


Decremental SPQR-trees for Planar Graphs

Jacob Holm¹

University of Copenhagen, Denmark


jaho@di.ku.dk

 <https://orcid.org/0000-0001-6997-9251>

Giuseppe F. Italiano²

University of Rome Tor Vergata, Italy


giuseppe.italiano@uniroma2.it

 <https://orcid.org/0000-0002-9492-9894>

Adam Karczmarz³

University of Warsaw, Poland


a.karczmarz@mimuw.edu.pl

 <https://orcid.org/0000-0002-2693-8713>

Jakub Łącki⁴

Google Research, USA


jłacki@google.com

 <https://orcid.org/0000-0001-9347-0041>

Eva Rotenberg

Technical University of Denmark, Denmark

erot@dtu.dk

 <https://orcid.org/0000-0001-5853-7909>

Abstract

We present a decremental data structure for maintaining the SPQR-tree of a planar graph subject to edge contractions and deletions. The update time, amortized over $\Omega(n)$ operations, is $O(\log^2 n)$. Via SPQR-trees, we give a decremental data structure for maintaining 3-vertex connectivity in planar graphs. It answers queries in $O(1)$ time and processes edge deletions and contractions in $O(\log^2 n)$ amortized time. The previous best supported deletions and insertions in $O(\sqrt{n})$ time.

2012 ACM Subject Classification Theory of computation \rightarrow Dynamic graph algorithms, Theory of computation \rightarrow Graph algorithms analysis, Theory of computation \rightarrow Data structures design and analysis

Keywords and phrases Graph embeddings, data structures, graph algorithms, planar graphs, SPQR-trees, triconnectivity

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.46

Related Version A full version of the paper is available at [27], <http://arxiv.org/abs/1806.10772>.

¹ Jacob Holm is supported by Mikkel Thorup's Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

² Giuseppe F. Italiano is partially supported by the Italian Ministry of Education, University and Research under Project AMANDA (Algorithmics for MAssive and Networked DATA).

³ Adam Karczmarz is supported by the grants 2014/13/B/ST6/01811 and 2017/24/T/ST6/00036 of the Polish National Science Center.

⁴ When working on this paper Jakub Łącki was partly supported by the EU FET project MULTIPLEX no. 317532 and the Google Focused Award on "Algorithms for Large-scale Data Analysis" and Polish National Science Center grant number 2014/13/B/ST6/01811. Part of this work was done while Jakub Łącki was visiting the Simons Institute for the Theory of Computing.



© Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Eva Rotenberg; licensed under Creative Commons License CC-BY

26th Annual European Symposium on Algorithms (ESA 2018).

Editors: Yossi Azar, Hannah Bast, and Grzegorz Herman; Article No. 46; pp. 46:1–46:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A graph algorithm is called *dynamic* if it is able to answer queries about a given property while the graph is undergoing a sequence of updates, such as edge insertions and deletions. It is *incremental* if it handles only insertions, *decremental* if it handles only deletions, and *fully dynamic* if it handles both insertions and deletions. In designing dynamic graph algorithms, one is typically interested in achieving fast query times (either constant or polylogarithmic), while minimizing the update times. The ultimate goal is to perform fast both queries and updates, i.e., to have both query and update times either constant or polylogarithmic. So far, the quest for obtaining polylogarithmic time algorithms has been successful only in few cases. Indeed, efficient dynamic algorithms with *polylogarithmic* time per update are known only for few problems, such as dynamic connectivity, 2-connectivity, minimum spanning tree and maximal matchings in undirected graphs (see, e.g., [6, 24, 25, 29, 37, 52, 54, 56]). On the other hand, some dynamic problems appear to be inherently harder. For example, the fastest known algorithms for basic dynamic problems, such as reachability, transitive closure, and dynamic shortest paths, have updates that run in only *polynomial* time (see, e.g., [9, 10, 11, 39, 49, 51, 55]).

A similar situation holds for planar graphs where dynamic problems have been studied extensively, see e.g. [3, 14, 16, 18, 20, 21, 26, 34, 42, 43, 44, 45, 53]. Despite this long-time effort, the best algorithms known for some basic problems on planar graphs, such as dynamic shortest paths and dynamic planarity testing, still have polynomial update time bounds. For instance, for fully dynamic shortest paths on planar graphs the best known bound per operation is $\tilde{O}(n^{2/3})$ amortized [19, 34, 36, 40] (using \tilde{O} -notation to hide polylogarithmic factors), while for fully dynamic planarity testing the best known bound per operation is $O(\sqrt{n})$ amortized [16].

In the last years, this exponential gap between polynomial and polylogarithmic bounds has sparked some new exciting research. On one hand, it was shown that there are dynamic graph problems, including fully dynamic shortest paths, fully dynamic single-source reachability and fully dynamic strong connectivity, for which it may be difficult to achieve subpolynomial update bounds. This started with the pioneering work by Abboud and Vassilevska-Williams [2], who proved conditional lower bounds based on popular conjectures. Very recently, Abboud and Dahlgaard [1] proved polynomial lower bounds for the update time for dynamic shortest paths also on planar graphs, again based on popular conjectures.

On the other hand, the question of improving the update bounds from polynomial to polylogarithmic, has, for several other dynamic graph problems, received much attention in the last years. For instance, there was a very recent improvement from polynomial to polylogarithmic bounds for decremental single-source reachability (and strongly connected components) on planar graphs: more precisely, the improvement was from $O(\sqrt{n})$ amortized [42] to $O(\log^2 n \log \log n)$ amortized [33] (both amortizations are over sequences of $\Omega(n)$ updates). Other problems that received a lot of attention are fully dynamic connectivity and minimum spanning tree in general graphs. Up to very recently, the best worst-case bound for both problems was $O(\sqrt{n})$ per update [15]: since then, much effort has been devoted towards improving this bound (see e.g., [37, 38, 47, 48, 57]).

In this paper, we follow the ambitious goal of achieving polylogarithmic update bounds for dynamic graph problems. In particular, we show how to improve the update times from polynomial to polylogarithmic for another important problem on planar graphs: decremental 3-vertex connectivity. Given a graph $G = (V, E)$ and two vertices $x, y \in V$ we say that x and

y are 2-vertex connected (or, as we say in the following, *biconnected*) if there are at least two vertex-disjoint paths between x and y in G . We say that x and y are 3-vertex connected (or, as we say in the following, *triconnected*) if there are at least three vertex-disjoint paths between x and y in G . The decremental planar triconnectivity problem consists of maintaining a planar graph G subject to an arbitrary sequence of edge deletions, edge contractions, and query operations which test whether two arbitrary input vertices are triconnected. We remark that decremental triconnectivity on planar graphs is of particular importance. Apart from being a fundamental graph property, a triconnected planar graph has only one planar embedding, a property which is heavily used in graph drawing, planarity testing and testing for isomorphism [31, 32, 35]. Furthermore, our extended repertoire of operations, which includes edge contractions, contains all operations needed to obtain a graph minor, which is another important notion for planar graphs.

While polylogarithmic update bounds for decremental 2-edge and 3-edge connectivity, and for decremental biconnectivity on planar graphs have been known for more than two decades [20], decremental triconnectivity on planar graphs presents some special challenges. Indeed, while connectivity cuts for 2-edge and 3-edge connectivity, and for biconnectivity have simple counterparts in the dual graph or in the vertex-face graph (see Section 2 for a formal definition of vertex-face graph), triconnectivity cuts (separation pairs, i.e., pairs of vertices whose removal disconnects the graph) have a much more complicated structure in planar graphs. Roughly speaking, maintaining 2-edge and 3-edge connectivity cuts in a planar graph under edge deletions corresponds to maintaining respectively self-loops and cycles of length 2 (pairs of parallel edges) in the dual graph under edge contractions. Similarly, maintaining biconnectivity and triconnectivity cuts in a planar graph under edge deletions corresponds to maintaining, respectively, cycles of length 2 and cycles of length 4 in the vertex-face graph. While detecting cycles of length 2 boils down to finding duplicates in the multiset of all edges, detecting cycles of length 4 under edge contractions is far more complex. We believe that this is the reason why designing a fast solution for decremental triconnectivity on planar graphs has been an elusive goal, and the best bound known of $O(\sqrt{n})$ per update [17] has been standing for over two decades.

Our results and techniques. Our main result is given in the following theorem.

► **Theorem 1.** *There is a data structure that can be initialized on a planar graph G on n vertices and $O(n)$ edges in $O(n \log n)$ time, and support any sequence of $\Omega(n)$ edge deletions or contractions in total time $O(n \log^2 n)$, while supporting queries to pairwise triconnectivity in worst-case constant time per query.*

This is an exponential speed-up over the previous $O(\sqrt{n})$ long-standing bound [17]. To obtain our bounds, we also need to solve decremental biconnectivity on planar graphs in constant time per query and $O(\log^2 n)$ amortized time per edge deletion or contraction. (A better $O(\log n)$ amortized bound can be obtained if no contractions are allowed [26].) In the description we assume that the graph is embedded in the plane (a so-called *plane* graph). However, the data structure may handle an arbitrary planar (non-embedded) graph by first embedding the initial graph in the plane in linear time. This choice of initial embedding has no effect on either the queries, or on which edge deletions or contractions are possible.

Our results are obtained using two new tools, which may be of independent interest. The first tool is an algorithm for efficiently detecting and reporting cycles of length 4 as they arise in a dynamic plane graph subject to edge contractions and insertions. The algorithm works for a graph with bounded face-degree, i.e, where each face is delimited by at most some

constant number of edges. Specifically, given a plane graph with bounded face-degree subject to edge-contractions and edge-insertions across a face, we can maintain the set of edges lying on cycles of length at most 4. The total running time is $O(n \log n)$. One of the challenges that we face is that a plane graph may have as many as $\Omega(n^2)$ distinct cycles of length 4. Still, we give a surprisingly simple algorithm for solving this problem. The difficulty of the algorithm lies in the analysis — in fact, this analysis is the most technically involved part of this paper.

The second tool is a new data structure that maintains the SPQR-tree [12] of each biconnected component of a planar graph subject to edge deletions and edge contractions, in $O(\log^2 n)$ amortized time per operation. While incremental algorithms for maintaining the SPQR-tree were known for more than two decades [12, 13], to the best of our knowledge no decremental algorithm was previously known.

Organization of the paper. The remainder of the paper is organized as follows. In Section 2, we introduce notation and definitions that we later use. Then, in Section 3 we present a high-level overview of our results. Finally, in Section 4 we give more details of our algorithm for maintaining an SPQR-tree under edge deletions and contractions.

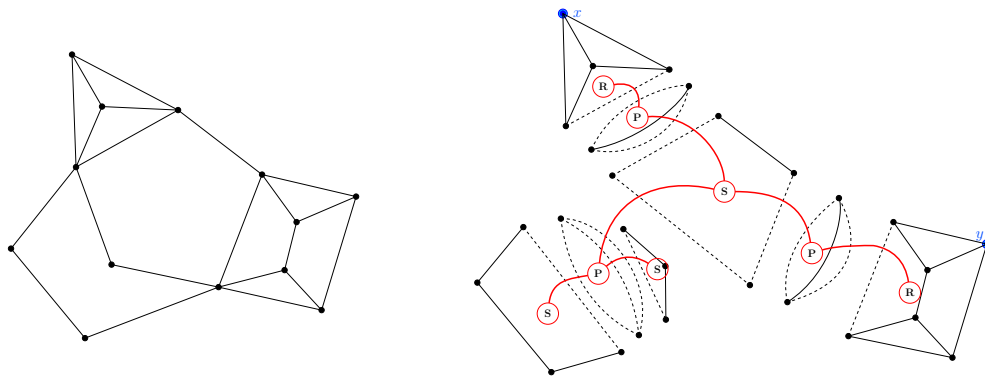
Due to space constraints, the algorithm for detecting cycles of length 4 under contractions, which is a key tool in maintaining an SPQR-tree, is deferred to the full version [27], along with the detailed discussion of how to use the SPQR-trees to maintain information about triconnectivity, and a selection of proofs omitted from Section 4.

2 Preliminaries

Throughout the paper we use the term *graph* to denote an undirected *multigraph*, that is, we allow the graphs to have parallel edges and self-loops. Formally, each edge e of such a graph is a pair $(\{u, w\}, \text{id}(e))$ consisting of a pair of vertices and a unique *integer identifier* used to distinguish between the parallel edges. For simplicity, in the following we skip the identifier and use just uw to denote one of the edges connecting vertices u and w . If the graph contains no parallel edges and no self-loops, we call it *simple*.

Given a graph G , we use $V(G)$ to denote the vertices, and $E(G)$ to denote the edges of G . For $e \in E(G)$, we use $G - e$ to denote the graph obtained from G by removing e . If e is not a self-loop, we use G/e to denote the graph obtained by contracting e . A *cycle* C of length $|C| = k$ in a graph G is a cyclic sequence of edges $C = e_1, e_2, \dots, e_k$ where $e_i = u_i u_{i+1}$ for $1 \leq i < k$ and $e_k = u_k u_1$. Note that this definition allows cycles of length 1 (a self-loop) or 2 (a pair of parallel edges). A cycle is *simple* if $\text{id}(e_i) \neq \text{id}(e_j)$ and $u_i \neq u_j$ for $i \neq j$. We sometimes abuse notation and treat a cycle as a set of edges or a cyclic sequence of vertices.

The *components* of a graph G are the minimal subgraphs $H \subseteq G$ such that for every edge $uv \in E(G)$, $u \in V(H)$ if and only if $v \in V(H)$. The components of a graph partition the vertices and edges of the graph. A graph G is *connected* if it consists of a single component. For a positive integer k , a graph is *k -vertex connected* if and only if it is connected, has at least k vertices, and stays connected after removing any set of at most $k - 1$ vertices. The *local vertex connectivity* of a pair of vertices u, v , denoted $\kappa(u, v)$, is the maximal number of internally vertex-disjoint u, v -paths. By Menger's Theorem [46], G is k -vertex connected if and only if $\kappa(u, v) \geq k$ for every pair of non-adjacent vertices u, v . We say that u, v are (locally) k -vertex connected if $\kappa(u, v) \geq k$. We follow the common practice of using *biconnected* as a synonym for 2-vertex connected and *triconnected* as a synonym for 3-vertex connected. An *articulation point* v of G is a vertex whose removal increases the number of



■ **Figure 1** A biconnected graph and its SPQR-tree. Note that adding the edge xy would collapse a path of SPQR-nodes into one. Deletion can thus result in the opposite transformation.

components of G . Thus, a graph is biconnected if and only if it is connected and has no articulation points.

The structure of the biconnected components of a connected graph can be described by a tree called the *block-cutpoint tree* [23, p. 36], or *BC-tree* for short. This tree has a vertex for each biconnected component (block) and for each articulation point of the graph, and an edge for each pair of a block and an articulation point that belongs to that block.

We recall that a graph G that is biconnected but not triconnected has at least one separation pair, i.e., a pair of vertices that can be removed to disconnect G :

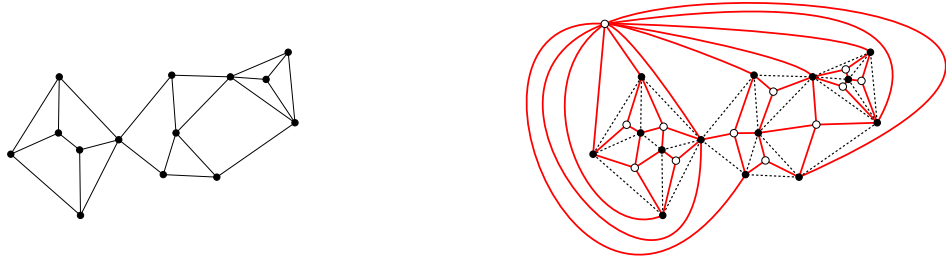
► **Definition 2** (Hopcroft and Tarjan [30, p. 6]). Let $\{a, b\}$ be a pair of vertices in a biconnected multigraph G . Suppose the edges of G are divided into equivalence classes E_1, E_2, \dots, E_k , such that two edges which lie on a common path not containing any vertex of $\{a, b\}$ except as 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 edge⁵.

The notion of the block cutpoint tree over biconnected components can be generalised to an *SPQR-tree* over triconnected components as follows:

► **Definition 3.** The 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 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.

⁵ These two exceptions actually make it easier to state some properties related to separation pairs.



■ **Figure 2** Left: a plane graph. Right: the corresponding vertex-face graph (red) and the underlying graph (dashed).

The SPQR-tree for a biconnected graph is unique (see e.g. [12]). The (skeleton graphs associated with) the SPQR-nodes are sometimes referred to as G 's triconnected components.

Let G be a plane graph (a planar graph embedded in the plane). For each component H of G , let H^* denote the dual graph of H , defined as the graph obtained by creating a vertex for each face in the embedding of H , and an edge e^* (called the *dual edge* of e), connecting the two (not necessarily distinct) faces that e is incident to. Let G^* denote the graph obtained from G by taking the dual of each component.

Each face f in a plane graph is bounded by a (not necessarily simple) cycle called the *face cycle* for f . We call the length of this cycle the *face-degree* of f . We call any other cycle a *separating cycle*.

Let G be a connected plane multigraph with at least one edge. Define the set $E^\diamond(G)$ of *corners*⁶ of G to be the set of ordered pairs of (not necessarily distinct) edges (e_1, e_2) such that e_1 immediately precedes e_2 in the clockwise order around some vertex, denoted $v(e_1, e_2)$. Note that if $(e_1, e_2) \in E^\diamond(G)$, then $(e_2^*, e_1^*) \in E^\diamond(G^*)$. We denote by G^\diamond the *vertex-face graph*⁷ of G (see Figure 2). This is a plane multigraph with vertex set $V(G) \cup V(G^*)$, and an edge between $v(e_1, e_2)$ and $v(e_2^*, e_1^*)$ for each corner $(e_1, e_2) \in E^\diamond(G)$. Abusing notation slightly, we can write G^\diamond as $(V(G) \cup V(G^*), E^\diamond(G))$. We use the following well-known facts about the vertex-face graph:

1. G^\diamond is bipartite and plane, with a natural embedding given by the embedding of G .
2. The vertex-face graphs of G and G^* are the same: $G^\diamond = (G^*)^\diamond$.
3. There is a one-to-one correspondence between the edges of G and the faces of G^\diamond (in the natural embedding, each face of G^\diamond contains exactly one edge of G interior, see Fig 2).
4. $(G^\diamond)^*$ (also known as the *medial graph*) is 4-regular.
5. G^\diamond is simple if and only if G is loopless and biconnected (See e.g. [8, Theorem 5(i)]).
6. G^\diamond is simple, triconnected and has no separating 4-cycles if and only if G is simple and triconnected (See e.g. [8, Theorem 5(iv)]).

If v is an articulation point in G or has a self-loop, then in any planar embedding of G there is at least one face f whose face cycle contains v at least twice. Any such f is either an articulation point or has a self-loop in G^* , and v and f are connected by (at least) two edges in G^\diamond .

The dynamic operations on G correspond to dynamic operations on G^* and G^\diamond . Deleting a non-bridge edge e of G corresponds to contracting e^* in G^* , that is, $(G - e)^* = G^*/e^*$. Similarly, contracting an edge e corresponds to deleting e^* from the dual, so $(G/e)^* = G^* - e^*$.

⁶ For alternative definitions, see e.g. [28] and [50]. The latter uses *angles* for what we call corners.

⁷ A.k.a. the *vertex-face incidence graph* [7], the *angle graph* [50], and the *radial graph* [5].

Finally, deleting a non-bridge edge or contracting an edge corresponds to adding and then immediately contracting an edge across a face of G° (and removing two duplicate edges).

Finally, the useful concept of a separation is well-defined, even for general graphs:

► **Definition 4.** Given a graph $G = (V, E)$, a *separation* of G is a pair of vertex sets (V', V'') such that the induced subgraphs $G' = G[V']$, $G'' = G[V'']$ contain all edges of G , and $V' \setminus V''$ and $V'' \setminus V'$ are both nonempty. A separation is *balanced* if $\max\{|V'|, |V''|\} \leq \alpha|V|$ for some fixed constant $\frac{1}{2} \leq \alpha < 1$. If (V', V'') is a separation of G , the set $S = V' \cap V''$ is called a *separator* of G . A separator S is *small* if $|S| = O(\sqrt{|V|})$, and it is a *cycle separator* if the subgraph of G induced by S is Hamiltonian.

Note that a *separation*, which is a pair of vertex sets, should not be confused with a *separation pair*, which is a pair of vertices (see Definition 2).

3 Overview of Our Approach

The SPQR-tree naturally reflects the triconnected components of the graph, so it is perhaps not surprising that an SPQR-tree can be augmented to answer pairwise triconnectivity queries in constant time. The challenge is to update the SPQR-tree under decremental updates. For this, we need a way to find all new separation pairs that arise. These separation pairs are related to separating 4-cycles in the vertex-face graph, in which decremental updates correspond to “collapsing” faces, i.e. the addition and immediate contraction of an edge across a face. So, the core of our approach is an algorithm for detecting separating 4-cycles in a particular kind of plane graph subject to valid edge insertions and contractions.

Detecting separating 4-cycles. A 4-cycle is a simple cycle of length 4. We say that a 4-cycle in a plane graph G is a *face 4-cycle* if it is a cycle bounding a face of G , and a *separating 4-cycle* otherwise. There is a one-to-one correspondence between separation pairs in G and separating 4-cycles in the vertex-face graph G° . (See [27] for details.)

Since no two parallel edges can lie on the same 4-cycle, and no self-loop can be contained in a 4-cycle, we can assume the input graph is simple. However, when we contract edges, new parallel edges and self-loops may arise. To handle this, we could detect and remove all parallel edges, but it turns out that both the algorithm and the analysis become simpler if we keep (most of) the additional edges, as long as no two parallel edges are consecutive in the circular ordering around both their endpoints. This is captured by the following definition.

► **Definition 5.** A plane graph is *quasi-simple* if the dual of each non-simple component has minimum degree 3. (In [41] these graphs are called *semi-strict*.)

Roughly speaking, a quasi-simple graph is obtained from a plane multigraph by merging parallel edges that lie next to each other in the circular orderings around both their endpoints.

We build a structure for 4-cycle detection by recursively using balanced separators, and by detecting, for each separator, the cycles that cross the separator. Detecting 4-cycles that cross a separator is not trivial, and our analysis introduces a complicated potential function which reflects how well connected the non-separator vertices are with the separator, that is, how many neighbors on the separator they have. At the same time, we make sure that all the work done can be paid with the decrease in the potential. Our analysis exploits the fact that for a subset of vertices S in quasi-simple planar graph, at most $O(|S|)$ vertices have 4 or more neighbors in S . Specifically, this holds when S is the set of separator vertices.

The recursive use of separators can be sketched as follows: Let S be a small balanced separator in $G = (V, E)$ that induces a separation (V_1, V_2) , that is, $V_1 \cap V_2 = S$ and

$V_1 \cup V_2 = V$. Moreover, let $n = |V|$. We observe that each 4-cycle is fully contained in V_1 or V_2 , or consists of two paths of length 2 that connect vertices of S . This motivates the following recursive approach. We compute a separator S of $O(\sqrt{n})$ vertices and then find all paths of length 2 that connect vertices of S . Since the size of S is $O(\sqrt{n})$, there are only $O(n)$ pairs of vertices of S , and for each pair of vertices, we can easily check if the two-edge paths connecting them form any separating 4-cycles. It then remains to find the 4-cycles that are fully contained in either V_1 or V_2 , which can be done recursively. Because S is a balanced separator, the recursion has $O(\log n)$ levels.

This algorithm can be made dynamic under contractions and edge insertions that respect the embedding of G . Contractions are easy to handle, as they preserve planarity. Moreover, a separator S of a planar graph can be easily updated under contractions. Namely, whenever an edge uw is contracted, the resulting vertex belongs to the separator iff any of u and w did. Insertions that preserve planarity, however, are in general harder to accommodate. To handle this we introduce a new type of separators that we call *face-preserving* separators, which (like cycle-separators) always exist when the face-degree is bounded. These are still preserved by contractions, but also ensure that any edge across a face can be inserted.

All in all, there are $O(\log n)$ levels of size $O(n)$ each, where each level handles insertions and contractions in constant time, leading to a total of $O(n \log n)$ time. (See [27] for details.)

► **Theorem 6.** *Let G be an n -vertex connected quasi-simple plane graph with bounded face degree. There exists a data structure that maintains G under contractions and embedding-respecting insertions, and after each update operation reports edges that become members of some separating 4-cycle. It runs in $O(n \log n)$ total time.*

Maintaining SPQR-trees. The main challenge in maintaining an SPQR-tree is handling the case when an edge within a triconnected component is deleted. First of all, the data structure should be able to detect whether or not the component is still triconnected.

For the skeleton Γ of any R -node in the SPQR-tree of G , we maintain a 4-cycle detection structure for the corresponding vertex-face graph Γ^\diamond . A separating 4-cycle in Γ^\diamond corresponds to a separation pair in Γ , which would witness that Γ is no longer triconnected. The deletion or contraction of the edge e in the triconnected component Γ of G corresponds to collapsing a face in Γ^\diamond by the insertion and immediate contraction of an edge. By detecting new 4-cycles in Γ^\diamond , we can therefore detect when the corresponding triconnected component falls apart.

However, this is not the only challenge. If Γ does indeed cease to be triconnected, the SPQR-tree of $(\Gamma - e)$ (or (Γ/e) when doing a contraction) is a path H . This is where we need the 4-cycle detection structure to output the edges contained in separating 4-cycles. Those edges correspond to a set of corners N of G . We use those corners to guide a search, which identifies the non-largest components of the SPQR-path H . More specifically, if a vertex v now belongs to two distinct triconnected components, there are two corners in N that separate the edges incident to v into two groups of edges, each belonging to a distinct triconnected component. We can afford to build a 4-cycle detection structure for Γ'^\diamond for any non-largest triconnected component Γ' on the path from scratch. To obtain the data structure representing the largest component, we delete or contract the edges of the smaller components from Γ , while updating Γ^\diamond . Since an edge only becomes part of a structure built from scratch when its triconnected component size has been halved, this happens only $O(\log n)$ times per edge. Since the time spent on building 4-cycle detection structures is $O(\log n)$ per contributing edge, the total time becomes $O(n \log^2 n)$.

Finally, since no two S -nodes can be neighbors and no two P -nodes can be neighbors, some S - or P -nodes in H may have to be merged with their (at most 2) neighbors of the

same type outside H . To handle this step efficiently, we keep the SPQR-tree rooted in an arbitrary node. While merging the skeleton graphs of two S - or P -nodes can be done in constant time, it is more costly to update the parent pointers in the children of the merged nodes. Hence, we move the children of the node with fewer children to the other node. This way, each node changes parent at most $O(\log n)$ times before it is deleted or split. The total number of distinct SPQR-nodes that exist throughout the lifetime of the data structure is $O(n)$, so the total time used for maintaining the parent pointers is $O(n \log n)$.

Since SPQR-trees are only defined for biconnected graphs, another challenge is to maintain SPQR-trees for each biconnected component, even as the decremental update operations cause the biconnected components to fall apart. We thus maintain also the BC-tree of the graph (see Section 2). If the BC-tree is rooted arbitrarily at any block, each non-root block has a unique articulation point separating it from its parent.

To handle updates, we notice that the SPQR-tree points to the fragile places where the graph is about to cease to be biconnected: An edge deletion in an S -node will break up a block in the BC-tree into a path, and an edge contraction in a P -node breaks a block in the BC-tree into a star. Upon such an update, we remove the aforementioned S - or P -node from the SPQR-tree, breaking it up into an SPQR-forest. Each tree corresponds to a new block in the BC-tree. They form a path (or a star), and the ordering along the path, as well as the articulation points, can be read directly from the SPQR-tree. (See Section 4 for details.)

On the other hand, in order to even know which SPQR-tree to modify during an update, we can search in the BC-tree for the right SPQR-structure in which to perform the operation.

Bi- and triconnectivity. Finally, we use SPQR-trees to facilitate triconnectivity queries. First of all, vertices need to be biconnected in order to be triconnected. In the rooted BC-tree, assign each vertex to its root-nearest block. It is enough that each vertex knows the name of its block, and each block knows the vertex separating it from its parent. Then, any two vertices are biconnected if and only if they either have the same block, or one is the unique vertex separating the block of the other from its parent.

For triconnectivity, the maintained information, as well as the query handling, is similar, using the SPQR-tree in place of the BC-tree. Namely: each non-root node in the SPQR-tree stores the *virtual edge* (see Definition 3) that separates it from its parent. Each vertex knows the root-nearest node containing it, and, if this is an S -node, its at most two children containing the vertex.

The main challenge is to handle updates. Note that the change to the SPQR-tree may involve both the split and merge of nodes. In particular, we have one split and up to several merges when a triconnected component falls apart into an SPQR-path. However, upon a merge, we can afford to update the information regarding vertices in the non-largest components, costing only an additive $\log n$ to the amortized running time. Similarly, upon a split, we update any information that relates to vertices in the non-largest components only.

The total running time is thus $O(n \log n + f(n))$, where $f(n)$ is the running time for maintaining the SPQR-tree. (See [27] for details.)

► **Theorem 1.** *There is a data structure that can be initialized on a planar graph G on n vertices and $O(n)$ edges in $O(n \log n)$ time, and support any sequence of $\Omega(n)$ edge deletions or contractions in total time $O(n \log^2 n)$, while supporting queries to pairwise triconnectivity in worst-case constant time per query.*

Algorithm 1 Removing an edge e from a P -node x of T .

```

1: function REMOVEP( $e, x, T$ )
2:   remove  $e$  from  $\Gamma(x)$ 
3:   if  $\Gamma(x)$  has two edges then
4:     if  $\Gamma(x)$  has no virtual edges then
5:       delete  $T$ 
6:     else if  $\Gamma(x)$  has one virtual edge then
7:        $y :=$  the only neighbor of  $x$ 
8:        $e_x :=$  the virtual edge in  $\Gamma(y)$  corresponding to  $x$ 
9:       replace  $e_x$  by the non-virtual edge of  $\Gamma(x)$ 
10:      remove  $x$  from  $T$ 
11:    else if  $\Gamma(x)$  has two virtual edges then
12:       $\{y, z\} :=$  neighbors of  $x$  in  $T$ 
13:      remove  $x$  from  $T$ , making  $y$  and  $z$  neighbors in  $T$ 
14:      if  $y$  and  $z$  are  $S$ -nodes then
15:        merge  $y$  and  $z$  into one node

```

4 Decremental SPQR-trees

In this section, we use the data structure of Theorem 6 to maintain an SPQR-tree (see Definition 3) for each biconnected component of G with at least 3 edges under arbitrary edge deletions and contractions. We start with some useful facts.

► **Lemma 7.** *Let G be a biconnected graph. If a 4-cycle $C = (v_1, f_1, v_2, f_2)$ in G^\diamond is a separating cycle, then v_1, v_2 is a separation pair of G and f_1, f_2 is a separation pair of G^* .*

► **Lemma 8.** *Let G be a loopless biconnected plane graph and u, w be a separation pair in G . Consider the set of edges E_x incident to $x \in \{u, w\}$. Then, the edges of E_x belonging to each separation class of u, w are consecutive in the circular ordering around both u and w .*

► **Lemma 9.** *Let G be a triconnected plane graph and $e = uw \in E(G)$. Assume that $G - e$ is not triconnected. Then, the SPQR-tree of $G - e$ is a path H (we call it an SPQR-path). Moreover, given all edges that lie on 4-cycles in $(G - e)^\diamond$, we can compute all nodes of H (i.e., their skeleton graphs) except for the largest one in time that is linear in their size.*

For a planar graph, there is a nice duality, as proven by Angelini et al. [4, Lemma 1]. Define the dual SPQR-tree as the tree obtained from the SPQR-tree by interchanging S - and P -nodes, and taking the dual of the skeletons.

► **Lemma 10** (Angelini et al [4]). *The SPQR-tree of G^* is the dual SPQR-tree of G .*

Let G be a connected plane graph. Since $(G^\diamond)^*$ is 4-regular, G^\diamond is quasi-simple and has bounded face-degree. Furthermore, any edge deletion or contraction in G that leaves G connected, corresponds to an edge insertion and immediate contraction in G^\diamond . Thus by Theorem 6 we can maintain a data structure for G under connectivity-preserving edge deletions and contractions, that after each update operation reports the corners that become part of a separating 4-cycle in G^\diamond .

In the algorithm we maintain one SPQR-tree for each biconnected component with at least 3 edges. We now describe how these trees are updated upon edge deletions. The procedures, depending on the type of the SPQR-tree node are given as Algorithms 1, 2 and 3. Note that the lines 4 and 5 in Algorithm 2 only introduce notation, that is the values of the variables are not computed. (See [27] for a proof of correctness.)

Algorithm 2 Removing an edge e from an R-node x of T .

```

1: function REMOVE( $e, x, T$ )
2:   remove  $e$  from  $\Gamma(x)$ 
3:   if  $\Gamma(x)$  has a separation pair then
4:      $X' :=$  SPQR-path representing  $\Gamma(x)$ 
5:      $x_{big} :=$  the node of  $X'$  st.  $\Gamma(x_{big})$ 
       has the most edges
6:     compute all nodes of  $X' \setminus x_{big}$ 
7:     remove and contract edges of  $\Gamma(x)$ 
       to obtain  $\Gamma(x_{big})$ 
8:     replace  $x$  in  $T$  by  $X'$  (connect
       each child of  $x$  to the correct
       node of  $X'$ )
9:     for each  $S$ - or  $P$ -node  $z \in X'$  do
10:      for each neighbor  $z' \notin X'$  do
11:        if  $z, z'$  are same type then
12:          merge  $z$  with  $z'$ 

```

Algorithm 3 Removing an edge e from an S -node x of T .

```

1: function REMOVES( $e, x, T$ )
2:   remove  $e$  from  $\Gamma(x)$ 
3:   remove  $x$  from  $T$ 
4:   for each edge  $e'$  in  $\Gamma(x)$  do
5:     Make a new BC-node  $z$ 
6:     if  $e'$  is a virtual edge then
7:        $y :=$  neighbor of  $x$  in  $T$  corre-
       sponding to  $e'$ 
8:       Make the tree containing  $y$  the
       SPQR-tree for the new BC-node
9:       if  $y$  is a  $P$ -node then
10:        removeP( $y, e', T$ )
11:       else
12:        removeR( $y, e', T$ )

```

We can now prove the main theorem of this section. Note that, as in the block-cutpoint tree, we root each SPQR-tree in an arbitrary vertex.

► **Theorem 11.** *There is a data structure that can be initialized on a simple planar graph G on n vertices in $O(n \log n)$ time, and supports any sequence of edge deletions or contractions in total time $O(n \log^2 n)$, while maintaining an explicit representation of a rooted SPQR-tree for each biconnected component with at least 3 edges, including all the skeleton graphs for the triconnected components. Moreover, during updates, the total number of times a node of an SPQR-tree changes its parent is $O(n \log n)$.*

Proof. We first partition the graph into biconnected components, and, as sketched in Section 3, maintain the block-cutpoint tree explicitly. Thus, given two vertices u, v , we can in $O(1)$ time access the biconnected component containing both of them, along with its auxiliary data. Now, for each biconnected component C_i , we compute the SPQR-tree T . This can be done in linear time due to [22]. We also root each SPQR-tree in an arbitrary node, and keep the trees rooted as they are updated.

For each node x of T we maintain the graph $\Gamma(x)$. Each virtual edge of $\Gamma(x)$ has a pointer to the neighbor of x it represents. Moreover, for each R-node r , we keep a data structure of Theorem 6 for detecting separating 4-cycles in the vertex-face graph $(\Gamma(r))^\diamond$. By Lemma 7, any separating 4-cycle in $(\Gamma(r))^\diamond$ corresponds to a separation pair in $\Gamma(r)$. Since r is an R-node, there are no separating 4-cycles to begin with, but some may appear after an update.

Since the total size of the R-components is n , it follows from Theorem 6 that the entire construction time is $O(n \log n)$.

Deletion. When an edge e is removed we find the node x of the SPQR-tree, such that e is a non-virtual edge in x . Then, we proceed according to Algorithms 1, 2 and 3.

Whenever an edge fg is deleted from an R-node r , we update the corresponding 4-cycle detection structure for $(\Gamma(r))^\diamond$. We first insert the dual edge $(fg)^*$ in the vertex-face graph, and then contract along that edge. This allows us to detect whether $\Gamma(r)$ has any separation pairs after each edge deletion.

Let us now analyze the running time. When processing an edge deletion, the following changes can take place in a SPQR-tree (all other changes can be handled in $O(1)$ time):

- an R -node is split into multiple nodes,
- two P -nodes or S -nodes are merged,
- an S - or P - node is deleted.

Note, a P - or S -node can never get split. So, though each edge may at first belong to nodes that are split, once it becomes a part of a P - or S -node, its node only participates in merges.

When two S - or P -nodes are merged, we can merge their skeleton graphs in constant time. These skeleton graphs have only two common nodes, and their lists of adjacent edges can be merged in constant time thanks to Lemma 8. When nodes are merged, we also have to update the parent pointers of their children. To bound the number of these updates, we merge the node with fewer children into the node with more. Thus, the number of parent updates caused by these merges is $O(n \log n)$, and so is the impact on the running time.

A similar analysis applies to the case when an R -node r is split into an SPQR-path. By Lemma 9, we can compute all but the largest node of the SPQR-path in linear time. Since the size of the skeleton graph in each of these nodes is at most half the size of $\Gamma(r)$, each edge takes part in this computation at most $O(\log n)$ times. For every new R -nodes, we also initialize their associated data structures for detecting 4-cycles. We charge the running time of each data structure to this initialization. From Theorem 6 we get that recomputing all the nodes and data structures takes $O(n \log^2 n)$ total time.

Taking care of the largest component of the SPQR-path is even easier, as we can simply reuse the skeleton graph of r and its associated data structure for detecting 4-cycles. To update the skeleton graph, we use the following lemma.

► **Lemma 12.** *If G is triconnected, $e \in E(G)$, and x is an R -node in the SPQR-tree for $G - e$, then there exists a sequence of $|E(G)| - |E(\Gamma(x))|$ edge deletions and contractions that transform $G - e$ into $\Gamma(x)$ while keeping the graph connected at all times.*

After an R -node r is split into a SPQR-path H we also need to update the parent pointers in the children of r . However, the number of children to update is at most the number of edges in the non-largest components of the SPQR-path. As we have argued, the total number of such edges across all deletions is $O(n \log n)$.

Contraction. The contraction of an edge of the plane graph G corresponds to the deletion of an edge of its dual graph, G^* . By Lemma 10, the SPQR-tree of G^* is the dual SPQR-tree of G . Thus, if the edge was in a P -node of the SPQR-tree, its contraction is handled like the deletion of an edge in a S -node, and vice versa.

If the contracted edge e belongs to an R -node, that R node may expand to a path in the SPQR-tree (because deletion in G^* may expand an R -node into a path). In the vertex-face graph, we may find all edges participating in new separating 4-cycles, corresponding to separating corners of the graph. To find the new components, we simply apply Lemma 9 to the dual graph and proceed analogously to a deletion. ◀

References

- 1 Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 477–486, 2016. doi:10.1109/FOCS.2016.58.

- 2 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.
- 3 Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1199–1218, 2012. doi:10.1145/2213977.2214084.
- 4 Patrizio Angelini, Thomas Bläsius, and Ignaz Rutter. Testing mutual duality of planar graphs. *Int. J. Comput. Geometry Appl.*, 24(4):325–346, 2014. doi:10.1142/S0218195914600103.
- 5 Dan Archdeacon and R Bruce Richter. The construction and classification of self-dual spherical polyhedra. *J. Comb. Theory, Series B*, 54(1):37–63, 1992. doi:10.1016/0095-8956(92)90065-6.
- 6 Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015. doi:10.1137/130914140.
- 7 Graham R. Brightwell and Edward R. Scheinerman. Representations of planar graphs. *SIAM J. Discrete Math.*, 6(2):214–229, 1993. doi:10.1137/0406017.
- 8 Gunnar Brinkmann, Sam Greenberg, Catherine Greenhill, Brendan D. McKay, Robin Thomas, and Paul Wollan. Generation of simple quadrangulations of the sphere. *Discrete Math.*, 305(1-3):33–54, 2005. doi:10.1016/j.disc.2005.10.005.
- 9 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Łącki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{O}(m\sqrt{n})$ total update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 315–324, 2016. doi:10.1109/FOCS.2016.42.
- 10 Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004. doi:10.1145/1039488.1039492.
- 11 Camil Demetrescu and Giuseppe F. Italiano. Maintaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008. doi:10.1007/s00453-007-9051-4.
- 12 Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996. doi:10.1007/BF01961541.
- 13 Giuseppe Di Battista and Roberto Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997, 1996. doi:10.1137/S0097539794280736.
- 14 Krzysztof Diks and Piotr Sankowski. Dynamic plane transitive closure. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 594–604, 2007. doi:10.1007/978-3-540-75520-3_53.
- 15 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914.
- 16 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification I: Planarity testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996. doi:10.1006/jcss.1996.0002.
- 17 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator-based sparsification II: Edge and vertex connectivity. *SIAM J. Comput.*, 28(1):341–381, 1998. Announced at STOC '93. doi:10.1137/S0097539794269072.
- 18 David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13(1):33–54, 1992. doi:10.1016/0196-6774(92)90004-V.

- 19 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006. doi:10.1016/j.jcss.2005.05.007.
- 20 Dora Giammarresi and Giuseppe F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 16(3):263–287, 1996. Announced at SWAT 1992. doi:10.1007/BF01955676.
- 21 Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theor. Comput. Sci.*, 203(1):123–141, 1998. doi:10.1016/S0304-3975(97)00291-0.
- 22 Carsten Gutwenger and Petra Mutzel. *A Linear Time Implementation of SPQR-Trees*, pages 77–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi:10.1007/3-540-44541-2_8.
- 23 Frank Harary. *Graph Theory*. Addison-Wesley Series in Mathematics. Addison Wesley, 1969.
- 24 Monika Rauch Henzinger and Mikkel Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997. doi:10.1002/(SICI)1098-2418(199712)11:4<369::AID-RSA5>3.0.CO;2-X.
- 25 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.
- 26 Jacob Holm, Giuseppe F Italiano, Adam Karczmarz, Jakub Lacki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 87. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- 27 Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Eva Rotenberg. Decremental SPQR-trees for Planar Graphs. *ArXiv e-prints*, 2018. arXiv:1806.10772.
- 28 Jacob Holm and Eva Rotenberg. Dynamic planar embeddings of dynamic graphs. *Theory of Computing Systems*, Apr 2017. doi:10.1007/s00224-017-9768-7.
- 29 Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, Sept. 14-16, 2015, Proceedings*, pages 742–753, 2015. doi:10.1007/978-3-662-48350-3_62.
- 30 John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973. doi:10.1137/0202012.
- 31 John E. Hopcroft and Robert Endre Tarjan. A $V \log V$ algorithm for isomorphism of triconnected planar graphs. *J. Comput. Syst. Sci.*, 7(3):323–331, 1973. doi:10.1016/S0022-0000(73)80013-3.
- 32 John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 172–184, 1974. doi:10.1145/800119.803896.
- 33 Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Piotr Sankowski. Decremental single-source reachability in planar digraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1108–1121, 2017. doi:10.1145/3055399.3055480.
- 34 Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 313–322, 2011. doi:10.1145/1993636.1993679.
- 35 Goossen Kant. Algorithms for drawing planar graphs, 2001.

- 36 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 338–355, 2012. URL: <http://portal.acm.org/citation.cfm?id=2095147&CFID=63838676&CFTOKEN=79617016>.
- 37 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142, 2013. doi:10.1137/1.9781611973105.81.
- 38 Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster worst case deterministic dynamic connectivity. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 53:1–53:15, 2016. doi:10.4230/LIPIcs.ESA.2016.53.
- 39 Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91, 1999. doi:10.1109/SFFCS.1999.814580.
- 40 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, BC, Canada, January 23-25, 2005*, pages 146–155, 2005. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070454>.
- 41 Philip N. Klein and Shay Mozes. Optimization algorithms for planar graphs, 2017. URL: <http://planarity.org>.
- 42 Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013. doi:10.1145/2483699.2483707.
- 43 Jakub Łącki, Jakub Oćwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the Steiner tree. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 11–20, 2015. doi:10.1145/2746539.2746615.
- 44 Jakub Łącki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 155–166, 2011. doi:10.1007/978-3-642-23719-5_14.
- 45 Jakub Łącki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, pages 608–621, 2015. doi:10.4230/LIPIcs.STACS.2015.608.
- 46 Karl Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–115, 1927.
- 47 Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2 - \epsilon})$ -time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129, 2017. doi:10.1145/3055399.3055447.
- 48 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Proceedings of the 58th Annual Symposium on Foundations of Computer Science, FOCS 2017*, 2017. To appear.
- 49 Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008. doi:10.1137/060650271.

- 50 Pierre Rosenstiehl. Embedding in the plane with orientation constraints: The angle graph. *Annals of the New York Academy of Sciences*, 555(1):340–346, 1989. doi:10.1111/j.1749-6632.1989.tb22470.x.
- 51 Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science FOCS 2004, 17-19 October 2004, Rome, Italy, Proceedings*, pages 509–517, 2004. doi:10.1109/FOCS.2004.25.
- 52 Shay Solomon. Fully dynamic maximal matching in constant update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 325–334, 2016. doi:10.1109/FOCS.2016.43.
- 53 Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings*, pages 372–383, 1993. doi:10.1007/3-540-57273-2_72.
- 54 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 343–350, 2000. doi:10.1145/335305.335345.
- 55 Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 112–119, 2005. doi:10.1145/1060590.1060607.
- 56 Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1757–1769, 2013. doi:10.1137/1.9781611973105.126.
- 57 Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143, 2017. doi:10.1145/3055399.3055415.