SIAM J. COMPUT. © 1997 Society for Industrial and Applied Mathematics
Vol. 26, No. 2, pp. 484–538, April 1997 009

AMBIVALENT DATA STRUCTURES FOR DYNAMIC 2-EDGE-CONNECTIVITY AND k SMALLEST SPANNING TREES*

GREG N. FREDERICKSON†

Abstract. Ambivalent data structures are presented for several problems on undirected graphs. These data structures are used in finding the k smallest spanning trees of a weighted undirected graph in $O(m \log \beta(m, n) + \min\{k^{3/2}, km^{1/2}\})$ time, where m is the number of edges and n the number of vertices in the graph. The techniques are extended to find the k smallest spanning trees in an embedded planar graph in $O(n + k(\log n)^3)$ time. Ambivalent data structures are also used to dynamically maintain 2-edge-connectivity information. Edges and vertices can be inserted or deleted in $O(m^{1/2})$ time, and a query as to whether two vertices are in the same 2-edge-connected component can be answered in $O(\log n)$ time, where m and n are understood to be the current number of edges and vertices, respectively.

Key words. analysis of algorithms, data structures, embedded planar graph, fully persistent data structures, k smallest spanning trees, minimum spanning tree, on-line updating, topology tree, 2-edge-connectivity

AMS subject classification. 68Q

PII. S0097539792226825

1. Introduction.

Efficient handling of on-line requests requires that data be stored flexibly. At each location in a data structure, it can be advantageous to keep track of a small number of alternatives, only one of which can in fact be valid. An example of such alternatives might be whether a path between vertices x and y in a spanning tree of a graph goes through a vertex w or through a vertex w'. We say that a data structure possesses *ambivalence* if at each of many locations in the structure it keeps track of several alternatives, even when a global examination of the data structure would identify for each location the alternative (or valence) that is in fact valid. (A more formal definition of this property is given at the beginning of section 7.) The structure necessarily organizes the data in such a way that the correct alternative is known for some crucial case. We apply this technique in the design of data structures for several graph problems related to connectivity. Our data structures are ambivalent with regard to the structure of a spanning tree as that spanning tree is being updated, and they yield algorithms faster than any previously known.

Our first problem is that of finding the k smallest spanning trees of a weighted undirected graph. Using data structures that are both ambivalent and fully persistent [DSST], we give an algorithm that uses $O(m \log \beta(m, n) + \min\{k^{3/2}, km^{1/2}\})$ time, where m is the number of edges and n is the number of vertices. Here $\beta(\cdot, \cdot)$ is a very slowly growing function, as defined by Fredman and Tarjan [FT], and the first term in our running time represents the best known time to find a minimum spanning tree [GGST]. Where appropriate, we shall substitute this time when quoting previous results. The amount of space used by our algorithm is $O(m + \min\{k^{3/2}, km^{1/2}\})$. For

* Received by the editors February 11, 1992; accepted for publication (in revised form) June 1, 1995. A preliminary version of this paper appeared in *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 632–641. This research was supported in part by National Science Foundation grants CCR-9001241 and CCR-9322501 and Office of Naval Research contract N00014-86-K-0689.

http://www.siam.org/journals/sicomp/26-2/22682.html

† Department of Computer Sciences, Purdue University, West Lafayette, IN 47907 (gnf@cs.purdue.edu).

the case of a planar graph, we give fully persistent data structures that are then used in an algorithm that takes $O(n + k(\log n)^3)$ time and $O(n + k(\log n)^2)$ space.

Our results compare with previous results on this problem as follows. The problem of enumerating the k smallest combinatorial objects of some particular type has been studied in a number of contexts, including the assignment problem [M], the shortest-path problem [Y], [L1], and the minimum-spanning-tree problem [BH], [CFM], [G], [KIM], [F1], [E]. Early algorithms for finding the k smallest minimum spanning trees can be found in [BH] and [CFM]. Gabow has given an $O(m \log m + km\alpha(m, n))$-time algorithm [G], Katoh, Ibaraki, and Mine have given an $O(m \log \beta(m, n) + km)$-time algorithm [KIM], Frederickson gave an $O(m \log \beta(m, n) + k^2 m^{1/2})$-time algorithm [F1], and Harel claimed an $O(m \log n + kn(\log n)^2)$-time algorithm [Hl2]. Most recently, in [E], Eppstein has given an elegant preprocessing step that allows him to achieve, in conjunction with the algorithm of [KIM], a running time of $O(m \log \beta(m, n) + \min\{k^2, km\})$, using $O(m + k)$ space. Our algorithm matches the running time of Eppstein's for $k \leq (m \log \beta(m, n))^{1/2}$ and is faster for larger values of k. While our algorithm uses more space than Eppstein's for sufficiently large k, the space used by our algorithm is $O(k + m)$ whenever $k \leq m^{2/3}$.

For the case of a planar graph, there are two previous results. In [F1], Frederickson gave an $O(n+k^2(\log n)^3)$-time algorithm, and in [E], Eppstein has given an $O(n+k^2)$-time algorithm. Thus the time for our algorithm is never worse than that in [E], and it is strictly better whenever $k > n^{1/2}$. The space of our algorithm is $O(n)$ whenever $k \leq n/(\log n)^3$.

Our second problem is that of maintaining a data structure for an undirected graph under the operations of inserting and deleting edges and vertices, so as to be able to answer queries about whether two given vertices are in the same 2-edge-connected component. Using ambivalent data structures, we achieve an update time of $O(m^{1/2})$ and a query time of $O(\log n)$, where m and n are understood to be the current number of edges and vertices, respectively.

The question of whether there are data structures with sublinear-time algorithms for maintaining 2-edge-connectivity information under the operations of both insertion and deletion of edges was posed by Westbrook and Tarjan [WT]. Galil and Italiano [GI] describe a data structure that achieves $O(m^{2/3})$ update and query times.

Recently, Eppstein, Galil, Italiano, and Nissenzweig [EGIN] and Eppstein, Galil, and Italiano [EGI] have introduced a sparsification technique that creates a data structure using multiple copies of other data structures. Their technique allows them to use the data structures in this paper and thus replace the m by an n in the times for updating 2-edge-connectivity information and for finding the k smallest spanning trees in a general graph.

Our approach is based on variants of the topology tree and the 2-dimensional topology tree structures presented in [F1]. To make the approach work, we present a different, and in some sense simpler, multilevel partition of the vertices, on which the topology tree and 2-dimensional topology tree are based. A version of our variant of the topology tree that is designed for rooted trees [F3] is appropriate for implementing dynamic trees. In addition to ambivalence, we introduce other novel ideas in our solutions. These include an encoding scheme for vertex names, with the encoded names changing as the topology of a spanning tree changes, and also a partition of the spanning tree into paths based on the multilevel partition.

Our paper is organized as follows. In section 2, we describe the new multilevel partition, and in section 3, we describe the data structures, including fully persistent data structures, used for updating spanning trees. In section 4, we characterize the adjacency of clusters in embedded planar graphs. In section 5, we describe the data structures, including fully persistent data structures, that are use for updating spanning trees in embedded planar graphs. In section 6, we describe the basic algorithm for finding the k smallest spanning trees, omitting the description of the key data structure. In section 7, we define ambivalence formally and then describe this key data structure for general graphs. In section 8, we describe this key data structure for planar graphs. In section 9, we give a data structure to maintain 2-edge-connectivity in general graphs.

2. Clustering vertices in spanning trees. In this section, we define basic data structures similar to but simpler than those in [F1]. The main contribution of the section is a new way to partition the vertices based on the topology of a spanning tree. We first describe a graph transformation that we use throughout. We then define vertex clusters and our new partition and discuss how the clusters change when an edge is inserted or deleted. We then show that the partition can be applied recursively for only $\Theta(\log n)$ levels.

Throughout this paper, we shall wish to deal with graphs that have maximum vertex degree 3. We first describe how to transform our graph into a graph in which every vertex has degree no greater than three. A well-known transformation in graph theory [Hy, p. 132] is used. By ∞ we designate a sufficiently large value, say equal to the largest value that can be represented in a single word of memory. For each vertex v of degree $d > 3$ and neighbors $w_0, w_1, \ldots, w_{d-1}$, replace v with new vertices $v_0, v_1, \ldots, v_{d-1}$. Add edges $\{(v_i, v_{i+1}) | i = 0, \ldots, d-2\}$, each of cost $-\infty$, and edge (v_{d-1}, v_0) of cost ∞, and replace edges $\{(w_i, v) | i = 0, 1, \ldots, d-1\}$ with $\{(w_i, v_i) | i = 0, \ldots, d-1\}$, of corresponding costs. Note that a minimum spanning tree for the transformed graph will be a minimum spanning tree for the original graph with every edge of cost $-\infty$ added. (The value $-\infty$ is used to ensure that these edges are not swapped out when identifying a best swap in section 6. For the purpose of identifying the cost of the minimum spanning tree in the original graph, treat the $-\infty$ as 0.)

We next define some terms that serve as the foundation for data structures from [F1] that we wish to use. Let $G = (V, E)$ be a connected undirected graph with maximum vertex degree at most 3, and let T be a subgraph of G that is a tree. A *vertex cluster* with respect to T is a set of vertices such that the subgraph of T induced on the cluster is connected. A *boundary vertex* of a cluster is a vertex that is adjacent in T to some vertex not in the cluster. The *tree degree* of a vertex cluster is the number of tree edges with precisely one endpoint in the cluster. Two disjoint vertex clusters are *adjacent* if there is a tree edge that contains one endpoint in each of the clusters. We illustrate the above definitions using Fig. 1, in which a spanning tree is shown with bold edges. The set of vertices $\{6, 7, 13\}$ is not a cluster, since the subset of edges of T incident on it is just $\{(6, 7)\}$ so that the subgraph of T induced on it is not connected. The set of vertices $\{4, 10\}$ is a cluster since the subset of edges of T incident on it is $\{(4, 10)\}$, which connects the vertices. The tree degree of cluster $\{4, 10\}$ is 4. Clusters $\{8, 9, 10, 11\}$ and $\{12, 13, 14\}$ are adjacent because there is a tree edge $(11, 12)$.

We define a partition of a set of vertices so that the resulting vertex clusters possess certain nice properties. Let z be a positive integer. A *restricted partition of order* z with respect to T is a partition of V such that:

1. Each set in the partition is a vertex cluster of tree degree at most 3.
2. Each cluster of tree degree 3 is of cardinality 1.

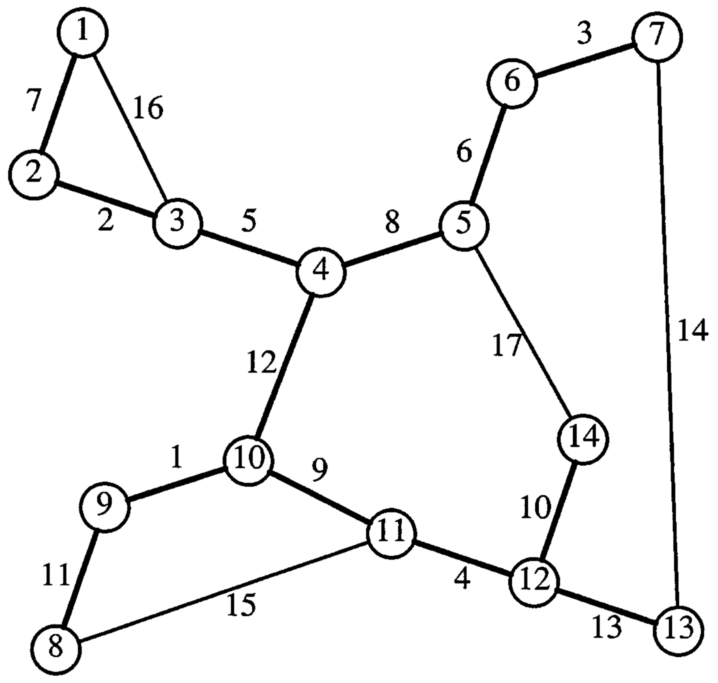

FIG. 1. *A weighted undirected graph with its minimum spanning tree in bold.*

3. Each cluster of tree degree less than 3 is of cardinality at most z.
4. No two adjacent clusters can be combined and still satisfy the above.

As an example, consider the spanning tree from the graph in Fig. 1. A restricted partition of order 2 is shown for this tree in Fig. 2. Note that vertices 10 and 11 cannot be clustered together due to the constraint on the cardinality of a cluster of tree degree 3. Also note that vertex 2 could have been clustered with vertex 1 rather than with vertex 3. In general, there are many different restricted partitions for a given tree and parameter z.

It is not hard to show that the number of clusters in a restricted partition of order z is $\Theta(m/z)$. We do this after the proof of the upcoming Lemma 2.2.

Given a tree described by adjacency lists, a restricted partition can be found as follows. Root the tree at a vertex of tree degree 1. Call the procedure *cluster* with the root as argument. Procedure *cluster*(v), defined below, does the following. It finds all clusters of the subtree rooted at v and outputs all except the partial cluster containing v, which it leaves as $C(v)$. It also finds $tdeg(v)$, the tree degree of $C(v)$, and $size(v)$, the number of vertices in $C(v)$. Upon the return of the call to *cluster*$(root)$, print out the set $C(root)$ as the final cluster. Function *cluster*(v) is defined as follows. It initializes $C(v)$ to $\{v\}$, $size(v)$ to 1, and $tdeg(v)$ to the tree degree of v. Then for each child w of v, it calls *cluster*(w) and then does the following. If $tdeg(v) + tdeg(w) - 2 \leq 2$ and $size(v) + size(w) \leq z$, then it resets $C(v)$ to be $C(v) \cup C(w)$, $tdeg(v)$ to be $tdeg(v) + tdeg(w) - 2$, and $size(v)$ to be $size(v) + size(w)$. Otherwise, $C(w)$ is output as a cluster. This completes the description of how each child is handled, and with it the description of procedure *cluster*. Note that the tree degree of any resulting partial cluster will be correctly computed since the tree degree of two unioned partial clusters will be 2 less than the sum of their tree degrees.

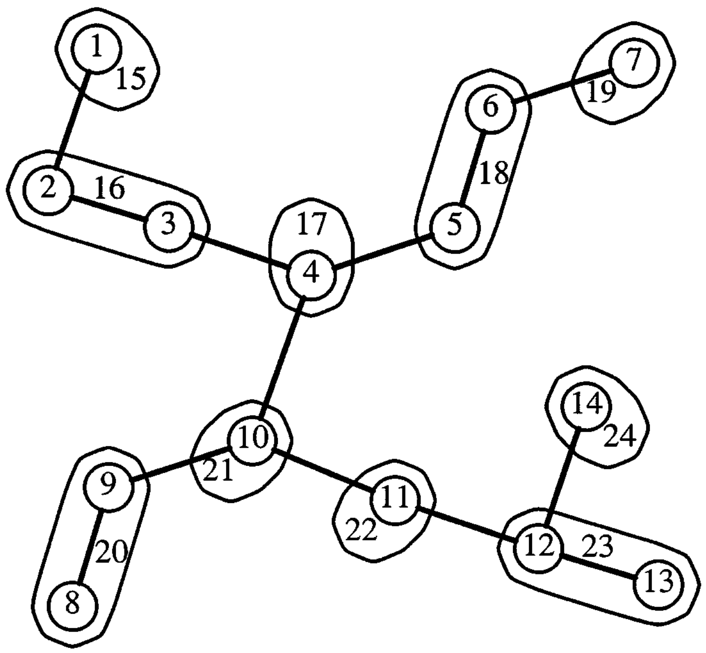

FIG. 2. *A restricted partition of the vertices of the spanning tree in Fig.* 1.

Furthermore, the resulting tree degree will always be less than 3.

The above procedure takes $O(n)$ time for a tree of n vertices. The procedure can be modified in a straightforward fashion to identify the boundary vertices for each cluster and for each vertex identify whether it is a boundary vertex and, if so, for which cluster. Using this representation, the neighboring clusters of any given cluster can be identified in constant time.

An operation that changes the structure of T may force a change in the clusters of a restricted partition. Consider the operation of removing an edge from T, which leaves two trees, T_1 and T_2. Given a restricted partition of order z with respect to T and given an edge in T that is deleted, we discuss how to efficiently generate restricted partitions with respect to the resulting trees T_1 and T_2. Trees T_1 and T_2 inherit the clusters of T, with the following exceptions. Either the edge to be removed has both endpoints contained in one cluster C, or it has one endpoint in a cluster C' and the other in a cluster C''. If the edge to be removed has both endpoints contained in one cluster C, then split C into two clusters, calling them C' and C'', so that the edge has one endpoint in C' and the other in C''. Deleting the edge now causes the tree degree of C' and C'' to change. We need to check if either cluster needs to be combined with other clusters. We discuss how to handle C', with handling C'' being completely analogous. Initialize vertex cluster A to be C'. Repeat the following until no change occurs to A on an iteration. If A has tree degree 1, then check to see if the combined size of it and its neighboring cluster is no greater than z and, if so, combine the neighboring cluster into A. If A has tree degree 2, check to see if there is a neighboring cluster of tree degree no greater than 2 such that the combined size of A and that neighboring cluster is no greater than z and, if so, combine that neighboring cluster into A. This completes the description of how to handle C'.

Next consider the inverse operation of combining two vertex disjoint trees T_1 and

T_2 into one tree T by adding an edge with one endpoint in each of the trees. (Here we assume that the edge was already in the graph, but just not in T.) Given restricted partitions of order z with respect to T_1 and T_2 and given an edge to be inserted that links the trees, we discuss how to efficiently generate a restricted partition with respect to the resulting tree T. Tree T will inherit the clusters of the trees T_1 and T_2, with the following exceptions. If the addition of the edge causes the tree degree of a cluster containing more than 1 vertex to increase from 2 to 3, we must do the following. Consider the three tree edges with exactly one endpoint in the cluster, and let w, w', and w'' denote these endpoints, which are boundary vertices. (Note that two or three of w, w', and w'' may be identical if two or all three of the edges share an endpoint.) Identify the common vertex x on paths in the tree between w and w', between w' and w'', and between w'' and w. Split the cluster by making the vertex x into a cluster by itself and taking the remaining parts of the cluster as clusters. For each cluster so formed, check if another cluster that is adjacent to it is of tree degree at most 2 and if these two clusters together have at most z vertices. If so, combine these clusters. This completes the description of how to handle a cluster when its tree degree increases to 3. Note that these operations can be performed in time proportional to the size of the cluster. If the addition of the edge causes the tree degree of a cluster to increase from 0 to 1 or from 1 to 2, we must do the following. Check to see if the cluster can be combined with the cluster newly adjacent to it. If so, combine these clusters. This completes the description of how to handle a cluster when its tree degree increases from 0 to 1 or from 1 to 2.

LEMMA 2.1. *At most a constant number of clusters are deleted or created when an edge insertion or deletion is performed with respect to restricted partitions of z. The time to perform the changes is $O(z)$.*

Proof. We first observe that only a constant number of clusters are deleted or created when an edge insertion is performed. In the case that a cluster has its tree degree increase from 2 to 3, each part of the split cluster except the part containing only vertex x has tree degree 2. It can be combined with a neighbor B_1 of tree degree 2 but not also combined with the other neighbor B_2 of B_1 since otherwise B_1 and B_2 would have already been combined. In the case of adding an edge that causes the tree degree of a cluster to increase from 0 to 1 or from 1 to 2, a similar argument ensures that only the two clusters containing the endpoints of the edge need to be considered for merging.

We next consider how many clusters are deleted or created when an edge deletion is performed. We perform a case analysis below for how C' can be combined with other clusters. The analysis for C'' is essentially the same. We consider size constraints only when they definitely rule out a case. Subcases are meant to inherit the conditions satisfied by parent cases. *Case* 1: (C' is of tree degree 1.) Then C' can be combined with a neighboring cluster B_1 of tree degree 1, 2, or 3. Let the resulting cluster be B_2. *Case* 1.1: (B_1 is of tree degree 1.) Then B_2 has tree degree 0 and we are done. *Case* 1.2: (B_1 is of tree degree 2.) Then B_2 is of tree degree 1. Let the other neighbor of B_1 be B_3. *Case* 1.2.1: (B_3 is of tree degree 1 or 2.) Then B_2 cannot be combined with B_3 since otherwise B_1 and B_3 would have already been combined. *Case* 1.2.2: (B_3 is of tree degree 3.) Then B_2 and B_3 may be combined, and the resulting cluster B_4 is of tree degree 2. Let the neighbors of B_3, besides B_1, be B_5 and B_6. *Case* 1.2.2.1: (B_5 is of tree degree 1.) Then B_5 cannot be combined with B_4 since it would already have been combined with B_3. *Case* 1.2.2.2: (B_5 is of tree degree 3.) Then B_5 cannot be combined with B_4 because the resulting cluster would have tree degree 3. *Case*

1.2.2.3: (B_5 is of tree degree 2.) Then B_5 can be combined with B_4, but the resulting cluster B_7 cannot be combined with the other neighbor B_8 of B_5 since otherwise B_5 would already have been combined with B_8. A similar discussion holds for cluster B_6, allowing that B_7 could be used in place of B_4 in the arguments. *Case* 1.3: (B_1 is of tree degree 3.) The argument mimics that in Case 1.2.2 and its subcases, but with B_1 in the role of B_3 and C' in the role of B_2. *Case* 2: (C' is of tree degree 2.) The argument mimics those in Cases 1.2.2.1, 1.2.2.2, and 1.2.2.3, with C' in the role of B_4 and with B_5 and B_6 being the neighbors of C'. This completes an analysis of the cases. It is clear that the above operation will examine just a constant number of clusters.

We next consider the time to perform an edge insertion. In the case that a cluster has its tree degree increase from 2 to 3, identifying w, w', and w'' takes constant time, and finding x takes time proportional to the size of the cluster, which is $O(z)$. Splitting the cluster will take $O(z)$ time. Checking neighboring clusters and merging as necessary will take constant time. In the case that a cluster has its tree degree increase to 1 or 2, the time to perform checking and merging is constant.

We next consider the time to perform an edge deletion. If the edge to be deleted has both endpoints contained in one cluster, then the time to split the list of vertices of the cluster into two lists is $O(z)$. All checking and combining will then take constant time. $\square$

We next define our restricted multilevel partition. A *restricted multilevel partition* is a set of partitions of V that satisfy the following:

1. For each level $l = 0, 1, \ldots, q$, the vertex clusters at level l form a partition of V.
2. The clusters at level 0 form a restricted partition of order z.
3. The clusters at any level $l > 0$ constitute a restricted partition of order 2 with respect to the tree resulting from viewing each cluster at level $l-1$ as a vertex.
4. There is precisely one vertex cluster at level q, which contains all vertices.

As an example, consider the spanning tree from the graph in Fig. 1. A restricted multilevel partition for this tree is shown in Fig. 3. Here we assume that $z = 1$ so that each basic vertex cluster contains precisely one vertex. The second level corresponds to the restricted partition in Fig. 2. There are six levels in this multilevel partition.

We note that the restricted multilevel partition is somewhat similar to a structure that may be inferred from applying the rake-and-compress paradigm to a rooted binary tree [MR], [CV], [ADKP].

A vertex cluster at level 0 of a restricted multilevel partition is called a *basic vertex cluster*. Since any basic vertex cluster of tree degree 3 consists of a single vertex and any cluster resulting from the union of two clusters will have tree degree at most 2, any nonbasic cluster of tree degree 3 will also consist of a single vertex. All three of its incident edges will be tree edges. Note that there are no nontree edges with an endpoint in a cluster of tree degree 3.

We next show that the restricted multilevel partition has other nice properties. Consider any level $l > 0$ of a restricted multilevel partition. Call any vertex cluster of level $l-1$ *matched* if it is unioned with another another cluster to give a vertex cluster at level l. Call all other vertex clusters at level $l-1$ *unmatched*. Since a cluster of tree degree 1 can be matched with a cluster of tree degree 1, 2, or 3, the only reason that a cluster of tree degree 1 is not matched is that its adjacent cluster is already matched with some other cluster. Since a cluster of tree degree 2 can be matched

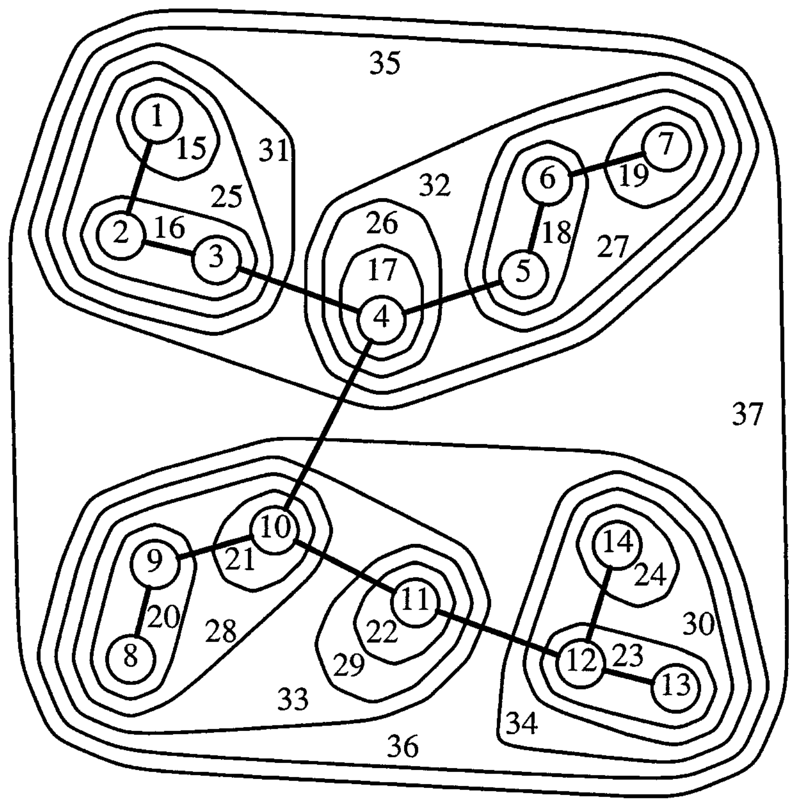

FIG. 3. *A restricted multilevel partition of the vertices of the spanning tree in Fig.* 1.

with an adjacent cluster of tree degree 2, the only reason that a cluster of tree degree 2 is not matched with an adjacent cluster of tree degree 2 is that that adjacent cluster is already matched with another cluster.

LEMMA 2.2. *For any level $l > 0$ of a restricted multilevel partition, the number of matched vertex clusters at level $l-1$ is at least $1/3$ of the total number of vertex clusters at level $l-1$.*

Proof. Consider any level $l > 0$ of a restricted multilevel partition. Contract the graph by contracting all tree edges both of whose endpoints are in the same cluster at level $l-1$. Let each vertex resulting from a matched cluster by such a contraction be called a *matched* vertex. Let the tree degree of a resulting vertex be the tree degree of the corresponding cluster. If all vertices are matched, then clearly the lemma follows. Otherwise, root the tree at an unmatched vertex of largest tree degree. We shall give 6 credits to each pair of vertices that have been matched together, and we will show that these credits can be spread around so that, in the end, each vertex will receive at least 1 credit. The lemma will then follow.

Consider any pair of vertices that have been matched together, and assume that the pair has been allocated 6 credits. Since neither is the root, and since the number of unmatched neighbors of the pair is at most 2, the higher of the two has a parent, which may be unmatched, and the second neighbor (if any) is a child, which may be unmatched. If the higher vertex of the pair has an unmatched parent, let the matched pair send 3 credits to this parent. If there is a second neighbor, let the matched pair send 1 credit to this child. Let each matched vertex in the pair retain at least 1 credit. Call any unmatched vertex of tree degree 2 that is the root or has an unmatched parent of tree degree 3 *sheltered*. From the properties of unmatched clusters discussed prior to the statement of this lemma, every unmatched vertex of tree degree 1 and every unsheltered unmatched vertex of tree degree 2 must have a neighbor that is matched. In particular, the parent of every such vertex will be matched so that the vertex will receive from its parent 1 credit, which it will retain.

The above credit-sharing rule guarantees that if an unmatched vertex has not received a credit from either its children or its parent, then it must be either a vertex of tree degree 3 or a sheltered vertex. We add the following two rules to handle these cases. For any sheltered nonroot vertex, if it receives 3 credits from its child, it should pass 2 credits to its parent and retain the other 1. For any unmatched nonroot vertex of tree degree 3, if it receives at least 2 credits from each of its two children, it should pass 3 credits to its parent and retain the other at least 1 credit. By a simple induction, it can be shown that every sheltered nonroot vertex will receive 3 credits from its child and retain 1 of them, and every unmatched nonroot vertex of tree degree 3 will receive at least 4 credits from its children and retain at least 1 of them. It follows that at the end of all credit passing, each vertex will retain (at least) 1 credit. A root of tree degree 3 will receive at least 2 credits from each of its 3 children, and a root of tree degree 2 will receive 3 credits from each of its 2 children. Since an unmatched vertex of tree degree 1 must have a matched neighbor, an unmatched tree root will receive 3 credits from its child. ∎

We now show that the number of clusters in a restricted partition of order z is $\Theta(m/z)$. Consider a restricted multilevel partition of order z. Level 0 of the multilevel partition is a restricted partition of order z. Consider the vertex clusters at level 0 that are matched together to form vertex clusters at level 1. The total number of vertices in a pair of matched vertex clusters must be greater than z since otherwise the pair could have been merged in the partition at level 0. Thus there are fewer than m/z such pairs of matched vertex clusters. By Lemma 2.2, the total number of these pairs is at least 1/3 of the total number of vertex clusters at level 0. Thus there are fewer than $3m/z$ clusters at level 0, i.e., in the restricted partition.

There is an infinite family of examples that match the bound of Lemma 2.2 in the following way. Let n_{l-1} be the number of clusters at level $l-1$. For $n_{l-1} \geq 13$ and $n_{l-1}+5$ a multiple of 6, the number of matched vertex clusters is at least $(n_{l-1}+5)/3$. It has not escaped our attention that we could match more vertex clusters if rule 3 in the restricted multilevel partition allowed unions whose resulting vertex cluster had tree degree 3. However, it appears difficult and inefficient to update the corresponding topology tree structures when changes occur. (Indeed, the difficulty encountered when trying to make things work with tree degree 3 rather than tree degree 2 in rule 3 was the reason that the multilevel partition was defined as it was in [F1].)

THEOREM 2.3. *The number of levels in a restricted multilevel partition is* $\Theta(\log n)$.

Proof. The number of vertex clusters at level 0 is $O(n)$. By Lemma 2.2, for any level $l > 0$, the number of matched vertex clusters at level $l-1$ is at least 1/3 of the total number of vertex clusters at level $l-1$. Since each pair of matched vertex clusters at level $l-1$ that are paired together are replaced by the union at level l, the number of vertex clusters at level l is at most 5/6 the number of vertex clusters at level $l-1$. Since the number of vertex clusters at level l is at least 1/2 the number of vertex clusters at level $l-1$, it follows that the number of levels is $\Theta(\log n)$. ∎

3. Data structures for maintaining spanning trees.

In this section, we define basic data structures similar to but simpler than those in [F1]. Following [F1], we define a "topology tree" based on the partition and show how to update the topology tree when an edge not in the spanning tree is swapped for an edge in the spanning tree. We then define a "2-dimensional topology tree," again following [F1]. We show how to make 2-dimensional topology trees fully persistent. Finally, we show how to update a 2-dimensional topology tree when edges and vertices are inserted into or deleted from the underlying graph.

As in [F1],we define data structures that describe our partitions. Given a restricted multilevel partition for a spanning tree T, a *topology tree* for T is a tree in which each nonleaf node has at most two children and all leaves are at the same depth, such that:

1. A node at level l in the topology tree represents a vertex cluster at level l in the restricted multilevel partition.
2. A node at level $l > 0$ has children that represent the vertex clusters at level $l-1$ whose union is the vertex cluster it represents.

We label a node in the topology tree by the indexed name of the vertex cluster that the node represents.

A topology tree for the restricted multilevel partition of Fig. 3 is given in Fig. 4. Each node in the topology tree is labeled with the index of the vertex cluster that it represents. The children and parent pointers are represented by the straight, bold edges. The adjacency between clusters is represented by the thin, curved edges.

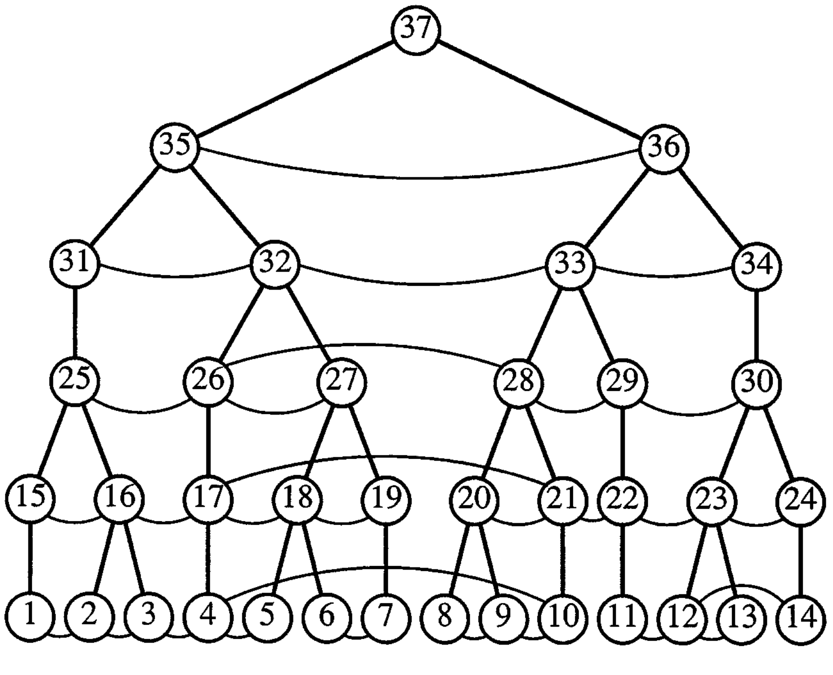

FIG. 4. *The topology tree corresponding to the restricted multilevel partition in Fig.* 3.

A topology tree based on a restricted multilevel partition has the same nice properties as a topology tree based on the multilevel partition of [F1]. In particular, it can be modified efficiently to show the result of inserting or deleting an edge or performing a swap. A *swap* (e, f) in a spanning tree T replaces a tree edge e by a nontree edge

f, yielding another spanning tree. We next discuss how to modify the topology tree when a swap is performed. First, reform the basic clusters to reflect the insertion and deletion of edges, as discussed previously. The number of basic vertex clusters that are changed, created, or deleted will be at most some constant. Then starting with the lists of basic vertex clusters that are changed, created, or deleted, we rebuild portions of the topology tree from the bottom up.

We describe this rebuilding carefully. We shall use three lists of nodes that need to be examined: Let L_D be a list of nodes that represent clusters that should be deleted, L_C a list of nodes that have parents and that represent clusters that have been changed, and L_A a list of nodes that represent clusters that have no parent, either because they are new or because their parent is on a list of nodes to be deleted. Note that we view a cluster as changing not only if its set of vertices changes, but also if its set of tree edges to other clusters changes. (Here we assume that a tree edge to another cluster has not changed if it is the same tree edge as before the swap, i.e., that it has the same vertices as endpoints in the underlying graph even though one of the endpoints may be in a different cluster than before.)

We initialize L_D, L_C, and L_A as follows. Insert into L_D each node representing a basic cluster that has been split or combined to form new basic clusters, and insert into L_A each node representing a new basic cluster. Let the adjacency information of neighboring nodes refer to the nodes inserted into L_A rather than L_D, and let the nodes on L_D retain their parent information but have their adjacency information set to null. For each node x representing a basic cluster whose set of vertices has not changed but whose set of edges incident on it has changed, update its adjacency information and insert it into L_C.

While we have not reached the root of the resulting topology tree, we will recluster the clusters represented by the nodes on these lists and reset the lists to contain the nodes representing the corresponding parents. We perform this activity in such a way that at any point in time, the total size of these lists does not exceed a fixed constant. At each level in the topology tree, we do the following. Assume that the adjacency information of the nodes on L_D, L_C, and L_A reflects the result of the swap but that the adjacency information of the parents of these nodes does not yet. We shall create lists L'_D, L'_C, and L'_A to hold the corresponding nodes at the next higher level. Initialize lists L'_D, L'_C, and L'_A to be empty.

First, we handle list L_D. For every node x in L_D, remove x from L_D (and return it to the available storage pool), remove x as a child from its parent y (if any), and if y then has no children, insert y into L'_D. If y had another child x' and this child is not already on L_C or L_D, then insert x' into L_C. Next, we scan L_C for nodes that have siblings. Let x be such a node on L_C, with parent y and sibling x'. If the cluster corresponding to node y remains a valid cluster, then remove x from L_C (and x' too if it resides on L_C) and insert y into L'_C. By a cluster remaining valid, we mean that there is actually an edge between the cluster representing x and the one representing x', and the tree degree of the cluster for y does not now exceed 2. If the cluster corresponding to y does not remain a valid cluster, then remove x and x' as children of y, remove x from L_C (and also x' if it resides on it), insert x' and x into L_A, and insert y into L'_D.

Finally, we handle nodes on L_A and the remaining nodes on L_C. Let x be such a node. Remove x from the appropriate list. We consider three cases. First, suppose that node x represents a cluster of tree degree 3. If x is adjacent to a node x' representing a cluster of tree degree 1, then do the following. Cluster x and x' together, removing one or both from lists L_A and L_C as necessary. If neither had a parent, then create a parent and insert it into L'_A. If both had a parent, then use the parent y of x, inserting y into L'_C and inserting the old parent y' of x' into L'_D. If just one of x and x' had a parent, use it and insert it into L'_C. This completes the description of how to handle x when it is adjacent to such a node x'. If x is not adjacent to such a node x', then do the following. Cluster node x by itself. If x has a parent, then insert this parent onto L'_C. If x has no parent, then create a parent and insert it onto L'_A. This completes the description of the first case.

The second and third cases are similar. Second, suppose node x represents a cluster of tree degree 2. If x is adjacent to a node x' representing a cluster of tree degree 1 or 2 such that or x' has no sibling (either because it is an only child or because it has no parent), then cluster x and x' together as discussed in the case above in which x represents a cluster of tree degree 3. If node x is not adjacent to such a node x', then cluster x with itself and handle the parent (or lack of one) as discussed in the case above for tree degree 3. Third, suppose node x represents a cluster of tree degree 1. If x is adjacent to a node x' such that x' has no sibling (either because it is an only child or because it has no parent), then cluster x and x' together as discussed in the case above for tree degree 3. Note that if x' represents a cluster of tree degree 1, the resulting cluster must have tree degree 0 so that the corresponding node will be the root of the topology tree. If node x is not adjacent to such a node x', then cluster x with itself and handle the parent (or lack of one) as discussed in the case above for tree degree 3. Fourth, if x represents a node of tree degree 0, then it is the root of a topology tree, and a pointer to it should be saved. This completes the discussion of how to handle a node x on L_A or on L_C.

When all nodes have been removed from L_D, L_C, and L_A, determine and adjust the adjacency information for all nodes on L'_D, L'_C, and L'_A and then reset L_D to be L'_D, L_C to be L'_C, and L_A to be L'_A. This completes the description of how to handle the lists L_D, L_C, and L_A. When L_C and L_A together contain only one node, then this node corresponds to the root of the topology tree. Any additional nodes in L_D should be removed. (These nodes and their ancestors should be returned to the available storage pool.) This completes the description of the algorithm to handle a swap, which we call algorithm *basic_swap*.

LEMMA 3.1. *Consider a topology tree based on a restricted multilevel partition. Algorithm basic_swap performs a swap in $O(z + \log n)$ time.*

Proof. We first consider the correctness of *basic_swap*. The algorithm processes the topology tree level by level in rounds. Before each round, we claim that the only nodes that need to be considered are on L_D, L_A, and L_C and that the adjacency information is valid for the nodes on the level being processed. Furthermore, we claim that for any level after the first that is being processed, clusters for all nodes on that level except for the ones on L_D correspond to a valid restricted partition of the clusters on the next lower level. We prove the above claims by induction on the number of rounds, with the final result being that the structure created is a valid topology tree.

For the basis, note that before the first round, the lists L_D, L_A, and L_C contain precisely those nodes whose corresponding clusters are undergoing some change. Also, the adjacency information for the nodes corresponding to basic clusters has been changed to accurately reflect the changes in basic clusters caused by replacing one edge by another. For the induction step, we consider the point in the execution of the algorithm just before the rth round, $r > 1$. We assume that the claims are true at the point in the execution of the algorithm just before the $(r-1)$st round. The algorithm deletes each node x from L_D and adjusts the information at its parent correctly. The algorithm also examines nodes on L_C that have siblings and splits any node from its sibling if they do not now form a valid cluster. Finally, any nodes that are only children and can be clustered together are clustered together. Any resulting node is put on L'_A if it is a new node and on L'_C if it represents a changed cluster. Thus at this point, the nodes at the next level minus those nodes on L'_D represent the clusters of a valid restricted partition. Nodes on L'_D are those nodes at the next level that should be deleted. Thus the only nodes that need to be considered at the next level are on L'_D, L'_A, and L'_C. Just before the end of the round, the adjacency information is adjusted for all nodes on L'_D, L'_A, and L'_C, and L'_D, L'_A, and L'_C are reassigned to be L_D, L_A, and L_C, respectively. Thus at the beginning of the rth round, all three claims hold.

We next consider the time complexity of *basic_swap*. Since a constant number of basic vertex clusters are changed, deleted, and created and each basic cluster that is altered in some way can be handled in $O(z)$ time, the total cost of handling the basic vertex clusters is $O(z)$. The time to perform this algorithm exclusive of changing the basic clusters will be proportional to the number of nodes in the topology tree that are deleted, examined, and created. We analyze how many nodes can be on the lists L_A, L_C, and L_D at the beginning of any round, and we show this to be bounded by a constant. For a *link*, there are no more than some constant number of nodes on L_A and L_C before the first round. A simple case analysis indicates that this number of nodes can be at most eight before the first round begins. This is realized when two vertices in clusters previously of tree degree 2 are linked together. Each of these may be split into at most four clusters. For a *cut*, there are also no more than some constant number of nodes on L_A and L_C before the first round.

Our analysis depends on the way in which the nodes on L_A and L_C relate to each other within the structure of the tree induced on the level corresponding to a round. If the operation is *link*, then the nodes on lists L_A and L_C form a subtree of the resulting tree induced on that level. If the operation is *cut*, then the nodes on lists L_A and L_C form a subtree of each of the two resulting trees induced on that level.

We first analyze the *link* operation. Before the first round, the subtree induced on nodes in L_A and L_C consists of no more than eight nodes and seven edges. Let a *border edge* be an edge in the tree but not in the subtree that is incident to a node of the subtree. We claim that at any point while the topology tree is being rebuilt, there are at most two border edges on each side of the linking edge in the tree. This is true initially since each of the linked clusters previously had tree degree at most 2. During a round, the subtree can be extended to include a larger portion of the tree in two ways. First, a cluster represented by node y can be recognized to be invalid, where the constituent clusters are represented by x and x', x is on L_C, and x' is not on any list. If the cluster is invalid because the tree degree would now be 3, then either of two cases holds. If x now has tree degree 3 and x' has tree degree 2, then including x' in the subtree does not increase the number of border edges. If x now has tree degree 2 and x' has tree degree 3, then including x' in the subtree increases by one the number of border edges. But in this case x must have previously had tree degree 1. It follows that the linking edge is incident on a vertex in the cluster represented by x and that this node was the only one in the subtree on its side of the linking edge. Thus there was just one border edge (the one incident on x) on its side of the linking edge, and we are now increasing the number to two.

If the cluster is invalid because the two constituent clusters are not adjacent, then before the previous round, the cluster corresponding to node x included a cluster adjacent to a constituent cluster of x'. If x' has tree degree 2, then including x' does not increase the number of border edges. If x' has tree degree 3, then including x' increases the number of border edges by one. But in this case, the old version of x previously had tree degree 1, and its cluster contained one endpoint of the linking edge and thus was the only node in the subtree on one side of the linking edge. So prior to including x' in the subtree, the portion of the subtree on that side of the linking edge had just one border edge. Thus that number is now increased to two.

The second way to extend the subtree is to union a cluster represented by a node x on L_A with a cluster represented by a node x' not on L_A or L_C. If x' has tree degree 2, then including x' does not increase the number of border edges. If x' has tree degree 3, then x must have tree degree 1. This means that the subtree consists only of x. On all subsequent rounds, the subtree will consist of only a single node, and the number of edges incident on it will be at most two. This completes our case analysis. In all cases, the number of border edges will never exceed four.

We next consider how many nodes will be in the subtree at the end of a round, when there are s nodes in the subtree at the beginning of the round. The only clusters that can be determined to be invalid must contain the endpoints of the original *link* operation. Thus at most two clusters will be determined to be invalid, yielding two more nodes for the subtree. Other clusters may enter the subtree by unioning a cluster in the subtree with one not in the subtree, but this results in no net gain in the number of nodes. In the worst case, each node in the subtree that has a border edge incident on it will not have its cluster unioned with one whose node is in the subtree. Of the remaining $s + 2 - 4$ nodes, Lemma 2.2 establishes that at most $5/6$ of that number, or $5(s-2)/6$, will remain. An upper bound on the largest value possible for s is thus determined by the inequality $s \leq 5(s-2)/6 + 4$. This implies that the total size of L_A and L_C will never exceed 14. (A more careful analysis of the proof of Lemma 2.2, considering the constant additive term, will reduce the bound substantially.)

To bound the number of nodes in L_D, consider the original two topology trees representing the two trees linked together. For each round, we consider the minimal subtrees of the induced trees that connect nodes on L_D. These subtrees are of the same form as the subtree induced on nodes of L_A and L_C. By similar arguments, it can be shown that the subtrees, and hence the length of L_D, are bounded by a constant. Thus the total size of lists L_A, L_C, and L_D is bounded by a constant on any round. Since the amount of work per list entry is constant, and by Theorem 2.3 the number of rounds is $O(\log n)$, the total time for a *link* is $O(\log n)$.

The analysis for a *cut* is similar. In each of the two subtrees, there will be at most two border edges. The subtrees can be extended in a fashion similar to that for a *link*. The analysis is essentially the same, yielding equivalent bounds for the lengths of lists L_A, L_C, and L_D. Thus the total time for a *cut* is also $O(\log n)$. $\square$

We trace through an example to illustrate algorithm *basic_swap*. Consider the graph in Fig. 2 and the spanning tree and multilevel partition in Fig. 3. We assume that each node in Fig. 2 represents a basic cluster. Suppose that the edge between V_7 and V_{13} is swapped in to replace the edge between V_4 and V_{10}. For simplicity, we shall assume that no basic cluster is changed as far as the set of vertices it contains. (This would be true if clusters V_3, V_4, V_9, and V_{11} have size exactly z, so V_4 or V_{10} cannot be combined with their neighboring clusters, and the size of V_7 plus the size of V_{13} is greater than z and the size of V_6 plus the size of V_7 is greater than z, so that neither V_7 nor V_{13} can be combined with a neighboring cluster.) For convenience, we use as the name of the node the index of its cluster. Initially, L_D and L_A are empty, and L_C comprises $4, 10, 7$, and 13. When L_C is examined on the first phase, node 13 and its sibling 12 no longer form a valid cluster. Thus node 12 is placed on L_C and node 23 is placed on L'_D. We then proceed to handling the remainder of L_C and L_A. Node 4 cannot be clustered with a neighbor, so its parent 17 is placed on L'_C. Node 10 is clustered with 11, and the resulting parent 21 is placed on L'_C, while the previous parent 22 of 11 is placed on L'_D. Nodes 7 and 13 get clustered, and the parent 19 is placed on L'_C. Node 12 can be clustered with 14, putting resulting parent 24 on L'_C. At this point, L_D, L_A, and L_C are empty, and the next phase begins with setting L_D, L_A, and L_C to L'_D, L'_A, and L'_C.

When L_D is handled on the second phase, nodes 22 and 23 are deleted. Since node 22 is an only child, its parent 29 is put on L'_D. Node 24, the sibling of 23, is already on L_C. Node 28, the parent of 21, remains a valid cluster and is put on L'_C. Similarly, node 27, the parent of 19, remains a valid cluster and is put on L'_C. We then proceed to handling the remainder of L_C and L_A. Node 17 cannot be clustered with a neighbor, so its parent 26 is placed on L'_C. Node 24 cannot be clustered with a neighbor, so its parent 30 is placed on L'_C.

At this point, the second phase ends, and the third phase begins with recopying the lists. Node 29 is removed from L_D, and its sibling 28 is already on L_C. Node 32, the parent of nodes 26 and 27, remains a valid cluster and is put on L'_C. We then proceed to handling the remainder of L_C and L_A. Node 28 is clustered with 30, and the resulting parent 33 is placed on L'_C, while the previous parent 34 of 30 is placed on L'_D. At this point, the third phase ends, and the fourth phase begins. Node 34 is removed from L_D, and its sibling 33 is already on L_C. Node 32 on L_C has a sibling, and its parent cluster 35 is still valid, so that 35 is placed on L'_C. Node 33 cannot be clustered with a neighbor, so its parent 36 is placed on L'_C. At this point, the fourth phase ends, and the fifth phase begins. List L_D is empty. Nodes 35 and 36 each have siblings (each other), and their parent 37 represents a valid cluster, so 37 is placed on L'_C. In the sixth phase, 37 is identified as having tree degree 0, so that it is the root of the new topology tree. The algorithm then terminates. The resulting topology tree, with the old vertices crossed out and the old edges dashed, is shown in Fig. 5. To minimize the clutter in Fig. 5, the edges representing adjacency between clusters are not shown.

A 2-*dimensional topology tree* for a given topology tree is a tree in which for every ordered pair of nodes labeled V_j and V_r at the same level in the topology tree, there is a node labeled $V_j \times V_r$, and there is a child of node $V_j \times V_r$, labeled $V_{j'} \times V_{r'}$, for each pair consisting of a child $V_{j'}$ of V_j and a child $V_{r'}$ of V_r in the topology tree.

A portion of the 2-dimensional topology tree for the topology tree of Fig. 4 is given in Fig. 6. Specifically, all nodes and children of nodes on the path from the root to the leaf 5×14 are shown. For any node that is shown in the figure but whose children are not shown, there is an edge for each child coming from the bottom of that node.

For the size bound z on the number of vertices in a basic cluster, we choose $z = \lceil m^{1/2} \rceil$. When one modifies a topology tree as the result of performing a swap, the 2-dimensional topology tree must be modified. This modification is essentially the same as that discussed in [F1].

LEMMA 3.2. *Consider a 2-dimensional topology tree for a topology tree that is based on a restricted multilevel partition. The space is $O(m)$, the time to set up the*

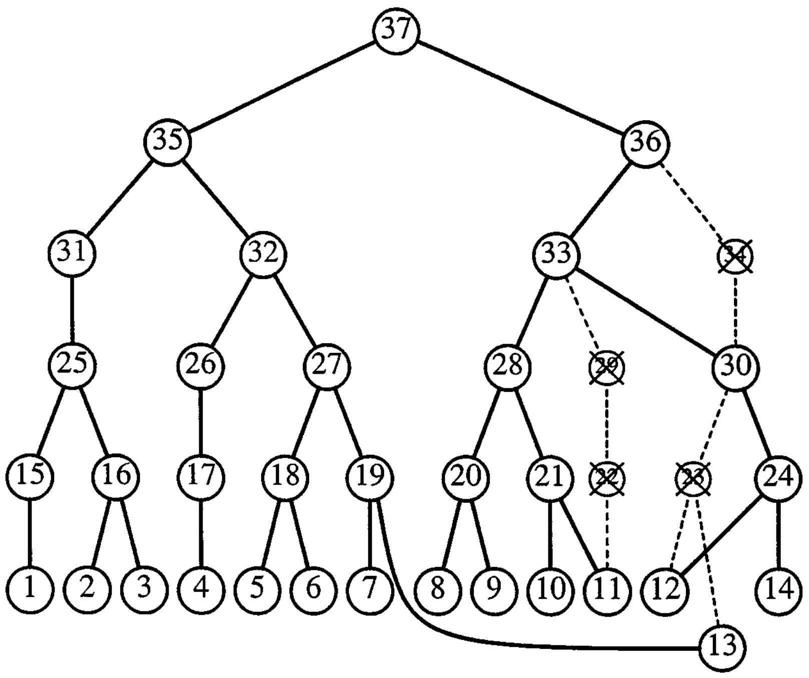

FIG. 5. *The changes to the topology tree in Fig.* 4 *after edge* $(7, 13)$ *replaces edge* $(4, 10)$ *in the spanning tree.*

2-*dimensional topology tree given its topology tree is* $O(m)$, *and the time required to modify the* 2-*dimensional topology tree to show the result of performing a swap is* $O(m^{1/2})$.

Proof. The space and times are derived in a fashion similar to that in [F1]. □

In section 7, we use a version of 2-dimensional topology trees that is fully persistent [DSST]. Thus when we perform a swap, we want to retain the old version of the 2-dimensional topology tree before the swap and also create a version of the 2-dimensional topology tree after the swap. This can be done efficiently by creating new nodes when they are needed and by allowing a new node in the 2-dimensional topology tree to have one or more children in the old 2-dimensional topology tree. Thus certain subtrees of the old 2-dimensional topology tree are shared by both old and new trees by virtue of having two pointers pointing at the root of any such subtree. (After creating a number of versions via swaps, a node could have any number of pointers less than or equal to the number of versions pointing to it.) To make such a scheme work, the version of 2-dimensional topology tree that is made fully persistent is not allowed to have any parent pointers in the nodes.

The discussion in [F1] regarding modifying a 2-dimensional topology tree implicitly supposes that there are parent pointers in the tree. We thus discuss how to perform a swap when the 2-dimensional topology tree does not have these pointers. We maintain a copy of the topology tree for each version of the 2-dimensional topology tree, and since the nodes of these topology trees will not be shared, we allow the nodes of the topology tree to have parent pointers. When we want to perform a swap (e, f) in tree T with topology tree $T1D(T)$ and a pointer to the 2-dimensional topology tree $T2D(T)$, we do the following. First, we make a copy of $T1D(T)$ and run *basic_swap*

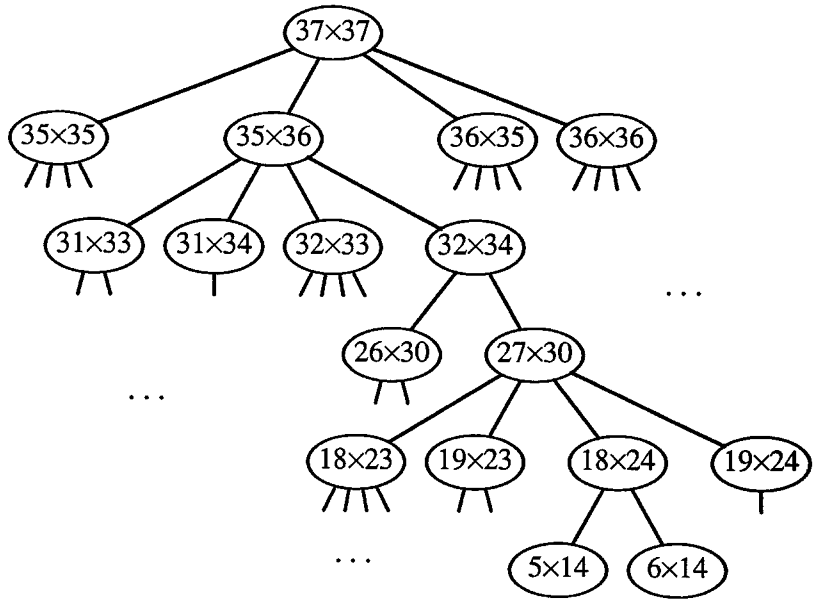

FIG. 6. *A portion of the* 2*-dimensional topology tree for the topology tree shown in Fig.* 4.

on this copy, generating $T1D(T')$. We actually use a modified version of *basic_swap* that marks each node in $T1D(T)$ that has been deleted or whose corresponding node in $T1D(T')$ represents a cluster that has changed. These nodes are all the nodes that have been inserted into L_D, L'_D, or L'_C.

For example, consider the graph and spanning tree in Fig. 1, whose topology tree is shown in Fig. 4, with edge $(7, 13)$ swapped in to replace edge $(4, 10)$, as shown shown in Fig. 5. The vertices in the topology tree in Fig. 4 that should be marked are $4, 7, 10, 13, 17, 19, 21, 22, 23, 24, 26, 27, 28, 29, 30, 32, 33, 34, 35,$ and 36. The subtrees shared by both the old 2-dimensional topology tree, as shown in Fig. 6, and the new 2-dimensional topology tree are those rooted at the nodes $V_{31} \times V_{31}$, $V_{18} \times V_{18}$, $V_{20} \times V_{20}$, $V_{15} \times V_{18}$, $V_{18} \times V_{15}$, $V_{15} \times V_{20}$, $V_{20} \times V_{15}$, $V_{16} \times V_{18}$, $V_{18} \times V_{16}$, $V_{16} \times V_{20}$, $V_{20} \times V_{16}$, $V_{18} \times V_{20}$, and $V_{20} \times V_{18}$ and leaves $V_j \times V_r$ and $V_r \times V_j$, for $j \in \{1, 2, 3, 5, 6, 8, 9\}$ and $r \in \{11, 12, 14\}$ or for $j, r \in \{11, 12, 14\}$.

Having marked those nodes in $T1D(T)$ whose corresponding nodes in the copy are involved in the restructuring that generates $T1D(T')$, we then set temporary parent pointers in a portion of the shared data structure that represents $T2D(T)$. The temporary pointers, as well as the marks in the nodes of $T1D(T)$, will be reset to a null value once $T2D(T')$ has been generated. The temporary pointers will be set for any node $V_j \times V_r$ such that either V_j or V_r or both are marked in $T1D(T)$. The procedure *set_tpp*$(p1, p2)$ will do this, where $p1$ points to a marked node in $T1D(T)$ representing some cluster V_j and $p2$ points to a node in $T2D(T)$ representing an ordered pair of clusters $V_j \times V_r$ or $V_r \times V_j$.

```
**proc** *set_tpp*($p1, p2$)
    **if** $p1$ is a leaf
    **then** Save ($p1, p2$) on a list.
    **else**
```

```
        if $p1$ has just one child $p1c$
        then
            if $p1c$ is marked
            then
                for each child $p2c$ of $p2$ do
                    $temp\_par(p2c) \leftarrow p2$
                    Call $set\_tpp(p1c, p2c)$.
                endfor
            endif
        else
            if the left child $p1L$ of $p1$ is marked
            then
                for each child $p2L$ of $p2$ that corresponds to $V_j \times V_r$
                            where $V_j$ is a left child do
                    $temp\_par(p2L) \leftarrow p2$
                    Call $set\_tpp(p1L, p2L)$.
                endfor
            endif
            if the right child $p1R$ of $p1$ is marked
            then
                for each child $p2R$ of $p2$ that corresponds to $V_j \times V_r$
                            where $V_j$ is a right child do
                    $temp\_par(p2R) \leftarrow p2$
                    Call $set\_tpp(p1R, p2R)$.
                endfor
            endif
        endif
    endif
```

Consider the list of pairs $(p1, p2)$ corresponding to certain leaves that is created by procedure *set_tpp*. This list is then used to initialize a simultaneous bottom-up traversal of the portion of $T1D(T)$ whose nodes are marked and the portion of $T2D(T)$ whose nodes have temporary parent pointers. During the traversal, actions are performed in $T2D(T)$ that are analogous to those of *basic_swap* on $T1D(T)$, except that no nodes in $T2D(T)$ are deleted. Instead, new nodes are created and the child pointers of these nodes are set to both new nodes and nodes in $T2D(T)$. The new nodes and their pointers are set up to be consistent with the structure of $T1D(T')$. A pointer to the root of the resulting 2-dimensional topology tree $T2D(T')$ is returned. Let the above approach be called algorithm *persist_swap*.

THEOREM 3.3. *Algorithm persist_swap maintains a fully persistent version of* 2*-dimensional topology trees, using* $O(m + km^{1/2})$ *space to store k versions and generating a new version reflecting the result of a swap in* $O(m^{1/2})$ *time.*

Proof. We first consider the correctness of algorithm *persist_swap*. Algorithm *basic_swap* correctly marks each node in $T1D(T)$ that has been deleted or whose corresponding node in $T1D(T')$ represents a cluster that has changed. Next, a proof by induction can be used to establish that if a node in $T1D(T)$ is marked, then all ancestors of that node are marked. Now any subtree rooted at a node $V_j \times V_r$ in $T2D(T)$ such that V_j and V_r are not marked will remain the same in $T2D(T')$. Thus the only nodes that may get deleted or changed in modifying $T2D(T)$ to get $T2D(T')$

are nodes $V_j \times V_r$ such that at least one of V_j and V_r are marked. A proof by induction establishes that procedure *set_tpp* correctly sets temporary parent pointers for all such nodes. A bottom-up procedure for deleting and changing all such nodes then can then simulate the effect of *basic_swap* in $T2T$ by following the temporary parent pointers.

We next consider the resource usage of *persist_swap*. Given the choice of z, there are $\Theta(m^{1/2})$ basic clusters, and each is of size $O(m^{1/2})$. It follows from Lemma 2.2 that there are $\Theta(m^{1/2})$ vertices in the topology tree $T1D(T)$. Thus a copy can be made in $O(m^{1/2})$ time. By Lemma 3.1 the modified version of *basic_swap* will take $O(m^{1/2})$ time. By Lemma 3.2, the time used in modifying the 2-dimensional topology tree $T2D(T)$ is $O(m^{1/2})$, and this is a bound also on the number of nodes examined. The time of *set_tpp* is proportional to the number of nodes examined, so that this time is also $O(m^{1/2})$. It follows that the total time to perform a swap is $O(m^{1/2})$. Since the number of new nodes is bounded by the time, each of $k-1$ versions after the first will use $O(m^{1/2})$ additional space. ▯

In section 9, we consider a problem in which the underlying graph can change by inserting or deleting edges or vertices. Inserting edges into or deleting edges from the original graph can be handled similarly to that discussed at the beginning of section 8 of [F1]. We supply some additional explanation since the discussion in [F1] is brief. In particular, we discuss what happens when the insertion of an edge increases the degree of a vertex above 3 or decreases the degree of a vertex that is above 3. In the either case, the transformation that replaces a vertex by a ring of vertices of degree 3 must be modified. For simplicity of discussion, we assume that every vertex of degree at least 2 is converted into a ring of degree-3 vertices. In the case of edge insertion, we define an *inflate* operation as follows. For each endpoint v of the inserted edge, do the following. If the degree of v was previously 1, then treat the existing v as v_0 and treat the endpoint of the new edge as v_1. Generate the topology tree data structures for the single edge. Then insert edges (v_0, v_1) and (v_1, v_0), rebuilding the data structures in a fashion similar to that done in *basic_swap*. If the previous degree d of v was greater than 1, then treat the endpoint of the new edge as v_d. Generate the topology tree data structures for the single edge. If edge (v_0, v_{d-1}) is a tree edge, then identify a nontree edge with which it can swap and perform the swap. Then delete edge (v_0, v_{d-1}) and insert edges (v_{d-1}, v_d) and (v_d, v_0), rebuilding the data structures in a fashion similar to that done in *basic_swap*. Note that the vertices v_0 through v_{d-1} are identified by position rather than labeled as such. Identifying a nontree edge which can participate in a swap can be accomplished by organizing the data structure for maintaining a minimum spanning tree and changing the cost of the edge to be deleted to ∞.

An operation *deflate* can be defined to perform essentially the reverse of *inflate*. If an edge to be deleted is a tree edge, first determine if there is a nontree edge that can be swapped for the edge to be deleted and, if so, then perform the swap.

It is not hard to cast the problems of edge and vertex insertion and deletion as edge insertion or deletion. We allow a vertex to be inserted whenever it is an endpoint of an edge that is being inserted and the other endpoint of the edge is already in the graph. A vertex is deleted whenever it is an endpoint of degree 1 and its incident edge is being deleted. (We thus force our graph to always be connected.)

THEOREM 3.4. *Consider a structure based on a restricted multilevel partition. The time required to insert or delete an edge or vertex is $O(m^{1/2})$, where m is the current number of edges.*

Proof. As in [F1], splitting and merging basic vertex sets will use $O(z)$ time. The time to modify all affected nodes in the topology and 2-dimensional topology tree will be $O(m/z)$. ∎

4. Adjacency in embedded planar graphs.

For embedded planar graphs, we characterize the adjacency relationships between clusters in the multilevel partition presented in section 3. Nontree edges with precisely one endpoint in any given vertex cluster will be grouped together and ordered according to the embedding. Our work will follow the general idea in [F1] but will elaborate the details with more care than in [F1].

We shall first choose a size for basic vertex clusters and then make a number of simple observations about the consequences of this choice. We then define sets of nontree edges, called "boundary sets," with precisely one endpoint in any given vertex cluster. We show how to generate a boundary set of a cluster that is the union of two other clusters from their boundary sets. The situation is complicated by what we call "separating edges," which are not always easy to identify efficiently. Our approach will first generate "pseudoboundary sets," which can contain the separating edges, and then later at opportune times it will remove the separating edges to give the boundary sets.

First, we choose $z = 1$ in our restricted multilevel partition so that each basic vertex cluster will be a vertex by itself. We carefully examine how to represent the nontree edges. Recall that a cluster of tree degree 3 will consist of a single vertex and will have no nontree edges incident on it. Next, consider a cluster V_j of tree degree 1. All nontree edges with exactly one endpoint in V_j can be ordered in clockwise order around V_j, starting with the first edge in a clockwise direction from the tree edge with one endpoint in V_j. This ordering will be entirely consistent with the embedding. Next, consider a cluster V_j of tree degree 2. There will be a unique path of tree edges between the two boundary vertices of V_j. We partition all nontree edges with precisely one endpoint in V_j into two sets, depending on which "side" of the path an edge is incident on. Each set can be ordered in a natural way corresponding to the embedding and represented by a balanced tree. Call each such ordered set of edges a *boundary set*. It is easy to identify the zero, one, or two boundary sets of a basic cluster in constant time, given a list of edges incident on the single vertex in the cluster, as well as an indication of which edges are tree edges.

Consider the embedded planar graph in Fig. 7. The spanning tree edges are in bold, the nontree edges are dashed, and a multilevel partition is shown by the closed curves. The vertex cluster $\{10, 11\}$ has an associated path from vertex 10 to vertex 11. Cluster $\{10, 11\}$ has edges $(10, 9)$ and $(11, 7)$ in the boundary set to the left of the path and no edges in the boundary set to the right of the path. Cluster $\{4, 5, 6, 7\}$ has an associated path from vertex 4 to vertex 7. It has edge $(5, 3)$ in one boundary set and edge $(7, 11)$ in the other boundary set. Note that edge $(6, 4)$ has both endpoints in this cluster and thus is not in either boundary set.

We next discuss how to determine the boundary sets of the clusters. Clearly, a cluster that has just one cluster as its child has the same boundary sets as its child. Given cluster V_j that is the result of the union of two clusters $V_{j'}$ and $V_{j''}$, we show how to generate the boundary set of V_j from the boundary sets of $V_{j'}$ and $V_{j''}$. We first discuss two simple cases. The first case is that $V_{j'}$ and $V_{j''}$ are both of tree degree 1. Then V_j is the set of all vertices, and all edges in the boundary sets of $V_{j'}$ and $V_{j''}$ will be interior with respect to V_j. Thus there will be no boundary set for V_j. The second case is that $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3. In this case, V_j will be of tree degree 2. Make the boundary set of $V_{j'}$ one of the two boundary sets

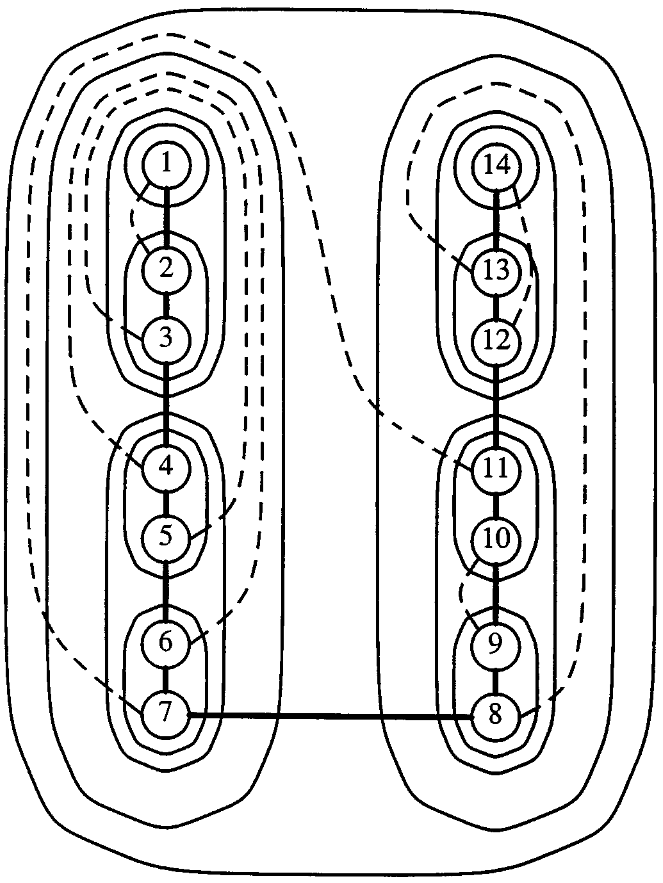

FIG. 7. *An embedded planar graph, its spanning tree, and a multilevel partition.*

of V_j. The other boundary set of V_j is empty.

The third case is that $V_{j'}$ and $V_{j''}$ are both of tree degree 2. Each of $V_{j'}$ and $V_{j''}$ will have two boundary sets, one on each side of the path between the boundary vertices of V_j. Consider a nontree edge with one endpoint in each of $V_{j'}$ and $V_{j''}$. If every such edge is a member of two boundary sets that are on the same side of the path, then things are easy: we shall describe in due course a simultaneous search of two boundary sets to identify the subset of edges contained in both boundary sets. However, suppose that there are nontree edges that are members of the boundary set of $V_{j'}$ on one side of the path and are also members of the boundary set of $V_{j''}$ on the other side of the path. We call any such edge e a *separating edge for* V_j since the cycle induced by e in the tree separates some pair of clusters that are different from V_j. Let the *separating set* of edges for V_j be the separating edges with one endpoint in $V_{j'}$ and the other endpoint in $V_{j''}$. Consider, for example, Fig. 7. The cluster containing vertices 4 and 5 has a separating edge $(4, 5)$. It separates the cluster containing vertices 2 and 3 from the cluster containing vertices 6 and 7. The presence of separating edges complicates matters considerably since there appears to be no efficient way to identify this set by merely examining the boundary sets for $V_{j'}$ and $V_{j''}$. Our solution will be to avoid identifying these edges when initially examining the union of two clusters of tree degree 2 and to identify only "pseudoboundary sets" at that time.

A *pseudoboundary set* of a cluster V_j of tree degree 2 is defined as follows. If V_j is a basic vertex cluster, then the pseudoboundary sets of V_j are identically the boundary sets of V_j. If V_j has just one child cluster, then the pseudoboundary sets of V_j are identically the pseudoboundary sets of the child cluster. If V_j is the union of two clusters of tree degree 1 and 3, then the pseudoboundary sets of V_j are identically the boundary sets of V_j. Otherwise, V_j is the union of two clusters $V_{j'}$ and $V_{j''}$ of tree degree 2. A simultaneous search of each pseudoboundary set of $V_{j''}$ and the pseudoboundary set of $V_{j'}$ that is on the same side of the path can be performed to identify the subset of edges common to both sets. Split these subsets off from the pseudoboundary sets of $V_{j'}$ and $V_{j''}$ and concatenate the remaining portions to get the two pseudoboundary sets of V_j. It follows that the edges in the pseudoboundary sets of V_j that are not in the boundary sets of V_j are separating edges of either V_j or certain clusters that are descendants of V_j. (In a multilevel partition, one cluster is a descendant of another cluster if the former is contained in the latter.)

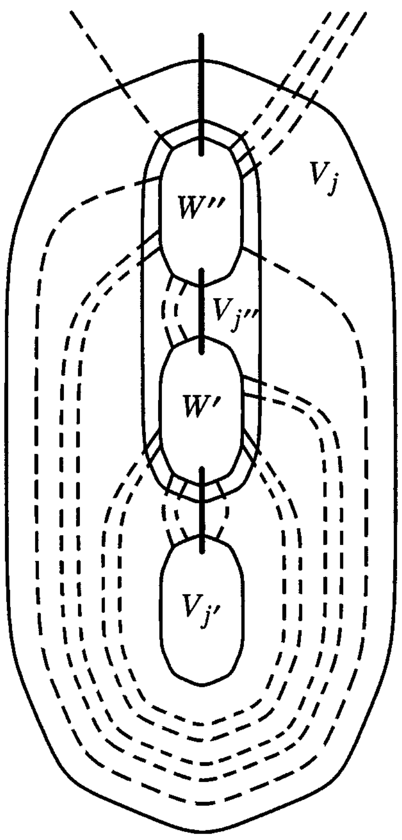

FIG. 8. *An example that illustrates finding sets of separating edges.*

Consider Fig. 8, in which the spanning tree edges are in bold, the nontree edges are dashed, and a portion of a multilevel partition is shown by the closed curves. There are two separating edges for cluster W', one seperating edge for cluster W'', and two separating edges for cluster $V_{j''}$. Looking just at $V_{j''}$, there is no efficient way to find the separating edges for $V_{j''}$ because they are sandwiched between edges to clusters that are not unioned yet (at this level) to $V_{j''}$. The left pseudoboundary set of W'' contains six edges, and the right contains four edges. The left pseudoboundary set of W' contains six edges, and the right contains five edges. The left pseudoboundary set of $V_{j''}$ contains eight edges, and the right contains nine edges.

The fourth and final case is that $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2. Let the sides of the path between the boundary vertices of $V_{j''}$ be designated as L for left and R for right. We differentiate the directions "down" and "up," with down being closer to $V_{j'}$ and up being farther away from $V_{j'}$. Perform a simultaneous search from either end of the boundary set of $V_{j'}$ with the appropriate pseudoboundary set of $V_{j''}$ to identify the two subsets of edges of set $V_{j'}$ that are in either of the pseudoboundary sets of $V_{j''}$. Split these two subsets off from the boundary set of $V_{j'}$ and from the pseudoboundary sets of $V_{j''}$. If any edges remain in the boundary set of $V_{j'}$, then there are no separating edges for $V_{j''}$. In this case, take the remaining portions of the two pseudoboundary sets of $V_{j''}$ and concatenate them with the remaining portion of the boundary set of $V_{j'}$ to give the boundary set of V_j. If no edges remain in the boundary set of $V_{j'}$, then one can identify all separating edges of V_j (and also of certain of its descendants) that are in the pseudoboundary set of $V_{j''}$. Perform a simultaneous search from the lower end of the remaining portions of the two pseudoboundary sets of $V_{j''}$ to identify all edges that are in the remaining portions of both pseudoboundary sets. Then split this subset off from the remaining pseudoboundary sets of $V_{j''}$. Take the remaining portions of the two pseudoboundary sets of $V_{j''}$ and concatenate them together to give the boundary set of V_j.

We return to our example in Fig. 7. When clusters $\{8, 9\}$ and $\{10, 11\}$ are unioned together, the edge $(9, 10)$ is identified as staying on the same side of the path (from vertex 8 to vertex 11). Thus it is not in the pseudoboundary sets for $\{8, 9, 10, 11\}$. When clusters $\{4, 5\}$ and $\{6, 7\}$ are unioned together, edge $(6, 4)$ is a separating edge, though it would not in general be determined as such at that time. The pseudoboundary sets of cluster $\{4, 5, 6, 7\}$ are $\{(5, 3), (6, 4)\}$ and $\{(4, 6), (7, 11)\}$. When cluster $\{1, 2, 3\}$ of tree degree 1 and cluster $\{4, 5, 6, 7\}$ of tree degree 2 are unioned together, we detect the separating edges as follows. First, cluster $\{1, 2, 3\}$ has a boundary set containing edge $(3, 5)$, which is deleted from that set as well as from a pseudoboundary set for $\{4, 5, 6, 7\}$. The separating edge $(6, 4)$ is uncovered from each pseudoboundary set of $\{4, 5, 6, 7\}$. Note that there are no separating edges to be discovered within either cluster $\{4, 5\}$ or $\{6, 7\}$. The boundary set for cluster $\{1, 2, \ldots, 7\}$ will then just contain edge $(7, 11)$.

Note that we have not yet completed our discussion of how to determine the boundary sets of all clusters since the above only determines pseudoboundary sets for the case of a cluster that is the union of tree degree 2. First, we make some additional remarks about how to represent and manipulate these sets of edges. For each cluster, keep track of the portions of pseudoboundary sets of the children that are not used in building the pseudoboundary set of the cluster. We call the set of such edges on each side to be the *newly interior set* of edges.

We next describe how to take the separating edges determined when clusters $V_{j'}$ of tree degree 1 and $V_{j''}$ of tree degree 2 are unioned together, and we determine the boundary sets and separating sets for the descendant clusters for which only pseudoboundary sets were previously known. Let W be any descendant cluster of $V_{j''}$ such that all ancestors of W that are also descendants of $V_{j''}$ have tree degree 2. Call such a cluster W a *relevant descendant* for $V_{j''}$. We introduce the following notation.

Let $pbs_L(W)$ be the pseudoboundary set of W on the left side of the path, and let $pbs_R(W)$ be the pseudoboundary set of W on the right side of the path. Let $s_T(W)$ be the number of separating edges with both endpoints in W. Let $d_L(W)$ be the number of edges in $pbs_L(W)$ with the other endpoint down from W, and let $u_L(W)$ be the number of edges in $pbs_L(W)$ with the other endpoint up from W. Similarly, let $d_R(W)$ be the number of edges in $pbs_R(W)$ with the other endpoint down from W, and let $u_R(W)$ be the number of edges in $pbs_R(W)$ with the other endpoint up from W.

To determine the boundary sets and separating sets for all relevant descendants of $V_{j''}$, call the recursive procedure *sep_edge* with arguments $V_{j''}$, $s_T(V_{j''})$, $u_L(V_{j''})$, $u_R(V_{j''})$, $d_L(V_{j''})$, and $d_R(V_{j''})$. Procedure $sep\_edge(W, s_T(W), u_L(W), u_R(W), d_L(W), d_R(W))$ will take a relevant descendant W of tree degree 2 and construct representations of the boundary sets and separating sets of W and all of its relevant descendants. We make the following notational simplifications. The argument (W) will be omitted, using, for example, u_L rather than $u_L(W)$. In the case that W has two children, W' will be down from W''. An argument such as (W'') will be replaced by a double prime, so that u''_L represents $u_L(W'')$ and similarly for (W'). The number of separating edges of W, i.e., separating edges with one endpoint in W' and the other in W'', will be s. The number of such separating edges with the left endpoint higher than the right will be s_L, and the number of such separating edges with the right endpoint higher than the left will be s_R. The number of edges that are in both $pbs_L(W'')$ and $pbs_L(W')$ will be c_L, and the number of edges that are in both $pbs_R(W'')$ and $pbs_R(W')$ will be c_R. Let $bs_L(W)$ and $bs_R(W)$ represent the left and right boundary sets, respectively, of W, and let $ss(W)$ represent the separating set of W.

```
**proc** $sep\_edge(W, s_T, u_L, u_R, d_L, d_R)$
    **if** the boundary sets of $W$ are not defined
    **then**
        **if** $W$ has a single child $W'$
        **then**
            Call $sep\_edge(W', s_T, u_L, u_R, d_L, d_R)$.
            $bs_L(W) \leftarrow bs_L(W')$; $bs_R(W) \leftarrow bs_R(W')$; $ss(W) \leftarrow \emptyset$
        **else**
            Let $W'$ and $W''$ be the children of $W$.
            $u'_L \leftarrow u_L$; $u''_L \leftarrow u_L$; $d'_L \leftarrow d_L$; $d''_L \leftarrow d_L$
            $u'_R \leftarrow u_R$; $u''_R \leftarrow u_R$; $d'_R \leftarrow d_R$; $d''_R \leftarrow d_R$
            **if** $s_T = 0$
            **then**
                $bs_L(W) \leftarrow pbs_L(W)$; $bs_R(W) \leftarrow pbs_R(W)$; $ss(W) \leftarrow \emptyset$
                Call $sep\_edge(W', 0, u'_L, u'_R, d'_L, d'_R)$.
                Call $sep\_edge(W'', 0, u''_L, u''_R, d''_L, d''_R)$.
            **else**
                Determine $c_L$ by a simultaneous search from the top of $pbs_L(W')$
                    and the bottom of $pbs_L(W'')$.
                Determine $c_R$ similarly.
                **if** the $(u_L + 1)$st edge down in $pbs_L(W'')$
                    is not the same as the $(u_R + 1)$st edge down in $pbs_R(W'')$
                **then** $s''_T \leftarrow 0$
                **else**
```

```
                    Determine $s''_T$ by a search down in $pbs_L(W'')$ and $pbs_R(W'')$,
                        starting at positions specified in the if-expression, re-
                        spectively.
                    $d''_L \leftarrow size(pbs_L(W'')) - u_L - s''_T$
                    $d''_R \leftarrow size(pbs_R(W'')) - u_R - s''_T$
                **endif**
                **if** the $(d_L + 1)$st edge up in $pbs_L(W')$
                    is not the same as the $(d_R + 1)$st edge up in $pbs_R(W')$
                **then** $s'_T \leftarrow 0$
                **else**
                    Determine $s'_T$ by a search up in $pbs_L(W')$ and $pbs_R(W')$,
                        starting at positions specified in the if-expression, re-
                        spectively.
                    $u'_L \leftarrow size(pbs_L(W')) - d_L - s'_T$
                    $u'_R \leftarrow size(pbs_R(W')) - d_R - s'_T$
                **endif**
                $s \leftarrow s_T - s'_T - s''_T$
                **if** $(s > 0)$ **and**
                    $[(s''_T > 0$ **and** the $(u_L + s''_T + 1)$st edge down in $pbs_L(W'')$
                     is the same as the $(c_R + 1)$st edge down in $pbs_R(W'))$ **or**
                    $(s'_T > 0$ **and** the $(d_R + s'_T + 1)$st edge up in $pbs_R(W')$
                        is the same as the $(c_L + 1)$st edge up in $pbs_L(W''))$ **or**
                    $(s'_T = 0$ **and** $s''_T = 0$ **and** $(u_L + 1)$st edge down in $pbs_L(W'')$
                        is the same as the $(d_R + s)$th edge up in $pbs_R(W'))]$
                **then** $s_L \leftarrow s$; $s_R \leftarrow 0$
                **else** $s_L \leftarrow 0$; $s_R \leftarrow s$
                **endif**
                Let $ss(W)$ be the corresponding set of $s$ edges.
                Call $sep\_edge(W', s'_T, u'_L, u'_R, d'_L, d'_R)$.
                Call $sep\_edge(W'', s''_T, u''_L, u''_R, d''_L, d''_R)$.
                Create $bs_L(W)$
                    by deleting from $bs_L(W'')$ the bottom $c_L$ edges and the
                        $(u''_L + 1)$st through $(u''_L + s_L)$th edges from the top,
                    deleting from $bs_L(W')$ the top $c_L$ edges and the
                        $(d''_L + 1)$st through $(d''_L + s_R)$th edges from the bottom,
                    and concatenating what remains of $bs_L(W'')$ and $bs_L(W')$.
                Create $bs_R(W)$ in a similar way.
            **endif**
        **endif**
    **endif**
```

As an example, consider the portion of an embedded planar graph in Fig. 8. The spanning tree edges are in bold, the nontree edges are dashed, and a portion of a multilevel partition is shown by the closed curves. In particular, a cluster V_j is shown that is the union of a cluster $V_{j'}$ of tree degree 1 and a cluster $V_{j''}$ of tree degree 2. As discussed above, the set of separating edges of $V_{j''}$, as well as the set of separating edges of certain descendants of $V_{j''}$, can be determined. There are three edges in the boundary set of $V_{j'}$ and eight and nine edges, respectively, in the left and right pseudoboundary sets of $V_{j''}$. There are two edges in the newly interior set to the left of the path in Fig. 8 and none in the newly interior set to the right of the path. There are one edge from the left pseudoboundary set and three from the right pseudoboundary set that will be in the boundary set of V_j. For the sake of discussion, let W be $V_{j''}$. Then $s_T = 5$, $u_L = 1$, $u_R = 3$, $d_L = 2$, and $d_R = 1$. On the recursive call $\mathit{sep\_edge}(V_{j''}, 5, 1, 3, 2, 1)$, it is determined that W has two children W' and W''. It would then be determined that $c_L = 2$, $c_R = 0$, $s''_T = 1$, $d''_L = 2$, $d''_R = 1$, $s'_T = 2$, $u'_L = 2$, $u'_R = 0$, $s = 2$, $s_L = 2$, and $s_R = 0$. By initialization, $u''_L = 1$, $d'_L = 2$, $u''_R = 3$, and $d'_R = 1$. Note that $\mathit{size}(\mathit{pbs}_L(W'')) = 6$, $\mathit{size}(\mathit{pbs}_L(W')) = 6$, $\mathit{size}(\mathit{pbs}_R(W'')) = 4$, and $\mathit{size}(\mathit{pbs}_R(W')) = 5$. We have not shown the internal structure of W' and W'', and so we will not discuss what happens during the recursive calls $\mathit{sep\_edge}(W', 2, 2, 0, 2, 1)$ and $\mathit{sep\_edge}(W'', 1, 1, 3, 2, 0)$. Upon the returns, $\mathit{bs}_L(W)$ would be created, containing three edges, and $\mathit{bs}_R(W)$ would be created, containing four edges.

We designate as algorithm *build_sets* the algorithm presented above to compute the boundary sets, newly interior sets, and separating sets for the clusters in a multilevel partition of a planar graph.

LEMMA 4.1. *Algorithm build_sets correctly computes the boundary sets, newly interior sets, and separating set for the clusters in a multilevel partition of a planar graph.*

Proof. We shall prove by induction that for any level $l \leq q$, *build_sets* correctly computes pseudoboundary sets and newly interior sets for all clusters of tree degree 2 whose level is at most l and that have no ancestor of tree degree 1 whose level is at most l, and it correctly computes boundary sets, newly interior sets, and separating set for all other clusters whose level is at most l. Since every cluster except the one containing all vertices has an ancestor of tree degree 1, the lemma will then follow.

The proof is by induction on level number. For the basis, $l = 0$. Any cluster at level at most 0 is a basic cluster, and the identification of boundary sets is clearly correct, as is the identification of pseudoboundary sets for clusters of tree degree 2. For the induction step, $l > 0$. We assume as the induction hypothesis that the above claim holds for clusters at any levels $l' < l$. For the newly interior sets of a cluster V_j, an examination of cases indicates that these are correctly computed from the boundary or pseudoboundary sets of the children of V_j.

For the sets other than newly interior, we consider cases for a cluster V_j at level l. Suppose cluster V_j at level l is of tree degree 0. It will have no boundary set since all vertices are contained within it. Suppose cluster V_j at level l is of tree degree 2. If V_j has just one child cluster, then by the induction hypothesis, that child's pseudoboundary set is correct. Clearly, the pseudoboundary set of V_j will be the same set. If V_j is the union of two clusters of tree degree 1 and 3, then its boundary sets and pseudoboundary sets are formed from the boundary set for its child of tree degree 1, which is correctly generated, by the induction hypothesis. Otherwise, V_j is the union of two clusters of tree degree 2. By the induction hypothesis, the pseudoboundary sets of the children are correctly computed. The only edges in those sets that are not in the pseudoboundary set of V_j are the edges from one child to the other that stay on the same side of the path between the boundary vertices of V_j. Algorithm *build_sets* correctly identifies these and removes them before concatenating the remaining lists of edges.

Next, suppose cluster V_j at level l is of tree degree 1. If V_j has just one child cluster, then by the induction hypothesis, that child's boundary set is correct. Clearly, the boundary set of V_j will be the same set. Otherwise, V_j is the union of two clusters

$V_{j'}$ and $V_{j''}$ of tree degree 1 and 2, respectively. By the induction hypothesis, the boundary set of $V_{j'}$ and the pseudoboundary set of $V_{j''}$ are computed correctly. If any edges in the boundary set of $V_{j'}$ have their other endpoint in a cluster up from $V_{j''}$, then no edge in the pseudoboundary set of $V_{j''}$ can be a separating edge. Thus removing those edges from the pseudoboundary sets of $V_{j''}$ that are in the boundary set of $V_{j'}$ and then concatenating the remainder will give the boundary set of V_j. Otherwise, there can be separating edges in the pseudoboundary sets of $V_{j''}$. A pseudoboundary set will contain first the edges with the other endpoint down from $V_{j''}$, then the separating edges for $V_{j''}$ and its relevant descendants, and finally the edges with the other endpoint up from $V_{j''}$. Thus, after removing edges from the boundary set of $V_{j'}$, the separating edges come next, in order, starting with the lowest edge in the remaining portions of the pseudoboundary sets of $V_{j''}$. Once these are identified and removed, the remaining portions of the pseudoboundary sets of $V_{j''}$ will comprise the boundary set of V_j.

Finally, consider the relevant descendants of cluster $V_{j''}$. We argue that *sep_edge* correctly computes the boundary sets and separating sets of all such clusters that have their boundary sets undefined. By the induction hypothesis, the pseudoboundary sets have been computed correctly. The proof that *sep_edge* correctly computes the boundary sets and separating sets is by induction on the distance to the deepest relevant descendant. For the basis, the distance is zero, and the cluster W has no proper relevant descendant. Then W is either a basic cluster or the union of clusters of tree degree 1 and 3. In both cases, the boundary sets of W will already be defined. For the induction step, we have the following. If the boundary set of W is already defined, then nothing need be done since all relevant descendants of W will have their boundary sets defined. Otherwise, if W has a single child W', then the recursive call for W' will, by the induction hypothesis for *sep_edge*, correctly compute the boundary sets of all relevant descendants of W'. Then the boundary set of W will be the boundary set of W', and there will be no separating set for W. Finally, if W has two children, then we have the following. If there are no separating edges with endpoints in W, then the boundary sets of W are the same as the pseudoboundary sets of W. In this case, there will be no separating edges with endpoints in either W' or W'', so that it does not matter what the u_L, u_R, d_L, and d_R parameters are in the recursive calls on W' and W''. Clearly, the number s of actual separating edges for W will equal the total number of separating edges with both endpoints in W minus the total number of separating edges with both endpoints in W' minus the total number of separating edges with both endpoints in W''. If $s > 0$, then one of s_L and s_R is 0 and the other is s. For $s_L > 0$, we must have one of the following cases. If $s''_T > 0$, then there are no edges from W' to a cluster up from W'', and thus the topmost separating edge out of the right side of W' is the $(c_R + 1)$st edge from the top in $pbs_R(W')$. If $s'_T > 0$, then there are no edges from W'' to a cluster down from W', and thus the bottommost separating edge out of the left side of W'' is the $(c_L + 1)$st edge from the bottom in $pbs_L(W'')$. If $s'_T = 0$ and $s''_T = 0$, then the $(u_L + 1)$st edge down in $pbs_L(W'')$ is the topmost separating edge for W, as is the $(d_R + s)$th edge up in $pbs_R(W')$. Upon the return from the recursive calls, the separating and nonseparating edges between W' and W'' are located and removed from the boundary sets of W' and W'', and the results are concatenated to give the boundary sets of W. Note that if $s_L > 0$, then edges up from the left of W are necessarily precisely the edges up from the left of W'', and if $s_L = 0$, it does not matter whether u''_L is set correctly or not. Similar remarks apply to the other cases. $\square$

5. Data structures for embedded planar graphs. We describe the data structures for representing embedded planar graphs and show how to update them quickly when an update occurs. We use the topology tree of section 3 as a basis for our update data structure. Edges in any boundary, pseudoboundary, or newly interior set of a vertex cluster will be ordered according to the embedding and then represented by a balanced tree structure called an "edge-ordering tree." Next we define an "edge-ordered topology tree," which is a topology tree augmented by the edge-ordering information. We discuss how to update the edge-ordered topology tree to show the result of a swap. We then discuss how to update the edge-ordered topology tree to show the effect of the insertion or deletion of an edge or vertex. Finally, we show how to make edge-ordered topology trees fully persistent by introducing "internal names" of vertices, which are based on a vertex's position in the topology tree, and by showing how to keep track of internal indices while performing operations that change the structure of edge-ordering trees.

Let each set of nontree edges associated with a cluster, either a boundary, pseudoboundary, or newly interior set, be represented by a balanced tree called an *edge-ordering tree*. Each leaf in the edge-ordering tree will represent an edge in the corresponding set. When the pseudoboundary set of a cluster V_j is formed from the pseudoboundary sets of the children, do not change the edge-ordering trees for the pseudoboundary sets of the children, but rather build a new edge-ordering tree by introducing some new nodes and sharing subtrees with the already existing edge-ordering trees. The same idea applies for generating a representation of the boundary sets and the newly interior sets.

We define an *edge-ordered topology tree* for an embedded planar graph of maximum degree 3 to be the topology tree, along with pointers from each node in the topology tree to the edge-ordering trees for its one or two boundary sets, and its one or two newly interior sets. It is understood that the root of the tree has a pointer to an empty boundary set. We then note that the algorithm *build_sets* from the last section can be adapted to build an edge-ordered topology tree.

We now consider how to swap a nontree edge into the tree, replacing a tree edge. We will perform an operation similar to *basic_swap* of section 3, with the following additional work. When a node in the topology tree is removed, its boundary, pseudoboundary, and newly interior sets should be removed. Since subtrees are being shared in the edge-ordering trees, we keep a reference count in each node in a balanced tree indicating how many pointers have been set to point at it. When a node in an edge-ordering tree is removed, the reference count in each of its two children should be decremented. If a reference count goes to zero, then its node should be deleted. (In the case that we wish to enforce a particular bound on the time per operation, if some operation would cause a very large number of nodes to have their reference counts go to zero, then these nodes are saved in a list that can be reduced in size of the subsequent operations.) Thus edge-ordering trees will be removed as nodes are removed from the topology tree. In rebuilding the topology tree, whenever a parent for two nodes is created or changed, the boundary, pseudoboundary, and newly interior sets are recomputed in the fashion discussed in algorithm *build_sets*.

This approach is related to that in [F1], but we are specifying it carefully since we believe there is an error in [F1] with regard to the analysis of the running time. In particular, we believe that the time to search for the correct point to split boundary sets is underestimated in [F1]. The reason is the following. Consider a cluster V_j created by the union of two clusters $V_{j'}$ and $V_{j''}$, each of tree degree 2. We wish to perform a simultaneous search in edge-ordering trees representing a boundary (or pseudoboundary) set for $V_{j'}$ and a boundary (or pseudoboundary) set for $V_{j''}$ to identify the edges common to both sets. It is easy to produce in constant time a suitable edge in one boundary set to test. The problem is determining whether that edge is also in the other boundary set. It is easy to keep a pointer from the leaf of one edge-ordering tree to the leaf in another edge-ordering tree representing the same edge but with respect to its other endpoint. However, it does not seem possible to deduce the name of the corresponding boundary set in constant time unless an excessive amount of work is performed on each update.

Since subtrees of the edge-ordering trees are shared, we give a top-down procedure to search within $V_{j'}$ and $V_{j''}$ simultaneously. To make the above searches efficient, we keep in each node of every edge-ordering tree the number of edges represented by the subtree rooted at that node. Then the position of the edges to be deleted can be computed quickly by keeping track of the positions of edges already deleted from the boundary sets. We then binary search to find the number of shared edges. For any test value, we search down through both trees to find the corresponding leaf in each tree. If the pointers in the leaves point at each other, then there are at least that many common edges; otherwise, there are fewer. Clearly, such a search will also involve $O(\log n)$ tests at $O(\log n)$ time per test, or $O((\log n)^2)$ time in total. Call the above procedure *plane_swap*.

LEMMA 5.1. *Procedure plane_swap correctly rebuilds an edge-ordered topology tree after a swap.*

Proof. Lemma 3.1 establishes that the topology tree is rebuilt correctly after a swap is performed. The simultaneous search of two boundary (or pseudoboundary) sets determines for any given size whether there is a common subset of that size, and it thus finds the size of the subset by binary search. Note that the search is indeed top-down, so that it works when subtrees of the edge-ordering trees are shared. Finally, the recomputing of the boundary, pseudoboundary, and newly interior sets is correct by arguments similar to those in the proof of Lemma 4.1. □

LEMMA 5.2. *The edge-ordered topology tree for an n-vertex embedded planar graph of maximum degree 3 uses $O(n)$ space, can be set up in $O(n)$ time, and can be updated to show the result of a swap in $O((\log n)^3)$ time.*

Proof. The topology tree itself uses $O(n)$ space. Each nontree edge will appear in two boundary sets (one for each endpoint) at the lowest level in the partition. Thus edge-ordering trees at level 0 use $O(n)$ space. We count the additional space used by the edge-ordering trees as follows. It follows from Lemma 2.2 that at level i, $i = 0, 1, \ldots, q$, there are at most $(5/6)^i n$ clusters. For a cluster V_j of size n_j, there are at most $c \log(2n_j)$ new nodes created in building additional boundary trees, where c is a constant. The sum of $c \log(2n_j)$ over all clusters V_j at level i is maximized when there are as many clusters as possible and each cluster is of roughly equal size. Thus we bound the total additional space used by edge-ordering trees by $\sum_{i=0}^{q}(5/6)^i n(1 + i \log(6/5))$. This quantity is clearly $O(n)$. It follows that the total space is $O(n)$.

We next discuss the setup time. Let V_j be a cluster of size n_j, with V_j being the union of clusters $V_{j'}$ and $V_{j''}$. If $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2, then charge the work of identifying each separating set to the parent W of the pair of clusters W' and W'' for which the separating set arose. This amounts to a charge of the cost of one search in the pseudoboundary sets of W, which is proportional to the square of the logarithm of the size of W. The time to remove the separating sets from the affected clusters should be apportioned similarly and will be of cost proportional to the logarithm of the cluster size. Then the charge to generate the edge-ordering tree for cluster V_j will be at most $c(\log(2n_j))^2$, where c is a constant. The sum of $c(\log(2n_j))^2$ over all clusters V_j at level i is within a constant multiplicative factor of maximum when there are as many clusters as possible and each cluster is of roughly equal size. Thus we bound the setup time by $\sum_{i=0}^{q}(5/6)^i n(1 + i\log(6/5))^2$. This quantity is clearly $O(n)$.

By Lemma 3.1, a topology tree can be updated in $O(\log n)$ time to show the result of a swap, and thus $O(\log n)$ nodes are affected. As in setting up the edge-ordered topology tree, first find the pseudoboundary sets, then identify separating edges at higher nodes, and then correct the pseudoboundary sets to be boundary sets. As in the analysis of the setup time, charge the time to identify the separating edges to the lowest cluster to which both endpoints belong. Since there are $O(\log n)$ nodes that will be created, there will be $O(\log n)$ separating sets created, at most 1 per node created. Since each such node gets charged $O((\log n)^2)$, the total charge is $O((\log n)^3)$. In addition, there are a constant number of concatenations or splits per node, and there are at most $O(\log n)$ concatenations and splits that must be performed. Each such concatenation or split uses at most one search, at $O((\log n)^2)$ per search. ∎

The edge-ordered topology tree for the embedded planar graph can also be updated to reflect the insertion or deletion of an edge, as long as the insertion is consistent with the current embedding. The approach is similar to what is described in the discussion preceding Lemma 3.2, except that there is no need to adjust the value of z since $z = 1$ is independent of the number of vertices in the graph.

LEMMA 5.3. *The edge-ordered topology tree for an embedded planar graph of maximum degree* 3 *can be updated to show the result of an edge or vertex insertion or deletion that is consistent with the embedding in* $O((\log n)^3)$ *time, where* n *is the current number of vertices.*

Proof. The number of inserted and deleted vertices and edges will be a small constant. Thus by reasoning similar to that in the proof of Lemma 3.1, the number of nodes in the topology tree that are changed will be $O(\log n)$. The time bound then follows by the same argument as in Lemma 5.2. ∎

As shown in Lemma 5.2, an edge-ordered topology tree can be updated to show the result of a swap in $O((\log n)^3)$ time. It would at first appear easy to modify this representation in the same fashion as we did to the 2-dimensional topology tree in section 2 to give a persistent data structure. The difficulty is that on each update in the persistent structure, we made an unshared copy of a topology tree, at a cost of $O(m^{1/2})$ time. This was done because there appears to be no good way to both share subtrees and still have a pointer from each node to its parent. If we wish to achieve $O((\log n)^3)$ time per update in a persistent structure for planar graphs, we must do it without making complete copies of objects such as topology trees. Instead, we shall present a scheme for encoding new *internal* names of vertices. These names will be generated bottom-up and read top-down.

The names will be based on the structure of the topology tree and will thus change as the topology of the tree it is representing changes. We first note that the topology tree is a binary tree of height $O(\log n)$. We shall assume that any child of a node with exactly one child will be designated as a left child. We shall also assume that a node with two children will have the children designated as left or right in an arbitrary but fixed fashion. We encode the *level-l internal index* of a vertex, for $l = 0, 1, \ldots, q$, as follows. Let $a_l(v)$ be the ancestor at level l of the leaf in the topology tree that represents the vertex v. The level-0 internal index of any vertex v is the empty string.

For any l, $0 < l \leq q$, the level-l internal index of v is formed as follows. Take the level-$l-1$ internal index of v and concatenate onto its left a 0 if $a_{l-1}(v)$ is a left child of $a_l(v)$ and a 1 otherwise. To differentiate the original names from these new names, the original index of a vertex or vertex cluster will be called its *external index*.

Consider the graph in Fig. 1 along with the restricted multilevel partition of Fig. 3. We give the internal indices of vertex 13, using the topology tree as shown in Fig. 4. The level-l indices of vertex 13, for $l = 1, 2, 3, 4, 5$, respectively, are 1, 01, 001, 1001, and 11001.

Each node V_j at level l in the topology tree will have its (at most two) boundary vertices specified by both external index and level-l internal index and the (at most three) tree edges with precisely one endpoint in the cluster specified by external indices of the endpoints. In addition, each leaf in the topology tree will have the external name of its only vertex. If node V_j has two children, then it will have the level-l internal index for each endpoint of the tree edge that connects the two clusters corresponding to the children.

To represent the internal names of the endpoints of edges referred to in an edge-ordering tree, an additional mechanism is needed. Each value in an edge-ordering tree associated with node V_j will have the endpoints of its corresponding edge or pair of edges specified by a level-l' internal index for some $l' \leq l$. There will be an additional field *substr* in each node of the edge-ordering tree. The concatenation of the *substr* fields on a path from the root down to any node in the edge-ordering tree, when concatenated with a level-l' index there, will give a level-l internal index for the corresponding vertex. When two clusters are unioned, the *substr* fields at the roots of the corresponding edge-ordering trees are appended with either a 0 or 1 before the trees are concatenated. Clearly, these fields can be maintained as the trees are split and concatenated. Indeed, we note that the edge-ordering trees will be balanced naturally if every time we concatenate, we just add a new root above the current two (or three) roots, rather than performing some complicated rebalancing. This follows since there are only $O(\log n)$ levels in the topology tree.

We next discuss updating a persistent structure when it has been determined that a swap should be performed. We assume that the external and internal indices of the endpoints of the edges in the swap pair (e, f) will be provided. Using the internal indices of these endpoints, we can search down in the topology tree to the leaves, using constant time per level and setting temporary pointers as follows. For each node v on a path from the root down to an endpoint of e or f, set the parent pointer of v, the pointers between v and the nodes representing clusters adjacent to the cluster for v, and the parent pointer for each node representing an adjacent cluster.

Then proceed as in algorithm *plane_swap*, but doing the necessary work to maintain the following invariant with respect to nodes on the lists L_D, L_A, and L_C:

1. For the ancestor of any node in the lists L_D and L_C, there are temporary pointers to its parent.
2. For the ancestor of any node in the lists L_D, L_A, and L_C, there are temporary pointers to nodes representing adjacent clusters.
3. For any node representing a cluster adjacent to an ancestor of any node in the lists L_D, L_A, and L_C, there is a temporary parent pointer.

Whenever we insert a node y onto a list, where y was not previously an ancestor of a member of some list, we do the following. Either node y is put on list L_A and thus has no parent, or it is put on list L_C and it or a child of it must have been adjacent to a node on a list. In the latter case, the temporary pointer to the parent of y is already set, as well as temporary parent pointers for all of its ancestors. We also know the external names of the endpoints of tree edges with precisely one endpoint in the cluster represented by node y. If such a tree edge does not have a temporary pointer associated with it, search up through the ancestors of y until we find an ancestor node x such that that edge is internal to x. We then use the internal indices for the endpoint of that edge to search down to a node z that represents a cluster that is adjacent to the cluster represented by node y. As the search goes back down, set temporary parent pointers along this path, as well as pointers from each ancestor of y to the node representing an adjacent cluster on the path up from z. When all such tree edges from the cluster represented by node y have been handled, the invariant is once again satisfied. We call this search from y an *invariant-enforcing search.*

Let the above approach be called algorithm *plane_persist_swap.*

THEOREM 5.4. *Algorithm plane_persist_swap maintains a fully persistent version of edge-ordered topology trees, using $O(n + k(\log n)^2)$ space to store k versions and generating a new version reflecting the result of a swap in $O((\log n)^3)$ time.*

Proof. By Lemma 5.1, algorithm *plane_swap* correctly updates an edge-ordered topology tree when a swap is performed. We verify that *plane_persist_swap* maintains the invariant stated above. The proof is by induction on the number of nodes that have been placed on L_D, L_A, and L_C. Initially, four nodes are placed on L_D and four nodes are placed on L_A. The nodes on L_D are the endpoints of edges e and f, and the ancestors of these nodes constitute the search paths along which temporary pointers are set. The four nodes on L_A are the replacements for the nodes on L_D, and copying the adjacency information of those nodes ensures that the ancestors of all adjacent nodes have the appropriate temporary pointers set. Subsequently, the following cases describe nodes placed on lists. These cases result from a close examination of *basic_swap.* For a node to be placed on L_D, a child of it must have been on L_D or L_C, or a child of it must have been adjacent to a node on L_C or L_A. The invariant is satisfied in all but the latter case, in which the invariant-enforcing search restores the invariant. For a node to be placed on L_C, either it had a sibling on L_D, or it had a child on L_C or L_A, or a child of it must have been adjacent to a node on L_C or L_A. The invariant is satisfied in all but the latter case, in which the invariant-enforcing search restores the invariant. For a node to be placed on L_A, either it or an adjacent node is on L_C, or a child is on L_C or L_A. The invariant-enforcing search restores the invariant in the former case. Thus the invariant is maintained. Given that the invariant is maintained, the algorithm is then able to access nodes to test whether or not to combine clusters.

Furthermore, we assert that for any quantity in an edge-ordering tree that is associated with an endpoint expressed by a level-l internal index, its level-l internal index can be maintained and manipulated as the edge-ordering trees are split and concatenated.

Finally, we establish the claimed resource bounds. From Lemma 5.1, the time used by *plane_swap* is $O((\log n)^3)$. In setting up temporary parent and adjacency pointers, the number of additional nodes examined is just a constant times the number of nodes examined in *plane_swap.* Furthermore, each node that is examined is examined just a constant number of times. Thus setting up temporary parent and adjacency pointers will take $O((\log n)^3)$ time. Each operation using level-l internal indices will take just constant time, so that the time to split and concatenate edge-ordering trees will be proportional to what it was in *plane_swap.* Thus the total time for one swap in a fully persistent version of edge-ordered topology trees will be $O((\log n)^3)$ time. By

Lemma 3.1, a topology tree can be updated in $O(\log n)$ time to show the result of a swap, and thus $O(\log n)$ nodes in the topology tree are affected. Since a constant number of boundary sets, newly interior sets, and separating sets are created for each node, and each split or concatenation of edge-ordering trees will create $O(\log n)$ nodes, $O((\log n)^2)$ additional space is used whenever a swap is performed. ∎

6. Basic approach for finding the k smallest spanning trees.

In this section, we discuss the overall structure of our algorithm for finding the k smallest spanning trees, leaving out the description of the particular data structure that we employ. We shall assume that there are at least k distinct spanning trees of the graph. (It is easy to modify the algorithm to detect the case in which there are fewer than k distinct spanning trees.) We shall also assume that all edge weights are nonnegative. (If not, we can add a positive value to each edge weight to give an equivalent problem with all edge weights nonnegative.)

We first find a minimum spanning tree of our graph using the fast algorithm of [GGST] for general graphs or [CT] for planar graphs. Then we use Eppstein's technique to reduce the problem to one in which there are $O(k)$ vertices and edges [E]. If $k < m - n$, this technique identifies and deletes $m - n - k$ edges that will be in none of the k smallest spanning trees, and if $k < n$, it identifies and contracts $n - k$ edges that will be in all of these trees. Identifying these edges uses an algorithm for the sensitivity analysis of minimum spanning trees, either Tarjan's algorithm [T1], [T2] for general graphs or the algorithm of Booth and Westbrook [BW] for planar graphs. Also used is the linear-time selection algorithm [BFPRT]. We call the resulting graph the *contracted graph*. Note that the k smallest spanning trees of the contracted graph are in one-to-one correspondence with the k smallest spanning trees of our original graph.

Next, we transform the contracted graph into a graph in which every vertex has degree no greater than 3 using the transformation discussed in section 2. Note that each edge of cost $-\infty$ will be in all of the k smallest spanning trees, and each edge of cost ∞ will be in none. It follows that the k smallest spanning trees of the transformed graph are in one-to-one correspondence with the k smallest spanning trees of the contracted graph.

Let T_i denote the ith smallest spanning tree of the transformed graph. Thus T_1 denotes the minimum spanning tree. Having already found T_1, our algorithm will generate the $k - 1$ spanning trees $T_2, \ldots, T_k$ one at a time. Each tree T_i with $i > 1$ will be derived from some tree T_j, $j < i$, by a swap (e_i, f_i), in which a tree edge e_i is replaced by a nontree edge f_i. To guarantee that no tree is derived more than once, the trees will have certain restrictions placed on them of the form that any tree derived from T_j must include certain edges and exclude certain other edges. This inclusion–exclusion approach was presented by Lawler in [L1] and [L2, pp. 100–104].

Associated with each spanning tree T_i that is generated will be a *best-swap structure* R_i. We shall discuss the best-swap structure in greater detail later but mention a few properties now. Structure R_i will represent all spanning trees derivable from T_i by a sequence of swaps and will identify a swap for T_i of minimum cost. The algorithm will maintain a heap on the costs of the trees obtainable via these minimum-cost swaps. (When k is very large, our final version of the algorithm will manage the heap somewhat differently; see the discussion at the end of this section.)

We now proceed with a description of the rest of the algorithm. Given the minimum spanning tree T_1, we generate a best-swap structure for T_1. We initialize the heap with the value representing the cost of the spanning tree derived from T_1 by applying the swap of minimum cost. We then repeat the following $k-1$ times. Extract the minimum from the heap. The extracted value represents the cost of a tree T_i produced by applying a swap (e_i, f_i) to spanning tree T_j. Generate a best-swap structure R_i from R_j using the fully persistent versions of the data structures discussed in sections 3 and 5. The changes in generating R_i from R_j should reflect the effect of two changes: replacing e_i by f_i in the spanning tree and resetting the cost of edge e_i to be the value ∞ for the purpose of determining the best swap. Resetting the cost of edge e_i effectively keeps edge e_i out of any of the spanning trees that are subsequently derived (transitively) from T_i. Finally, modify R_j to reflect the resetting of the cost of e_i to be $-\infty$ for the purpose of determining the best swap. Resetting the cost of edge e_i in this manner effectively forces edge e_i to be in all of the spanning trees that are subsequently derived (transitively) from T_j. The minimum costs identified by each of R_i and R_j correspond to swaps to be applied to T_i and T_j, respectively. Compute the costs of the trees generated by these trees and insert them into the heap. Note that the original costs of edges should be used in computing these costs. This completes the description of the repeat loop.

As we have described the algorithm, its output will be in the form of a minimum spanning tree plus a sequence of triples (e_i, f_i, j_i), $i = 2, 3, \ldots, k$. Note that it is easy to include the cost of tree T_i with the triple.

We can visualize the inclusion–exclusion using a binary tree B. Each node x in B represents a modified version $G(x)$ of the original graph G based on the inclusion and exclusion conditions. Associated with each node is the minimum spanning tree $T(x)$ for $G(x)$, along with a value that is the cost of $T(x)$ with respect to the edge weights in G. The root of B represents G, $T(root)$ is the minimum spanning tree T_1 of G, and the value associated with the root is the cost of T_1. For any node x in B, we determine the children of x as follows. If there is a swap of finite cost that can be applied to $T(x)$, let $(e(x), f(x))$ be the minimum-cost such swap. Then x will have right and left children. Graph $G(right(x))$ will be graph $G(x)$ with the cost of $e(x)$ reset to ∞, and spanning tree $T(right(x))$ will be $T(x) - e(x) + f(x)$. Graph $G(left(x))$ will be graph $G(x)$ with the cost of $e(x)$ reset to $-\infty$, and spanning tree $T(left(x))$ will be $T(x)$. This completes the definition of binary tree B.

As an example, we consider the spanning trees for the graph in Fig. 1. We shall name edges by their weights. In Fig. 9, we give binary tree B for this graph. The minimum spanning tree, as shown in Fig. 1, has a cost of 91 and is represented by the root of the tree in Fig. 9. The best swap for this tree is $(13, 14)$. The right child of the root represents the resulting tree, with cost 92. Note that edge 13 is excluded from being a member of any of the spanning trees represented by this node or any of its descendants. Conversely, edge 13 is required to be included in any tree represented by the left child of the root or any of its descendants. The minimum-cost swap given that edge 13 must be included is $(12, 14)$, yielding a tree with cost of 93. In our representation, the edge to the right child is labeled with a tree edge that is excluded, and the edge to the left child is labeled with a tree edge (the same edge) that must be included. To make the representation less cluttered, we subsequently put the included/excluded tree edge between the edges to the right and left children. Note that we label a nonroot node with its cost only if its spanning tree differs from that of its parent. Also, we do not draw the complete representation but only the first four levels, noting that all nodes shown have children except the lower rightmost one.

The time required by the algorithm will be the following. From [GGST], [E], [T1],

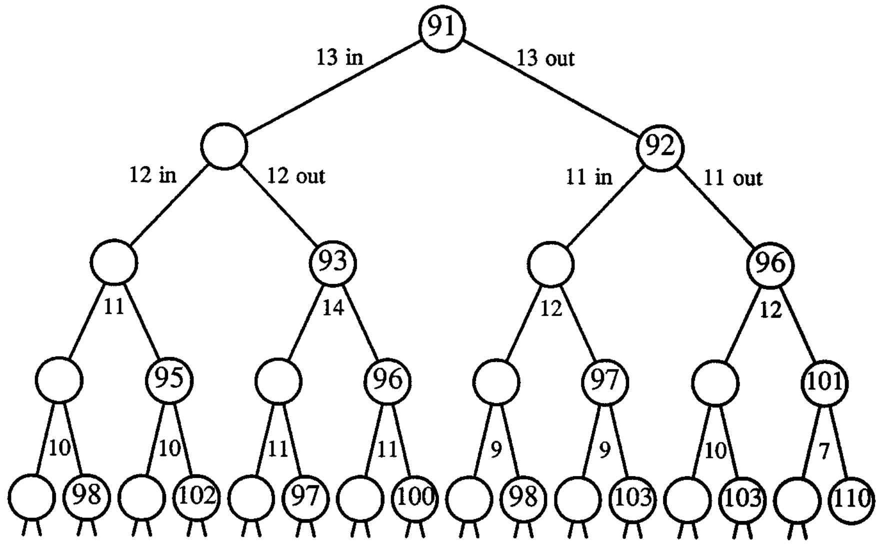

FIG. 9. *The first four levels of inclusion/exclusion for Fig.* 1*, with spanning tree costs indicated.*

[T2], [BFPRT], and [F1], finding the contracted graph and transforming it into one with maximum degree 3 will take $O(m \log \beta(m, n))$ time and $O(m)$ space. From [CT], [E], [BW], [BFPRT], and [F1], finding the contracted graph of a planar graph and transforming it into one with maximum degree 3 will take $O(n)$ time and space. In addition to setting up R_1, the algorithm will perform $2(k-1)$ updates of best-swap structures. With regard to the heap, $k-1$ *extractmin*s and $2(k-1)$ *insert*s will be performed. Thus the total time for all heap operations is $O(k \log k)$.

For very large values of k, the total time for maintaining the heap on the costs of trees may dominate the total time for updating the best-swap structures. In such a case, we may reduce the $O(k \log k)$ charge for maintaining the heap to $O(k)$ as follows. Note that when a best-swap structure is modified, the cost of the new spanning tree induced by the new best swap is never smaller than the spanning tree from which it was derived.

Suppose that these costs can be viewed as forming a min-heap. From [F4] it is known that the kth smallest value in a min-heap can be selected in $O(k)$ time. This algorithm is then used in place of the simple heap mechanism. Given $O(k)$ values that include the costs of all k smallest spanning trees, it is then straightforward to identify the costs of the k smallest spanning trees. Note, however, that these costs will not necessarily be output in sorted order.

It remains to show that the costs in binary tree B can be viewed as forming a min-heap. Since the spanning tree for each left child is the same as that of its parent, we need to compress B to get our min-heap. Note that for any node x such that $right(left(x))$ is defined, the value labeling $right(x)$ is no larger than the value labeling $right(left(x))$. This follows since a swap of smallest cost relative to $T(x)$ in $G(x)$ is of cost no larger than a swap of smallest cost relative to $T(left(x))$ in $G(left(x))$. Thus we generate our min-heap to contain nodes that correspond to a subset of the nodes in B in the following way. The root of the min-heap corresponds to the root of B. For any node y in the min-heap corresponding to node x in B, we determine the children of y as follows. If $right(x)$ is defined, then $right(y)$ is defined

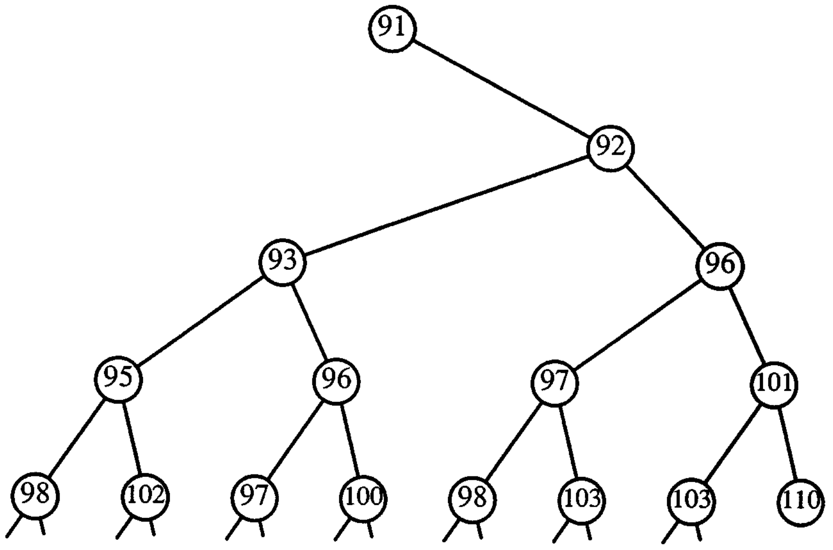

Fig. 10. *The min-heap induced by inclusion/exclusion on Fig.* 1.

to be a node that corresponds to $right(x)$. If node x has a parent, $parent(x)$, and if $right(left(parent(x)))$ is defined, then $left(y)$ is defined to be a node that corresponds to $right(left(parent(x)))$. Since the algorithm in [F4] first accesses an element in the min-heap only after having accessed its parent, the portion of the min-heap actually accessed by that algorithm can be constructed on the fly as we create and access our replacement data structures. Thus only $O(k)$ nodes in the min-heap need to be created. The binary min-heap corresponding to binary tree B in Fig. 9 is shown in Fig. 10.

7. Ambivalent data structures I: Best-swap structures.

In this section, we adapt the data structures from section 3 to give an efficient best-swap structure for the case of general graphs. This will lead to an efficient algorithm for finding the k smallest spanning trees of a graph. We first give a more formal definition of an ambivalent data structure. Then we define what we call a "pseudoswap," which will allow us to design an ambivalent data structure. Using pseudoswaps, we next describe the information maintained in the nodes of the 2-dimensional topology tree and show how to generate this information for a node, given the information for its children. We then specify the best-swap data structure and discuss how to update this structure. We conclude with a claim of the time and space bounds on our algorithm for finding the k smallest spanning trees.

We now give a more formal definition of an ambivalent data structure. An item or set of items of data is said to be *substantiated* if that item or set of items represents an actual state of affairs. If that item or set of items does not represent an actual state of affairs but rather a hypothetical state of affairs that does not actually hold, then it is said to be *nonsubstantiated*. Let an item or set of items be called an *alternative* if examination of that item or set of items in isolation cannot determine whether it is substantiated or nonsubstantiated, but examination of a larger context of data will determine whether it is substantiated or nonsubstantiated. A *substantiated set of alternatives* is a set of two or more alternatives, one of which will be substantiated and the rest of which will be nonsubstantiated. A data structure is said to be *ambivalent* if at many locations within itself it maintains substantiated sets of alternatives, and the data structure as a whole contains sufficient data to determine which alternative in every substantiated set of alternatives is substantiated. The ambivalence in our particular data structures comes from considering two clusters of vertices in a spanning tree and then attempting to represent how a nontree edge with one endpoint in each cluster relates to tree edges in these clusters. Such a relation might be whether a tree edge is in the cycle induced by the nontree edge. Determining whether a tree edge is in the cycle induced by the nontree edge may require information not stored with either cluster, i.e., it is information about the topology of the spanning tree. However, this information will always be available in our data structure as a whole.

We seek to build a data structure in which we can maintain a large set of swaps in a heap-like fashion so that a best swap can be identified quickly. We do this by considering nontree edges that have both endpoints in the same cluster and nontree edges that have their endpoints in different clusters. It is not hard to compute the most advantageous swap involving edges both of whose endpoints are in the same basic cluster. Thus the more challenging task is handling nontree edges that have their endpoints in different clusters. We set up ambivalent information for each cluster V_j of tree degree 2 as follows. Let V_r be a cluster at the same level as V_j. For any boundary vertex w of V_j, a *pseudoswap* is a pair (e, f) of edges, where f is a nontree edge having one endpoint in each of V_j and V_r and e is an edge on the path in the tree from w to the endpoint of f in V_j. Let w and w' be the boundary vertices of V_j. Suppose (e, f) is a pseudoswap for w and (e', f) is a pseudoswap for w'. Then one of those pseudoswaps is actually a swap, depending on whether w or w' is nearer V_r in tree T. Thus the set of pseudoswaps $\{(e, f), (e', f)\}$ is a substantiated set of alternatives.

Consider clusters V_j and V_r shown in Fig. 11. The spanning tree edges within these clusters are shown in bold, and the only nontree edge with an endpoint in each cluster is indicated by a dashed line. Some of the edges are labeled with their costs. Since the whole spanning tree is not shown, it is not clear whether the path in the tree between the endpoints of edge 18 goes through vertex w or vertex w'. Pseudoswap $(16, 18)$ is the best for clusters V_j and V_r and boundary vertex w. Pseudoswap $(17, 18)$ is the best for clusters V_j and V_r and boundary vertex w'.

We next discuss carefully the additional information that will be maintained in the nodes of the 2-dimensional topology tree. This includes the cost of a maximum-weight edge on the path between certain pairs of boundary vertices, the cost of a minimum-cost nontree edge between given pairs of clusters, the cost of a minimum-cost pseudoswap from a certain class of pseudoswaps, and the cost of a minimum-cost swap from a certain class of swaps.

We first discuss the cost of a maximum-weight edge on the path between certain pairs of boundary vertices. Let V_j be a vertex cluster with tree degree 2, and let $treemax(j)$ be the cost of a tree edge of maximum weight on the path between the two boundary vertices. We store $treemax(j)$ for a given j in node $V_j \times V_j$ in the 2-dimensional topology tree. If V_j is a basic vertex cluster, then $treemax(j)$ can be determined by inspection of V_j. If V_j is not a basic vertex cluster, then $treemax(j)$ can be computed in constant time given the *treemax* values of the children in the topology tree and the cost of the tree edge between the children's corresponding clusters. Note that it is easy to keep track of the edge that yields the $treemax(j)$ value.

We next discuss the cost of a minimum-cost nontree edge between a given pair of

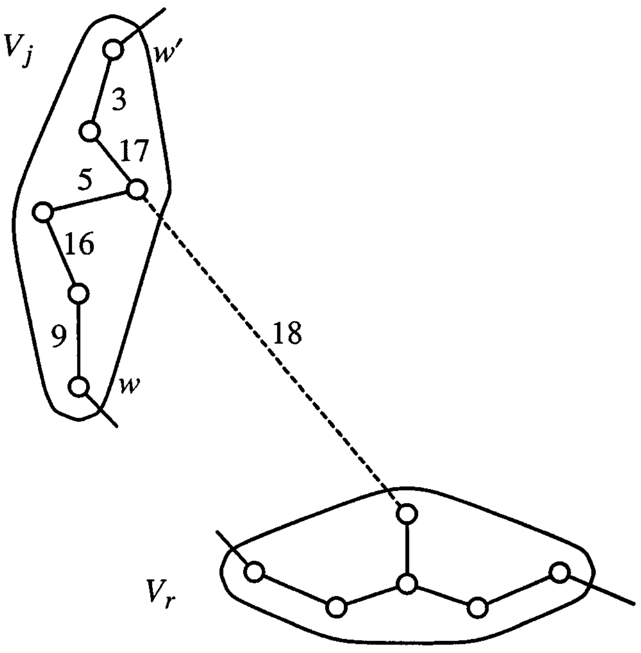

FIG. 11. *Example for illustrating best pseudoswaps.*

clusters. Let V_j and V_r be two distinct clusters at the same level. Let $nontreemin(j,r)$ be the cost of a nontree edge of minimum cost with an endpoint in each of V_j and V_r. Store $nontreemin(j,r)$ at node $V_j \times V_r$. If V_j and V_r are basic vertex clusters, then $nontreemin(j,r)$ can be computed for any particular j and all r by inspection of V_j. If V_j and V_r are not basic vertex clusters, then $nontreemin(j,r)$ is the minimum of the values $nontreemin(j',r')$, where $V_{j'} \times V_{r'}$ is a child of $V_j \times V_r$ in the 2-dimensional topology tree. Note that it is easy to keep track of the edge that yields each $nontreemin(j,r)$ value.

We next discuss the cost of a minimum-cost pseudoswap from a certain class of pseudoswaps. Let V_j and V_r be two distinct clusters at the same level. For each boundary vertex w of V_j, let $pswapmin(j,r,w)$ be the minimum value from the set of differences consisting of the cost of a nontree edge f with an endpoint in each of V_j and V_r, minus the cost of an edge e of maximum cost on the path in T from w to the endpoint of edge f that is in V_j. The values $pswapmin(j,r,w)$ for any particular value of j and r are stored in the node labeled $V_j \times V_r$. If V_j and V_r are basic vertex clusters, then $pswapmin(j,r,w)$ can be computed for any particular j, all boundary vertices w of V_j, and all r by inspection of V_j.

If V_j and V_r are not basic vertex clusters, then $pswapmin(j,r,w)$ can be computed in constant time given the *treemax* values for all children of V_j, and the *nontreemin* and *pswapmin* values for all children of $V_j \times V_r$ in the 2-dimensional topology tree. We specify this computation in detail. If the node for V_j in the topology tree has a single child $V_{j'}$, then $pswapmin(j,r,w)$ is the minimum of $pswapmin(j',r',w)$ taken over the one or two clusters $V_{r'}$ that form V_r. Otherwise, V_j is formed from two clusters $V_{j'}$ and $V_{j''}$, and we assume without loss of generality that w is contained in $V_{j'}$. Let w' be the boundary vertex of $V_{j'}$ adjacent to $V_{j''}$, and let w'' be the boundary vertex of $V_{j''}$ adjacent to $V_{j'}$. Then $pswapmin(j,r,w)$ is the minimum taken over the one or two clusters $V_{r'}$ that form V_r of $pswapmin(j', r', w)$, $pswapmin(j'', r', w'')$, $nontreemin(j'', r') - c(w', w'')$, and $nontreemin(j'', r') - treemax(j')$. Once again, it is easy to keep track of the pair of edges that yield each $pswapmin(j, r, w)$ value.

We finally discuss the cost of a minimum-cost swap from a certain class of swaps Let V_j be a vertex cluster. Let $swapmin(j)$ be the cost of the minimum-cost swap such that the nontree edge has both endpoints in V_j. This value can be maintained in node $V_j \times V_j$ of the 2-dimensional topology tree. If V_j is a basic vertex cluster, then $swapmin(j)$ can be computed by inspection of V_j. If V_j is not a basic vertex cluster, then $swapmin(j)$ can be computed in constant time given the $swapmin$, $nontreemin$, and $pswapmin$ values for children of $V_j \times V_j$ in the 2-dimensional topology tree. We specify this computation in detail. If the node for V_j in the topology tree has a single child $V_{j'}$, then $swapmin(j) = swapmin(j')$. Otherwise, V_j is formed from two clusters $V_{j'}$ and $V_{j''}$. Let w' be the boundary vertex of $V_{j'}$ adjacent to $V_{j''}$, and let w'' be the boundary vertex of $V_{j''}$ adjacent to $V_{j'}$. Then $swapmin(j)$ is the minimum of $pswapmin(j', j'', w')$, $pswapmin(j'', j', w'')$, and $nontreemin(j', j'') - c(w', w'')$. Once again, it is easy to keep track of the pair of edges that yield each $swapmin(j)$ value.

Our best-swap structure will be based on the fully persistent data structure described in section 3. A best-swap structure R_i will consist of a topology tree for tree T_i, a pointer to a 2-dimensional topology tree, many of whose subtrees are shared with other 2-dimensional topology trees, and pointers to representations of basic vertex clusters, most of which are shared. Each node in the topology tree will have the index of the corresponding cluster and a pointer to the node's parent. Each node in the 2-dimensional topology tree will have the indices of the corresponding clusters, along with the following. If the node is of type $V_j \times V_j$, it will have a $treemax$ value and a $swapmin$ value, while if it is of type $V_j \times V_r$, for $r \neq j$, it will have a $nontreemin$ value and $pswapmin$ values. Note that each such $treemax$, $nontreemin$, $pswapmin$, and $swapmin$ value should also carry with it the index of the edge or edges involved and the indices of its basic vertex clusters. The representation of a basic vertex cluster V_j will consist of a list of vertices, a list of edges with both endpoints in the cluster (both tree and nontree edges), and, for every other basic cluster V_r, a pointer to a list of nontree edges with one endpoint in each of V_j and V_r. In addition, for each nontree edge with both endpoints in the same basic cluster, there will be the largest-cost tree edge with which it can swap.

We now discuss how to update a best-swap structure. If the swap causes a basic vertex cluster to be split or combined, generate a description of each new basic cluster V_j consisting of a list of vertices, a list of edges with both endpoints in V_j, and, for every other basic cluster V_r, a list of nontree edges with one endpoint in each of V_j and V_r. If the new basic cluster V_j is merged from old basic clusters V_j' and V_j'', determine the best swap for each nontree edge with one endpoint in each of V_j' and V_j'' as follows. Let tree edge (w', w'') connect V_j' to V_j'' with w' in V_j' and w'' in V_j''. Find the maximum-cost edge from each vertex in V_j' to w' and from each vertex in V_j'' to w''. Given nontree edge (v', v'') with v' in V_j' and v'' in V_j'', the best tree edge that can swap with (v', v'') can then be found in constant time. A similar approach can be used if a tree edge (w', w'') with both endpoints in basic cluster V_j has its cost set to $-\infty$ to find the new swaps that replace those involving edge (w', w'').

We next generate the new topology tree and the new 2-dimensional topology tree. For each basic vertex cluster V_j that has changed, do the following. For each pair of boundary vertices w and w' in V_j, determine the value $treemax(j, w, w')$. Next, determine the value $swapmin(j)$ by finding the minimum-cost swap over all best swaps for nontree edges with both endpoints in V_j. For each set of nontree edges with one endpoint in each of V_j and V_r, set the appropriate pointers in the descriptions of V_j and V_r. Also, find the minimum-cost edge in the set, giving the *nontreemin*(j, r) value. For each vertex v in V_j and each boundary vertex w of V_j, determine the maximum-cost tree edge on the path from v to w. For every other basic cluster V_r, examine every edge with one endpoint in each of V_j and V_r to find the best pseudo-swap for each boundary vertex w of V_j. Thus we can determine the *pswapmin*(j, r, w) values. Create a new copy of the topology tree, and then modify the structure of the new topology tree and the 2-dimensional topology tree. As selected portions of the 2-dimensional topology tree are being rebuilt bottom-up, modify the information in the *treemax*, *nontreemin*, *pswapmin*, and *swapmin* fields.

THEOREM 7.1. *Let G be a graph with m edges and n vertices for which we know the minimum spanning tree T_1 and the best swap for each nontree edge. The best-swap structure R_1 can be set up in $O(m)$ time and space. A best-swap structure R_i can be updated in $O(m^{1/2})$ time, using $O(m^{1/2})$ additional space.*

Proof. The above algorithm indicates how to update basic clusters as a result of a swap or resetting the cost of a tree edge to $-\infty$. Once basic clusters have been updated, for any basic cluster V_j that has changed, the fields associated with nodes $V_j \times V_r$ must be recomputed. Then the fields for all ancestors of such nodes must be recomputed. The above algorithm does this.

Basic vertex clusters can be found in $O(m)$ time using procedure *cluster* from section 2. Similar to that in [F1], a restricted multilevel partition, a topology tree, and a 2-dimensional topology tree can be found in $O(m)$ time. Generating all other values can be done in time proportional to the number of them.

We next discuss the resources needed to update R_i. By Theorem 3.3, the time and space to perform the structural changes to the data structure is $O(m^{1/2})$. The time to compute each value in a newly created node is constant if these values are computed bottom-up. Thus the total time to update R_i is $O(m^{1/2})$. □

THEOREM 7.2. *The k smallest spanning trees of a weighted undirected graph can be found in $O(m \log \beta(m, n) + \min\{k^{3/2}, km^{1/2}\})$ time and $O(m + \min\{k^{3/2}, km^{1/2}\})$ space.*

Proof. The algorithm used is that described in section 6, with the best-swap structure R_1 just described. The algorithm uses the algorithm for finding the kth smallest element in a min-heap, as discussed at the end of section 6. Correctness follows from the discussion in section 6, plus Theorem 7.1.

As discussed in section 6, the time to find the minimum spanning tree and also find a transformed graph with $O(\min\{k, m\})$ edges will be $O(m \log \beta(m, n))$. By the discussion in section 6, there will be $O(k)$ such updates. By the discussion at the end of section 6, the cost of the kth smallest spanning tree can be found by performing $O(k)$ updates which produce $O(k)$ values, from which one selects the kth smallest in $O(k)$ time. By Theorem 7.1, updating a best-swap structure for a graph with $O(\min\{k, m\})$ edges will take $O((\min\{k, m\})^{1/2})$ time. The time bound then follows. By Theorem 7.1, each update will introduce $O(\min\{k^{3/2}, km^{1/2}\})$ additional space. The space bound then follows. □

8. Best-swap data structures for embedded planar graphs.

In this section, we describe our ambivalent data structure to find a best swap for a spanning tree of an embedded planar graph. We first discuss how to store in fully persistent edge-ordering trees ambivalent information with respect to boundary sets, pseudoboundary sets, newly interior sets, and separating sets. We then describe how to compute the minimum-cost swap for a vertex cluster, given the appropriate information about the cluster's children. We next describe the best-swap structure and its updating. Finally, we claim the time and space bounds for our algorithm that finds the k smallest spanning trees in a planar graph.

We first describe how to maintain ambivalent information in the edge-ordering tree for each set of edges. Consider a cluster V_j of tree degree 2. Consider the path P_j between the two boundary vertices of V_j. For any vertex u in V_j, we define $proj(j, u)$, the *projection* of u onto P_j, to be the vertex on P_j that is closest to u in the tree. First, consider the left boundary set $bs_L(V_j)$. For each edge (u, v) in $bs_L(V_j)$ with u in V_j, consider $proj(j, u)$. For edges (u, v) in $bs_L(V_j)$, there may be some vertex on P_j that has several vertices u projected onto it, and there may be some vertex that has no vertices projected onto it. We consider the *left modified path* $m_L(P_j)$ in which every vertex of $m_L(P_j)$ has exactly one vertex projected onto it except for the endpoints, which have none. Between two consecutive vertices x and y of $m_L(P_j)$, we shall have an edge whose cost is the cost of a maximum-cost edge on the subpath between x and y in P_j. In the case that x and y represent the same vertex in P_j, this cost will be $-\infty$. Note that $m_L(P_j)$ should be set up so as to be consistent with the planar embedding. We define and represent the *right modified path* $m_R(P_j)$ in a similar way. It is clear that a modified path is a generalization of a boundary set and can be represented by an edge-ordering tree.

We represent information about a modified path within the edge-ordering tree as follows.

- For each leaf, we keep
  1. the cost of the edge in the boundary set,
  2. the cost of the next edge in a given direction on the modified path,
  3. the cost of the swap using this next edge,
  4. the cost of the next edge in the other direction on the modified path,
  5. the cost of the swap using that next edge.
- For each nonleaf node in the edge-ordering tree, we keep
  1. the cost of the minimum-cost edge in the portion of the boundary set represented in the subtree rooted at the nonleaf node,
  2. the maximum of the costs of next edges in the given direction on the modified path in the subtree,
  3. the cost of the best swap if all edges in the portion of the boundary set for the subtree were able to swap with their next edges in this given direction,
  4. the maximum of the costs of next edges in the other direction on the modified path in the subtree,
  5. the cost of the best swap if all edges in the portion of the boundary set for the subtree can swap with their next edges in that other direction.

Given these values for any two siblings in the edge-ordering tree, the values for the parent can be computed in constant time.

If cluster V_j has tree degree 1, then the information is represented in an especially simple form. Since we know in which direction any nontree edge goes, we consider in computing *pswapmin* the swap of any nontree edge with a tree edge in V_j. We let P_j be the trivial path (of no edges) whose endpoints are both the single boundary vertex of V_j. The endpoints of all edges in the boundary set of V_j then project onto this single point in P_j, and all edges in $m(P_j)$ will have cost $-\infty$. For a cluster V_j of tree degree 2, the information corresponding to *treemax*, *nontreemin*, and *pswapmin* which we maintained in section 7 is now held within the edge-ordering trees for the modified paths corresponding to the boundary sets, pseudoboundary sets, newly interior sets, and separating set. For a cluster V_j of tree degree 1, the same is true, except that no *treemax* is maintained.

As in section 7, we shall keep for each cluster V_j the cost $swapmin(j)$ of the best swap found within V_j. We discuss the additional changes that are necessary in the handling of edge-ordering trees when two clusters $V_{j'}$ and $V_{j''}$ are unioned to give cluster V_j. This involves examining four cases. Suppose $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3. We take $swapmin(j)$ to be the minimum of $swapmin(j')$ and $c(f)-c(e)$, where e is the tree edge between $V_{j'}$ and $V_{j''}$ and f is the edge of smallest cost in the boundary set of $V_{j'}$. In a fashion similar to that discussed before, the now modified edge-ordering tree for the boundary set of $V_{j'}$ will represent the modified edge-ordering tree for one of the two boundary sets of V_j.

Suppose $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2. We split and concatenate boundary sets as in sections 4 and 5. For any remaining portions of these sets, we now know in which direction the connection lies. For each newly interior set, we query the corresponding edge-ordering tree to get the minimum swap in the appropriate direction. (In particular, suppose $V_{j'}$ is "down" from $V_{j''}$. Then for the edge-ordering tree that represents the portion of the left boundary set of $V_{j''}$ that is newly interior, identify the minimum-cost swap in a downward direction. Do the same for the portion of the right boundary set of $V_{j''}$ that is newly interior.) Also, we query the edge-ordering tree for the remaining portion of each boundary set of $V_{j''}$ to identify the minimum-cost swap in an upward direction. We then take the minimum of the costs of these swaps, along with the values $swapmin(j')$, $swapmin(j'')$, and $c(f)-c(e)$, where e is a maximum-weight tree edge on the path between the top of $V_{j'}$ and the top of $V_{j''}$ and f is the edge of smallest cost in the boundary set of $V_{j'}$. Since V_j is of tree degree 1, we must reset the cost of tree edges in the modified path to be $-\infty$. We do this symbolically by letting the cost of the maximum-cost edge in this be set to $-\infty$. In any subsequent splits that affect this node, we propagate this value down as necessary in the edge-ordering tree. (This can be done by creating new nodes.)

When two boundary sets are concatenated together, values in the edge-ordering trees must be changed. One important change is that on each side of the modified path, we must find the maximum cost of a tree edge on the path between the nearest pair of edges, one in the boundary set of $V_{j'}$ and the other in the boundary set of $V_{j''}$, that will be in boundary sets for V_j. This can be done by taking the maximum over the tree edges in the edge-ordering trees representing newly interior edges on that side, along with the tree edge between $V_{j'}$ and $V_{j''}$.

Suppose $V_{j'}$ and $V_{j''}$ are both of tree degree 2. As discussed in sections 4 and 5, we split and concatenate pseudoboundary and boundary sets and form newly interior sets and a separating set. For each newly interior set, we query the corresponding edge-ordering tree to find the minimum swap in the appropriate direction. (Suppose $V_{j'}$ is "down" from $V_{j''}$. Then for the edge-ordering tree that represents the portion of the left boundary set of $V_{j''}$ that is newly interior, identify the minimum-cost swap in a downward direction. Do the same for the portion of the right boundary set of $V_{j''}$ that is newly interior. Then for the edge-ordering tree that represents the portion of the left boundary set of $V_{j'}$ that is newly interior, identify the minimum-cost swap in an upward direction. Do the same for the portion of the right boundary set of $V_{j''}$ that is newly interior.) For the separating set, we query the corresponding portions of the edge-ordering trees for the boundary sets of $V_{j'}$ and $V_{j''}$ to find the minimum swap in the appropriate direction. (If the edges go from the left of $V_{j''}$ to the right of $V_{j'}$, then for the edge-ordering tree that represents the portion of the separating set incident on the left of $V_{j''}$, identify the minimum-cost swap in a downward direction, and for the edge-ordering tree that represents the portion of the separating set incident on the right of $V_{j'}$, identify the minimum-cost swap in an upward direction. The mirror-image case is similarly handled.) We then take the minimum of the cost of these swaps, along with *swapmin*(j'), *swapmin*(j''), and $c(f) - c(e)$, where e is as before and f is the minimum-cost edge over all the newly interior sets and the separating set of V_j.

Finally, suppose $V_{j'}$ and $V_{j''}$ are both of tree degree 1. We take the minimum of *swapmin*(j'), *swapmin*(j''), and $c(f) - c(e)$, where e is as before and f is the minimum-cost edge over all the newly interior sets of V_j. This concludes the examination of the four cases when two vertex clusters are unioned.

We next specify the swap structure. The best-swap structure R_i will consist of a pointer to a persistent edge-ordered topology tree for tree T_i. Shared by all R_i will be descriptions of the basic vertex clusters. The description of each basic vertex cluster will be in the form of the original name of the vertex contained in it, along with each edge incident on it, specified by the original names of the endpoints and the cost. Each node V_j at level l in the topology tree will have the cost of the pair of edges realizing its *swapmin* value, along with the level-l internal indices of the endpoints of the edges. As discussed in section 5, each value in an edge-ordering tree associated with node V_j will have its corresponding edge or pair of edges specified by a level-l' internal index for some $l' \leq l$.

We discuss how to update a best-swap structure when a swap occurs. Perform the procedure *plane_persist_swap* from section 5. Note that this procedure swaps nontree edge f in for tree edge e. We also wish to reset edge e to have cost ∞, so we reset the cost of this edge before rebuilding the edge-ordered topology tree. As we rebuild these structures, we update the *swapmin* value and the values in the edge-ordering trees of the affected nodes. As in section 7, a similar but simpler approach is used to handle an update when no swap is performed but the cost of a tree edge is reset to $-\infty$.

The full algorithm for generating the k smallest spanning trees uses the algorithm in section 6 and the persistent best-swap structure R_i.

THEOREM 8.1. *The k smallest spanning trees of a weighted undirected planar graph can be found in $O(n + k(\log n)^3)$ time and $O(n + k(\log n)^2)$ space.*

Proof. The algorithm used is that described in section 6, with the best-swap structure R_1 just described. The algorithm uses the algorithm for finding the kth smallest element in a min-heap, as discussed at the end of section 6. Correctness follows from the discussion in section 6, Theorem 5.4, plus the following. We first verify that *swapmin*(j) is correctly computed when two clusters $V_{j'}$ and $V_{j''}$ are unioned to give cluster V_j. When $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3, then no nontree edges are incident on $V_{j''}$, and *swapmin*(j) results either from within $V_{j'}$ or from swapping the edge between $V_{j'}$ and $V_{j''}$ with an edge in the boundary set of $V_{j'}$. When $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2, then *swapmin*(j) results either from within $V_{j'}$ or within $V_{j''}$, or from swapping a tree edge between the top of $V_{j'}$ and the top of $V_{j''}$ with an edge contained in the boundary sets of both $V_{j'}$ and V_j, or from swapping a tree edge in $V_{j''}$ with an edge in the boundary sets of both $V_{j''}$ and V_j, or from swapping an edge in a newly interior set with a tree edge. When both $V_{j'}$ and $V_{j''}$ are of tree degree 2, then *swapmin*(j) results either from within $V_{j'}$

or within $V_{j''}$, or from swapping an edge in a newly interior set with a tree edge, or from swapping an edge in the separating set with a tree edge. When both $V_{j'}$ and $V_{j''}$ are of tree degree 1, the effect of all nontree edges in the boundary sets has already been accounted for, except for swapping with the tree edge connecting $V_{j'}$ and $V_{j''}$.

The time claim in the theorem follows from Theorem 5.4, the discussion at the end of section 6, and the following. We first consider setting up the trees for R_1. Clearly, the structure of the topology tree can be determined and set up in $O(n)$ time. The information in each leaf can be set up in constant time. Then the information in nonleaf nodes can be determined in a bottom-up fashion, at a constant cost per operation in setting up an edge-ordered topology tree as in section 5. To form the value *swapmin*(j) for a cluster V_j, a constant number of operations must be performed on edge-ordering trees. As discussed in Lemma 5.2 and Theorem 5.4, these operations will take $O((\log n)^2)$ time. The space follows from Theorem 5.4 and the observation that a constant number of nodes per level are changed in the edge-ordered topology tree for each update and that the edge-ordering trees at a node can be updated by introducing $O(\log n)$ new nodes. $\square$

Note that for values of $k < n$, $n + k(\log k)^3$ would seem to be better than $n + k(\log n)^3$. However, $k(\log n)^3 < n$ for $k < n/(\log n)^3$, and for $n/(\log n)^3 \leq k \leq n$, $\log k$ is $\Theta(\log n)$.

9. Ambivalent data structures II: 2-edge-connectivity information. In this section, we adapt the data structure from section 3 to give a data structure for updating and querying 2-edge-connectivity information in the case of general graphs. We first give two simple characterizations of 2-edge-connectivity and 2-edge-connected components. We then discuss the set of "complete paths," which are a partition of a subset of the spanning tree, and "partial paths," from which the complete paths are formed. We show how to generate these paths for a cluster when the paths of the children are known. We motivate how complete paths are used in answering a query. We next define "pseudocovering edges," which allow us to define an ambivalent data structure. We discuss the additional information stored at a node in the 2-dimensional topology tree and show how to generate it given the information of the children. We then give a summary of the update structure, including the specification of information associated with a basic cluster. We then describe how to perform queries and updates. Finally, we establish the resource bounds of our approach.

Let G be an undirected graph. Graph G is 2-*edge-connected* if there is no edge whose removal disconnects G. An edge whose removal disconnects G is called a *bridge*. The 2-*edge-connected components* of G are the subgraphs that result when all bridges are removed. We first present two propositions that characterize 2-edge-connectivity and 2-edge-connected components. Let T be a spanning tree of graph G. For each edge e in T, let *cover*$(e) = 1$ if there is a nontree edge f such that e is on the path in T between the endpoints of f, and let *cover*$(e) = 0$ otherwise.

PROPOSITION 9.1. *Graph G is 2-edge-connected if and only if cover$(e) = 1$ for each edge e in T.*

PROPOSITION 9.2. *Vertices v' and v'' are in the same 2-edge-connected component if and only if there is no edge e on the path from v' to v'' in T with cover$(e) = 0$.*

We seek to build a data structure in which we can maintain *cover* values easily. We partition the edges of the tree into two sets and maintain the *cover* information about each set differently. We use the topology tree and 2-dimensional topology tree to organize this information. Let the *boundary tree* be the set of all tree edges that are on a path in the tree between any two boundary vertices. The first set of edges consists of tree edges that are not in the boundary tree, and the second set consists of all edges that are in the boundary tree. It is relatively easy to maintain information about the first set, so we shall concentrate for the moment on the second set of tree edges.

We next define partial and complete paths. We define a partition of the edges in the boundary tree into paths, which we call *complete paths.* This partition is based on the multilevel partition. The complete paths are built up from what we call *partial paths* in a manner that we now describe. There will be a partial path associated with each cluster, and there will be a complete path associated with each cluster that is the union of two clusters of odd tree degree. For any multilevel partition with more than one level, we have the following. No basic vertex cluster will have a complete path associated with it. A basic vertex cluster of tree degree 1 will have a partial path containing the path of zero length beginning and ending at its single boundary vertex. A basic vertex cluster of tree degree 2 will have a partial path consisting of the path in the tree between its two boundary vertices. A basic vertex cluster of tree degree 3 will have the partial path consisting of no edges and the single vertex contained in the cluster.

Let PP_j and CP_j designate the partial path and complete path (if any) of vertex cluster V_j. If the node in the topology tree for vertex cluster V_j has a single child $V_{j'}$, then $PP_j = PP_{j'}$. When two vertex sets $V_{j'}$ and $V_{j''}$ are unioned to form V_j, the partial paths are handled as follows. If $V_{j'}$ is of tree degree 1 or 2 and $V_{j''}$ is of tree degree 2, then the resulting vertex cluster will have a partial path that is the concatenation of $PP_{j'}$ and $PP_{j''}$ and the tree edge between them. If $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3, then the resulting vertex cluster will have the following two paths associated with it. First, it will have a complete path that is the concatenation of $PP_{j'}$ and the tree edge between the two clusters. Second, it will have a partial path consisting of the single vertex of $V_{j''}$. If $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 1, then the resulting vertex cluster will have a complete path that is the concatenation of $PP_{j'}$ and $PP_{j''}$ and the tree edge between them.

Three of the four cases discussed above are shown in Fig. 12. Each small circle represents a single vertex. The top part of Fig. 12 shows two vertex clusters of tree degree 2 being unioned together. Their partial paths are shown in bold, and the new partial path for the unioned cluster is the path containing the bold edges and the dashed edge. The middle part of Fig. 12 shows vertex clusters of tree degree 1 and 3 being unioned together. The partial path of the cluster of tree degree 1 is shown in bold, and the new complete path for the unioned cluster is the path containing the bold edges and the dashed edge. The bottom part of Fig. 12 shows two vertex clusters of tree degree 1 being unioned together. Their partial paths are shown in bold, and the new complete path for the unioned cluster is the path containing the bold edges and the dashed edge. Note that the clusters shown are not basic clusters, which is why the partial paths of clusters of tree degree 1 appear to start in the "middle" of those clusters.

Note that a vertex cluster will have a complete path if and only if it is the union of two clusters of odd tree degree. For any complete path generated when clusters of tree degree 1 and 3 are unioned together, let the single vertex in the cluster of tree degree 3 be called the *top* of the path. We say that complete path P' *dominates* complete path P if and only if the top of path P is contained in P'.

Consider the restricted multilevel partition shown in Fig. 3. The complete paths are shown in Fig. 13. The complete path between vertices 12 and 13 will be associated

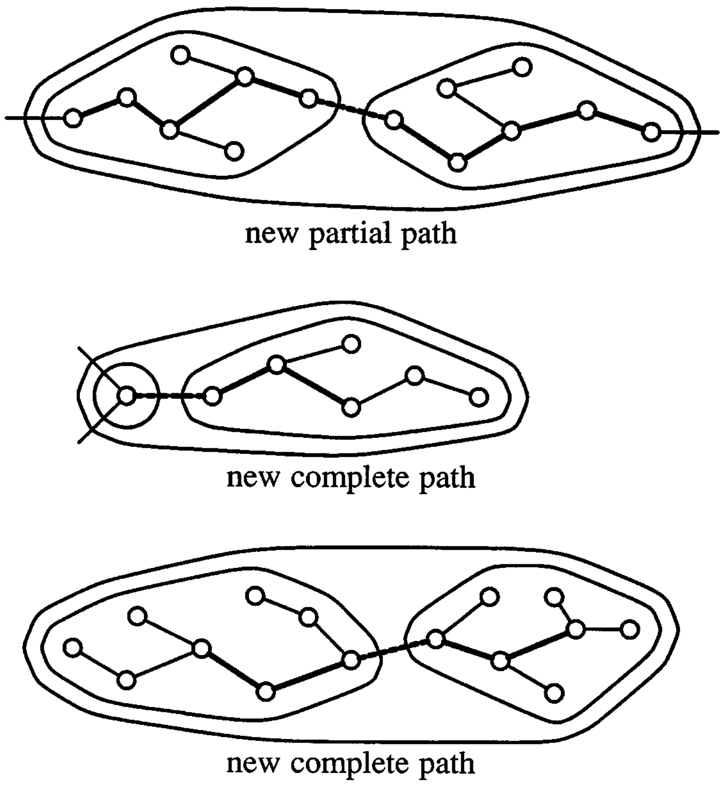

FIG. 12. *Combining partial paths.*

with V_{23} of Fig. 3, the complete path between 8 and 10 will be associated with V_{28}, the complete path between 4 and 7 will be associated with V_{32}, and the complete path between 1 and 14 will be associated with V_{37}. Each of the complete paths except the last has a top, and these are 12, 10, and 4, respectively. The complete path from 1 to 14 dominates each of the other paths. Examples of partial paths are the following. The partial path PP_{15} consists of the single vertex 11, PP_{16} is the path from 2 to 3, PP_{25} and PP_{31} are both the path from 1 to 3, PP_{32} consists of the single vertex 4, PP_{35} is the path from 1 to 4.

We describe how complete paths are used in answering a *same-2-edge-component* query. When a *same-2-edge-component* query on vertices v' and v'' is made, the path in the spanning tree T between v' and v'' is considered in the following way. Either the path contains no edges in the boundary tree, or it contains at least one edge in the boundary tree. In the former case, the path will be wholly contained in one basic cluster, and information associated with the basic cluster will be examined to see if there is a bridge between v' and v''. In the latter case, the path between v' and v'' will consist of three subpaths: the first and third subpaths will contain only edges not in the boundary tree, while second subpath will contain only edges in the boundary tree. First, information associated with the basic clusters containing v' and v'' will be examined to see if there is a bridge on the first or third subpath. If no bridge is found on these subpaths, then information associated with the complete paths containing edges on the path from v' to v'' will be examined.

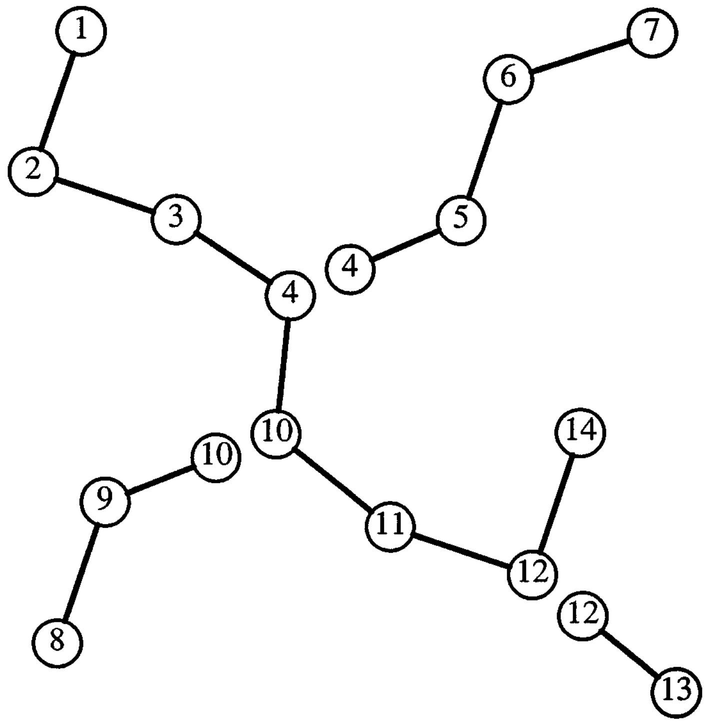

FIG. 13. *Complete paths for the restricted multilevel partition in Fig.* 3.

We maintain ambivalent information for V_j as follows. For boundary vertex w of V_j and cluster V_r at the same level, a nontree edge f with one endpoint in each of V_j and V_r is a *pseudocovering edge* for tree edge e if e is in PP_j and e is on the path in T from w to the endpoint of f in V_j. Pseudocovering edge f actually covers e if w is on the path in T between the endpoints of f. For each boundary vertex w of V_j and cluster V_r at the same level, we shall maintain a best pseudocovering edge in the following sense. A *best pseudocovering edge* for V_j, V_r, and w is a pseudocovering edge with respect to V_j, V_r, and w for the most tree edges in PP_j. This set of tree edges constitutes a subpath of PP_j with one endpoint at w.

Consider the graph in Fig. 1, with a restricted multilevel partition in Fig. 3. For cluster V_{18} and boundary vertex 6, edge 17 is a pseudocovering edge and also a best pseudocovering edge (since it is the only one). However, edge 17 does not actually cover any edges in V_{18}. For cluster V_{27} and boundary vertex 5, edge 17 is a pseudocovering edge. However, edge 17 is not a best pseudocovering edge for V_{27} and boundary vertex 5 since it covers zero edges in the partial path of V_{27} up to vertex 5, while edge 14 is a pseudocovering edge for V_{27} and vertex 5 that covers two edges in the partial path of V_{27} up to vertex 5.

We next discuss carefully the additional information that will be maintained in the nodes of the 2-dimensional topology tree. This includes pointers to partial and complete paths associated with the node, the number of edges in the partial paths, the number of edges from the top of a complete path to the first bridge (if any) in that path, and the best pseudocovering edges.

We first discuss the length of various paths. Note that by *distance* we mean the number of edges in a path in the spanning tree. For the remainder of this section, we shall use the term distance in this way. Let V_j be a vertex cluster of tree degree 1 or 2. Let $length(j)$ be the length of the partial path in V_j. We store $length(j)$ at the node $V_j \times V_j$ in the 2-dimensional topology tree. If V_j is a basic vertex cluster, then any value $length(j)$ can be determined by inspection of V_j. If V_j is not a basic vertex cluster, then any value $length(j)$ can be computed in constant time given the lengths of the partial paths of the children in the topology tree.

We next discuss the pseudocovering edges. Let V_j and V_r be two distinct clusters at the same level, with V_j of tree degree 1 or 2. Let w be a boundary vertex of V_j, and let PP_j be a partial path that contains w as an endpoint. For any vertex u in V_j, recall from the previous section the definition of $proj(j, u)$, which is the vertex on PP_j that is nearest to u in the tree. (Here we use a slight extension of the definition from the last section, in that now we allow V_j to be also of tree degree 1.) For each boundary vertex w of V_j, let $maxcover(j, r, w)$ be the maximum of the distances from w to $proj(j, u)$ taken over the set of nontree edges (u, v) with u in V_j and v in V_r. (If there is no edge with one endpoint in each of V_j and V_r, let $maxcover(j, r, w)$ be $-\infty$.) A *best pseudocovering edge* is a pseudocovering edge that realizes a particular $maxcover(j, r, w)$ value. The values $maxcover(j, r, w)$ for any particular value of j and r are stored in node $V_j \times V_r$ of the 2-dimensional topology tree.

If V_j and V_r are basic vertex clusters, then $maxcover(j, r, w)$ can be computed for any particular j, all boundary vertices w of V_j, and all r by inspection of V_j. Then $maxcover(j, r, w)$ is the maximum distance $d(w, proj(j, u))$ taken over all edges (u, v) with u in V_j and v in V_r. If V_j and V_r are not basic vertex clusters, then $maxcover(j, r, w)$ can be computed in constant time given the $length$ values for all children of V_j in the topology tree and the $maxcover$ values for all children of $V_j \times V_r$ in the 2-dimensional topology tree. We specify this computation in detail. If the node for V_j in the topology tree has a single child $V_{j'}$, then $maxcover(j, r, w)$ is the maximum of $maxcover(j', r', w)$ taken over the one or two clusters $V_{r'}$ that form V_r. Otherwise, V_j is formed from two clusters $V_{j'}$ and $V_{j''}$, and we assume without loss of generality that w is in $V_{j'}$. If $V_{j''}$ is of tree degree 3, then the partial path of V_j is trivial and thus for the single vertex w in $V_{j''}$, $maxcover(j, r, w)$ is 0 if there is some nontree edge with one endpoint in each of V_j and V_r and is $-\infty$ otherwise. Such a nontree edge exists if $maxcover(j', r', w') > -\infty$ for $V_{r'}$ a child of V_r and w' the single boundary vertex of $V_{j'}$. Suppose that neither $V_{j'}$ nor $V_{j''}$ are of tree degree 3. Let w' be the boundary vertex of $V_{j'}$ adjacent to $V_{j''}$, and let w'' be the boundary vertex of $V_{j''}$ adjacent to $V_{j'}$. Then $maxcover(j, r, w)$ is the maximum taken over the one or two clusters $V_{r'}$ that form V_r of $maxcover(j', r', w)$ and $maxcover(j'', r', w'') + 1 + length(j')$. Note that it is easy to keep track of an edge that yields each particular $maxcover(j, r, w)$ value. (This is needed when we consider deleting a tree edge.)

We next discuss partial and complete paths. Let V_j be a vertex cluster. We maintain partial and complete paths in the following form. Each such path is represented by a balanced tree in which the leaves from left to right represent consecutive edges on the path. Associated with each node in the tree is a value $somecov$ such that $cover(e) = 1$ for edge e on the path between boundary vertices if and only if $somecov(x) = 1$ for some node x on the path from the root to the leaf representing edge e. In addition, there is a value $allcov$ such that $allcov(x)$ is 1 if and only if $somecov(x) = 1$ or $allcov(y) = 1$ for each child y of x. We assume that partial path PP_j and complete path CP_j (if it exists) are specified as a pointer to a structure of the above type.

We describe how to set up partial paths. If V_j is a basic vertex cluster, then the partial path can be set up by inspection of V_j. If V_j is not a basic vertex cluster and has just one child in the topology tree, then its partial path is the same as its child. If V_j is the union of two clusters $V_{j'}$ and $V_{j''}$ and is also not the set of all vertices, then its partial path is formed by concatenating the relevant partial paths of children, as discussed previously. The tree edge between $V_{j'}$ and $V_{j''}$ is initially assumed to have a *cover* value of 0. Certain *somecov* values are adjusted to reflect the effect of the best pseudocovering edges between $V_{j'}$ and $V_{j''}$. For example, suppose V_j, $V_{j'}$, and $V_{j''}$ are all of tree degree 2. Let w' be the boundary vertex of $V_{j'}$ adjacent to a vertex in $V_{j''}$, let w'' the boundary vertex of $V_{j''}$ adjacent to w', and suppose $maxcover(j', j'', w') \neq -\infty$. Then we modify the *somecov* values to reflect the fact that a subpath of length $maxcover(j', j'', w') + 1$, starting in $V_{j'}$ and ending at w'', is covered. This can be done by searching in the tree structure for the partial path to find the extreme edges in the subpath. A set of $O(\log n)$ nodes in the tree cover all and only the edges in the subpath, and the *somecov* values of these nodes should be set to 1. The *allcov* values of these nodes and their ancestors should also be adjusted. A similar operation would be performed with respect to $maxcover(j'', j', w'')$. Other cases, in which V_j is of tree degree less than 2 (meaning that the sum of the tree degrees of $V_{j'}$ and $V_{j''}$ is less than 4), are handled similarly.

We next discuss complete paths. Let V_j be a vertex cluster. Suppose V_j is a cluster that is the union of a cluster $V_{j'}$ of tree degree 1 and a cluster $V_{j''}$ of tree degree 3. Then the single vertex of $V_{j''}$ is the top of the complete path at V_j. Complete path CP_j is the result of concatenating the edge between $V_{j'}$ and $V_{j''}$ onto $PP_{j'}$. In addition, the *somecov* and *allcov* values must be modified to show the effect of nontree edges with precisely one endpoint in $V_{j'}$. Let $mcov(j)$ be one plus the maximum of the values $maxcover(j', r, w)$, where $r \neq j'$ and w is the single boundary vertex in $V_{j'}$. Then modify the *somecov* values to reflect the fact that the first $mcov(j)$ edges of CP_j from the top are covered. Let $toptobr(j)$ be the distance from the top of the complete path to the first bridge (if any) in the complete path. This can be found by searching in the tree structure for the complete path. There will be a bridge in the complete path if and only if the *allcov* value of the root of the tree structure is 0. Search down from the root, always taking the child representing a subpath closer to the top of the complete path when there are two children and both have *allcov* value equal to 0. Note that if both $V_{j'}$ and $V_{j''}$ are of tree degree 1, then V_j is the set of all vertices, and there is no top of the complete path.

The operations performed on partial and complete paths are concatenation and the update of *allcov* and *somecov* values. To make these operations efficient, we share common subtrees in the tree representations of partial and complete paths. When two paths are concatenated together, no nodes in the existing structures are changed or deleted. Rather, new nodes are allocated to handle structural changes. When an *allcov* or *somecov* value is changed, new copies are made of any node that has a value change or has a descendant that has a value that changes. To prevent the space from increasing without bound, a reference-count system should be used. The appropriate pointers PP_j and/or CP_j are maintained in node $V_j \times V_j$ of the 2-dimensional topology tree. Whenever a node V_j is identified as changing during an update, the appropriate pointers PP_j and/or CP_j are set to null, and then any nodes whose reference counts go to zero are deleted.

An update data structure Q will consist of a topology tree for a spanning tree T, a 2-dimensional topology tree, and representations of basic vertex clusters. Each node in the topology tree will have the index of the corresponding cluster and a pointer to the node's parent. Each node in the 2-dimensional topology tree will have the indices of the corresponding clusters, along with the following. If the node is of type $V_j \times V_j$, it will have *length*, *PP*, *CP*, and *toptobr* values. If the node is of type $V_j \times V_r$, it will have *maxcover* values. The representation of a basic vertex cluster V_j will consist of a list of vertices, a list of edges with both endpoints in the cluster (both tree and nontree edges), and, for every other cluster V_r, a list of nontree edges with one endpoint in each of V_j and V_r and the associated $maxcover(j, r, w)$ values. In addition, we keep information that will help to determine if there is a bridge between two vertices in V_j, as discussed below.

We can find the *cover* values for edges not in the boundary tree as follows. Generate a *reduced cluster* for the basic cluster V_j as follows. The vertex set will be the same in the reduced cluster as in the basic cluster. Any edge (tree or nontree) with both endpoints in the basic cluster will be in the reduced cluster. For any nontree edge (u, v) with u in the basic cluster, v not in it, and $proj(j, u) \neq u$, edge $(u, proj(j, u))$ will be in the reduced cluster. We then find the biconnected components of the reduced cluster. Any tree edge that is in a biconnected component consisting of more than one edge will have nonzero *cover* value. To represent this information, we maintain the following. For each vertex u, keep the distance $d(u, proj(j, u))$, and also keep the distance $disttobr(u)$ to the bridge nearest to u on the path from u to $proj(j, u)$. (Let $disttobr(u)$ be ∞ if there is no bridge between u and $proj(j, u)$.) For each y on a partial path in a basic cluster, keep the distance to both endpoints of the partial path, and keep a data structure to find the lowest common ancestor [HT], [SV] with respect to the tree rooted at y and induced on all vertices u such that $proj(j, u) = y$.

We are now ready to discuss how a *same-2-edge-component*(v', v'') query is handled. Let v' and v'' be in basic clusters $V_{j'}$ and $V_{j''}$, respectively. If $j' = j''$ and $proj(j', v') = proj(j', v'')$, we do the following. Let $y = proj(j', v') = proj(j', v'')$. Find the lowest common ancestor z of v' and v'' in the tree rooted at y. It follows that v' and v'' are in the same bridge-component if and only if $disttobr(v') \geq d(v', y) - d(z, y)$ and $disttobr(v'') \geq d(v'', y) - d(z, y)$.

Suppose $j' \neq j''$ or $proj(j', v') \neq proj(j'', v'')$. If $disttobr(v') < d(v', proj(j', v'))$ or $disttobr(v'') < d(v'', proj(j'', v''))$, then v' and v'' are not in the same bridge-component. Otherwise, we examine the set of complete paths containing edges in the path from $proj(j', v')$ to $proj(j'', v'')$. We examine these paths as we search up through the topology tree. At the top of each complete path in the sequence except the highest one, we shall use the *toptobr* value to test if a bridge comes in the relevant portion of the complete path. At the highest complete path, we examine the appropriate *allcov* values to see if a bridge appears in the relevant portion.

We present the identification and the search of the complete paths in our algorithm *search_cps*. As input to our algorithm are the vertices v' and v'' and pointers to the nodes representing $V_{j'} \times V_{j'}$ and $V_{j''} \times V_{j''}$ in the 2-dimensional topology tree, where $V_{j'}$ and $V_{j''}$ are the basic clusters containing v' and v'', respectively. The algorithm sets variable *isbridge* to **true** if there is a bridge between $proj(j', v')$ and $proj(j'', v'')$. The algorithm maintains two variables $vert'$ and $vert''$, representing vertices on the path from $proj(j', v')$ to $proj(j'', v'')$, to indicate that it has checked for bridges between $proj(j', v')$ and $vert'$ and between $vert'$ and $proj(j', v')$. For the partial path of a cluster containing v' that it is examining, it maintains the distances $dist1'$ and $dist2'$ from $vert'$ to the endpoints of the partial path, and similarly for v'' and $vert''$. It uses these distances when checking a complete path for a bridge in the relevant portion of the complete path, then resets $vert'$ and $vert''$ to reflect the additional portions of the path from $proj(j',v')$ to $proj(j'',v'')$ that have been checked. Note that it is not necessary to maintain both $dist1'$ and $dist2'$ if a preliminary search is used to determine which direction to search in on a cluster of tree degree 2. We have chosen not to employ such a preliminary search.

```
proc search_cps($v', v'', j', j''$)
        $isbridge \leftarrow$ false
        /* Initialize $vert'$, $w1'$, $w2'$, $dist1'$, $dist2'$, and $r'$ as follows: */
        $vert' \leftarrow proj(j',v')$
        $w1' \leftarrow$ one endpoint of $PP_{j'}$
        $w2' \leftarrow$ other endpoint of $PP_{j'}$
        $dist1' \leftarrow$ distance from $vert'$ to $w1'$
        $dist2' \leftarrow$ distance from $vert'$ to $w2'$
        $r' \leftarrow j'$
        /* Initialize $vert''$, $w1''$, $w2''$, $dist1''$, $dist2''$, and $r''$ similarly. */
        while $V_{r'}$ has tree degree greater than 0 and $vert' \neq vert''$ do
                /* Handle $V_{r'}$ as follows: */
                if $V_{r'}$ is of tree degree at most 2 and has a sibling
                then
                        $V_{s'} \leftarrow$ sibling of $V_{r'}$
                        Let $ws1'$ and $ws2'$ be the endpoints of the $PP_{s'}$
                        Without loss of generality, assume $w2'$ is adjacent to $ws1'$.
                        $dist2' \leftarrow dist2' + 1 + length(s')$
                        $w2' \leftarrow ws2'$
                        if $V_{s'}$ is of tree degree 3
                        then
                                if the toptobr value at the parent of $V_{r'}$ is less than $dist2'$
                                then $isbridge \leftarrow$ true
                                endif
                        endif
                endif
                $V_{r'} \leftarrow$ parent of $V_{r'}$
                /* Handle $V_{r''}$ similarly. */
        endwhile
        if $vert' \neq vert''$
        then
                Check the portion of $CP(r')$ between $vert'$ and $vert''$.
                if an allcov value along this portion of $CP(r')$ equals 0
                then $isbridge \leftarrow$ true
                endif
        endif
```

This concludes the discussion of how to handle a *same-2-edge-component* query. Let the above algorithm be called *2ec_query*.

LEMMA 9.3. *Algorithm* 2ec_query *identifies a bridge between two given vertices in* $O(\log n)$ *time.*

Proof. We first consider correctness. If $j' = j''$ and $proj(j',v') = proj(j',v'')$, then no edge of the boundary tree is in the path in T from v' to v''. The path from

v' to v'' will go from v' to z to v'', where z is the lowest common ancestor of v' and v'' in the tree rooted at $proj(j', v')$. Checking for a bridge in the two portions of this path yields the correct result.

If $j' \neq j''$ or $proj(j', v') \neq proj(j', v'')$, then the path from v' to v'' in T consists of a subpath from v' to $proj(j', v')$ that is not in the boundary tree, followed by a subpath from $proj(j', v')$ to $proj(j'', v'')$ in the boundary tree, followed by a subpath from $proj(j'', v'')$ to v'' that is not in the boundary tree. A bridge in the first or third subpath is identified by checking $disttobr(v')$ and $disttobr(v'')$. The second subpath is checked correctly by *search_cps*, as we now argue. Algorithm *search_cps* simultaneously searches up through the 2-dimensional topology tree from $V_{j'}$ and $V_{j''}$. For each pair of ancestors $V_{r'}$ and $V_{r''}$ of $V_{j'}$ and $V_{j''}$, respectively, the algorithm maintains the distance of $vert'$ and $vert''$ to the endpoints of the partial paths. In particular (referring to $V_{r'}$, with a similar argument for $V_{r''}$), if $V_{r'}$ has no sibling, then the same information will be maintained at its parent. If $V_{r'}$ is of tree degree 3, then it consists of a single vertex, which is necessarily $vert'$, so that the distances to the endpoints of the partial path of the parent, even when $V_{r'}$ is unioned with a sibling (of tree degree 1), will both remain 0. If $V_{r'}$ is of tree degree at most 2 and has a sibling, then one of the two distances to endpoints of the partial path must be updated. If the sibling is of tree degree 3, then the parent will contain a complete path, in which case the portion of the complete path from $vert'$ to the vertex of $V_{s'}$ must be checked and $vert'$, $dist1$, and $dist2'$ must be reset. When the tree degree of $V_{r'}$ is 0, then the complete path of $V_{r'}$ should be checked between $vert'$ and $vert''$ in the case that $vert' \neq vert''$.

We next consider the time. If v' and v'' are presented in terms of pointers, then clusters $V_{j'}$ and $V_{j''}$ can be identified in constant time; otherwise, $O(\log n)$ time can be used to search a dictionary. The values $proj(j', v')$ and $proj(j', v'')$ can be looked up in constant time. If $j' = j''$ and $proj(j', v') = proj(j', v'')$, then the lowest common ancestor z of v' and v'' can be found in constant time, and the values *disttobr* can be accessed in constant time as well. If $j' \neq j''$ or $proj(j', v') \neq proj(j', v'')$, then checking the first and third subpaths takes constant time to access and compare *disttobr* values. The second subpath can be checked in $O(\log n)$ time, as we now argue. Each iteration of the while loop of *search_cps* takes constant time. Searching the complete path for the cluster of tree degree 0 will take $O(\log n)$ time since there are $O(\log n)$ nodes in the complete path that cover exactly the portion of the path between $vert'$ and $vert''$, and it takes constant time to check the *allcov* value of each node. □

We consider our graph in Fig. 1, using the restricted multilevel partition of Fig. 3, with its associated complete paths, as shown in Fig. 13. There is only one bridge in the graph, edge $(3, 4)$. The value *toptobr* value is 0 for each of the complete paths from 4 to 7, 8 to 10, and 12 to 14. Recall that each basic vertex cluster is a single vertex. The query *same*-2-*edge*-*component*$(6, 8)$ will determine that there are no bridges from 8 to 10 on that complete path, that there are no bridges from 6 to 4 on the corresponding complete path, and that there are no bridges from 4 to 10 on the topmost complete path. Thus 6 and 8 are in the same 2-edge-connected component. The query *same*-2-*edge*-*component*$(2, 7)$ would examine the complete path from 7 to 4 and then the portion of the topmost complete path from 4 to 2, identifying a bridge, namely edge $(3, 4)$.

We next discuss how to insert or delete an edge. The approach builds on the way an edge was inserted or deleted in section 3. If a tree edge is to be deleted, then we first attempt to swap a nontree edge in to replace it. If there is no edge with which it can swap, then deleting the edge splits the spanning tree and thus also the topology-tree-based data structures. On each of the swap, insert, or delete operations specified by the appropriate *inflate* or *deflate*, the following is done. First, all necessary changes are made to the basic clusters, and all information local to the changed or new basic clusters is computed. For a basic cluster, this information includes a description of each new cluster V_j, consisting of a list of vertices, a list of edges with both endpoints in V_j, and, for every other cluster V_r, a list of nontree edges with one endpoint in each of V_j and V_r, as well as the partial path PP_j, the values $proj(j,u)$, $length(j)$, $maxcover(j,r,w)$, and $disttobr(u)$, and the lowest common ancestor structures for each subtree of T rooted at a vertex on PP_j. For any basic cluster V_r, regroup edges with one endpoint in V_r and the other in a basic cluster V_j that has changed or is new, and recompute $maxcover(r,j,w)$ values.

After rebuilding certain basic clusters, rebuild portions of the 2-dimensional topology tree bottom-up. At nodes that are examined, recompute the information in the *maxcover*, *PP*, *CP*, *length*, and *toptobr* fields. Recall that rules were given earlier on how to generate a partial path from the partial paths of the children, including the *maxcover* values with an endpoint in each of two children. Also discussed is how to generate the complete path, computing and using the *mcov* value.

We discuss a little more how partial and complete paths are handled, since subtrees in the tree structures representing these paths are shared, and there are thus no parent pointers. Each vertex in a partial path will be identified by its distance from the end of the path. Each internal node of the balanced tree representing the path will contain a count of the number of edges in the subpath represented by that node. Thus a vertex in a path can be located in the tree using this positional information.

THEOREM 9.4. *Let G be a graph with n vertices and m edges at the current time. The update data structure Q can be set up in $O(m)$ time and space. Structure Q can be updated in $O(m^{1/2})$ time, while still using $O(m)$ space, and accommodates same-2-edge-component queries in $O(\log n)$ time.*

Proof. Given a spanning tree T for G, basic vertex clusters can be found in $O(m)$ using procedure *cluster* in section 2. Similar to that in [F1], a restricted multilevel partition, a topology tree, and a 2-dimensional topology tree can be found in $O(m)$ time. Since there are $O(m^{1/2})$ basic clusters, it follows from Lemma 2.2 that the number of nodes in the topology tree is $O(m^{1/2})$. Creating a partial path by concatenating the partial paths of the children will cost $O(\log n)$ for each node in the topology tree, or $O(m^{1/2}\log n)$ overall. Generating all other values can be done in time proportional to the number of them.

An edge insertion or edge deletion is handled by *inflate* or *deflate*, respectively. Since all data regarding clusters that change is recomputed, the updating is performed correctly. We next discuss the resources needed to update Q. The size of a description of a basic vertex cluster is $O(m^{1/2})$, and at most a constant number of basic vertex clusters are changed by any update operation. The time to generate the new information associated with a new basic cluster is $O(m^{1/2})$ if we are given the description of the basic cluster(s) from which it is formed. The number of nodes examined and created in generating the new 2-dimensional topology tree is $O(m^{1/2})$ by an argument similar to one in [F1]. The time to compute each value except $mcov(j)$ in a newly created node of the 2-dimensional topology tree is constant if these values are computed bottom-up. For $mcov(j)$ values, these can be found by scanning $maxcover(j',r,w)$ values. Each such *maxcover* value is from a node $V_{j'} \times V_r$ that is examined as the 2-dimensional topology tree is rebuilt, so that each of $O(m^{1/2})$ nodes in the 2-dimensional topology tree can be charged a constant. There are $O(\log n)$ nodes of the form $V_j \times V_j$ that change, so that $O(\log n)$ complete or partial paths must be recomputed, at $O(\log n)$ time each. For the nearest bridge values, $O(\log n)$ complete or partial paths can have their nearest bridges change. The new values can be found in $O(\log n)$ time each. Thus the total time to update Q is $O(m^{1/2})$. By using a reference-count scheme, the form in the updated structure will be the same as if the structure were recomputed from scratch. Thus the space usage will remain $O(m)$.

The correctness and time for queries are established by Lemma 9.3. □

Acknowledgments. I am grateful to John Hershberger, Subhash Suri, Monika Rauch, Pino Italiano, Carsten Bjerring, Haim Kaplan, Sean Ahern, Sean Vyain, and Pok-yin Yu for especially helpful comments. I would also like to thank the referees for their thorough reading of the manuscript and many helpful suggestions.

REFERENCES

[ADKP] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, *A simple parallel tree contraction algorithm*, J. Algorithms, 10 (1989), pp. 287–302.

[BFPRT] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1972), pp. 448–461.

[BW] H. Booth and J. Westbrook, *A linear algorithm for analysis of minimum spanning and shortest-path trees of planar graphs*, Algorithmica, 11 (1994), pp. 341–352.

[BH] R. N. Burns and C. E. Haff, *A ranking problem in graphs*, in Proc. 5th Southeast Conference on Combinatorics, Graph Theory and Computing 19, Utilitas Mathematica Publishing, Winnepeg, MB, Canada, 1974, pp. 461–470.

[CFM] P. M. Camerini, L. Fratta, and F. Maffioli, *The k shortest spanning trees of a graph*, Technical Report Int. Rep. 73-10, IEE–LCE Politecnico di Milano, Milan, Italy, 1974.

[CT] D. Cheriton and R. E. Tarjan, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 310–313.

[CV] R. Cole and U. Vishkin, *The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time*, Algorithmica, 3 (1988), pp. 329–346.

[DSST] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, *Making data structures persistent*, J. Comput. System Sci., 38 (1989), pp. 86–124.

[E] D. Eppstein, *Finding the k smallest spanning trees*, BIT, 32 (1992), pp. 237–248.

[EGI] D. Eppstein, Z. Galil, and G. F. Italiano, *Improved sparsification*, Technical Report 93-20, Department of Information and Computer Science, University of California at Irvine, Irvine, CA, 1993.

[EGIN] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, *Sparsification: A technique for speeding up dynamic graph algorithms*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 60–69.

[F1] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[F2] G. N. Frederickson, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees.* in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 632–641.

[F3] G. N. Frederickson, *A data structure for dynamically maintaining rooted trees*, in Proc. 4th ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 175–184.

[F4] G. N. Frederickson, *An optimal algorithm for selection in a min-heap*, Inform. and Comput., 104 (1993), pp. 197–214.

[FT] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[G] H. N. Gabow, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Comput., 6 (1977), pp. 139–150.

[GGST] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, *Efficient algorithms for minimum spanning trees on directed and undirected graphs*, Combinatorica, 6 (1986), pp. 109–122.

[GI] Z. Galil and G. F. Italiano, *Fully dynamic algorithms for* 2*-edge-connectivity*, SIAM J. Comput., 21 (1992), pp. 1047–1069.

[Hy] F. Harary, *Graph Theory*, Addison–Wesley, Reading, MA, 1969.

[HT] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[Hl2] D. Harel, private communication, 1983.

[KIM] N. Katoh, T. Ibaraki, and H. Mine, *An algorithm for finding k minimum spanning trees*, SIAM J. on Comput., 10 (1981), pp. 247–255.

[L1] E. L. Lawler, *A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem*, Management Sci., 18 (1972), pp. 401–405.

[L2] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.

[MR] G. L. Miller and J. H. Reif, *Parallel tree contraction part* I*: Fundamentals*, in Randomness and Computation, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 47–72.

[M] K. G. Murty, *An algorithm for ranking all the assignments in order of increasing cost*, Oper. Res., 16 (1968), pp. 682–687.

[SV] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[T1] R. E. Tarjan, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[T2] R. E. Tarjan, *Sensitivity analysis of minimum spanning trees and shortest path trees*, Inform. Process. Lett., 14 (1982), pp. 30–33.

[WT] J. Westbrook and R. E. Tarjan, *Maintaining bridge-connected and biconnected components on-line*, Algorithmica, 7 (1992), pp. 433–464.

[Y] J. Y. Yen, *Finding the k shortest loopless paths in a network*, Management Sci., 17 (1971), pp. 712–716.