Dynamic Planar Embedding Is in DynFO

Samir Datta
Chennai Mathematical Institute & UMI ReLaX, Chennai, India

Asif Khan
Chennai Mathematical Institute, India

Anish Mukherjee
University of Warwick, Coventry, UK

Abstract

Planar Embedding is a drawing of a graph on the plane such that the edges do not intersect each other except at the vertices. We know that testing the planarity of a graph and computing its embedding (if it exists), can efficiently be computed, both sequentially [24] and in parallel [33], when the entire graph is presented as input.

In the dynamic setting, the input graph changes one edge at a time through insertion and deletions and planarity testing/embedding has to be updated after every change. By storing auxilliary information we can improve the complexity of dynamic planarity testing/embedding over the obvious recomputation from scratch. In the sequential dynamic setting, there has been a series of works [17, 25, 20, 22], culminating in the breakthrough result of polylog(n) sequential time (amortized) planarity testing algorithm of Holm and Rotenberg [21].

In this paper we study planar embedding through the lens of DynFO, a parallel dynamic complexity class introduced by Patnaik et al [31] (also [16]). We show that it is possible to dynamically maintain whether an edge can be inserted to a planar graph without causing non-planarity in DynFO. We extend this to show how to maintain an embedding of a planar graph under both edge insertions and deletions, while rejecting edge insertions that violate planarity.

Our main idea is to maintain embeddings of only the triconnected components and a special two-colouring of separating pairs that enables us to side-step cascading flips when embedding of a biconnected planar graph changes, a major issue for sequential dynamic algorithms [22, 21].

2012 ACM Subject Classification Theory of computation → Complexity theory and logic; Theory of computation → Finite Model Theory

Keywords and phrases Dynamic Complexity, Planar graphs, Planar embedding

Digital Object Identifier 10.4230/LIPIcs.MFCS.2023.39

Related Version *Full Version*: `https://arxiv.org/abs/2307.09473`

Funding *Samir Datta*: Partially funded by a grant from Infosys foundation.
Asif Khan: Partially funded by a grant from Infosys foundation.
Anish Mukherjee: Research supported in part by the Centre for Discrete Mathematics and its Applications (DIMAP), by EPSRC award EP/V01305X/1.

Acknowledgements Thanks to Nils Vortmeier and Thomas Zeume for illuminating discussions.

1 Introduction

Planar graphs are graphs for which there exists an embedding of vertices on the plane such that the edges can be drawn without intersecting with each other, except at their endpoints. The notion of planar graphs is fundamental to graph theory as underlined by the Kuratowski theorem [26]. The planarity testing problem is to determine if the encoded graph is planar and the planar embedding problem is to construct such an embedding. These are equally fundamental questions to computer science and their importance has been recognized from the early 1970s in the linear time algorithm by Hopcroft and Tarjan [24]. Since then there have been a plethora of algorithmic solutions presented for the planarity testing and embedding problems such as [28, 4, 18] that culminated in an alternative linear-time algorithm [18], a work efficient parallel algorithm running in $O(\log n)$ time [33], a deterministic logspace algorithm [1, 13], and many more.

All of the above algorithms are static i.e., the input is presented at once and we need to answer the planarity testing query and produce an embedding only once. However, in many real-life scenarios, the input is itself dynamic and evolves by insertion and deletion of edges. The same query can be asked at any instance and even the embedding may be required. Rather than recomputing the result from scratch after every update to the input in many scenarios, it is advantageous to preserve some auxiliary data such that each testing or embedding query can be answered much faster than recomputation from scratch. The notion of "fast" can be quantified via the sequential time required to handle the updates and queries which can be achieved in polylogarithmic time as in the recent breakthrough works of [22, 21]. These in turn built upon the previous work that dealt with only a partially dynamic model of computation – insertion only [32, 3, 34] or, deletion only [25] or the fully dynamic model (that supports both insertions and deletions) but with polynomial time updates [17].

Our metric for evaluating updates is somewhat different and determined according to the Dynamic Complexity framework of Immerman and Patnaik [31], and is closely related to the setting of Dong, Su, and Topor [16]. In it, a dynamic problem is characterised by the smallest complexity class in which it is possible to place the updates to the auxiliary database and still be able to answer the queries (notice that if the number of possible queries is polynomial we can just maintain the answers of all queries in the auxiliary database).

Notable amongst these has been the first-order logic formulas or equivalently, the descriptive complexity class FO. Thus we obtain the class DynFO of dynamic problems for which the updates to the auxiliary data structure are in FO given the input structure and stored auxiliary data structures. The motivation to use first-order logic as the update method has connections to other areas as well e.g., it implies that such queries are highly parallelisable, i.e., can be updated by polynomial-size circuits in constant-time due to the correspondence between FO and uniform AC^0 circuits [2]. From the perspective of database theory, such a program can be translated into equivalent SQL queries.

A particular recent success story of dynamic complexity has been that directed graph reachability (which is provably not in FO) can be maintained in DynFO [8] resolving a conjecture from Immerman and Patnaik [31], open since the inception of the field. Since then, progress has been made in terms of the size of batch updates (i.e., multiple simultaneous insertions and deletions) that can be handled for reachability, distance, and maximum matching [12, 30]. Later, improved bounds have been achieved for these problems in various special graph classes, including in planar graphs [9, 5]. Problems in planar graphs have been studied in the area of dynamic complexity starting much earlier e.g., before the reachability conjecture was resolved, it was shown in [6] that reachability in embedded planar graphs is in DynFO. Also, in [29] it was shown that 3-connected planar graph isomorphism too is in DynFO with some precomputation. However, despite these works the dynamic complexity of the planarity testing problem itself is not yet resolved, let alone maintaining a planar embedding efficiently.

Our contribution. In this paper, we build on past work in dynamic complexity to show that a planar embedding can be maintained efficiently, where we test for planarity at every step. Here, by planar embedding we mean a cyclic order on the neighbours of every vertex in some drawing of the graph on the plane (also known as combinatorial embedding [15]).

▶ **Theorem 1.** *Given a dynamic graph undergoing insertion and deletion of edges we can maintain a planar embedding of the graph in* DynFO *(while never allowing insertion of edges that cause the graph to become non-planar).*

Organization. We start with preliminaries concerning graph theory and dynamic complexity in Section 2. We present a technical overview of our work in Section 3. In Section 4 we develop the graph theoretic machinery we need for our algorithm. Next, we describe the dynamic planar embedding algorithm in Section 5. For the implementation details please refer to the full version of this paper [7].

2 Preliminaries

We start with some notations followed by graph theoretic preliminaries related to connectivity and planarity – see [14, Chapters 3, 4] for a thorough introduction. Then we reproduce some essentials of Dynamic Complexity from [9, 12].

Given a graph $G = (V, E)$, we write $V(G)$ and $E(G)$ to denote the sets of vertices and edges of G, respectively. For a set of edges $S \subseteq E(G)$ we denote by $G - S$, the graph with the edges in S deleted. Similarly for $S \subseteq V(G) \times V(G)$ we denote by $G + S$ the graph to which new edges in S have been added. For a set of vertices $T \subseteq V(G)$, by $G - T$ we refer to the induced graph $G[V(G) \setminus T]$. An undirected path between u and v is denoted by $u \leftrightsquigarrow v$.

Biconnected and Triconnected Decomposition. We assume familiarity with common connectivity related terminology including 2-vertex connectivity, 3-vertex connectivity and the related separating sets viz. cut vertices, separating pairs and the notion of virtual edges in the triconnected decomposition [23]. Please see the definitions of these terms in the full version [7, Section 2.1]. The following are two data structures that help in representing tree decompositions associated with biconnectivity and 3-connectivity respectively.

1. **BC-tree** or block-cut tree of a connected component of the graph, say H, denoted by $T_2(H)$. The nodes of the tree are the biconnected components (block nodes) and the cut vertices (cut nodess) of H and the edges are only between cut and block nodes. Block nodes are denoted by B and the cut nodes are denoted by C.
2. **SPQR-tree** or the triconnected decomposition tree of a biconnected component of the graph say B, is denoted by $T_3(B)$. The nodes in the SPQR-tree are of one of four types: S denotes a cycle component (serial node), P denotes a 3-connected separating pair (parallel node), Q denotes that there is just a single edge in B, and R denotes the 3-connected components or the so-called rigid nodes. There is an edge between an R-node, say R_i and a P-node, say P_j if $V(P_j) \subset V(R_i)$, and similarly, edges between S and P-nodes are defined.

We will conflate a node in one of the two trees with the corresponding subgraph. For example, an R-node interchangeably refers to the tree node as well the associated rigid subgraph.

Planar Embedding. A planar embedding of a graph $G = (V, E)$ is a mapping of vertices and edges in the plane $\mathbb{R}^2$ such that the vertices are mapped to distinct points in the plane and every edge is mapped to an arc between the points corresponding to the two vertices incident on it such that no two arcs have any point in common except at their endpoints. This embedding is called a *topological embedding*. Corresponding to a given topological embedding, the faces of the graph are the open regions in $\mathbb{R}^2 \setminus G$ (plane with points corresponding to the vertices and edges removed), call the set of faces as F. For a face $f \in F$, the set of all the vertices that lie on the boundary of f, is denoted by $V(f)$. The unbounded face is called the outer face. An embedding on the surface of a sphere is similarly defined. On the sphere, every face is bounded. Two topological embeddings are equivalent if, for every vertex, the cyclic order of its neighbours around the vertex is the same in both embeddings. So, the cyclic order (or rotation scheme) around each vertex defines an equivalence on the topological embeddings. The vertex rotation scheme around each vertex encodes the embedding equivalence class (combinatorial embedding). We now recall two important results. The first one is due to Whitney [35] that says that a 3-connected planar graph has unique planar embedding on the sphere (up to reflection). The second one is due to Mac Lane [27] that shows a biconnected graph is planar iff its triconnected components are planar.

2.1 Dynamic Complexity

The goal of a dynamic program is to answer a given query on an *input structure* subject to changes that insert or delete tuples. The program may use an auxiliary data structure represented by an *auxiliary structure* over the same domain. Initially, both input and auxiliary structure are empty; and the domain is fixed during each run of the program.

For a (relational) structure $\mathcal{I}$ over domain D and schema σ, a change $\Delta\mathcal{I}$ consists of sets R^+ and R^- of tuples for each relation symbol $R \in \sigma$. The result $\mathcal{I} + \Delta\mathcal{I}$ of an application of the change $\Delta\mathcal{I}$ to $\mathcal{I}$ is the input structure where $R^{\mathcal{I}}$ is changed to $(R^{\mathcal{I}} \cup R^+) \setminus R^-$. The *size* of $\Delta\mathcal{I}$ is the total number of tuples in relations R^+ and R^- and the set of *affected elements* is the (active) domain of tuples in $\Delta\mathcal{I}$.

Dynamic Programs and Maintenance of Queries. A dynamic program consists of a set of update rules that specify how auxiliary relations are updated after changing the input structure. An *update rule* for updating an ℓ-ary auxiliary relation T after a change is a first-order formula φ over schema $\tau \cup \tau_{\text{aux}}$ with ℓ free variables, where τ_{aux} is the schema of the auxiliary structure. After a change $\Delta\mathcal{I}$, the new version of T is $T := \{\bar{a} \mid (\mathcal{I}+\Delta\mathcal{I}, \mathcal{A}) \models \varphi(\bar{a})\}$ where $\mathcal{I}$ is the old input structure and $\mathcal{A}$ is the current auxiliary structure. Note that a dynamic program can choose to have access to the old input structure by storing it in its auxiliary relations.

For a state $\mathcal{S} = (\mathcal{I}, \mathcal{A})$ of the dynamic program $\mathcal{P}$ with input structure $\mathcal{I}$ and auxiliary structure $\mathcal{A}$, we denote by $\mathcal{P}_\alpha(\mathcal{S})$, the state of the program after applying a change sequence α and updating the auxiliary relations accordingly. The dynamic program *maintains* a q-ary query Q under changes that affect k elements (under changes of size k, respectively) if it has a q-ary auxiliary relation *Ans* that at each point stores the result of Q applied to the current input structure. More precisely, for each non-empty sequence α of changes that affect k elements (changes of size k, respectively), the relation *Ans* in $\mathcal{P}_\alpha(\mathcal{S}_\emptyset)$ and $Q(\alpha(\mathcal{I}_\emptyset))$ coincide, where $\mathcal{I}_\emptyset$ is an empty input structure, $\mathcal{S}_\emptyset$ is the auxiliary structure with empty auxiliary relations over the domain of $\mathcal{I}_\emptyset$, and $\alpha(\mathcal{I}_\emptyset)$ is the input structure after applying α. If a dynamic program maintains a query, we say that the query is in DynFO.

3 Technical Overview

It is well known from Whitney's theorem [35] that 3-connected planar graphs are rigid i.e., they (essentially) have a unique embedding. Thus, for example, under the promise that the graph remains 3-connected and planar it is easy to maintain an embedding in DynFO (see for example [29]). An edge insertion occurs within a face and there are only local changes to the embedding – restricted to a face. Deletions are exactly the reverse.

On the other extreme are trees, which are minimally connected. These are easy to maintain as well because any vertex rotation scheme is realisable. However, biconnected components are not rigid and yet not every rotation scheme for a vertex is valid (see Figure 3a for an illustration). The real challenge is in maintaining embeddings of biconnected components.

This has been dealt with in literature by decomposing biconnected graphs into 3-connected components (which are rigid components in the context of planar graphs). The 3-connected components are organized into trees[1] popularly called SPQR-trees [3]. The approach is to use the rigidity of the 3-connected planar components and the flexibility of trees to maintain a planar embedding of biconnected graphs. In order to maintain a planar embedding of connected graphs we need a further tree decomposition into biconnected components that yields the so-called block-cut trees or BC-trees ([14, Lemma 3.1.4], [19]). Notice that the tree decomposition into SPQR-trees and BC-trees is Logspace hard [10] and hence not in FO. Thus, in the parallel dynamic setting, we emulate previous sequential dynamic algorithms in *maintaining* (rather than computing from scratch) SPQR-trees and BC-trees in our algorithm.

Issues with biconnected embedding. The basic problem with maintaining biconnected planar components is their lack of rigidity (with reaching complete flexibility). Thus insertion of an edge into a biconnected component might necessitate changing the embedding through operations called *flips* and *slides* in literature [22, 21] (see [7, Figure 4]). We might need lots of flips and slides for a single edge insertion – causing an exponentially large search space. We now proceed to describe these changes in more detail.

In the simplest form consider a biconnected graph and a separating pair contained within, separating the component into two 3-connected components. We can reflect one of the 3-connected components, that is, the vertex rotation for each vertex in the 3-connected component is reversed. More intuitively, mirror a piece across one of its separating pairs. In more complicated cases there may be cascading flips i.e., reflections across multiple separating pairs in the biconnected component. We also need to deal with *slides*, that is changes in ordering of the biconnected components at a separating pair. A single edge insertion might need multiple flips and slides.

This induces the definition of flip-distance i.e., the minimum number of flips and slides to change one embedding of the graph to another. Intuitively, the flip-distance lower bounds the sequential time needed to transition from one embedding to another.

Thus a crucial part of previous algorithms [22] deals with maintaining an embedding of small flip distance with every possible embedding that can arise after a single change. In [21] this algorithm is converted to a fully dynamic algorithm that handles updates in $O(\log^3 n)$ time using a sophisticated amortization over the number of flips required to transition to an appropriate embedding dominates the running time of their algorithm. Notice that [22] handles changes in $O(\log^3 n)$ worst case time but only in the incremental setting.

Our approach for dynamic planarity testing. We now switch to motivating our approach in the parallel dynamic setting that is fully dynamic and does not use amortization. There are fundamentally two issues to be resolved while inserting an edge – one is whether the resulting graph is planar. The other is, how to update the embedding, possibly by performing multiple flips, when the graph remains planar. Let us first focus on biconnected graphs and the corresponding tree decomposition SPQR-tree introduced in Section 2.

[1] The tree decomposition of a biconnected graph into 3-connected pieces is a usual tree decomposition ([14, Chapter 12.3]).

To check for planarity on insertion of an edge, we introduce the notion of P_i, P_j-*coherent paths*. A path between two P-nodes P_i, P_j in an SPQR-tree is said to be coherent if for every R-node R_k on the path, all the (at most four) vertices of the two adjacent P-nodes are all on one face of R_k (see Lemma 4). This yields a combinatorial characterisation of coherent paths, given an embedding of each rigid component. The embedding of a rigid component is dealt with separately, later on. The significance of coherent paths stems from a crucial lemma (Lemma 4). This shows that an edge (a, b) is insertable in the graph preserving planarity if and only if the "projection" of any simple a, b path in the graph[2] onto the corresponding SPQR-tree roughly corresponds to a coherent path. This yields a criterion for testing planarity after an edge insertion which can be implemented in FO.

Insertions in biconnected components. Having filtered out non-planarity causing edges we turn to the question of how to construct the new planar embedding of the biconnected components after an edge insertion. The answer will lead us to investigate how to embed a rigid component when it is synthesized from a biconnected component.

It is in this context that we introduce the notion of two-colouring of separating pairs. This is a partial sketch of the new 3-connected component formed after an edge insertion. More concretely, the separating pairs along the path in the SPQR-tree are no longer separating pairs after the edge insertion and the common face (as ensured by the crucial lemma alluded to above) on which all the endpoints of the previous separating pairs lie splits into two faces. Since the embedding of a 3-connected planar graph is unique, after the edge insertion the two new faces formed are also unique, i.e., do not depend on the embedding. Thus the endpoints of each previous separating pair can be two coloured depending on which of the two faces a separating pair belongs to. We prove in the two-colouring lemma (Lemma 5) that no separating pair has both vertices coloured with the same colour.

Notice that when an edge is inserted such that its endpoints lie on a coherent path, all the rigid components on the path coalesce into one large rigid component (see Figure 3c). Two-colouring allows us to deal with flips by telling us the correct orientation of the coalescing rigid components on edge insertion. This, in turn, allows us to obtain the face-vertex rotation scheme of the modified component. In addition, it helps us to maintain the vertex rotation scheme in some corner cases (when two or more separating pairs share a vertex).

Face-vertex rotation scheme. The sceptical reader might question the necessity of maintaining the face-vertex rotation scheme for a 3-connected component. This is necessary for two reasons – first, to apply the planarity test we need to determine the existence of a common face containing a 4-tuple (or 3-tuple) of vertices. The presence of a face-vertex rotation scheme directly shows that this part is in FO. Second and more crucially, we need it to check if a particular triconnected component needs to be reflected after cascaded flips. Maintaining the vertex rotation scheme for biconnected components is now simple – we just need to collate the vertex rotation schemes for individual rigid components into one for the entire graph.

Handling deletions. On deleting edges while it's not necessary to perform additional flips, the rest of the updates is roughly the reverse of insertion. On deleting an edge from a rigid component, we infer two-colourings from the embedding of erstwhile rigid components

[2] which satisfies a minimality condition – it does not pass through both vertices of a separating pair

that decompose into pieces. Further, we have to update the coherent paths since possibly more edges are insertable preserving planarity. Notice that when an edge is deleted from a biconnected component this might lead to many simultaneous virtual edge deletions that might in turn cause triconnected components to decompose. Many ($O(n)$) invocations of the above triconnected edge deletion will be needed, but they can be handled in constant parallel time because they independent of each other as far as the updates required are concerned (see Figure 2).

Extension to the entire embedding. BC-trees for connected components have blocks and cut vertices as their nodes. We can maintain an embedding for the graph corresponding to a block or B-node as above. Since a non-cut vertex belongs to precisely one block, we can inherit the rotation scheme for such vertices from that of the blocks. For cut vertices, we need to splice together the vertex rotation scheme from each block that the cut vertex is incident on as long as the order respects the ordering provided by individual blocks.

Low level details of the information maintained. We maintain BC-tree for each connected component and SPQR-trees for each biconnected component thereof. In each of these trees we maintain betweenness information, i.e., for any three nodes X_1, X_2 and X_3 whether X_2 occurs on the tree path between X_1 and X_3. We also maintain a two-colouring of separating pairs for each P_i, P_j-coherent path in every SPQR-tree. For each rigid component R_i and each cycle component S_j we maintain their extended planar embedding. Specifically, we maintain the vertex rotation scheme in the following form. For every vertex v, we maintain triplet(s) (v_i, v_j, v_k) of neighbours of v that occur in the clockwise order though not necessarily consecutively. This enables us to insert and delete an arbitrarily large number of neighbours in FO making it crucial for the planar embedding procedure. This would not be possible if we were to handle individual insertions and deletions separately. See [7, Figure 6] for an example. We use a similar representation for the face-vertex rotation scheme.

For biconnected components, we maintain only a planar embedding (not the extended version) since the face-vertex rotation scheme is not necessary.

Comparison with existing literature. The main idea behind recent algorithms for planar embedding in the sequential dynamic setting has been optimizing the number of flips necessitated by the insertion of an edge. This uses either a purely incremental algorithm or alternatively, a fully dynamic but amortized algorithm. Since our model of computation is fully dynamic and does not allow for amortization, each change must be handled (i.e., finding out the correct cascading flips) in worst case $O(1)$-time on CRCW-PRAM. We note that filtering out edges that violate planarity in dynamic sequential $t(n)$ time (a *test-and-reject* model) implies an amortized planarity testing algorithm with $O(t(n))$ time (i.e., a *promise-free* model). In contrast, although we have a test-and-reject model we are unable to relax the model to promise-free because of lack of amortization.

There are weaker promise models such as the one adopted in [11] where for maintaining a bounded tree-width decomposition it is assumed that the graph has tree-width at most k without validating the promise at every step. In contrast our algorithm can verify the promise that no non-planarity-causing edge is added.

In terms of query model support, most previous algorithms [22, 21, 25] only maintain the vertex rotation scheme in terms of clockwise next neighbour, in fact, [22, 21] need $O(\log n)$ time to figure out the next neighbour. In contrast, we maintain more information in terms of arbitrary triplets of neighbours in (not necessarily consecutive) clockwise order. This allows us to sidestep following arbitrarily many pointers, which is not in FO. Finally, in terms of parallel time our algorithm (since it uses $O(1)$ time per query/update on CRCW-PRAMs) is optimal in our chosen model. In contrast, the algorithm of [21] comes close but fails to achieve the lower bound (of $\Omega(\log n)$) in the sequential model of dynamic algorithms.

4 Graph Theoretic Machinery

In this section, we present some graph theoretic results which will be crucial for our maintenance algorithm. We begin with a simple observation and go on to present some criteria for the planarity of the graph on edge insertion based on the type of the inserted edge.

▶ **Observation 2.** *For a 3-connected planar graph G, two planar embeddings $\mathcal{E}_1$ and $\mathcal{E}_2$ in the plane have the same vertex rotation scheme if between the two embeddings only the clockwise order of vertices on the boundary of the outer faces of the two embeddings are reverse if each other and all the clockwise order of vertices on the boundary of internal faces is same.*

Notice that due to the above fact, given a planar embedding with its outer face F_0 and an internal face F_1 specified, we can modify it to make F_1 the outer face while keeping the vertex rotation scheme unchanged by just reversing the orientation of the faces F_0 and F_1.

Next, we present some criteria to determine if an edge to be inserted in a planar graph causes it to become non-planar.

▶ **Lemma 3.** *For any 3-connected planar graph G, $G + \{\{a, b\}\}$ is planar if and only if a and b lie on the boundary of a common face.*

Next, let us consider the case where the vertices a and b are in the same block, say B_i, of a connected component of the graph. Let $R_a, R_b \in V(T_3(B_i))$ be two R-nodes in the SPQR-tree of B_i such that $a \in V(R_a)$ and $b \in V(R_b)$. Consider the path between R_a and R_b nodes in $T_3(B_i)$, $R_a, P_1, R_1, P_2, \ldots, R_k, P_{k+1}, R_b$, where R_i and P_i are R-nodes and S-nodes in $T_3(B_i)$ respectively, that appear on the path between R_a, R_b (see Figure 3). We have the following lemma (see the full version [7, Section 8] for the proof).

▶ **Lemma 4.** *$G + \{\{a, b\}\}$ is planar if and only if*
(a) *all vertices in $\{a\} \cup V(P_1)$ lie on a common face boundary in the embedding of R_a*
(b) *all vertices in $V(P_{k+1}) \cup \{b\}$ lie on a common face boundary in the embedding of R_b, and*
(c) *for each $i \in [k]$ all vertices in $V(P_i) \cup V(P_{i+1})$ lie on a common face in the embedding of R_i. Equivalently $R_a \rightsquigarrow R_b$ tree path is a coherent path.*

Consider the case in which $[2 \xrightarrow{+} 3]$ edge $\{a, b\}$ can be inserted into G preserving planarity. Notice that the 3-connected components $R_a, R_1, \ldots, R_k, R_b$ coalesce into one 3-connected component after the insertion of the edge. Let this coalesced 3-connected component be R_{ab}. Obviously, $\{a, b\}$ would be at the boundary of exactly two faces of R_{ab}, say F_0, F_1. We claim that the separating pair vertices in $\bigcup_{i\in[k+1]} V(P_i)$ all lie either on the boundary of F_0 or F_1. See Figure 1. We provide the proof of the following lemma in the full version of this paper [7, Section 8].

▶ **Lemma 5.** *The faces F_0 and F_1 define a partition into two parts on the set of vertices in the separating pairs $\bigcup_{i\in[k+1]} V(P_i)$ such that, for all $i \in [k+1]$ the two vertices in $V(P_i)$ belong to different blocks of the partition.*

Finally, if the vertices a and b are in the same connected component but not in the same biconnected component then we use the following lemma to test for edge insertion validity. Let the vertices a, b lie in a connected component C_i. Let $B_a, B_b \in N(T_2(C_i))$ be two block

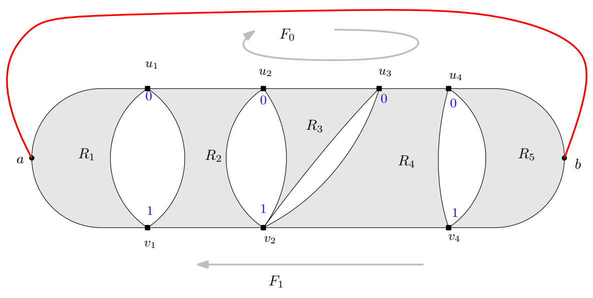

Figure 1 Two-colouring of P-node vertices.

nodes in the BC-Tree of the connected component C_i such that $a \in V(B_a)$ and $b \in V(B_b)$. Consider the following path between B_a and B_b in $T_2(C_i)$, $B_a, c_1, B_1, c_2, \ldots, c_k, B_k, c_{k+1}, B_b$ where B_i and c_i are block and cut nodes respectively, in $T_2(C_i)$ that appear on the path between B_a and B_b (see Figure 2). We abuse the names of cut nodes to also denote the cut vertex's name. Insertion of such an edge, i.e, $[1 \xrightarrow{+} 2]$ leads to the blocks $B_a, B_1, \ldots, B_k, B_b$ coalescing into one block, call it B_{ab}. In the triconnected decomposition of B_{ab} a new cycle component is introduced that consists of the edge $\{a, b\}$ and virtual edges between the consecutive cut vertices c_i, c_{i+1}, $i \in [k]$. See Figure 2d. We provide the proof of the following lemma in the full version of this paper [7, Section 8].

▶ **Lemma 6.** *$G + \{\{a, b\}\}$ is planar if and only if (a) $G[V(B_a)] + \{\{a, c_1\}\}$ is planar, (b) $G[V(B_b)] + \{\{c_{k+1}, b\}\}$ is planar, and (c) for each $i \in [k]$, $G[V(B_i)] + \{\{c_i, c_{i+1}\}\}$ is planar.*

5 Dynamic Planar Embedding

Our idea is to maintain planar embeddings of all triconnected components (S and R-nodes) of the graph and use those to find the embedding of the entire graph. Insertions and deletions of edges change the triconnected components of the graph, i.e., a triconnected component might decompose into multiple triconnected components or multiple triconnected components may coalesce together to form a single one. The same is true of biconnected components, i.e., a biconnected component might decompose into multiple biconnected components or multiple triconnected components may coalesce together to form a single biconnected component.

We discuss here how we update the embeddings of the triconnected components under insertions and deletions, assuming that we have the SPQR-tree and BC-tree relations available at every step (which we show how to maintain in the full version of this paper [7, Section 6]).

Some of the edge insertions/deletions are easier to describe, for example if the edge is being inserted in a rigid component then only the embedding of that rigid component has to change to reflect the presence of the new edge and introduction of two new faces. Thus, we first establish some notation to differentiate between classes of edges for ease of exposition.

▶ **Definition 7.** *A graph is* actually i-connected *if it is i-connected but is not $i+1$-connected for $i \in \{0, 1, 2\}$. For $i = 3$, a graph is actually i-connected if the graph is 3-connected.*

▶ **Definition 8.** *The* type *of an edge is $[i \xrightarrow{\sigma} j]$ where $i, j \in \{0, 1, 2, 3\}$ and $\sigma \in \{+, -\}$ such that*

- *$\sigma = +$ if the edge is being inserted into G. $\sigma = -$ if the edge is being deleted.*
- *both the endpoints are in a common actually i-connected component before the change and in a common actually j-connected component after the change.*

In the next two subsections we outline the updates in the triconnected planar embedding relations as well as the two-colouring relations which are described in complete detail in the full version of this paper [7, Section 10 and 11].

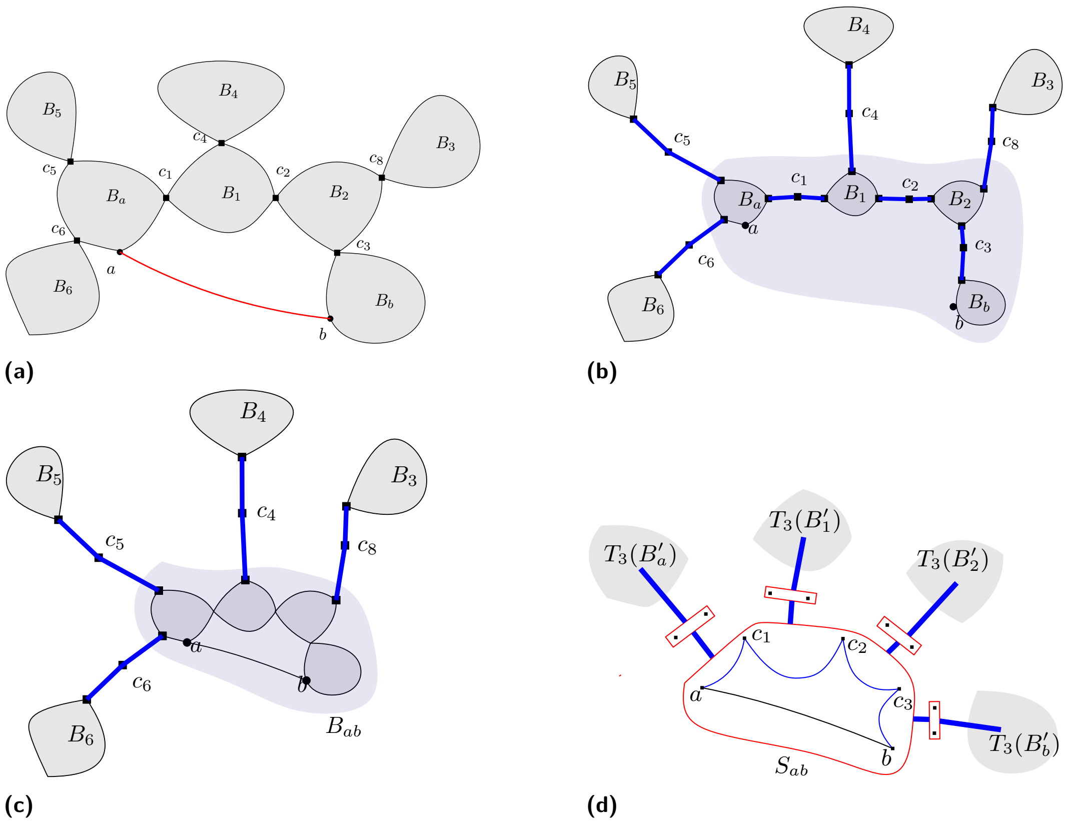

Figure 2 $[1 \xrightarrow{+} 2]$ edge $\{a, b\}$ insertion. (2a) To-be-inserted edge is highlighted in red. (2b) The path between block nodes B_a and B_b is highlighted in the BC-tree. (2c) The BC-tree of the connected component CC_i. $B_a \rightsquigarrow B_b$ path has shrunk to a single node B_{ab} in the BC-tree. (2d) The SPQR-tree of the block B_{ab} contains a cycle component S_{ab} made of the inserted edge and the new virtual edges that are introduced after its insertion.

5.1 Edge insertion

Find the type $[i \xrightarrow{+} j]$ of the inserted edge $\{a, b\}$ (where $i, j \in \{0, 1, 2, 3\}$). This can be done using [7, Lemma 10 and 11]. Depending on the type we branch to one of the following options:

(a) $[0 \xrightarrow{+} 1]$ This case affects only the BC-tree and SPQR-tree relations. Embedding and colouring relations remain unaffected.

(b) $[1 \xrightarrow{+} 2]$: In this case, both the endpoints a, b are in the same connected component C but not in the same biconnected component B_a, B_b (see Figure 2a). On this insertion, the BC-tree of the connected component changes. All the biconnected components on the $B_a \leftrightsquigarrow B_b$ path in the BC-tree of C coalesce into one biconnected component B_{ab} (see Figure 2c). In the BC-tree of the connected component, for each pair of consecutive cut-vertices on the $B_a \leftrightsquigarrow B_b$ path, a virtual edge is inserted in the biconnected component shared by the cut-vertex pair. This edge is a $[2 \xrightarrow{+} 3]$ or $[3 \xrightarrow{+} 3]$ edge and is handled below. Since the biconnected components involved are distinct, all these edges can be simultaneously inserted. In addition, a cycle component is introduced for which the face-vertex rotation scheme is computed via the betweenness relation in the BC-tree (see Figure 2d). The two-colouring relation is updated by extending the colouring of old biconnected components across the new cycle component (in Figure 2d, a coherent path from a P-node in $T_3(B'_a)$ to a P-node in $T_3(B'_2)$ will have to go via an S-node S_{ab}).

(c) $[2 \xrightarrow{+} 3]$: Both a, b are in the same biconnected component, say B_i, but not in the same rigid component. Let $a \in V(R_a)$ and $b \in R_b$, where R_a and R_b are two R-nodes in the SPQR-tree of B_i. The SPQR-tree of the biconnected component changes after the insertion as follows. All the rigid components on the $R_a \rightsquigarrow R_b$ SPQR-tree path coalesce into one rigid component (see Figures 3a, 3b). The embedding of the coalesced rigid component is obtained by combining the embeddings of the old triconnected components that are on the $R_a \rightsquigarrow R_b$ path, with their correct orientation computed from the two-colouring of the separating pairs for the corresponding coherent path. To update the two-colouring of the separating pair vertices, first we discard those old coherent paths and their two-colouring, that are no longer coherent as result of the insertion of $\{a, b\}$. While for the subpaths of the old coherent paths that remain coherent we obtain their two-colouring from that of the old path by ignoring colourings of old P-nodes on the $R_a \rightsquigarrow R_b$ path.

(d) $[3 \xrightarrow{+} 3]$: In this case, both the vertices are in the same 3-connected component. Due to this insertion, connectivity relations do not change. We identify the unique common face in the embedding of the 3-connected component that the two vertices lie on. We split the face into two new faces with the new edge being their common edge, i.e, the face vertex rotation scheme of the old face is split across the new edge. In the vertex rotation scheme of the two vertices, we insert the new edge in an appropriate order.

5.2 Edge deletion

Find the type of the edge $\{a, b\}$ that is being deleted. Depending on the type we branch to one of the following options:

(a) $[1 \xrightarrow{-} 0]$: This case occurs when both a, b are in the same connected component but not in the same biconnected component. The edge is a cut edge and only BC-tree and SPQR-tree relations are affected.

(b) $[2 \xrightarrow{-} 1]$: In this case, both vertices are in the same biconnected component but not in the same 3-connected component before the change. This is the inverse of the insertion operation of an edge of the type $[1 \xrightarrow{+} 2]$. The biconnectivity of the old biconnected component changes and it unfurls into a path consisting of multiple biconnected components that are connected via new cut vertices (in Figure 2c if the edge $\{a, b\}$ is deleted, the block B_{ab} will unfurl into the highlighted path in Figure 2b). Old virtual edges that are still present in the new biconnected components are deleted. These virtual edge deletions are of type $[3 \xrightarrow{-} 3]$ or $[3 \xrightarrow{-} 2]$ which we describe how to handle in the following cases.

(c) $[3 \xrightarrow{-} 2]$: In this case a, b are in the same rigid component R_{ab} (see Figure 3c) before the change. But after the deletion of $\{a, b\}$, R_{ab} decomposes into smaller triconnected components that are unfurled into a path in the SPQR-tree (see Figure 3b). We compute an embedding of the triconnected fragments of R_{ab} using the embedding of R_{ab}. From the updated embedding of R_{ab}, we compute the two-colouring of the separating pair vertices pertaining to the unfurled path on an SPQR-tree. We compose the obtained colouring with the rest of B_{ab}. The two-colouring of the elongated coherent paths is made consistent by flipping the colouring of the P-nodes of one subpath, if necessary.

(d) $[3 \xrightarrow{-} 3]$: In this case both a, b are in the same 3-connected component before and after the deletion of $\{a, b\}$. Connectivity relations (connectedness, bi/tri-connectivity) remain unchanged. The two faces in the embedding of the 3-connected component that are adjacent to each other via the to-be-deleted edge are merged together to form a new face. From the vertex rotation schemes of the two vertices, the edge is omitted. Pairs of coherent paths that merge due to the merging of two faces have to be consistently two-coloured.

Notice that we do not describe changes of types $[0 \xrightarrow{+} 2]$, $[0 \xrightarrow{+} 3]$, $[1 \xrightarrow{+} 3]$, $[3 \xrightarrow{-} 0]$, $[3 \xrightarrow{-} 1]$, $[2 \xrightarrow{-} 0]$, since edge insertion or deletion only changes the number of disjoint paths between any two vertices at most by one and thus these type of edge change are not possible. Also, we omit the changes of type $[2 \xrightarrow{+} 2]$ and $[2 \xrightarrow{-} 2]$ (related to S-nodes) as they are corner cases that we discuss in the full version of this paper [7, Section 11.4].

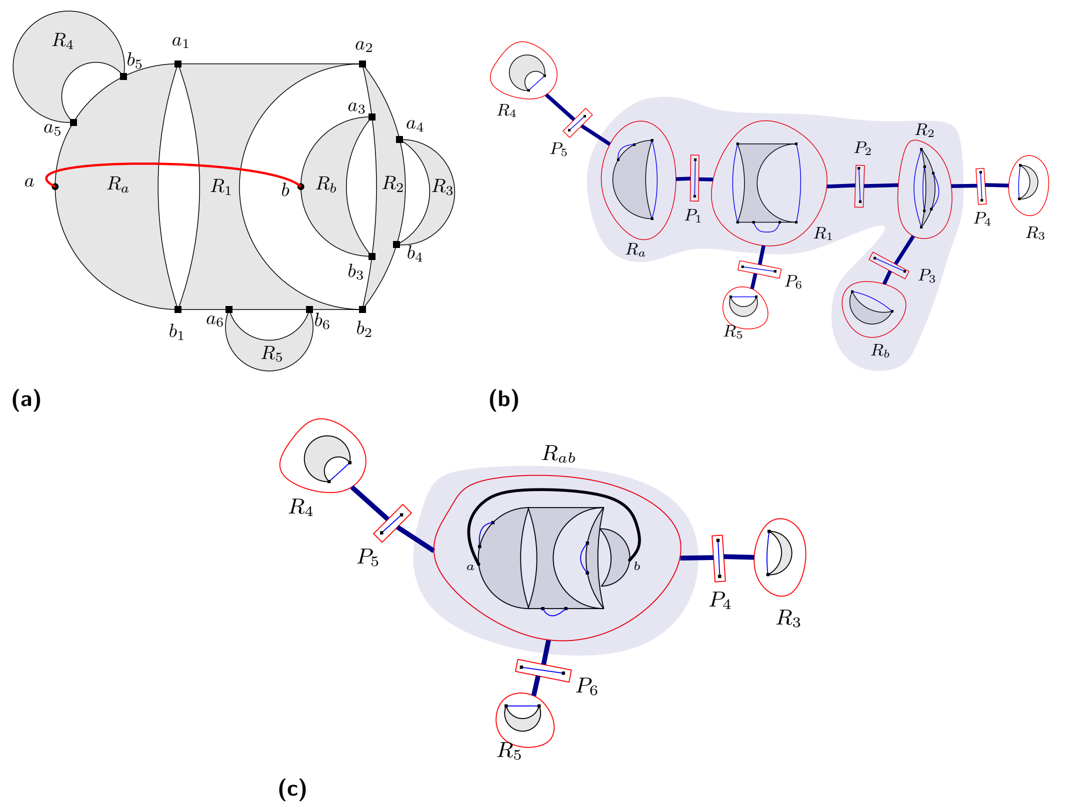

Figure 3 A $[2 \xrightarrow{+} 3]$ edge $\{a, b\}$ insertion. (3a) To-be-inserted edge is highlighted in red. (3b) The SPQR-tree of block B_i before insertion. The path between R-nodes R_a and R_b is highlighted. (3c) SPQR-tree of B_i after insertion. The $R_a \rightsquigarrow R_b$ path has shrunk to a single node R_{ab}.

5.3 Embedding the biconnected components

For each biconnected component, we maintain an embedding in the form of the vertex rotation scheme for its vertices. Note that we do not need to maintain the face-vertex rotation scheme for the biconnected components. The relevant updates for maintaining the biconnected components embedding are edge changes of type $[1 \xrightarrow{+} 2]$, $[2 \xrightarrow{+} 3]$, $[3 \xrightarrow{+} 3]$, $[3 \xrightarrow{-} 3]$, $[3 \xrightarrow{-} 2]$, and $[2 \xrightarrow{-} 1]$. For a $[2 \xrightarrow{+} 3]$ change, the part of the block that becomes triconnected, after the insertion of the edge, we patch together its vertex rotation scheme with unchanged fragments of the existing vertex rotation scheme in the affected block. For example, in Figure 3, after the insertion of edge $\{a, b\}$, R-nodes R_a, R_1, R_2 and R_b coalesce together to form R_{ab} (part of the block that has become triconnected). The embedding of the unchanged fragments of the block, viz., R_3, R_4 and R_5 is patched together at the appropriate P-nodes with the updated embedding of R_{ab} to update the embedding of the entire block. In a $[1 \xrightarrow{+} 2]$ change, multiple biconnected components coalesce together. First, we update the vertex rotation scheme of each block by inserting a required virtual edge and then patch together the vertex rotation schemes at the old cut vertices. For a $[3 \xrightarrow{-} 2]$ change, we splice in new virtual edges, that arise as a result of the change, in appropriate places in the vertex rotation scheme of the affected block. Details of the updates required in all types of edge changes are described in the full version of this paper [7, Section 12.1].

5.4 Embedding the entire graph

Assuming that we have planar embedding (in terms of the vertex rotation scheme) of each biconnected component of the graph we can compute a planar embedding of the whole graph as follows. For the vertices that are not cut vertices, their vertex rotation scheme is the same as their vertex rotation scheme in the embedding of their respective blocks that they belong to. For a cut vertex v, we join together its vertex rotation schemes from each block v belongs to. We only need to take care that in the combined vertex rotation scheme of v, its neighbours in each block appear together and are not interspersed.

We now complete the proof of the main theorem. We provide the complete details in the full version of this paper [7, Section 10, 11 and 12].

▶ **Lemma 9.** *The extended planar embeddings of all cycle, rigid components and a planar embedding for each biconnected component of the graph can be updated in* DynFO *under edge changes of type:* $[1 \xrightarrow{+} 2], [2 \xrightarrow{+} 3], [3 \xrightarrow{+} 3], [2 \xrightarrow{+} 2], [2 \xrightarrow{-} 2], [2 \xrightarrow{-} 1], [3 \xrightarrow{-} 2], [3 \xrightarrow{-} 3]$.

The above lemma combined with the implementation details provided in the full version [7, Section 12] allows us to prove the main theorem:

▶ **Theorem 1.** *Given a dynamic graph undergoing insertion and deletion of edges we can maintain a planar embedding of the graph in* DynFO *(while never allowing insertion of edges that cause the graph to become non-planar).*

Proof. If an edge insertion is within a 3-connected component the criterion in Lemma 3 tells us when the resulting component becomes non-planar. Analogously, Lemma 4 informs us when a 2-connected component is non-planar on an edge insertion within it (but across different 3-connected components). Similarly Lemma 6 allows us to determine if an edge added within a connected component but between two different 2-connected components causes non-planarity. An invocation of Lemma 9 shows how ot update the planar embedding for biconnected graphs. Moving on to connected graphs, vertex rotation schemes of the cut-vertices is obtained by combining their vertex rotation schemes in each block in any non-interspersed order, using FO primitive *merge vertex rotation schemes* (see [7, Section 9]). Remaining vertices inherit their vertex rotation scheme from the unique block they belong to. This completes the proof. ◀

6 Conclusion

We show that planarity testing and embedding is in DynFO where we are able to ensure that an edge is inserted if and only if it does not cause the graph to become non-planar. This is potentially an important step in the direction of solving problems like distance and matching where only an upper bound of DynFO[$\oplus$] (where FO[$\oplus$] is FO with parity quantifiers) was known in planar graphs. This is because we might be able to improve the known bound making use of planar duality which presupposes a planar embedding. It might also make problems like max flow, graph isomorphism, and counting perfect matchings which are all statically parallelisable when restricted to planar graphs, accessible to a DynFO bound.

References

1 Eric Allender and Meena Mahajan. The complexity of planarity testing. *Inf. Comput.*, 189(1):117–134, 2004.

2 David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.

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

4 Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.

5 Samir Datta, Chetan Gupta, Rahul Jain, Anish Mukherjee, Vimal Raj Sharma, and Raghunath Tewari. Dynamic meta-theorems for distance and matching. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 118:1–118:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

6 Samir Datta, William Hesse, and Raghav Kulkarni. Dynamic complexity of directed reachability and other problems. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming – 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 2014.

7 Samir Datta, Asif Khan, and Anish Mukherjee. Dynamic planar embedding is in DynFO, 2023. `arXiv:2307.09473`.

8 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018.

9 Samir Datta, Pankaj Kumar, Anish Mukherjee, Anuj Tawari, Nils Vortmeier, and Thomas Zeume. Dynamic complexity of reachability: How many changes can we handle? In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 122:1–122:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

10 Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Planar graph isomorphism is in Log-space. *ACM Trans. Comput. Theory*, 14(2):8:1–8:33, 2022.

11 Samir Datta, Anish Mukherjee, Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. A strategy for dynamic programs: Start over and muddle through. *Log. Methods Comput. Sci.*, 15(2), 2019.

12 Samir Datta, Anish Mukherjee, Nils Vortmeier, and Thomas Zeume. Reachability and distances under multiple changes. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, Prague, Czech Republic*, pages 120:1–120:14, 2018.

13 Samir Datta and Gautam Prakriya. Planarity testing revisited. In *Theory and Applications of Models of Computation – 8th Annual Conference, TAMC 2011, Tokyo, Japan, May 23-25, 2011. Proceedings*, pages 540–551, 2011.

14 Reinhard Diestel. *Graph Theory.* Springer Publishing Company, Inc., 5th edition, 2017.

15 Hristo N. Djidjev. On drawing a graph convexly in the plane (extended abstract). In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing*, pages 76–83, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

16 Guozhu Dong, Jianwen Su, and Rodney W. Topor. Nonrecursive incremental evaluation of datalog queries. *Ann. Math. Artif. Intell.*, 14(2-4):187–223, 1995.

17 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. I. planary testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996.

18 Shimon Even and Robert Endre Tarjan. Computing an *st* -numbering. *Theor. Comput. Sci.*, 2(3):339–344, 1976.

19 F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA, 1969.

20 Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Lacki, and Eva Rotenberg. Decremental SPQR-trees for planar graphs. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 46:1–46:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

21 Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 167–180, 2020.

22 Jacob Holm and Eva Rotenberg. Worst-case polylog incremental spqr-trees: Embeddings, planarity, and triconnectivity. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2378–2397, 2020.

23 J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

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

25 Giuseppe F. Italiano, Johannes A. La Poutré, and Monika Rauch. Fully dynamic planarity testing in planar embedded graphs (extended abstract). In *Algorithms – ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 – October 2, 1993, Proceedings*, pages 212–223, 1993.

26 Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930.

27 Saunders Mac Lane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460–472, 1937.

28 A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. *Theory of Graphs, International Syposium, Rome, July 1966, Rosenstiel, P. edit.*, 1967.

29 Jenish C. Mehta. Dynamic complexity of planar 3-connected graph isomorphism. In Edward A. Hirsch, Sergei O. Kuznetsov, Jean-Éric Pin, and Nikolay K. Vereshchagin, editors, *Computer Science – Theory and Applications – 9th International Computer Science Symposium in Russia, CSR 2014, Moscow, Russia, June 7-11, 2014. Proceedings*, volume 8476 of *Lecture Notes in Computer Science*, pages 273–286. Springer, 2014.

30 Anish Mukherjee. *Static and Dynamic Complexity of Reachability, Matching and Related Problems.* PhD thesis, CMI, 2019.

31 Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997.

32 Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 706–715, 1994.

33 Vijaya Ramachandran and John H. Reif. Planarity testing in parallel. *J. Comput. Syst. Sci.*, 49(3):517–561, 1994.

34 Jeffery R. Westbrook. Fast incremental planarity testing. In *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, pages 342–353, 1992.

35 Hassler Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932.