threaded into a doubly-linked list, and to perform *Find min*, *Find parent*, *Find lca*, and *Find root* auxiliary balanced trees are required. We begin by giving implementations of the edge-ordered tree operations in the first group. For convenience, we will use the notation e to represent both an edge and the appropriate corresponding tree subnode.

$Link(u, v)$ **begin**
    $x :=$*make node*; $y :=$*make node*;
    $evert(u_{last})$;
    $link(u_{last}, x)$; $link(x, y)$; $link(y, v_{first})$;
    $u_{last} := x$; $v_{first} := y$;
    return x,y;
end

$Split(u, e)$ **begin**
    $v :=$*Make node*; $w :=$*Make node*;
    $evert(u_{last})$;
    if *find* $lca(u_{first}, e) \neq e$ **then error** (e not in node path of v);
    $y := findparent(e)$;
    $cut(e)$;
    $v_{first} := u_{first}$; $v_{last} := e$;
    $w_{first} := y$; $w_{last} := u_{last}$;
    Destroy node(u);
    return v, w;
end

$Merge(u, v)$ **begin**
    $w :=$*Make node*;
    $evert(u_{last})$;
    $link(u_{last}, v_{first})$;
    $w_{first} := u_{first}$; $w_{last} := v_{last}$;
    Destroy node(u); *Destroy node*(v);
    return w;
end

```
Cycle(v, e) begin
        evert(v_last);
        if find lca(u_first, e) ≠ e then error (e not in node path of v);
        if e = v_last then return;
        x := find parent(e);
        cut(e);
        link(v_last, v_first);
        v_first := x; v_last := e;
end

Add cost(v, Δ) begin
        evert(v_last);
        add cost(v_first, Δ);
end
```

The operations *Evert*(v) and *Find cost*(v) are simply implemented by $evert(v_{last})$ and $find\ cost(v_{last})$, respectively. If the tree is to be rooted at node r, then those operations whose implementation uses an evert must be followed by a final $evert(r_{last})$.

If the operation $Cut(v)$ is needed, we thread each node path into a circular doubly-linked list. We denote the predecessor and successor of subnode s by $pred(s)$ and $succ(s)$.

```
Cut(v) begin
        x := find lca(v_first, v_last);
        y := find parent(x);
        cut(x);
        for s in {x, y} do begin
            evert(succ(s)); cut(s);
            cut(pred(s)); link(pred(s), succ(s));
            succ(pred(s)) := succ(s); pred(succ(s)) := pred(s);
        end
end
```

Note that in order to maintain the node path linked lists, each *link* or *cut* that occurs in the implementation of the first group of operations must be followed by the appropriate operation on the linked list. After the edge is cut, the storage used by the two subnodes x, y, which are no longer needed, is reclaimed.

To include *Find min*(v), *Find parent*(v), *Find lca*(u, v), and *Find root*(v) in the repertoire of edge-ordered tree operations, we need the operation *Find node*(s), which given subnode s returns the node v whose node path contains s. By maintaining each node path in an auxiliary balanced binary

tree such as a red-black tree or splay tree (see [31, pp. 45–53]), *Find node*(s) can be performed in $O(\log n)$ time, either worst-case or amortized, depending on the choice of data structure. Again, appropriate insertions, deletions, splits and concatenations must be done in the auxiliary data structure when operations such as link or cut occur in the implementation of the first group of tree operations. The balanced trees mentioned above support insertions, deletions, splits, and concatenations in $O(\log n)$ time.

Using *Find node*, we implement the remaining operations as follows:

Find min(v) **begin**
    return *Find node*(*find min*(v_{first})); **end**

Find parent(v) **begin**
    return *Find node*(*find parent*(*find lca*(v_{first}, v_{last}))); **end**

Find lca(u, v) **begin**
    return *Find node*(*find lca*(u_{first}, v_{first})); **end**

Find root(v) **begin**
    return *Find node*(*find root*(v_{last})); **end**

Since each edge-ordered tree operation is implemented using a constant number of dynamic tree operations, the overall amortized running time per operation remains $O(\log n)$.

We now discuss the application of edge-ordered trees to the minimum spanning tree maintenance problem. Let $\mathbf{G}$ denote the multigraph induced by the vertices and edges of a collection of subdivisions, and let $\mathbf{G}^*$ denote the multigraph given by their duals. As in Subsection 2.2, the vertices of $\mathbf{G}$ are represented by tree nodes of cost $-\infty$ and the vertices of $\mathbf{G}^*$ by nodes of cost $+\infty$.

We wish to ensure that each directed edge e is represented in the edge list of the node $v = orig(e)$. To do this, we create a dummy node $\hat{e}$ of cost $w(\mathbf{e})$, and make it a child of v. With e we store the pair of subnodes that represent e in the node paths of v and $\hat{e}$. This allows the use of *Find node* to determine $orig(e)$ and $\hat{e}$. The counterclockwise order of directed edges around v determines the linear order in the edge list of v; the first edge in the linear order is chosen arbitrarily.

If $\mathbf{e}$ is a spanning edge of $\mathbf{G}$ then the dummy nodes for e_0 and e_2 are merged to give a degree-two node representing $\mathbf{e}$ that connects nodes $u = orig(e_0)$ and $v = orig(e_2)$. Similarly, if $\mathbf{e}^*$ is a spanning edge of $\mathbf{G}^*$, then

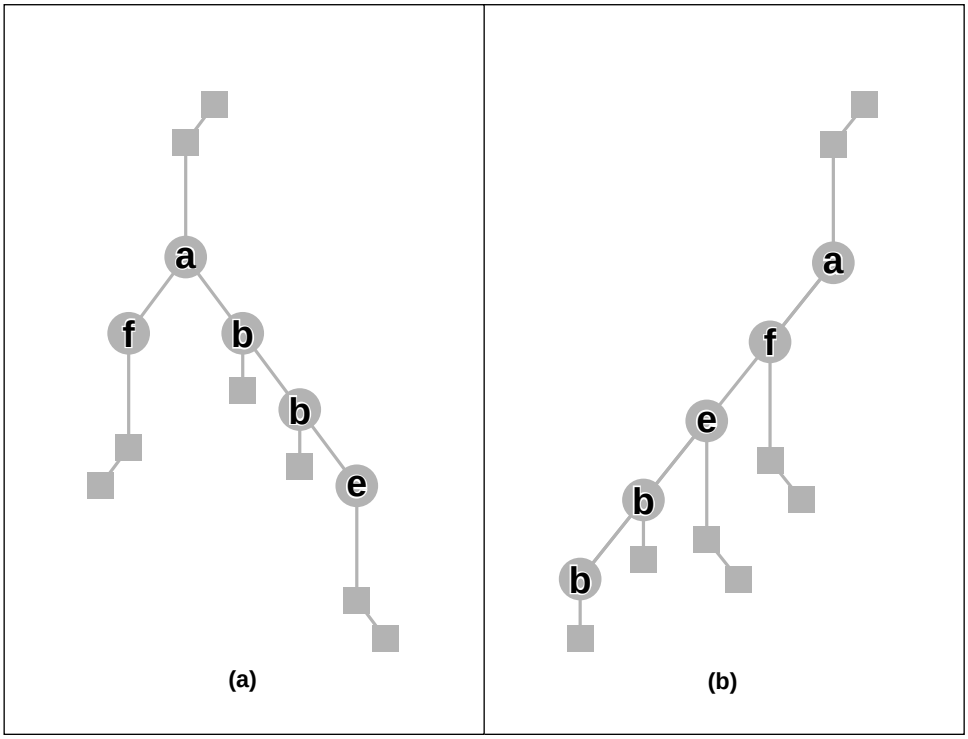


Figure 5: a) Node path for vertex d of the spanning tree of Figure 2. Each subnode is labeled by the vertex to which it is adjacent. Unlabeled squares are $\hat{e}_i$ nodes. b) Node path for d after executing $Cycle(d, e)$, where $e = \{d, a\}$.

the dummy nodes for e_1 and e_3 are merged. There are $O(m)$ tree nodes, so the total space required is $O(m)$. Note that each loop edge gives rise to two sibling dummy nodes, one for each directed version of the loop. Figure 5 gives an example of a node path.

The algorithm given in Subsection 2.2 for *change weight* operations can be adapted for use with edge-ordered trees. If non-spanning primal edge **e** decreases in weight, we find the edge **d** of maximum weight on the path connecting the endpoints of **e** by executing $Evert(orig(e_0))$ and $Find\ max(orig(e_2))$. Edge **d** is represented in T by a degree-two node u with incident edges corresponding to d_0 and d_2. To replace edge **d** by edge **e** in the primal spanning tree, we perform $Split(u, u_{first})$ followed by $Merge(\hat{e}_0, \hat{e}_2)$. Similarly, we split the node w representing $\mathbf{e}^*$ in T^*, then merge $\hat{d}_1$ and $\hat{d}_3$.

A *make edge* request creates two new vertices in the primal graph, connected by a new edge **e** with $w(\mathbf{e}) = -\infty$. Simultaneously, the dual graph is augmented by a single vertex with the incident loop edge $\mathbf{e}^*$. The primal

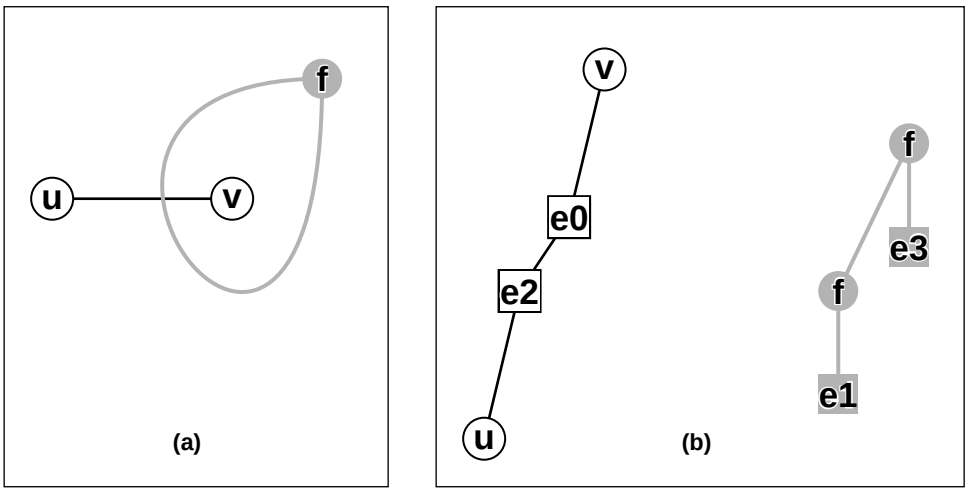


Figure 6: a) Primal (black) and dual (grey) subdivisions produced by e =*make edge*. b) Primal and dual edge-ordered trees for subdivisions of (a).

edge **e** is automatically a spanning edge of **G**. To satisfy the request, the algorithm allocates storage for a new primal/dual spanning tree pair. The primal tree T consists of two singleton node paths connected through a node that is the merge of $\hat{e}_0$ and $\hat{e}_2$. The dual tree T^* consists of a node path containing two subnodes, with children $\hat{e}_1$ and $\hat{e}_3$. (See Figure 6.)

A $splice(d, e)$ operation has more complicated behavior. The most complex situation occurs when directed edges d and e have distinct origins but the same left face (or symmetrically, the same origin but distinct left faces.) Let δ and ϵ be the dual directed edges given by $rot(next(d))$ and $rot(next(e))$ respectively. Combining the vertices $u = orig(d)$ and $v = orig(e)$ into a single vertex will create a cycle in the primal spanning tree. This cycle is broken by removing the edge **x** of maximum weight on the cycle. The algorithm for processing a *change weight* request can be used to find **x**. Splitting of the face $f = orig(\delta) = orig(\epsilon)$ breaks T^* into two fragments. They are then joined together by linking in the edge $\mathbf{x}^*$. Thus the tree modifications caused by the splice are equivalent to those occurring if initially the two vertices had been joined by an edge that changed weight from $+\infty$ to $-\infty$. The specific processing is as follows:

1. As discussed above, find **x** and perform $Split(\mathbf{x}, x_{first})$. This breaks T into two fragments.

2. Reconnect the two fragments of T with $Cycle(u, d)$, $Cycle(v, e)$, and $Merge(u, v)$.

3. Perform $Cycle(f, \delta)$ and $Split(f, \epsilon)$.

4. Reconnect the two fragments of T^* with $Merge(\hat{x}_1, \hat{x}_3)$.

The processing for the other cases of $splice(d, e)$ is simpler. If both edges have the same origin v and left face f, then v is an articulation point of $\mathbf{G}$. The splice breaks one subdivision into two subdivisions of distinct surfaces and correspondingly breaks one component of $\mathbf{G}$ into two components. The two fragments into which T is broken by the splice remain valid minimum spanning trees for the new components, since T previously spanned the entire graph, and the fragments were connected only through v. Therefore we need only execute the cycle and split of Step 3 above, once on the primal edges d, e and vertex v, and once on the dual edges δ, ϵ and face f.

Similarly, if the edges belong to different components, and hence different subdivisions of distinct surfaces, then the splice operation joins the components through a new articulation vertex w given by the merge of $u = orig(d)$ and $v = orig(e)$. The two dual components are simultaneously joined through a vertex h given by the merge of $f = orig(\delta)$ and $g = orig(\epsilon)$. By assumption, the two initial components are correctly spanned, so by combining the two vertices a valid minimum spanning tree for the unified graph is created. Therefore, in this case we need only execute the cycles and merge of Step 2, once on u, v, d and e, and once on f, g, δ and ϵ.

The *make edge* operation requires constant time, while each splice performs a constant number of edge-ordered tree operations, each of which requires $O(\log m)$ amortized time per operation, where m is the number of edges in the subdivision.

Theorem 1 *The minimum spanning tree of a planar subdivision undergoing both changes in edge weight and changes to its structure can be maintained in $O(\log m)$ amortized time per operation and $O(m)$ space.*

Again, the time bound can be made worst-case by using the biased-tree implementation of dynamic trees [26].

We note that, given a minimum spanning tree, we can answer connectivity queries, such as $find(u, v)$, which asks if vertices u and v are in the same component of $\mathbf{G}$, by taking representative subnodes in the vertex paths for u and v and finding the roots of the spanning trees containing them. (This query can be used to check the validity of splice operations.)

The data structure we have presented encodes the entire structure of the subdivisions. The entire range of dynamic tree operations described above and in references [26, 27] is available for use with the spanning trees, making the overall data structure quite powerful and flexible.

4 A Lower Bound

Let A be an algorithm for maintaining a minimum spanning tree of an arbitrary edge-weighted (multi)graph G. Let A be such that the operation *change weight*(e, Δ) returns the edge f that replaces e in the minimum spanning tree, if e is replaced. Clearly, any dynamic minimum spanning tree algorithm A' can be modified to return f (furthermore, all known algorithms compute f as part of the *change weight* routine). One can use algorithm A to sort n positive numbers $x_1, x_2, \ldots, x_n$. Simply construct a multigraph G consisting of two nodes connected by $n+1$ edges $e_0, \ldots, e_n$, such that edge e_0 has weight 0 and edge e_i has weight x_i. The initial spanning tree is e_0. Increase the weight of e_0 to $+\infty$ (i.e., to something larger than the sum of the x_i's). Whichever edge replaces e_0, say e_i, is the edge of minimum weight. Now increase the weight of e_i to $+\infty$; the replacement edge of e_i is the edge of second smallest weight. Continuing in this fashion gives the numbers in sorted order.

Similarly, suppose only decreases in costs are allowed. Form a cycle of $n+1$ edges, $e_1, \ldots, e_n$ of weight $x_1, \ldots, x_n$, respectively, and e_0 with weight $+\infty$. The initial spanning tree is the edges $e_1, \ldots, e_n$. Decrease the weight of e_0 to $-\infty$. Whichever edge leaves the tree is the edge of largest weight. Repeatedly lowering the weight of the remaining edge of largest weight gives the numbers sorted in reverse order.

Paul and Simon [22] have shown that any unit-cost random access sorting algorithm whose operations include addition, subtraction, multiplication, and comparison with 0, but not division or bit-wise Boolean operations, takes $\Omega(n \log n)$ worst-case time to sort n numbers. The currently known algorithms for maintenance of a minimum spanning tree fit into this model. Thus the $O(\log n)$ amortized time per operation of our algorithm can only be improved by taking advantage of a more powerful computational model, or by avoiding the computation of the replacement edge f following each edge cost update (that is, avoiding an explicit representation of the current tree).

Since the reduction to sorting uses only edge weight changes, the lower bound is also applicable to algorithms for the static subdivision version of the problem; hence the Gabow-Stallmann result is also optimal within the above machine model.

5 Discussion

In implementing edge-ordered tree operations we used balanced trees as auxiliary data structures to maintain the node paths while performing splits and merges. These auxiliary data structures are used primarily to answer *find node* queries in logarithmic time. In fact, Sleator-Tarjan dynamic trees

may also be used as the auxiliary data structures, with each edge list maintained as a linear branch always rooted at the head node. This suggests that it may be possible to combine the auxiliary functions into the primary dynamic tree and eliminate the auxiliary data structures entirely. We are currently unable to do so, however.

In our approach to the dynamic spanning tree problem, modification operations are specified by edges. Tamassia [29] gives a data structure for maintaining a dynamic embedding of a biconnected planar graph that can test in $O(\log n)$ time whether two vertices u and v lie on a common face. With this auxiliary data structure we can allow some modifications to be specified in terms of vertices. For example, we can support *insert edge*(u, v), which inserts an edge between vertices u and v if they lie on a common face, by using Tamassia's data structure to find the two edges that are adjacent to a common face and have as origins u and v respectively. These edges can then be used as input to *splice*.

Our planar subdivision algorithms can be used to maintain planar graphs, but the modifications permitted are limited by the embedding and thus fit applications where the embedding is given. Even if one planar graph G_1 can be derived from another G_2 by a single edge addition, a large number of modifications to the subdivision that embeds G_1 may be required to build a subdivision that embeds G_2. From a theoretical point of view, however, it would be more satisfying to have an algorithm that allowed the following operations: insert a new vertex; delete a disconnected vertex; delete an edge; and insert an edge if the resultant graph remains planar. If such an algorithm were based on the primal/dual spanning tree relationship, however, then it would need to move quickly (i.e., in $O(\log n)$ amortized time) between topologically distinct embeddings. In recent work Di Battista and Tamassia [2, 3] give data structures and algorithms that can do this in $O(\log n)$ time in the restricted case that only edge insertions are allowed. If a modification primitive powerful enough to allow edge deletions is allowed, however, the problem becomes significantly more difficult, and currently no solution better than repeated application of a static planarity-testing algorithm is known.

References

[1] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 12–21, 1990.

[2] G. D. Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 436–441, 1989.

[3] G. D. Battista and R. Tamassia. On-line planarity testing. Technical Report CS-89-31, Department of Computer Science, Brown University, 1989.

[4] 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.

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

[6] 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.

[7] F. Chin and D. Houck. Algorithms for updating minimum spanning trees. *J. Comput. System Sci.*, 16:333–344, 1978.

[8] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

[9] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. ACM*, 28:1–4, 1981.

[10] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14:781–798, 1985.

[11] H. N. Gabow and M. Stallmann. Efficient algorithms for graphic matroid intersection and parity (extended abstract). In *Automata, Languages, and Programming, 12^{th} Colloquium, Lecture Notes in Computer Science, vol. 194*, pages 210–220. Springer-Verlag, Berlin, 1985.

[12] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Prog.*, to appear.

[13] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Trans. on Graphics*, 4:74–123, 1985.

[14] F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA., 1972.

[15] D. Harel. On-line maintenance of the connected components of dynamic graphs. Unpublished manuscript, 1982.

[16] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[17] T. Ibaraki and N. Katoh. On-line computation of transitive closure for graphs. *Inf. Process. Lett.*, 16:95–97, 1983.

[18] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoret. Comput. Sci.*, 48:273–281, 1986.

[19] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Inf. Process. Lett.*, 28:5–11, 1988.

[20] G. F. Italiano, A. M. Spaccamela, and U. Nanni. Dynamic data structures for series parallel digraphs. In *Proc. Workshop on Algorithms and Data Structures, (WADS 89), Lecture Notes in Computer Science, vol. 382*, pages 352–372. Springer-Verlag, Berlin, 1989.

[21] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science, (WG 87), Lecture Notes in Computer Science, vol. 314*, pages 106–120. Springer-Verlag, Berlin, 1988.

[22] J. Paul and W. Simon. Decision trees and random access machines. In *Symposium uber Logik und Algolrithmik*, 1980. Also in K. Mehlhorn, *Sorting and Searching*, pages 85–97, Springer-Verlag, Berlin, 1984.

[23] F. P. Preparata and R. Tamassia. Fully dynamic techniques for point location and transitive closure in planar structures. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 558–567, 1988.

[24] J. H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25:65–70, 1987.

[25] H. Rohnert. A dynamization of the all pairs least cost path problem. In *Proc. 2nd Annual Symp. on Theoretical Aspects of Computer Science, (STACS 85), Lecture Notes in Computer Science, vol. 182*, pages 279–286. Springer-Verlag, Berlin, 1985.

[26] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[27] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32:652–686, 1985.

[28] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Comput.*, 4:375–380, 1975.

[29] R. Tamassia. A dynamic data structure for planar graph embedding. In *Proc. 15th Int. Conf. on Automata, Languages, and Programming, (ICALP 1988), Lecture Notes in Computer Science, vol. 317*, pages 576–590. Springer-Verlag, Berlin, 1988.

[30] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Inf. Process. Lett.*, 14:30–33, 1982.

[31] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA., 1983.

[32] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6:306–318, 1985.

[33] W. T. Tutte. *Graph Theory.* Addison-Wesley, Menlo Park, CA., 1984.

[34] D. Yellin. A dynamic transitive closure algorithm. Technical report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, 1988.