1984

Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications

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

Report Number:
83-449

TR#449

DATA STRUCTURES FOR ON-LINE UPDATING OF MINIMUM SPANNING TREES, WITH APPLICATIONS*

Greg N. Frederickson
Revised May 1984

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

* This research was supported in part by the National Science Foundation under Grant MCS-8201083.

Abstract. Data structures are presented for the problem of maintaining a minimum spanning tree on-line under the operation of updating the cost of some edge in the graph. For the case of a general graph, maintaining the data structure and updating the tree are shown to take $O(\sqrt{m})$ time, where m is the number of edges in the graph. For the case of a planar graph, a data structure is presented which supports an update time of $O((\log m)^2)$. These structures contribute to improved solutions for the on-line connected components problem and the problem of generating the K smallest spanning trees.

Keywords. connected components, data structures, edge insertion and deletion, K smallest spanning trees, minimum spanning tree, on-line computation, planar graphs.

1. Introduction

Consider the following on-line update problem: A minimum spanning tree is to be maintained for an underlying graph, which is modified repeatedly by having the cost of an edge changed. How fast can the new minimum spanning tree be computed after each update? In this paper we present novel graph decomposition and data structures techniques to deal with this update problem, including a useful characterization of the topology of a spanning tree. Furthermore, while dynamic data structures have been applied with success to various geometric problems [OV, LW], our results are among the first [ST, HI1] in the realm of graph problems.

Let m be the number of edges in the graph, and n the number of vertices. The current best time to find a minimum spanning tree is $O(m \log\log_{(2+m/n)} n)$ [CT, Y]. If only straightforward descriptions of the underlying graph and its current minimum spanning tree are maintained, then it has been shown in [SP] that the worst-case time to perform an edge-cost update is $\Theta(m)$. The problem of determining the replacement edges for all edges in the spanning tree can be solved in $O(m\alpha(m,n))$ time [T2], where $\alpha(\cdot,\cdot)$ is a functional inverse of Ackermann's function [T1]. However, that solution is essentially static, so that actually performing replacements can necessitate considerable recomputation.

We show how to maintain information about the graph dynamically so that edge costs can be updated repeatedly with efficiency. After each edge cost change, the change in the minimum spanning tree is determined, and the data structures are updated. We are able to realize an $O(\sqrt{m})$ update time. Moreover, if the underlying graph is planar, we show how to achieve an $O((\log m)^2)$ update time. Our structures require $O(m)$ space and $O(m)$ preprocessing time, aside from the time to find the initial minimum spanning tree. These compare favorably with those developed recently in [HI2], which realize $O(n \log n)$

update times.

Our results are both of practical and theoretical interest. On the one hand, a minimum spanning tree may be used to connect the nodes of a communications network. Variable demand, or transmission problems, may cause the cost of some some edge in the network to change, and the tree will need to be reconfigured dynamically. On the other hand, by focusing on edge cost changes, we have formulated a natural version of the problem of updating a minimum-cost base of a matroid [W]. (In this case, the matroid is a graphic matroid.) Our work leads naturally into the updating of minimum-cost bases of certain simple matroid intersections. These are investigated in [FS1, FS2], in which our data structures are used extensively. The problem of maintaining a minimum spanning tree when vertices are inserted and deleted has been studied in [SP, CH], but the best performance to date is $O(n^2)$. This suggests that because of its connection to matroids, the edge-update problem is perhaps more natural than the vertex-update problem.

We also show how to apply our data structures to a number of related problems to yield improved performance bounds. We cast the problems of edge insertion and deletion into an edge update framework, and realize $O(\sqrt{m_t})$ update times, where m_t is the current number of edges in the graph. Using this, we improve on the update time for the on-line connected components problem in a graph in which edges are being inserted and deleted. The problem is to maintain a data structure so that a query asking if two vertices are in the same connected component can be answered in constant time. A version involving deletions only was examined in [ES], for which the total time for m updates was $O(mn)$. A more general version has been discussed recently in [HH], for which $O(n)$ time per individual update was realized. Our solution uses $O(\sqrt{m_t})$ time per update.

Our data structures can also be used in generating the K smallest spanning trees in increasing order [G]. The best published solution [KIM] requires $O(m \log\log_{(2+m/n)} n + Km)$ time and $O(K + m)$ space. Quite recently, this has been improved in [H12] to $O(Kn (\log n)^2 + m \log n)$ time at the expense of $O(Kn \log n + m \log n)$ space. We improve the time complexity for instances with relatively small K. If K is $O(\sqrt{m})$, our solution uses $O(m \log\log_{(2+m/n)} n + K^2\sqrt{m})$ time and $O(m)$ space. If the graph is planar, then the solution in [KIM] uses $O(Kn)$ time and $O(K + n)$ space. If K is $O(n/(\log n)^2)$ and the graph is planar, our solution uses $O(n + K^2(\log n)^2)$ time and $O(n)$ space.

A preliminary version of this paper appeared in [F].

2. Preliminaries

There are several cases to be handled in edge-cost updating. The cost of an edge may either be increased or decreased, and this edge may currently be either in the minimum spanning tree or or not in the tree. If the cost of a tree edge is decreased, or the cost of a non-tree edge is increased, then there will be no change in the minimum spanning tree.

In the two remaining cases, the minimum spanning tree may be forced to change. However, at most one edge will leave the tree, and one edge will enter the tree. If the cost of a non-tree edge (v,w) is decreased, then this edge may enter the tree, forcing out some other edge. This case may be detected by determining if the maximum cost of an edge on the cycle that (v,w) induces in the tree has greater cost than $c(v,w)$. An obvious implementation of this test would use $\Theta(n)$ time. A faster approach uses the dynamic tree structures of Sleator and Tarjan [ST]. A maximum cost edge (x,y) can be found using the operations $evert(v)$ and $findmax(w)$. The operation $evert(v)$ makes v the root of the dynamic tree structure, and $findmax(w)$ finds the maximum cost edge on the path from w to the root. The dynamic tree may be updated using $cut(x,y)$ and $link(v,w)$. The operation $cut(x,y)$ deletes edge (x,y) from the tree, and $link(v,w)$ adds edge (v,w). As discussed in [ST], the worst-case time required to perform these operations is $O(\log n)$.

The most interesting case is if the cost of a tree edge (x,y) increases. Then the edge may be replaced by some non-tree edge. This case may be detected by determining if the minimum cost non-tree edge (v,w) that connects the two subtrees created by removing (x,y) has cost less than $c(x,y)$. In worst case, there can be $\Omega(m)$ edges that are candidates for the replacement edge. Consequently, this case appears to be the most troublesome to deal with.

Our structures are designed to handle graphs in which no vertex has degree greater than three. Given a graph $G_0 = (V_0, E_0)$, we shall produce a graph $G = (V,E)$ in which each vertex satisfies this degree constraint. A well-known transformation in graph theory [H, p. 132] is used. For each vertex v of degree $d > 3$, where $w_0, \cdots, w_{d-1}$ are the vertices adjacent to v, replace v with new vertices $v_0, \cdots, v_{d-1}$. Add edges $\{(v_i, v_{(i+1) \bmod d}) \mid i=0, \cdots, d-1\}$, each of cost 0, and replace the edges $\{(w_i, v) \mid i=0, \cdots, d-1\}$ with $\{(w_i, v_i) \mid i=0, \cdots, d-1\}$, of corresponding costs.

Let $n' = |V|$ and $m' = |E|$. Then it is not hard to see that $n' \leq 2m$ and $m' \leq 3n'/2 \leq 3m$. Thus there are $\Theta(m)$ vertices in the new graph G, and $\Theta(m)$ storage is required. Given a minimum spanning tree $T_0 = (V_0, E_{0t})$ for G_0, it is easy to find a minimum spanning tree $T = (V, E_t)$ for G. For each new vertex v, include $\{(v_i, v_{i+1}) \mid i=0, \cdots, d-2\}$, and replace any edge (w_i, v) with the corresponding edge (w_i, v_i). In sections 3 through 7 of the paper, we shall assume that we are dealing with graph of $O(m)$ vertices, in which each vertex has degree no greater than 3.

3. Topological partitions of the vertex set

In this section we examine a simple solution to our problem that allows for $o(m)$ update times. We first give a procedure for organizing vertices into clusters, based on the topology of the minimum spanning tree. Using this partition, we show how to achieve $O(m^{2/3})$ update times.

We partition the vertices of the minimum spanning tree T on the basis of the topology of the tree. Let z be a positive integer to be specified later. Let E' be a set of edges whose removal from T leaves connected components with between z and $3z-2$ vertices. The vertex set of each resulting connected component will be called a *vertex cluster*, and the collection of clusters will be called a *topological partition of order z*. Such a partition always exists and is in general not unique.

Given a tree with more than $3z-2$ vertices, and of maximum degree 3, a topological partition may be generated as follows. Perform a depth-first search of T starting at any leaf vertex, which shall be identified as the root. Now call *csearch*(*root*), where *csearch*(v) partitions v and its descendants into zero or more clusters of size between z and $3z-2$, and one set of size between 0 and $z-1$. The set is returned to the calling procedure.

```
proc csearch(v)
        local clust
        clust ← {v}
        for each child w of v do clust ← clust ∪ csearch(w) endfor
        if |clust| < z then return(clust)
              else print(clust); return(ϕ) endif
endproc
```

Let a procedure *FINDCLUSTERS* be the procedure that initially calls *csearch*. If *csearch* returns a non-empty set to *FINDCLUSTERS*, *FINDCLUSTERS* should union it in with the last cluster printed.

Lemma 1. Procedure FINDCLUSTERS partitions the vertex set of a spanning tree with maximum degree 3 into vertex clusters of cardinality between z and $3z-2$ in $O(m)$ time.

Proof. It is not hard to see that the clusters which are output do form connected components with respect to tree T. Since vertices are of degree no greater than 3, and the root has degree 1, each vertex in T will have at most two children. Since sets of size at most $z-1$ are returned by *csearch*, and any vertex will have at most two children, any cluster formed at a vertex v will have size at most $2z-1$. A set of at most $z-1$ vertices can be returned to *FINDCLUSTERS*, and when this set is unioned with the last cluster printed out, a cluster of size at most $3z-2$ will result. Thus all clusters are within the the prescribed size bounds. If the sets are implemented as linked lists, then the whole procedure will require time proportional to the size of T. ▪

The number of vertex clusters will be $\Theta(m/z)$. If $z \geq \sqrt{m}$, then there will be $O(\sqrt{m})$ vertex clusters. Once the vertices are partitioned, partition the edges in $E - E_t$ into sets E_{ij} such that an edge in E_{ij} has one endpoint in vertex cluster V_i and the other endpoint in vertex cluster V_j. Thus there will be $O(m)$ sets E_{ij}. For each set E_{ij}, a minimum cost edge is determined. Both of these tasks can be performed in $O(m)$ time. Thus once a minimum spanning tree for G_0 is determined, all other initialization will take $O(m)$ time. The amount of space used may be seen to be $O(m)$.

We now describe how to handle the two more interesting update operations. Suppose the cost of a non-tree edge (v,w) is decreased, so that tree edge (x,y) must be removed from the tree, and (v,w) must be added. Edge (x,y) can be determined in $O(\log m)$ time, as discussed in section 2. Several cases are possible. If x, y, v, and w are in the same cluster, or if x and y are in different clusters, then the cluster need not be changed.

The crucial case is when x and y are in the same vertex cluster, say V_i, which does not contain both v and w. Then this vertex cluster must be split into V_i' and V_i'', and the sets E_{ij} must be split for all j. Since $|V_i|$ is $O(z)$ and each vertex is of degree no greater than 3, $|\bigcup_j E_{ij}|$ is $O(z)$. Thus the splitting may be carried out in $O(z)$ time. If either V_i' or V_i'' has fewer than z vertices, then combine it with a neighboring vertex cluster. If this neighbor now has more than $3z-2$ vertices, it can be split into two clusters by using *csearch*. The total time to determine and perform whatever splits are necessary will be $O(z)$.

If the cost of a tree edge (x,y) is increased, then a minimum cost replacement edge $(v,w) \neq (x,y)$ must be found. To find (v,w), do the following. If (x,y) connects two vertices in the same cluster V_i, split V_i into V_i' and V_i'', and adjust the sets E_{ij}, as above. Removing (x,y) will partition the vertex clusters into two sets. Check the minimum cost edges between every pair of vertex clusters V_i and V_j, where V_i and V_j are in different sets of the partition. Choose the minimum of these to be (v,w). There can be $\Theta(m/z)$ vertex clusters in each set of the partition, so that the time required to check all pairs of vertex clusters will be $\Theta(m^2/z^2)$. As before, splitting V_i into V_i' and V_i'' will use $O(z)$ time.

We may realize best performance for this approach if we choose $z = \lceil m^{2/3} \rceil$. This structure is called structure I.

Theorem 1. Structure I allows the on-line edge-update problem for minimum spanning trees to be solved in $O(m^{2/3})$ time per update, using $O(m)$ space and $O(m)$ preprocessing time, aside from the time required to find the initial minimum spanning tree.

Proof. The preprocessing requirements have already been established. The update times are dominated by $O(z + m^2/z^2)$. Choosing $z = \lceil m^{2/3} \rceil$ gives the desired result. ▪

4. Topology trees

In the previous section we showed how to partition the vertices into clusters to improve update times. In this section we show how to build clusters of clusters, yielding a hierarchical characterization of the minimum spanning tree. This characterization is then used in the next section to aggregate edge set information.

Given a spanning tree T in which each vertex has degree no greater than three, we define a data structure that describes the topology of the tree in a convenient manner. Let the ***external degree*** of a vertex cluster be the number of spanning tree edges with exactly one endpoint in the vertex cluster.
A ***multi-level topological partition*** of the set of vertices satisfies the following:

1. For each level i, the vertex clusters at level i will form a partition of the set of vertices.
2. A vertex cluster at level 0 will contain a single vertex.
3. A vertex cluster at level $i > 0$ is either
   a. the union of 2, 3 or 4 vertex clusters of level $i-1$, where the clusters are connected together in one of the three ways shown in Figure 1, and the external degree no greater than 3, or
   b. a vertex cluster of level $i-1$ whose external degree is 3.

A ***topology tree*** for spanning tree T is a tree in which each internal node has at most four children, and all leaves are at the same depth, such that:

1. a node at level i in the topology tree represents a vertex cluster in level i of the multi-level topological partition, and
2. a node at level $i > 0$ has children which represent the vertex clusters

whose union is the vertex cluster it represents.

Given the vertex clusters for level $i-1$, we can determine how the vertex clusters are unioned together to give vertex clusters at level i. Consider a spanning tree T_{i-1} derived from T by collapsing each vertex cluster of level $i-1$ to a single vertex. Apply procedure FINDCLUSTERS to the tree T_{i-1}, with parameter $z = 2$. This will identify clusters of vertices in the tree T_{i-1} of cardinality two, three, or four, grouped as in Figure 1. For each cluster in T_{i-1} that would have external degree greater than 3, subdivide the cluster so that the resulting subsets each have degree 3. The vertices in T_{i-1} so grouped, represent the vertex clusters of level $i-1$ that should be unioned to get vertex clusters on level i. An example of tree T is shown in Figure 2. The corresponding topology tree is shown in Figure 3.

Lemma 2. Let n be the number of vertices in a spanning tree T. The height of a corresponding topology tree will be $\Theta(\log n)$.

Proof. Consider the generation of the vertex clusters of level $i > 0$, using the vertex clusters of level $i-1$ and the corresponding tree T_{i-1}. Over half the vertices in T_{i-1} will be of degree less than three, and all of them will participate in a unioning from level $i-1$ to i. Since one vertex cluster will replace at least two for each vertex cluster that is unioned, fewer than

$$n - \frac{1}{2}(\frac{1}{2}n) = \frac{3}{4}n$$

vertex clusters will remain after the unionings.

Since the number of vertex clusters unioned at each level is at least a constant fraction of the remaining number, the number of levels until a single vertex is reached is $O(\log n)$. It follows that the the topology tree is of height $O(\log n)$. ▪

Lemma 3. A topology tree can be generated for a given spanning tree T in time proportional to the number of vertices in T.

Proof. Let n be the number of vertices in T. The first iteration will require $O(n)$ time. From the proof of Lemma 2, at least $\frac{1}{4}$ of the remaining vertices are removed on any iteration. Thus total time will be $O(\sum_{i=0}^{\infty} n(\frac{3}{4})^i)$, which is $O(n)$. ▪

We are interested in the operations of deleting an edge from the minimum spanning tree, and connecting two trees via an edge into a minimum spanning tree. These spanning tree operations will force corresponding operations of splitting a topology tree and merging two topology trees. We shall show that each of these topology tree operations can be performed in $O(\log m)$ time.

At first glance, merging and splitting of topology trees would appear similar to the merging and splitting of 2-3 trees [AHU]. However the topology trees represent clusters that satisfy, among other things, degree constraints, and thus must be handled carefully. Adding an edge to merge two trees into a spanning tree may cause the external degree of a vertex cluster to increase from 3 to 4. In this case the vertex cluster must be split, and the tree must be restructured accordingly. On the other hand, deleting an edge may make it possible to include a vertex cluster in some union at a lower level than before.

We first discuss in detail the merging of two topology trees. Consider the edge that is added to connect the two corresponding trees to give the spanning tree. If some vertex cluster has its external degree increased from 3 to 4, choose the most deeply nested such cluster, say W. It must be the union of at least two clusters, and its constituent clusters can be regrouped into two adjacent vertex clusters, W' and W'', such that the external degree of each is now three. We thus replace a vertex cluster W, originally of external degree 3, and now of degree 4, with two vertex clusters, each of degree 3. An example in which a cluster must be split into two clusters is shown in Figure 4a, and the resulting clusters are shown in Figure 4b. The resulting clusters may force the cluster in which they are located to be split, and this effect may propagate upwards in the multilevel partition. The next level up from the cluster in Figure 4a is shown in Figure 4c, with the result in Figure 4d.

Once any critical change in external degree has been handled, the root of the topology tree of smaller height can be joined at the appropriate level of the other topology tree. The operation is similar to inserting a node as a child of some node in a 2-3 tree, in that the insertion of the new node may force the parent and children to be reorganized so that there are two parents, and this effect may then propagate upward. It is not hard to see that nodes along only one path to the root are affected. An example is shown in Figure 5, with levels beneath the root of the smaller topology tree not shown in either tree. The multilevel partition and topology trees are shown before the edge insertion in Figures 5a and 5b, and after the insertion in Figures 5c and 5d. The set V_9 becomes a child of V_{12}, which is then split into V_{14} and V_{15}, which then forces the splitting of V_{13} into V_{16} and V_{17}.

We now discuss the splitting of a topology tree. The edge is deleted, and all clusters containing that edge are split. These clusters are represented by nodes on a path in the topology tree. The pieces of the topology tree are merged back into two trees, in a fashion similar to what is done when fragments of a 2-3 tree are merged after a splitting. Here again the constraints on the clustering shown in Figure 1 must be preserved. An example of clusters that are split is shown in Figure 6a, and the resulting clusters for the two trees are shown in Figure 6b.

Suppose there is a vertex cluster that is an only child and has had its external degree drop from 3 to 2. Choose the most deeply nested such cluster, say W. Identify a cluster W' at the same level as W in the multilevel partition, and that has the lowest common ancestor with W of such clusters in the topology tree. Combine W and W', rearranging the enclosing clusters as necessary. The newly formed cluster may need to be split, because it is not one of the three forms in Figure 1. An example of this is shown in Figure 7a and 7b. Otherwise, there will be one fewer node, and this may cause the combinations to propagate back up in the tree. An example is shown in Figure 7c, with an intermediate result shown in Figure 7d. (The outermost cluster shown must still be unioned with some other cluster at its level.)

Theorem 2. The time required to perform a split of a topology tree, caused by the deletion of an edge in a spanning tree, or to merge two topology trees, caused by adding an edge to create a spanning tree, is $O(\log n)$.

Proof. From the previous discussion, it may be seen that a constant amount of work is done for each node along a constant number of paths in the topology tree. The theorem then follows. ▪

5. Aggregating edge costs using topology trees

In section 3 we outlined a first strategy for updating minimum spanning trees on-line, using a partition of the vertices based on the topology of the minimum spanning tree. We determined that an expensive operation is finding an edge to replace a tree edge that has increased in cost. This operation could take time proportional to the square of the number of vertex clusters. In this section we use the topology tree described in the last section and show how to avoid examining so many edge sets, by aggregating edge set information in a manner based on the topology tree. Using this approach, we show how to achieve $O(\sqrt{m \log m})$ update times.

We would like to generate a data structure in the following manner. Shrink each vertex cluster in a topological partition to a single vertex, yielding a shrunken tree T_r. Now generate a topology tree for T_r. Unfortunately, this is not in general possible, since vertices in T_r may have degree greater than 3. The difficulty is in our rather simple definition of a topological partition, which we now extend to a *simply-connected topological partition*. Such a partition consists of $\Theta(m/z)$ vertex clusters of size $O(z)$, such that any cluster is adjacent to at most three other clusters in the spanning tree, and any cluster with fewer than z vertices must have external degree 3.

Procedure *csearch* from section 3 can be modified to generate the desired partition. Besides returning a set of vertices, the procedure should return the current external degree of the set. The size and external degree of a set generated at v can then be determined. If this set has at least z vertices or has external degree 3, then it should be printed out. The set generated at v will never have external degree greater than 3, for the following reason. As before, each vertex in T will have at most two children. Suppose nonempty sets of vertices are returned from recursive calls to each child. Each of these sets will have external degree at most two. But of this degree of at most two, one was contributed by the child's adjacency to v, which will not be counted. Hence the external degree of the set generated at v will be at most three: at most one from each of the at most two children, plus one for the adjacency of v with its parent (if any).

The simply-connected partition will induce shrunken tree T_s. Note that each leaf in T_s will represent a vertex cluster of size between z and $3z-2$. This follows since such a cluster, generated at vertex v, will have in effect no external degree contributed by its children. Such a set will not have external degree equal to three, so it is output only because its size is at least z. Hence there will be $O(m/z)$ leaves in T_s. Every vertex cluster of cardinality less than z will be represented by a vertex of degree 3 in T_s. Since there will be fewer vertices of degree 3 than leaves in T_s, there will be $\Theta(m/z)$ vertex clusters in a simply-connected partition. We call these vertex clusters *basic vertex clusters*.

We may now generate a topology tree for tree T_s, the tree resulting by shrinking basic vertex clusters in a simply-connected topological partition. We show how to use these structures to improve update times. For each basic vertex cluster V_i, we maintain an image of the topology tree. At the leaf representing basic vertex cluster V_j in tree i, store the set E_{ij}, along with the minimum cost edge in that set. If there is no such edge, then assume a default cost of ∞. At each internal node in the topology tree, maintain the minimum value from among its children. Thus the topology tree is augmented to maintain a heap on edge costs. The space required by the topology tree for one cluster V_i will be $\Theta(m/z)$ for the nodes, and $\Theta(z)$ for the elements in $\bigcup_j E_{ij}$. Thus total space requirements for $\Theta(m/z)$ trees for all the clusters will be $\Theta(m^2/z^2 + m)$, which is $\Theta(m)$ if $z \geq \sqrt{m}$.

Given a basic vertex cluster V_j, suppose we wish to find a path from the root to the leaf representing V_j in V_i's copy of the topology tree. It is sufficient to maintain an original copy of the topology tree with pointers from children to parents. The location of basic vertex cluster V_j in V_i's copy can be found by tracing up from V_j in the original copy of the topology tree. Thus locating the path from the root to basic vertex cluster V_j will use $O(\log(m/z))$ time.

We now consider handling the two more interesting update operations. If the cost of a non-tree edge is decreased, then finding the edge to replace, splitting a basic vertex cluster, and recombining the pieces is similar to that discussed before, except that now there are consequences in terms of the structure of the topology tree. We have already discussed how to split and merge topology trees. In particular, a topology tree can be split on a leaf representing basic vertex cluster V_i in $O(\log(m/z))$ time. Merging two topology trees that are to be joined via an edge will use $O(\log(m/z))$ time to adjust external degrees, and $O(h_1-h_2)$ time to merge the topology trees, where h_1 and h_2 are their heights. It is straightforward to maintain the heap property on the topology trees as they are merged or split. Since each image of the topology tree must be changed, total time is $O((m/z)\log(m/z))$ for all the topology tree manipulations.

In the case in which the cost of a tree edge is increased, we can use the topology trees to find the replacement edge for (x,y) more quickly than before. If x and y are in the same basic vertex cluster V_i, split V_i into V_i' and V_i''. If either is too small, given its external degree, combine it with a neighboring basic cluster, if there is one, and adjust the upper levels of the topology tree as necessary. Now split each copy of the topology tree on edge (x,y) to give two topology trees for each copy before. This split induces a partition of the set C of basic vertex clusters into C' and C''. In Figure 6b, for example, C' would consist of basic clusters 1, 2, and 3, while C'' would consist of the remaining basic clusters 4 through 9. For each basic vertex cluster V_i, one of its now two topology trees will be a heap on edge costs for edges with one endpoint in V_i and the other endpoint in a cluster in C', and the other tree will be a heap on edge costs for edges with one endpoint in V_i and the other in a cluster in C''.

We find the minimum cost replacement edge (u,v) as follows. For each basic vertex cluster V_i in one of the sets, say C', consider V_i's topology tree for the other set C''. Take the minimum value from among those in the roots of all such topology trees. If this value is smaller than the new cost of edge (x,y), then the edge corresponding to this value becomes the replacement edge.

Once the minimum cost replacement edge (u,v) has been chosen, the topology trees can be merged on this edge. Choosing $z = \lceil\sqrt{m \log m}\,\rceil$, we get structure II.

Theorem 3. Structure II allows the edge-update spanning tree problem to be solved in $O(\sqrt{m \log m})$ time per update, using $O(m)$ space and $O(m)$ preprocessing time, aside from the time to find the initial minimum spanning tree.

Proof. Splitting and merging the basic vertex sets will use time $O(z)$. Splitting $\Theta(m/z)$ copies of the topology tree on an edge will take time $O((m/z)\log(m/z))$, and merging the will take the same. The time to examine the roots of $O(m/z)$ topology trees for the replacement edges will be $O(m/z)$. ∎

6. The 2-dimensional topology tree

It is possible to improve the update time over that of structure II by doing the following. In structure II there is a separate copy of the topology tree for every basic vertex cluster V_i. If we combine all the images of the topology tree into one large tree, we can realize slightly faster update times. The leaves of the large tree will be essentially the same as the set of leaves in all copies of the topology tree, with one leaf for each pair of basic vertex clusters. The root of the large tree may be viewed as the union of the roots of all of the copies. Other internal nodes may be viewed as the unions of various internal nodes in the copies of the topology trees. The organization of the large tree will be such that the time to split or merge the structure will be $\Theta(m/z)$, rather than the $\Theta((m/z)\log(m/z))$ of structure II.

We define the *2-dimensional topology tree* in terms of the topology tree. Let V_α and V_β be vertex clusters represented by nodes at the same level in the topology tree. Then there is a node labeled with $V_\alpha \times V_\beta$ in the 2-dimensional topology tree, which represents the set of edges in $E-E_t$ with one endpoint in V_α and the other in V_β. Since edges are undirected, we shall understand $V_\beta \times V_\alpha$ to denote the same node as $V_\alpha \times V_\beta$. The root of the 2-dimensional topology tree is labeled $V \times V$ and represents the set of all edges in $E-E_t$. If a node in the 2-dimensional topology tree represents $V_\alpha \times V_\alpha$, where V_α has children V_{α_1}, V_{α_2}, $\cdots$, V_{α_r} in the topology tree, then $V_\alpha \times V_\alpha$ has children $\{V_{\alpha_i} \times V_{\alpha_j} \mid 1 \le i \le j \le r\}$. Similarly, if a node represents $V_\alpha \times V_\beta$, where $\alpha \ne \beta$ and V_β has children V_{β_1}, V_{β_2}, $\cdots$, V_{β_s} in the topology tree, then $V_\alpha \times V_\beta$ has children $\{V_{\alpha_i} \times V_{\beta_j} \mid 1 \le i \le r,\ 1 \le j \le s\}$. A portion of the 2-dimensional topology tree corresponding to the topology tree in Figure 2 is given in Figure 6. In our structure *III*, leaves of the 2-dimensional topology tree will store the edge sets E_{ij}, along with the minimum cost edge of each set. Internal nodes will have the minimum of the values of their children.

We discuss how to modify a 2-dimensional topology tree when its corresponding topology tree is modified. Each modification in the topology tree affects nodes along a path from the root to some node representing a vertex cluster V_α, which may or may not be a basic vertex cluster. In the corresponding 2-dimensional topology tree, nodes are affected along paths from the root to nodes of the form $V_\alpha \times V_\beta$ for all clusters V_β for which node $V_\alpha \times V_\beta$ exists. For any node V_γ on the path to V_α in the topology tree, all nodes of the form $V_\gamma \times V_\delta$ will be on these paths in the 2-dimensional topology tree. (It is straightforward to verify that for each node V_δ on the same level of the topology tree as V_γ, there will be a node $V_\alpha \times V_\delta$ in the 2-dimensional topology tree.) These nodes $V_\gamma \times V_\delta$ together form a subtree T_α of the 2-dimensional topology tree. In fact the subtree T_α will be isomorphic to that subtree of the topology tree with nodes at the same level as V_α or above. Hence there are $O(m/z)$ nodes in T_α.

Since each node has a number of children bounded by a constant, the time to modify or replace each node in the subtree T_α will be constant. Thus the operations of merging or splitting a 2-dimensional topology tree can be done in time proportional to the number of nodes in T_α, which is $O(m/z)$. To find a replacement edge, one must examine the values in appropriate nodes once the 2-dimensional topology tree has been split. Suppose that the topology tree has been split into two trees, whose vertex sets are the clusters V_α and V_β, with the V_β set having no fewer levels than V_α set. The replacement edge can be found by examining the values at the nodes $V_\alpha \times V_\gamma$ for all such nodes, and taking the minimum. It will take $O(m/z)$ time to find and examine these nodes.

Choosing $z = \sqrt{m}$, we get our structure *III*.

Theorem 4. Structure *III* allows the on-line edge-update problem for minimum spanning trees to be solved in $O(\sqrt{m})$ time per update, using $O(m)$ space and $O(m)$ preprocessing time, aside from the time to find the initial minimum spanning tree.

Proof. As before, splitting and merging the basic vertex sets will use time $O(z)$. All other operations will take $O(m/z)$ time. ▪

7. A data structure for planar graphs

As stated previously, we have been able to do much better in the case that the underlying graph is planar. In this case we do not deal at all with basic vertex clusters, but merely use the multilevel partition. Thus we use a topology tree for the minimum spanning tree T, augmented with additional information. Consider an internal node of the topology tree representing vertex cluster W, whose children represent vertex clusters $W_1, W_2, \cdots, W_r$. Since the graph is planar, it may be laid out so that each cluster W_i is in its own connected region of the plane. All edges between a pair of vertices in any one W_i may be laid out so that they are wholly contained in the appropriate region. An example of such a correspondence is shown in Figure 9, with the tree edges shown as bold lines, the nontree edges as solid lines, and the boundary of the regions shown with dashed lines.

Given a planar embedding of the graph, consider any vertex cluster W_i and its region. The region is either simple, or it has between one and three "holes" in it. One such example is shown in Figure 10a, in which region W_1 has a closed curve bounding it, which separates W_1 from W_2, W_3 and W_4. For each closed curve bounding a region W_i, the edges with one endpoint in W_i and the other not in W_i may be ordered in a natural way, *e.g.* clockwise around the closed curve. A ***boundary*** between two regions is a maximal set of edges between the regions that are consecutive in their ordering with respect to both regions. It is possible that two regions have more than one boundary between. For example, note that in Figure 10b the clusters W_3 and W_4 have two boundaries between them. For $r \geq 3$ regions, it is not hard to show that there are at most $3r - 6$ boundaries between them. Since no vertex cluster will have more than four children in the topology tree, there will be at most six boundaries between the children. Two such cases are shown in Figures 9 and 10b.

The operations that we shall perform on boundaries are splitting a boundary, concatenating two boundaries, and finding a minimum-cost nontree edge in the boundary. We thus represent the boundaries with mergeable heaps, such as those in [AHU]. The merging and splitting of regions are similar to what has already been discussed with respect to topology trees, except now boundaries of regions must also be maintained. We shall discuss the splitting of a region in some detail, leaving the simpler operation of merging to the reader.

We first consider how to split a vertex cluster W in the topology tree, assuming that edge e, a tree edge with endpoints in W, is removed. Our split routine will return two clusters W' and W'', and boundary B between them. Let W be the union of subsets $W_1, W_2, \cdots, W_r$. If e is in some boundary between a pair of the W_i's, then do the following. Determine which of the W_i's will still be connected to each other, *i.e.*, which W_i's will be in W', and which will be in W''. Determine those boundaries between W_i's that will form the boundary between W' and W'', and concatenate them together to form B. Each remaining boundary between the W_i's will separate subclusters of either W' or W''. Return W', W'' and the boundary B between them.

As an example, consider the region W_1 from Figure 9, shown by itself in Figure 11a. Suppose it consists of two subregions, W_{11} with the upper four vertices, and W_{12} with the lower five vertices. Suppose that W_1 is to be split on the dashed edge between W_{11} and W_{12}. The resulting two regions $W_1' = W_{11}$ and $W_1'' = W_{12}$ are shown in Figure 11b, along with the boundary between them, shown as a dotted line.

If e is not in the boundary between a pair of the W_i's, then it is contained in one of the W_i's, say W_j. Recursively split W_j on edge e, which should return W_j', W_j'', and a boundary between them. Split as necessary the boundaries that W_j had with any of the other W_i's. Determine which of the W_i's, along with W_j' and W_j'', are connected, to form the basis of W' and W''. If W_j' is of level one less, merge it with a neighbor. Handle W_j'' similarly. Determine the boundaries between the new W_i's that will form the boundary between W' and W''. As before concatenate these boundaries, and assign remaining boundaries to W' and W''.

Now suppose that the whole graph W in Figure 9 is to be split on the same edge as in Figure 11a. The boundary that W_1 shared with W_3 must be split, as well as the boundary that W_1 shared with W_2. Note that W_1' will be merged with

W_3, absorbing the boundary between into the result W', and W_1'' will be merged with W_2 and W_4, yielding the result W''. The boundary between W' and W'' will be the concatenation of the boundaries between W_3 and W_4, W_1' and W_4, W_1' and W_2, W_1' and W_1'', W_3 and W_1'', and W_3 and W_2, in that order.

The time to split vertex cluster V on edge e may be seen to be $O((\log m)^2)$. As established earlier, the height of the topology tree will be $O(\log m)$. For each level, the number of boundaries that will be split or concatenated will be no greater than some constant. Since each split and concatenation will take $O(\log m)$ time, this work is bounded by $O(\log m)$ per level. Merging vertex clusters together, as in a small W_j' with a larger neighbor, will require $O((h-h')\log m)$, where h' is the level of W_j', and h is the level of its larger neighbor. The level of the resulting vertex cluster will be at least h. Thus the total of the difference in levels will be $O(\log m)$. Thus the merging will be $O((\log m)^2)$ also.

Theorem 5. The edge-update spanning tree problem may be solved in $O((\log m)^2)$ time per update, using $O(m)$ space and $O(m)$ preprocessing time. ▪

6. Edge insertion and deletion, and maintaining connected components

It is not hard to cast the problems of edge insertion and deletion into an edge update framework. When an edge is inserted, the degree of the incident vertices in the original graph increases. If the degree of such a vertex has become four, then the transformation discussed in section 2 must be applied to the vertex. If the degree has become greater than four, then the transformation from section 2 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 may be performed in reverse if an edge is deleted.

When edges are being inserted or deleted, the number m of edges is of course changing. Let m_t be the number of edges in the graph at time t. We claim that an update at time t can be carried out in time $O(\sqrt{m_t})$. This can be achieved as follows. Let $z_t = \lceil\sqrt{m_t}\,\rceil$. We shall also allow basic vertex clusters of size $3z_t - 1$, and basic vertex clusters of external degree less than three of size $z_t - 1$. When the value of z changes due to an insertion or deletion, there will be at least $\sqrt{m_t}$ 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 $\sqrt{m_t}$ 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 6. A minimum spanning tree may be maintained under the operations of insertion and deletions of edges in $O(\sqrt{m_t})$ time per update, where m_t is the current number of edges. ▪

If the graph is planar, then things are even easier, since no parameter z will be adjusted. Thus edge insertion and deletion can be performed in $O((\log m)^2)$ time, provided that the graph remains planar.

Using the above modifications to our basic structure, we can solve the problem of maintaining connected components of a graph on-line. Given a graph in which edges are being inserted and deleted, a data structure must be maintained so that a query about whether two vertices are in the same connected component can be answered in constant time. In addition to our above structure we use the following. Let each edge in the graph have cost 1. In addition, keep a sufficient number of "dummy" edges of cost 2 in the graph to link together the connected components. Any dummy edge included in the augmented graph must be in the minimum spanning tree of the graph.

Give each connected component a number. In each basic vertex cluster, maintain lists of vertices that are in the same connected component. An array *listnum* will give for each vertex v, the index of the list holding v. A second array *compnum* will give for each list l the index of the connected component containing the vertices in l. To answer a query on vertices u and v, compare *compnum*(*listnum*(u)) and *compnum*(*listnum*(v)) for equality.

To insert an edge (u,v) do the following. If the edge is currently a dummy edge, make it a real edge by decreasing its cost to 1. Otherwise insert it with cost 1. If u and v were in different components, merge the components by doing the following. First identify the at most one basic vertex cluster that contains vertices from both components, and concatenate the lists for these components, changing the *listnum* values of the vertices on one list to be the smaller of the two component numbers. Then in each basic vertex cluster containing a list in the higher numbered component, change the *compnum* value of the list to be the index of the lower numbered component. Since there are $O(\sqrt{m_t})$ vertices in any basic vertex cluster, and $O(\sqrt{m_t})$ basic vertex clusters altogether, the time required will be $O(\sqrt{m_t})$. Inserting edge (u,v) may force a dummy edge out of the minimum spanning tree. Delete this edge. This will require work proportional to the total size of a constant number of basic vertex clusters, or $O(\sqrt{m_t})$.

The ideas for deletion are similar. If the edge e to be deleted is not in the minimum spanning tree, delete it. If it is in the tree, and there is a replacement edge for e of cost 1, delete e. Otherwise, increase the cost of e from 1 to 2. Then renumber the component that has split off, split the at most one list in some basic vertex cluster that has vertices in both resulting components, and give the new number to the $O(\sqrt{m_t})$ lists (at most one per basic vertex cluster) containing vertices in the new component.

Theorem 7. The on-line connected components problem can be solved using data structures that allow edge insertion and deletion times of $O(\sqrt{m_t})$. ▪

9. Generating the K smallest spanning trees

In this section we show how to use our data structures to generate the K smallest spanning trees of a graph in increasing order of cost. Each tree in the sequence except for the first can be described in terms of a preceding tree in the sequence, with one tree edge swapped out and replaced by a nontree edge. Thus our output will be in the following form. The minimum spanning tree will be output first, followed by a succinct description of each of the remaining trees. Each remaining tree will be characterized by its cost, a reference to the tree from which it can be derived using a single swap, and that swap.

Our approach is based on a branch-and-bound technique described in [L] and used in [G,KIM]. The set of all spanning trees not yet selected is partitioned on the basis of the inclusion or exclusion of certain edges. When the minimum spanning tree is selected, the set is partitioned as follows. For each edge e_i in the minimum spanning tree, there is a replacement edge f_i of minimum cost. Without loss of generality assume the swap pairs (e_i,f_i) are indexed in increasing order of $c(f_i)-c(e_i)$. We assume that all such costs are unique, with ties broken by lexicography, if necessary. The set of remaining spanning trees is partitioned into $n-1$ subsets with the ith subset containing all spanning trees with edge e_i excluded and $\{e_1, \cdots, e_{i-1}\}$ included.

When the next smallest spanning tree T' is chosen from one of these subsets, the remainder of the subset is partitioned as follows. Let

$\{e_i' | i=1, \cdots, n'-1\}$ be the set of edges in tree T' that are not required to be included in T' because of membership in the subset, and $\{f_i'\}$ the corresponding set of replacement edges of minimum cost that are not required to be excluded from spanning trees in that subset. Once again assume that the pairs (e_i', f_i') are in order of increasing cost. The subset is partitioned into $n'-1$ subsets with the ith subset containing all spanning trees satisfying the previous conditions plus e_i' excluded and $\{e_1', \cdots, e_{i-1}'\}$ included.

The above discussion seems to imply that every time the next smallest spanning tree is chosen, a large number of replacement edges must be found. However, the determination of some replacement edges may be delayed, as the next lemma suggests. This will allow us to realize our improved strategy when K is small.

Lemma 4. Let T be a minimum spanning tree of graph (V,E). Let f_i and f_j be the replacement edges for e_i and e_j, resp., in T, and let $\hat{f}$ be the replacement edge for e_j in the minimum spanning tree $T-e_i+f_i$ in $(V,E-e_i)$. If $c(f_i)-c(e_i) < c(f_j)-c(e_j)$, then

$$c(T-e_i+f_i-e_j+\hat{f}) > c(T-e_j+f_j)$$

Proof. Since T is a minimum spanning tree, $c(f_i) > c(e_i)$. Thus the proof reduces to showing that $c(\hat{f}) \geq c(f_j)$. If $\hat{f} = f_j$, then we are done. Otherwise, e_j must be on the cycle induced by f_i in T. Since e_j has a replacement edge of f_j in T chosen among edges including f_i, $c(f_i) \geq c(f_j)$. Since $\hat{f} \neq f_j$, $T-e_i+\hat{f}$ is a spanning tree. Since e_i has replacement edge f_i chosen instead of $\hat{f}$, $c(\hat{f}) > c(f_i)$. The lemma then follows. ▫

Our approach is as follows. First use a fast algorithm to find the minimum spanning tree T_1. Generate our data structure for T_1. For each tree edge, find its replacement edge, using the algorithm in [T2]. Each such swap infers a spanning tree. Name each spanning tree, label it with a reference to T_1 and the swap that generates it. Create a heap on the costs of these spanning trees. Set up a list L_1 of such trees resulting from T_1 that have already been chosen. Initially, L_1 is empty.

We then iterate the following step until $K-1$ additional spanning trees have been chosen. Suppose $i-1$ trees have already been chosen. Select the minimum value out of the heap. The corresponding spanning tree will be T_i. Let T_j be the spanning tree from which T_i is generated, e_i the edge removed, and f_i the replacement edge. Generate our data structure for T_i from that of T_j, setting the cost of e_i in the graph to be ∞. Traverse the list L_j. For each T_l on the list, determine the replacement edge for e_i in T_l. Name the corresponding spanning tree, label it with T_l and the new swap, and enter its cost into the heap. Now add T_i to the list T_j. Find the replacement edge for f_i in T_i. Name the new spanning tree, and as before label it and enter its cost into the heap. Set up a list L_i, initially empty. Repeat until $i=K$.

The correctness of the above algorithm may be seen as follows. The inclusion-exclusion strategy is being implemented, with an edge excluded by setting its cost to ∞ in the data structure that is the source of the appropriate subset of spanning trees. Inclusion of edges is enforced by the mechanism of building and traversing the lists $\{L_j\}$. The only edges in tree T_i that can be swapped out are f_i and those edges involved in swaps with respect to T_j that are more expensive than (e_i,f_i). Lemma 4 guarantees that a spanning tree arising from such a swap need not be examined until the corresponding list has been traversed during execution of the algorithm.

We now consider the time complexity of our algorithm. Finding the minimum spanning tree requires $O(m \log\log_{(2+m/n)} n)$ time. The time to find all replacement edges in the minimum spanning tree is $O(m\alpha(m,n))$, which is dominated by the above time. We bound the iteration time as follows. Every time an iteration is performed, the length of some list is increased by one. The total number of elements on all lists when tree T_i is chosen is $i-2$. Thus at most $i-1$ replacement edges in various trees must be found after selecting tree T_i. Since the time to find a replacement edge is $O(\sqrt{m})$, the ith iteration requires $O(i\sqrt{m})$ time. For $K-1$ iterations, this time is $O(K^2\sqrt{m})$. When K is $o(\sqrt{m})$ the resulting time is $o(Km)$.

As presented, the time and space to generate the T_i's is $O(Km)$, since the space for our basic data structure is $O(m)$. However it is possible to save space in the following way. Since the time required to generate our data structure for T_i by modifying the structure for T_j is $O(\sqrt{m})$, the number of new nodes is $O(\sqrt{m})$. The idea is to not destroy any nodes of the structure for T_j, but simply share the appropriate subtrees. This reduces the space to $O(m+K\sqrt{m})$, which is $O(m)$ when K is $O(\sqrt{m})$.

Theorem 8. The K smallest spanning trees of a graph can be found in $O(m \log\log_{(2+m/n)} n + K^2\sqrt{m})$ time and $O(m+K\sqrt{m})$ space. ▪

As discussed in the introduction, this result is better than previous results in [KIM] whenever K is $o(\sqrt{m})$ and $\omega(\log\log_{(2+m/n)} n)$. Our results are better than those in [H12] whenever $K\sqrt{m}$ is $o(n(\log n)^2)$ or K is $o(m^{1/4}(\log n)^{1/2})$. If the given graph is planar, then our corresponding structures for planar graphs should be used. These results are better than the corresponding ones for [KIM] whenever K is $o(n/(\log n)^2)$. They are also never worse than those in [H12], and are better when K is $o(n)$.

Theorem 9. The K smallest spanning trees of a planar graph can be found in $O(n + K^2(\log n)^2)$ time and $O(n + K(\log n)^2)$ space. ▪

References

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

[CT] D. Cheriton and R. Tarjan, Finding minimum spanning trees, *SIAM J. Comput. 5* (1976) 724-742.

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

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

[F] G. N. Frederickson, Data structures for on-line updating of minimum spanning trees, *Proc. 15th ACM Symp. on Theory of Computing*, Boston (April 1983) 252-257.

[FS1] G. N. Frederickson and M. A. Srinivas, Data structures for updating constrained spanning trees, abstract (1983).

[FS2] G. N. Frederickson and M. A. Srinivas, Data structures for on-line updating of matroid intersection solutions, *Proc. 16th ACM Symposium on Theory of Computing*, Washington, D. C. (April 1984) 383-390.

[G] H. N. Gabow, Two algorithms for generating weighted spanning trees in order, *SIAM J. Computing 6*, 1 (March 1977) 139-150.

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

[Hl2] Dov Harel, personal communication (1983).

[Hy] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass. (1969).

[KIM] N. Katoh, T. Ibaraki, and H. Mine, An algorithm for finding *K* minimum spanning trees, *SIAM J. Computing 10*, 2 (May 1981) 247-255.

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

[OV] M. H. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comp. Sys. Sci. 23*, 2 (October 1981) 166-204.

[ST] D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comp. Sys. Sci. 26* (1983) 362-391.

[SP] P. M. Spira and A. Pan, On finding and updating spanning trees and shortest paths, *SIAM J. Comput. 4*, 3 (September 1975) 375-380.

[T1] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. ACM 22*, 2 (April 1975) 215-225.

[T2] R. E. Tarjan, Applications of path compression on balanced trees, *J.ACM* **26** (1979) 690-715.

[W] D. J. A. Welsh, Matroid Theory, Academic Press, New York (1976).

[WL] D. E. Willard and G. Lueker, A transformation for adding range restriction capability to data structures, to appear in *J.ACM*.

[Y] A. C. Yao, An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees, *Inf. Proc. Lett.* *4* (1975) 21-23.

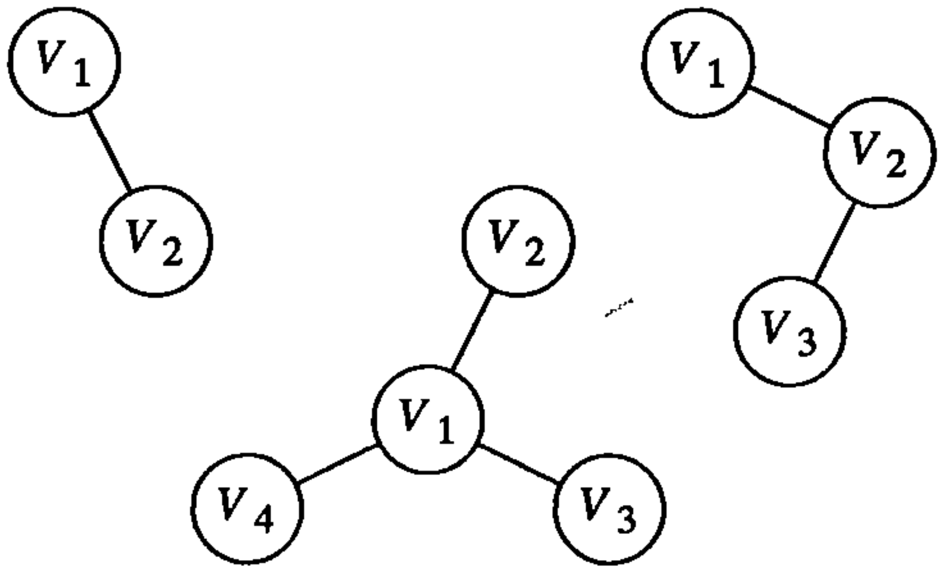


Figure 1. The allowable topologies for vertex clusters that may be unioned together.

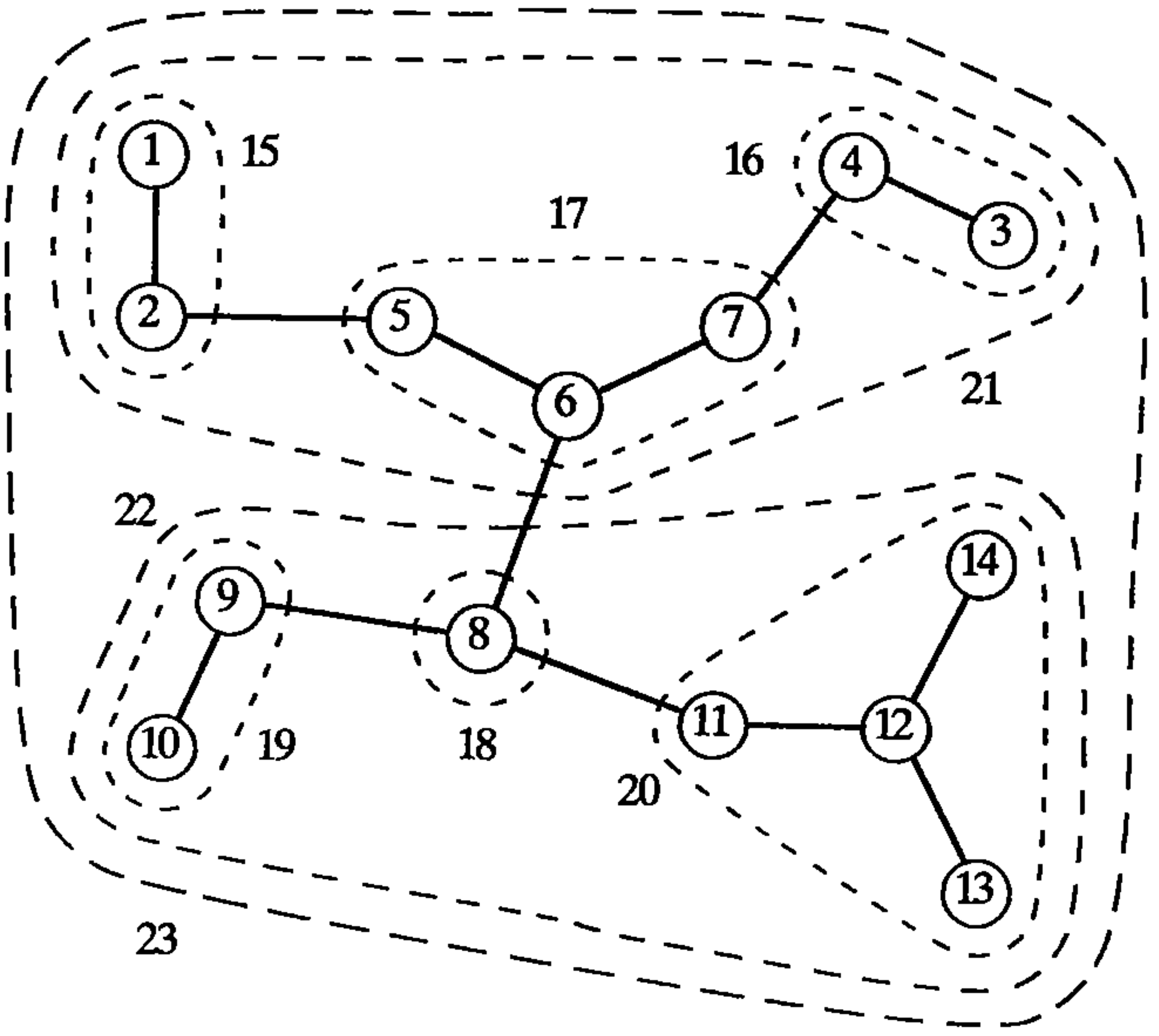


Figure 2. A multilevel partition of the vertices in a spanning tree.

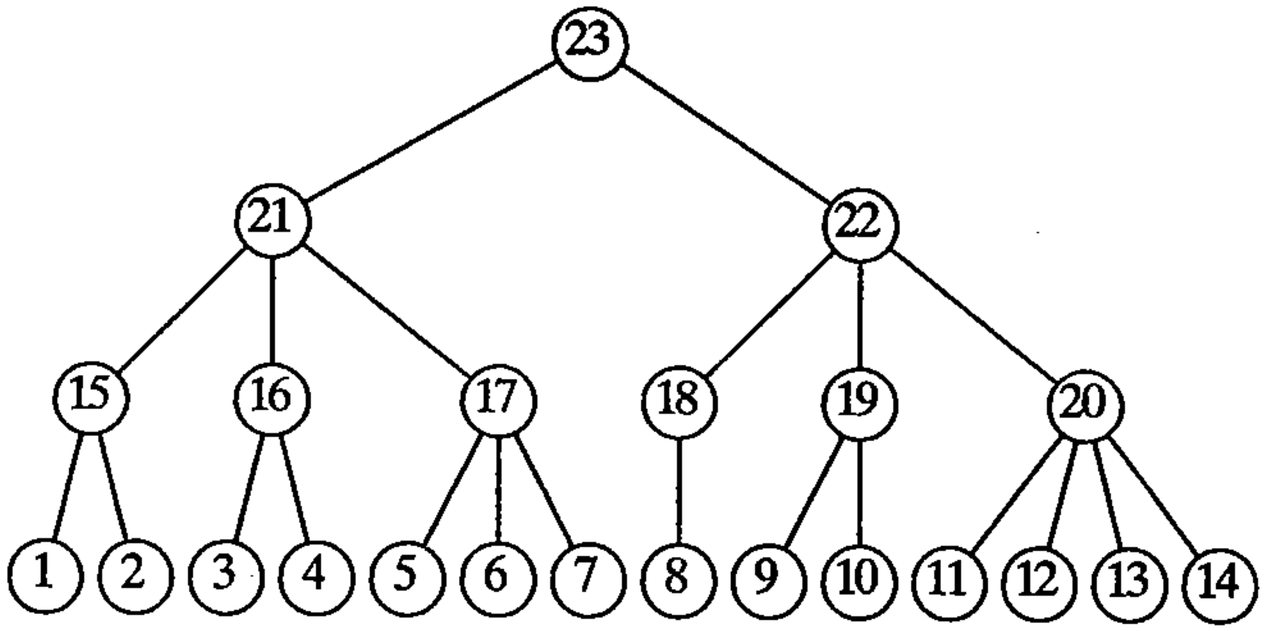
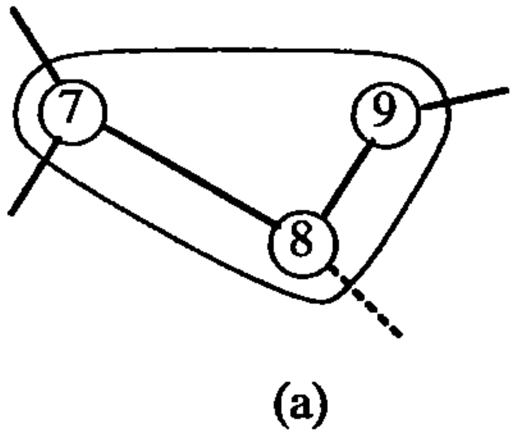
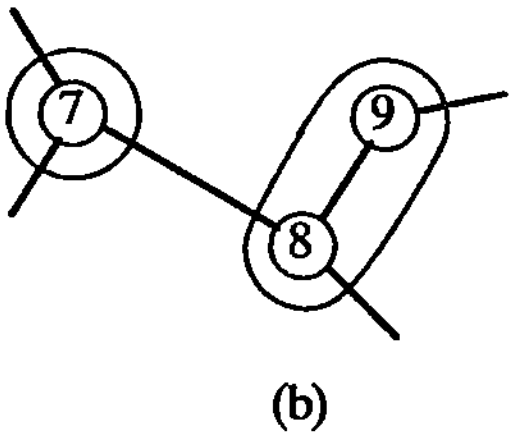
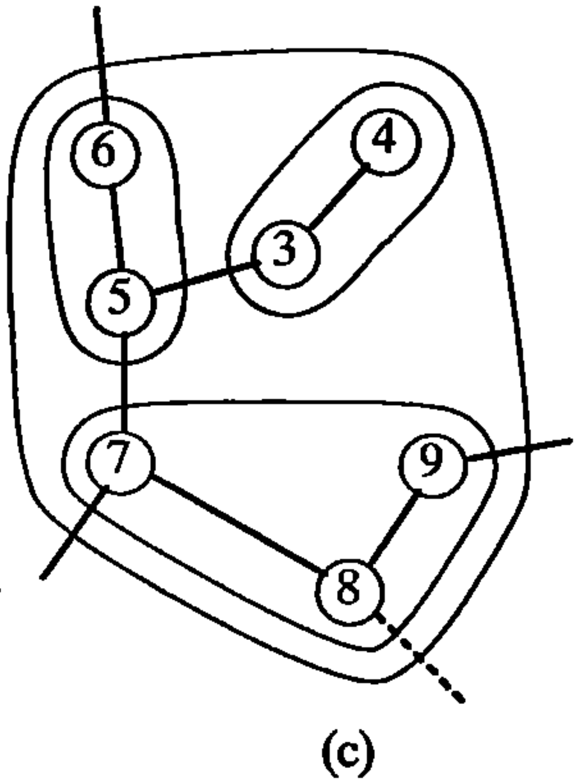


Figure 3. The topology tree corresponding to the topological partition in Figure 2.







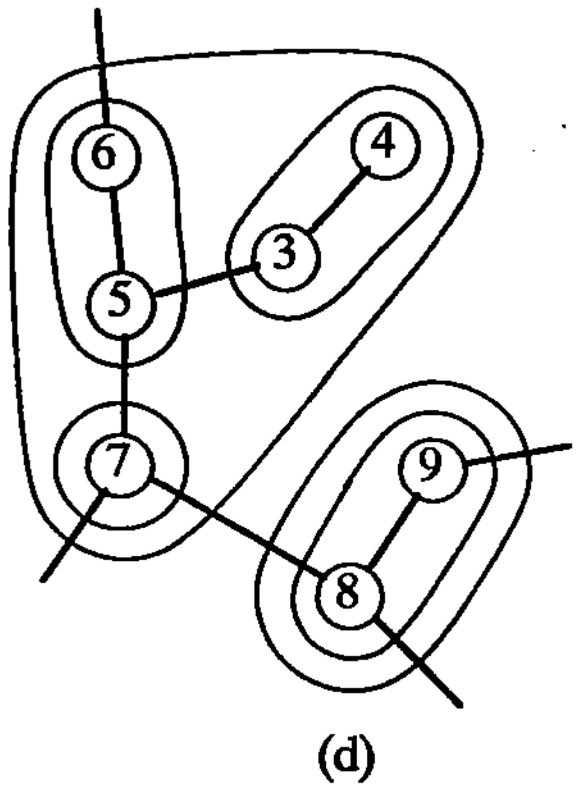


Figure 4. An example of external degree increasing from 3 to 4.

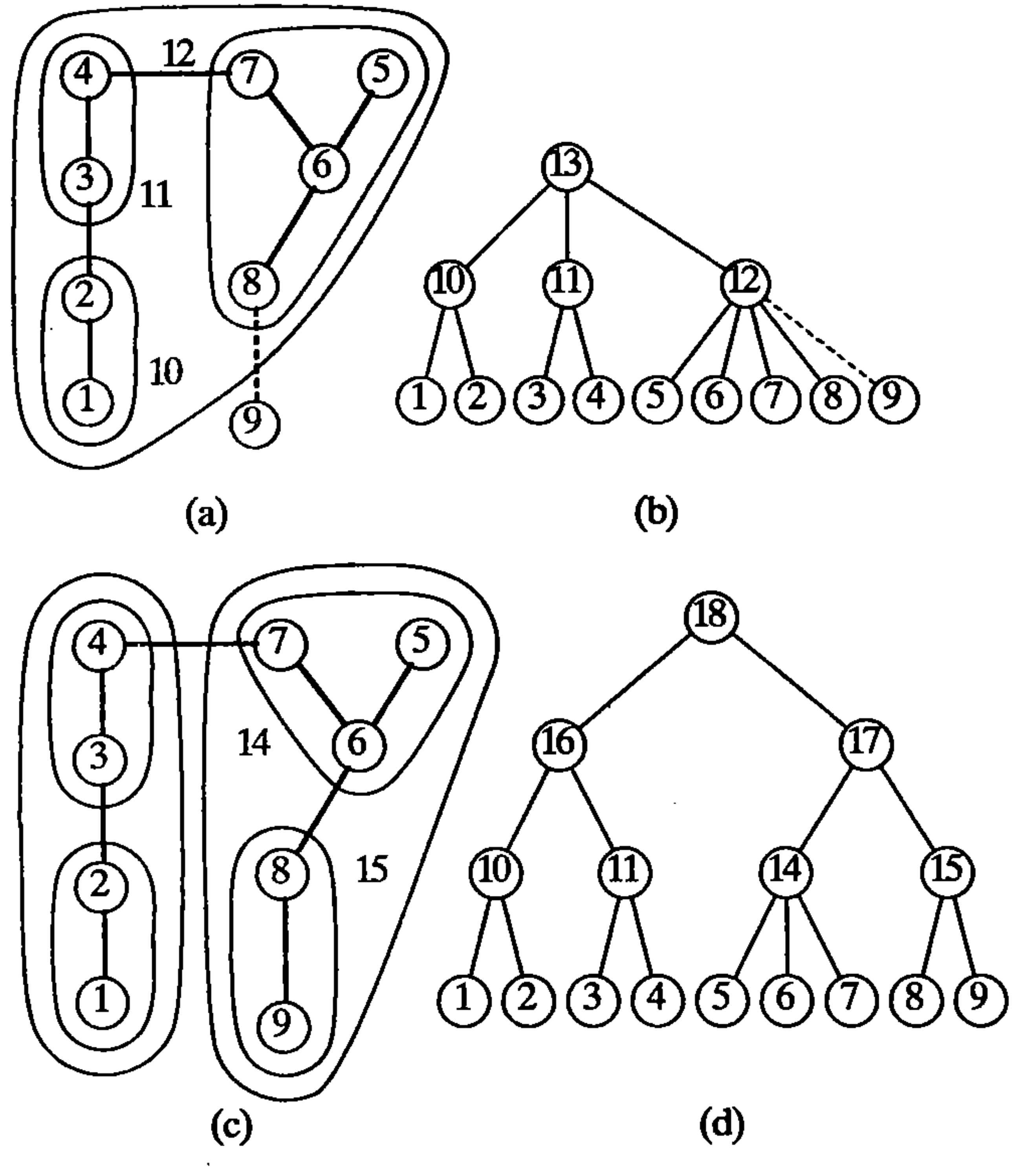


Figure 5. An example of inserting a tree edge and merging two topology trees.

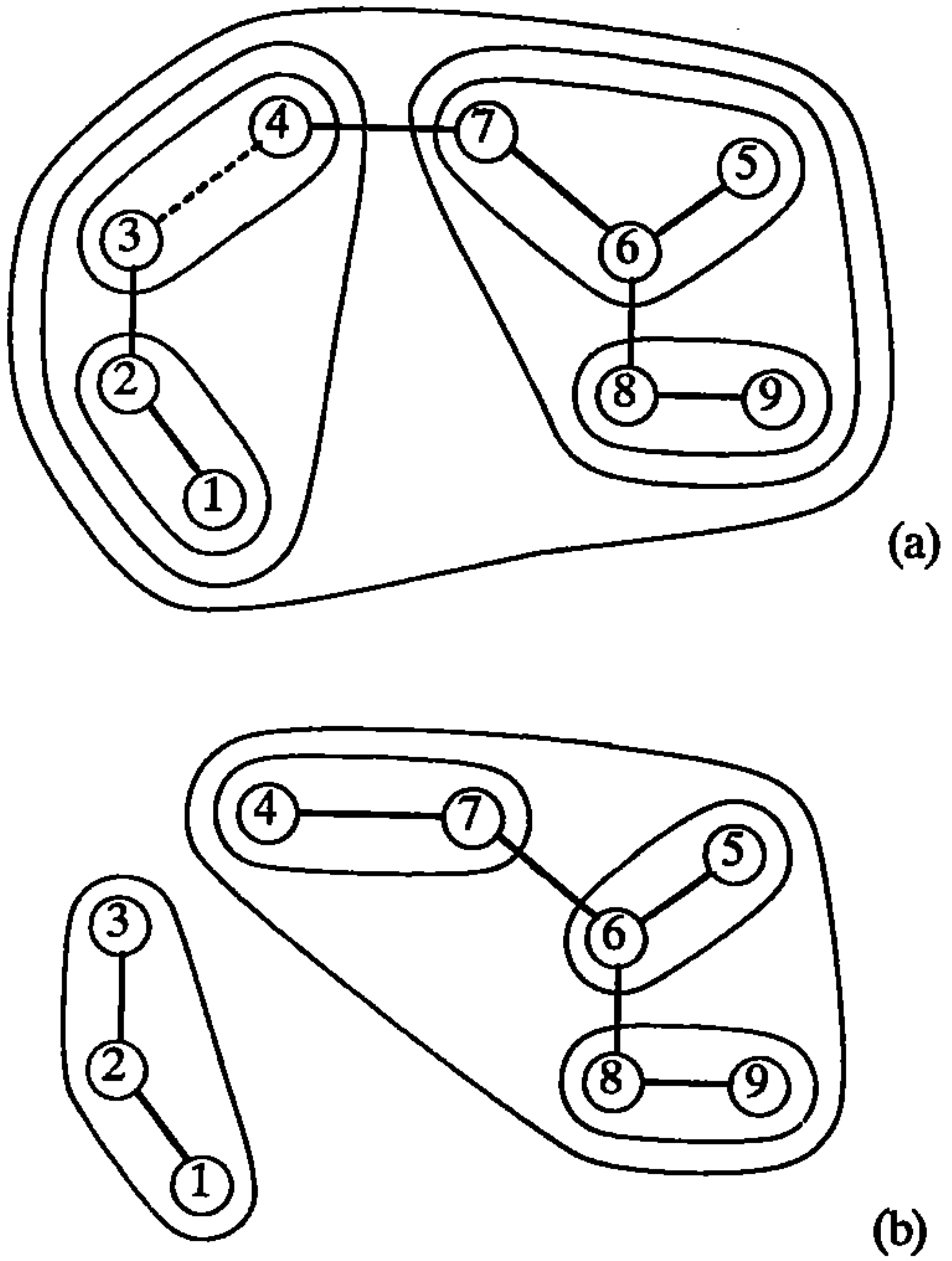


Figure 6. An example of deleting a tree edge and splitting a multilevel partition.

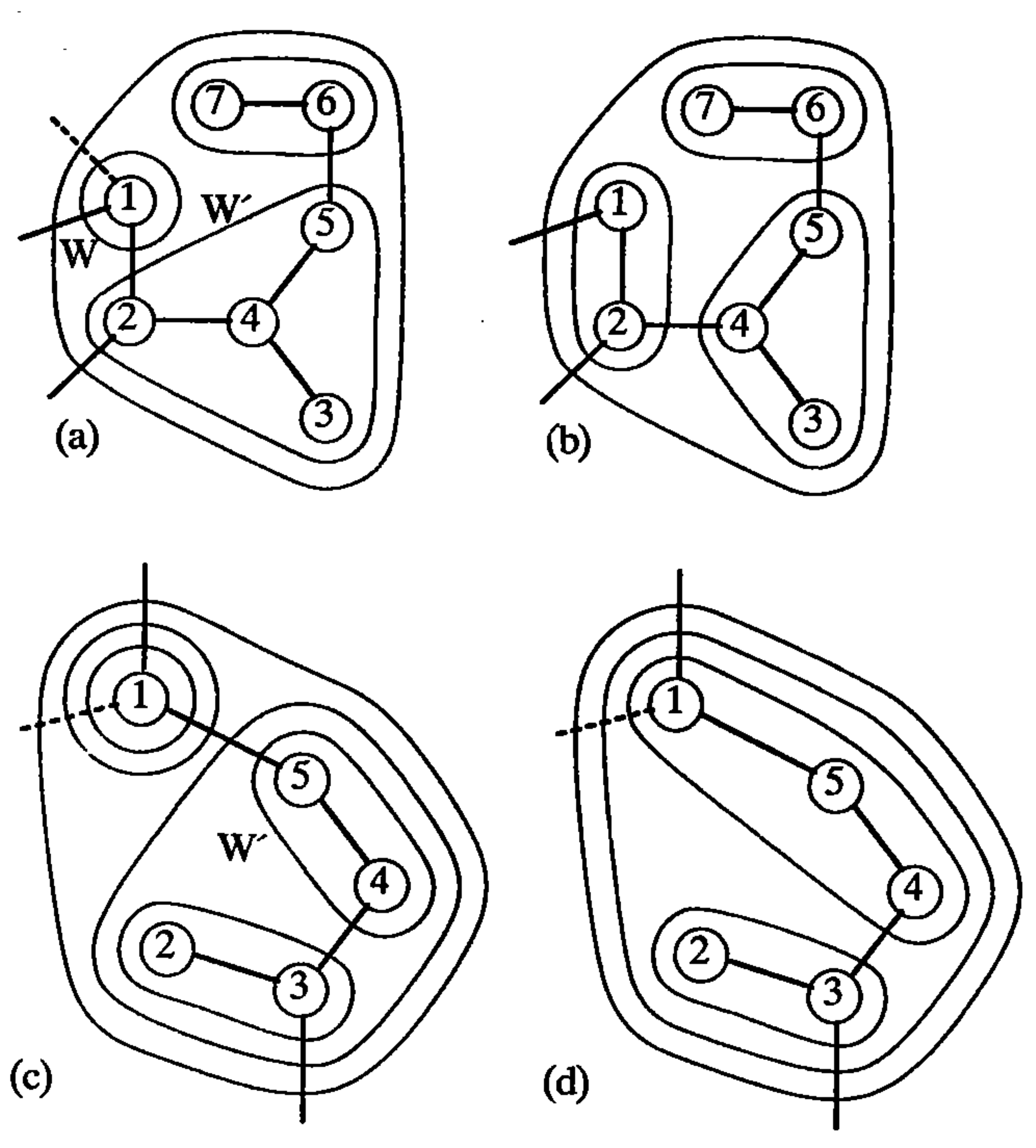


Figure 7. Examples of external degree decreasing from 3 to 2.

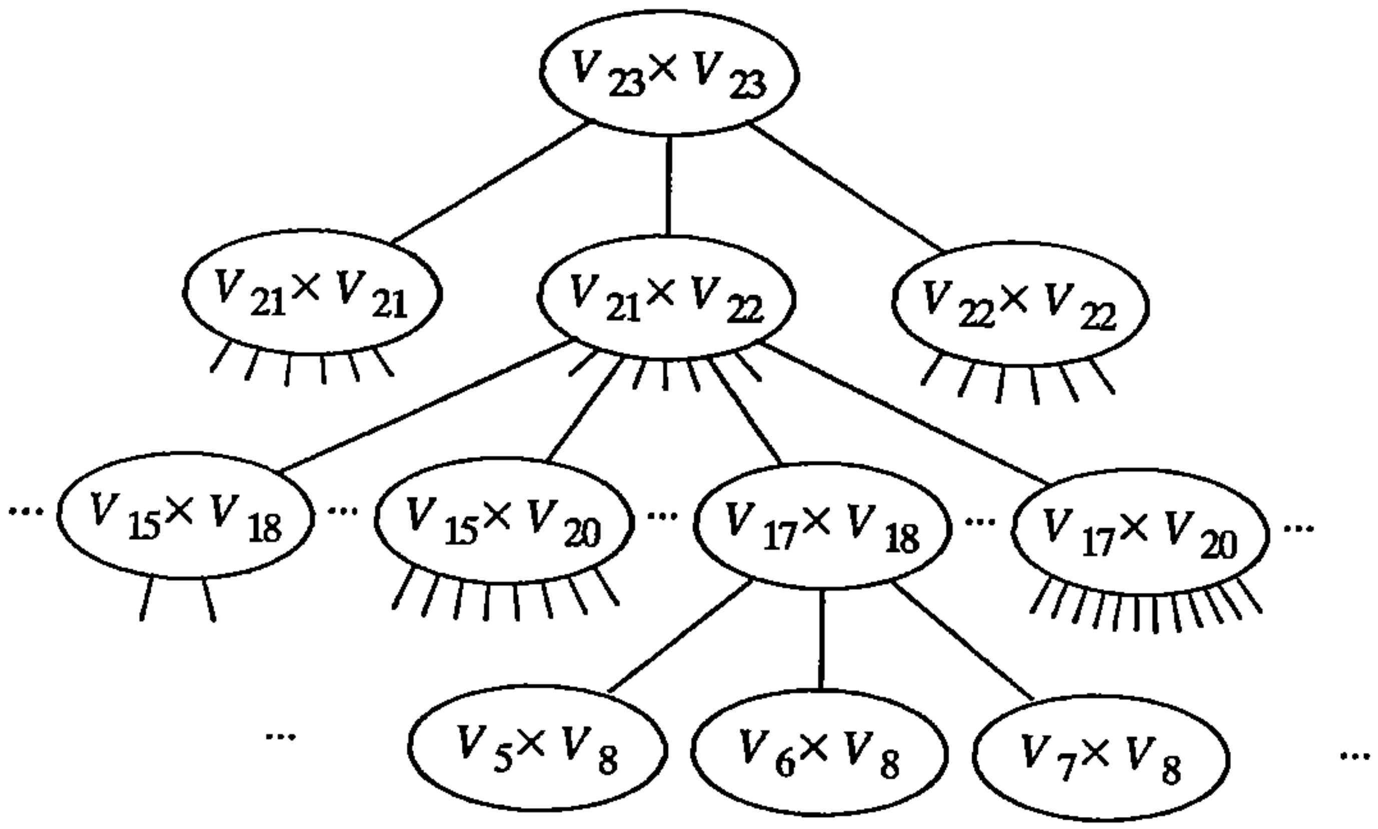


Figure 8. A portion of the 2-dimensional topology tree corresponding to the topology tree in Figure 3.

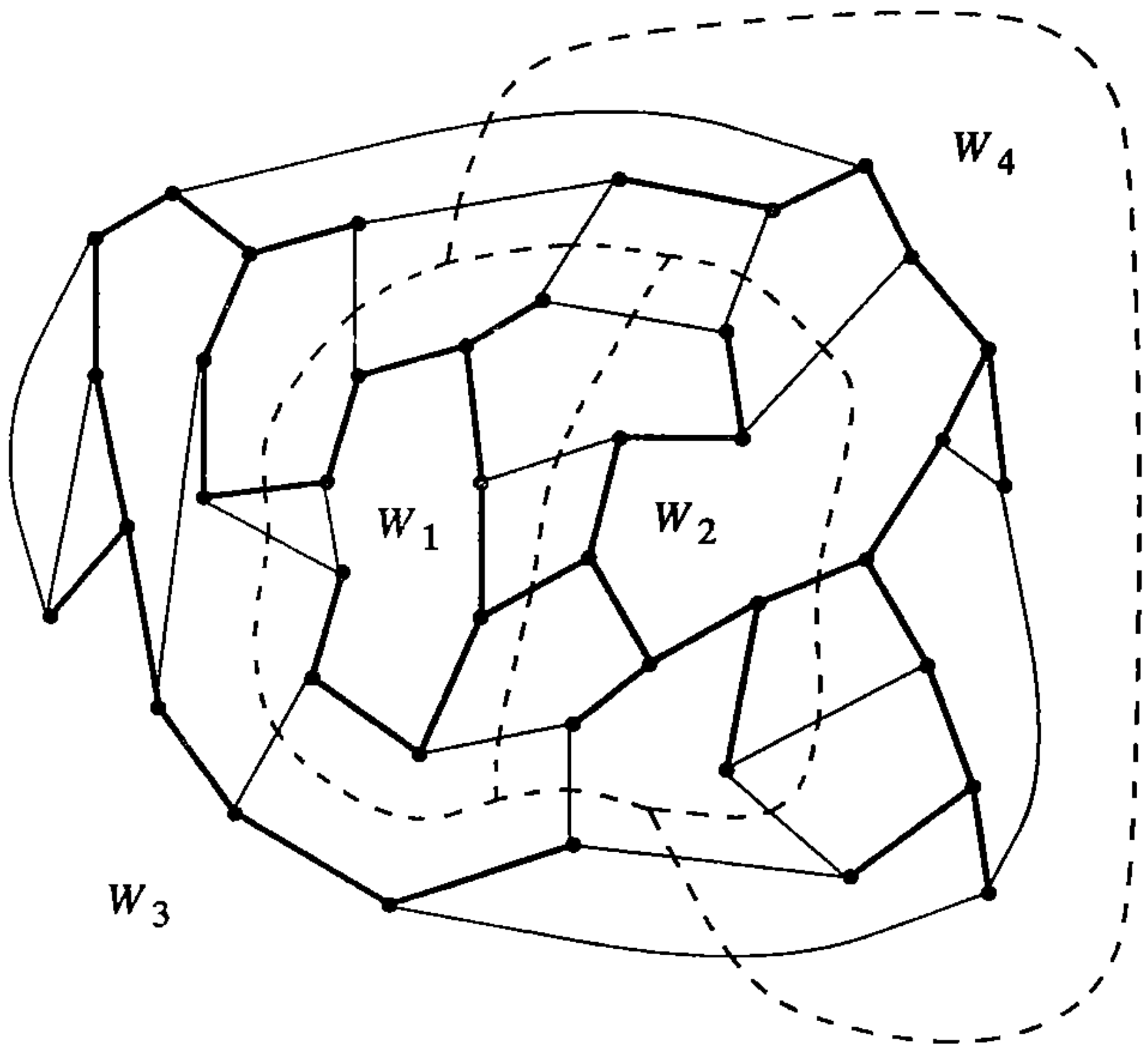


Figure 9. A planar graph, with the tree edges shown in bold, the nontree edges in solid, and the vertices grouped with dashed lines.

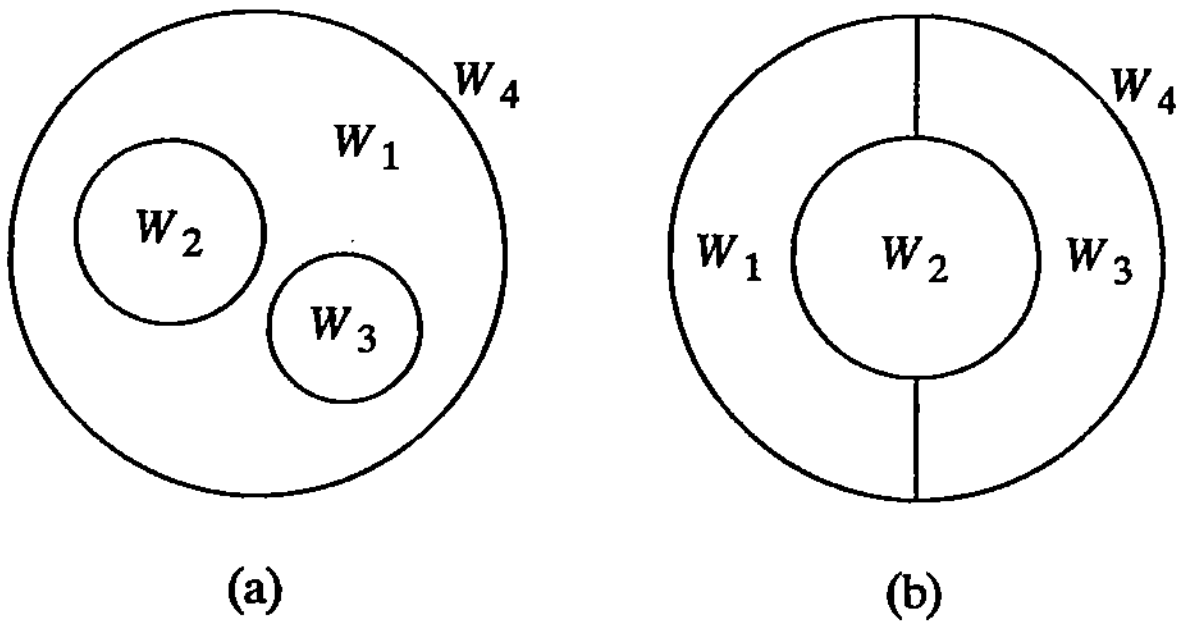


Figure 10. Examples of relationships between regions.

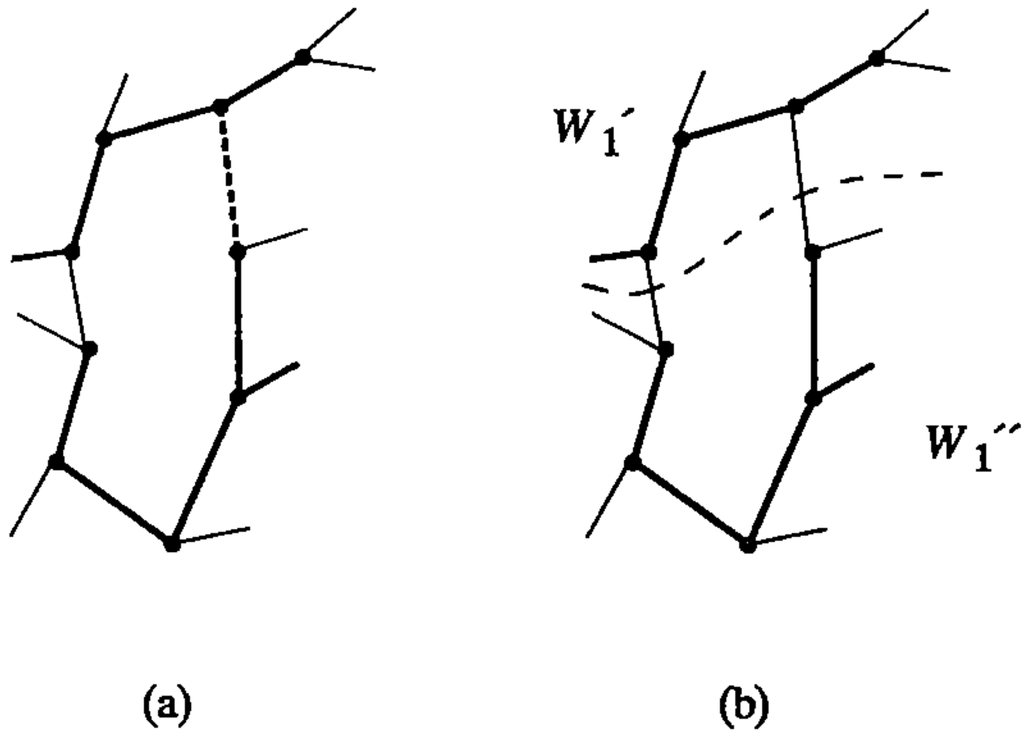


Figure 11. Splitting a planar region.