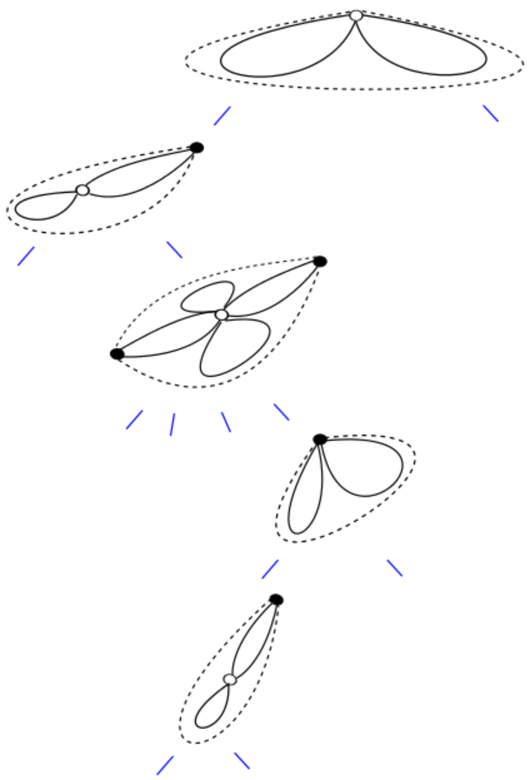
Master's thesis

Maintaining alterable planar embeddings of dynamic graphs

Eva Rotenberg

Supervised by Christian Wulff-Nilsen, DIKU

January 31, 2014



Abstract

We present a data structure for maintaining a planar embedding of a dynamic plane graph. The graph may be updated with edge deletions and insertions of edges when it does not violate the embedding. Queries of vertex pairs return the list of possible places for inserting an edge. Furthermore, the embedding may be updated with flips that alter the embedding by swapping the orientation of a subgraph, if the subgraph has certain properties of being not too well connected to the rest of the graph. All operations run in time $\mathcal{O}(\lg(n)^2)$, where n is the number of vertices in the graph. We do this using top-trees, and using elegant properties about the dual graph.

Previous best result in this direction was by Italiano et al. [20], a data structure using topology trees which supports edge deleting, edge insertion (when compatible with the embedding), and query. They have a running time of $\mathcal{O}(\lg(n)^2)$ per update, but their construction did not allow for alterations of the embedding.

A new lower bound for dynamic planarity testing is presented, namely $\Omega(\lg(n))$ for the slowest operation. The previous best lower bound was $\Omega(\lg(n)/\lg\lg(n))$ by Henzinger et al. [13].

1 Introduction

Given a connected graph $G = (V, E)$, a planar embedding of the graph corresponds to a drawing of the graph on a sphere, such that the vertices are distinct, and such that no edges cross. Note that a graph can be drawn on the plane, $\mathbb{R}^2$, if and only if it can be drawn on the sphere, S^2. We view embeddings as equivalent if one can be obtained from the other by stretching and contracting the sphere as if it were made of some elastic material. More formally, homotopic embedding functions $f : G \to S^2$ are considered equivalent. It follows that an embedding is given by, for each vertex v in G, an ordering of the edges incident to v. Several planar embeddings of the same graph may exist. We say that a graph is planar, if *some* planar embedding of it exists. We use the word *plane graph* about a planar graph equipped with a planar embedding. Not all graphs are planar; e.g. the complete graph with 5 vertices, or the complete bipartite graph with $3 + 3$ vertices.

In this set-up, a *dynamic graph* is a graph on a given set of vertices, where edges can be removed or inserted. We will present a data structure for maintaining a planar embedding of a dynamic graph. In our variant, an edge (u, v) may only be inserted, if u and v belong to some common face (or separate connected components), and the insert operation inserts the edge without altering the rest of the embedding of the graph. The operation delete(u, v) simply deletes the edge (u, v). The operation query(u, v) returns the (possibly empty) list of common faces incident to both u and v, and for each such face, corners of that face incident to u and corners incident to v – a corner corresponds to specifying where in the ordering of the edges around u or v, a new edge would fit. Given a face, f, and two corners on the face, c_u and c_v, incident to u and

v, respectively, the operation insert(c_u, c_v) inserts an edge between u and v in the dynamic graph, and embeds it in the specified corners.

Many different embeddings of a planar graph may exist. It may often be relevant to alter the embedding, in order to be able to insert an edge. We allow the user to alter the embedding by what we call flips, that is, to turn part of the graph upside down in the embedding. Of course, the relevance of this depends on what we want to describe with a dynamic plane graph. If the application is to describe roads on the ground, flipping orientation would not make much sense. But if we have the application of graph drawing or chip design in mind, flips are indeed relevant. In the case of chip design, one wants to make a planar embedded circuit, which can be thought of as a planar embedded graph.

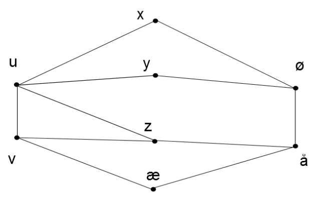


Figure 1: This embedding does not allow for the insertion of the edge (x, z), although an embedding exists where (x, z) may be inserted.

An small example of a planar embedded graph is that of Figure 1. The subgraph consisting of vertices x, y and all their incident edges may be flipped in the embedding, see Figure 2. In the resulting embedding, the edges x and z belong to the same face - for the original embedding this was not the case.

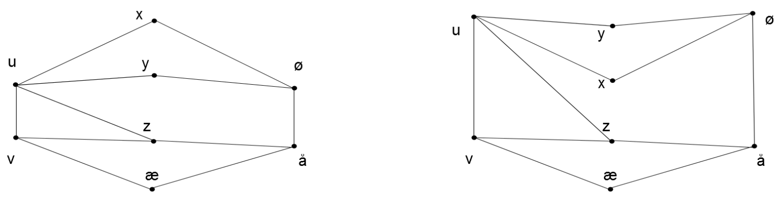


Figure 2: We flip part of the graph, and now, the edge (x, z) can be inserted.

An interesting and related problem is that of dynamic planarity testing of graphs. That is, we have a planar graph, we insert some edge, is the graph still planar? Knowing that the graph was already planar, only tells us of the existence of some embedding ϕ of the graph. It could very well be, that ϕ does not extend to a planar embedding of the new graph after the insertion of an edge, yet some other planar embedding, ϕ', allows for the particular edge insertion. The operations supported by a data structure for planarity testing of graphs, are: insert(u, v), which alters the graph by inserting the edge (u, v), delete(u, v) which alters the graph by deleting such an edge, and query(u, v), which returns true if the edge (u, v) can be inserted without destroying the planarity of the graph.

Although the problem of dynamic planarity testing appears technically harder, it is only relevant when the user has no interest in the actual embedding of the graph. What we do is to support the user to have control over the embedding at all times. We tell the user where edges may be inserted, but leave the actual choice to the user. We also support the user in changing their mind about the embedding, e.g. by flipping components, so as to make edge insertions possible.

Part of our contribution is a simple duality-based dynamic representation of a planar embedded graph, maintaining primal and dual trees which share (some modified version of an) Euler Tour.

We hope that alterable planar embeddings of dynamic graphs can be used as a stepping stone to polylogarithmic planarity testing of graphs. If a data structure supports flips in polylogarithmic time, and in order to insert an edge, at most polylogarithmic many flips are needed, and an algorithm finds those places to flip in polylogarithmic time, then we have a planarity testing data structure with polylogarithmic time per operation, which uses our construction as a sub-routine.

1.1 Previous work

Dynamic graphs have been studied for several decades. Usually, a fully dynamic graph is a graph that may be updated by the deletion or insertion of an edge, while decremental or incremental refers to graphs where edges may only be deleted or inserted, respectively. A dynamic graph can also be one where vertices may be deleted along with all their incident edges, or some combination of edge- and vertex updates [31]. One of the fundamental questions about a dynamic graph is the question of dynamic connectivity in undirected graphs. An update is an insertion or deletion of an edge, and a query is to ask whether two vertices belong to the same connected component.

Dynamic Connectivity for graphs was studied by Frederickson [7], who achieved an update time of $\mathcal{O}(\sqrt{m})$, where m is the number of edges, and constant query time. This was done using topology trees to maintain a spanning forest of the graph. The result of Frederickson was improved using the sparsification technique of Eppstein, Galil, Italiano, and Nissenzweig [6] to an update time of $\mathcal{O}(\sqrt{n})$, where n is the number of vertices. The first data structure with polylogarithmic update time was that of Henzinger and King [11], namely expected, amortized $\mathcal{O}(\lg(n)^3)$ and constant query time. Henzinger and King used Euler Tour trees to maintain information about the spanning forest. A deterministic data structure for supporting connectivity queries in fully dynamic graphs was first presented by Holm, de Lichtenberg, and Thorup [18], and they used top-trees to maintain information about the dynamic graphs. The time bounds were improved by Thorup [34] who obtained an expected update time of $\mathcal{O}(\lg(n) \cdot (\lg\lg(n))^3)$ and query time $\mathcal{O}(\lg(n)/\lg\lg\lg(n))$, and by Wulff-Nilsen [39] who obtained a deterministic update time $\mathcal{O}(\lg(n)^2/\lg\lg(n))$ and query time $\mathcal{O}(\lg(n)/\lg\lg(n))$.

Related dynamic graph problems include dynamic minimum spanning forest (MSF), which was studied by Frederickson [7], who maintained a minimum spanning forest for a dynamic graph with update time $\mathcal{O}(\sqrt{m})$, where m is the number of edges in the graph. The sparsification technique of Eppstein et al. [6] improves this to $\mathcal{O}(\sqrt{n})$, where n is the number of vertices. The first data structure with polylogarithmic running time was that of Henzinger and King [15], with (randomized, amortized) update time of $\mathcal{O}(\lg(n)^3)$. Holm et al. [18] used top-trees to make a data structure for decremental MSF with deterministic, amortized update time $\mathcal{O}(\lg(n)^2)$, and a reduction to fully dynamic MSF which has update time $\mathcal{O}(\lg(n)^4)$. Other related dynamic graph problems are those of bi-connectivity (do there exist two vertex-disjoint paths from u to v?) [10, 16, 15, 18] and 2-edge connectivity (do there exist two edge-disjoint paths from u to v?) [8, 14, 18]. Related to k-edge connectivity, the fully dynamic min-cut problem is studied by Thorup [35].

Hopcroft and Tarjan [19] were the first to make an algorithm for planarity testing of static graphs in linear time. The incremental planarity problem is that of testing planarity of a dynamic graph where only edge-insertions are allowed. Incremental planarity testing was solved by La Poutre [26], who improved on work by Di Battista, Tamassia, and Westbrook [4, 5, 38], to obtain a total running time of $\mathcal{O}(\alpha(q, n))$ where q is the number of operations. Galil, Italiano, and Sarnak [9] made a data structure for fully dynamic planarity testing with $\mathcal{O}(n^{2/3})$ worst case time per update. For maintaining embeddings of planar graphs, the best result so far is that of Italiano, La Poutr, and Rauch [20]. The data structure presented by Italiano et al. is for maintaining a (combinatorial) embedding of a dynamic graph, and has $\mathcal{O}(\lg(n)^2)$ update- and query time.

The highest lower bound for this type of problems is Pătraşcu's $\Omega(\lg(n))$ lower bound for fully dynamic connectivity [30]. However, the best known lower bound for dynamic planarity testing is that of Fredman and Henzinger [13] and Miltersen et al. [27], a lower bound of $\Omega(\lg(n)/\lg\lg(n))$.

1.2 Our idea: The top-tree and the co-top-tree

We want to maintain an embedding of the graph which facilitates edge insertion, edge deletion, edge query (that is, where may one insert this given edge?), and alteration of the embedding in the form of flips. One may flip the embedding in an articulation point or in a separation pair.

For a vertex with k incident edges, there are k incident *face corners* between the edges. The reply to a query(u, v) should contain not only the list of common faces of u and v, but also the place in the ordering of the incident edges to each vertex, that is, the usable corners. Similarly, the insertion of an edge should specify which corners should be used for the insertion. The time of query depends on the length of the face list, but if the user only wants k common faces (e.g. $k = 1$), and k corner pairs, the function easily generalises to a query$((u, v), k)$.

The idea is to maintain a spanning forest, and use it to make a tree/co-tree decomposition of each connected component of the graph. For each connected component, we maintain two top-trees. One for the spanning tree of the component, and one for the co-tree. We define the extended Euler Tour (which visits all edges twice), and note that it is the same for the tree and for the co-tree.

Upon a query of (u, v), we expose u and v in the primal top-tree, using the dual top-tree to keep track of which corners are incident to either u or v, and which are not. We also use the top-tree to maintain the list of "good" faces (incident to both u and v). All candidate faces for being common, that is, incident to both u and v, must lie on a path in the dual tree. We can easily (that is, in $\mathcal{O}(\lg(n))$ time) calculate the endpoints of that path, and then these can be exposed in the dual tree: the good faces on the cluster path of the exposed cluster are exactly the common faces of u and v.

To alter the embedding, there are two possibilities. If there is an articulation point, then there is a vertex with two corners to the same face. Then the embedding is flipped in those corners, and the orientation of the subgraph is reversed. Similarly, if there is a separation pair, that is, a pair of vertices, u, v, which have two common faces, f, g, then there is a subgraph delimited by those vertices and faces — the embedding of this subgraph may be flipped and the orientation reversed.

In Section 2, I give a short presentation of the basic graph theory needed to understand my algorithm and the basic data structures it uses as building blocks. In Section 3, I will give a short outline of the ideas used by Italiano et al. in their data structure for maintaining an embedding of a dynamic graph using topology trees. In Section 4, a new data structure for the same task, maintaining an embedding of a dynamic graph using topology trees, is presented. In Section 5, I will show how this data structure extends to flips, that is, how we can allow the user to alter the embedding by flipping a sub graph while holding the rest of the graph still. In Section 6, I improve the $\Omega(\lg(n)/\lg\lg(n))$ lower bound by Henzinger [12] to $\Omega(\lg(n))$, using techniques introduced by Pǎtraşcu [30]. I conclude with a few thoughts about future work.

Acknowledgements

I would like to thank Christian Wulff-Nilsen, Jacob Holm, and Mikkel Thorup for many helpful and interesting discussions and ideas.

2 Basic concepts

The new data structure I present in Sections 4 and 5, maintains an embedding of a planar graph. In this section I will present some basic notions and vocabulary for planar graphs, and I will go through some data structures used for maintaining information about dynamic graphs.

2.1 Planar graphs and planar embeddings

There are two ways of viewing embedded graphs. The classical way and the combinatoric way. The classical approach is more intuitive, but the combinatoric approach makes for simpler proofs and makes it easier to generalise results.

Definition In the edge-centric view, a (multi-) graph consists of a set of edges E and a set of vertices V, such that the vertices form a partition of $E \times \{0,1\}$.

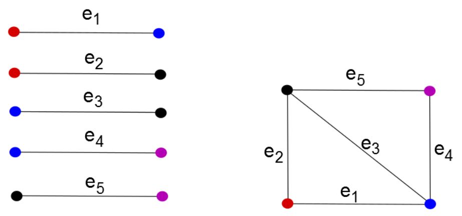


Figure 3: Left: Five edges, $E = \{e_i\}_{i=1\ldots5}$, depicted as $e_i \times [0,1]$, and a division of $E \times \{0,1\}$ into four vertices: red, blue, black, and purple. E.g. blue is the set $\{(e_1,1),(e_3,0),(e_4,0)\}$. Right: The corresponding graph.

In the classical view, an *embedding* of (V,E) is a homotopy class of a continuous, injective function: $(E \times I)/V \to S^2$, where I is the interval $[0,1]$, where S^2 is the two-dimensional sphere, and where by $/V$ is meant that we quotient out by the equivalence relation corresponding to the partition V.

That the function is continuous and injective means that each vertex is mapped to its own separate point, and that the edges are drawn as curves which do not cross. We consider a homotopy class, because we consider embeddings as equivalent if one is obtained from the other by rotation or stretching of the sphere.

For a connected graph, an embedding corresponds uniquely to an orientation of the edges around each vertex, or equivalently, a permutation π of $E \times \{0,1\}$ such that the orbits of π are the classes of V. A proof of this will not be included here. We call π a *combinatoric embedding* of the graph.

Not all graphs are planar. The complete graph on 5 vertices, K_5, and the complete bipartite graph with $3+3$ vertices, $K_{3,3}$, are examples of graphs that cannot be embedded in the plane. (In fact, a theorem by Kuratowski [25] states that any non-planar graph contains a subgraph, which is a sub-division of K_5 or $K_{3,3}$. Wagner [37] shows that any non-planar graph contains K_5 or $K_{3,3}$ as a minor.)

For a planar embedded connected graph, the set of points not in the image of the embedding will form connected components, which we call *faces*.

Definition The *dual* multi-graph G^* of a plane (multi-) graph G has one vertex corresponding to each face of G, and for each edge in G with neighbouring faces f_1 and f_2, G^* contains an edge between f_1 and f_2.

Note that there may be several edges between two vertices, for instance, the dual of the graph in Figure 1 has two edges between all incident faces.

Note 2.1. *Given a plane, connected graph, G, and a spanning tree T for G, then the edges of $G \setminus T$ form a spanning tree for the dual graph G^*.*

Proof. Let T^* denote $G \setminus T$ as a subgraph in G^*. T^* is acyclic, because G was assumed to be connected, and because T spans G. T^* spans G^*, because T was assumed to be acyclic. $\square$

Definition We call this the *tree-co-tree decomposition* of the graph.

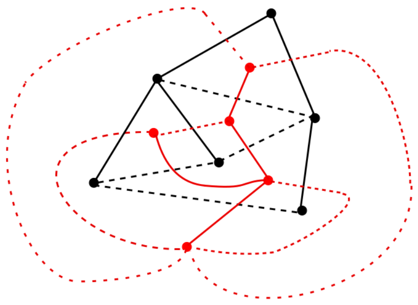


Figure 4: The tree-co-tree decomposition of a connected plane graph. The black lines belong to the spanning tree, the dotted black lines are non-tree edges. The red lines are the "dual" edges, which go from face to face. When each red-black cross contains one dotted and one non-dotted edge, the red non-dotted lines make a spanning tree for the dual graph.

Note that the dual graph may be embedded such that the dual vertices are inside the face they correspond to, and such that the intersections of e with e^* for $e \in E$ are the only intersections between the two. (Without loss of generality, we can assume this property holds.)

For a vertex with k incident edges, there are k incident face *corners* between the edges. A corner can be thought of as something that links a face to a vertex. Note that a corner in the primal graph is also a corner in the dual graph, and vice versa.

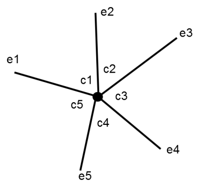


Figure 5: A vertex with five edges and 5 incident corners.

Given this definition of corners, we can extend the definition of Euler tour not only to include non-tree edges but also to include corners, as follows. (See

Figure 5 for an example.)

Definition Given a planar embedded graph G with a spanning tree T embedded on an oriented plane, we construct the *extended Euler Tour* as the following linked cycle.

1. Pick a corner $s = c_0$ incident to the vertex v.
2. Using the orientation of the plane, starting with s, add all corners and non-tree edges incident to v until you reach a tree-edge $e' = (v, u)$. Break if you reach c_0, unless the list is empty.
3. Set $v = u$ and $c = e'$, and goto `2.`
4. Starting with e, add all corners and edges up to e_0. Link the end to the beginning.

In step 2. above, s may be an edge or a corner, inductively. When we see c_0 for the second time, we have traversed the entire tree, and may stop. This way, each edge, whether tree or non-tree, will be visited exactly twice, and each corner exactly once during the extended Euler Tour.

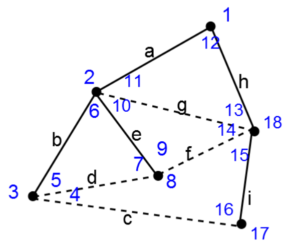


Figure 6: This graph has extended euler tour `1a2b3c4d5b6e7d8f9e10g...18h`, or to write only the edges: **ab**cd**be**df**e**g**ah**gf**i**c**ih**.

Lemma 2.2. *Using the tree co-tree decomposition of a plane embedded, connected graph, G, the extended Euler Tour of G using one orientation of the plane, is the same as that for the co-graph G^* using the opposite orientation of the plane.*

In order to prove the lemma, we use the following definition of an *Euler Cut*:

Definition Given any closed simple curve C on the sphere S^2, then $S^2 \setminus C$ has two components, A and B, each homeomorphic to the plane[1]. Given an edge e of the embedded connected graph G with dual graph G^*, and a cut C, we say

[1] Jordan curve thm.

that C intersects e if C intersects either e of G, or e^* of G^*. Given a tree-co-tree decomposition for G, we call C an *Euler Cut* if the tree is entirely contained in A and the co-tree is entirely contained in B, and the curve only intersects each edge, e or e^*, twice.

Proof of lemma. Given any Euler Cut C, note that the edges visited along the Euler Tour of G come in exactly the same order as they are intersected by C, if we choose the same ordering of the plane (clockwise or counter-clockwise). The placement of the corners is uniquely defined by the ordering of the edges.

Such an Euler Cut always exists, because both tree and cotree are connected, and because they share no common point. One can construct it by contracting one of the two trees to a point, and then considering an ε-circle B_ε around that point, where ε is small enough such that there is a minimal number of edge-intersections with that circle. The pre-image of B_ε under the contraction will be an Euler Cut. (See Figure 7.)

Now, the Euler tour of G is uniquely given by C. But we also know that G is the dual of G^*, so C is also an Euler Cut for G^*, and thus, the Euler Tour of G^* is uniquely given by C.

Lastly, notice that in order for C to determine the same ordering of the edges, we need to orient the two components oppositely, that is, one clockwise and one counter-clockwise. □

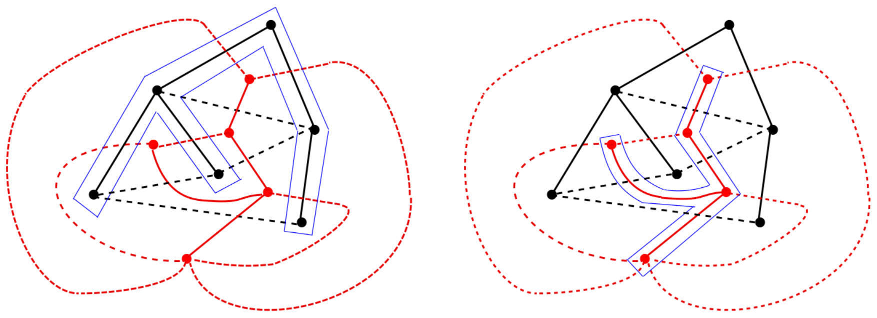


Figure 7: Two Euler Cuts separating the primal tree from the dual tree. Each is made by following the tree in (roughly) ϵ distance.

2.1.1 Combinatorial embeddings

Although this is not strictly necessary for understanding the thesis, I would like to present the idea behind combinatorial embeddings, as they may give some intuition about planar embedded graphs.

A permutation π of $E \times \{0, 1\}$ such that the orbits of π are the classes of V is called a *combinatorial embedding* of the graph.

Define *rev* : $E \times \{0, 1\} \to E \times \{0, 1\}$ as $\text{rev}(e, 1) = (e, 0)$ and $\text{rev}(e, 0) = (e, 1)$. Then the *dual* of an embedded graph (G, π) is $\pi^* = \pi \circ \text{rev}$. The orbits of π^* are the *faces* of the embedded graph. We say that the combinatorial embedding is planar, if Euler's formula holds. For a connected graph, this definition of dual is the same as the classical one. If the graph consists of more than one component, this corresponds to embedding each component of the graph to its own sphere. When we later maintain a spanning forest of a graph, we will maintain the tree co-tree decomposition of each connected component. We do not distinguish between the two non-homotopic planar embeddings of the graph in Figure 8, and consider them equivalent, since they have the same ordering of the edges around any vertex.

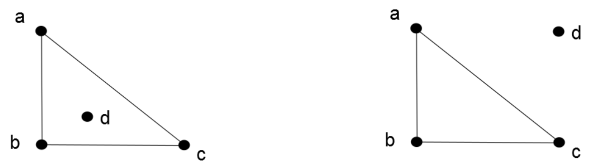


Figure 8: These two embeddings of the graph are considered equivalent, because they have the same ordering of edges around any vertex.

Using this definition of an embedding, the definition of an extended Euler Tour is straightforward, and the proof of Lemma 2.2 is trivial.

2.1.2 Basic graph vocabulary

In a tree, $T = (V, E)$, for $a, x, y \in V$, we use $\texttt{lca}_a(x, y)$ to denote the nearest common ancestor for (x, y) when the tree is rooted in a. Note that $\texttt{lca}_a(x, y) = \texttt{lca}_x(y, a)$ and so forth; we denote it also by $\text{meet}(a, x, y)$. Given a path p in the tree, from x to y, we define the *projection* of the vertex a on the path p as $\pi_p(a) = \text{meet}(a, x, y)$.

In a graph, an *articulation point* is a vertex with the property that removing it would increase the number of components of the graph. For an embedded graph (without self-loops), a vertex is an articulation point iff it has two corners incident to the same face.

We say that a pair of vertices is k-connected if k (pairwise, internally) vertex-disjoint paths between them exist. We say that a graph is k-connected, if all pairs of vertices are k-connected. A 2-connected pair of vertices (u, v) is a *separation pair*, if the graph is the union of two proper subgraphs A and B, neither of which is an edge, such that $A \cap B = \{u, v\}$. For an embedded graph, (v_1, v_2) is a separation pair iff they are 2-connected, and there exist four corners $c_1, \ldots, c_4$, pairwise incident to v_1, v_2 and to two faces f_1, f_2, such that no edge $e = (u, v)$ with corners $c_1, \ldots, c_4$ exists. (See Figure 9.) A 3-connected graph has a unique embedding up to reflection.

See [28, 23] for more on planar graphs and their embeddings.

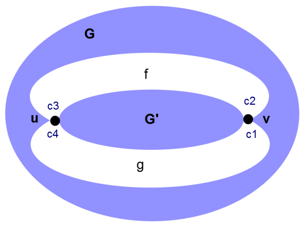


Figure 9: (u, v) is an articulation pair if neither G nor G' consists only of one edge.

2.2 Maintaining information about dynamic trees.

I will now present data structures which can maintain information about a tree with n vertices in time $\mathcal{O}(\lg n)$. These data structures are all in the shape of trees: ET-trees, topology trees and top-trees. Each have the property that local information can be "bubbled up" through the root-path of the edge or vertex where the local information has been dynamically changed.

2.2.1 Euler Tour trees

The Euler tour of a rooted tree visits each path twice and each node with $d-1$ children is visited d times. The Euler Tour tree is simply a binary tree over the Euler tour. Each vertex of the tree keeps pointer to the first and the last time it appears in the Euler Tour. The Euler Tour is very good for maintaining information associated to vertices, such as for instance the XOR of the labels of all outgoing edges, as used in [21] to achieve worst-case polylogarithmic dynamic graph connectivity.

2.2.2 Topology trees

Given a ternary graph, $G = (V, E)$, that is, a graph where each vertex has degree at most 3, let T be a spanning tree for G. Given a subset S of V, we say that an edge is *incident* to S if exactly one of its endpoints is in S. We say two subsets are *adjacent* if there exists an edge with an endpoint in each subset. A *vertex-clustering* of T is a partitioning of the vertices into subsets, called *vertex-clusters*, such that

1. A vertex-cluster with external degree 3 (that is, with 3 edges incident to the vertex-cluster) contains only one vertex.
2. No vertex-cluster has external degree > 3.
3. Each vertex-cluster contains 1 or 2 vertices.
4. No two adjacent clusters can be unioned and still satisfy the above.

Item 2. above is not written explicitly in [20], but is a necessary condition.

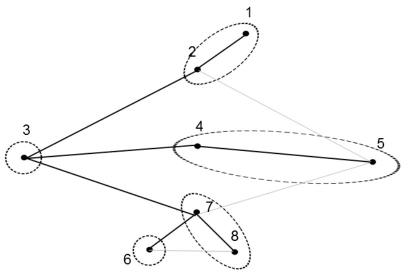


Figure 10: A graph with a spanning tree and a vertex-clustering of the spanning tree. Note that several vertex clusterings of the same tree may exist: In the bottom triangle, one could have chosen the other leaf to be the singleton.

Now, one can consider the clusters as vertices in a new ternary tree, and play the game of vertex-clustering again. Doing this recursively, one obtains a *Topology Tree*: In each level, we have a ternary tree. At level 0, we have the original tree, and the vertices at level $i + 1$ correspond to vertex-clusters on level i. Since each vertex at level $i + 1$ corresponds to one or two vertices at level i, this construction gives a binary rooted tree, where the leaves are the original vertices, the root is the cluster corresponding to the entire graph, and where adjacency describes vertex cluster membership. Topology trees were first described by Frederickson in [7], where it is also proved that the tree shrinks by a factor of 5/6 for each step, yielding a total tree height of $\mathcal{O}(\lg n)$.

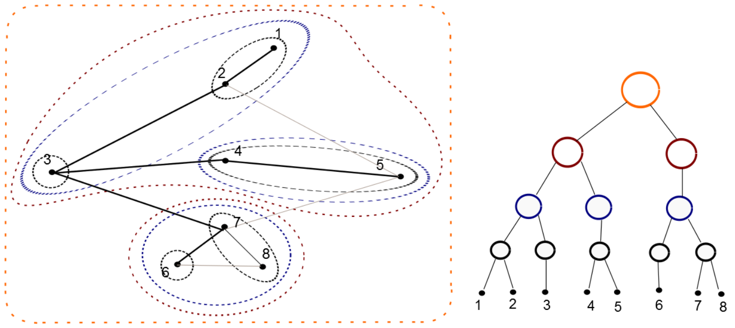


Figure 11: Left: The vertex-clusters at different levels of the topology tree for the tree from before. First, 8 vertices at level 0, then 5 vertex-clusters yielding 5 vertices at level 1, then 3 at level 2 (blue), 2 at level 3 (red), and finally 1 at level 4 corresponding to the whole graph. Right: The topology tree.

To maintain such a topology tree dynamically, upon an update involving a constant set of vertices, one may need to split all clusters in the root path of each vertex involved in the update. Then, at each level, one can greedily merge at most two clusters to maintain invariant 4. If, for example, the edge $e = (u, v)$ is deleted. Then two new trees are created: T_u and T_v. There are two cases (at each level), e is contained in a cluster, or e goes between clusters. If e is contained in a cluster, then that cluster must be split into two new clusters. Each may have to be merged with at most one other cluster to maintain invariant one, and each has at most 2 adjacent clusters to 'check' whether merging would be legal (because the tree was assumed to be ternary). If on the other hand the edge went between clusters, then u and v now have a lower degree, and their (at most 2 each) neighbours must be checked for possible merges.

Since the tree shrinks with a factor of 5/6 for each step, the tree remains height $\mathcal{O}(\lg n)$ during these updates.

2.2.3 Ternary graphs

To obtain a ternary graph, $\hat{G}$, from an arbitrary graph, G, one may choose an ordering around each vertex v of the incident edges, $e_1, \ldots, e_d$, and if $d > 1$ split the vertex into d new vertices $v_1, \ldots, v_d$ such that e_i is incident to v_i and not v, and insert edges (v_i, v_{i+1}). We call $v_1, \ldots, v_d$ the chain representing v.

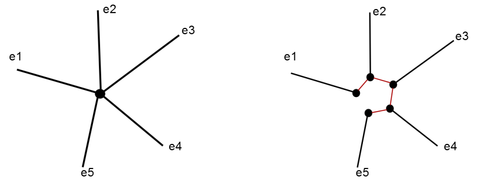


Figure 12: Left: A vertex with 5 incident edges. Right: The vertex splitting.

In this process, we insert at most two new vertices per edge, and between them, less than two new edges (v_i, v_{i+1}) per edge. This gives a total of $\mathcal{O}(n+m)$ vertices and $\mathcal{O}(m)$ edges in the ternary representation of the graph, where the original graph had n vertices and m edges.

If the original graph is a planar embedded graph, one may choose as the ordering around each vertex of the incident edges the ordering given by the embedding. Then, the ternary graph is again planar. When the graph is planar, there are only $\mathcal{O}(n)$ edges, and then, the vertex splitting does not alter, asymptotically, the number of vertices.

2.2.4 Edge-clusters

In order to define top-trees, we define *(edge-)clusters*; the building blocks of top-trees.

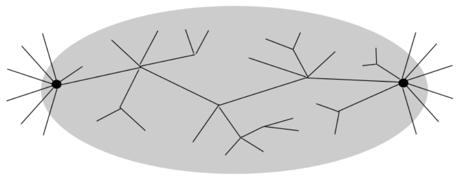


Figure 13: An edge-cluster in a tree with two boundary nodes.

Given a (not necessarily ternary) tree, $T = (V, E)$, we define an *(edge-)cluster* as a connected non-empty subgraph C with at most two boundary nodes, b_1 and b_2, such that any edge (u, v) between $u \in C$ and $v \in T \setminus C$ is incident to one of the two boundary nodes. That is, $u = b_1$ or $u = b_2$. A *clustering* of a tree is a division of the tree into clusters, which only overlap in the boundary nodes, and whose union is the entire tree.

Given a cluster with two boundary nodes, we talk about the *cluster path* as the unique path from b_1 to b_2. We call clusters with two boundary nodes *path clusters* and all other clusters *leaf clusters*.

To visualise this, an edge-cluster in a tree is a subgraph with at most two boundary nodes, which is connected to the rest of the graph via those two boundary nodes (see figure 13).

For plane graphs with a tree-co-tree decomposition, an edge-cluster in the *dual* graph is a connected collection of faces, all surrounded by primal tree edges, such that only the two boundary faces may contain edges that do not belong to the primal tree (see figure 14).

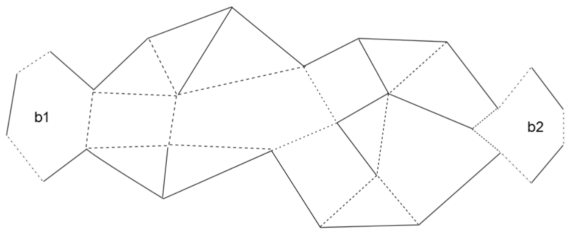


Figure 14: A cluster in the dual graph with boundary faces b_1 and b_2. The primary tree (not all of which is depicted) builds a wall around the cluster, such that all holes in the wall are incident to the boundary faces.

2.2.5 Top-trees

Given a tree T, a top-tree for T is a binary tree whose leaves are the edges of T, and whose nodes are clusters in T, and whose root is T. Any internal node is formed by the merged union of its (at most two) children. Given an orientation of the edges incident to a node, $e_1, \ldots, e_d$, two clusters C_1 and C_2 may only be merged, if there exists a pair of edges $e_i \in C_1$ and $e_{i+1} \in C_2$. That is, the clusters must follow each other in the ordering around their common vertex. If this is the case, we say the nodes are *neighbours*.

The tree is built bottom-up by merging nodes, following some given rules for merging, such that an internal node is the merged union of its children.

To use top-trees as an interface for topology trees, we need to ensure that legal merges in the topology tree of $\hat{G}$ correspond to legal merges of edge-clusters in G. By interface, I mean that merges in the top-tree are determined by merges in the topology tree. In the top-tree, merges are only allowed, if the resulting vertex-cluster has external degree less than 2. Let u_i and v_j denote the boundary nodes of the resulting vertex cluster, C. The vertex cluster now corresponds to an edge-cluster, induced by C, which has u and v as boundary nodes. Assume C is formed by merging C_u and C_v which are connected by an edge, $a = (u_0, v_0)$ (See figure 15). Then the merge in the topology tree may correspond to two merges in the top-tree: **First,** merge C_u with a to form the intermediary cluster I. Since a is incident to C_u, u_0 must be a boundary vertex in C_u. The merged cluster now has v_0 as a boundary vertex in stead of u_0. **Second,** merge I with C_v.

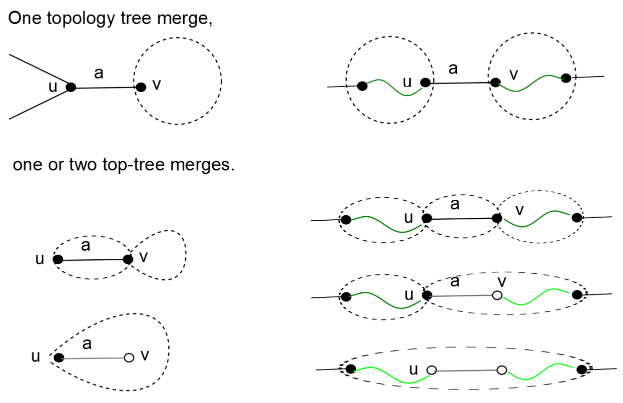


Figure 15: There are two cases for merges determined by the topology tree, given the external degree of the resulting cluster is 2. Either it corresponds to "eating a leaf" (left) or to a path-merge (right). The white dots are those boundary vertices which become internal. Similarly, if the external degree is less than 2, one topology tree merge corresponds to one or two top-tree merges.

This way, the top-tree is twice as tall as the topology tree, which is still $\mathcal{O}(\lg n)$.

Given a planar embedded graph, when forming the ternary tree, we choose an orientation of the edges around the vertex which corresponds to the planar embedding. This way, edge-clusters which share the boundary vertex b will only be merged if correspondingly they include vertices b_i and b_{i+1} in the chain representing b, which means, only neighbours will be merged.

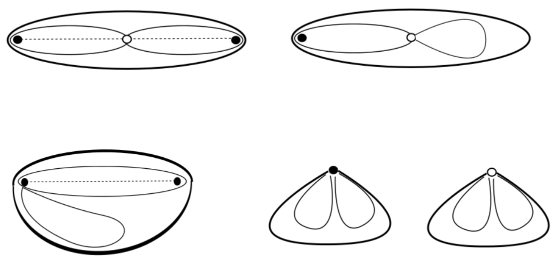


Figure 16: The five different merges of clusters in a top-tree. The white dots are those vertices which were boundary before the merge but cease to be boundary, and the black dots are vertices which are still boundary.

Cluster merges have one of the following five forms: (See figure 16.)

1. The path cluster with cluster-path $u \rightsquigarrow v$ and the path cluster with cluster-path $v \rightsquigarrow w$ are merged to form the path cluster with cluster-path $u \rightsquigarrow w$.
2. The path cluster with cluster-path $u \rightsquigarrow v$ is merged with the leaf cluster with boundary vertex v to form a path cluster with the same cluster-path and with the same boundary nodes, u and v. That is, there are still external edges incident to the vertex v, and v does not cease to be boundary.
3. The path cluster with cluster-path $u \rightsquigarrow v$ is merged with the leaf cluster with boundary vertex v to form the leaf cluster with boundary vertex u. That is, the vertex v ceases to be boundary.
4. Two leaf clusters with boundary vertex v are merged to one leaf cluster with boundary vertex v.
5. Two leaf clusters sharing boundary vertex are merged to form a leaf cluster without boundary vertices.

Dynamically, we allow the following updates to the top-trees over an underlying forest:

1. link(u, v), where u and v are vertices in different trees: link the trees by adding an edge (u, v).

2. cut(u, v): remove the edge (u, v) from the dynamic forest.
3. expose(u, v) where u and v belong to the same tree, T: make u and v boundary vertices in the root of the top-tree over T.
4. expose(v): make v a boundary vertex in the root of the top-tree over its tree T, $v \in T$.

To implement these, the top-tree allows the following modifications

- create(e) and destroy(e), to create or destroy the top-tree with a single cluster consisting of the edge e.
- join(A,B) and split(C), when A and B are neighbours such that $C = A \cup B$. If A and B are roots of top-trees $\mathcal{R}_A$ and $\mathcal{R}_B$, the top-tree $\mathcal{R}_C$ is formed, having $C = A \cup B$ as its root.

  If C is the root of $\mathcal{R}_C$ with children A and B, then C is destroyed and two top-trees $\mathcal{R}_A$ and $\mathcal{R}_B$ are formed.

Along with the top-tree, information about the cluster can be stored. For example, the weight of the lightest edge on the cluster path can be stored: That way, if the root cluster is a path cluster with boundary nodes u and v, we can use binary search to find the smallest weight edge on the path between u and v.

Now, the tree is dynamic, so we may want to e.g. alter the weight of a given edge. We call this a modification of the tree. To modify the edge e, we identify the root-path of e in the top-tree. Then, top-down, perform a sequence of splits of all clusters in the root-path of e. Then perform destroy(e), update the forest, and create e'. Finally, using the join-operation, we recreate the top-tree bottom-up.

To use the top-tree, we will have to specify what happens to information in the case that a cluster is split, and in the case clusters are merged. For instance, when keeping track of the minimum weight edge on a path, the information $i(C)$ stored with cluster C which is the union of path-clusters A and B, is the minimum of $i(A)$ and $i(B)$.

Theorem 2.3 ([1]). *For a dynamic forest we can maintain top trees of height $\mathcal{O}(\lg n)$ supporting each link, cut, or expose with a sequence of $\mathcal{O}(1)$ create and destroy, and $O(\lg n)$ join and split. These top tree modifications are identified in $\mathcal{O}(\lg n)$ time.*

Similarly, with the techniques presented by Alstrup et al. in [1], we may expose any constant number of vertices with $\mathcal{O}(\lg(n))$ joins and splits, identified in $\mathcal{O}(\lg n)$ time.

3 Planarity Testing using Topology Trees

In this section, I will present the results of Italiano et al. [20].

To maintain a planar embedding of a graph using topology trees, it is necessary to keep track of edge bundles, coverage bits and face lists.

3.1 Edge bundles and targets

Recall that the topology tree is a binary tree whose root is the entire connected component, and whose leaves are the vertices, and whose internal nodes are clusters, such that when a level i cluster is formed as the union of level $i-1$ clusters, this is reflected by the parenthood of the level i cluster to the level $i-1$ clusters it is formed by.

Definition Given vertices u and v, the *`lca`-target* for the edge (u, v), denoted $C_{u,v}$, is the cluster in the topology tree that is the nearest common ancestor of the leaves u and v. Given a cluster, C, in the topology tree, an *edge bundle* is a maximal sequence of edges incident to C that share the same `lca`target.

Note that if $C_{u,v}$ is the `lca`-target for u and v, then it must have two children, C_u and C_v, such that $u \in C_u$ and $v \in C_v$. The edge bundle containing (u, v) must consist of edges between C_u and C_v.

A topology tree may be expanded at a constant number of given vertices, e.g. the u and v we want to find a common face for. That the tree is expanded means the clusters containing those given vertices are all split, and then at the "top level" one has a graph of clusters and edges between them. This is called the expansion topology tree, or the expanded topology tree.

Definition Given a cluster C of the expansion top tree and a vertex $u \in C$, then the *precise target* for the edge (u, v) is the cluster in the expanded topology tree is the cluster containing v. Given a cluster, C, in the expanded topology tree, an *edge bundle* is a maximal sequence of edges incident to C that share the same precise target.

For an edge bundle (of either form), the following information is maintained: The edge count, the first and last edge (following the orientation), and the target cluster of the bundle. The edge bundles are represented as balanced binary trees, and upon split or merge of a clusters, they are split or merged. It is a lemma that there are no more than $\mathcal{O}(\lg(n))$ edge bundles (see Lemma 3.1), which means that splitting or merging takes $\mathcal{O}(\lg \lg(n))$ time.

3.2 Coverage bits and face lists

For a cluster, C, with two boundary vertices, that is, with external degree 2, the cluster path $p(C)$ denotes the unique path between the two boundary vertices. By the projection $\pi(v)$ of some vertex v to the cluster path, we mean the usual meet$(b_1, b_2, v) = \texttt{lca}_v(b_1, b_2)$, where b_1 and b_2 are the boundary vertices. For a cluster C with one boundary vertex b, we set $p(C) = \{b\}$.

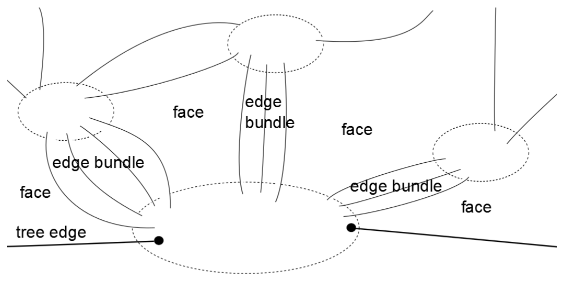


Figure 18: A cluster of external degree two keeps track of an edge bundle list and a coverage list to either side of the cluster path. A cluster of external degree one just keeps track of a coverage list and an edge bundle list. In the picture we see an example of edge bundles and faces incident to the northern side of a cluster with two boundary nodes.

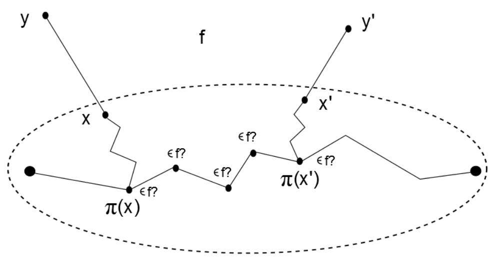


Figure 17: The face, f, has a coverage bit of 1, if any of the vertices on the cluster path between $\pi(x)$ and $\pi(x')$ belong to f.

A face is incident to a cluster if it is incident to at least one vertex in the cluster and at least one vertex not in the cluster. The *coverage bit* in the cluster C of a face f incident to C is one if and only if there exist edges (x, y) and (x', y') incident to f with $x, x' \in C$ and $y, y' \notin C$, such that there exists a vertex v_p on the cluster-path between $\pi(x)$ and $\pi(x')$ which is incident to f (see figure 17). A face incident to a cluster C and a cluster C' is called a *good face* if the coverage bit for f is one, both in C and in C'. If a face is not a good face, then no edge can possibly be inserted between vertices $v \in p(C)$ and $v' \in p(C')$.

Each bundle, B, of edges between clusters C and C' keeps track of the good faces which lie between edges in the bundle. This is called the *face list* of B. Each cluster keeps track of the coverage bit of all those incident faces that lie between bundles, or between the spanning tree and a bundle, in the embedding (see Figure 18). The list of such faces with coverage bit 1 is called the *coverage list* for the cluster. There are at most $\mathcal{O}(\lg(n))$ faces in the coverage list, according to Lemma 3.1, when the topology tree is expanded in a constant number of vertices.

For clusters of external degree 2, we maintain two face lists and two edge bundle lists, one for each side of the cluster path. For clusters of degree 1, we maintain only one face list and one edge bundle list. In either case, the order in the list follows the counter-clockwise ordering of the edges incident to the cluster in the planar embedding.

Again, a balanced binary tree is used for maintaining these lists, and thus, for coverage lists, a split or merge can be done in time $\mathcal{O}(\lg\lg(n))$. For an edge bundle, there can be up to $\mathcal{O}(n)$ faces in the face list, and thus, a split or merge can be done in time $\mathcal{O}(\lg(n))$.

3.3 Maintaining lists and bundles

Although edge bundles, coverage lists, and face lists are defined for all clusters in the topology tree, it proves more efficient to maintain only a part of this information at each cluster, and with each cluster a *procedure* for passing information to its children in case the cluster is split.

Of course if the cluster has one child, there is no difference between the bundles and lists of the parent, and those of the child. If the cluster has two children, the recipe depends on the degree of the children, and upon some properties of the graph.

The different cases for splitting a cluster C are: (See figure 19)

1. C has one child. (Then the lists are identical)
2. C has children Y and Z such that Y has degree 3 and Z has degree 1. (Then there are no non-tree edges incident to Y, since the graph is ternary.)
3. C has two children which both have degree 1 (this only happens when C is the root).
4. C has children Y and Z such that Y has degree 2 and Z has degree 1. In this case, there are two subcases: (4b) If Y has a self-loop, then Z cannot be connected to the rest of the graph, that is, Z has no edge bundles. (4a) Otherwise, Z may have non-trivial edge bundle list and face list.
5. C has two children, Y and Z, both of degree 2. In this case, the children will have lists on either side of the spanning tree. Call those the northern and the southern side. Then there may exist bundles between the northern sides of Y and Z and between the southern sides of Y and Z. Additionally, there may be either an edge bundle from the northern side of Y to the southern side of Z, *or* from the northern side of Z to the southern side of Y, but *not* both, or it would violate planarity.

In the recipe, information about the edge bundles that go between children is stored (blue lines in Figure 19), as is (for 4a and 5) *location descriptors* of where the lists of C must be split to create the lists of C's children (in Figure 19, this corresponds to remembering where in the original ordering of edges around C the edges change colour from red to black.).

For details, I refer to [20], and skip to the main lemmas:

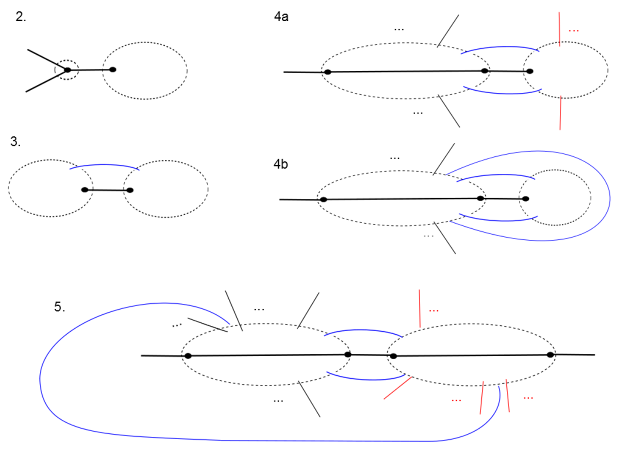


Figure 19: The four cases for splitting a cluster in two. The blue lines are edge bundles that go between the sibling clusters.

Lemma 3.1 ([20]). *The expansion of a topology tree at a constant number of vertices takes time $\mathcal{O}(\lg(n)^2)$ and creates $\mathcal{O}(\lg(n))$ clusters, edge bundles and coverage bits.*

Furthermore, when the topology tree is expanded at a constant number of vertices, it is possible to compute the relevant set of precisely targeted edge bundles in time $\mathcal{O}(\lg(n)^2)$. Together, this entails the following lemma:

Lemma 3.2. *The cost of expanding the topology tree at a constant number of vertices to obtain a cluster graph is $\mathcal{O}(\lg(n)^2)$.*

3.4 Queries and updates

We want to know if the edge (u, v) may be inserted without violating planarity, or equivalently, if u and v belong to a common face. To answer a query of (u, v), expand the topology tree in the first and last vertex on the chain of u and of v in the ternary representation of the graph. This takes time $\mathcal{O}(\lg(n)^2)$ (Lemma 3.2). Let a *u-cluster* denote a cluster that contains one of the vertices on the chain of u, and similarly for v. The edge may be inserted iff:

- There is a face in the cluster graph with coverage bit of 1 both at a u-cluster and at a v-cluster, or
- There exists an edge bundle between a u-cluster and a v-cluster whose face list is nonempty.

This can be tested in time $\mathcal{O}(\lg(n))$ (Lemma 3.1). Finally, the splitting of the expose operation are reversed.

Theorem 3.3. *The query takes time $\mathcal{O}(\lg(n)^2)$, dominated by the expose-operation.*

To update the graph, there are several cases.

To insert an edge between two vertices u and v in the same component, the query-operation is first performed to test if this is at all possible. If it is possible, the query operation can return the place in the chains of v and u where the edge would fit. The chains are updated with a new node, and the edge is inserted. This creates a new edge bundle, and two new faces, both with coverage bit 1. In short, deleting a non-tree edge does the opposite: Two faces are removed and one new with coverage bit one is created.

When a tree edge, e, is deleted, a replacement edge is sought. The topology tree is expanded at the endpoints of the edge. If e was not a bridge, a new reconnecting edge must be incident to one of the two faces incident to e. Finding a replacement edge (bundle) can be done in time $\mathcal{O}(\lg(n))$. Let e' denote the replacement edge. Now, e is swapped with e' as a tree edge, and then e is deleted just like a non-tree edge.

Theorem 3.4. *[20] Edge deletions and insertions take time $\mathcal{O}(\lg(n)^2)$.*

4 New approach for planarity

In this section I will present a data structure for dynamic planarity testing of an embedded graph using top-trees.

To use top-trees for this purpose, we have to make some smaller extensions and adjustments.

Basically, we keep track of the Extended Euler Tour (eET), the primal top-tree, and the dual top-tree. We use the dual top-tree to keep track of information about the primal top-tree, namely, whether corners are incident to exposed vertices. In order to convey this information in a top-tree setting, we need to be able to expose corners. First, I will present these tools: the eET, exposing corners, and a variation of top-trees called *slim-path* top-trees.

Then, the primal and dual top-trees are described, as is how they work together. I describe insert and delete of edges.

Finally, I describe how to find common faces of two given vertices, that is, the query operation. To describe this, or rather to prove correctness, we need to realise some little lemmas about planar embedded graphs.

Using the tree-co-tree decomposition of a dynamic planar graph was also used as a technique by Klein in a data structure for multiple source shortest paths in planar embedded graphs [24].

4.1 The Extended Euler Tour

The eET consists of objects that are corners or edges, alternatingly. Each object has a successor and a predecessor pointer, like an oriented double-linked cycle.

Additionally, each corner also points to the place in each tree where they become unexposed, that is, where the corner for the first time becomes internal to an edge-cluster.

In order to point out a segment of the eET, we just need to point to the starting place and the ending place of the segment.

4.2 Exposing corners

Given a top-tree implemented by reduction to topology trees, we can allow the user to expose not only vertices, but also incident corners. The result of such an operation should be a top-tree which in the top has (up to) two clusters: One path-cluster in case the corners are incident to different vertices, or two leaf clusters in case the corners are incident to the same vertex.

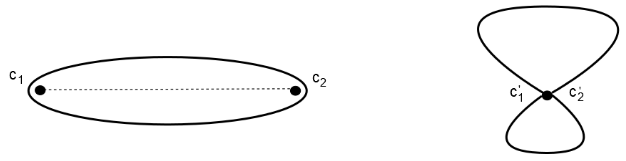


Figure 20: Exposing corners. Left: corners incident to different vertices. Right: corners incident to the same vertex.

In the ternarisation of the graph, when a vertex v with incident edges $e_1 \dots e_d$ is substituted by $v_1, \dots v_d$, all corners but one correspond to edges between v_i and v_{i+1} (See figure 12). The last corner is the one between e_1 and e_d and is automatically exposed. If we want to expose corners corresponding to some links $v_i \leftrightarrow v_{i+1}$, we simply delete that given link (or those given links). Once one or two links are deleted, we join $v_1 \leftrightarrow v_d$. In this way, we can split and cut the vertex v into two vertices v and v'. In this case, after the top-tree has rebalanced itself in accordance with this underlying ternarisation, one of the two links are relinked. See figure 21.

In the top-tree, exposing two corners, c_1 and c_2, incident to u and v, respectively, is done in the following way. First, all clusters on the root-path of each corner are split. That is, clusters containing u or v as a non-boundary node are split, and then, all clusters containing c_1 or c_2 are split. That is, we split those clusters where c_1 or c_2 is internal, not those where they are delimiting. Then, clusters are merged again. But now, when merging as part of an expose(u, v) operation, the top-tree adheres to the new enumeration of edges incident to u or v. This is done with $\mathcal{O}(\lg(n))$ splits and merges, and results in a path with c_1 and c_2 exposed.

If in stead we want to expose two corners c_1 and c_2 both incident to v, we do the following. (See Figure 21 and Figure 20 (right).) First, all clusters on the root-path of the corners are split. Then, clusters are merged again, but in the underlying ternary graph, the chain of v is split in two, meaning the tree is

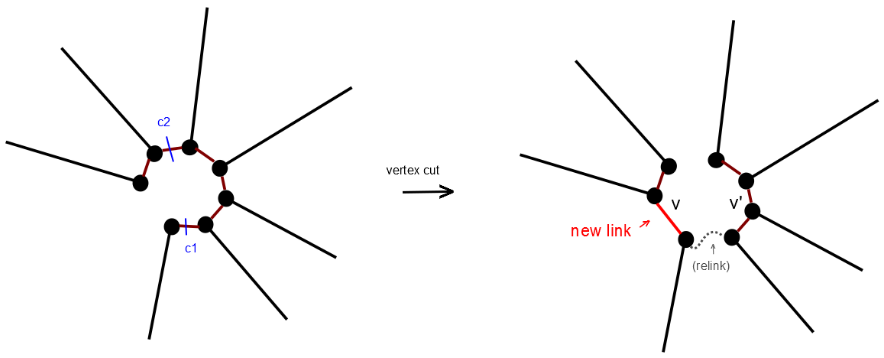


Figure 21: To expose two corners incident to the same vertex, we delete the corresponding links, and make a new link $v_1 \leftrightarrow v_d$. We have now cut the vertex in two, forming v and v'. After having exposed v and v' in their respective top-trees, we join them again by relinking one of the two exposed corners.

split in two. When we merge again, the merges corresponds to exposing each copy of "v" in either of the two resulting trees. Finally, the two copies of "v" are glued together (relinked), and the result is a top-tree with two leaves in the top, exposing v and the two corners c_1 and c_2. This is done with $\mathcal{O}(\lg(n))$ splits and merges.

4.3 Slim-path top-trees and four-way merges

We can tweak the top-tree such that the following property, called *the slim path invariant*, is maintained:

- for any path-cluster, $u \rightsquigarrow v$, the only edge incident to a boundary node is that belonging to the cluster path.

This can be done by allowing the following merge: Given path-clusters $u \rightsquigarrow v$ and $v \rightsquigarrow w$, and given two leaf-clusters with boundary vertex v, allow the merge of those four, to obtain one path cluster $u \rightsquigarrow w$. For lack of a better word, I call this a *four-way merge*.

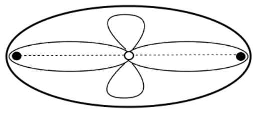


Figure 22: Four-way merge of paths and leaves.

We allow the following merges:

- Two leaves merged to one
- A path cluster $u \rightsquigarrow v$, and a leaf cluster with boundary vertex v, merge to form a leaf with boundary vertex u.
- Two paths $u \rightsquigarrow v$, $v \rightsquigarrow w$, and up to two leaves with boundary node v make a four-way merge to form the path $u \rightsquigarrow w$.

Since the slim-path invariant has no restriction on leaves, clearly the first two do not break the invariant. The four-way merge does not break the invariant: If to begin with there was only one edge incident to u and only one incident to w, this is still maintained after the merge.

Definition A *slim-path top-tree* is a top-tree which maintains the slim-path invariant by allowing four-way merges.

Lemma 4.1. *It is possible to maintain a slim-path top-tree over a dynamic tree with height $\mathcal{O}(\lg(n))$, where n is the number of vertices.*

Proof. Proof by reduction to ordinary top-trees (see figure 23). For each path cluster in the ordinary top-tree, keep track of a slim path cluster, and up to four internal leaves; two for each boundary node, one to each side of the cluster path. In the slim-path top-tree, this would correspond to four different clusters, but this doesn't harm the asymptotic height of the tree. When merging two path clusters in the ordinary top-tree, with paths $u \rightsquigarrow v$ and $v \rightsquigarrow w$, this simply correspond to two levels of merges in the slim-path top-tree: First, the incident leaf clusters are merged, pairwise, and then, we perform a four-way merge. Other merges in the ordinary top-tree correspond exactly to merges in the slim-path top-tree. Thus, each merge in the ordinary top-tree corresponds to at most 2 levels of merges in the slim-path top-tree, and the maintained height is $\mathcal{O}(\lg(n))$. □

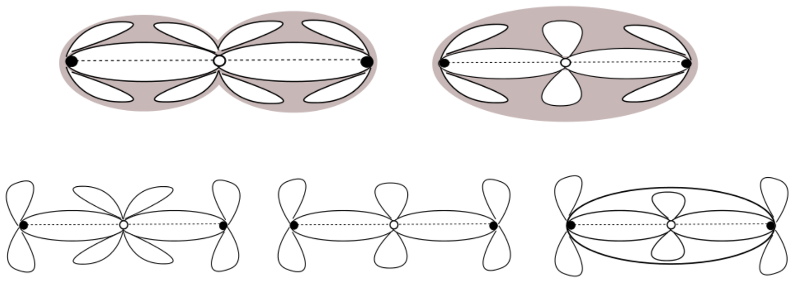


Figure 23: One merge in ordinary top-trees corresponds to two merges in slim-path top-trees.

Note that four-way merges and the slim-path invariant are not necessary. One could simply maintain within each path cluster, information about the internal leaf clusters incident to each boundary node; one to each side of the cluster path. I present four-way merges because they provide some intuition about this particular form of usage of top-trees.

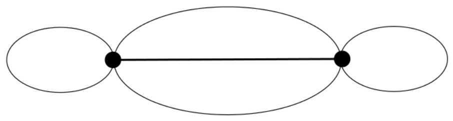


Figure 24: In a slim-path top-tree, expose(u, v) exposes a path and two leaves.

We also note that the expose(u, v)-operation is different in a slim-path top-tree. In stead of returning a top-tree which has at its top a path cluster with path $u \rightsquigarrow v$, the top-tree has at its top a path cluster with path $u \rightsquigarrow u$, and, up to two leaf clusters, one at each exposed vertex.

4.4 The primal and the dual top-tree

For each connected component of a dynamic planar embedded graph, we maintain two top-trees. One for the primal, and one for the dual tree of the tree-cotree-decomposition of the embedded graph component. We call these the *primal top-tree* and the *dual top-tree*, respectively.

When calling "expose" on two vertices u, v in the primal top-tree, we would like to be able to reply whether u and v share a face, and if so, return the list of all incident corners to that face - the corners corresponds to those gaps between existing edges where an edge between u and v can be inserted.

We do this by using the notion of a deactivation count of all corners. We do not keep track of the deactivation count explicitly, but use the top-tree to do it in a lazy way. The deactivation count has the following property: If a vertex is internal to the merge of clusters in the primal top tree, the deactivation count of all incident corners is increased. Finally, after exposing u, v, all corners incident to u and v will have a count of 0, while other corners have a count > 0.

Definition Given a vertex, v, we define the *deactivation depth* of v as follows: In the primal top-tree, let C_0 be one of the first clusters from the root where v is a boundary vertex, and let $C_0, C_1, \ldots, C_t$ be the root path of that cluster. We now simply count how many times C_i is a leaf vertex, whose boundary node b_i is not boundary in C_{i+1}, and this count is the deactivation depth of v. (See figure 28.)

Definition Given a corner c incident to the vertex v, its *deactivation count* is the deactivation depth of v.

4.4.1 The dual top-tree.

For a path cluster with path $u \rightsquigarrow v$ in the dual tree, incident corners fall in two categories: those right of the cluster path $u \rightsquigarrow v$ (southern corners), and those left of the cluster path $u \rightsquigarrow v$ (northern corners).

Definition For a path-cluster in the dual tree, let $s_{\min}$ denote the minimum deactivation count for internal incident southern corners, and let $n_{\min}$ denote the minimum deactivation count for internal incident northern corners. A corner is *incident*, if it is incident to a face in the cluster. A corner is *internal*, if it is either incident to a non-boundary face in the cluster, or if it is incident to a boundary face and has co-tree edges on both sides.

Definition For a path-cluster in the dual tree, a *good* face is a non-boundary face on the cluster path, which has incident southern corners of deactivation count $s_{\min}$ and incident northern corners of deactivation count $n_{\min}$.

Given a cluster, we can keep track of delimiting corners for that cluster. A *delimiting corner* is a corner which is between one edge in the cluster and one edge not in the cluster. The extended Euler Tour of the whole tree is denoted as the *universal eET*. We will denote the sublist delimited by a pair of corners as an *ET segment*. Note that neighbouring clusters are exactly those that share a delimiting corner.

In the dual tree, we store the following information along with each cluster:

- For a path cluster $u \rightsquigarrow v$,
  - we store the minimum deactivation count for southern corners, $s_{\min}$, and for norhtern corners, $n_{\min}$.
  - we maintain the list of good faces. For each such face, store two lists of minimally deactivated corners.
  - for each boundary vertex, v, we maintain two minimum values $s_{\min}(v)$ and $n_{\min}(v)$ of deactivation counts for incident internal corners, and two lists of minimally deactivated corners, one to each side of the path.
  - we store two pairs of pointers to corners in the universal extended Euler Tour, defining segments of the extended Euler tour: One from u to v, and one from v to u. We will also keep track of two "middle pointers", which point to delimiting corners of the children, in case the cluster is split.
  - we maintain two lazy δ values, δ_s and δ_n, for altering the deactivation counts, one for the southern Euler Tour, and one for the northern one.
- For a leaf cluster,
  - we store the minimum deactivation count for corners incident to the boundary vertex, $c_{\min}$.

- we maintain a list of incident corners with minimum deactivation count.
- we store one pair of pointers to corners in the universal Euler tour, corresponding to delimiting the extended Euler tour of the cluster, rooted in the boundary node. We keep track of one "middle pointer", which points to the shared corner of the children in case the cluster is split.
- we maintain a lazy δ value for altering the deactivation counts stored with its children in case it is later split.

When clusters are split, we update their delimiting corner with its corresponding δ-value.

In the dual top-tree, when clusters are merged, their Euler Tours are concatenated.

Note 4.2. *The Euler Tour segments of each cluster are well maintained under split and merge.*

Proof. We walk through all cases of merge, and split is similar.

- In case two leaf clusters are merged to a new leaf, one Euler Tour starts with the corner c and ends with c': cTc', and the other must start with c' and ends with c'': $c'T'c''$. If they did not share a corner, the two were not neighbours, and could not be merged. The resulting extended Euler Tour (eET) must then be $cTc'T'c''$ (if $c \neq c''$). If per assumption the two first Euler Tour segments corresponded to the unique segments of the universal eET starting with their respective corners, then, inductively, this concatenation returns the unique segment of the universal eET starting with c and ending with c''.

  If $c = c''$, the result is the entire eET of the connected component, the linked cycle $cTc'T'$.

- In case the path cluster P with path $u \to v$ is merged with the leaf cluster L with boundary vertex v to form a leaf cluster with boundary vertex u, then there are two eET segments in P: $c_0T_0c_1$ and $c_2T_2c_3$. In L there is one segment: $c_1T_1T_2$. Note that the corners that L's segment starts and ends with are the same as those that occur in the Euler Tours of P, otherwise the result would not be a leaf, as there would still be incident edges to v, and v could not cease to be boundary. Then the resulting eET is $c_0T_0c_1T_1c_2T_2c_3$. Again, if the original segments corresponded well to the universal eET, so will the result.

- In case the path cluster P is merged with the leaf cluster L to form a new path cluster, let the segments in P be denoted $c_0T_0c_1$ (from u to v) and $c_2T_2c_3$ (from v to u). Since the two clusters are merged, they must be neighbours, which means that L must share a corner with P. Without loss of generality we can assume L is incident to v, say, sharing the corner

c_1, such that the segment stored in L is of the form $c_1T_1c_4$. Then, the new segment from u to v is $c_0T_0c_1T_1c_4$, while the other segment stays intact.

- In case two path clusters are merged to a path cluster, then they must share both delimiting corners incident to the vertex, v, which ceases to be boundary. If they did not share one corner, they would not be neighbours, and if they didn't share both corners, the result would have not two but three boundary nodes - not a cluster. Therefore, if one path cluster has eET segments $c_0T_0c_1$ and $c_4T_3c_5$ and the other has eET segments $c_1T_1c_2$ and $c_3T_2c_4$, then the merged path cluster has eET segments $c_0T_0c_1T_1c_2$ and $c_3T_2c_4T_3c_5$.

□

Lemma 4.3. *We can maintain the list of "good" faces of path clusters during splits and merges. We can also maintain the minimum deactivation counts.*

Proof. When two clusters are merged, then depending on their respective minimum deactivation count(s), there are different strategies.

- When a path cluster P and a leaf cluster L are merged to form a path cluster, then let $s_{\min}$ and $n_{\min}$ denote the minimum deactivation count for P, and let $c_{\min}$ denote the minimum deactivation count for L. Since the two are neighbours, they must share a corner, so assume this corner is one of the southern corners to P incident to v, and let c denote its deactivation count.

  - If $s_{\min} \leq \min\{c, c_{\min}\}$, then the face list of the merged cluster is equal to that of P.
  - If $s_{\min} > c_{\min}$, or $s_{\min} > c$, then the face list of the merged cluster is empty, and the list of P stays in P.

  Depending on the values of c, $c_{\min}$, and $s_{\min}(v)$, the value of $c_{\min}(v)$ and the corner-list is updated. As above, lists are merged in case of equality, and in case of inequality the smallest are included on the higher level and the larger ones stay at the lower level.

- When two path clusters $P : u \rightsquigarrow v$ and $P' : v \rightsquigarrow w$ are merged, then they have some values $s_{\min}$ and $n_{\min}$ for one cluster, and $s'_{\min}, n'_{\min}$ for the other cluster. They must share two corners, one on each side of the central node, s and n. Let c_s and c_n denote their deactivation counts. (See figure 25.)

  We have 3 pairs, (c_s, c_n), $(s_{\min}, n_{\min})$, and $(s'_{\min}, n'_{\min})$, associated with v, and the lists of L and L', respectively.

  If a collection of these pairs have the property that they are equal to each other and less than all the other pairs (on both coordinates), then the face list after the merge is the concatenation of the lists associated with these pairs, and possibly v is also included.

If no such collection exists, then v is the only candidate for the resulting list.

In either case, v is included if it contains southern corners of deactivation count $s_{\min}^{new} = \min\{c_s, s_{\min}, s'_{\min}\}$ and northern corners of deactivation count $n_{\min}^{new} = \min\{c_n, n_{\min}, n'_{\min}\}$. This is determined by examining $c_s, s_{\min}(v), s'_{\min}(v)$, and $c_n, n_{\min}(v), n'_{\min}(v)$.

- When two leaves, L and L' are merged, if they share the corner c, then the new minimum deactivation count is the minimum of $c_{\min}$ (from L), $c'_{\min}$ (from L'), and c. Depending on whether they are the same, or which is smaller, the corner list will include all or some of them.

- When a path cluster, P, with path $u \rightsquigarrow v$, and a leaf cluster with boundary vertex v are merged to form a leaf cluster with boundary vertex u, then the resulting minimum deactivation count and corner list, are those associated with $u \in P$. The rest stay at the lower level. (See figure 26.)

Note also that each face appears only once as a good face in the final top-tree. □

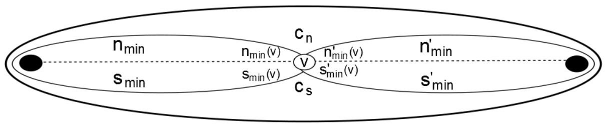


Figure 25: When two paths are merged, the resulting face-list may contain the face-list of either or both or none of them. The central node may or may not be included.

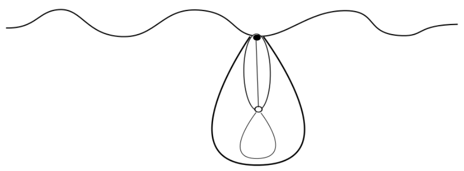


Figure 26: When a path becomes part of a leaf, its face-list becomes irrelevant.

4.4.2 The primal top-tree.

We chose to employ the four-way merge strategy and maintain slim paths in the primary top-tree. Recall, we want to deactivate all corners if they are incident to a non-boundary node. This information is maintained in the following way, in all three cases of merge in the slim-path primary top-tree.

- When two leaf-clusters are merged to a new leaf with the same boundary node, we do not do anything.
- When the path cluster $u \rightsquigarrow v$ is merged with the leaf cluster L with boundary vertex v to form a leaf cluster with boundary vertex u, then the deactivation count of the corners incident to v must be increased. Since we maintain the slim-path invariant, we know that all of those will be in the leaf cluster L. Then, we can look at the cluster path $u \rightsquigarrow v$ which has the edge e_l incident to v. Let f and f' denote the endpoint faces of e_l^*, and let c and c' denote the corresponding incident corners.

  In the dual top-tree, we should expose f and f', along with the incident corners, c and c'. After the expose, all corners on the extended Euler tour from c to c' should have a "plus one" – this is done by updating the lazy δ-value for the root of the dual top-tree after the expose.
- Upon a four-way merge, of the path clusters $u \to v$ and $v \to w$, with two leaf-clusters, U and L, incident to v, let e_l and e'_0 be the cluster path edges incident to v. Let f and f' be the endpoint faces of e_l^* (corners c_0 and c_1), and let g and g' be the endpoint faces of $e_0'^*$ (corners c_2 and c_3), such that f and g are incident to U and f' and g' are incident to L.

  We now need to make two exposes: one exposing f' and g' with the corners c_1 and c_3, and one exposing g and f along with the corners c_2 and c_0. After each expose, we update the lazy delta-value of the root, increasing the deactivation count of all incident corners.
- Finally, when two leaf-clusters are merged to a new leaf with no boundary node, all corners are deactivated, or, correspondingly, the root of the dual tree is updated with a δ++.

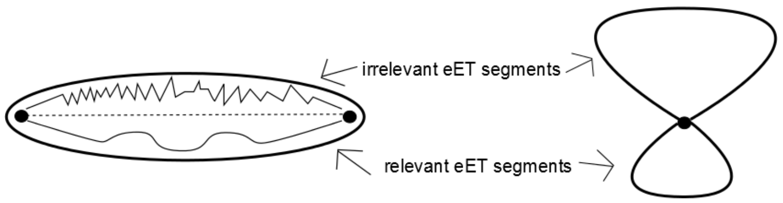


Figure 27: When a vertex ceases to be boundary, then we do the procedure of exposing some corners in the dual tree, once or twice. Depending on whether these corners belong to different faces (left) or the same face (right), we end up with a path or two leaves. This gives us, in either case, two extended Euler Tour, and we need to deactivate the corners belonging to one of them.

Lemma 4.4. *When a vertex v ceases to be boundary, then all corners incident to v are de-activated.*

Proof. There are two non-trivial cases. Either v ceases to be boundary after the path cluster $u \to v$ is merged with the leaf cluster L, or v was the central node in a four-way merge.

In the first case, all corners incident to v are either the exposed two corners, c_1 and c_2, or internal in L. Note that in the primal tree, the extended Euler Tour of the leaf L is the sub-list of the universal eET delimited by c_1 and c_2. But since, according to Lemma 2.2, the extended Euler Tour of the primal tree is identical to that of the dual tree, we just need to ensure we have the same delimiting corners exposed in the dual tree, and, if the top is a path, that we choose the right one of the two delta-values to increase.

In the second case, the central node, v, ceases to be boundary. Since both path clusters are slim by assumption, all corners incident to v are incident to U or L; either they are internal, or they are the ones we expose. But then the same argument as above applies.

The trivial case is when two leaf clusters are merged to form a leaf cluster without a boundary node, but then, all corners on all of the eET are deactivated. □

If we follow these rules when we build the tree, initially, then all corners are deactivated, and their deactivation number corresponds to how deep in the tree their vertex ceases to be boundary.

Upon split, we do the opposite: If a new vertex, v, becomes boundary, then the corners incident to v are re-activated, and their deactivation count decreased. Depending on whether the split were a four-way split or a path-leaf split, we do the procedure of exposing two corners and reactivating their eET segments once or twice.

Lemma 4.5. *By using these rules for split and merge, the deactivation number for a corner incident to a vertex v equals the deactivation depth of v.*

Proof. The proof of this lemma is similar to that of Lemma 4.4; it follows from the definitions and from Lemma 2.2. □

Lemma 4.6. *After exposing u and v in the primary tree, all corners not incident to u or v will have deactivation counts > 0, and those incident to either u or v will have a deactivation count of 0.*

Similarly, if we only expose one vertex v in the primary tree, then all corners not incident to v will have deactivation count > 0, and those incident to v will have deactivation count 0.

Proof. This follows from Lemma 4.5. □

4.5 Updates to the graph

When the graph is updated, by the insertion or deletion of edges, we need to make updates to the top-trees. When edges are inserted or deleted, the number of faces does not stay constant, even if the number of vertices stays constant. For this reason, we need to define the operations of cut and join of faces.

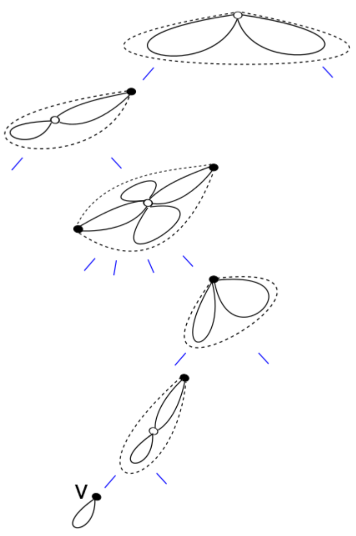


Figure 28: The root path of the highest cluster with v as a boundary node in the primal top-tree. v has deactivation depth 4.

Insert has two cases, depending on whether the inserted edge is to be added to the primary tree (that is, it connects two components) or the dual tree. Symmetrically, there are two cases for delete, depending on whether we delete a bridge, or we must find a replacement edge.

4.5.1 Cut and join of faces.

Given a face f and two corners incident to the face, say, c_1 and c_2, we may cut the face in those corners, producing new faces f and f'. We go about this in the same manner as expose(c_1, c_2) in the dual tree, except we do not re-link the two chains for f, but rename them as f and f'.

Similarly, faces may be joined. Given a face f and a face g, a corner c_f incident to f and a corner c_g incident to g, we may join the two faces to a new face h, with two new corners c_1 and c_2, such that all edges incident to h between c_1 and c_2 are exactly those from f, in the same order, and those between c_2 and c_1 are those from g. We do this by exposing f and c_f in one top-tree, and g and c_g in the other top-tree, and then by the link and join operation.

4.5.2 Inserting an edge

To insert an edge between two components, we must add it to the primary tree. Given a vertex v with an incident corner c_v to the face g, and a vertex u with an incident corner c_u to the face f, we may insert an edge a between u and v.

In each primal top-tree, we expose v, c_v and u, c_u, respectively. In the dual top-tree, we expose g, c_v and f, c_u, respectively. We then link u and v, and update the eET correspondingly: New corners $c_1, \ldots, c_4$ are formed, such that the new edge $a = (u, v)$ appears the successor of c_4 and the predecessor of c_1, and the successor of c_2 and the predecessor of c_3 (see figure 29). That is, we may view the top tree in its current form as the top-tree with an exposed path (u, v) which is trivial (consisting of one edge) and two leaves, one at each end.

In the dual tree, the faces f and g are simply joined at the incident corners c_u and c_v, respectively.

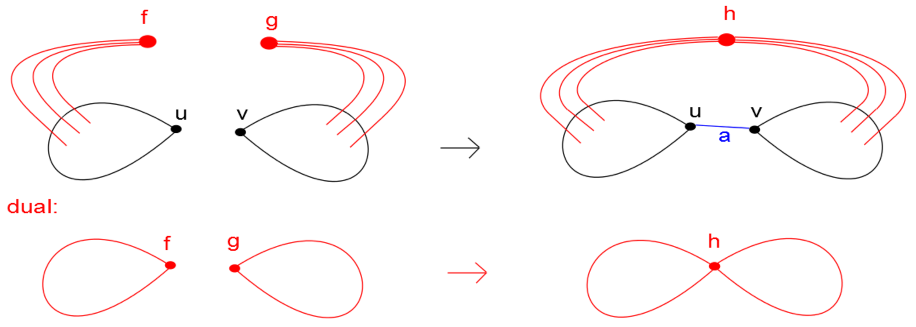


Figure 29: The component of u is joined with the component of v by the edge a, and new corners are formed.

To insert an edge inside a component, the two vertices must belong to the same face. Let u and v be vertices with corners c_u and c_v incident to a face f. We can then expose the corners c_u and c_v, which gives us a path cluster in the primal top-tree (and some leaves, because we use slim-path top-trees), and in the dual tree, exposing those corners gives us two leaves, both incident to f.

We now create new corners $c_1, \ldots, c_4$, and update the eET to list the new edge, $a = (u, v)$, as the successor of c_1 and predecessor of c_2, replacing c_v, and as the successor of c_3 and predecessor of c_4, replacing c_u (see figure 30). In the dual tree, the face f is split in the corners c_u and c_v, creating new faces g and h, and a new edge (g, h) is added to the dual tree.

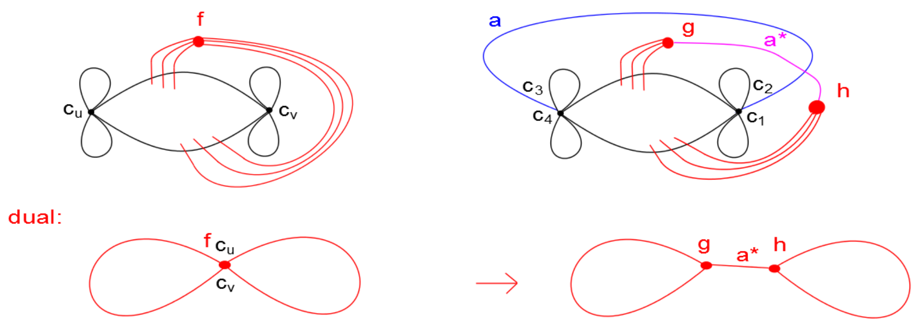


Figure 30: The corners c_u and c_v are exposed, and an edge $a = (u, v)$ (blue) is inserted in those corners. The dual of a, namely $a^* = (f, g)$ (pink), is inserted in the dual tree to connect the two new faces, that appear when the face f is cut in two by the insertion of the edge a.

4.5.3 Deleting an edge

There are three cases, depending on whether the edge to be deleted is a bridge, a non-tree edge, or a replacement edge exists.

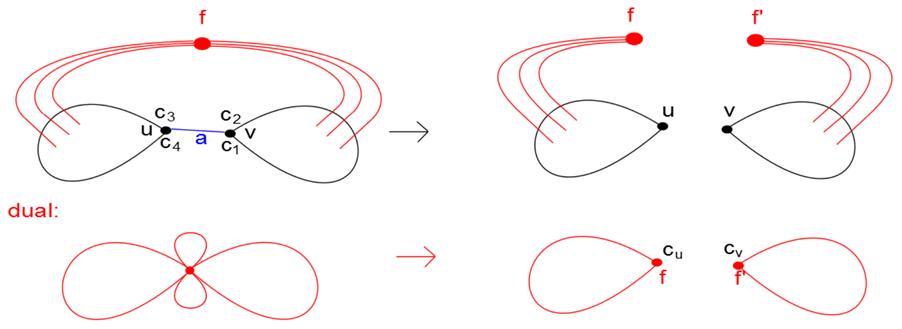


Figure 31: The bridge (u, v) is deleted.

An edge, e, is a bridge if its dual e^* is a self-loop of some face. To delete a bridge $e = (u, v)$ incident to the face f in corners $c_1, \ldots c_4$ (see figure 31). Expose u and v in the primary tree. Expose the corners $c_1, \ldots c_4$ in the dual tree, meaning we have at the top a four-leaf clover of clusters incident to f. Cut f into those four parts, and delete the leaves corresponding to c_4ec_1 and c_2ec_3. Let c_u denote the newly formed corner incident to u and c_v that incident to v. We have now split the face f in two, and have two corresponding eETs and two dual top-trees. In the primal top-tree, delete the edge u; this leaves us with two top-trees, on with a leaf with boundary vertex v exposed and one with boundary vertex u exposed – update the corresponding corners to be exactly the c_v and c_u created in the dual top-tree, respectively.

If a non-bridge tree-edge is to be deleted, its dual induces a cycle in the dual tree, $C(e^*)$. Any edge on this cycle would reconnect the components and can be added to the primal tree as a replacement edge. For simplicity, we can just choose the edge after e^* on that cycle.

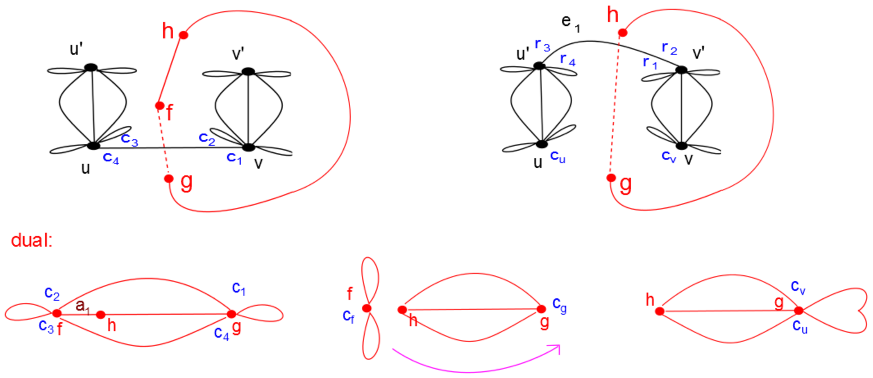


Figure 32: We delete the tree-edge $e = (u, v)$ from the graph. The non-tree edge (u', v') is added to the spanning tree, and the faces on either side of e are joined.

Let $e = (u, v)$ be the edge we want to delete. Let f and g be the faces incident to e. First, expose u and v in the primal tree. The top of the primal top-tree is now of the form leaf-path-leaf, where the path-cluster consists of one edge, (u, v). Since $f \neq g$, exposing corners c_1, c_2 incident to e (and to v) in the dual tree returns a top-tree with a path-cluster as root. Let $a_1 = (f, h) = e_1^*$ be the first edge on the cluster path from g to f. We want to replace e with $e_1 = (u', v')$. (See figure 32.)

Let c_h and c_h' be the corners of a_1 incident to h. Expose the vertices u, v, u', v' and the corners c_h and c_h' in the primal top-tree.

Expose the faces f and g and the corners $c_1 \ldots c_4$ in the dual top-tree. In the dual tree, delete the edge $a_1 = (f, h)$ from the tree. Cut f in c_2, c_3 and g in c_1, c_4, delete the leaves corresponding to c_2ec_3 and c_4ec_1, and name the replacing corners c_f and c_g (updating the eET). Then, join the faces f and g in the corners c_f and c_g, and name the newly constructed corners c_u and c_v. The dual top-tree now has the path cluster with boundary vertices h and g as root.

In the primary tree, delete the edge (u, v). The newly formed corners are exactly the c_u and c_v formed in the dual top-tree. Link the vertices u' and v', and update the eET correspondingly. That is, if r_1, r_2, r_3, r_4 are corners incident to $e_1 = (u', v')$, then exchange the segments $r_2e_1r_1$ and $r_4e_1r_3$ to $r_2e_1r_3$ and $r_4e_1r_1$.

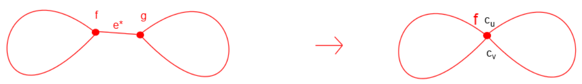


Figure 33: A non-tree edge is exposed in the dual top-tree, and removed.

Deleting a non-tree edge is the opposite of inserting an edge inside a face (see figure 30). If a non-tree edge $e = (u, v)$ is deleted, its incident faces f and g must be joined. In the primal tree, expose u, v and the corners $c_1, \ldots, c_4$ incident to e. In the dual tree, expose the edge $e^* = (f, g)$, which must belong to the co-tree, and all four corners incident to e. Cut g in the corners c_1, c_4, cut f in the corners c_2, c_3. Delete the cluster containing e^*, and join f and g in the corners c_f and c_g which were created from the cut. The join produces new corners c_u and c_v. In the primal tree, the leaves with eET c_2ec_1 and c_4ec_3 are deleted, and the delimiting corners for the cluster path are updated to be the newly formed c_u and c_v. During this procedure, the eET is updated to contain c_v in place of c_2ec_1 and c_u in place of c_4ec_3.

4.6 Finding a common face

Lemma 4.7. *If $e_0 \ldots e_l$ is a primal path from u to v, then for any e_i on the path, any face containing both u and v will be contained in $C(e_i^*)$, the cycle in the dual graph induced by the edge e_i^*.*

Proof. The cycle $C(e_i^*)$ induces a cut of the graph, which separates u and v.

Let f be a face which is incident to both u and v. Then there will be two paths $u \rightsquigarrow v$ on the face f, choose one of them, p. Since $C(e_i^*)$ induces a cut of the graph, which separates u and v, some edge $e_C \in p$ must belong to the cut. But then f is a dual endpoint of e_C^*, and thus, f belongs to $C(e_i^*)$. $\square$

Corollary 4.8. *If $e_0 \ldots e_l$ is a primal path from u to v, then any face containing both u and v will be contained in the intersection of $C(e_0^*)$ with $C(e_l^*)$.*

Since that intersection is necessarily a path, we can expose its endpoints in the dual top-tree, and if the minimal un-exposition counts are 0, the maintained list will automatically be the reply to the query.

To expose the endpoints of the intersection path, we need to find them.

Definition Given three vertices x, y, z on a tree, define $\operatorname{meet}(x, y, z)$ as $nca_x(y, z)$, the nearest common ancestor of y and z in the tree rooted in x.

Note 4.9. *For any triple of vertices (x, y, z) on a tree, for any permutation σ of the triple, $\operatorname{meet}(x, y, z) = \operatorname{meet}(\sigma(x, y, z))$. That is, $\operatorname{meet}(x, y, z) = \operatorname{meet}(x, z, y) = \operatorname{meet}(y, x, z) = \ldots$ and so on.*

Lemma 4.10. *Given a tree, T, the intersection I of two paths in the tree $a \rightsquigarrow b$ and $p \rightsquigarrow q$, falls in two cases:*

- *The intersection is empty if and only if* $\operatorname{meet}(a,p,q) = \operatorname{meet}(b,p,q) \neq \operatorname{meet}(a,b,p) = \operatorname{meet}(a,b,q)$,
- *The intersection is non-empty if and only if it is a path if and only if either:*
  - $\operatorname{meet}(a,p,q) = \operatorname{meet}(a,b,q)$ *and* $\operatorname{meet}(a,b,p) = \operatorname{meet}(b,p,q)$, *and then I is the path between* $\operatorname{meet}(a,p,q)$ *and* $\operatorname{meet}(a,b,p)$.
  - *Or,* $\operatorname{meet}(a,p,q) = \operatorname{meet}(a,b,p)$ *and* $\operatorname{meet}(a,b,q) = \operatorname{meet}(b,p,q)$, *and then I is the path between* $\operatorname{meet}(a,p,q)$ *and* $\operatorname{meet}(a,b,q)$.

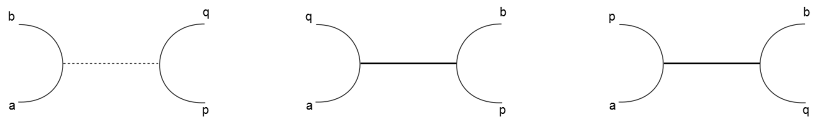


Figure 34: The three different scenarios.

Proof. Clearly, the intersection is either empty or not-empty, so if the bi-implications hold, these are all of the cases.

If the intersection is empty, then one can contract each path to a point, say, $a \rightsquigarrow b$ to a'. Then, $\operatorname{meet}(a,p,q)$ and $\operatorname{meet}(b,p,q)$ in the original graph both equal $\operatorname{meet}(a',p,q)$ in the contracted graph. Similarly, one could have contracted $p \rightsquigarrow q$. Furthermore, since the intersection is empty, $p' \neq a'$, and thus, $\operatorname{meet}(b,p,q) \neq \operatorname{meet}(a,b,p)$.

To see the other direction, assume $\operatorname{meet}(a,p,q) \neq \operatorname{meet}(a,b,p)$. Then, $p^\dagger = \operatorname{meet}(a',p,q) \neq a^\dagger = \operatorname{meet}(p',a,b)$. But then, if we contract both paths, there will exist a path $p^\dagger \rightsquigarrow a^\dagger$ between them, and thus, since the original graph was acyclic, they must contract to two different points, implying $I = \emptyset$.

Assume $I \neq \emptyset$. Root the tree in a. Now I is the intersection with the root-path of b with the path $p \rightsquigarrow q$. Let $u \in I$ denote the closest point to a on the path to b, and let $v \in I$ denote the furthest. Then, the entire path $u \rightsquigarrow v$ must belong to I, since the graph is acyclic, and thus, $u \rightsquigarrow v = I$. Since $u \rightsquigarrow v$ is also a contained in the path $q \rightsquigarrow p$, assume, without loss of generality, that u is closer to q than v is. Then, since $u \in I$, u is a common ancestor of (q,b). But since u must be the point closest to q, on $p \rightsquigarrow q$, u must also be the nearest common ancestor for (q,b). Similarly, u is also the nearest common ancestor for (q,p). That is, $u = \operatorname{meet}(a,q,b) = \operatorname{meet}(a,q,p)$. Turning the tree on its head and rooting it in b we obtain $v = \operatorname{meet}(a,b,p) = \operatorname{meet}(b,p,q)$.

If in stead u had been closer to p than to q, then we would achieve the second sub case.

To see the other direction, assume $v = \operatorname{meet}(a,b,p) = \operatorname{meet}(q,b,p)$, and $u = \operatorname{meet}(a,b,q) = \operatorname{meet}(a,p,q)$. Again, root the tree in a. Then because v is an ancestor of b and so is u, then the entire path $u \rightsquigarrow v$ belongs to the rootpath of b, that is $u \rightsquigarrow v \subseteq a \rightsquigarrow b$. By a similar argument, rooting the tree in q, one obtains $u \rightsquigarrow v \subseteq q \rightsquigarrow p$. And thus, $u \rightsquigarrow v \subseteq I$. We can now subdivide the path $a \rightsquigarrow b$ into three: $a \rightsquigarrow u \rightsquigarrow v \rightsquigarrow b$, and similarly, $q \rightsquigarrow u \rightsquigarrow v \rightsquigarrow p$. We need to realise that the "snippet" subpaths $a \rightsquigarrow u$, etc., do not belong to I. If again we root the tree in a, then because u is the nearest common ancestor of q, p, no part of the path $a \to u$ belongs to the path $q \rightsquigarrow p$. Similarly, $u \rightsquigarrow v$ intersects trivially with $q \rightsquigarrow u$, with $v \rightsquigarrow b$ and with $v \rightsquigarrow p$. And then, we must have equality $I = u \rightsquigarrow v$.

If we swap the names for p and q, the above is an argument for the second sub case. □

Corollary 4.11. *Given a spanning tree of a (multi) graph and given two edges e and e', with endpoints a, b and p, q respectively, then if we know*

$$\operatorname{meet}(a,b,p),\ \operatorname{meet}(a,b,q),\ \operatorname{meet}(a,p,q),\ \text{and}\ \operatorname{meet}(b,p,q),$$

we can determine if the intersection between the two cycles $C(e)$, $C(e')$ induced by the edges e and e', is empty, or, we can determine its endpoints.

Proof. If e and e' share both endpoints, the intersection equals $C(e) = C(e')$. Otherwise, we have three or four endpoints, and we calculate meet as in the theorem above to determine if the intersection is empty, or, which are the endpoints of its interval. □

But then, given a path $e_0 \dots e_l$ from u to v, to locate the intersection of $C(e_0^*)$ with $C(e_l^*)$, we simply need to calculate the four meet operations on the endpoint faces of e_0^* and e_l^*.

Lemma 4.12. *Given a top-tree over a tree T, with vertices $a, b, c \in T$, we can find $\operatorname{meet}(a,b,c)$ in logarithmic time.*

Proof. Split all clusters containing a, b, or c as a non-boundary vertex. There are only $\mathcal{O}(\lg(n))$ of those. Now in the top of the top-tree, we have a tree with $\mathcal{O}(\lg(n))$ vertices. Use this tree to find $\operatorname{meet}(a,b,c)$ in linear time. □

The algorithm is now obvious.

Algorithm 4.13. If we want to find a common face for u and v, expose u and v in the primal tree, using the dual tree to keep track of un-exposition counts. Now either u and v belong to different trees in the spanning forest and can just be connected, or they belong to the same component, and a unique path $e_0 \dots e_l$ exists. Using the dual top-tree, we can find the intersection of $C(e_0^*)$ with $C(e_l^*)$. If this intersection is trivial, we return that no common face exists. If the intersection is the interval between f and f', we expose f and f' in the dual top-tree: If the minimum deactivation counts are 0, the maintained corner-list will be exactly the answer to our query.

Furthermore, the boundary faces may also be candidates. If their minimum deactivation count is > 0, we may discard them. If not, we do not yet know whether they have incident corners to both u and v. To find this, expose first

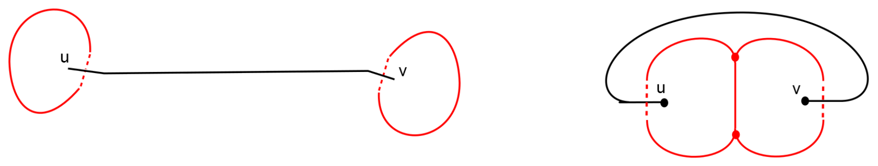


Figure 35: The two cases of query(u, v). The black lines are tree-edges on the exposed cluster path $u \rightsquigarrow v$, the red lines are edges in the dual graph, the dotted ones are non-tree edges corresponding to e_1^* and e_l^*. Left: The intersection of the cycles in the dual tree is trivial, and no common face may possibly exist. Right: The intersection forms a path between two meets (red dots).

u in the primal tree (this is done with one merge), and then, expose f and f' again in the dual tree. If the minimum deactivation count is 0 for both f and f', return, otherwise remember the corer-list of either of them, and likewise, $c'_{\min}$. Then, do the same for v. If f or f' (or both) has minimum deactivation count 0 in both cases, then it is a common face, and its corresponding corner lists, those we remembered, consist of corners incident to u and v respectively.

Theorem 4.14. *The algorithm above is correct and runs in time $\mathcal{O}((\lg(n))^2)$.*

Proof. Correctness follows from Lemma 4.12, Lemma 4.11, Lemma 4.6.

The running time comes from the time it takes to expose in the primary tree. To expose in the primary tree, we do $\mathcal{O}(\lg(n))$ splits and merges. For each of these, we do at most two exposes in the dual tree. An expose in the dual tree requires $\mathcal{O}(\lg(n))$ splits and merges, each of which take constant time. So each split or merge in the primary tree takes logarithmic time. In total, this gives a time of $\mathcal{O}((\lg(n))^2)$ for the primary top-tree expose. All other steps of the algorithm require only $o((\lg(n))^2)$ time. □

5 Extending to flips

We allow for the user to update the planar embedding of the graph by the two flip-operations. A subgraph may be flipped if it is not too well-connected to the rest of the graph. If the graph has an articulation point, v, then v has two corners incident to the same face, and we may flip the subgraph between those corners. If the graph has a separation pair, then there are two faces, which are both incident to both vertices, and we may flip the subgraph delimited by those faces and the separation pair. Internally, both flip operations are done by cutting out a subgraph, altering its orientation, and joining the subgraph back in.

If two (2-connected) vertices, v_1 and v_2, are both incident to two faces, f_1 and f_2, (that is, v_1, v_2 are a separation pair,) then a procedure for altering the embedding is the flip-operation, which takes the subgraph bounded by v_1, f_1, v_2, f_2, and turns it upside down in the embedding of the graph. For the embedding, this means the enumeration of edges incident to $v_i, i = 1, 2$ is changed: If all incident edges are enumerated $1, \ldots, l_i$, then for some $1 < k_i < p_i < l_i$, the enumeration will be $1, \ldots k_i - 1, p_i, \ldots, k_i, p_i + 1 \ldots l_i$ or $l_i, \ldots p_i + 1, k_i, \ldots, p_i, k_i - 1, \ldots, 1$. The two enumerations cover both scenarios: edge no. 1 is outside the flippable subgraph, and edge no. 1 is one of the edges in the graph to be flipped.

In the dual graph, the flip corresponds to two splits $f_i \to (a_i, a_i')$, two cuts $a_i \nleftrightarrow a_i'$, two links $a_1 \rightsquigarrow a_2'$ and $a_2 \rightsquigarrow a_1'$, and two merges $(a_1, a_2') \to f_1$ and $(a_2, a_1') \to f_2$.

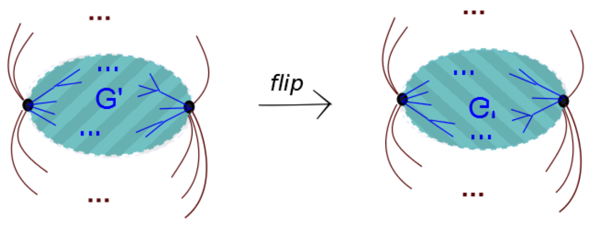


Figure 36: The flip-operation

Similarly, if one vertex has two incident corners of the same face, c_1 and c_2, we may flip in that one vertex such that the corresponding subgraph is flipped in the embedding.

To facilitate flip, we allow the following updates: Seclude and include. Seclude takes one graph and four corners and returns two graphs. In Figure 36, the returned graphs would be $G \setminus G'$ and G'. Include does the opposite. Given two graphs, an outer graph G^o with a designated face and two vertices u and v on the face, and an inner graph G^i with a designated face (corresponding to the outer face) and two vertices on that face, u' and v', we glue the inner graph on the designated face of the outer graph.

That is, the flip consists of three parts: Seclude, alter, include. By "alter" is simply meant that the orientation of an entire graph is reversed; the graph is flipped. As a tool for this operation, we need to be able to perform vertex cuts and vertex joins.

First, the articulation flip will be described. To flip in an articulation point, we only need to be able to cut a vertex, alter the orientation of a graph, and join two vertices to one. (See figure 37.)

5.1 Vertex cuts

Given a vertex v and two corners c_1, c_2 incident to v, we may cut the vertex in two such that all edges between c_1 and c_2 belong to the new vertex v', and all edges between c_2 and c_1 belong to v. (See Figure 37.) The corners c_1 and c_2 are deleted, and new corners c and c' are created, and the extended Euler Tours are updated correspondingly. (See figure 38.)

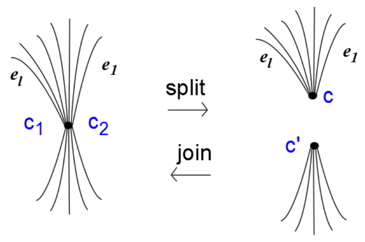


Figure 37: Left to right: The vertex is split in two specified corners, and two new vertices are formed. Right to left: Two vertices are joined to one.

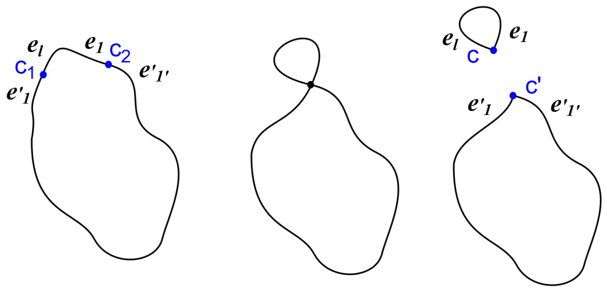


Figure 38: The extended Euler Tour is cut in c_1 and c_2, and glued back together to form two tours. New corners, c and c' are formed.

The eET cycle is split into two cycles; let e_1 and e_l denote the successor of c_2 and the predecessor of c_1, respectively, and let e'_1 and $e'_{l'}$ denote the successor of c_1 and the predecessor of c_2, respectively. The corner c will take place between e_l and e_1 in the eET containing v, and the corner c' will take place between $e'_{l'}$ and e'_1 in the eET containing v'. (See figure 38.)

In the underlying ternary representation of the tree, we have already explained how to cut the edges that correspond to c_1 and c_2. See figure 21. Unlike expose(c_1, c_2), we do not make a new edge connecting v to v', we leave them to be disjoint. To see the description of what happens in the top-tree, see section 4.2.

5.1.1 Primal and dual vertex cuts

We need to be able to cut both primal and dual vertices. For convention, the extended Euler Tour is updated when primal vertices are cut, and never when dual vertices are cut.

5.2 Altering the orientation

In order to flip, we need to alter the orientation of a graph. In the eET, the predecessor becomes a successor and vice versa.

To implement this, we let the clusters of the primal and dual top-tree contain one more piece of information, namely the orientation of the cluster, which is plus $+$, or minus $-$. When a cluster is split, its sign is multiplied with the signs of its children. That is, if a cluster with a minus is split, the minus is propagated down to the children. When clusters are merged, the new union-cluster is simply equipped with a plus.

When a cluster is "negative", some of its information changes character:

- its left-hand child becomes a right-hand child and vice versa,
- the begin- and end- vertices swap interpretation, as do begin- and end-corners,
- if it is the path-cluster from u to v, it should be interpreted as the path-cluster from v to u, that is, north and south also swap,
- and, finally, in the extended Euler Tour, predecessor is interpreted as successor and vice versa.

In the 3-step program of "seclude, alter, include," the alter-step simply consists of changing the sign of the top cluster of the dual top-tree.

5.3 Vertex joins

Given two vertices, and given a corner incident to either vertex, we may join the two vertices, as an inverse operation to the vertex cut.

If c is a corner incident to the vertex v and c' is incident to v', we may perform the operation $\text{join}(c, c')$.

We will assume, we have exposed the vertex-corner pairs, v, c and v', c' in either top-tree.

Now, the extended Euler Tour is updated correspondingly. We must create new corners, c_1 and c_2. In the euler tour, c_1 must point to the predecessor of c as its predecessor, and the successor of c' as its successor, and similarly for c_2. And vice versa, the object of those edges must point to c_1 and c_2. But, since either exposed top-cluster may have a negative sign, the predecessor or successor of c or c' *may* be interpreted the opposite way. That is, we may have the case where a pointer to c_2 is placed as the predecessor-pointer for some edge e'_1, although e'_1 is also the predecessor of c_2 – simply because the negative sign means they swap interpretation.

5.4 Articulation flips

We now have the necessary tools for flipping in an articulation point, that is, if a vertex has two corners incident to the same face, we may flip the subgraph spanned by those corners. This is done by the three-step scheme of cut, alter, join.

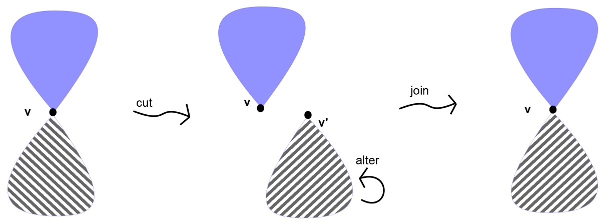


Figure 39: To flip in an articulation point: cut, alter, join.

Given a vertex v and two corners c_1 and c_2 incident both to v and some common face f, we first perform the vertex cut, $\text{cut}(v, c_1, c_2)$ in the primal top-tree. Then, we perform the operation of $\text{cut}(f, c_1, c_2)$ in the dual top-tree (See figure 40). Together, these two operations return two graphs, G and G', with respectively v, c and v', c' exposed in their primal top-trees, and with f, c and f', c' exposed in their dual top-trees. Then, we alter the sign of the top cluster of the cut-out subgraph G'. Finally, we perform the operation $\text{join}(c, c')$, first in the dual, and then in the primal top-tree, gluing G' back to G - but since the sign of G' was altered, this means we have flipped G'.

This operation takes $\mathcal{O}(\lg(n)^2)$ time, dominated by the expose-operation in the primal top-tree.

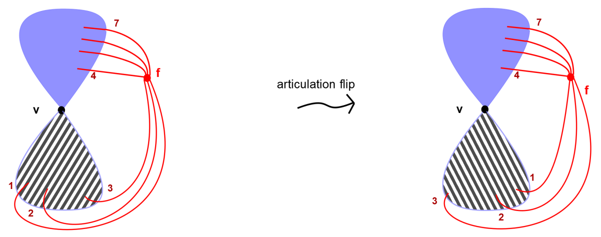


Figure 40: In order to flip the component, the orientation must be flipped, not only in the primal tree, but also in the dual tree. The figure shows, how the co-tree edges from f to the flippable component change their order, when the component is flipped.

5.5 Seclude

Given four corners $c_1, \ldots, c_4$ such that c_1, c_2 are incident to the vertex v, c_3, c_4 are incident to the vertex u, c_2, c_3 are incident to the face f and c_4, c_1 are incident to the face g. See figure 41.

To $\text{seclude}(c_1, c_2, c_3, c_4)$, cut the vertex v through the corners c_1 and c_2, cut u through c_3 and c_4. In the dual top-tree, cut the face f in c_2 and c_3 (obtain

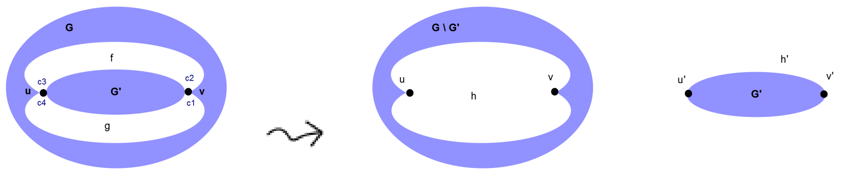


Figure 41: The subgraph G' is delimited by two vertices and two faces. The picture shows how G' is cut out of the graph G.

f and f'), and cut the face g in c_4 and c_1 (which returns g and g'). Then, join f' with g', and f with g in the dual top-tree. We now have two graph-stumps, denote them G and G', where G contains u and v which are both incident to one face h which was formed by joining f with g, and where G' contains u' and v', both incident to the new face h'.

This procedure consists of two vertex cuts in the primal tree, time $\mathcal{O}(\lg(n)^2)$, two vertex cuts in the dual tree, time $\mathcal{O}(\lg(n))$

5.6 Include

Include is the inverse operation of seclude above.

Given two graphs, G and G', and given two vertices $u, v \in G$ and two vertices $u', v' \in G'$, and given a designated face $h \in G$ incident to both u and v, and similarly a face $h' \in G'$ incident to u' and v'. Let four corners be given, such that $c_{u,h}$ is incident to u and h, and so on, $c_{v,h}, c_{u',h'}, c_{v',h'}$.

To perform `include`$(c_{u,h}, c_{v,h}, c_{u',h'}, c_{v',h'})$: Cut the face h through the corners $c_{v,h}$ and $c_{v,h}$, and cut the face h' similarly. Let the resulting nodes be denoted f, g, and f', g', respectively. Now, join the faces f with f' and g with g'. Finally, join the vertices u with u' and v with v'.

5.7 Flip

Similarly to flipping in an articulation point, we now show how to flip in a separation pair.

Given two vertices which are incident to two faces, and given four delimiting corners $c_1 \dots c_4$ pairwise incident to both vertices and both faces (as in Section 5.5), we perform flip(c_1, c_2, c_3, c_4) with a three-step procedure: Seclude, alter, include.

First, seclude the subgraph delimited by the corners. Then, alter its orientation by changing its sign. Finally, perform include, such that f joins with g', and g with f'.

This takes time $\mathcal{O}(\lg(n)^2)$, dominated by the seclude and include operations.

Theorem 5.1. *We can support flips in a separation pair in time* $\mathcal{O}(\lg(n)^2)$.

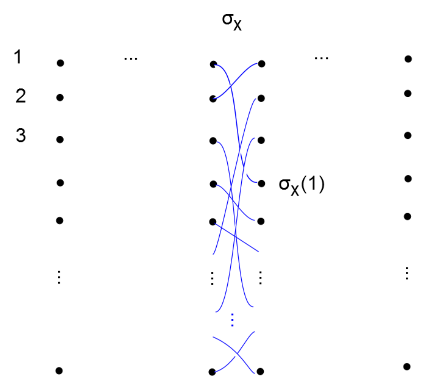


Figure 42: The update assigning a new value to σ_x corresponds to p edge deletions and p edge insertions in the dynamic forest.

6 Logarithmic lower bound for planarity

So far, the best lower bound for dynamic planarity testing was $\Omega(\lg(n)/\lg\lg(n))$ by Henzinger et al. [13], which is shown by reduction to the parity problem. Pătraşcu later developed a technique for showing $\Omega(\lg(n))$ lower bounds, e.g. the bound of $\Omega(\lg(n))$ for the maximal of update- and query time in fully dynamic graph- or forest connectivity. He does this by proving that same lower bound for the partial-sums problem, and then making a reduction. In fact, he shows a stronger result: Let t_q be the query time and t_u the update time, then $t_q \lg(t_u/t_q) \in \Omega(\lg(n))$ and $t_u \lg(t_q/t_u) \in \Omega(\lg(n))$.

In Pătraşcu's article, the dynamic p-permutation prefix problem is considered. We have p permutations of p elements, $\sigma_1 \dots \sigma_p$. Initially, these are all the identity. An update takes an index $j \in [p]$ and a permutation $\sigma \in S_p$ and assigns $\sigma_j := \sigma$. Given some j, $1 \leq j \leq p$, we can define the total composed permutation up to j: $\Sigma_j = \sigma_j \circ \dots \circ \sigma_1$. A query takes an index j and a permutation $\Sigma \in S_p$, and asks if $\Sigma = \Sigma_j$.

Lemma 6.1 ([30])**.** *For the p-permutation prefix problem (defined above), the average time per operation is $\Omega(p\lg(p))$.*

This is not stated explicitly as a lemma, but is implicitly used in [30, Section 7.3] and described in [30, Section 6].

To reduce to connectivity from the p-permutation prefix problem, Pătraşcu considers a set of $p \cdot (p+1)$ vertices, organised in a $p \times (p+1)$ rectangle. (See Figure 42.) Between consecutive columns, a permutation is represented as a perfect matching. All together, we have $\sqrt{(n)}$ paths of length $\sqrt{(n)}$, going from the first to the last column. A vertex $v_{i,j}$ has the position (i, j) in the rectangle. An update of a permutation corresponds to a bulk-update of p edge deletions and p edge insertions. A prefix query corresponds to verifying for some Σ and some j, $1 \leq j \leq p$, that $\Sigma = \Sigma_j$, this means for all i that $v_{0,\Sigma(i)}$ belongs to the same connected component as the vertex $v_{j,i}$. Thus, a prefix query corresponds to p connectivity queries.

Since there are p updates or queries per bulk-operation, lower bound for each of them becomes $\Omega(p \lg(p)/p) = \Omega(\lg(p))$. We want to express this in terms of the number of vertices in the graph, there are $n = p \cdot (p+1)$ vertices, that is, $p = \Theta(\sqrt{n})$, and thus the lower bound is $\Omega(\lg(\sqrt{n})) = \Omega(\lg(n))$.

6.1 Reduction to dynamic planar embedding

Consider the problem of maintaining an embedding of a dynamic planar graph. The operations are delete, insert, and query, which take time t_d, t_i, t_q, respectively.

Theorem 6.2. *For maintaining an embedding of a dynamic planar graph with n vertices, there is a lower bound* $\max\{t_i, t_d, t_d\} = \Omega(\lg(n))$, *which holds regardless of amortisation and randomisation.*

My idea is to use an approach which is similar to Pătraşcu's reduction from the p-permutation prefix problem to dynamic connectivity. As with connectivity, permutations correspond to perfect matchings, but in stead of a forest we use a connected embedded graph.

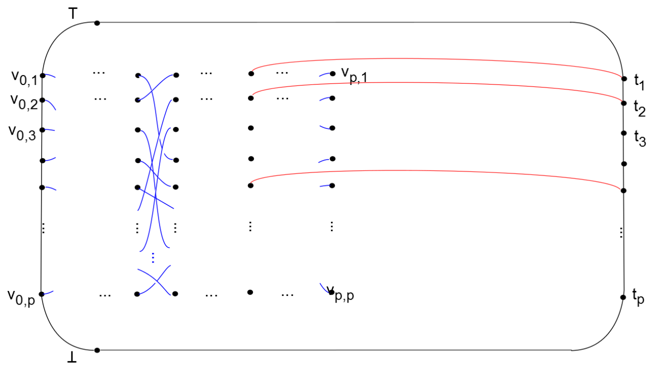


Figure 43: $p \cdot (p+1)$ vertices are connected by perfect matchings of columns, and lie inside a face. Updates exchange one perfect matching with another. Upon a query, the red edges are inserted, until either there is a pair where the red edge cannot be inserted (returns false), or all t_i have been matched (returns true), and then all the red edges are removed again before the game goes on.

Consider the following construction. (See Figure 43.) As before, $p \cdot (p+1)$ vertices are arranged in a rectangle, and we name them $v_{i,j}$ according to their position. We create vertices $\top$ and $\bot$, and vertices $t_1 \ldots t_p$. The path $\top, t_1, \ldots, t_p, \bot$ is linked, as is the path $\top, v_{0,1}, \ldots, v_{0,p}, \bot$. Thus, a cycle is formed, and the plane is divided in two faces. We refer to this cycle as *the frame*.

For maintaining an embedded graph, the update instructions correspond to those in our data structure. Edges may be deleted, or inserted in specified corners. The query operation replies whether a common face exists, and returns the appropriate face and corners. As with connectivity, all permutations correspond to a perfect matching of the vertices in two columns. Only in the case of σ_1 we have to be careful, and insert all the the edges in the corner incident to the same face (e.g. the internal face in Figure 43). Again, updates correspond to interchanging a permutation.

The query corresponding to verifying the permutation Σ as the j'th prefix permutation Σ_j now corresponds to linking $v_{j,i}$ to $t_{\Sigma^{-1}(i)}$, for all i. The corresponding targets $t_1 \ldots t_p$ are on one side of the frame (the right side in Figure 43). That is, for $i = 1 \ldots p$, query $(v_{j,i}, t_{\Sigma^{-1}(i)})$. If the returned list is empty, return "false" and delete all the inserted edges (the red edges between the j'th column and the t's). If the returned list is non-empty, it contains exactly one face and up to two corners incident to $v_{i,j}$. Insert the edge in either of those corners and proceed. If the edge insertions were all possible, return "true", and delete all the new (red) edges again.

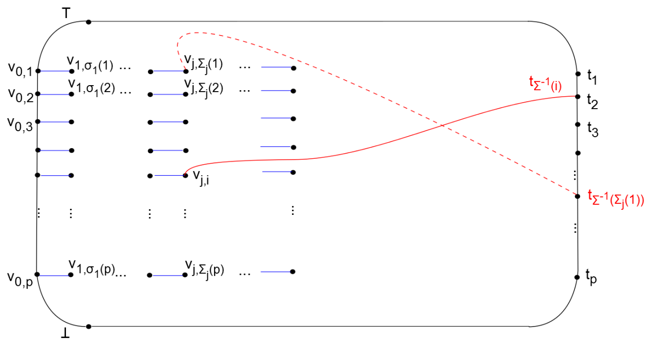


Figure 44: When we maintain a planar embedding of the graph, the paths look like this, $v_{0,i}, v_{1,\Sigma_1(i),\ldots,\Sigma_p(i)}$. When we make a query, we start by inserting the edge between $v_{j,1}$ and $t_{\Sigma^{-1}(1)}$, and proceed for other values $i > 1$, until all are matched or an insertion is impossible. The red lines depict a query which returns "false".

Clearly, if $\Sigma = \Sigma_p$, the bulk-query returns "true". And vice versa, if $\Sigma \neq \Sigma_p$, then there exists a pair i, i' such that $\Sigma(i) < \Sigma(i')$, but $\Sigma_p(i) > \Sigma_p(i')$. But then, inserting the edge $(v_{\Sigma(i),j}, t_i)$ separates $v_{\Sigma(i'),j}$ from $t_{i'}$, which means the bulk-query will return "false" as desired. (See Figure 44.)

All in all, the update corresponds to p deletions and p insertions, and the query corresponds to at most p insertions, p queries, and p deletions. From Lemma 6.1, we obtain a lower bound of $\Omega(\lg(n))$ for the maximum of delete-, insert-, and query time, in a data structure which maintains a planar embedding as described.

6.2 Reduction to planarity testing

Planarity testing of a dynamic graph supports the operations delete, insert, and query. Delete is as before, insert simply inserts an edge (u, v), no corners or face is specified. The query operation returns a boolean for whether an edge may be inserted without violating the planarity of the graph. (That is, if any embedding of the graph exists where the edge would fit.) The operations delete, insert, and query take time t_d, t_i, t_q, respectively.

Theorem 6.3. *For maintaining an embedding of a dynamic planar graph with* n *vertices, there is a lower bound* $\max\{t_i, t_d, t_d\} = \Omega(\lg(n))$*, which holds regardless of amortisation and randomisation.*

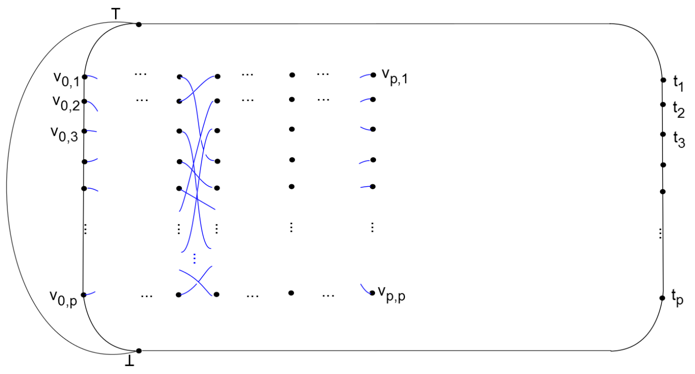


Figure 45: An edge from $\top$ to $\bot$ is added to the construction. Once an edge from some $v_{i,j}$ to some t_k is inserted, we have a subgraph which is a subdivision of a 3-connected graph, which means the embedding is unique up to reflection.

Proof. The same bound holds for general planarity testing of graphs, and almost the same construction can be used: Add the edge $(\top, \bot)$. (See Figure 45). Now in this setting, one doesn't need to specify in which corner edges are inserted. But upon a bulk-query, as soon as the first edge is inserted $(v_{1,j}, t_{\Sigma(1)})$, we have a subgraph which is a subdivision of a three-connected graph, which means its embedding is unique up to reflection. But then the argument from before for why this returns true iff $\Sigma = \Sigma_p$ still holds. From this it follows that $\Omega(\lg(n))$ is also a lower bound for the slowest operation for planarity testing of a graph with n vertices. □

7 Conclusion and future work

We have described a data structure for planarity testing of dynamic planar embedded graphs, which supports updates and queries in polylogarithmic time,

$\mathcal{O}(\lg(n)^2)$, and which allows for a flip operation which alters the embedding, also in polylogarithmic time, $\mathcal{O}(\lg(n)^2)$. The data structure allows for the user to cut vertices through two given corners, and reversely, to join vertices.

The algorithm uses top-trees and relies on an elegant observation about the Euler Tours of the primal and that of the dual tree in the tree-co-tree decomposition of the planar graph.

The lower bound for the dynamic planarity problem has been raised from $\Omega(\lg(n)/\lg\lg(n))$ to $\Omega(\lg(n))$ per operation using the set-up defined by Pătraşcu.

7.1 Adding functionalities to the data structure

A natural functionality to enrich our data structure with, are ones to facilitate queries for the classical properties of bi-connectivity or 2-edge connectivity.

The data structure can easily be modified to support a minimum spanning forest as the primal forest. The only differences are upon insertion and deletion of edges. When an edge is deleted, we should not choose an arbitrary replacement edge, but one of minimum weight. The dual top-tree can maintain the minimum weight edge of any cluster path (using the techniques described in [18]), which means that when we expose the two faces adjacent to the edge to be deleted, we know exactly which replacement edge to use, as it is on the exposed path in the dual tree. The join in the dual tree may then be of one path with an other, and not only of a path with a leaf like now, but this is well-supported by the vertex join operation in the top-tree data structure. Upon edge insertion, an edge may have to leave the primal spanning tree and enter the co-tree, but this is also well-supported by the data structure.

Since our data structure is relevant in circuit design and graph drawing, maintaining dynamic minimum cycle basis would be relevant. Given a graph, a set of cycles, $\mathcal{C}$ forms a *cycle basis*, if any cycle in the graph is the `xor` of some $C_1 \dots C_l \in \mathcal{C}$, and no $C \in \mathcal{C}$ is the `xor` of cycles in $\mathcal{C} \setminus C$. The number of cycles in a static graph is a constant ($m - n + c$ for a graph with m edges, n vertices, and c components). If the graph is weighted, a minimum cycle basis is one of minimum weight. If weights may be both positive and negative, the static problem is NP complete, but if weights are positive, polynomial time solutions exist. Minimum cycle basis for a static planar graph is studied by Borradaile, Sankowski, and Wulff-Nilsen in [3]. Maintaining a minimum cycle basis of a dynamic planar graph is equivalent to maintaining the Gomory-Hu tree for the dual graph. And thus, our dynamic tree-co-tree decomposition may be of use.

An other interesting question is: Given two vertices which do not share a common face, is it possible to connect them after having altered the embedding with only one flip? And which flip would that be? How many flips would be needed? Is it possible to connect them after only one edge deletion? Which edge?

Finally, it would be very interesting to work on general planarity testing of dynamic graphs, and a hope is that this data structure can be used as a sub-routine in a more efficient data structure for planarity testing.

References

[1] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully-dynamic trees with top trees. *CoRR*, cs.DS/0310065, 2003.

[2] Stephen Alstrup, Jens Peter Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *IPL*, 64:64–4, 1997.

[3] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. Min st-cut oracle for planar graphs with near-linear preprocessing time. *CoRR*, abs/1003.1320, 2010.

[4] Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 436–441. IEEE, 1989.

[5] Giuseppe Di Battista and Roberto Tamassia. On-line graph algorithms with spqr-trees. In *Automata, Languages and Programming*, pages 598–611. Springer Berlin Heidelberg, 1990.

[6] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997.

[7] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 252–257, New York, NY, USA, 1983. ACM.

[8] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, April 1997.

[9] Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *J. ACM*, 46:28–91, 1999.

[10] M. R. Henzinger. Fully dynamic biconnectivity in graphs, 1992.

[11] Monika R. Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs, 1997.

[12] Monika Rauch Henzinger. Improved data structures for fully dynamic biconnectivity. In *Proc. 26th ACM Symp. Theory of Computing*, pages 686–695, 1997.

[13] Monika Rauch Henzinger and Michael L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[14] Monika Rauch Henzinger and Valerie King. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation, 1997.

[15] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *JOURNAL OF THE ACM*, 46:516, 1999.

[16] Monika Rauch Henzinger and Han La Poutré. Certificates and fast algorithms for biconnectivity in fully dynamic graphs. In *Third Annual European Symposium on Algorithms (ESA'95*, pages 171–184, 1995.

[17] Monika Rauch Henzinger and Mikkel Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997.

[18] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic graph algorithms i: connectivity and minimum spanning tree. In *JOURNAL OF THE ACM*, pages 79–89. ACM Press, 1997.

[19] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.

[20] Giuseppe F. Italiano, Johannes A. La Poutré, and Monika H. Rauch. Fully dynamic planarity testing in planar embedded graphs, 1993.

[21] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1142, 2013.

[22] David R. Karger. Random sampling in cut, flow, and network design problems. In *Mathematics of Operations Research*, pages 648–657, 1994.

[23] Philip Klein and Shay Mozes. Optimization algorithms for planar graphs, 2014. Book draft. See http://planarity.org/.

[24] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 146–155, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[25] Casimir Kuratowski. Sur le problme des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930.

[26] Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 706–715, New York, NY, USA, 1994. ACM.

[27] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203 – 236, 1994.

[28] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*. North-Holland Mathematics Studies. Elsevier Science, 1988.

[29] Mihai Patrascu. Lower bound techniques for data structures, 2008.

[30] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. See also STOC'04, SODA'04.

[31] Mihai Pătraşcu and Mikkel Thorup. Planning for fast connectivity updates. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 263–271, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983.

[33] Mikkel Thorup. Decremental dynamic connectivity. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 305–313, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[34] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM.

[35] Mikkel Thorup. Fully-dynamic min-cut. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 224–230, New York, NY, USA, 2001. ACM.

[36] W. T. Tutte. How to Draw a Graph. *Proceedings of the London Mathematical Society*, s3-13(1):743–767, 1963.

[37] K. Wagner. ber eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114(1):570–590, 1937.

[38] Jeffery Westbrook. Fast incremental planarity testing. In W. Kuich, editor, *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 342–353. Springer Berlin Heidelberg, 1992.

[39] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *SODA*, pages 1757–1769, 2013.