Worst-Case Polylog Incremental SPQR-trees: Embeddings, Planarity, and Triconnectivity.*

Jacob Holm† Eva Rotenberg‡

Abstract

We show that every labelled planar graph G can be assigned a canonical embedding $\phi(G)$, such that for any planar G' that differs from G by the insertion or deletion of one edge, the number of local changes to the combinatorial embedding needed to get from $\phi(G)$ to $\phi(G')$ is $\mathcal{O}(\log n)$.

In contrast, there exist embedded graphs where $\Omega(n)$ changes are necessary to accommodate one inserted edge. We provide a matching lower bound of $\Omega(\log n)$ local changes, and although our upper bound is worst-case, our lower bound hold in the amortized case as well.

Our proof is based on BC trees and SPQR trees, and we develop *pre-split* variants of these for general graphs, based on a novel biased heavy-path decomposition, where the structural changes corresponding to edge insertions and deletions in the underlying graph consist of at most $\mathcal{O}(\log n)$ basic operations of a particularly simple form.

As a secondary result, we show how to maintain the pre-split trees under edge insertions in the underlying graph deterministically in worst case $\mathcal{O}(\log^3 n)$ time. Using this, we obtain deterministic data structures for incremental planarity testing, incremental planar embedding, and incremental triconnectivity, that each have worst case $\mathcal{O}(\log^3 n)$ update and query time, answering an open question by La Poutré and Westbrook from 1998.

1 Introduction

The motivation behind dynamic data structures such as dynamic graphs is that local changes, such as the insertion of an edge, should only have limited influence on the global properties of the graph, and thus, after a local change, one needs not process the entire graph again in order to have substantial information about properties of the graph. An example of a class of well-studied questions is that of connectivity and k-edge/vertex-connectivity: upon the insertion or deletion of an edge, a representation of the graph is quickly updated, such that later *queries* to whether an at query-time specified pair of vertices are connected or k-edge/vertex-connected, can be answered promptly.

For planarity, the most well-studied query is that of whether a given edge can be added without violating planarity, or without violating the planar embedding. Other natural queries include questions about whether a component of the graph is itself planar. In this paper, our main focus of study is not the question of *whether* the graph is planar, but *how* the graph is embedded in the plane. Here, and throughout the paper, we view embeddings from a combinatorial point of view: a graph is embedded if each edge, for each of its endpoints, can compute its right and left neighbors in the circular ordering. An embedding is planar if said circular ordering around the endpoints is realizable as a planar drawing of the graph (see [20]).

In this setting, *queries* to the neighbors around an endpoint of an edge are well-defined, and local changes obtain a new and interesting meaning: for each edge in the graph, we can say that it is *affected* by an update if its set of neighbors of either endpoint has changed. A *local change* to the graph *or its embedding* is any change such that only a constant number of edges are affected (where the constant depends only on the type of operation changing the graph). The local changes we will consider are: the insertion of an edge across a face, or the deletion of an edge, as well as the *flip* operations that keeps the graph fixed, but changes the embedding by taking a subgraph that is separated from the rest of the graph by either a separation pair (a *separation flip*), or an articulation point (an *articulation flip*), and either *reflecting* it, or *sliding* it to a new position in the edge order of the separating vertices (see Figure 1).

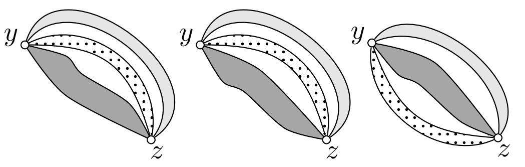
In other words, one can say that there are two types of local changes: The ones that keep the embedding fixed but changes the graph by deleting an edge or inserting an edge across a face – i.e. *updates to the graph* – and the ones that keep the graph fixed but change the embedding as in Figure 1 – i.e. *updating the embedding.* We show an interesting connection between these two types of local changes, namely a class of embeddings where only worst-case $\mathcal{O}(\log n)$ updates to the embedding are necessary to accommodate any graph update. This is asymptotically tight, matching an $\Omega(\log n)$ lower bound.

We give an analysis of changes to the embedding that can accommodate fully-dynamic changes to the graph, that is, both edge insertions and edge deletions. As long as the resulting graph can be embedded in the

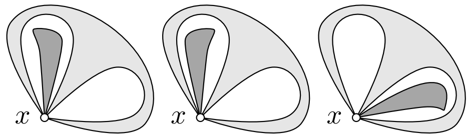
*Partially supported by the VILLUM Foundation grant 16582, “BARC”, and by Independent Research Fund Denmark grant 2018-2021 (8021-00249B), “AlgoGraph”.

†University of Copenhagen jaho@di.ku.dk

‡Technical University of Denmark erot@dtu.dk



(a) Separation flips: reflect and slide.



(b) Articulation flips: reflect and slide.

Figure 1: Local changes to the embedding of a graph. Separation flips of the reflect type are also known as Whitney flips [28].

plane, we will maintain an implicit representation of its embedding. We need only analyze the edge-insertion case. In terms of counting changes to the embedding, the deletion case is symmetric: to delete e, find the "canonical" embedding of $G - e$, add e, and record the necessary changes to the embedding; to get from G to $G - e$ perform their opposites. Thus, computation time aside, in terms of counting changes to the embedding, it is enough to study the incremental case.

This notion of an analysis that counts only the *changes of heart* is akin to the field of *online algorithms with recourse* introduced by Imase and Waxman [18] for the problem of Dynamic Steiner Trees, and since studied e.g. for Packing, Covering, Edge-Orientations, and Perfect Matchings [2, 4, 10].

As a welcome side effect, our results entail improved computation times for a series of related problems in the incremental setting. Specifically, since our bound for the changes to the embedding is worst-case, we obtain new *worst-case* polylog incremental algorithms for planarity and 3-connectivity, problems that were previously only efficiently solved in terms of amortized analysis. This is interesting on two accounts: One is from a practical point of view, where it is an asset to guarantee fast update-times, another is from a theoretical point of view, namely that this allows for unlimited undo.

In terms of technical contribution, our path towards understanding how embeddings are related, and edges are accommodated, goes via *BC-trees and SPQR-trees.* Intuitively, for every connected graph, its BC-tree describes all its 2-connected components, its articulation points, and how these are related. Similarly, the SPQR-tree describes the 3-connected components and their relations via separation pairs. It is well-known that the SPQR-trees and the BC-tree can be used to count the number of embeddings – here, we use them to dynamically maintain a specification of an embedding.

An edge insertion in the graph becomes a path contraction in the BC-tree, and several path contractions in SPQR-trees; one for each block on said BC-path. To obtain our $\mathcal{O}(\log n)$ bound, we need to ensure that these paths, though they may be long, only lead to limited changes to the embedding. They should be, so to speak, almost everywhere ready-to-contract. For solving this, our main idea is to maintain heavy path decompositions over the BC-tree and the SPQR-trees. Briefly speaking, we assign each edge to be either heavy or light with the property that the heavy edges form a forest of paths, and every pair of vertices are at most $\mathcal{O}(\log n)$ light edges apart. Once we have made sure that the heavy paths are ready-to-contract, we only need to accommodate the $\mathcal{O}(\log n)$ light edges, leading to the $\mathcal{O}(\log n)$ total changes to the embedding. Since a single edge insertion in a connected component can lead to contractions of paths in the BC-tree and in several SPQR-trees, we introduce a weighting of nodes that ensures that the total number of light edges that are affected is still bounded.

1.1 Our results.

In this paper, we present the following theorems:

Theorem 1.1. *For every planar graph G there exists a* canonical embedding *$\phi(G)$, such that for any edge e in G, the number of local changes between $\phi(G - e)$ and $\phi(G) - e$ is bounded by $\mathcal{O}(\log n)$.*

A matching lower bound is derived from the following construction:

Theorem 1.2. *There exists a family of planar graphs $\{G_h\}_{h\in\mathbb{N}}$ where G_h has $\mathcal{O}(2^h)$ vertices, and any embedding ϕ of G_h has an edge $e \notin G_h$ such that inserting e requires h flips to ϕ.*

Thus, by repeatedly inserting an edge and deleting it, we get a lower bound that matches Theorem 1.1:

Corollary 1.1. *For every $n \in \mathbb{N}$, we can initiate a dynamic graph with n vertices, and provide an adaptive sequence (of arbitrary length) of alternating edge insertions and edge deletions, where every edge insertion requires $\Omega(\log n)$ flips.*

We provide algorithmic results for incremental graphs on n vertices with worst-case guarantees:

Theorem 1.3. *There exists a deterministic data structure for incremental triconnectivity that handles edge insertions in $\mathcal{O}(\log^3 n)$ worst-case time, and answers queries in $\mathcal{O}(\log^3 n)$ worst-case time.*

Note that this (Theorem 1.3) holds for general graphs, not only planar graphs.

THEOREM 1.4. *There exists a deterministic data structure for incremental planarity that handles edge insertions in $\mathcal{O}(\log^3 n)$ worst-case time, and answers queries to whether an edge can be inserted, and to the neighbors of a given existing edge in the current embedding, in $\mathcal{O}(\log n)$ worst-case time.*

1.2 Previous work

Biconnectivity. In 1998 [23] La Poutré and Westbrook gave algorithms for maintaining 2-edge and 2-vertex connectivity (here called *biconnectivity*) in worst case $\mathcal{O}(\log n)$ time per operation, under edge insertions with *backtracking.* They used a number of the same techniques that we do, and stated as an open problem whether they could be extended to triconnectivity and planarity testing.

SPQR-trees. Understanding 3-vertex connectivity (here called *triconnectivity*) in a graph via a structure over its triconnected components, its separation pairs, and their relations, dates back to Saunders Mac Lane [21], who also in his work discussed the connections between embeddings and triconnectivity via said structure. This was the mathematical foundation of the algorithms by Hopcroft and Tarjan [16, 17] for calculating the triconnected components and, respectively, for determining whether a graph is planar, in linear time. Later, this structure was dubbed SPQR-tree by Di Battista and Tamassia [5, 6], who devised the first efficient incremental algorithm for planarity testing, handling updates to the graph in amortized $\mathcal{O}(\log n)$ time.

Planarity testing and triconnectivity. The amortized update time of $\mathcal{O}(\log n)$ by Di Battista and Tamassia was soon improved to expected $\mathcal{O}(k\alpha(k, n))$ total time for k operations by Westbrook [27], where α denotes the inverse of Ackermann's function. The optimal total time of deterministic $\mathcal{O}(k\alpha(k, n))$ for k operations is due to La Poutré [22]. Further α-time algorithms for incremental triconnectivity were given by Di Battista and Tamassia [7], specifically, their algorithm spends $\mathcal{O}(k\alpha(k, n))$ total time for k insertions and queries when the initial graph is biconnected, and $\mathcal{O}(n \log n + k)$ for general graphs.

For the fully-dynamic case, Galil et al.[9], give a data structure that handles any insertion or deletion of an edge in deterministic $\mathcal{O}(n^{2/3})$ time while maintaining whether the graph is planar, and facilitating queries to vertex triconnectivity in $\mathcal{O}(n^{2/3})$ time.

Biased search trees and dynamic trees Our new technique goes via maintaining a heavy path decomposition of the SPQR-tree, as our graph changes dynamically. The heavy path decomposition was invented by Sleator and Tarjan [25], as a tool for handling dynamic forests subject to links and cuts in worst case $\mathcal{O}(\log n)$ time per operation.

This data structure for dynamic trees in turn uses the biased search trees by Bent et al. [3] to represent each heavy path. In a biased search tree T, each node v has an associated weight $w(v)$, and the depth of v is $\mathcal{O}(1 + \log \frac{w(T)}{w(v)})$, where $w(T)$ is the sum of weights $\sum_{u \in T} w(u)$. This is in contrast to other balanced binary search trees, where the depth of a node is typically $\mathcal{O}(\log n)$.

Dynamic embeddings The question about maintaining a dynamic embedded graph can be asked in many ways. Tamassia [26] gives an algorithm for a fixed embedding of a graph that handles edge-insertion across a face and "undo" in amortized $\mathcal{O}(\log n)$ time per operation. Italiano et al. [19] give an algorithm that maintains an embedded graph subject to deletions of edges and insertions across a face in $\mathcal{O}(\log^2 n)$ time per update.

While the previous papers only allowed for edge insertions across a face, Eppstein [8] takes a different approach: The edge may be inserted in any way specified, possibly increasing the genus of the embedding, but as long as the genus g is low, the update time for minimum spanning tree, the fundamental group, and orientability of the surface is fast: $\mathcal{O}(\log n + \operatorname{poly}\log g)$. The data structure allows for changes in the embedding corresponding to the flips in Figure 1, and also allows for contraction of edges and splits of vertices.

The construction by Eppstein [8] uses a primal-dual decomposition that is updated dynamically. Combining this idea with ideas from Italiano et al. [19] and using top-trees [1], Holm and Rotenberg [14] give a data structure for maintaining a dynamic planar embedding that allows for edge-deletions, insertions across a face, and flips (Figure 1) in worst case $\mathcal{O}(\log^2 n)$ time. The data structure in [14] also facilitates queries to whether a pair of vertices would be linkable after only one flip operation, to which it responds with the specific flip operation in the affirmative case.

1.3 Overview of techniques

For the purpose of our algorithmic results, we cannot afford to maintain constructions such as the SPQR-tree explicitly; the insertion or deletion of just one edge can lead to $\Theta(n)$ changes, which is challenging to handle in worst-case time. We show, however, that we can maintain an implicit representation of the forest of BC-trees and the SPQR-trees of blocks. It turns out that this implicit representation also simplifies the presentation of the structural result stated in Theorem 1.1. In this section, we point out some of the challenges with maintaining embeddings and triconnected components of incremental graphs, and give a high level overview of how we solve them.

1.3.1 For biconnected graphs

Once a graph is triconnected, its planar embedding is fixed. On the other hand, as soon as the graph has a separation pair, there is flexibility corresponding to flipping in said pair. Assuming first we have a biconnected graph, our approach goes via keeping track of the triconnected components and the separation cuts they have in common.

The SPQR-tree is a well-known structure that maintains information about triconnectivity in a graph. The triconnected components appear as nodes in the SPQR-tree, so called R-nodes, and the separation pairs that separate them appear as edges. However, because of the complex nature of triconnectivity, these are not the only gadgets in the tree: The fact that one separation pair may split the graph into more than two parts is reflected in the SPQR-tree by having P-nodes represent such *parallel* splits, and the fact that one may find more than two vertices arranged around a cycle such that any pair of them form a separation pair, is reflected in the SPQR-tree by having S-nodes represent such *series* splits.

It is well-known that the SPQR-tree maintains information about the possible embeddings of the biconnected graph. In fact, given a rooted SPQR-tree, any embedding can be encoded by annotating edges and vertices of the SPQR tree. The edges can be annotated with a bit telling whether the subgraph separated from the root by said pair should be flipped or not, and the P-nodes of degree k can given an annotation specifying one of the $(k-1)!$ possible different orderings of their children. When we want to make the embedding such that it is nearly ready for an edge insertion, this means that most of these annotations on the edges and nodes should be set in a way that need not change when the adversarial edge insertion happens. Specifically, we want to maintain the invariant that only $\mathcal{O}(\log n)$ annotations change upon any edge-insertion, and that each change to a P node annotation only corresponds to a *slide* operation (see Figure 1).

Our idea is simple: Given the rooted SPQR-tree, calculate the heavy path decomposition, and make sure that the annotations on the heavy edges and on those P nodes that are internal on the heavy path, are set in a favorable way: Assume the endpoints of an edge-to-be-inserted are represented by SPQR-nodes, and consider the path in the SPQR-tree connecting them. Then, we can afford to perform $\mathcal{O}(1)$ flips for each light edge, since there are only $\mathcal{O}(\log n)$ of those. We can also afford to perform $\mathcal{O}(1)$ flips for each distinct heavy path along the way, since there are only $\mathcal{O}(\log n)$ of those. But we can not afford to perform a number of flips proportional to the length of the heavy paths. Thus, we need to ensure that all the maximal segments of heavy paths that can be "covered" by an edge-insertion without violating planarity, are already embedded in a way such that said insertion is possible.

However, once the edge actually is inserted, the SPQR-tree may change drastically; up to $\Theta(n)$ SPQR-nodes may disappear, and up to $\Theta(n)$ new SPQR-nodes may arise. Providing each of the new SPQR-nodes with a new heavy child without breaking the invariant could cause up to $\Theta(n)$ flips. Fortunately, the nature of the new vertices is very predictable. The changes caused by an edge insertion all happen along a path: first, each S- and P-node on the path can give rise to up to two new SPQR-nodes of the same type, inheriting a subset of the original node's children, and then, the path is contracted. Again, we use the heavy path decomposition to overcome this challenge: By pre-splitting most S- and P-nodes with a heavy child (see Figure 2), we ensure that only $\mathcal{O}(\log n)$ end vertices of heavy paths may need to be split in order for an edge to be inserted.

Note that we can not just require all such nodes to be pre-split, as in some cases that would require $\Theta(n)$ nodes to be pre-split. In fact, finding a deterministic, history-independent rule for when to pre-split that needs only $\mathcal{O}(\log n)$ flips in the worst case, was one of the main technical challenges. By carefully choosing which nodes to pre-split, we become able to prove that the act of pre-splitting and choosing a heavy child does not cascade.

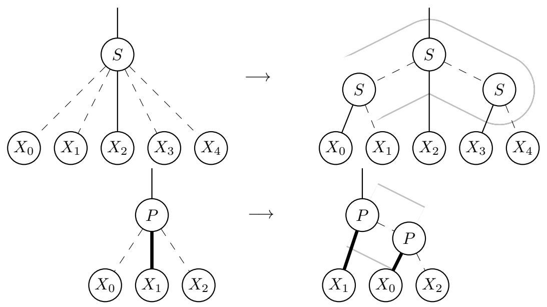


Figure 2: To handle edge insertions quickly, we pre-split the internal S- and P-nodes on a heavy path.

1.3.2 Connected graphs

If the graph is not biconnected, then it can be described via the block-cut tree (or *BC-tree*), as consisting of *blocks* or *biconnected components*, separated by cutvertices. The basic idea is to maintain an SPQR-tree for each block.

The main challenge is almost exactly the same as in the previous section: Insertion of an edge causes changes along a path, consisting of splits of cut-vertices along the path followed by a contraction of the path. This similar problem has a similar solution: again, we decompose the tree into heavy paths, pre-split vertices along each heavy path, and represent each heavy path in an easy-to-contract way. The easy-to-contract representation of each heavy path is easy to understand; we simply insert a *strut*, an edge from the head to the tail of the heavy path, thus covering the entire path and making it artificially biconnected. We then represent the solid path with an SPQR-tree, whose root is the S-node formed by adding the strut, and where the children of the root correspond to the blocks along the path.

In terms of planarity, this approach appears to be problematic, since the strut may violate planarity. To overcome this, we define a best-effort planar embedding, such that if the actual graph is always plane, the embedding allows the insertion of each strut across a face whenever possible, and we always know which struts are compatible with the planarity of the graph.

To maintain this heavy path decomposed BC-tree dynamically, we need to augment our structure for SPQR-trees with some extra operations that translate into link and cut in the tree. Namely, to concatenate two heavy paths, we want to go from two SPQR-trees, each rooted in an S-node, each having children corresponding to the blocks on the path, to one SPQR-tree rooted in one S-node, whose children correspond to all the blocks on the combined path. Similarly, cleaving a heavy path corresponds to splitting an S-node into two S-nodes, each inheriting the section of the children that corresponds to their particular sub-path.

Furthermore, because the inserted strut is only virtual, and can thus be removed or changed, we need the SPQR-trees to handle *undo* of edge-insertions. Loosely speaking, we maintain a dependency forest over the inserted edges, and allow the deletion of any edge that is the root of its tree in the forest. Vice versa, inserting an edge in the graph corresponds to inserting a new root in the dependency forest, linking it to some of the roots of the existing trees.

We have now sketched a structure for maintaining embeddings of a connected incremental graph: To insert an edge, find the corresponding path in the BC-tree. If we apply the well-known trick of forcing this path to be heavy, then this corresponds to only $\mathcal{O}(\log n)$ so-called “sever” and “meld” operations on heavy paths. Each of these operations perform $\mathcal{O}(1)$ updates to $\mathcal{O}(1)$ SPQR-trees. Each of these updates are handled by our structure for the SPQR-trees with $\mathcal{O}(\log n)$ changes. Thus, we have sketched how to embed an incremental connected graph with only $\mathcal{O}(\log^2 n)$ flips per insertion. To improve this to $\mathcal{O}(\log n)$, we need to apply a carefully chosen weighting of the SPQR-nodes in each SPQR-tree, and redefine the heavy paths and light edges accordingly. This ensures that for each edge insertion, the total number of involved light edges the BC-tree and in the SPQR-trees is still bounded by $\mathcal{O}(\log n)$.

1.3.3 General graphs. When the incremental graph is not connected, we maintain an embedding of each component. When an edge-insertion connects two components, the idea is to root each BC-tree in the endpoint of the new edge, and then link those to a root block corresponding to the new edge. This calls for an *evert operation*, forcing a node of the BC-tree to be root.

1.3.4 Algorithmic challenges. In addition to the combinatorial result about embeddings, we also give several data structures:

Biased dynamic trees To support the rest of our algorithmic results, we need to extend Sleator and Tarjan’s dynamic trees [25] (which maintain a heavy path decomposition of an unweighted tree) to handle vertex weights. While the extension seems obvious after the fact, we have been unable to find a prior description. We call the resulting data structure *biased dynamic trees*, by analogy with the Biased search trees by Bent et al. [3] (which are used as subroutines by both versions of dynamic trees).

Incremental SPQR/BC trees We present our biased dynamic trees as a nicely wrapped data structure with certain operations available, but for our actual use on pre-split BC/SPQR trees we have to open the black box and extend it to e.g. handle different kinds of node splitting. Also, the weights we use turn out to be hard to maintain explicitly. This is not an issue for the structural result, but costs an $\mathcal{O}(\log^2 n)$ factor in running time.

In relation to the description above, we need to argue that the contraction of a path in the SPQR-tree can be done in efficient worst-case time. To overcome this challenge, we use a representation of the SPQR-tree where the difference between a heavy path and a contracted heavy path is concisely encoded. The encoding has to be done in a clever way that allows a heavy path to be cleaved into two heavy paths, and, vice versa, allows adjacent heavy paths to be concatenated, when the heavy path decomposition undergoes dynamic changes. This is technically somewhat similar to, but more involved than, the corresponding data structure for biconnectivity [23].

Triconnectivity In order to answer triconnectivity queries, we need to augment our implicit representation of an SPQR-tree with enough information for us to use it to answer queries to whether a pair of vertices are triconnected.

The implicit representation of the SPQR-tree contains information about triconnectivity between vertices of the graph: to query for triconnectivity of a pair of vertices, all we need is to find an R- or P-node in the SPQR-tree that contains both of them.

Planarity testing The main argument for the bounded number of changes to the embedding has been sketched in Subsection 1.3.1. Those ideas lead to a data structure for maintaining an implicit representation of the SPQR-tree of a biconnected graph subject to edge-insertions, that handles each update in worst-case polylog time. Together with a well-chosen encoding of its embedding, we obtain a scheme for maintaining an embedding such that only $\mathcal{O}(\log n)$ *flips* (see Figure 1) are necessary to accommodate each new edge. Roughly speaking, this can be used to obtain an incremental worst-case $\mathcal{O}(\log^3 n)$ algorithm for planarity testing, by using [14] as a subroutine; we have argued that only $\mathcal{O}(\log n)$ flips are necessary, but each of them can be found in $\mathcal{O}(\log^2 n)$ time using the algorithm from [14].

Note that given the data structure from [14], we can answer in only $\mathcal{O}(\log^2 n)$ time whether the query edge is already compatible with the current embedding. Only when the embedding needs to change, the full $\mathcal{O}(\log^3 n)$ time is necessary.

Queries to the embedding Along with maintaining the canonical embedding dynamically, we are able to answer queries to the embedding. Given an edge incident to a specific vertex v, we can output its two neighbors in the circular ordering around v in $\mathcal{O}(1)$ time. If we need to distinguish between its right and left neighbor, we can use [14] to find the correct order in $\mathcal{O}(\log n)$ time.

1.4 Organization of the paper

We begin, in Section 2, with a formal definition of the distance between a pair of embedded graphs, which allows us to formally state our result about existence of "good" embeddings, as well as prove the matching lower bounds, that is, Theorem 1.2 and its corollary. In Section 3, we show how to extend the dynamic trees of Sleator and Tarjan [25] to *biased dynamic trees*, which we need in order to handle graphs that are not necessarily biconnected. Our main technical contribution lies in Section 4, in which we show how to efficiently maintain an implicit representation of the SPQR-tree of a block, and (in Section 4.6) we give a data structure for incremental triconnectivity for biconnected graphs. The corresponding versions for BC-trees, which we use to extend the incremental triconnectivity data structure to general graphs, is deferred to the full version [15]. In Section 5, we sketch how to annotate the implicit SPQR-tree to describe an implicit canonical embedding for biconnected graphs, and in the full version, we extend this to general graphs. Implementation details of the algorithmic results are deferred to the full version [15].

2 Flip-distance for labelled plane graphs

Unless otherwise noted, we will be working with undirected planar loopless multigraphs, with distinct vertex labels and distinct edge labels, both from some totally ordered, countable set.

Let Emb denote the (countably infinite) set of all plane embedded graphs that are embeddings of such graphs, and given a graph G, let $\mathrm{Emb}(G) \subseteq \mathrm{Emb}$ denote the set of all plane embeddings of G. We say that two plane embedded graphs $H, H' \in \mathrm{Emb}$ are *adjacent* if either:

- There is an edge $e \in H$ such that $H - e = H'$, or an edge $e' \in H'$ such that $H = H' - e'$, or
- There is a single *flip* (either an *articulation-flip* or a *separation-flip*, see Figure 1) that transforms H into H'.

Define the *flip-graph* $\mathcal{G}$ as the (countably infinite) graph with Emb as its vertices, and an edge for each pair of adjacent embeddings. For any two plane embedded graphs $H, H' \in \mathrm{Emb}$ we define the *flip-distance* $\mathrm{dist}(H, H')$ to be the length of any shortest path between H and H' in $\mathcal{G}$. Note that for any planar graph G with n vertices, the flip-distance between any two plane embeddings of G is $\mathcal{O}(n)$. We will extend this notion of distance to sets $A, B \subseteq \mathrm{Emb}$ by defining $\mathrm{dist}(A, B)$ to be the *Hausdorff distance* between A and B, that is:

$$\mathrm{dist}(A,B) := \max\left\{ \max_{a\in A} \min_{b\in B} \mathrm{dist}(a,b),\ \max_{b\in B} \min_{a\in A} \mathrm{dist}(a,b) \right\}$$

Our main results about embeddings can be stated as

Theorem 2.1. *We define a set* $\mathrm{Emb}^\star \subseteq \mathrm{Emb}$ *of* good *embeddings, and define* $\mathrm{Emb}^\star(G) := \mathrm{Emb}^\star \cap \mathrm{Emb}(G)$, *such that for any planar graph G with n vertices,* $\mathrm{Emb}^\star(G) \neq \emptyset$ *and for any edge e in G,*

$$\mathrm{dist}(\mathrm{Emb}^\star(G-e), \mathrm{Emb}^\star(G)) \in \mathcal{O}(\log n)$$

Proof. The proof for biconnected graphs is deferred to Section 5.2, see Theorem 5.1, and for general graphs, the proof is in the full version. □

Theorem 2.2. *Furthermore, we can maintain a good embedding for a dynamic planar graph under edge insertions and deletions in worst case $\mathcal{O}(\log n)$ flips per operation. The algorithm uses linear space and $\mathcal{O}(\log^3 n)$ worst case time per edge insertion, and worst case linear time per edge deletion.*

Thus, we can restrict ourselves to the set of *good* embeddings and never need more than $\mathcal{O}(\log n)$ flips per edge insertion or deletion. Note that the set of good embeddings is not necessarily small. There are graphs where $|\mathrm{Emb}^\star(G)|$ may be as large as $2^{\Omega(n \log n)}$ (See Figure 5).

Theorem 2.3. (Restatement of Theorem 1.1) *For any planar graph G with n vertices we can define a* canonical *embedding $\phi(G) \in \mathrm{Emb}^\star(G)$ such that for any edge e in G,*

$$\mathrm{dist}(\phi(G-e), \phi(G)) \in \mathcal{O}(\log n)$$

Proof. See the full version. the full version. ∎

So in fact, we can restrict ourselves to a particular *canonical* embedding of each graph and still not need more than $\mathcal{O}(\log n)$ flips per edge insertion or deletion. The choice of canonical embedding is, however, not unique.

These results should be contrasted with the following easy lower bounds.

Theorem 2.4. (Lower bound 1) *For any n, there is a planar graph G with $\mathcal{O}(n)$ vertices, and an edge $e \in G$ such that* $\mathrm{dist}(\mathrm{Emb}(G-e), \mathrm{Emb}(G)) \in \Omega(n)$.

Proof. The wheel graph W_n on $n \geq 5$ vertices is such an example. For any edge e on the outer rim, $W_n - e$ has an embedding that requires $n-4$ flips before e can be added (see Figure 3, top).

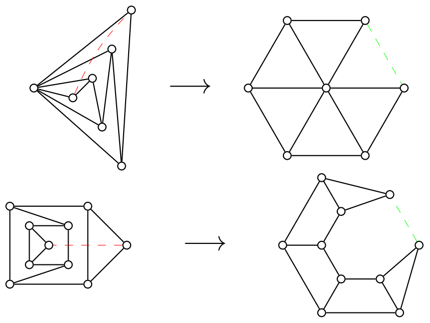


Figure 3: $W_7 - e$ and $G_6 - e$ may each need 3 separation flips to admit e.

The wheel graph has a vertex of high degree, but the result still holds for cubic graphs (see Figure 3, bottom). Start with the circular ladder graph CL_k with $2k$ vertices, contract two consecutive rungs, remove one of the duplicate edges, and call the other edge e. The resulting graph G_k has $n = 2k-2$ vertices, and $G_k - e$ has an embedding that requires $k-3 = n/2-2$ flips before e can be added. ∎

In other words, if your current embedding is bad, adding an edge may require $\Omega(n)$ flips.

Proof of Theorem 1.2. Let $h \geq 2$ be given. Take two identical complete binary trees of height h, and identify the corresponding leaves. Let L be the set of these joined leaves. The resulting graph G_h has $n = 3 \cdot 2^h - 2$ vertices, and for any pair of distinct $x, y \in L$, $G_h \cup (x,y)$ is planar (See Figure 4).

However, in any plane embedding $H \in \mathrm{Emb}(G_h)$ there is a pair of leaves $x, y \in L$ such that adding (x,y) to H requires $h-1 = \log_2((n+2)/3)-1$ flips. In other words, $\min_{H' \in \mathrm{Emb}(G_h \cup (x,y))} \mathrm{dist}(H, H') = \log_2((n+2)/3)$.

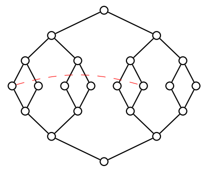


Figure 4: G_3 may need 2 separation flips to admit e.

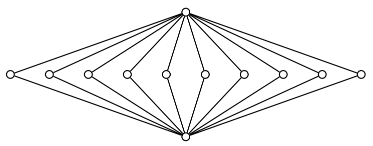


Figure 5: The graph $K_{2,k}$ has $n = k+2$ vertices and $(k-1)! = (n-3)! \in 2^{\Omega(n \log n)}$ distinct embeddings, all of which are good.

In other words, there are planar graphs where for any plane embedding there is some edge insertion that preserve the planarity of the underlying graph but where $\Omega(\log n)$ flips are required to get to an embedding that admits the edge.

3 Biased dynamic trees

This section is dedicated to extending existing data structures for dynamic trees such that they can handle vertex weights. We need these to exist in order to prove our combinatorial results, and we need to be able to maintain them efficiently in order to prove our algorithmic results.

Let T be a rooted tree, and suppose we mark each edge in T as either *solid* or *dashed* in such a way that each node has at most one solid child edge. This partitions the nodes of T into disjoint *solid paths* (a node incident to no solid edges is a path by itself), and we call such a collection of paths a *(naive) path decomposition* of T.

In their seminal paper on dynamic trees, Sleator and Tarjan [25] showed how to maintain a so-called *heavy path decomposition* of a tree in worst case $\mathcal{O}(\log n)$ time per operation. Here we extend their definition in the obvious way to trees where the nodes have nonnegative integer weights (with some restrictions), and the heavy paths are defined with respect to these weights. We will support the following "standard" update operations:

link(v, u)**:** Where v is a root and u is in some other tree. Link the two trees by adding the edge (v, u), making v a child of u.

cut(v)**:** Where v is not a tree root. Delete the edge $(v, p(v))$ between v and its parent, splitting the tree in two.

evert(v)**:** Change the root of the tree containing node v to be v.

reweight(v, i)**:** Where v is a node and i is a nonnegative integer (with some restrictions on when it can be zero). Change the weight of v to i.

Our goal is to have the operation times and the number of changes to the path decomposition during each of these operations be *biased* with respect to the weights of the relevant nodes. Informally, we can describe the result as follows:

There is a data structure for node-weighted dynamic trees that supports link(v, u), cut(v), evert(v), and reweight(v, w_{new}) in time proportional to $\log \frac{W}{\max\{w,1\}}$, where $w = \min\{w(v), w_{\text{new}}\}$ is the (current or subsequent) weight of the node v, and $W = \max\{w(T), w(T_{\text{new}})\}$ is the (current or subsequent) sum of node weights in the tree $T \ni v$.

However, in order to state our results precisely we need to dive a bit deeper into how these dynamic trees work. All proofs are deferred to the full version.

3.1 Definitions

Given a family of trees with nonnegative node weights, let $w(v)$ denote the weight of node v, and let $w(T_v)$ denote the total weight of the nodes in the subtree T_v rooted in v of the tree T that contains v. Given a child c of a node p, we call c and its parent edge (c, p) *heavy* if and only if $w(T_p) < 2w(T_c)$, and *light* otherwise. Note that with this definition, each node has at most one heavy child.

A *(proper) heavy path decomposition* is a path decomposition where an edge is solid if and only if it is heavy. It is sometimes useful to deviate slightly from this exact decomposition, in particular during certain updates. One particularly useful extension that warrants its own name is the *u-exposed heavy path decomposition*, which for any node u is defined as the unique path decomposition such that an edge $e = (v, p(v))$ is solid if and only if either

- $e \in r \cdots u$, or
- $p(v) \notin r \cdots u$ and e is heavy,

where r is the root. A proper heavy path decomposition is always a u-exposed heavy path decomposition for some choice of u, but not vice versa.

Given any path decomposition, for each node v, let $c(v)$ denote its solid child (or $\perp$ if v has no solid child), and let $h(v)$ denote its heavy child (or $\perp$ if v has no heavy child).

Define the *light depth* $\ell(v)$ of a node v as the number of light edges on the $r \cdots v$ path. A core property of the heavy/light definition is the following:

LEMMA 3.1. (LIGHT DEPTH) *In any tree T_r with nonnegative node weights, a node v with $w(T_v) > 0$ has* light depth $\ell(v) \leq \left\lfloor \log_2 \frac{w(T_r)}{w(T_v)} \right\rfloor$.

Similarly, we can define the *dashed depth* of a node v as the number of dashed edges on the $r \cdots v$ path. Note that in a u-exposed heavy path decomposition, the dashed depth of v is at most 1 more than $l(v)$.

3.2 Standard Operations

The main operations usually used for manipulating the path decomposition are:

expose(v)**:** Where v is a node in a proper heavy path decomposition of T. Make it v-exposed instead.

conceal(r)**:** Where r is the root of a v-exposed heavy path decomposition. Make it proper instead.

reverse(r)**:** Where r is the root of a v-exposed heavy path decomposition. Make v the root instead, and change it into an r-exposed heavy path decomposition (reverse every edge on $r \cdots v$).

link-exposed$(v, u; \ldots)$ Where v is the root of a proper heavy path decomposition and u is a node in an u-exposed heavy path decomposition with root $r \neq v$. Combine T_v and T_r into a single new tree by making u the dashed parent of v. The remaining parameters are used to initialize the new edge. The resulting tree is u-exposed.

cut-exposed(v)**:** Where v is a node in a u-exposed heavy path decomposition, where $u = p(v)$. Split T into two new trees T_r and T_v by deleting (v, u). Here T_r is u-exposed, and T_v is proper.

And to this we add

reweight-root(r, i)**:** Where r is the root of a u-exposed or proper tree T_r. Set the weight of r to i. If the original tree was proper, r may gain or lose a solid (heavy) child to become proper again. Otherwise the resulting tree is u-exposed (but not proper).

The expose and conceal operations are internally implemented in terms of the following 2 operations:

splice(x)**:** Where x is a light child of $y = p(x)$. If (x, y) is dashed, make the solid edge $(c(y), y)$ dashed and make (x, y) solid (set $c(y) := x$).

An expose(v) works by first making the solid child of v (if any) dashed, and then calling splice on each light child on the path from v to the root, in that order.

slice(x)**:** Where x is a light child of $y = p(x)$. If (x, y) is solid, make (x, y) dashed and make the dashed edge $(h(y), y)$ solid (set $c(y) := h(y)$).

A conceal(r) works by first calling slice on each light child on the selected path $r \cdots v$ from r to v in that order. Then if v has a heavy child, it makes the edge $(h(v), v)$ solid.

Note that during an expose or conceal, the path decomposition may temporarily not be u-exposed for any u, according to our definition. We could "fix" this by reversing the order of operations, but the number, order, and exact locations of these operations are going to be important for the following:

Lemma 3.2. *Calling expose(v) on a proper tree, or conceal(r) on a v-exposed tree, causes at most $2\ell(v) + 1$ edge changes. Each of link-exposed(v, r), cut-exposed(v), and reweight-root(v, i) change at most one edge, and reverse(r) changes no edges.*

Lemma 3.3. *If the cost of changing an edge $(v, p(v))$ where $w(T_v) > 0$ from solid to dashed or vice versa is $\mathcal{O}(1 + \log \frac{w(T_{p(v)})}{w(T_v)})$, then the cost of an expose(v) or conceal(r) where $w(T_v) > 0$ is $\mathcal{O}(1 + \log \frac{W}{w(T_v)})$, where W is the sum of the weights in the tree.*

Definition 3.1. *We say that a nonnegative weight function w is* k-positive *if and only if*

1. *Every node v of degree 1 has $w(v) > 0$.*
2. *Every path $u \cdots v$ of at least k nodes of degree 2 in T contains a node $c \in u \cdots v$ with $w(c) > 0$.*

Lemma 3.4. *We can implement biased dynamic trees with k-positive integer weights such that: expose(v), conceal(r), link-exposed($v, u; \ldots$), and cut-exposed(v) each take worst case $\mathcal{O}(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}})$ time; and reverse(r), and reweight-root(r, i) takes worst case constant time.*

It now follows that our algorithm maintains heavy path decompositions for a dynamic collection of rooted trees with k-positive integer node weights, in worst case $\mathcal{O}(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}})$ time per operation for link(v, u) and cut(v), worst case $\mathcal{O}(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}} + \log \frac{W}{\max\{w(r), \frac{1}{k}\}})$ time per operation for evert(v), and worst case $\mathcal{O}\Big(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}} + \log \frac{W'}{\max\{i, \frac{1}{k}\}}\Big)$ time per operation for reweight(v, i), where W and W' are the sums of the weights in the trees involved before and after the operation, and r is the root of the tree containing v before the operation.

More importantly, we can count both the number of separation flips in the SPQR trees (each such change costs 1), and the sum of those changes in the BC-trees, to get a total change cost of $\mathcal{O}(\log n)$:

Lemma 3.5. *If the cost of changing an edge $(v, p(v))$ from solid to dashed or vice versa is $\mathcal{O}\Big(1 + \log \frac{w(T_{p(v)})}{w(T_v))}\Big)$, then our algorithm maintains heavy path decompositions for a dynamic collection of rooted trees with k-positive integer node weights, in worst case $\mathcal{O}\Big(1 + \log \frac{W}{w(T_v)}\Big)$ cost per operation for link(v, u) and cut(v), worst case $\mathcal{O}\Big(1 + \log \frac{W}{w(T_v)} + \log \frac{W}{w(T'_r)}\Big)$ cost per operation for evert(v), and worst case $\mathcal{O}\Big(1 + \log \frac{W}{w(T_v)} + \log \frac{W'}{w(T'_v)}\Big)$ cost per operation for reweight(v, i), where W and W' are the sums of the weights in the trees T,T' involved before and after the operation, and r is the root of the tree containing v before the operation.*

Note that the denominators in each case have changed from the weights of single nodes to the weights of whole subtrees.

3.3 Heavy paths in a weighted tree decomposition

We want to use our algorithm for biased dynamic trees on BC and SPQR trees where the underlying graph have positive vertex weights. These are both special cases of *tree decompositions*[12, 24]. A tree decomposition for a connected graph G, is a pair $(T, \mathcal{B})$ where T is a tree and each node $u \in V[T]$ is associated with a *bag* $\mathcal{B}(u) \subseteq V[G]$, such that:

T1. $\bigcup_{v \in V[T]} \mathcal{B}(v) = V[G]$.

T2. For every path $u \cdots v$ in T, if $c \in V[u \cdots v]$, then $\mathcal{B}(u) \cap \mathcal{B}(v) \subseteq \mathcal{B}(c)$.

T3. For every $(x, y) \in E[G]$ there exists $u \in V[T]$ such that $\{x, y\} \subseteq \mathcal{B}(u)$.

For an edge (u, v) we write $\mathcal{B}((u, v)) = \mathcal{B}(u) \cap \mathcal{B}(v)$.

In principle, the tree in a tree decomposition can be arbitrarily large, because an edge can be subdivided any number of times. The traditional definitions of BC trees and SPQR trees can be described as the unique minimal tree decompositions with certain properties, which overcomes this. Our "relaxed" pre-split trees will not necessarily be minimal, but will have the property that there is a constant $a_{\max}$ (called the *adhesion*), such that for every edge (u, v):

R1. $1 \le |\mathcal{B}(u) \cap \mathcal{B}(v)| \le a_{\max}$. (bounded adhesion)

R2. If u is a leaf, then $\mathcal{B}(u) \setminus \mathcal{B}(v) \neq \emptyset$. (bounded leaves)

R3. If both u and v have degree 2 then $\mathcal{B}(u) \neq \mathcal{B}(v)$. (bounded paths)

For each vertex $x \in V[G]$ let $b(x)$ be the node u closest to the current root such that $x \in \mathcal{B}(u)$. Then for each tree node v we can define

$$b^{-1}(v) := \{x \in \mathcal{B}(v) \mid b(x) = v\}$$
$$w(v) := \sum_{x \in b^{-1}(v)} w(x)$$

Lemma 3.6. *For every leaf u, $w(u) > 0$.*

Lemma 3.7. *Let $a_{\max}$ be the adhesion of the tree decomposition, then any path consisting of $2a_{\max} + 2$ nodes of degree 2 in T has a node v with $w(v) > 0$.*

Thus, this system of weights is $(2a_{\max}+2)$-positive, and we can use them to define our biased dynamic trees. Furthermore, following from Lemma 3.1, we have the following useful property:

Lemma 3.8. *For any vertex $x \in V[G]$, and any node $v \in V[T]$ that is an ancestor to $b(x)$, the light depth of v is at most $\left\lfloor \log_2 \frac{w(T_r)}{w(x)} \right\rfloor$.*

Combining Lemma 3.8 with Lemma 3.2, we get the following Lemma that counts the number of heavy/light changes in the SPQR tree based on the weights from the BC tree:

Lemma 3.9. *For any vertex $x \in V[G]$, and any node $v \in V[T]$ ancestor to $b(x)$, the number of edge changes made to the heavy path decomposition by expose(v) on a proper tree, or by conceal(r) on a v-exposed tree is at most $1 + 2\left\lfloor \log_2 \frac{w(T_r)}{w(x)} \right\rfloor$. Each of link-exposed$(v, r)$, cut-exposed$(v)$, and reweight-root$(v, i)$ change at most one edge, and reverse(r) changes no edges.*

In fact, by combining with Lemma 3.3 we observe: If the cost of changing an edge $(v, p(v))$ from solid to dashed or vice versa is $\mathcal{O}\left(1 + \log \frac{w(T_{p(v)})}{w(T_v)}\right)$, then for any vertex $x \in V[G]$ and any node $v \in V[T]$ that is an ancestor to $b(x)$, the cost of an expose(v) or conceal(r) is $\mathcal{O}\left(1 + \log \frac{W}{w(x)}\right)$, where W is the sum of the weights in the tree. This observation above is useful, because it lets us sum the changes from the SPQR tree as we work in the BC tree. Furthermore, we have the following useful lemma, that will allow us to find critical paths in SPQR-trees:

Lemma 3.10. *Given a node v and a vertex $y \in \mathcal{B}(v)$, with v exposed, we can in constant time find the node v' closest to the root such that $y \in \mathcal{B}(v')$, and, symmetrically, given a vertex $x \in \mathcal{B}(r)$, find the node r' furthest from the root such that $x \in \mathcal{B}(r')$.*

Our definition of $b(x)$ and $w(v)$ presents a problem in connection with the reverse(r) operation. In particular, we can't store $b(x)$ or $w(v)$ explicitly, since too many of these values may change. Instead, for each vertex x, we store an *arbitrary* node $\hat{b}(x)$ such that $x \in \mathcal{B}(\hat{b}(x))$. And we explicitly maintain a weight $\hat{w}(v) = \sum_{x \in \mathcal{B}(v), \hat{b}(x) = v} w(x)$ for each node v. The relationship between $w(v)$ and $\hat{w}(v)$ is that

$$w(v) = \begin{cases} \hat{w}(v) & \text{if } v \text{ is the root} \\ \hat{w}(v) - \sum_{\substack{x \in \mathcal{B}((v,p(v))), \\ \hat{b}(x) \in T_v}} w(x) & \text{otherwise} \end{cases}$$

Thus, we can compute $w(v)$ from $\hat{w}(v)$ in the time that it takes to determine if v is an ancestor to $\hat{b}(x)$ for each of the at most $a_{\max}$ values of $x \in \mathcal{B}((v, p(v)))$.

Lemma 3.11. *Our algorithm maintains heavy path decompositions for a dynamic collection of weighted and rooted tree-decompositions of bounded adhesion $a_{\max}$, in worst case $\mathcal{O}\left(\left(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}}\right) \log^2 n\right)$ time per operation for link(v, u) and cut(v), worst case $\mathcal{O}\left(\left(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}} + \log \frac{W}{\max\{w(r), \frac{1}{k}\}}\right) \log^2 n\right)$ time per operation for evert(v), and worst case $\mathcal{O}\left(\left(1 + \log \frac{W}{\max\{w(v), \frac{1}{k}\}} + \log \frac{W'}{\max\{i, \frac{1}{k}\}}\right) \log^2 n\right)$ time per operation for reweight(v, i), where $k = 2a_{\max} + 2$, W and W' are the sums of the weights in the trees involved before and after the operation, and r is the root of the tree containing v before the operation.*

4 Dynamic SPQR trees

We now proceed to give a data structure for maintaining an implicit representation of an SPQR-tree of a weighted biconnected graph subject to edge insertions and undo edge insertion.

When an edge is inserted, all changes to the SPQR-tree happen along a path. In the most complicated case, this path is nontrivial, and everything along it becomes triconnected, and we have to update the SPQR-tree by substituting the path with a node representing its contraction. Thus, to handle such an edge-insertion, we need to find that SPQR-path and contract it.

Our approach goes via maintaining a path decomposition of the SPQR-tree, and regarding all solid paths as *precontracted* in a way that admits undo. Intuitively, the precontractions in question approximate insertions of imaginary edges. However, simply contracting a path in the SPQR-tree may differ in more than a constant number of places from what could be the SPQR-tree of any graph. Thus, we introduce the notion of *relaxed SPQR trees*, where internal nodes on heavy paths are *pre-split* in the way they would be if an edge would cover them. Once all solid paths are pre-contracted, the actual insertion of an edge covering the solid path, or its undo, can be implemented by toggling a bit for the heavy path. We show that we can maintain a path decomposition with pre-contracted solid paths with only $\mathcal{O}(\log n)$ changes per insertion or undo, where here changes include pre-splitting and its reverse operation.

Now, if the insertion of an edge in the biconnected graph causes a path in the SPQR-tree to be contracted, we need to find the endpoints of that path. Recall that either endpoint of the inserted edge may reside in several SPQR-nodes. Nonetheless, if there is no node containing both endpoints, there is a unique shortest path connecting them; their *critical path*, and we give an algorithm for finding this path. Otherwise, in case there exist SPQR-nodes that contain both endpoints of the inserted edge, we give an algorithm for finding the at most 3 such nodes.

Section overview: In Section 4.1, we recall the definition of SPQR-trees, and present our notion of a *relaxed* SPQR-tree that allows for pre-split nodes. In Section 4.2, we describe how the SPQR-tree may change upon an edge insertion. In Section 3.3, we show how to maintain a path decomposition in a weighted SPQR-tree; we generalize to tree-decompositions of bounded adhesion, and show how to maintain path decompositions of those. In Section 4.3, we show how to find that path in the SPQR-tree that needs to be modified in case an edge is inserted. In Section 4.4, we give the details of pre-splitting of SPQR-nodes. Finally, in Section 4.5, we sketch the pre-contracting of solid paths.

4.1 Strict and relaxed SPQR trees

Definition 4.1. ([16, p. 6]) *Let $\{a, b\}$ be a pair of vertices in a biconnected multigraph G. Suppose the edges of G are divided into equivalence classes $E_1, E_2, \ldots, E_k$, such that two edges which lie on a common path not containing any vertex of $\{a, b\}$ except as an end-point are in the same class. The classes E_i are called the* separation classes *of G with respect to $\{a, b\}$. If there are at least two separation classes, then $\{a, b\}$ is a* separation pair *of G unless (i) there are exactly two separation classes, and one class consists of a single edge , or (ii) there are exactly three classes, each consisting of a single edge.*

Definition 4.2. ([13]) *The* (strict) SPQR-tree *for a biconnected multigraph $G = (V, E)$ with at least 3 edges is a tree with nodes labeled S, P, or R, where each node x has an associated* skeleton graph $\Gamma(x)$ *with the following properties:*

- *For every node x in the SPQR-tree, $V(\Gamma(x)) \subseteq V$.*
- *For every edge $e \in E$ there is a unique node $x = b(e)$ in the SPQR-tree such that $e \in E(\Gamma(x))$.*
- *For every edge (x, y) in the SPQR-tree, $V(\Gamma(x)) \cap V(\Gamma(y))$ is a separation pair $\{a, b\}$ in G, and there is a* virtual edge *ab in each of $\Gamma(x)$ and $\Gamma(y)$ that corresponds to (x, y).*
- *For every node x in the SPQR-tree, every edge in $\Gamma(x)$ is either in E or a virtual edge.*
- *If x is an S-node, $\Gamma(x)$ is a simple cycle with at least 3 edges.*
- *If x is a P-node, $\Gamma(x)$ consists of a pair of vertices with at least 3 parallel edges.*
- *If x is an R-node, $\Gamma(x)$ is a simple triconnected graph.*
- *No two S-nodes are neighbors, and no two P-nodes are neighbors.*

The SPQR-tree for a biconnected graph is unique (see e.g. [7]). The P- and R-nodes are sometimes referred to as G's triconnected components.

In this paper, we use the term *relaxed SPQR tree* to denote a tree that satisfies all but the last of the conditions. Unlike the strict SPQR-tree, the relaxed SPQR-tree is never unique.

4.2 Changes during insert. Given a (strict) SPQR tree, we would like to be able to maintain it as the underlying graph changes. When adding edge (x, y), the change to the SPQR tree happens along the *critical path*, denoted $m(x, y)$ defined as:

1. If no SPQR node contains both x, y, then $m(x, y)$ is the unique shortest path $u \cdots v$ in the SPQR tree such that u contains x and v contains y.
2. If exactly one SPQR node v contains both x and y, then $m(x, y) = v$.
3. If exactly two SPQR nodes u, v contain both x and y, then they are adjacent and $m(x, y) = u \cdots v$.
4. If more than two SPQR nodes contain both x and y, then $m(x, y)$ is the unique P node containing both x and y.

Note that the path is considered to be undirected, so $m(x, y) = m(y, x)$.

This notion of a critical path is implicit in [5], where $m(x, y)$ is either the path connecting their representatives $\mu(x)$ and $\mu(y)$ of x and y, or, in the case where one of them (say, z) is a descendant of the representative of the other (say, z'), the path connecting $\mu(z)$ to μ, where μ is the lowest SPQR-node containing x and y in itself or its descendants.

The actual change to the SPQR-tree caused by inserting (x, y) can be described in terms of $m(x, y)$:

1. If $m(x, y)$ has at least two edges, or has a single edge that does not correspond to the separation pair $\{x, y\}$ (Case 1 above):

- Each S node s that is at the end of the path is split into at most 3 pieces wrt. the vertex in $\Gamma(s) \cap \{x,y\}$ and the closest edge on the path.
- Each S node that is internal to $m(x,y)$ is split into at most 3 pieces wrt. its neighboring edges on the path.
- Each P node that is internal to $m(x,y)$ is split into at most 2 pieces wrt. its neighboring edges on the path.
- And then the remainder of $m(x,y)$ is contracted into a single R node.

2. If $m(x,y)$ is a single S node s (part of Case 2 above): The node s is split into two S nodes s_1, s_2, with the edge (s_1, s_2) corresponding to the separation pair $\{x,y\}$ and we proceed as if the path had a single edge.
3. If $m(x,y)$ is a single edge (v_1, v_2) that corresponds to the separation pair $\{x,y\}$ (Case 3): The edge is subdivided by adding a new P node p with $V(\Gamma(p)) = \{x,y\}$, and we proceed as if the path consisted of the single node p.
4. If $m(x,y)$ is a single R node v and $\Gamma(v)$ already contains an edge $e = (x,y)$ (part of Case 2): A new leaf P node v' is added as neighbor of v, turning e into a virtual edge. Then the new (x,y) edge is added to $\Gamma(v')$, and we proceed as if the path consisted only of v'.
5. If $m(x,y)$ is a single P or R node v (rest of Case 2 and Case 4): The edge is simply added to $\Gamma(v)$.

4.3 Critical paths in SPQR-trees.

We will be working a lot with paths in T defined by a pair of vertices in $x, y \in V[G]$, and it will be useful to have the following query operation.

common-path(x,y)**:** Return the unique subpath $u \cdots v$ of $b(x) \cdots b(y)$ that is either

- the longest subpath of $b(x) \cdots b(y)$ such that $\{x,y\} \subseteq \mathcal{B}(u) \cap \mathcal{B}(v)$; or
- the unique shortest path in T such that $x \in \mathcal{B}(u)$ and $y \in \mathcal{B}(v)$.

Now, we can use Lemma 3.10 as a subroutine for finding and exposing $m(x,y)$:

LEMMA 4.1. *Given vertices x, y with $x \in \mathcal{B}(r)$ and $y \in \mathcal{B}(u)$, where u is exposed, $m(x,y)$ can be found in constant time.*

Proof. Let r' and v' denote the bags found in Lemma 3.10. They form the endpoints of the common path of x and y. If v' is further from the root than r', or r' is further from the root than v' and no P node lies between them, then v' and r' are the endpoints of the critical path. Otherwise, v' is at most 2 edges closer to the root than r', and between them, there exists a P node containing x and y; this P node is $m(x,y)$. ☐

LEMMA 4.2. *Let G be a vertex weighted biconnected graph with vertices $x, y, y' \in V[G]$. Let T be the SPQR tree for G. Let $m_T(x,y) = r \cdots u$ where $x \in \mathcal{B}(r)$ and $y \in \mathcal{B}(u)$, and let $m_T(x,y') = r' \cdots u'$ where $x \in \mathcal{B}(r')$ and $y' \in \mathcal{B}(u')$ and r' is on the path $r \cdots u'$. Then converting a u-exposed heavy path decomposition of T with root r into a u'-exposed heavy path decomposition of T with root r' can be done by: conceal(r), expose(r'), reverse(r), conceal(r'), expose(u'). During this sequence, $\mathcal{O}\left(1 + \log \frac{w(T)}{w(y)} + \log \frac{w(T)}{w(y')}\right)$ edges change from solid to dashed or vice versa.*

Proof. By Lemma 3.2 conceal(r) changes at most $2\ell(u) + 1$ edges. By definition of $m_T(x,y)$, if $y \in \mathcal{B}(r)$ then u and r are neighbors, so $\ell(u) \leq 1$; otherwise $u = b(y)$ and by Lemma 3.8 we have $\ell(u) \leq \left\lfloor \log_2 \frac{w(T)}{w(y)} \right\rfloor$. Thus, the number of edge changes made by conceal(r) is at most

$$2 \max\left\{1, \left\lfloor \log_2 \frac{w(T)}{w(y)} \right\rfloor\right\} + 1 \leq 2\left\lfloor \log_2 \frac{w(T)}{w(y)} \right\rfloor + 3$$

Similarly, by Lemma 3.2 expose(r') changes at most $2\ell(r') + 1$ edges. If $y' \notin \mathcal{B}(r')$ then r' is an ancestor to $b(y')$, so by Lemma 3.8 we have $\ell(r') \leq \left\lfloor \log_2 \frac{w(T)}{w(y')} \right\rfloor$. Otherwise, by definition of $m_T(x,y)$, $\{x, y'\} \subseteq \mathcal{B}(r') \cap \mathcal{B}(y')$, and since no 3 consecutive edges in a (strict) SPQR tree can correspond to the same separation pair, $b(y') \in \{r', p(r')\}$, so $\ell(r') \leq \ell(b(y')) + 1$ and so by Lemma 3.8 $\ell(r') \leq \left\lfloor \log_2 \frac{w(T)}{w(y')} \right\rfloor + 1$. Thus, the number of edge changes made by expose(r') is at most

$$\begin{aligned} 2 \max\left\{\left\lfloor \log_2 \frac{w(T)}{w(y')} \right\rfloor, \left\lfloor \log_2 \frac{w(T)}{w(y')} \right\rfloor + 1\right\} + 1 \\ = 2\left\lfloor \log_2 \frac{w(T)}{w(y')} \right\rfloor + 3 \end{aligned}$$

The reverse(r) does not change any edges, and by completely symmetric arguments, conceal(r') changes at most $2\left\lfloor \log_2 \frac{w(T)}{w(y)} \right\rfloor + 3$ edges, and expose(u') changes at most $2\left\lfloor \log_2 \frac{w(T)}{w(y')} \right\rfloor + 3$ edges. The total number of edge changes is thus at most

$$\begin{aligned} & 2\left(2\left\lfloor \log_2 \tfrac{w(T)}{w(y)} \right\rfloor + 3\right) + 2\left(2\left\lfloor \log_2 \tfrac{w(T)}{w(y')} \right\rfloor + 3\right) \\ & = 4\left\lfloor \log_2 \tfrac{w(T)}{w(y)} \right\rfloor + 4\left\lfloor \log_2 \tfrac{w(T)}{w(y')} \right\rfloor + 12 \\ & \in \mathcal{O}\left(1 + \log_2 \tfrac{w(T)}{w(y)} + \log_2 \tfrac{w(T)}{w(y')}\right) \end{aligned}$$ ☐

4.4 Pre-splitting.

We want to maintain an SPQR tree efficiently as the underlying graph changes. However, the strict SPQR tree can change by up to $\mathcal{O}(n)$

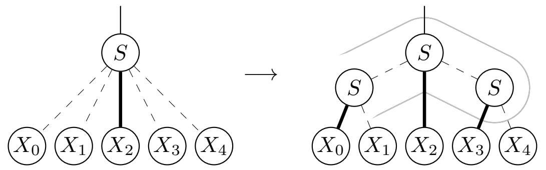


Figure 6: Pre-splitting an S node creates a primary and ≤ 2 secondary child nodes. The primary inherits the solid child, the secondaries partition the remaining children and choose their heaviest child as solid.

splits and the contraction of a path with up to $\mathcal{O}(n)$ nodes during a single edge insertion in the underlying graph, so we can not hope to maintain an explicit representation of that. In this section we define a *pre-split* SPQR tree, which is just a particular relaxed SPQR tree with a (u-exposed or proper) heavy path decomposition, that we have better control over. In particular, we ensure that an edge insertion only takes $\mathcal{O}(\log n)$ splits, followed by the contraction of a solid path. In Section 4.5 we show how we can handle this contraction efficiently. The main difficulty in handling the splits is that the expose/conceal/etc. operations will actually be changing the tree we are working on, by splitting and merging certain nodes. For the definition and analysis of our pre-split SPQR tree, we thus start with a (u-exposed or proper) heavy path decomposition of the strict SPQR tree.

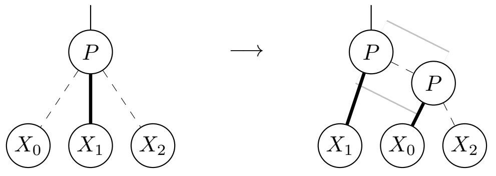


Figure 7: Pre-splitting a P node creates a primary and ≤ 1 secondary child node. The primary inherits the solid child, the secondary inherits the remaining children and chooses its heaviest child as solid.

Define a node v in a rooted relaxed SPQR tree as *primary* if either v is the root; or v is an R node; or v is an S or P node whose parent $p(v)$ has different type. call all other nodes *secondary.*
In any relaxed SPQR tree T there is a one-to-one correspondence between the primary nodes in T and the nodes in the corresponding strict SPQR tree.

Definition 4.3. *For each node u, we choose a set $s(u) \subseteq \mathcal{B}(u)$ of* selected *vertices, and use them to define* 1 *or* 2 child groups *of u as disjoint subsets of the edges of $\Gamma(u)$, as follows:*

- *If u is a primary S node with both a parent edge e_p and a solid child edge e_c, then $s(u) = \emptyset$, and the two child groups are simply the (possibly empty) ordered sets of edges on the clockwise and counterclockwise paths in $\Gamma(u)$ between the virtual edges corresponding to e_p and e_c (both excluded).*
- *For the at most one primary non-root S node u that has a parent edge e_p but no solid child and is connected to the root by a solid path, we may choose a vertex $x \in \mathcal{B}(u)$ and set $s(u) = \{x\}$. Then the two (possibly empty) child groups are the ordered sets of edges edges that correspond to the clockwise and counterclockwise paths in $\Gamma(u)$ from e_p to x (excluding e_p).*
- *If u is the root and is an S node with a solid child edge e_c we may choose a vertex $x \in \mathcal{B}(u)$, and set $s(u) = \{x\}$. Then the two (possibly empty) child groups are the ordered sets of edges that correspond to the clockwise and counterclockwise paths in $\Gamma(u)$ from x to e_c (excluding e_c).*
- *If u is the root and is an S node with no solid children, we may choose two selected vertices $x, y \in \mathcal{B}(u)$, and set $s(u) = \{x, y\}$. Then the two (possibly empty) child groups are the ordered sets of edges that correspond to the clockwise and counterclockwise paths in $\Gamma(u)$ between x and y.*
- *All other nodes (including R and P nodes and all secondary nodes) have $s(u) = \emptyset$, and only one (possibly empty) child group, consisting of all edges in $\Gamma(u)$ except the ones corresponding to the parent edge or the solid child.*

Note that for every dashed child, its corresponding edge in $\Gamma(u)$ belongs to a unique child group, and that the parent edge and solid child edge (if any) do not belong to a child group.

We are going to use the sets of selected nodes to control the exact locations where S nodes are split when preparing to insert a new edge. To do this we define a new operation:
select(v, X): Where $v \in \{r, u\}$ is an S node in a u-exposed SPQR tree with root r, and X (possibly empty) is a valid choice of selected vertices according to definition 4.3. Set $s(v) := X$.

Definition 4.4. *Define a node v to be* splittable *if:*

- *v is a P node with both a parent and a solid child and a child group with at least 2 edges; or*
- *v is an S node with two groups of dashed children and at least one child group with at least 2 edges.*

All other nodes are unsplittable. *In particular, the root, the leaves, and the R nodes are unsplittable.*

A node is mergeable *if it has a secondary child. All other nodes are* unmergeable.

In a strict SPQR tree many nodes may be splittable, but no node is mergeable. The relaxed SPQR tree is obtained by *pre-splitting* some of the splittable nodes of a strict SPQR tree, using the following operations:

split(v)**:** Where v is a splittable node. For each child group at least 2 edges, replace the group of dashed children corresponding to that group with a single new dashed secondary child node c inheriting the edges that it replaces, and with $s(c) = \emptyset$. For each new node c created this way, if it has a heavy child h make (h, c) solid.
Note that somewhat counter-intuitively, this definition may split an S node v with $|s(v)| = 2$ into 3, where the root node is a 2-cycle.

As the underlying graph changes, we also need the inverse operation:

merge(v)**:** Where v is a mergeable node. For each secondary child c: if (c, v) is dashed and c has a solid child c_2, make (c_2, c) dashed; then contract edge (c, v).

Lemma 4.3. *If we start with a heavy path decomposition of a strict SPQR tree with some set of selected vertices and only split or merge primary nodes, the following invariants hold:*

I-1. No node is both splittable and mergeable.

I-2. Each secondary node has a skeleton graph with at least 3 *edges, and is a dashed child of a primary node which is either a P node with both a parent and a solid child, or an S node with two groups of children.*

Furthermore, any heavy path decomposition of a relaxed SPQR tree where these invariants hold can be constructed from the corresponding strict SPQR tree by splitting a unique subset of the splittable primary nodes in any order.

In particular, invariant I-1 and I-2 means that each node in the original strict SPQR tree is represented by at most 3 (a primary and at most 2 secondary) nodes in the relaxed SPQR tree. This is somewhat analogous to the way a 2-3-4-tree is related to a red-black tree (See Guibas and Sedgewick [11]).

So far, we have said very little about which nodes to actually split and merge. Each time an edge in the strict SPQR tree changes from solid to dashed or vice versa during an update, our relaxed tree may have to change. We want to ensure that each such edge change causes at most a constant number of splits and merges. In order to get there, we need to distinguish between two types of dashed edges.

Define the *solid degree* of a node v to be $|s(v)|$ plus the number of solid edges incident to v. Define an edge $(c, p(c))$ to be *stable* if it is dashed and either

- $p(c)$ is an R node; or
- $p(c)$ is an S or P node with solid degree 2 and the child group of $p(c)$ containing $(c, p(c))$ contains no other edges.

All other dashed edges are *unstable*. In particular, every dashed child edge of an S or P node that is at the end of a solid path not containing the root is unstable, as are all child edges of a splittable node that would get moved during a split. Observe that if the edge $(c, p(c))$ is stable, then contracting the solid path containing $p(c)$ into an R node during an edge insertion does not change this.

Lemma 4.4. *If splitting or merging the node p does not give its dashed child c a new parent, then (c, p) is unstable after the operation if and only if it was unstable before the operation.*

Proof. Since c does not get a new parent, (c, p) was stable if and only if p was an S node of solid degree 2 and the child group p containing (c, p) had no other elements. The split does not change the solid degree of p, and only changes p's other child group. ∎

By similar arguments (deferred to the full version) we get the following two lemmas:

Lemma 4.5. *If splitting the node p gives its dashed child c a new parent v, then (c, p) was unstable before the split, (c, v) is either solid or unstable after the split, and $\Gamma(v)$ has at least* 3 *edges.*

Lemma 4.6. *If merging node p contracts the dashed edge (v, p) where v has a child c and $\Gamma(v)$ has at least* 3 *edges, then (c, v) was solid or unstable before the merge, and (c, p) is unstable after the merge.*

Thus, if the edge $(c, p(c))$ is solid or unstable then after a split or merge that changes $p(c)$, the new parent edge of c is still either solid or unstable (but not necessarily the same one as before). Given these properties we can now state

Lemma 4.7. *Given a u-exposed (or proper) heavy path decomposition of a strict SPQR tree T with root r, and sets $s(u)$ and $s(r)$, there is a unique corresponding relaxed SPQR tree T' that satisfies the invariants of Lemma 4.3 as well as the following:*

I-3. The root, and each primary node with a solid or unstable parent edge is not splittable.

I-4. A primary node with a stable parent edge is not mergeable.

We call T' the pre-split *SPQR tree corresponding to $(T, s(u), s(r))$.*

Proof. The strict SPQR tree trivially satisfies all invariants except I-3. Let X be the set of nodes in the strict

SPQR tree that violate I-3. For each node $v \in X$, calling split(v) satisfies I-3 at v, without breaking it for any other nodes. ∎

LEMMA 4.8. *If a relaxed SPQR tree violates the pre-splitting invariants in at most k nodes, then they can be restored by at most $20k$ split or merge operations.*

Proof. For each secondary node v that is a solid child, we have a violation of invariant I-2, and we can set $p = p(v)$ and do merge(p), followed (if the resulting node is splittable) by a split(p). Each such merge+split removes the offending node and violation, but may create up to 2 new violations of I-2 (at the new dashed children of p), and at most 4 new violations of I-3 (at stable children of v and $p(v)$ whose parent edges become unstable). After doing this for the at most k secondary nodes that are solid children (at a cost of at most $2k$ operations), every secondary node is a dashed child, and we have at most $6k$ other violations. For each secondary node v that is now a child of a secondary node, there is a violation of invariant I-2, and we can do a merge$(p(v))$. Each such merge removes the offending node and violation, but may create a new violation of I-1, I-2, or I-3 at the parent. However, after at most $6k$ such merges, every secondary node has a primary parent. If invariant I-2 is now violated at a secondary node v, then a merge$(p(v))$ will remove both the node and the violation, but may create a new violation of invariant I-1 or I-3 at the parent instead. So after at most $6k$ additional merges, invariant I-2 is satisfied at every secondary node, but we may still have $6k$ violations of the other invariants. If invariant I-1 and/or I-4 is now violated at a node v, then a merge(v) fixes it and reduces the total number of violations by at least one. Finally, if invariant I-3 is now violated at a v, then a split(v) fixes it and reduces the total number of violations by one. ∎

This is a very loose upper bound, and we can often do much better by analyzing the specific violations.

LEMMA 4.9. *In a u-exposed pre-split SPQR tree with root r, restoring the invariants after a select(u, X) or select(r, X) requires at most one split or merge.*

Proof. For $v \in \{u, r\}$, the select(v, X) operation can only affect invariants I-1 and I-3 at v and I-2 at children of v. A single merge or split will fix this without introducing any new violations. To see this, first note that if $s(v) \neq \emptyset$ then the only valid value for X is $X = \emptyset$. In this case, invariant I-2 may be violated at a child c of v because v only has one child group after the operation, and this is fixed by a merge(v). Afterwards, v is neither mergeable or splittable, so both I-1 and I-3 are also satisfied at v. If $s(v) = \emptyset$ and $X \neq \emptyset$, then v may become splittable and violate I-3, but this is fixed by split(v). ∎

LEMMA 4.10. *In the pre-split SPQR tree, if c and $p = p(c)$ are both primary, then changing edge (c, p) from solid to dashed or vice versa may violate the invariants at p or c or at most 3 of their children, but nowhere else. The invariants can be restored by at most 3 split or merge operations.*

Proof. Suppose first that (c, p) is changed from solid to dashed. This does not cause I-1 to be violated anywhere, but may cause I-2 to be violated at secondary children of p (because they lose a solid sibling), cause I-3 to be violated at at most one primary child of each of p or c (because their parent edge changes from stable to unstable), and cause I-4 to be violated at c (because its parent edge changes from solid to stable). Note that I-2 can only be violated at a secondary child of p if p *is not* an R node, and I-4 can only be violated at c if p *is* an R node, so these two types of violation do not occur together. To restore the invariants takes at at most one merge operation at either p (to fix the violation of I-2 at the secondary children of p) or c (to fix the violation of I-4 at c), and at most one split operation at each primary child v of p or c whose parent edge changes from stable to unstable (to fix the violation of I-3).

The case where (c, p) is changed from dashed to solid is near-symmetric (see the full version). ∎

To handle expose(u) of a primary node u in a proper heavy path decomposition of the pre-split SPQR tree with root r, or conceal(r) in a u-exposed heavy path decomposition of the pre-split SPQR tree with root r and primary node u, we need to slightly change the implementation of splice(v) and slice(v) as follows:

splice(v)**:** Where v is a primary node that is a dashed light child. If $p(v)$ is a secondary node, let $p = p(p(v))$, else let $p = p(v)$. Now p is a primary node. If p has a solid child then make it dashed and do the necessary splits and merges to restore the invariants. Now p is the parent of v and (v, p) is dashed. Make (v, p) solid, and do the necessary splits and merges to restore the invariants.

slice(v)**:** Where v is a primary node that is a solid light child. Let $p = p(v)$, then p is primary and (v, p) is solid. Make (v, p) dashed and do the necessary splits and merges to restore the invariants. Now p has no solid children. If p has a heavy child h, make (h, p) solid and do the necessary splits and merges to restore the invariants.

LEMMA 4.11. *An expose(u) where u is a primary node in a proper heavy path decomposition of the pre-split*

SPQR tree with no selected vertices and root r, or a conceal(r) in a u-exposed heavy path decomposition of the pre-split SPQR tree with no selected vertices and root r each costs at most $6\ell(u)+3$ splits or merges, in addition to the at most $2\ell(u)+1$ edges changing from solid to dashed or vice versa, where $\ell(u)$ is the light depth of u in the corresponding strict SPQR tree.

Proof. With the modified definition of splice and slice given, expose(u) and conceal(r) work exactly as before, except that after changing the solid/dashed state of a child of u we need to do the necessary splits and merges to restore the invariants. The number of edges we explicitly change is $2\ell(u)+1$. Each edge that we explicitly change from solid to dashed or vice versa has a primary node at each end, so by Lemma 4.10, restoring the invariants costs at most 3 splits or merges per explicit edge change. Thus the total number of merges and splits is at most $3(2\ell(u)+1) = 6\ell(u)+3$. □

Lemma 4.12. *Given a proper pre-split SPQR tree T_v with no selected vertices, and an u-exposed pre-split SPQR tree T_r, where (x,y) is a non-virtual edge in both $\Gamma(v)$ and $\Gamma(u)$, restoring the invariants after a link-exposed(v,u) takes at most one split or merge.*

Proof. After link-exposed(v,u), invariant I-2, I-3, or I-4 may be violated at v, but no other violations occur anywhere in the tree. If I-2 is violated at v, by definition of secondary v must have the same type as u, so the violation is fixed with a merge(u). If I-3 or I-4 is violated at v then v was a node of different type from u before the operation, so the violation is fixed with either a split(v) or a merge(v). Thus, in any case, at most one split or merge operation is needed in total to restore the invariants. □

Lemma 4.13. *Given a u-exposed pre-split SPQR tree T_r and a child v of u, restoring the invariants after a cut-exposed(v) takes at most one merge operation if v is a primary node, and no violations of the invariants occur otherwise.*

Proof. No violations of the invariants occur in the part of the tree containing u, because no parents or child groups are changed there. If v was a primary node before, then after cut-exposed(v) invariant I-2 may be violated at children of v (because $s(v)=\emptyset$ and v loses its parent). If this happens, a merge(v) fixes it. If v was secondary, every child of v is primary so no violations of I-2 can occur there. By definition, $s(v)=\emptyset$ so v is not splittable so I-3 is satisfied at v, and v has no parent, so I-4 is satisfied at v. All other invariants remain unchanged. □

Lemma 4.14. *In the pre-split SPQR tree with root r, reverse(r) does not violate any of the invariants.*

Proof. Let u be the other end of the solid path containing r. Since no node changes whether it is splittable or mergeable, invariant I-1 is trivially preserved. By invariant I-2, each node $v \in r\cdots u$ is primary. By definition of splittable and invariant I-3, every node $v \in r\cdots u$ is unsplittable before the operation. This will still be true after the operation, so invariant I-3 is preserved for these nodes. None of the nodes on $r\cdots u$ have a stable parent before or after the operation, so invariant I-4 is trivially preserved for these nodes. Each child c of a node $p \in r\cdots u$ is either a secondary node where I-2 is preserved, or a primary node, whose type of parent edge (stable or unstable) is preserved and thus I-3 and I-4 is preserved for c. The invariants are trivially preserved for the remaining nodes. □

Combining these Lemmas, we can show that we can support the aforementioned operations while only making changes proportional to the number of edges that change status between solid and dashed:

Theorem 4.1. *The number of splits and merges needed to maintain a pre-split SPQR tree under expose, conceal, reverse, link-exposed, and cut-exposed is proportional to the number of edges changing from solid to dashed or vice versa during the same operation in the strict SPQR tree.*

4.5 Pre-contracting

We have now shown that a pre-split SPQR tree does not change too much during each of the operations we need. However, we still need to describe how to handle the actual path contraction that happens during an edge insertion, and the corresponding path expansion that happens during an uninsert, in worst case $\mathcal{O}(\log n)$ time.

Given a pre-split SPQR tree T_r, we may define the operations contract and expand as follows: If T_r is u-exposed and each $v \in \{r,u\}$ is either an R node, or an S node with $|\mathcal{B}(v)| = 3$, *contract-path*(u) contracts the path $r\cdots u$ into a single *pseudo-R* node v, resulting in a v-exposed pre-split SPQR tree. Reversely, if T_r is r-exposed and r is a pseudo-R node contracted from a path $u\cdots v$, *expand-path*(r) expands the node r into that path. The result is either a u-exposed pre-split SPQR tree with root v, or a v-exposed pre-split SPQR tree with root u.

The idea is to represent each R node and each solid path as variations of the same tree data structure, such that one can be turned into the other just by changing information at the root of that tree.

Note that every R node we have, will be the result of some earlier contraction, and that because of pre-splitting every S or P node that gets contracted has a skeleton graph with 3 or 4 edges.

We define a *path representation* $\Pi(v)$ of an SPQR node v, as a graph constructed from v by creating a vertex $\pi(v_i)$ for each original S or P node v_i it was contracted from, and connecting them with edges into a path. We assign each edge a positive integer weight. Note that for an S or P node v, the path representation is just the single-vertex graph $\Pi(v) = (\{\pi(v)\}, \emptyset)$.

Given path representations $\Pi(c)$ for each node c on a solid path $u \cdots v$ in the SPQR tree, we can create a combined path representation $\Pi(u \cdots v)$ of the whole path by connecting the ends of the paths for each node with new edges of weight 0. Thus, in $\Pi(u \cdots v)$, each node in $u \cdots v$ corresponds to a maximal segment of the path where every edge has positive weight. In particular, each R node corresponds to a segment with at least one edge. Now, the weight of a vertex x is added to its representative $a(x)$ defined in Section 3.3. However, since we do not explicitly maintain the SPQR-nodes, the weight of x is added to a fragment on the path representing this node.

The idea is now that when doing a contract-path(r), resulting in a new root r', we can get from $\Pi(r \cdots u)$ to $\Pi(r')$ by simply adding 1 to all edges on $\Pi(r \cdots u)$. Similarly, if we later do an expand-path(r'), we can reconstruct $\Pi(r \cdots u)$ from $\Pi(r')$ by subtracting 1 from every edge. By storing each path using a balanced binary search tree, and doing the edge updates lazily, we can turn this idea into an effective data structure with $\mathcal{O}(\log n)$ worst case time for each operation.

Lemma 4.15. *[25] We can maintain such a tree in $\mathcal{O}(\log n)$ time per join, split, or add-to-path.*

In the resulting tree over $\Pi(u \cdots v)$, for every node c in $u \cdots v$ there is a unique deepest node $\pi(c)$ such that the subtree rooted in $\pi(c)$ contains all vertices of $\Pi(c)$. If v is an S or P node in the SPQR tree, this will be a leaf (and so $\pi(c)$ is the same as already defined above), but if c is an R node this will be not be the case.

We have defined all our operations on pre-split SPQR trees in terms of the actual nodes, but because we want the path representation tree to be balanced at all times, the node in this tree that correspond to an R node may change. Thus, every pointer to an R node will have to be implicit, in the sense that what is actually stored is a pointer to one of the (now obsolete) S or P nodes that it was contracted from, and every time we need the node we have to find it in the tree.

Lemma 4.16. *Let $u \cdots v$ be a solid path in the pre-split SPQR tree, with u an ancestor to v. Let $\Pi = \Pi(u \cdots v)$ be a path representation tree for $u \cdots v$, and let u' be en end fragment of Π that is part of $\Pi(u)$.*

Given a fragment $\pi(c') \in \Pi$ of an SPQR node $c \in u \cdots v$, we can in $\mathcal{O}(\log n)$ time find the first zero-weight edge on $\pi(c') \cdots \pi(u')$ (if any); and the internal node representing $\pi(u)$. And in $\mathcal{O}(\log^2 n)$ time we can find the first zero-weight light *edge on $\pi(c') \cdots \pi(u')$ (if any).*

The proof is deferred to the full version.

As the path decomposition (and the tree) changes, we will use the following internal operations to update the path representations of each solid path:

join-path$(\pi(v))$**:** Where v is a dashed child with no solid siblings. This operation is called when the edge $(v, p(v))$ changes from dashed to solid. Join the path representation trees Π_v and Π_p containing $\pi(v)$ and $\pi(p(v))$ into a single tree by adding a new internal node representing a weight 0 path-edge between an end of Π_v and an end of Π_p, and rebalancing as needed.

split-path(v)**:** Where v is a solid child. This operation is called when the edge $(v, p(v))$ changes from solid to dashed. Let a be the node on the solid path containing v that is closest to the root. Split the path representation tree Π_v containing $\pi(v)$ into two by deleting the first 0-weight edge on the path from $\pi(v)$ to $\pi(a)$ in Π_v.

These join- and split-operations can be supported in a time bounded by that stated in Lemma 4.16

Observation 4.1. *Join-path and split-path can be supported in $\mathcal{O}(\log^2 n)$ time, and contract can be supported in $\mathcal{O}(\log n)$ time.*

We are now ready to formally define the insert and undo insert operations:

insert$(r; x, y)$**:** In the SPQR-tree rooted in r with $m(x, y)$ exposed, update the SPQR-tree reflecting that the edge (x, y) is inserted.

uninsert$(r; x, y)$**:** In the SPQR-tree rooted in r with r exposed and $(x, y) \in \Gamma(r)$, update the SPQR-tree reflecting that the edge (x, y) is removed.

Lemma 4.17. *Given vertices $x, y \in V[G]$ and a u-exposed pre-split SPQR tree T_r where $r \cdots u = m(x, y)$, restoring the invariants during insert$(T_r; x, y)$ takes at most $\mathcal{O}(1)$ splits and merges. For undoing the insert in the resulting tree T with an uninsert$(T; (x, y))$, restoring the invariants takes the same number of splits and merges.*

Proof. If $x \in \mathcal{B}(r) \setminus \mathcal{B}(u)$ and $y \in \mathcal{B}(u) \setminus \mathcal{B}(r)$, then the insert consist of a select$(r, \{x\})$, a select$(u, \{y\})$, and a contract-path(r). By Lemma 4.9, each select costs at most one split, and then the contract-path does not violate any of the invariants.

If $r = u$ is an S node, then the insert starts by a select$(r, \{x, y\})$. Restoring the invariants then costs at most one split, which turns r into a kind of pseudo-S node with only two children. This node is then converted to a pseudo-P node (which doesn't violate any of the invariants), and the edge (x, y) is added to $\Gamma(r)$ turning it into a real P node.

If $r \cdots u$ consists of a single edge (r, u) with $\mathcal{B}((r, u)) = \{x, y\}$, this edge is first subdivided by adding a new pseudo-P node c between r and u (which doesn't violate any of the invariants), and then, the edge (x, y) is added to $\Gamma(c)$, turning it into a real P node.

If $r = u$ is a real R node and $\Gamma(r)$ already contains an edge (x, y), then a new pseudo-P root r' is added containing (x, y), and the new duplicate (x, y) edge is added to $\Gamma(r')$ making it a real P node.

Finally, if $r = u$ is an R node without an (x, y) edge, or a P node, (x, y) is simply added to $\Gamma(r)$.

Thus, in all cases, the number of merges and splits is at most 2. □

4.6 Incremental SPQR-trees of biconnected graphs In conclusion, there is a data structure for updating an implicit representation of the SPQR-tree of a biconnected graph subject to edge insertion and undo edge insertion. Each update gives rise to $\mathcal{O}(\log n)$ changes in the implicit representation, and the update is supported in $\mathcal{O}(\log^2 n)$ time.

Proof of Theorem 1.3 (biconnected version). Let all vertices have weight 1, and build the implicit representation of the SPQR-tree. Upon query, given vertices a, b of the graph, find their critical path $m(a, b)$. If $m(a, b)$ is a single R or P node, the answer is yes, otherwise, the answer is no. Since the time for finding $m(x, y)$ is $\mathcal{O}(\log n)$ and insertions are handled in $\mathcal{O}(\log^2 n)$ time, the data structure has an update time of $\mathcal{O}(\log^2 n)$ and a query time of $\mathcal{O}(\log n)$.

5 Embeddings of biconnected planar graphs

In this section we show how we can use the dynamic SPQR trees to control the embedding of a biconnected graph under edge insertions with backtracking. First, we define an abstract scheme for describing an embedding, given a relaxed SPQR tree. The crucial part of this description is the definition of certain *flip-bits*, where toggling a flip-bit corresponds to performing a separation-flip in the embedded graph (see Section 5.1). We go on to define *good embeddings* as those where almost all flip-bits on the heavy paths are set in a way that is favorable of an edge insertion covering it (see Section 5.2). As an example of a good embedding, the canonical embedding is specified by making arbitrary but consistent choices whenever the embedding is not uniquely determined by our definition of good embeddings (see full version).

5.1 Specifying an embedding in an SPQR tree

Given a biconnected planar graph $G = (V, E)$, and any relaxed SPQR tree for it, we can define a unique embedding of G by choosing:

- A *root* for the SPQR tree.
- A *flip bit* for each SPQR edge.
- A *local embedding* of $\Gamma(v)$ for each SPQR node v. Note: For S, P, and R nodes, respectively, the number of choices are 1, $(d(v) - 1)!$, and 2.

Given these choices, we can now construct a unique embedding of G. For each node v in the SPQR tree, define G_v as the subgraph of G induced by the vertices in T_v, possibly together with a virtual edge representing the rest of the graph. Thus $G_r = G$. In bottom-up order, construct $\phi(G_v)$ from the embeddings $\phi(G_{c_1}), \ldots, \phi(G_{c_k})$ of its children as follows:

- If v is a leaf, $\phi(G_v)$ is the local embedding of $\Gamma(v)$.
- Otherwise, take each child c_i, construct $\phi(G_{c_i})$ recursively, flip it if the flip-bit for (v, c_i) is set, and choose an outer face that contains the virtual edge corresponding to (v, c_i), and delete that virtual edge to get a plane graph C_i. In the local embedding of $\Gamma(v)$, take each virtual edge corresponding to a child edge (v, c_i) and replace it with its C_i.

Note that not all combinations of these give distinct embeddings, e.g.:

- Changing the root either does nothing or flips the whole embedding.
- For a non-root S node v, changing all the incident flip-bits gives exactly the same embedding.
- For a non-root P node or R node, changing all the incident flip-bits and flipping the local embedding of $\Gamma(v)$ gives exactly the same embedding.

Thus, toggling a flip bit or moving a single edge to a new position in the local embedding of a P node each correspond to a *separation flip* in the corresponding embedding G. Thus, if we can bound the number of these changes during an edge insertion, we bound the number of separation flips.

5.2 Good embeddings For a biconnected planar graph $G = (V, E)$, we will define our class of *good* embeddings $\text{Emb}^\star(G)$ based on the family of pre-split SPQR trees for G, by restricting the values of the flip-bits for some solid edges.

Let $x, y \in V$ be distinct vertices in the biconnected planar graph $G = (V, E)$. Let T be the unique pre-split

SPQR tree for G that is u-exposed and has root r such that $m(x,y) = r \cdots u$, and where for $v \in \{r,u\}$, if v is an S node, then $s(v) = \mathcal{B}(v) \cap \{x,y\}$.

We will define a set $\mathrm{Emb}^\star(G;x,y)$ of good embeddings of G by defining flip-bits for (most of) the solid edges in T. Given those, we may define the set of good embeddings $\mathrm{Emb}^\star(G)$ as the union $\bigcup_{x,y\in V} \mathrm{Emb}^\star(G;x,y)$. Our goal is to define $\mathrm{Emb}^\star(G;x,y)$ such that if $G \cup (x,y)$ is planar then any $H \in \mathrm{Emb}^\star(G;x,y)$ admits inserting (x,y).

Lemma 5.1. *For any biconnected planar graph G with n vertices, and any edge $(x,y) \in G$:* $\mathrm{dist}(\mathrm{Emb}^\star(G-(x,y);x,y), \mathrm{Emb}^\star(G;x,y)) \in \mathcal{O}(1)$.

By additionally making the flip-bits for each solid edge depend only on its closest few neighbors on the solid path, we then immediately get:

Lemma 5.2. *For a biconnected planar n-vertex graph G, and any pairs of distinct vertices x,y and x',y':* $\mathrm{dist}(\mathrm{Emb}^\star(G;x,y), \mathrm{Emb}^\star(G;x',y')) \in \mathcal{O}(\log n)$.

And combining Lemma 5.1 and 5.2 then gives:

Theorem 5.1. *For any biconnected planar graph G with n vertices, and any edge $e \in G$:* $\mathrm{dist}(\mathrm{Emb}^\star(G-e), \mathrm{Emb}^\star(G)) \in \mathcal{O}(\log n)$

Proof. Let $e = (x,y)$. Then, by the triangle inequality

$$\begin{aligned}\mathrm{dist}(&\mathrm{Emb}^\star(G-e), \mathrm{Emb}^\star(G))\\ &\le \mathrm{dist}(\mathrm{Emb}^\star(G-e), \mathrm{Emb}^\star(G-e;x,y))\\ &\quad + \mathrm{dist}(\mathrm{Emb}^\star(G-e;x,y), \mathrm{Emb}^\star(G;x,y))\\ &\quad + \mathrm{dist}(\mathrm{Emb}^\star(G;x,y), \mathrm{Emb}^\star(G))\end{aligned}$$

By Lemma 5.2 the first and last terms are $\mathcal{O}(\log n)$ and by Lemma 5.1 the middle term is $\mathcal{O}(1)$. ☐

Let T be a pre-split SPQR tree that is u-exposed and has root r where $r \cdots u = m(x,y)$. For each solid path M in T, define the *relevant part* of M as the maximal subpath M_r of M that does not end in a P node. If M consists only of a P node, M_r is the empty path. For any path H, let $\Gamma(H)$ denote the graph "glued" from the skeleton graphs on H.

We define flip-bits for M_r by defining a unique embedding for $\Gamma(M_r)$ and letting the flip-bits be exactly the ones required to describe that embedding.

For each R node that is internal to a solid path, we say that it is *happy* if the two virtual edges corresponding to the path are in the same face, and *cross* otherwise. Any other node is happy if it is internal on a solid path, and cross if it is isolated or at the end of a solid path.

Define a *run* on the relevant part of a solid path as any maximal subpath with at least one edge and no internal cross nodes. Note: a run cannot end in a P node. For every run $A = v_1 \dots v_2$, define the *strut* $r(A)$ as the edge $r(A) := (r(A,v_1), r(A,v_2))$, where $r(A,v_i)$ is the vertex with smallest id that is in the same face in $\phi(\Gamma(v_i))$ as the virtual edge e_{v_i} incident to v_i on A, but is not an endpoint of e_{v_i}. As a special case, if $A = m(x,y)$ then treat x and y as having $-\infty$ in this comparison. Thus, if $G \cup (x,y)$ is planar and $m(x,y)$ is not a single P node, $r(m(x,y)) = (x,y)$.

Definition 5.1. *Consider the graph $\Gamma'(M_r)$ obtained from $\Gamma(M_r)$ by adding $r(A_1),\dots,r(A_k)$ where $A_1,\dots,A_k$ are the runs of M_r. This graph is a subdivision of a planar triconnected graph and thus has exactly two planar embeddings that are mirror images of each other. Choose one of these embeddings to be $\phi(\Gamma'(M_r))$, and define $\phi(\Gamma(M_r))$ to be the corresponding embedding of M_r obtained by removing the struts from $\phi(\Gamma'(M_r))$. In particular, if $G \cup (x,y)$ is planar and $m(x,y)$ is not a single P node, the embedding of $\Gamma(m(x,y))$ must have x and y in the same face.*

Call a P node on the relevant part of a solid path *up-free* (resp. *down-free*) if is incident to an S node containing the root-nearest (resp. root-furthest) endpoint of the strut.

Lemma 5.3. *Let M_r be the relevant part of a solid path. Given $\phi(\Gamma(M_r))$, and given local embeddings of all P and R nodes on M_r, there is a unique assignment of flip-bits to all internal edges on M_r, such that for every internal S node or up-free P node $v \in M_r$, the flip-bit of $(v,p(v))$ is 0, and for every internal down-free P node $v \in M_r$ with heavy child $h(v)$, the flip-bit of $(h(v),v)$ is 0.*

Proof. For an edge between R nodes on M_r, this is trivial, since every assignment of flip-bits to the edge gives a different embedding, and only one of these is $\phi(\Gamma(M_r))$. If v is an internal S node on M_r, then changing the flip-bits of both its incident edges does not change the embedding. In particular, we can always choose the flip-bit of $(v,p(v))$ to be zero.

If v is a P node, then because of pre-splitting, v has degree 3. Thus, there are only two different circular orderings of its incident edges, both of which have its heavy edges as neighbors. If v is not free, every setting of flip-bits would lead to a different embedding, and only one of these is $\phi(\Gamma(M_r))$. If v is a free P node, then the flip-bit on the heavy edge connecting it to an endpoint of the strut is irrelevant to the embedding, and can be set to 0. ☐

For any biconnected planar graph $G = (V, E)$ and any $x, y \in V$, let $\text{Emb}^\star(G; x, y) \subseteq \text{Emb}(G)$ be the class of embeddings of G corresponding to choosing arbitrary embeddings of P and R nodes, setting the flip-bits of all internal edges of the relevant part of each solid path in the pre-split SPQR tree that is u-exposed and has root r where $r \cdots u = m(x, y)$ as in Lemma 5.3, and letting all other flip-bits be arbitrary.

Lemma 5.4. *Changing an edge from solid to dashed or vice versa, or doing a merge or split in the pre-split SPQR tree, requires changes to at most $\mathcal{O}(1)$ flip-bits.*

Reversing the solid root path requires no flips.

Proof of Lemma 5.1. By Definition 5.1, the embedding $\phi(\Gamma(m(x, y)))$ admits adding (x, y). By Lemma 4.17 $\text{insert}(T; (x, y))$ changes only a constant number of edges, so by Lemma 5.4 at most a constant number of flip-bits need to be changed.

Proof of Lemma 5.2. By Lemma 4.2, the number of edge changes when transforming the pre-split SPQR tree is $\mathcal{O}(1 + \log \frac{w(T)}{w(x)} + \log \frac{w(T)}{w(y)} + \log \frac{w(T)}{w(x')} + \log \frac{w(T)}{w(y')})$, and by Lemma 5.4 this is also the number of flip-bits that need to be changed.

References

[1] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.

[2] T. Avitabile, C. Mathieu, and L. Parkinson. Online constrained optimization with recourse. *Inf. Process. Lett.*, 113(3):81–86, 2013.

[3] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[4] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In *WADS '99, Vancouver*, pages 342–351, 1999.

[5] Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing (extended abstract). In *FOCS'89*, pages 436–441, 1989.

[6] Giuseppe Di Battista and Roberto Tamassia. On-line graph algorithms with SPQR-trees. In *ICALP'90, Warwick*, pages 598–611, 1990.

[7] Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.

[8] David Eppstein. Dynamic generators of topologically embedded graphs. In *SODA '03, Baltimore*, pages 599–608. ACM/SIAM, 2003.

[9] Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *J. ACM*, 46(1):28–91, 1999.

[10] Edward F. Grove, Ming-Yang Kao, P. Krishnan, and Jeffrey Scott Vitter. Online perfect matching and mobile computing. In *WADS '95, Kingston, Ontario*, pages 194–205, 1995.

[11] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS'78, Ann Arbor*, pages 8–21, 1978.

[12] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, Mar 1976.

[13] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Lacki, and Eva Rotenberg. Decremental spqr-trees for planar graphs. In *ESA '18, Helsinki*, pages 46:1–46:16, 2018.

[14] Jacob Holm and Eva Rotenberg. Dynamic planar embeddings of dynamic graphs. *Theory Comput. Syst.*, 61(4):1054–1083, 2017.

[15] Jacob Holm and Eva Rotenberg. Worst-case polylog incremental spqr-trees: Embeddings, planarity, and triconnectivity, 2019. arXiv.org/1910.09005.

[16] John E. Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

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

[18] Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM J. Discrete Math.*, 4(3):369–384, 1991.

[19] Giuseppe F. Italiano, Johannes A. La Poutré, and Monika Rauch. Fully dynamic planarity testing in planar embedded graphs (extended abstract). In *ESA '93, Bad Honnef*, pages 212–223, 1993.

[20] Philip Klein and Shay Mozes. Optimization algorithms for planar graphs. `http://planarity.org/`. Accessed: 2019-03-31.

[21] Saunders Mac Lane. A structural characterization of planar combinatorial graphs. *Duke Math. J.*, 3(3):460–472, 09 1937.

[22] Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *STOC '94, Montréal*, pages 706–715, 1994.

[23] Johannes A. La Poutré and Jeffery Westbrook. Dynamic 2-connectivity with backtracking. *SIAM J. Comput.*, 28(1):10–26, 1998.

[24] Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[25] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[26] Roberto Tamassia. On-line planar graph embedding. *J. Algorithms*, 21(2):201–239, 1996.

[27] Jeffery Westbrook. Fast incremental planarity testing. In *ICALP '92, Vienna*, pages 342–353, 1992.

[28] Hassler Whitney. 2-isomorphic graphs. *American Journal of Mathematics*, 1933.