Data structures for two-edge connectivity in planar graphs

John Hershberger

Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301-1044, USA

Monika Rauch

Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544-2087, USA

Subhash Suri

Bell Communications Research, 435 South Street, Morristown, NJ 07960-1910, USA

Abstract


We present a data structure for maintaining 2-edge connectivity information dynamically in an embedded planar graph. The data structure requires linear storage and preprocessing time for its construction, supports online updates (deletion of an edge or insertion of an edge consistent with the embedding) in $O(\log^2 n)$ time, and answers a query (whether two vertices are in the same 2-edge-connected component) in $O(\log n)$ time. The previous best algorithm for this problem requires $O(\log^3 n)$ time for updates.

1. Introduction

Connectivity in graphs is an important class of problems that has received considerable attention since the early work of Hopcroft and Tarjan, who designed linear-time algorithms for computing bi- and tri-connected components of a graph [1, 12, 16]. NC algorithms for 2- and 3-connectivity have been proposed in [14, 18]. In recent years, attention has turned toward dynamic algorithms for graph connectivity. These algorithms maintain connectivity information as the underlying graph is modified by the insertion/deletion of edges or vertices.
A standard model for dynamic graph connectivity involves a sequence of inter-mixed updates and queries: an update inserts or deletes an edge and a query asks for certain connectivity information. The goal is to build a data structure that can support both these operations in sublinear time. If only insertions or only deletions are permitted, then the model is called semi-dynamic, and if both insertions and deletions are allowed, the model is called fully dynamic. In this paper, we present a fully dynamic algorithm for maintaining 2-edge connectivity in embedded planar graphs. Our data structure requires linear space and preprocessing time, supports 2-edge-connectivity queries in $O(\log n)$ time, and requires $O(\log^2 n)$ update time to insert or delete an edge. A 2-edge-connectivity query specifies two query vertices and asks whether there exist two edge-disjoint paths connecting them. Insertions must be consistent with and specified relative to the embedding.

There has been a considerable amount of work on dynamic graph algorithms in recent years. The 1-connectivity problem is to maintain the connected components of a graph under insertion/deletion of edges. If only insertions are allowed, then this problem reduces to the disjoint set union problem, and hence a sequence of $n$ insertions and queries can be processed in $O(nx(n))$ time, using the union-find data structure [17]. The problem becomes significantly harder if both insertions and deletions are allowed, and the best result to date for general graphs is an algorithm due to Eppstein et al. [2] that takes $O(\sqrt{n} \log(m/n))$ time per update, where $m$ is the number of edges in the graph. Eppstein et al. have improved the time bound to $O(\sqrt{n})$; this improvement will appear in the journal version of their paper. (It is worth pointing out that the 1-connectivity problem, with both insertions and deletions, is quite different from the Union-Find-Deunion problem, for which a logarithmic update procedure is known [8].) If the graph is planar and embedded, then a result of Eppstein et al. [4] achieves $O(\log n)$ time per operation for the 1-connectivity problem.

The semi-dynamic versions of the 2-edge connectivity and the 2-vertex connectivity problems were considered by Westbrook and Tarjan [19]; they showed that a sequence of $n$ insertions and queries can be processed in $O(nx(n))$ time. Galil and Italiano [7] managed to obtain a sublinear time algorithm for fully dynamic 2-edge connectivity. For general graphs, their algorithm takes $O(m^{2/3})$ time per operation (update or query), where $m$ is the current number of edges in the graph. For planar graphs, their time complexity improves to $O(\sqrt{n} \log \log n)$. Soon afterwards, Frederickson [6] improved the time bound in [7] to $O(\sqrt{m})$ per operation. Frederickson also presented a faster algorithm for planar embedded graphs, with query time $O(\log n)$ and update time $O(\log^3 n)$.

Rauch [15] gave a fully dynamic algorithm for maintaining 2-vertex connectivity. An update operation takes time $O(m^{2/3})$. In embedded planar graphs the running time reduces to $O(\sqrt{n} \log n)$. Eppstein et al. [2] developed a general technique that speeds up the algorithms by Frederickson for fully dynamic connectivity and 2-edge connectivity to time $O(\sqrt{n} \log(m/n))$ per update operation. They also achieve a time of $O(\log(m/n))$ per update operation for fully dynamic biconnectivity. (The journal version of their paper removes the $\log(m/n)$ factor.)
The main result of our paper improves the embedded planar graph result of Frederickson for 2-edge connectivity by a factor of $\log n$. Our data structure supports online updates (insertion and deletion of an edge) in $O(\log^2 n)$ time, and answers a query whether two vertices are in the same 2-edge-connected component in $O(\log n)$ time.\(^1\) The data structure is based on a spanning tree of the graph. It can be built in linear time and requires linear storage. The machine model is a RAM with word size $\Omega(\log n)$.

Our algorithm introduces two new concepts, *edge bundles* and *coverage graph recipes*, which appear to be of general interest and may find applications in other planar graph algorithms. The former is a method for collapsing and manipulating edges that belong to the same equivalence class, for a given partitioning of vertices. The second is a method for compressing and uncompressing portions of a planar graph, for an efficient traversal of a data structure called a topology tree. These concepts are the keys to our improved data structure.

This paper is organized in nine sections. In Section 2, we introduce the basic concepts; in Section 3, we describe the topology tree and prove a structural lemma about partially expanded topology trees. In Sections 4 and 5, we introduce edge bundles and coverage graphs. Section 6 describes the concept of recipes for building coverage graphs, and Sections 7 and 8 describe our query and update procedures. Finally, we conclude in Section 9 with some discussion and directions for future research. A preliminary version of this result was published in [11].

2. Preliminaries

Let $G=(V,E)$ be an undirected planar graph, embedded in the plane. The initial embedding of $G$ remains fixed throughout the course of the algorithm, and all the updates must respect the embedding. We let $n$ denote the number of vertices of $G$ and $m$ the number of edges; the number of edges changes with updates, but planarity implies that $m \leq 3n - 6$.

We perform a standard transformation on $G$ to convert it into a graph with maximum vertex-degree 3.\(^2\) Suppose $v \in V$ is a vertex of degree $d > 3$, adjacent to vertices $u_1, u_2, \ldots, u_d$ in this cyclic order. In the transformed graph, the vertex $v$ is replaced by a cycle $(v_1, v_2, \ldots, v_d)$, and the edge $(v, u_j)$, for $1 \leq j \leq d$, is replaced by the edge $(v_j, u_j)$. In order to maintain the vertex identity, we label exactly one vertex on the cycle to be $v$, and leave the others unlabeled. At $v$ we also store an ordered list of all the other vertices on the cycle. Observe that the transformed graph has $O(m)$ vertices.

---

\(^1\) After the conference publication of the present paper, Eppstein et al. [3] developed an algorithm for 2-edge connectivity in nonembedded planar graphs whose running time is $O(\log n)$ per query and edge insertion, and $O(\log^2 n)$ per edge deletion.

\(^2\) To maintain textual consistency and readability, we adopt the convention that vertex degrees are written as numerals ("1, 2, 3" rather than "one, two, three") in this paper.
and that it inherits the 2-edge-connectivity properties of the original graph: two vertices are 2-edge connected in $G$ if and only if they are 2-edge connected in the transformed graph [10]. Thus, from now on, we assume that $G$ has maximum vertex degree 3. Furthermore, since we will be primarily concerned with edge connectivity, we use the term 2-connectivity instead of 2-edge connectivity.

Our data structure is based on a spanning tree of $G$. The updates to the graph may force us to change this spanning tree, but between two successive updates we always maintain a spanning tree of $G$. We use this tree to build a hierarchical decomposition of $G$, define unique paths between two vertices, and other tasks. We use the notation $T$ for the current spanning tree of $G$, and the notation $\pi(u,v)$ for the unique path in $T$ between two vertices $u$ and $v$.

Our first lemma establishes the important connection between 2-connectivity and edge-covering. We say that a tree edge $(x,y)$ is covered by a nontree edge $(u,v)$ if $(x,y)$ lies on $\pi(u,v)$. The edge $(x,y)$ is covered if there exists a nontree edge that covers it. A tree edge is called a bridge if it is not covered. It is easy to see that the two endpoints of a bridge lie in different 2-connected components. See Fig. 1.

![Fig. 1. Coverage. Tree edges are solid and nontree edges are dashed. Covered tree edges are drawn heavy. Thin solid edges are bridges.](image-url)
Lemma 2.1. Two vertices $u$ and $v$ in $G$ are 2-edge connected if and only if every edge of the tree path $\pi(u,v)$ is covered.

Proof. If an edge of $\pi(u,v)$ is not covered, then its removal disconnects the graph and thus $u$ and $v$ are not 2-edge connected.

If $u$ and $v$ are not 2-edge connected, then there exists an edge $e$ that is used by all paths from $u$ to $v$ (and thus also by the tree path). If $e$ were covered, then there would exist a path from $u$ to $v$ that does not use $e$. Thus, $e$ is an edge on $\pi(u,v)$ that is not covered. □

3. Topology trees

We build a hierarchical representation of $G$ based on the spanning tree $T$. The representation is a tree, called the topology tree, that has depth $O(\log n)$ [5, 6]. Each level of the topology tree partitions the vertices of $G$ into connected subsets called clusters. Two clusters are said to be adjacent if they are joined by an edge in the spanning tree $T$. The external degree of a cluster is the number of tree edges with exactly one endpoint inside the cluster. Our clusters will have maximum external degree 3. We now describe the rules for building the topology tree.

Each cluster has a level associated with it. Each vertex of $T$ is a level-0 cluster. A higher-level cluster is either an unmodified cluster of the next lower level, or the union of two adjacent lower-level clusters subject to an external degree constraint. More precisely, a cluster at level $i$, for $i > 0$, is formed by either

1. the union of two adjacent clusters of level $i-1$ such that the external degree of one cluster is 1 or the external degree of both is 2, or
2. one cluster of level $i-1$, if the previous rule does not apply.

A cluster of level $i-1$ belongs to exactly one cluster of level $i$, thus ensuring that the vertices of $G$ are partitioned at each level. Note that our cluster-forming scheme implements a locally greedy heuristic and it does not always perform the maximum number of cluster unions. However, this locally greedy method of forming clusters is powerful enough for our purposes.

The topology tree has a node corresponding to each cluster that we form. Level-0 clusters correspond to leaves of the topology tree, and higher-level clusters correspond to internal nodes of the topology tree. The ancestor-descendant relationships in the tree encode the cluster containment information: the node corresponding to a cluster $C$ at level $i$ is the parent of the (one or two) clusters of level $i-1$ whose union produced $C$. The greedy method of forming clusters reduces the number of clusters at each level by a constant fraction, thus ensuring that the height of the topology tree is $O(\log n)$.

The following lemma is due to Frederickson [6].

Lemma 3.1 (Frederickson [6]). The number of clusters at level $i+1$ is at most $5/6$ times the number of clusters at level $i$. 
Corollary 3.2. The topology tree for the graph $G$ has height $\Theta(\log n)$.

When $G$ is updated, the spanning tree $T$ may change, and hence the topology tree may need restructuring. All restructuring operations follow a common routine: we break up all clusters containing some constant number of vertices, and then recombine the unbroken clusters, perhaps in different combinations. In terms of the topology tree, this corresponds to removing all the nodes along a constant number of root-to-leaf paths, transplanting certain subtrees, and rebalancing the tree. To facilitate our discussion, we introduce the following terminology. To expand the topology tree at a vertex $u$ means that we break up all the clusters that contain $u$; observe that the clusters containing $u$ correspond precisely to the nodes lying along the path between $u$ and the root of the topology tree. We denote by $C_i(u)$ a cluster at level $i$ and by $C_i$ the unique cluster at level $i$ that contains the vertex $u$. A level $i-1$ cluster $C_{i-1}$ contained in $C_i$ is called a subcluster of $C_i$.

Let us consider the expansion of the topology tree at a vertex $u$. We start from the root, whose level is defined to be $k$, and walk down the tree, expanding the cluster $C_i(u)$ at each level. If $C_i(u)$ has only one child $C_{i-1}(u)$, we replace $C_i(u)$ by its child. Otherwise, $C_i(u)$ has two children, and we replace $C_i(u)$ by its two children and add appropriate tree edges linking the children to each other and to other clusters. In either case, we then recursively expand $C_{i-1}(u)$. The expansion stops when we reach the leaf-cluster representing the vertex $u$.

At each level, the expansion process removes one cluster and creates at most two clusters at the next lower level. It follows that after the expansion stops, there is at most one cluster for each level $i$ above the leaf-level ($0 < i \leq k$), and at most two at the leaf-level, one of which is the vertex $u$. We use the notation $K_i$ for the cluster that exists at level $i$ after this expansion, if any; at the leaf-level, $K_0$ denotes the cluster containing the vertex other than $u$, if any exists. See Fig. 2. The expansion procedure results in a tree of clusters linked by edges of $T$, called the expansion tree. The following lemma shows that the tree is well structured; in particular, the sum of the level differences between neighboring clusters is $O(\log n)$.

Lemma 3.3 (Structural lemma). Let $\overline{T}$ be the expansion tree obtained by expanding the topology tree at a vertex $u$. For any edge $e$ of $\overline{T}$, let $\text{ldiff}(e)$ be the absolute value of the
difference between the levels of the clusters containing e’s endpoints. Then the sum of \( \text{ldiff}(e) \) over all edges \( e \in T \) is \( O(\log n) \).

**Proof.** Let \( S_i \) be the sum of the level differences between pairs of neighboring clusters that have already been generated at level \( i \) of the expansion (i.e. clusters \( K_j \) with \( j \geq i \)). Let \( S'_i \) be the sum of the level differences between \( C_i(u) \) and its neighbor clusters. We prove by induction on \( i \) that at each step of the expansion process, \( S_i + S'_i \) is at most \( 3(k - i - 1) \).

The base case is easy. After one expansion, \( S_i = 0 \) (because there is only one cluster \( K_j \) with \( j \geq i \)), and \( S'_i = 0 \) (because \( C_i(u) \) is adjacent to only one cluster, \( K_{k-1} \), which has level \( k - 1 = i \)). Thus, \( S_i + S'_i = 0 = 3(k - i - 1) \).

Now consider expanding \( C_{i+1}(u) \) to produce \( C_i(u) \), for \( i < k - 1 \). We show that \( \Delta(S_i + S'_i) \equiv (S_i + S'_i) - (S_{i+1} + S'_{i+1}) \) is at most \( 3 \).

If \( C_i(u) = C_{i+1}(u) \), either because \( C_{i+1}(u) \) has external degree 3 or because \( C_i(u) \) was unable to union with a neighbor during the construction of the topology tree, then no new cluster \( K_i \) is produced, and \( S_i = S_{i+1} \). Since \( C_i(u) \) has the same neighbors as \( C_{i+1}(u) \), we have \( S'_i \leq S'_{i+1} + 3 \). Thus, \( \Delta(S_i + S'_i) \leq 3 \).

If \( C_{i+1}(u) \) splits into \( K_j \) and \( C_i(u) \), we consider two cases based on the external degree of \( C_{i+1}(u) \).

If \( C_{i+1}(u) \) has external degree 2, then let \( K_j \) and \( K_h \) be the neighbors of \( C_{i+1}(u) \), and suppose first that \( K_i \) is adjacent to \( K_j \). Then \( S_i = S_{i+1} + (j - i) \). Because \( S'_{i+1} = (j - i - 1) + (h - i - 1) \), and \( S'_i = (i - i) + (h - i) \), we have \( \Delta(S_i + S'_i) = 2 \). If \( K_i \) is not adjacent to either \( K_j \) or \( K_h \) (\( C_i(u) \) has degree 3), then \( S_i = S_{i+1} + 1 \) and \( S'_i = S'_{i+1} + 2 \), leading to the same bound on \( \Delta(S_i + S'_i) \).

If \( C_{i+1}(u) \) has external degree 1, let its neighbor be \( K_j \); then \( S'_{i+1} = j - i - 1 \). If the new cluster \( K_i \) lies between \( K_j \) and \( C_i(u) \), then \( S_i = S_{i+1} + j - i \) and \( S'_i = 0 \); otherwise, if \( C_i(u) \) lies between \( K_j \) and \( K_i \), we have \( S_i = S_{i+1} + 1 \) and \( S'_i = j - i \). In either case \( \Delta(S_i + S'_i) = 1 \).

We finish the expansion when we reach \( C_0(u) = u \). The final sum of level differences, which is \( S_0 + S'_0 \), is bounded by \( 3(k - 1) = O(\log n) \). \( \square \)

Now suppose we wish to expand at two vertices \( u \) and \( v \). Suppose that after expanding at \( u \), \( K_j \) is the unexpanded cluster that contains \( v \). We apply the same expansion procedure to \( K_j = C_j(v) \) as we applied to \( C_i(u) \). The same procedure lets us expand the topology tree at any constant number of vertices in \( O(\log n) \) time. The resulting partially expanded tree has a constant number of clusters of each level. The proof of Lemma 3.3 extends to this case as well, establishing the following more general lemma.

**Lemma 3.4.** Let \( T \) be an expansion tree obtained by expanding the topology tree at a constant number of vertices. Define \( \text{ldiff}(e) \) as in Lemma 3.3. Then the sum of \( \text{ldiff}(e) \) over all edges \( e \in T \) is \( O(\log n) \).
To execute a query \((u, v)\) or an update of an edge \((u, v)\), we first expand the topology tree at \(u\) and then at \(v\). In the case of a query, we answer the query and then merge the topology tree together in the same way as we expanded it. If we perform an update, edges and vertices are inserted and deleted (vertices are inserted or deleted to make sure that each vertex still has degree at most 3), and we may have to expand further at additional vertices. This changes the spanning tree and leads to a different topology tree, since only clusters that are neighbors in the spanning tree can be combined into larger clusters.

We merge back the clusters using a locally greedy heuristic: whenever possible, we merge two adjacent clusters of the lowest level, until the final topology tree is obtained; the merges respect the two clustering rules given earlier (see the full version of [6] for details). This process may force further expansion of some clusters. The number of expansions in each cluster is proportional to the sum of the differences between the level of the cluster and the levels of its neighbors. By Lemma 3.4, the total number of additional expansions is \(O(\log n)\).

4. Edge bundles

In the previous section, we described a hierarchical method for storing the spanning tree \(T\). We now describe a data structure, called an edge bundle, for storing nontree edges. Edge bundles are a compromise between two conflicting requirements: (1) we want to be able to insert and delete edges quickly, and (2) we want to maintain enough information about the nontree edges to decide quickly whether a tree edge is covered or not.

Each edge bundle represents a set of “equivalent” edges that have exactly one endpoint in a particular cluster \(C\). Edge equivalence is defined using the concept of edge targets: an edge’s target is a cluster that contains its other endpoint (more about this below). The edges with one endpoint in \(C\) form a circular sequence in the embedding; an edge bundle is a maximal subsequence with the same target.

We use two different interpretations for the targets of edge bundles in our data structure. The default target of an edge bundle is the lowest common ancestor (LCA) in the topology tree of the endpoints of the constituent edges. That is, the target cluster of the bundle is the lowest node \(v\) in the topology tree such that the edges in the bundle have both endpoints inside the cluster represented by \(v\). The advantage of this targeting scheme is that the target of an edge bundle is independent of how the topology tree has been expanded. A possible disadvantage is that the other ends of the edges in a single edge bundle may be incident to multiple clusters. In Fig. 3(a), three edge bundles are shown, one incident to each solid cluster. The target of all three bundles is the largest enclosing cluster, shown dashed.

During queries and restructuring of the topology tree, we use precise targeting for edge bundles. In this case, the targets are defined relative to a particular expansion of the topology tree. Each edge bundle consists of a set of edges connecting exactly two
clusters, and the target stored at each bundle is just the other cluster to which the bundled edges are incident. Figure 3(b) shows the edge bundles for the clusters and edges of Fig. 3(a), using precise targeting. The target of each of the four edge bundles is shown by a label on the bundle.

The data structure that represents an edge bundle is very simple: it is a record containing (1) a count of the number of edges in the bundle, and (2) the target cluster of the bundle.

5. Supernodes and coverage graphs

At each cluster that has external degree at most 2, we maintain one additional data structure, called a coverage graph. Suppose $C$ is a cluster with external degree 2 and $b_1, b_2$ are its boundary vertices (the vertices incident to the tree edges that leave $C$). Furthermore, let $p = \pi(b_1, b_2)$ be the path in $T$ between $b_1$ and $b_2$, and let $T'$ be the subtree of $T$ that connects the vertices in $C$. We build a data structure that records which parts of $p$ are covered by edges incident to $C$.

The following scenario provides a motivation for this data structure. Let $(a, b)$ be a non-tree edge, with $a \in C$ and $b \notin C$. Suppose that $b$ lies in the part of $T$ that is connected to $C$ by $b_1$. Then $(a, b)$ covers the whole path from $a$ to $b_1$ in $C$. However, if an update changed $T$ outside $C$, then $b$ could suddenly be connected to $C$ through $b_2$, in which case $(a, b)$ would cover the path from $a$ to $b_2$. If we had stored the information that $(a, b)$ covers the path from $a$ to $b_1$, we would have to change it, which would make update operations too expensive. Now suppose that there is a second edge $(a', b')$, with $a' \in C$ and $b' \notin C$, and further suppose that $b$ and $b'$ are in the same cluster when the topology tree is expanded at $a$. Then the tree path between $a$ and $a'$ is covered, no matter how $b$ and $b'$ are connected to $C$. Thus, for the purpose of computing coverage, we can collapse the path $p \cap \pi(a, a')$ to a single node, which we call a supernode. Instead of remembering the path $p$, we store only the ordered list of supernodes at $C$.

More precisely, we define the tree path $p(C)$ of a cluster $C$. If $C$ has external degree 1, $p(C)$ consists of its boundary vertex; if $C$ has external degree 2, $p(C)$ consists of the
path in $T$ between the boundary vertices of $C$. As described below, we store $p(C)$ at $C$ in a condensed form. Assume $(a, b)$ and $(a', b')$ are the extreme edges of an edge bundle, such that $a$ and $a'$ are contained in $C$. We say that the projection of $a$ on $p$ is the closest vertex to $a$ in $T'$ that lies on $p$. We call the tree path from the projection of $a$ to the projection of $a'$ the projection of the edge bundle on $p$. A subpath of $p(C)$ is internally covered if it is covered by edges that have both endpoints in $C$. See Fig. 4.

A supernode of a cluster $C$ is a maximal subpath $\pi(x, y)$ of $p(C)$ such that $\pi(x, y)$ intersects the projection of some edge bundle, and every edge of $\pi(x, y)$ either is covered by an edge internal to $C$ or lies on the projection of some edge bundle on $p(C)$.

Since an edge bundle may be a single edge, a supernode may be just a vertex. The path from $x$ to $y$ is called the path of the supernode. The paths of two supernodes are disjoint; otherwise, they would create a single supernode.

Instead of $p(C)$ we store at each $C$ the list of supernodes in the order they appear along $p(C)$. Two consecutive supernodes are connected by a superedge. A superedge represents a subpath of $p(C)$ that is not completely internally covered. We call this representation of $p(C)$ by supernodes and superedges the coverage graph of $C$. If no supernode lies on $p(C)$, we introduce a single supernode to represent $p(C)$. This guarantees that if $p(C)$ has more than one supernode, then every supernode is incident to at least one edge bundle. See Fig. 5.

We represent the coverage graph of a cluster $C$ as a doubly linked path of supernodes. Each supernode stores up to two doubly linked lists of the edge bundles incident to it, one list for each side of the tree path $p(C)$. If $C$ has external degree 1, there is only one supernode, and only one edge bundle list. The edge bundles are
listed in the counterclockwise order of their embedding. Only the first and last edge bundles in a list have direct access to the supernode to which they are incident; with this data structure, we must find the supernode corresponding to an edge bundle by following the edge bundle list to its end. The data structure lets us coalesce two adjacent supernodes into one in constant time; we can also split a supernode in two in constant time if we are given pointers that tell where to split its edge bundle lists.

6. Recipes

We would like to store the coverage graph at each cluster, but that turns out to be too expensive, both in storage and time. Therefore, we keep a recipe at each cluster C that explains how the coverage graphs of C’s children can be computed from C’s coverage graph. The coverage graph of the root of the topology tree is empty. We use the recipes to compute coverage graphs during a top–down traversal of the topology tree.

A recipe contains three kinds of instructions:

1) **Split an edge bundle.** Replace a bundle of m edges that have the same target by two adjacent edge bundles that have that target and whose (specified) sizes sum to m.

2) **Split a supernode.** Split the two edge bundle lists on either side of the supernode at specified locations. Replace the old supernode by two new ones linked by a
superedge, and give the appropriate portion of each edge bundle list to each of the new supernodes.

(3) Create a new edge bundle. Create an edge bundle with a specified target and number of edges, and insert it at a specified place in an edge bundle list of some supernode.

The transformations needed to produce the coverage graphs of a cluster C’s children from the coverage graph of C can be expressed as a sequence of such instructions.

To compute the recipe for C, we build C’s coverage graph by modifying the coverage graphs of its children, and then store at C a procedure that tells us how to reverse the construction. This procedure is the recipe. Whenever we expand the topology tree, we use the recipes to create the coverage graphs along the expanded path. Whenever we merge the topology tree, we first determine how to combine the coverage graphs of two clusters to create the coverage graph of their parent, and then we remember how to undo this operation in a recipe. Section 8 gives more details of the merging process.

Recipes use a specialized kind of pointer called a location descriptor to remember where the coverage graph of C has to be modified to create the coverage graphs of C’s children. A location descriptor consists of a pointer to an edge bundle and an offset into the edge bundle (in terms of number of edges). It takes constant time to follow a location descriptor.

We now describe the structure of the recipes, which varies depending on the number of children of C and their external degrees.

Case 1: C has only one child. In this case the coverage graph of C is identical to the coverage graph of its child. The recipe is therefore empty.

Case 2: C has two children with external degrees 3 and 1. The external degree of C is 2. The coverage graph of C contains only one supernode, with at most one nonempty edge bundle list. This list is exactly the edge bundle list of the child with external degree 1. The recipe is trivial, since the only action necessary is moving the bundle list from C’s supernode to the supernode of its degree-1 child.

Case 3: C has two children, both with external degree 1. In this case C is the root of the topology tree. Its coverage graph is empty. The coverage graphs of the children contain at most one edge bundle apiece, corresponding to the set of nontree edges linking the children. The recipe stores these edge bundles, i.e., the number of nontree edges linking the children.

Case 4: C has two children with external degrees 2 and 1. We will describe the process of going from children clusters to their parent. The recipe describes exactly the reversal of this process.

Let Y be the child of degree 2 and Z be the child of degree 1 (see Fig. 6). On each side of the tree edge between Y and Z there may be an edge bundle that connects Y and Z. To create the coverage graph for C, these edge bundles have to be removed. Then all remaining edge bundles have to be made incident to the single supernode of C. To do this we coalesce all the supernodes of Y with the single supernode of Z.
a single supernode with a single edge bundle list. If two newly adjacent edge bundles have the same target, we merge them into one edge bundle.

In the recipe we need a location descriptor to point to each edge bundle where we coalesced supernodes and concatenated their edge bundle lists (and possibly merged adjacent bundles). We also have to store any edge bundles that connect $Y$ and $Z$, and any edge bundles that were merged when they became adjacent. The number of location descriptors we store is proportional to the number of supernodes of $Y$.

Case 5: $C$ has two children, both with external degree 2. As above, we describe the merging of children clusters to get their parent’s cluster; the recipe reverses this process. Let $Y$ and $Z$ be the children of $C$. As in case 4, we have to remove the edge bundles that connect $Y$ and $Z$. In this case, however, there may also be an edge bundle that starts in $Y$ on one side of the spanning tree path, loops around the whole tree and ends on the other side of the spanning tree path in $Z$. Section 8 describes how we find such edges. We remove all the edge bundles connecting $Y$ and $Z$ and coalesce all the supernodes between the bundle endpoints into one supernode. We also merge newly adjacent edge bundles into a single edge bundle if they have the same target (see Fig. 7).

The recipe contains a location descriptor pointing to each bundle where we coalesced supernodes and concatenated edge bundle lists (and possibly merged adjacent bundles). We also store the edge bundles that were merged together or deleted, and location descriptors pointing to the merged edge bundles. If there is an edge bundle that loops around the tree, we need two more location descriptors to

---

![Fig. 6. Combining coverage graphs; recipes for case 4.](image)

![Fig. 7. Combining coverage graphs; recipes for case 5.](image)
mark its endpoints. The number of location descriptors is proportional to the number of coalesced supernodes in $Y$ and $Z$.

Edge bundles may be created during recipe evaluation in cases 3, 4, and 5. For each new edge bundle, the recipe stores a bundle record, preloaded with the count of bundle edges, and a location descriptor pointing to the place in the old edge bundle list where the new bundle is to be inserted. The target field of the bundle is easy to set: the LCA of the bundled edges is exactly the node at which the recipe is being evaluated. Each recipe evaluation creates $O(1)$ new bundles; this leads to the following lemma.

**Lemma 6.1.** If the topology tree is expanded at a constant number of vertices, and recipes are evaluated at the expanded clusters, the total number of edge bundles, supernodes, and superedges created is $O(\log n)$. The expansion takes $O(\log n)$ time.

**Proof.** Case analysis shows that each recipe evaluation creates at most six edge bundles (at most three distinct groups of parallel edges), and splits at most two edge bundles. Thus, the total increase in the number of edge bundles is at most eight. Because $O(\log n)$ clusters are expanded, $O(\log n)$ edge bundles are created.

A supernode that is not incident to an edge bundle is alone in its cluster. Since there are $O(\log n)$ clusters, there are no more than $O(\log n)$ such supernodes. Each edge bundle is incident to a supernode, and hence there are $O(\log n)$ supernodes altogether. Because the supernodes and superedges form a tree, the number of superedges is also $O(\log n)$.

The number of location descriptors in the recipe stored at a cluster $c$ is $O(s + 1)$, where $s$ is the number of supernodes coalesced while producing the coverage graph of $c$ from the coverage graphs of its children. Since there are $O(\log n)$ supernodes, and all but $O(\log n)$ location descriptors correspond to a coalesced supernode, there are $O(\log n)$ location descriptors altogether.

In cases 1, 2, and 3, only a constant amount of work has to be done. In cases 4 and 5 the work is proportional to the number of coalesced supernodes, which is proportional to the number of location descriptors. Thus, the expansion takes time $O(\log n)$. \qed

After we expand the topology tree at a constant number of vertices, we are left with an $O(\log n)$-size collection of clusters and edge bundles. The bundles use LCA targeting. To answer queries or perform updates, we need to transform the edge bundles to use precise targeting.\footnote{Actually, we can answer queries using LCA targeting, but it simplifies our presentation to use precise targeting.}

During the retargeting procedure, each edge bundle may be split into several smaller bundles, though all remain incident to the same supernode. The reason for this
follows: When edge bundles are created (recipe cases 3–5), they identify a set of parallel edges connecting two clusters $Y$ and $Z$. During the rest of the expansion process, $Y$ and/or $Z$ may be further subdivided into clusters, and hence the original set of parallel edges may connect up to $O(\log n)$ clusters. However, planarity ensures that the edges do not cross, and hence we can match the edge bundles originally incident to $Y$ with those originally incident to $Z$.

**Lemma 6.2.** Suppose that the topology tree has been partially expanded to get a collection of $O(\log n)$ clusters joined by tree edges and LCA-targeted edge bundles. Then we can compute a new set of precisely targeted edge bundles, incident to the same supernodes, in $O(\log n)$ time.

**Proof.** Recall that the tree formed by the unexpanded clusters and the edges of $T$ that link them is called the expansion tree. The algorithm uses an Eulerian tour of the expansion tree to match edge bundles that refer to the same nontree edges. Each edge $e$ of $T$ that links two adjacent clusters corresponds to a cluster $C$ that was split in two during some recipe evaluation. We label the subclusters of $C$ as left and right. The cluster $C$ is the target of any edge bundles that were created when the recipe at $C$ was evaluated. We create two lists of edge bundles (left and right) for each target $C$. We perform an Eulerian tour of (the bidirected version of) the expansion tree, visiting edge bundles in counterclockwise order, and append each bundle to one of the two lists belonging to its target $C$, depending on whether the edge bundle is in the left or right subcluster of $C$. (We switch between left and right when we traverse the edge $e$ corresponding to $C$.) When we finish the Eulerian tour, each target cluster has two lists of edge bundles; the two lists contain the same edges, one list in reverse order of the other. To find exactly which cluster is connected to which, we merge the two lists of edge bundles using the bundles' count fields. See Fig. 8.

Each list of edge bundles tells how the set of parallel edges is partitioned by the clusters at one end. After merging the two lists, we break each edge bundle at the

![Fig. 8. 3 + 4 = 7 LCA-targeted edge bundles are replaced by 2 x 6 = 12 precisely targeted edge bundles.](image-url)
breakpoints induced at the other end of the edges in the bundle. We are left with at most twice as many edge bundles as we started with, each representing one end of a group of parallel edges that links some pair of clusters in the expansion.

Since there are $O(\log n)$ edge bundles, and the Eulerian tour and the list-merging take time linear in the number of edge bundles, the total running time of the retargeting procedure is $O(\log n)$.

For convenience, we refer to the graph formed by supernodes, superedges, and precisely targeted edge bundles as the cluster graph. The previous two lemmas imply the following theorem.

**Theorem 6.3.** The cost of expanding the topology tree to get a cluster graph is $O(\log n)$.

7. Queries

To answer the query whether two vertices $u$ and $v$ are 2-edge connected, we want to test whether there is a bridge on $\pi(u,v)$, i.e., whether any edge on $\pi(u,v)$ is uncovered. We expand the topology tree at $u$ and $v$, which results in a cluster graph. By Lemmas 6.1 and 6.2, this graph has $O(\log n)$ supernodes, superedges and edge bundles. The vertices $u$ and $v$ are 2-edge connected in the original graph if $s(u)$ and $s(v)$ are 2-edge connected in the cluster graph, where $s(u)$ and $s(v)$ are the supernodes containing $u$ and $v$, respectively. We check this in $O(\log n)$ time using a classical static algorithm [1,9,16]. Then we merge the topology tree together in the same way we expanded it, leaving it as it was before the query.

**Lemma 7.1.** Using the data structure described in Sections 3–6, we can determine whether two query vertices $u$ and $v$ are 2-edge connected in $O(\log n)$ time.

8. Updates

The general structure of an update is as follows. First we expand the topology tree at a constant number of vertices. Next we retarget the edge bundles to give each edge bundle a supernode as a target (see Lemma 6.2). Now a cluster graph has been created.

On this graph we perform our changes, which may include insertion and deletion of edges and vertices, and making nontree edges into spanning tree edges and vice versa. This part varies depending on what kind of update we are performing. Finally, we merge the topology tree back together. In the previous sections we have explained how to create the cluster graph. In this section we explain the changes to the graph during updates and the merging of the topology tree. We support two kinds of updates, edge insertions and deletions. (It is straightforward to extend this to allow insertions/deletions of isolated vertices.)
Because we assume that $G$ is an embedded planar graph, it is important that the updates preserve this property. This is trivial for deletions: the user may request the deletion of any edge. However, when the user asks for an edge insertion, she must ensure that the edge to be inserted will preserve planarity. Furthermore, she must specify where the edge is to be inserted so as to maintain a proper embedding. Therefore, an edge-insertion request specifies two vertices (the endpoints of the new edge) and an edge incident to each of the vertices. The new edge will be inserted immediately counterclockwise of the specified edges at each endpoint. If inserting an edge at the specified position causes a nonplanar embedding, the data structure will behave unpredictably.

8.1. Inserting an edge

To insert an edge $(u, v)$, we expand the topology tree at $u$ and $v$, after first checking the degrees of $u$ and $v$ in the original graph. Let $x$ stand for either $u$ or $v$. If the degree of $x$ before the insertion is 1 or 2, we expand the topology tree at $x$, create the cluster graph, and give $x$ a new nontree edge.

If the degree of $x$ before the insertion is 3, we must create a four-vertex cycle for $x$ and connect it appropriately. We expand the topology tree at $x$, then replace $x$ by a four-vertex cycle. We connect the three original edges plus the newly inserted edge to the cycle in the proper order. We make three of the four cycle edges be tree edges; the last cycle edge and the newly inserted edge are nontree edges.

If the degree of $x$ becomes larger than 4, we already have a cycle for $x$. We are told between which edges incident to $x$ in the original graph the new edge has to be inserted. Each of these edges is incident to exactly one vertex in the cycle. Therefore, we know between which two vertices $x_1$ and $x_2$ a new vertex has to be inserted. We expand the topology tree at these two vertices and create the cluster graph. We insert a new vertex $x_3$ between $x_1$ and $x_2$. The edge $(x_2, x_3)$ is a tree edge; $(x_1, x_3)$ is a tree edge iff $(x_1, x_2)$ was a tree edge. The newly inserted edge incident to $x_3$ is a nontree edge. Figure 9 shows examples of expanding the cycle corresponding to a vertex of the original graph.

After making the appropriate modifications at $u$ and $v$ in the cluster graph, we merge the topology tree back together. (The details of this merge appear below.)

8.2. Deleting a nontree edge

Deleting a nontree edge $(u, v)$ is essentially the inverse of an insertion. For each endpoint of the deleted edge we have to do the following. If the degree of an endpoint $x$ is at most 3, we expand the topology tree at $x$, create the cluster graph, and remove the nontree edge from $x$.

If the degree of a vertex $x$ before the deletion is 4, at least one of the four vertices on the cycle must be connected by a tree edge with the rest of the spanning tree. We label
this vertex $x$. We have to delete the other three vertices from the cycle of $x$. Therefore, we expand the topology tree at all four vertices, create the cluster graph, and delete the three vertices. The edges that were incident to the cycle but were not deleted are connected to $x$.

If the degree of $x$ is larger than 4, the cycle of $x$ is not destroyed. We are told (or can compute in constant time) between which edges incident to $x$ in the original graph the edge to be deleted lies. Each of these edges is incident to exactly one vertex in the cycle. Therefore, we know the vertex $x_3$ that has to be deleted and its neighbors $x_1$ and $x_2$ in the cycle. We expand the topology tree at these three vertices, create the cluster graph, and delete the vertex $x_3$ and all three edges incident to it. If $x_3$ was connected to both $x_1$ and $x_2$ by tree edges, we connect $x_1$ and $x_2$ by a tree edge. Otherwise we connect $x_1$ and $x_2$ by a nontree edge. See Fig. 9.

In all cases we finish by merging the topology tree back together.

8.3. Deleting a tree edge

When a tree edge $(u, v)$ is deleted, the spanning tree is broken up into two parts. We expand the topology tree at the endpoints of the edge and create the cluster graph. Then we run along one of the two faces adjacent to $(u, v)$ and find a group of parallel edges (represented by a pair of edge bundles) that connects the two parts of the spanning tree. This can be done in $O(\log n)$ time by examining all the edge bundles in the cluster graph. (We are assuming here that the edge $(u, v)$ was not a bridge edge of the graph $G$.) By repeatedly expanding the clusters incident to the pair of edge bundles, we can in $O(\log n)$ time identify a nontree edge $(u', v')$ that connects the two spanning tree components. We expand the topology tree at $u'$ and $v'$ and make $(u', v')$ into a tree edge and $(u, v)$ into a nontree edge. Then we continue as for the deletion of a nontree edge. See Fig. 10 for examples of topology tree expansion.
8.4. Merging the topology tree

We now describe how the topology tree is merged together after an edge insertion or deletion. The procedure has three steps. First, we compute the new topology tree for the updated cluster graph. Second, we compute the new edge bundles and their LCA targets based on the new topology tree. Third, we update the recipes in all the clusters affected by the changed edge bundles. There are $O(\log^2 n)$ such clusters, and we spend amortized constant time apiece, for a total update time of $O(\log^2 n)$; all the other operations take $O(\log n)$ time.

We first determine the structure of the new topology tree for the modified cluster graph, as described in Section 3. This step is just preparatory for the full reconstruction of edge bundles, supernodes, and recipes, and does not involve any of those elements. We expand all the clusters affected by the restructuring algorithm (still only $O(\log n)$ clusters) to get a larger cluster graph.

The clusters of the cluster graph correspond to a fringe of nodes in the new topology tree, namely, the boundary between expanded and unexpanded topology tree nodes: for each cluster $C$ in the cluster graph, all its topology tree ancestors have been expanded, and none of its descendants have been expanded. To create recipes for the nodes above the fringe, we must compute the new edge bundles and their LCA targets based on the new topology tree. To do this we first compute precise targets for the edge bundles in the cluster graph. Now each edge bundle identifies one end of a group of parallel edges linking two clusters. For each pair of linked clusters, we compute their lowest common ancestor in the new topology tree and label each edge bundle with the LCA as its target. This takes $O(\log n)$ time altogether. Finally we merge any edge bundles that have the same target and are adjacent along the boundary of some cluster. This may involve merging supernodes as well.

There are two sets of topology tree nodes (clusters) for which recipes need to be computed. First, it is clear that the nodes above the fringe in the topology tree must
have recipes created for them: these nodes may never have had recipes at all. Second, and less obviously, we must recompute recipes for certain nodes (clusters) below the fringe. The affected clusters are those that contain any of \( O(\log n) \) vertices, namely the extreme vertices of edge bundles in the original or revised cluster graphs. More precisely, if some vertex is an endpoint of an extreme edge in a bundle in the original cluster graph, but not in the revised cluster graph (or vice versa), then all clusters containing it need new recipes. (The edges of the edge bundles do not change, but their LCA targets may change, thereby changing the boundaries between bundles.)

There are two reasons that recipes must be updated below the fringe. First, the recipes below the fringe use location descriptors that point to edge bundles in fringe clusters. For these location descriptors to be meaningful, we must update the recipes below the fringe to match the edge bundles above it. Second, and more importantly, some supernodes in clusters below the fringe are defined by the projections of edge bundles from the fringe clusters. When these projections change, the supernodes and their recipes must be updated.

To prepare for recipe (re)creation, we expand the topology tree yet again at all \( O(\log n) \) of the vertices identified above, then set the targets for edge bundles according to LCAs in the new topology tree. The \( O(\log^2 n) \) expanded nodes lie on \( O(\log n) \) disjoint monotone paths in the topology tree; call these expansion paths. (By monotone we mean that the level of the path nodes strictly increases from one end of the path to the other.) The expansion paths can be partially ordered by “aboveness”: one path is above another if any of its nodes is an ancestor of a node on the other path. We process the expansion paths in ascending order of aboveness. The clusters that are children of a path are incident to a total of \( O(\log n) \) edge bundles: this follows because the cluster at the top of the path has \( O(\log n) \) incident bundles, and we know from Section 6 that the expansion of each cluster along the path adds \( O(1) \) LCA-targeted edge bundles incident to the child clusters of the path.

We compute the coverage graphs and recipes along each expansion path by bottom-up merging (cf. Section 6): for each cluster we combine the coverage graphs of its subclusters, than record how to reverse the operation in a recipe. The aboveness ordering of the expansion paths ensures that when we want to merge a node on the path with one off the path, we will have already computed coverage graphs for both nodes. We process each path in \( O(\log n) \) time.

We now present a high level algorithm for merging two clusters; the subroutine details appear below. Each of the two clusters to be merged has a coverage graph as described in Section 5. We also assume that each edge bundle has been labeled with the depth of its target in the topology tree (labeling takes constant time per bundle).

We first identify the edge bundles incident to both clusters that will be internal to the merged cluster. These are exactly the bundles whose targets have the same depth as the merged cluster; we find them by selecting bundles by depth. Second, we identify the supernodes of the coverage graphs to which these bundles are incident. These supernodes and all the supernodes on the tree path connecting them will be coalesced. We identify the supernodes to coalesce by walking outward from the tree edge joining
the two clusters. It follows that coalescing takes time proportional to the number of supernodes being coalesced, and hence amortizes to $O(\log^2 n)$ overall. (This description is most applicable to the case of merging two degree-2 nodes into one; there are minor differences when any of the nodes has degree 1 or 3, but the basic ideas are the same.)

After we identify the bundles that will be internal to the merged node, we follow three steps: (1) delete these bundles; (2) coalesce supernodes; and (3) merge adjacent bundles that have the same target. The recipe is just the reverse of the merging operation.

There are two subtasks to be explained more fully: (1) identifying bundles of a particular depth, and (2) locating these bundles’ supernodes in the coverage graph.

Each of the $O(\log n)$ expansion paths in the topology tree has $O(\log n)$ nodes that are children of the path. When we process the path, these nodes already have coverage graphs; we call these nodes static nodes. The nodes on the path are called dynamic nodes. We compute recipes and coverage graphs for the dynamic nodes by walking up the path from bottom to top. The following operations are applied on each expansion path separately.

**Operation 1.** Identify the bundles of a given depth at each cluster along an expansion path. We maintain an array of length $O(\log n)$, indexed by depth, of the edge bundles incident to the current dynamic cluster. Each array entry points to a circular list of bundles with the same depth. We initialize the array at the node at the bottom of the expansion path, then maintain it as we walk to the top of the path. Thus, the initialization overhead is $O(\log n)$, but the lookup time is constant, as desired. When we merge in the bundles from a static node, we can afford to look at all of them (because there are only $O(\log n)$ edge bundles incident to the static nodes altogether) and insert them into the proper array slots.

**Operation 2.** Given an edge bundle, map it to the supernode in the coverage graph to which it is incident. To do this we use a bit vector of length $O(\log n)$, each bit corresponding to one of the edge bundles incident to the clusters on the expansion path. We use the predecessor operation ("find the first 1-bit left of the query position") to identify the supernode to which an edge bundle is incident. This operation takes constant time on a RAM with word size $\Omega(\log n)$.

To map edge bundles to bit-vector indices, we perform an Eulerian tour of the static clusters belonging to the expansion path, numbering the edge bundles in order along the way. The bundles incident to any cluster are ordered consistently with this numbering.

The 1-bits in the bit vector represent the edge bundles that are directly linked to their supernode (these are the first and last bundles in the supernodes’ edge bundle lists). To find the supernode to which an edge bundle belongs, we do a predecessor query on the bit vector. For each expansion path, setting up the bit vector takes
O(\log n) time. Maintaining the bit vector as the supernodes coalesce and as static nodes are merged into the dynamic node also takes O(\log n) time per expansion path. We use the bit vector to perform O(\log n) constant-time queries.

The preceding discussion establishes the following update lemma.

**Lemma 8.1.** The 2-edge-connectivity data structure can be updated in response to an edge insertion or deletion in \(O(\log^2 n)\) time.

All the operations needed to merge two clusters can easily be performed in time proportional to the number of edge bundles incident to the clusters. This means that our data structure can be built from scratch by bottom-up merging in \(O(n)\) time. This observation finishes the proof of our main theorem.

**Theorem 8.2.** An embedded planar graph can be preprocessed into a linear-space data structure that supports insertion or deletion of an edge in \(O(\log^2 n)\) time and answers a 2-edge-connectivity query between any two vertices in \(O(\log n)\) time. If the graph has \(n\) vertices, then the preprocessing cost is \(O(n)\).

9. Concluding remarks and open problems

We have proposed a new data structure for maintaining 2-edge connectivity in embedded planar graphs. Along the way we have introduced several new ideas on dynamically maintaining a planar embedded graph, which may have broader applications.

Testing planarity in an embedded graph is one application of our data structure. A recent algorithm of Italiano et al. [13] uses the data structure to support insertions that do not destroy the planar embedding, deletions, and queries of the form "would the insertion of edge \((x, y)\) destroy the planar embedding?"

Our algorithm permits only those updates that maintain the planar embedding. In many applications of dynamic planar graphs, such as routing and chip design, all updates come from an underlying planar subdivision and thus our model is appropriate. Of course, from a theoretical point of view, it would be more satisfying to have an algorithm that was not restricted by a particular embedding. (The recent algorithm of Eppstein et al. [3] achieves this goal.)

With some small modifications, our data structure may work for 2-vertex connectivity as well. We are currently investigating this. It also seems likely that many of our ideas generalize to 3-edge connectivity, and we continue to explore that possibility. However, it appears that \(k\)-edge connectivity for \(k > 3\) will require a different approach, since the extension of our basic edge-covering lemma (cf. Lemma 2.1) does not hold. Dynamic connectivity in directed graphs also remains largely unexplored.
References