1991

Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

Report Number:
91-048

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

Greg N. Frederickson

CSD-TR-91-048
June 1991

Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees

Greg N. Frederickson*

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

gnf@cs.purdue.edu

June 13, 1991

Abstract. Ambivalent data structures are presented for several problems on undirected graphs. They 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 can be 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 maintain dynamically 2-edge-connectivity information. Edges and vertices can be inserted or deleted in $O(\sqrt{m})$ 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, resp. Again, the techniques can be extended to maintain an embedded planar graph so that edges and vertices can be inserted or deleted in $O((\log n)^3)$ time, and a query answered in $O(\log n)$ time.

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

*This research was supported in part by the National Science Foundation under grant CCR-9001241 and by the Office of Naval Research under contract N00014-86-K-0689.

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 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. 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 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 ambivalent data structures, 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 the case of a planar graph, we give an algorithm that uses $O(n + k(\log n)^3)$ 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 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 $O(km \log \beta(m,n))$-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 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, resp. For an embedded planar graph, we achieve an update time of $O((\log n)^3)$ and a query time of $O(\log n)$.

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. For 2-edge-connectivity for planar graphs, a data structure in [GI] achieves $O(n^{1/2} \log \log n)$ update and query times, although the associated algorithms cannot verify whether the insertion of any given edge would ruin the planarity of the graph. Note that our update times are considerably better than those in [GI] (except in the case that the graph is planar but no embedding is maintained, in which case they are slightly better) and that our query times are dramatically better than those in [GI].

Our update times for 2-edge-connectivity information in general graphs match the best update times for two simpler problems. Data structures for maintaining connected component information have been studied in [ES], [Hl1] and [F1]. The best of these is in [F1], with an update time of $O(m^{1/2})$ and a query time that is $O(1)$. For the related problem of updating minimum spanning trees, data structures have been investigated in [SP], [CH], [Hl2], [F1] and [EITTWY], with the best time achieved for general graphs being $O(m^{1/2})$ in [F1]. 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, multi-level partition of the vertices, on which the topology tree and 2-dimensional topology tree are based. In addition to ambivalence, we apply other 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 multi-level partition.

Our paper is organized as follows. In section 2 we describe the new multi-level partition and the data structures used for updating spanning trees. In section 3 we describe the variations necessary to make the approach in section 2 work for embedded planar graphs. In section 4 we describe the basic algorithm for finding the k smallest spanning trees, omitting the description of the key data structure. In section 5 we describe this key data structure for general graphs. In section 6 we describe this key data structure for planar graphs. In section 7 we give a data structure to maintain 2-edge-connectivity in general graphs. In section 8 we give a data structure to maintain 2-edge-connectivity in embedded planar graphs.

2. Data structures for maintaining spanning trees

In this section we review basic data structures from [F1] and show how to adapt them for our problems. 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 show that the partition can be applied recursively for only $\Theta(\log n)$ levels. 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], and show how to update this tree when edges and vertices are inserted into or deleted from the underlying graph.

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 $w_0, w_1, \cdots, w_{d-1}$, replace v with new vertices $v_0, v_1, \cdots, v_{d-1}$. Add edges $\{(v_i, v_{i+1}) | i = 0, \cdots, d-2\}$, each of cost $-\infty$, and edge (v_{d-1}, v_0) of cost ∞, and replace edges $\{(w_i, v) | i = 0, 1, \cdots, d-1\}$ with $\{(w_i, v_i) | i = 0, \cdots, d-1\}$, 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.

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 spanning tree of G. 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 vertex clusters are *adjacent* if there is a tree edge that contains one endpoint in each of the clusters.

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.
3. Each cluster of tree degree less than 3 is of cardinality at most $\max\{1, 2z-2\}$.
4. No two adjacent clusters can be combined and still satisfy the above.

It is not hard to show that the number of clusters in a restricted partition of order z is $\Theta(m/z)$.

An operation that changes the structure of T may force a change in the clusters of a restricted partition. If the removal of any edge from T causes the tree degree of a cluster to decrease, then we must check if it should be combined with an adjacent cluster or clusters. If the addition of an edge to T causes the tree degree of a cluster containing more than 1 vertex to increase from 2 to 3, we must do the following. Let the cluster now have boundary vertices w, w' and w''. 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 subtrees as clusters. For each resulting subtree that contains fewer than z vertices, check if the other cluster that is adjacent to it has fewer than z vertices and is of tree degree at most 2. 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.

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

1. For each level $l = 0, 1, \cdots, q$, 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 by viewing each cluster at level $l-1$ as a vertex.
4. There is precisely one vertex cluster at level q, which contains all vertices.

A vertex cluster at level 0 of a restricted multi-level 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 cluster of tree degree 3 will 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.

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

We next show that the restricted multi-level partition has other nice properties. Consider any level $l > 0$ of a restricted multi-level 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*.

Lemma 2.1. 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 multi-level 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 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 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 a restricted multi-level partition, every unmatched vertex of tree degree 1 and every unsheltered unmatched vertex of tree degree 2 must have a neighbor that is matched. Thus each such vertex will receive from some neighbor at least 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 of tree degree 2. We add the following two rules to handle these cases. For any sheltered unmatched nonroot vertex of tree degree 2, 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 unmatched nonroot vertex of tree degree 2 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 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. □

There is an infinite family of examples that match the bound of Lemma 2.1 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 multi-level 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 multi-level partition was defined as it was in [F1].)

Theorem 2.2. The number of levels in a restricted multi-level partition is $\Theta(\log n)$.

Proof. The number of vertex clusters at level 0 is $O(n)$. By Lemma 2.1, 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 $l-1$. It follows that the number of levels is $O(\log n)$. $\square$

We next define structures from [F1] that we will use directly. A *topology tree* for spanning tree 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 multi-level 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 multi-level partition of Figure 2 is given in Figure 3. Each node in the topology tree is labeled with the index of the vertex cluster that it represents.

A topology tree based on a restricted multi-level partition has the same nice properties as a topology tree based on the multi-level 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. We first consider a swap in which a nontree edge replaces a tree edge that spans between two basic clusters. When such a swap is performed, consider the endpoints of the edge swapped in and the edge swapped out.

These endpoints are contained in at most four basic vertex clusters. Remove from the topology tree each node that is a proper ancestor of one of the nodes for these basic clusters. This leaves a forest of topology trees. We then rebuild the topology tree from the bottom up as follows. While the tree is not finished, let l be the lowest level of any of the roots in the forest. First we consider roots at level l representing a clusters whose tree degree is 3. Suppose there is a root at level l representing a cluster W' whose tree degree is 3. If W' is adjacent to a cluster W'' at level l of tree degree 1, and W'' corresponds to some root, then union W' and W'' to give a cluster W at level l+1 (with tree degree 2), and join the two trees by creating a new root and making each of the two previous roots a child of it. Otherwise, make W' a cluster at level $l+1$, by creating a new root and making the previous root a child of it.

Next we consider roots at level l representing a clusters whose tree degree is 2. Suppose there is a root at level l representing a cluster W' whose tree degree is 2. If W' is adjacent to a cluster W'' at level l of tree degree 1 or 2 corresponding to some root, then union the two clusters to give a cluster W at level $l+1$ (with tree degree 1 or 2), and join the two trees by creating a new root and making each of the two previous roots a child of it. Otherwise, if W' is adjacent to a cluster W'' at level l of tree degree 2 and the node for W'' has a single parent named W in its tree, then union W' and W'' to give a cluster named W at level $l+1$ (with tree degree 2), and make the parent of the node for W' the node for W. Otherwise, make W' a cluster at level $l+1$, by creating a new root and making the previous root a child of it.

Next we consider roots at level l representing a clusters whose tree degree is 1. Suppose there is a root at level l representing a cluster W' whose tree degree is 1. If W' is adjacent to a cluster W'' at level l of tree degree 1 corresponding to some root, then union the two clusters to give a cluster W at level l+1 (with tree degree 0), and join the two trees by creating a new root and making each of the two previous roots a child of it. (At this point, the new topology tree will be complete.) Otherwise, if W' is adjacent to a cluster W'' at level l of tree degree 2 or 3 and the node for W'' has a single parent named W in its tree, then union W' and W'' to give a cluster named W at level $l+1$ (with tree degree 1 or 2), and make the parent of the node for W' the node for W. Otherwise, make W' a cluster at level $l+1$, by creating a new root and making the previous root a child of it. This completes the handling of all roots at level l.

The case in which the deleted edge has both endpoints in the same cluster can be handled similarly, once the cluster has been split and recombined as necessary. For each basic vertex cluster that has split or whose tree degree has changed, delete all proper ancestors in the topology tree, and then proceed as discussed above. Call the above algorithm *basic-swap*.

Lemma 2.3. Consider a topology tree based on a restricted multilevel partition. The time required by algorithm *basic-swap* to modify a topology tree to when performing a swap is $O(\log n)$.

Proof. Note that we are not counting the cost of splitting and recombining the basic vertex clusters in the cost of modifying the topology tree. The time to perform this algorithm exclusive of changing the basic clusters will be proportional to the number of nodes deleted, examined and created. There will be at most a constant number of such nodes per level. Since the topology will be of height $O(\log n)$, the lemma follows. □

A *2-dimensional topology tree* for a given topology tree is a tree in which for every 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.

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 2.4. Consider a 2-dimensional topology tree for a topology tree that is based on a restricted multi-level partition. The time required to modify the 2-dimensional topology tree to show the result of performing a swap is $O(m/z)$.

Proof. As in [F1], the time to modify all affected nodes will be $O(m/z)$. □

As in [F1], it is not hard to cast the problems of edge and vertex insertion and deletion into an edge update framework. 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.) When an edge is inserted, the degree of the incident vertices in the original graph increases. If the degree of such a vertex becomes four, then the transformation discussed earlier must be applied to the vertex. If the degree with respect to the original graph becomes greater than four, then the transformation discussed earlier has already been applied but now must be modified. In both cases, the number of new edges and vertices introduced is a small constant. Similar transformations can be performed in reverse if an edge is deleted.

When edges are being inserted or deleted, the number of edges is of course changing. We understand m to be the number of edges currently in the graph. We claim that an update can be carried out in time $O(\sqrt{m})$. This can be achieved as follows. As before, let $z = \lceil \sqrt{m} \rceil$. When the value of z changes due to an insertion or deletion, there will be at least $\sqrt{m}$ updates before z advances to the next value up or down in the same direction. The idea is to adjust a small constant number of basic vertex clusters each time that there is a new update. Since there will be no more than $\sqrt{m}$ clusters that need to be adjusted, the adjustment may be accomplished before a new round of adjustments is initiated. Thus every time an insertion occurs, the clusters can be scanned to find any cluster that is too small and this cluster can be combined with a neighbor as necessary. Similar operations are performed upon a deletion.

Theorem 2.5. Consider a structure based on a restricted multi-level partition. The time required to insert or delete an edge or vertex is $O(\sqrt{m})$, 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 $O(m/z)$. $\square$

3. Data structures for embedded planar graphs

For embedded planar graphs, we use the topology tree of section 2 as a basis for an update data structure. Nontree edges with precisely one endpoint in any given vertex cluster will be ordered according to the embedding and then represented by a balanced tree structure. Our work will follow the general idea in [F1] for representing embedded planar graphs, 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 next define sets of nontree edges, called "boundary sets", with precisely one endpoint in any given vertex cluster, and then define the corresponding edge-ordering balanced trees. We then show how to generate a boundary set of a cluster that is the union of two other clusters from their edge-ordering trees. Next we define the edge-ordered topology tree, which is a topology tree augmented by the edge-ordering information. We then discuss how to update the edge-ordered topology tree to show the result of a swap.

Finally we discuss how to update the edge-ordered topology tree to show the effect of the insertion or deletion of an edge or vertex.

First, we choose $z = 1$ in our restricted multi-level partition, so that each basic vertex cluster will be a vertex by itself. We examine carefully 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 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*, and the corresponding balanced tree an *edge-ordering tree*. Each leaf in the edge-ordering tree will represent an edge in the boundary set. It is easy to identify the 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.

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 $V_{j''}$. If V_j is the set of all vertices, then its boundary set is empty. Suppose that $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3. Make the boundary set of $V_{j'}$ one of the two boundary sets of V_j. The other boundary set of V_j is empty. Suppose that $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2. Then we search each boundary set of $V_{j'}$ to identify the subset of edges with other endpoint in $V_{j''}$. This subset consists of all edges in the boundary set starting at one end and going up to some edge in the ordering. With the boundary sets represented by edge-ordering trees, the "last" edge in this subset can be determined by an appropriate tree search. We then split the boundary sets of $V_{j''}$ at these points, and also split the single boundary set of $V_{j'}$ at the corresponding points. We take those portions of the boundary sets whose edges have one endpoint not in V_j, and concatenate them together to give the boundary set for V_j.

Suppose that $V_{j'}$ and $V_{j''}$ are both of tree degree 2. We note an annoying anomaly that may occur. There may be one or more nontree edges with one endpoint in each of $V_{j'}$ and $V_{j''}$, such that the endpoint in $V_{j''}$ is on the "opposite" side of the path from the endpoint in $V_{j'}$. We call any such edge e a *separating edge*, since the cycle induced by e in the tree separates two clusters that are different from V_j. The possible presence of these edges complicates the search slightly, as we must search the boundary sets of both $V_{j'}$ and $V_{j''}$ to identify the subset of edges in one cluster with other endpoint in the other cluster. As before, such a subset consists of all edges in the boundary set starting at one end and going up to some edge in the ordering. We split the boundary sets at the appropriate points, and concatenate the two remaining subsets on one side of V_j to give the boundary set on that side, and similarly for the boundary set on the other side.

When we form the boundary set of a cluster V_j from the boundary sets of the children, we do not change the edge-ordering trees for the boundary 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. For each cluster, we keep track of the portions of boundary sets of the children that are not used in building the boundary set of the cluster. We call the set of such edges on each side to be the *newly-interior set* of edges. Each newly-interior set of edges will be represented by an edge-ordering tree. In our one remaining case, suppose 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 edges in the newly-interior sets of V_j.

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.

To swap a nontree edge into the tree, replacing a tree edge, we would perform an operation similar to *basic-swap* of section 2, with the following additional work. When nodes are removed from the topology tree, they should be removed from the top down. When a node in the topology tree is removed, its boundary sets 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. Thus edge-ordering trees will be removed as nodes are removed from the topology tree. In rebuilding the topology tree, whenever a parent is created for two nodes, the corresponding operations are done with respect to boundary and newly-interior sets.

This approach is related to that in [F1], but we are specifying it carefully, as 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 search an edge-ordering tree representing a boundary set for $V_{j'}$ to find the last edge that has one endpoint in each of $V_{j'}$ and $V_{j''}$. It is easy to produce in constant time a suitable edge in this boundary set to test. The problem is determining whether the other endpoint of that edge is in $V_{j''}$. 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 cluster V_r 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. We are actually interested in finding the maximum number of edges in common in a given boundary set of $V_{j'}$ and the corresponding boundary set of $V_{j''}$. We store in each node of the edge-ordering tree the number of edges represented by the subtree rooted at that node. 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 *planar-swap.*

Lemma 3.1. 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 2.1 that at level i, $i = 0, 1, \cdots, q$, 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 set-up time. For a cluster V_j of size n_j the time to generate its edge-ordering trees from those of its children is 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 total additional space used by edge-ordering trees by $\sum_{i=0}^{q}(5/6)^i n(1+i\log(6/5))^2$. This quantity is clearly $O(n)$.

By Lemma 2.3, 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. Since there are a constant number of concatenations or splits per node, 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 2.4, 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 3.2. 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 2.3, 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 3.1. □

Theorem 3.3. Using the edge-ordered topology tree, the minimum spanning tree of an embedded planar graph can be updated in $O((\log n)^3)$ time. The data structures will use $O(n)$ space.

Proof. By Lemma 3.1, the edge-ordered topology tree can be updated to show the result of a swap in $O((\log n)^3)$ time. By Lemma 3.2, the edge-ordered topology tree can be updated to show the result of an edge or vertex insertion or deletion in $O((\log n)^3)$ time. Resetting the values in the additional fields of the edge-ordering trees will take constant time per node examined. Thus the total time is as claimed. The space bound follows from Lemma 3.1. □

4. 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 i-th 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 $T_2, \cdots, T_k$ 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, pages 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, not by changing R_j but by creating new nodes and sharing common subtrees as necessary. 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 ∞. 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 $-\infty$. 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 should now be inserted into the heap. 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), $i = 2, 3, \cdots, k$. 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 $G(\mathit{left}(x))$, will be graph $G(x)$ with the cost of $e(x)$ reset to $-\infty$, and spanning tree $T(\mathit{left}(x))$ will be $T(x)$. This completes the definition of binary tree B.

As an example, we consider the spanning trees for the graph in Figure 1. We shall name edges by their weights. In Figure 4 we give binary tree B for this graph. The minimum spanning tree, as shown in Figure 1, has a cost of 91, and is represented by the root of the tree in Figure 4. 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 a 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 put the included/excluded tree edge between the edges to the right and left children. Note that 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], [T2], [BFPRT], [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], [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, $k-1$ *extractmins* and $2(k-1)$ *inserts* 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.

From [F2] it is known that the k-th 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. Suppose that these costs can be viewed as forming a min-heap. Then from [F2] it is known that the k-th 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 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 [F2] 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 Figure 4 is shown in Figure 5.

5. Ambivalent data structures I: best-swap structures

In this section we adapt the data structures from section 2 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 define what we call a "pseudo-swap", which will allow us to design an ambivalent data structure. Using pseudo-swaps, 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 seek to build a data structure in which we can maintain a large set of swaps in 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 is handling nontree edges that have their endpoints in different clusters. We set up ambivalent information for each cluster V_j as follows. For each boundary vertex w of V_j, and every other cluster V_r at the same level, a *pseudo-swap* 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. A pseudo-swap is a valid swap if e is in the path in the tree between the endpoints of f. This condition clearly holds if w is on this path.

Consider the graph shown in Figure 1, with a restricted multi-level partition as shown in Figure 2. Let V_j be V_{18}, boundary vertex w of V_{18} be vertex 6, nontree edge f be edge 17, and tree edge e be edge 6. Then $(6, 17)$ is a pseudo-swap, but it is not a swap, since vertex 6 is not on the path in the tree between vertices 5 an 14, the endpoints of edge 17. Note that if edge 14 were swapped into the tree for edge 12, then $(6, 17)$ would also be a swap.

We next discuss carefully the additional information that will be maintained in the nodes of the 2-dimensional topology tree. These include 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 a given pair of clusters, the cost of a minimum-cost pseudo-swap from a certain class of pseudo-swaps, 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 boundary 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 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 pseudo-swap from a certain class of pseudo-swaps. 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 with an endpoint in each of V_j and V_r, minus the cost of an edge of maximum cost on the path in T from w to the endpoint 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 $nontreemin(j'',r') - treemax(w,w')$. 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 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 $pswapmin(j',r'',w')$, $pswapmin(j'',j',w'')$, 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.

We choose $z = \lceil m^{1/2} \rceil$. 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 *treemax* values and a *swapmin* value, while if it is of type $V_j \times V_r$, for $r \neq j$, it will have *nontreemin* 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 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 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. If the new cluster V_j is merged from 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 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 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.

We discuss carefully how the 2-dimensional topology tree is handled, since subtrees are shared and there are thus no parent pointers. When a swap is performed, the indices of the basic clusters containing the endpoints of the swap edges are available at the root of the 2-dimensional topology tree being updated. For each endpoint, search up in the topology tree from the leaf for the corresponding basic cluster. Then, given this path in the topology tree, one can search down in the 2-dimensional topology tree. As one searches down, one can set temporary parent pointers, so that one can come back up the new 2-dimensional topology tree, computing new values for the various fields of new nodes.

Theorem 5.1. 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 and space.

Proof. Basic vertex clusters can be found in $O(m)$ using an algorithm in [F1]. Similar to that in [F1], a restricted multi-level 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. 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 cluster is $O(m^{1/2})$, if we are given the description of the cluster(s) from which it is formed. The size of the new topology tree is $O(m^{1/2})$, and can be created in that much time. 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 in a newly created node is constant, if these values are computed bottom-up. Thus the total time to update R_i is $O(m^{1/2})$. The space needed is no larger. □

Theorem 5.2. 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. As discussed in section 4, 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 4, there will be $O(k)$ such updates. By the discussion at the end of section 4, the cost of the k-th smallest spanning tree can be found by performing $O(k)$ updates which produce $O(k)$ values, from which one selects the k-th smallest in $O(k)$ time. By Theorem 5.1, 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 5.1, each update will introduce $O(\min\{k^{3/2}, km^{1/2}\})$ additional space. The space bound then follows. □

6. 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 the edge-ordering trees ambivalent information with respect to the boundary 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, while ignoring one significant problem. We then discuss the problem, namely that we have no parent pointers in our tree structures. We introduce internal names of vertices, which are based on a vertex's position in the topology tree, and show how to keep track of internal indices while performing operations that change the structure of edge-ordering trees. We then discuss updating the best-swap structure while using these internal indices. 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 boundary set. 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. Now consider one of the two boundary sets. For each edge (u,v) in the boundary set with u in V_j, consider $proj(j,u)$. 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 a *modified path* $m(P_j)$ in which every vertex of $m(P_j)$ has exactly one vertex projected onto it except for the endpoints, which have none. Between two consecutive vertices x and y of $m(P_j)$ 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 $m(P_j)$ should be set up so as to be consistent with the planar embedding.

We represent information about the modified path within the edge-ordering tree as follows. For each leaf, we keep the cost of the edge in the boundary set, the cost of the next edge in a given direction on the modified path, the cost of the swap using this next next edge, the cost of the next edge in the other direction on the modified path, and the cost of the swap using that next next edge. For each nonleaf node in the edge-ordering tree, we keep the cost of the minimum-cost boundary edge in the subtree rooted at the nonleaf node, the maximum of the costs of next edges in the given direction on the modified path in the subtree, the cost of the best swap if all boundary edges in the subtree can swap with their next edges in this given direction, the maximum of the costs of next edges in the other direction on the modified path in the subtree, and the cost of the best swap if all boundary edges in 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. 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 boundary edges of V_j then project onto this single point in P_j, and all edges in $m(P_j)$ will have cost $-\infty$. We note that the information corresponding to *treemax*, *nontreemin*, and *pswapmin* which we maintained in section 5 is now held within the edge-ordering trees for the boundary sets and the newly-interior sets.

As in section 5, we shall keep for each cluster V_j the cost *swapmin*(j) of the best swap found within V_j. We discuss what additional changes 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 boundary edge of smallest cost in $V_{j'}$. In a fashion similar to that discussed before, the now modified edge-ordering tree for $V_{j'}$ 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 section 3. 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. 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 and f are as before, and also the cost of the minimum-cost swap for the resulting boundary set for 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 we must find the maximum cost of a tree edge on the path between the nearest pair of boundary edges, one edge from each of $V_{j'}$ and $V_{j''}$, 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 previously, we split and concatenate boundary sets and form newly-interior sets. This case is similar to the case in which $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2, except that the set of separating edges (if any) needs to be identified as a *separating newly-interior set.* For each newly-interior set, we query the corresponding edge-ordering tree for the minimum swap in the appropriate direction. 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 of V_j. This concludes the examination of the four cases when two vertex clusters are unioned.

Finally, suppose $V_{j'}$ and $V_{j''}$ are both of tree degree 1. For each newly-interior set, we query the corresponding edge-ordering tree for the minimum swap in the appropriate direction. 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 of V_j.

We next specify a first and somewhat incomplete description of the swap structure. We shall then augment the structure to address a problem that we shall raise shortly. The best-swap structure R_i will consist of a pointer to an edge-ordered topology tree, many of whose subtrees are shared. Note also that subtrees of the edge-ordering trees at various nodes in the edge-ordered topology tree are also shared.

We discuss how to update a best-swap structure when a swap occurs. Perform the procedure *planar-swap* as discussed earlier. 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 5, 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 $-\infty$.

As shown in Theorem 3.3, a minimum spanning tree of a planar graph can be updated in $O((\log n)^3)$ time using structures based on the edge-ordered topology tree. It would at first appear easy to use this representation in the same fashion as we did the 2-dimensional topology tree in section 5 to give a best-swap structure for a spanning tree of a planar graph that can be updated in $O((\log n)^3)$ time. This would then give the desired $O(n + k(\log n)^3)$-time algorithm for finding the k smallest spanning trees. The difficulty is that on each update in the best-swap 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 the best-swap 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 its topology 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 as follows. Consider the path up to the ancestor at level l from the leaf node representing the vertex. Starting with the empty string, concatenate on the right of the string a 0 if the path comes from a left child and a 1 the path comes from a right child. 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 Figure 1, along with the restricted multi-level partition of Figure 2. We give the internal indices of vertex 13, using the topology tree as shown in Figure 3. The level-l indices of vertex 13, for $l = 1, 2, 3, 4, 5$ respectively, are 1, 01, 001, 1001, and 11001.

We now proceed with a complete description of the best-swap structure. The best-swap structure R_i will consist of a pointer to an edge-ordered topology tree for tree T_i and a description of each basic vertex cluster. 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. The edge-ordered topology tree will share subtrees with other best-swap structures.

Each node V_j at level l in the topology tree will have the external index of its corresponding cluster, a list of its boundary vertices specified by both external index and level-l internal index, its *swapmin* value, and the level-l internal indices of the pair of edges realizing that value. 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. 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 $l' \leq l$. 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 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 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 now discuss the full algorithm for generating the k smallest spanning trees. 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 3.

We next discuss updating a best-swap structure when it has been determined that a swap should be performed. This would occur when a value has been extracted from the heap, representing the cost of a tree T_i plus the cost of some swap for that tree, which is represented by the *swapmin* value at the root of its edge-ordered topology tree. Associated with that *swapmin* value will be both the external and internal indices of the endpoints of the edges in the swap pair (e, f). 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 parent pointers. Then we may proceed as in the proof of Lemma 3.3. We first remove from the topology tree each node that is the proper ancestor of one of the nodes representing these endpoints. For each such node we keep track of the external indices of its boundary vertices of the corresponding cluster. We then proceed to rebuild the topology tree from the bottom-up, as described in the proof of Lemma 3.3.

Testing whether two clusters at some level are adjacent can be done by determining if two of their boundary vertices are adjacent. If one cluster W' corresponds to the root of some tree, and we wish to locate a neighboring cluster W'' whose node is in some other tree, we can do the following. We know the edge (w', w'') through which the two clusters are adjacent, and thus know a boundary vertex w'' of cluster W''. We search back up through the topology tree, using the temporary parent pointers, to find the lowest cluster that contains both w' and w''. This corresponds to the lowest node on the path up that has a child whose corresponding cluster has a boundary vertex w''. The internal index of w'' can then be used to search down the tree to find W''. We note that this searching up and down the tree will involve only a constant amount of work per level, over all levels, since every time a search is performed we need not search over a previously searched path again, but merely save pointers to the relevant nodes.

Theorem 6.1. 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 time claim in the theorem follows from Lemma 3.1 and the discussion at the end of section 4. The space follows from Lemma 3.1 and the observation that a constant number of nodes 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. □

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 $\Theta(\log n)$.

7. Ambivalent data structures II: 2-edge-connectivity information

In this section we adapt the data structure from section 2 to give a data structure for updating and querying 2-edge-connectivity information 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 "pseudo-covering 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 7.1. Graph G is 2-edge-connected if and only if $cover(e) = 1$ for each edge e in T.

Proof. Suppose that $cover(e) = 0$ for some edge e. Then removing this edge partitions G into two nonempty subgraphs. Thus G is not 2-edge-connected.

Suppose that G is not 2-edge-connected. Then there is some edge e whose removal separates the graph. But then there is no nontree edge f such that e is on the path in T between the endpoints of f. Thus $cover(e) = 0$. □

Proposition 7.2. 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 $cover(e) = 0$.

Proof. Delete every edge e with $cover(e) = 0$. Each remaining edge will be on a cycle induced in T by a nontree edge. The resulting components are thus the 2-edge-connected components of G. Vertices v' and v'' are not in the same 2-edge-connected component if and only if there was an 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. The first set of edges consists of each tree edge both of whose endpoints are in the same basic cluster, and whose *cover* value can be inferred just by examining information associated with this basic cluster. The second set consists of all other tree edges. 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. Let the *boundary tree* be the smallest subtree of the spanning tree that contains all the boundary vertices of the clusters. The edges of the boundary tree are the edges in the second set. We define a partition of these edges into paths, which we call *complete paths*, based on the multi-level 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 that is of tree degree either 1 or 2, and there will be a partial path associated with each cluster that is the union of two clusters of odd tree degree. For any multi-level 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. As stated before, a basic vertex cluster of tree degree 3 will have no partial path associated with it.

When two vertex sets $V_{j'}$ and $V_{j''}$ are unioned, 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 the partial paths of $V_{j'}$ and $V_{j''}$ 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 the partial path of $V_{j'}$ 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 the partial paths of $V_{j'}$ and $V_{j''}$ and the tree edge between them. 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 multi-level partition shown in Figure 2. The complete paths are shown in Figure 6. The complete path between vertices 12 and 13 will associated with V_{23}, the complete path between 8 and 10 will associated with V_{28}, the complete path between 4 and 7 will associated with V_{32}, and the complete path between 1 and 14 will 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 for V_{15} consists of the single vertex 11, the partial path for V_{16} is the path from 2 to 3, the partial path for both V_{25} and V_{31} is the path from 1 to 3, the partial path for V_{32} consists of the single vertex 4, and the partial path for V_{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 between v' and v'' is considered in the following way. Note that this path will consist of three subpaths, the first and third subpaths containing only edges not in the boundary tree, while second subpath will contain only edges from 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.

We maintain ambivalent information for V_j as follows. Let P be the partial path of V_j. 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 *pseudo-covering edge* for tree edge e if e is in P and e is on the path in T from w to the endpoint of f in V_j. Pseudo-covering 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 pseudo-covering edge* in the following sense. If some edge with endpoints in V_j and V_r covers the edge on P incident on w, then there is such an edge that covers a longest subpath of P ending at w.

Consider the graph in Figure 1, with a restricted multi-level partition in Figure 2. For cluster V_{18} and boundary vertex 6, edge 17 is a pseudo-covering edge, and also a best pseudo-covering edge (since it is the only one). However, edge 17 does not cover any edges in V_{18}. For cluster V_{27} and boundary vertex 5, edge 17 is a pseudo-covering edge. However edge 17 is not a best pseudo-covering 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 pseudo-covering 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 best pseudo-covering 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 pseudo-covering 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 P_j be a partial path that contains w as an endpoint. Let u be a vertex in V_j. Recall from the previous section the definition of $proj(j,u)$, which is the nearest vertex in the tree that is on P_j. (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 distance from w to $proj(j,u)$ such that there is a nontree edge (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 pseudo-covering edge* is a pseudo-covering 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 one of $V_{j'}$ and $V_{j''}$ is of tree degree 3, then the partial path of V_j is trivial and thus $maxcover(j, r, w)$ is $-\infty$. 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 of tree degree 1 or 2. Let $PP(j)$ be a pointer to a partial path (if any) in V_j. We maintain partial paths (and also complete paths) in the following form. Each such path is represented by 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

$cov(x) = 1$ 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 $cov(x) = 1$ or $allcov(y) = 1$ for each child y of x. To save time when we concatenate paths or update values, we share common subtrees wherever possible. The pointer $PP(j)$ is maintained in node $V_j \times V_j$ of the 2-dimensional topology tree.

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 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 pseudo-covering 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 $maxcover(j'', j', w'')$. The other cases in which not both $V_{j'}$ and $V_{j''}$ are of tree degree 2 are handled similarly.

We next discuss complete paths. Let V_j be a vertex cluster. Let $CP(j)$ be a pointer to a complete path in V_j. 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. 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 x 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.

We choose $z = \lceil m^{1/2} \rceil$. 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 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 and 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, keep a data structure to find the lowest common ancestor [HT], [SV] of any two 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' and V'', respectively. If $V' = V''$ 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 $disttobr(v'') \geq d(v'', y) - d(z, y)$.

Next suppose $V' \neq V''$ 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 identify 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. If no bridge is found from such testing, then we shall search the highest path to see if a bridge appears in the relevant portion.

We shall now present the identification and the search of the complete paths in detail. Initially set W' to be V', W'' to be V'', *vert'* to $proj(j', v')$, *vert''* to $proj(j'', v'')$, *dist'* to 0 and *dist''* to 0. While *vert'* and *vert''* are not on the same partial or complete path, do the following. First handle W' as follows. Let U' be the parent of W' in the topology tree. If W' is an only child of U', then reset W' to U'. Otherwise, if the other child of U' contains vertices that are beneath vertices of W' on a complete path, then reset W' to U'. Otherwise, if the other child of U' has tree degree 2, then add to *dist'* the quantity 1 plus the length of the partial path of the other child, and reset W' to U'. Otherwise, if the other child of U' has tree degree 3, then do the following. Add 1 to *dist'*. If the *toptobr* value of the complete path at U' is less than *dist'*, then there is a bridge between v' and v'', and we exit the procedure with this information. Otherwise, reset *dist'* to be 0, W' to U', and *vert'* to be the single vertex of the child of the cluster U' of tree degree 3, This completes the discussion of the other child having tree degree 3. If the other child of U' has tree degree 1, then reset W' to U'. This completes the handling of W'. Perform similar operations with respect to W'', *dist''* and *vert''*. This completes the description of the while loop. If no bridge is found during the execution of the while loop, then do the following. Note that W' will now equal W''. If *vert'* and *vert''* are not on a complete path for $W' = W''$, then they are on the same partial path. Search up through the topology tree to find the complete path that contains this partial path. Then search this path between *vert'* and *vert''*, using the *allcov* values. If *allcov* is 1 for each portion of the complete path between *vert'*and *vert''*, then there is no bridge between v' and v''.

We consider our graph in Figure 1, using the restricted multi-level partition of Figure 2, with its associated complete paths, as shown in Figure 6. 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 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 2, and then the portion of the topmost complete path from 2 to 4, identifying a bridge.

We next discuss how to insert or delete an edge. If a tree edge is deleted, then we identify a nontree edge that causes the *cover* value of the edge to be nonzero. (We are assuming for simplicity that the graph remains connected, but this assumption is actually not necessary. We could use "phony" edges to keep the graph connected, and whenever we insert an edge that renders a phony edge useless, delete that phony edge.) If the tree edge has a nonzero *cover* by virtue of information available in the description of some cluster, then it is easy to identify the appropriate nontree edge. Otherwise, we search the topology tree to find the first covering edge that covered the tree edge we wish to delete. After making whatever changes are necessary to handle the change in the tree, we make the required structural changes to basic clusters, the topology tree, and the 2-dimensional topology tree, as discussed at the end of section 2. For each basic cluster that changes, a new description of the cluster must be computed. As the topology tree and 2-dimensional topology tree are being constructed bottom-up, modify the information in the *length*, *maxcover*, PP, CP and *toptobr* fields.

We discuss a little more how partial 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 7.3. 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 $O(\sqrt{m})$ time and accommodates *same-2-edge-component* queries in $O(\log n)$ time.

Proof. Given a spanning tree T for G_3, basic vertex clusters can be found in $O(m)$ using an algorithm in [F1]. Similar to that in [F1], a restricted multi-level 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 Q. The size of a description of a basic vertex cluster is $O(\sqrt{m})$, 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 cluster is $O(\sqrt{m})$, if we are given the description of the cluster(s) from which it is formed. The number of nodes examined and created in generating the new 2-dimensional topology tree is $O(\sqrt{m})$, by an argument similar to one in [F1]. The time to compute each value in a newly created node of the 2-dimensional topology tree is constant, if these values are computed bottom-up. 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 $O(\sqrt{m})$. □

8. Dynamic 2-edge-connectivity in embedded planar graphs

In this section we sketch our ambivalent data structures for maintaining 2-edge-connectivity in an embedded planar graph. We first discuss which portions of section 7 are used. We next describe how to adapt the modified paths from section 6. We then discuss how to generate augmented edge-ordering trees of a node, given the information of its children. We follow this with a summary of the update structure, along with a short description of how query and update are performed. Finally, we analyze the performance of our structure.

We shall use the edge-ordered topology tree as in section 3 as a basis for our data structure for maintaining 2-edge-connectivity information in embedded planar graphs. We shall thus use boundary sets and newly interior sets. In addition, we shall use the partial and complete paths as in section 7. Note that in some sense things are easier, since basic clusters contain only single vertices. Thus every edge in the path between any two vertices is contained in some complete path.

We shall maintain partial and complete paths as in section 7. We shall keep track of the lengths of these paths, and represent them as balanced trees. As before, the balanced trees will share subtrees, and shall have *cov* and *allcov* fields as before. For each V_j other than the set of all vertices that has a complete path, there will be a *toptobr* value. Also as in section 7, there will be a field in each node of the balanced tree that allow one to find a vertex at any given distance down the path from either of the endpoints.

As in section 6 we shall maintain a modified path $m(P)$ for each partial path, but will maintain different information in the corresponding edge-ordering tree. Between two consecutive vertices x and y of $m(P)$ we shall have an edge with a value *bcinit*. The value *bcinit* will be 0 if x and y represent different vertices, and 1 otherwise. Each leaf in an edge-ordering tree will contain the *bcinit* values of the path edges to the left and right of the path vertex corresponding to the leaf. In each node of the edge-ordering tree there will be a field *project*. This will either be null or the name of a vertex on the partial path. For an edge in the boundary set with endpoint u, $proj(u)$ will be the first non-null *project* value on the path from the root to the leaf representing the edge in the edge-ordering tree. We also maintain a field *distproj* in each node such that the sum of the *distproj* values of nodes on a path from the root to down to a node containing a non-null *project* value, where all ancestors have null *project* values, is the distance on the partial path from $proj(u)$ to one end of the partial path.

We note some special cases. Consider a cluster V_j of tree degree 1. We shall always assume that it has two boundary sets. If it is a basic cluster, then its partial path P will be trivial. In this case we partition its incident nontree edges between the two boundary sets arbitrarily (but consistent with the embedding), and let all edges in $m(P)$ will have a *bcinit* value of 1. If V_j is not a basic cluster, then it will have a nontrivial partial path, and clearly will have two boundary sets.

Consider the handling of edge-ordering trees when two clusters $V_{j'}$ and $V_{j''}$ are unioned to give cluster V_j. Suppose $V_{j'}$ is of tree degree 1, and $V_{j''}$ is of tree degree 3. Append the tree edge between $V_{j'}$ and $V_{j''}$ onto the partial path of $V_{j'}$, giving the complete path P for V_j. Identify the edge (if any) in the boundary set(s) of $V_{j'}$ whose endpoint u is such that $proj(u)$ is as far as possible from $V_{j''}$ on P. Adjust the *cov* and *allcov* values of $O(\log n)$ nodes in the balanced tree for the path to reflect the fact that an edge covers all path edges from $proj(u)$ to the *top*. Merge the edge-ordering trees of the two boundary sets of $V_{j'}$ into one tree. Take the resulting edge-ordering tree and make it the edge-ordering tree for one side of V_j, with the other side having an empty tree. Set the *project* field of the root of the first edge-ordering tree to the single vertex of $V_{j''}$.

Next suppose $V_{j'}$ is of tree degree 1 or 2, and $V_{j''}$ is of tree degree 2. Merge their partial paths together, along with the tree edge connecting them. Split and concatenate boundary sets as before. For each newly interior set, identify the best edge as far as covering edges in P. Note that if there is a separator newly interior set, then there can be two such edges, one on either end of the set. The position of the corresponding $proj(u)$ vertices in P can be determined by using the *distproj* information in the edge-ordering trees. Set *cov* and *allcov* values in the balanced tree for P to reflect the effect of these best covering edges.

Finally, suppose $V_{j'}$ and $V_{j''}$ are both of tree degree 1. This is similar to the last case, except that the path P formed will be a complete path.

The update structure will consist of a pointer to an edge-ordered topology tree. Note that subtrees of the edge-ordering trees at various nodes in the edge-ordered topology tree will be shared. The query *same-2-edge-component*(v', v'') is handled the same as in the final portion of the query discussed in section 7. Identify the set of complete paths containing edges in the path from v' to v''. Examine these paths in the same manner as in section 7.

We next discuss how to modify the update structure when an update occurs. We perform the appropriate restructuring operations on the edge-ordered topology tree, as discussed in section 3. When a node in this tree is deleted, decrease the reference count of any child of the node, and delete any node whose reference count goes to 0. For any node that is deleted, decrease the count of any node that is a root of one of its associated edge-ordering trees. When a node in the edge-ordered topology tree is created, form the edge-ordering trees associated with it, as discussed previously.

Theorem 8.1. A 2-edge-connectivity update structure of an embedded planar graph can be queried in $O(\log n)$ time and updated in $O((\log n)^3)$ time to show the result of an edge or vertex insertion or deletion. The data structure will use $O(n)$ space.

Proof. By Lemma 3.2, the edge-ordered topology tree can be updated to show the effect of an edge or vertex insertion or deletion in $O((\log n)^3)$ time. Maintaining the additional fields in the various nodes of the 2-edge-connectivity update structure will take constant time per operation performed in updating the edge-ordered topology tree. Thus the total time is as claimed. The space bound follows from Lemma 3.1. □

References

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

[BW] H. Booth and J. Westbrook. Linear algorithms for analysis of minimum spanning and shortest path trees in planar graphs. Technical Report TR-763, Yale University, Department of Computer Science, February 1990.

[BH] R. N. Burns and C. E. Haff. A ranking problem in graphs. In *Proceedings of the 5th Southeast Conference on Combinatorics, Graph Theory and Computing 19*, pages 461–470, 1974.

[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, Italy, 1974.

[CT] D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. on Computing*, 5:310–313, 1976.

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

[E] D. Eppstein. Finding the k smallest spanning trees. manuscript, 1990.

[EITTWY] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 1–11, 1990.

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

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

[F2] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, to appear.

[FT] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[G] H. N. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. on Computing*, 6:139–150, 1977.

[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*, 6:109–122, 1986.

[GI] Z. Galil and G. F. Italiano. Fully dynamic algorithms for edge-connectivity problems. In *Proc. 23nd ACM Symposium on Theory of Computing*, 1991.

[Hy] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Massachusetts, 1969.

[HT] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13:338–355, 1984.

[Hl1] Dov Harel. On line maintenance of the connected components of dynamic graphs. manuscript, 1982.

[Hl2] Dov Harel. private communication, 1983.

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

[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.*, 18:401–405, 1972.

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

[M] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16:682–687, 1968.

[SV] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. on Computing*, 17:1253–1262, 1988.

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

[T1] R. E. Tarjan. Applications of path compression on balanced trees. *J.ACM*, 26:690–715, 1979.

[T2] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Info. Proc. Lett.*, 14:30–33, 1982.

[WT] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. Technical Report CS-TR-228-89, Princeton University, Department of Computer Science, October 1989.

[Y] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

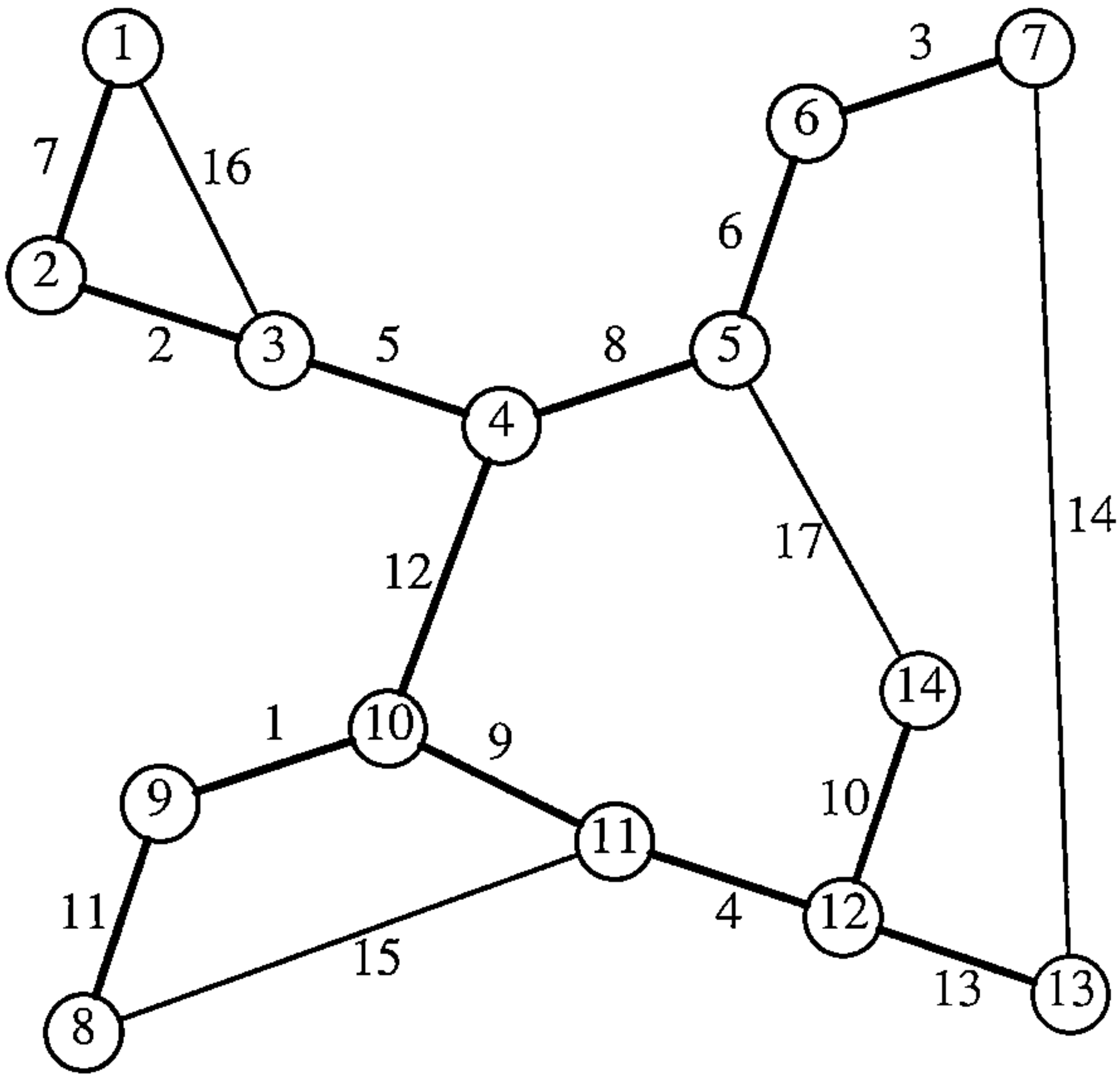

Figure 1. A weighted undirected graph with its minimum spanning tree in bold.

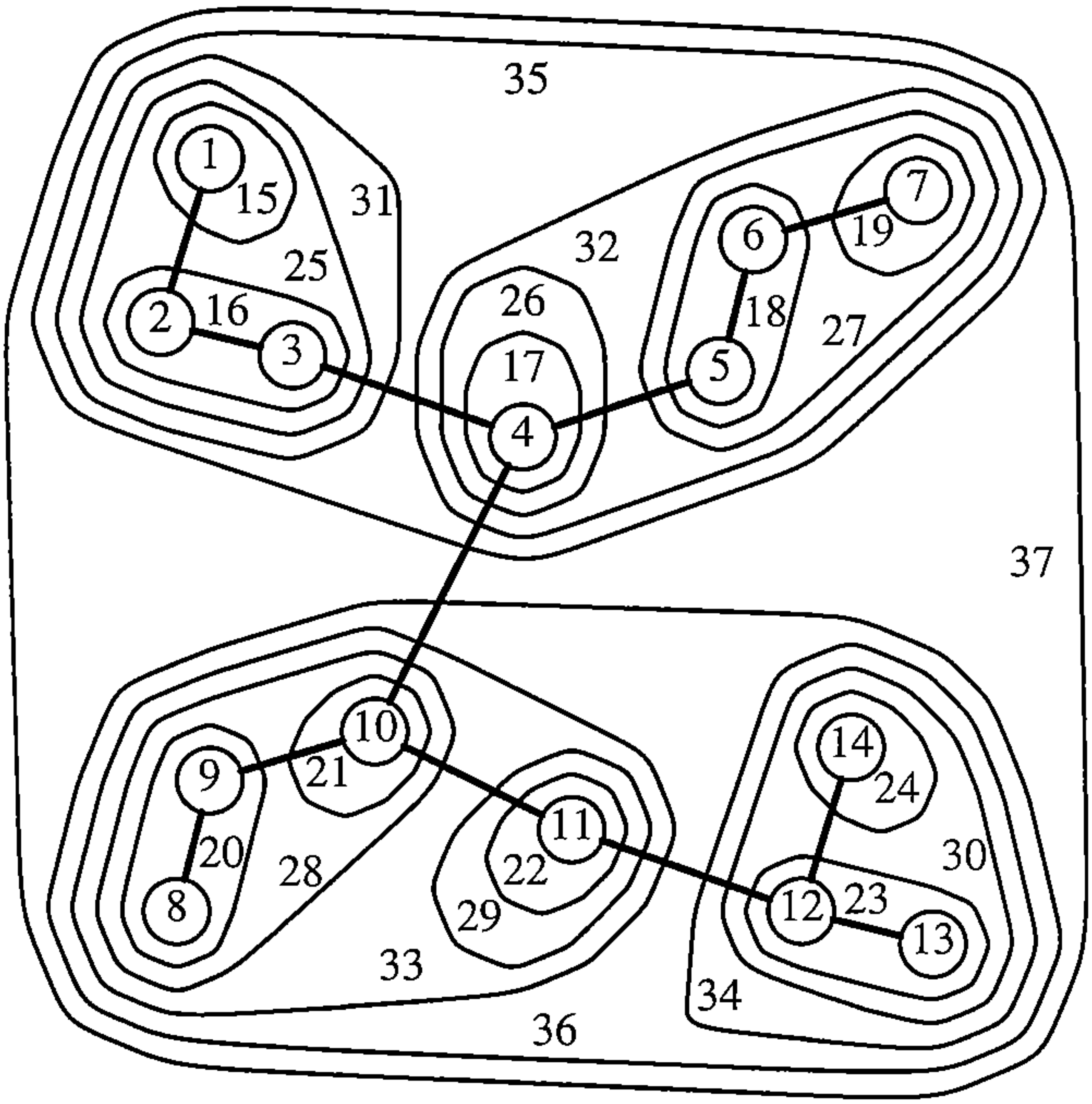

Figure 2. A restricted multi-level partition of the vertices in a spanning tree.

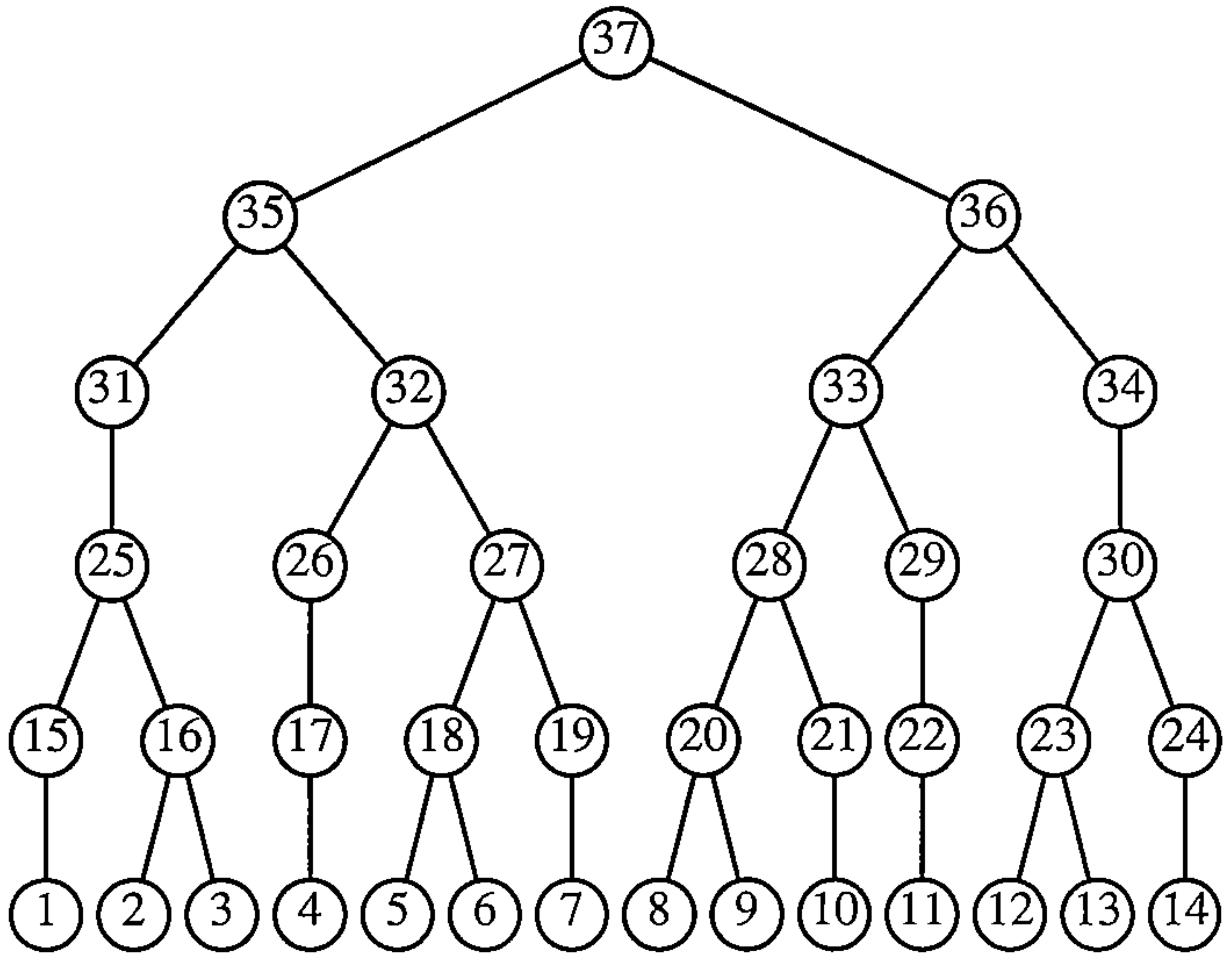

Figure 3. The topology tree corresponding to the restricted multi-level partition in Figure 2.

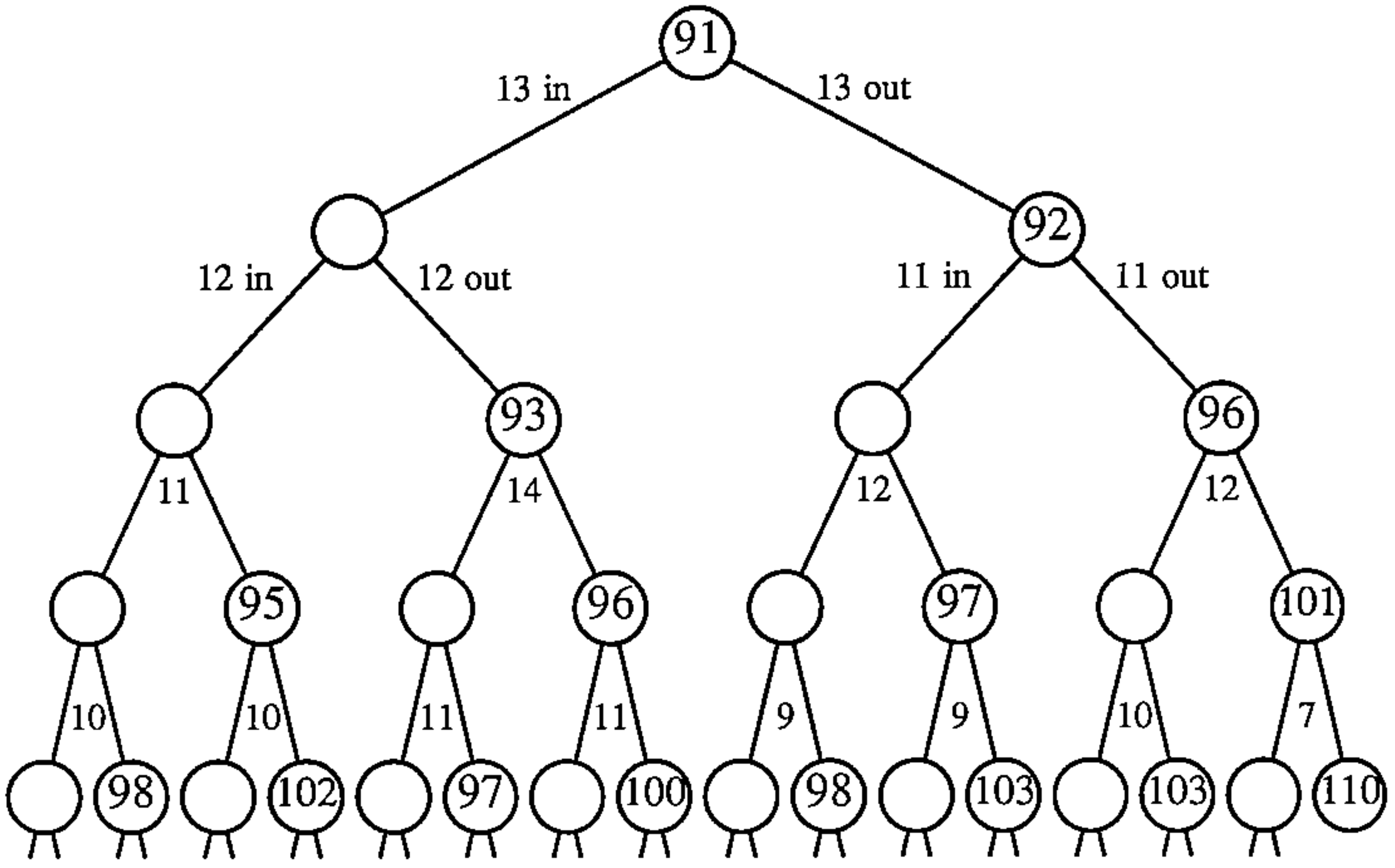

Figure 4. The first four levels of inclusion/exclusion, with spanning tree costs indicated.

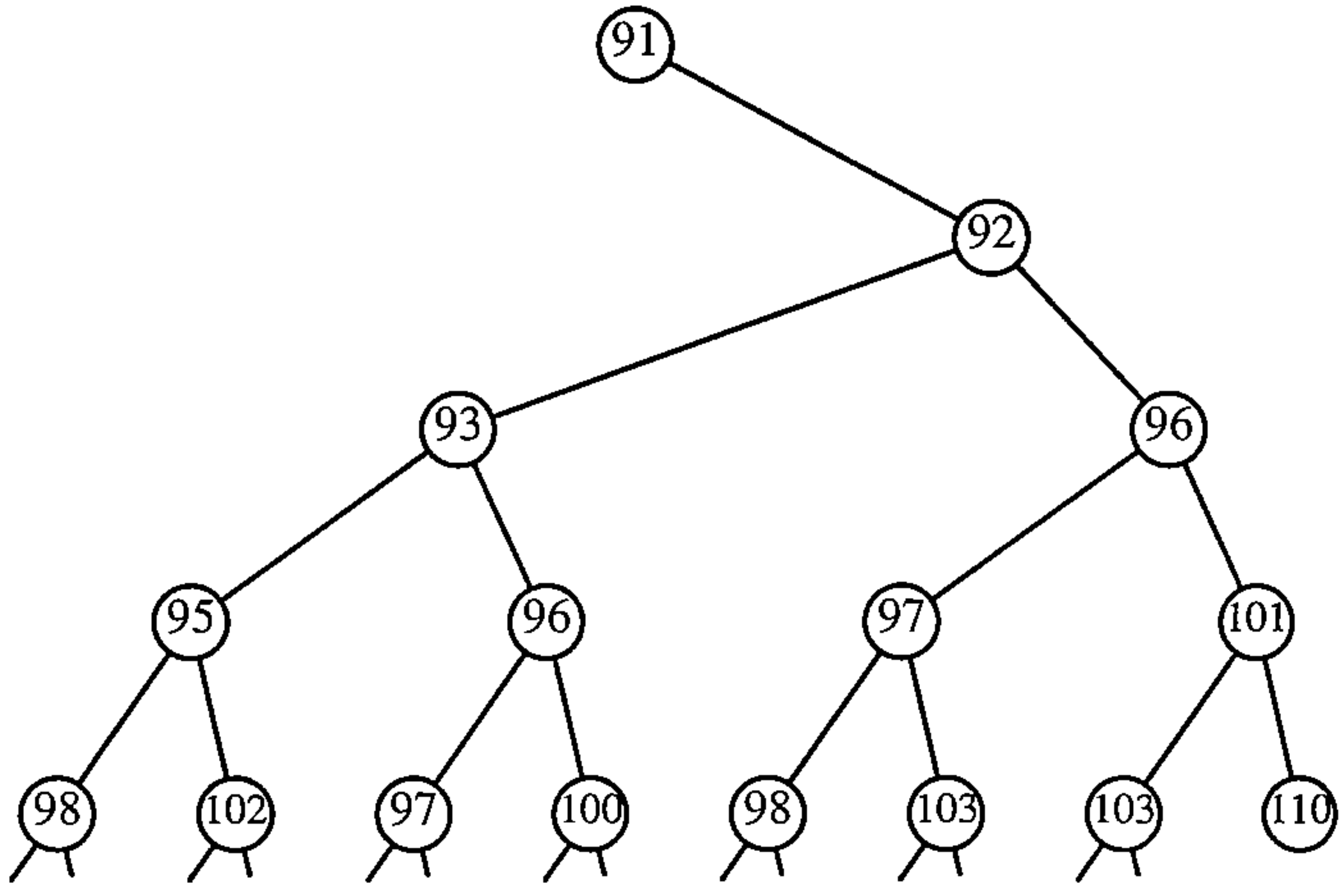

Figure 5. The min-heap induced by inclusion/exclusion.

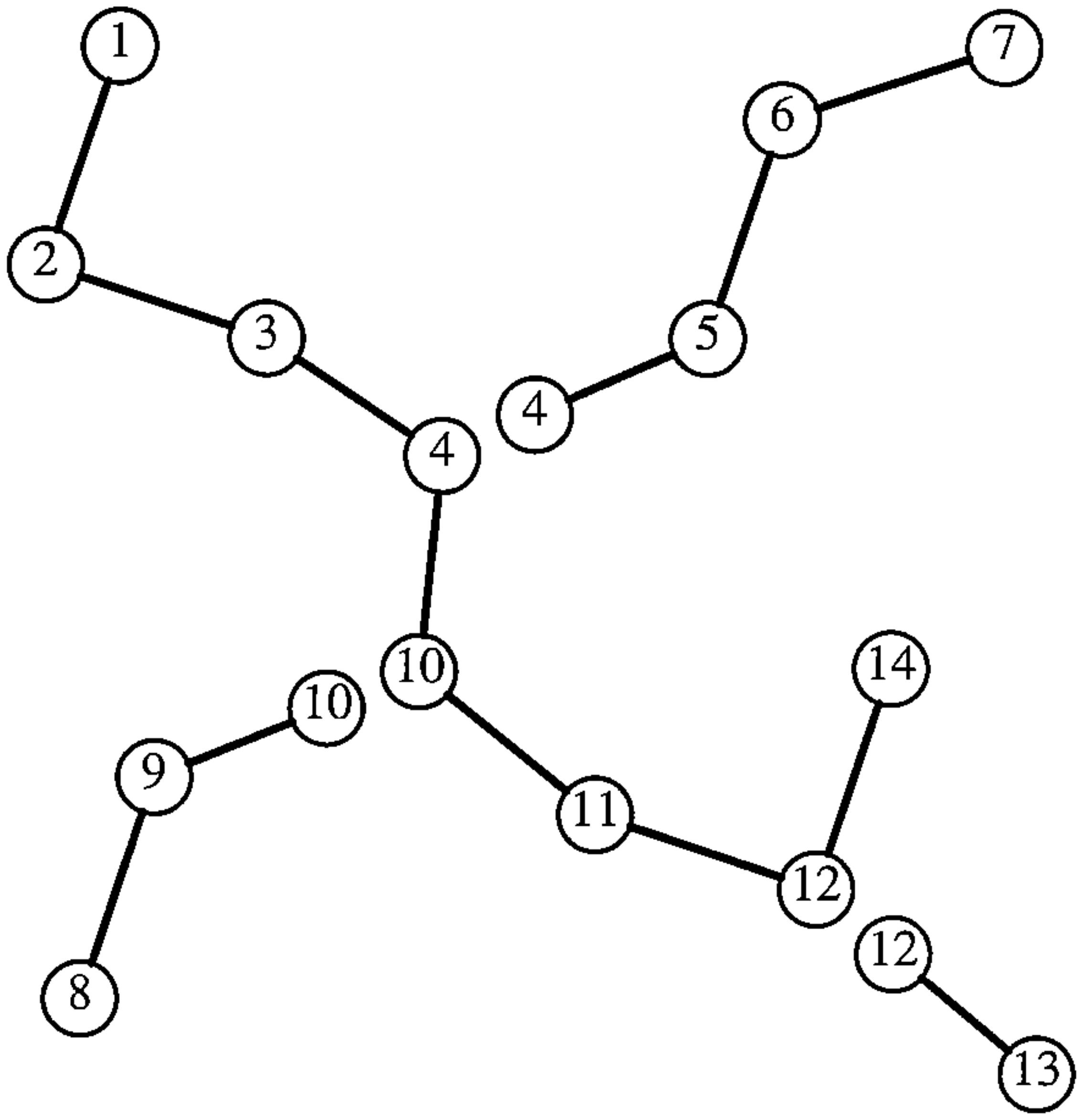

Figure 6. Complete paths for the restricted multi-level partition in Figure 2.