Worst-Case Deterministic Fully-Dynamic Biconnectivity in Changeable Planar Embeddings

Jacob Holm
University of Copenhagen, Copenhagen, Denmark

Ivor van der Hoog
Technical University of Denmark, Lyngby, Denmark

Eva Rotenberg
Technical University of Denmark, Lyngby, Denmark

Abstract

We study dynamic planar graphs with n vertices, subject to edge deletion, edge contraction, edge insertion across a face, and the splitting of a vertex in specified corners. We dynamically maintain a combinatorial embedding of such a planar graph, subject to connectivity and 2-vertex-connectivity (biconnectivity) queries between pairs of vertices. Whenever a query pair is connected and not biconnected, we find the first and last cutvertex separating them.

Additionally, we allow local changes to the embedding by flipping the embedding of a subgraph that is connected by at most two vertices to the rest of the graph.

We support all queries and updates in deterministic, worst-case, $O(\log^2 n)$ time, using an $O(n)$-sized data structure.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases dynamic graphs, planarity, connectivity

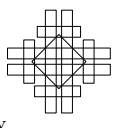
Digital Object Identifier 10.4230/LIPIcs.SoCG.2023.40

Related Version *Full Version*: https://arxiv.org/abs/2209.14079

Funding *Ivor van der Hoog* and *Eva Rotenberg*: Partially supported by Independent Research Fund Denmark grants 2020-2023 (9131- 00044B) "Dynamic Network Analysis".
Jacob Holm: Partially supported by the VILLUM Foundation grant 16582, "BARC".
Ivor van der Hoog: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

1 Introduction

In dynamic graph algorithms, the task is to efficiently update information about a graph that undergoes updates from a specified family of potential updates. Simultaneously, we want to efficiently support questions about properties of the graph or relations between vertices. Two vertices u and v are 2-vertex connected (i.e. biconnected) in a graph G, whenever after the removal of any vertex in G (apart from u and v) they are still connected in G. This work considers dynamically maintaining a combinatorial embedding of a graph that is planar, subject to biconnectivity queries between vertices. We show how to efficiently maintain G in $O(\log^2 n)$ time per update operation using linear space. We additionally support biconnectivity queries in $O(\log^2 n)$ time. The competitive parameters for dynamic algorithms include update time, query time, the class of allowed updates, the adversarial model, and whether times are worst-case or amortized. We present a deterministic algorithm: which means that all statements hold in the strictest adversarial model; against adaptive adversaries. Interestingly, for general graphs, there seems to be a large class of problems for which the deterministic amortized algorithms grossly outperform the deterministic worst-case time algorithms: for dynamic connectivity the state-of-the-art worst-case update time is of




the form $O(n^{o(1)})$ [14], whilst the state-of-the-art amortized update time is $\tilde{O}(\log^2 n)$ [21, 44]; for planarity testing, the best amortized solution has $O(\log^3 n)$ [26] update time, compared to $O(n^{2/3})$ worst-case [12] (in a restricted setting). For biconnectivity in general graphs the current best worst-case solution has update time $O(\sqrt{n})$ [6], while the best amortized update time is $\tilde{O}(\log^3 n)$ [21, 43, 27].

In this work, we provide algorithms for updating connectivity information of a combinatorially embedded planar graph, that is both deterministic, worst-case, and fully-dynamic.

▶ **Theorem 1.** *We maintain a planar combinatorial embedding in $O(\log^2 n)$ time subject to:*

- delete(e)*: where e is an edge, deleting the edge e,*
- insert(u, v, f)*: where u, v are incident to the face f, inserting an edge uv across f,*
- find-face(u, v)*: returns some face f incident to both u and v, if any such face exists.*
- contract(e)*: where e is an edge, contract the edge e,*
- split(v, c_1, c_2)*: where c_1 and c_2 are corners (corresponding to gaps between consecutive edges) around the vertex v, split v into two vertices v_{12} and v_{21} such that the edges of v_{12} are the edges of v after c_1 and before c_2, and v_{21} are the remaining edges of v,*
- flip(v)*: for a vertex v: flip the orientation of the connected component containing v.*

We may answer the following queries in $O(\log^2 n)$ time:

- connected(u, v)*, where u and v are vertices, answer whether they are connected,*
- biconnected(u, v)*, where u and v are connected, answer whether they are biconnected. When not biconnected, we may report the separating cutvertex closest to u.*

Our update time of $O(\log^2 n)$ should be seen in the light of the fact that even just supporting edge-deletion, insertion, and find-face(u, v), currently requires $O(\log^2 n)$ time [30]. We briefly review the concepts in this paper and the state-of-the-art.

Biconnectivity. For each connected component of a graph, the cutvertices are vertices whose removal disconnects the component. These cutvertices partition the edges of the graph into *blocks* where each block is either a single edge (a *bridge* or *cut-edge*), or a biconnected component. A pair of vertices are biconnected if they are incident to the same biconnected component, or, equivalently, if there are two vertex-disjoint paths connecting them. This notion generalises to k-connectivity where k objects of the graph are removed. While k-edge-connectivity is always an equivalence relation on the vertices, k-vertex-connectivity happens to be an equivalence relation for the edges only when $k \leq 2$.

Dynamic higher connectivity. Dynamic higher connectivity aims to facilitate queries to k-vertex-connectivity or k-edge-connectivity as the graph undergoes updates. For two-edge connectivity and biconnectivity in general graphs, there has been a string of work [11, 16, 6, 17, 21, 43, 27], and the current best deterministic results have $O(\log^2 n \log\log^2 n)$ amortized update time for 2-edge connectivity [27], and spend an additional amortized $\log(n)$-factor for biconnectivity [21, 43]. Thus, the current state of the art for deterministic two-edge connectivity is $\log\log(n)$-factors away from the best deterministic connectivity algorithm [44], while deterministic biconnectivity is $\log(n)$-factors away. See [10, 19, 18, 21, 43, 31, 29, 44, 32, 35, 14] for more work on dynamic connectivity. For k-(edge-)connectivity with $k > 2$, only partial results have appeared, including incremental [39, 3, 38, 37, 25] and decremental [13, 42, 22, 1] results. The strongest lower bound is by Pătraşcu et al. [40], and implies that the update- and query time cannot both be $o(\log n)$ for any of the mentioned fully dynamic problems on general graphs, and this holds even for planar

embedded graphs. For special graph classes, such as planar graphs, graphs of bounded genus, and minor-free graphs, there has been a bulk of work on connectivity and higher connectivity, e.g. [9, 20, 13, 15, 5, 33, 34, 23, 22]. For dynamic planar embedded graphs, there are poly-logarithmic worst-case algorithms for two-edge [20] and two-vertex [16] connectivity, that assume a fixed planar embedding, allowing only edge-deletions and edge-insertions across a face. In this paper, we obtain the same $O(\log^2 n)$ time bound as in [20, 16], but our graphs are subject to a wider range of dynamic updates, including Whitney-flips, and edge-contractions.

An open question remains whether higher connectivity can generally be maintained in polylogarithmic worst-case time for dynamic planar graphs (k-connectivity and k-edge connectivity, $k > 2$). Particularly, this is highly motivated already when $k = 3$: In the quest for fully-dynamic planarity testing with worst-case polylogarithmic update times, an efficient algorithm for 3-vertex connectivity would be a major milestone. Namely, a 3-connected graph has a unique planar embedding (up to reflection). Much of the work on (dynamic) planarity testing [28, 36, 7, 25, 24] goes via understanding (changes to) the SPQR-tree; a tree over the 3-connected components and their interrelations. Given this, it is likely that any efficient worst-case fully-dynamic planarity testing algorithm would rely upon an efficient worst-case fully-dynamic 3-connectivity data structure. Note here that supporting changes to the embedding is crucial for this venture, since a deletion-insertion-sequence may require changes to the embedding, in order to remain planar. This paper presents the first step in this quest towards *worst-case* polylogarithmic fully-dynamic planarity testing, as we present a *worst-case* polylogarithmic data structure for 2-vertex connectivity subject to the required embedding-changing operations.

Techniques. Exploiting properties of planar graphs, we use the tree-cotree decomposition: a partitioning of edges into a spanning tree of the graph and a spanning tree of its dual. Using tree-cotree decompositions to obtain fast dynamic algorithms is a technique introduced by Eppstein [4], who obtains algorithms for dynamic graphs that have efficient genus-dependent running times. Note that the construction in [4] does not facilitate inserting edges in a way that minimises the resulting genus. Such queries are, however, allowed in the structure by Holm and Rotenberg [23], which also utilises the tree-cotree decomposition.

On this spanning tree and cotree, we use top-trees to handle local biconnectivity information. Much of our work concerns carefully choosing which biconnectivity information is relevant and sufficient to maintain, as top-tree clusters are merged and split. Note that the ideas for two-edge connectivity introduced by Hershberger et al. in [20], i.e. ideas of using topology trees on a vertex-split version of the graph to keep track of edge bundles, do not transfer to the problem at hand, since vertex-splitting changes the biconnectivity structure.

2 Preliminaries

We study a dynamic plane embedded graph $G = (V, E)$, where V has n vertices. We assume access to G and some *combinatorial embedding* [4] of G that specifies for every vertex in G the cyclical ordering of the edges incident to that vertex. Throughout the paper, we maintain some associated spanning tree $T_G = (V, E')$ over G. We study the combinatorial embedding subject to the update operations specified in Theorem 1. This is the same setting and includes the same updates as by Holm and Rotenberg [23].

Spanning and co- trees. If G is a connected graph, a spanning tree T_G is a tree where its vertices are V, and the edges of T_G are a subset of E such that T_G is connected. Given T_G, the cotree T_G^Δ has as vertices the faces in G, and as edges of T_G^Δ are all edges dual to those in $G \setminus T_G$. It is known that the cotree is a spanning tree of the dual graph of G [8].

Induced graph. We adopt the standard notion of (vertex) induced subgraphs: for any $V' \subseteq V$, $G[V']$ is the subgraph created by all edges $e \in E$ with both endpoints of e in V'. For any G and V', we denote by $G \setminus G[V']$ the graph G minus all edges in $G[V']$. Observe that for (V_1, V_2) the set $G[V_1 \cup V_2]$ is not necessarily equal to $G[V_1] \cup G[V_2]$ (Figure 1).

Top trees. Our data structure maintains a specific variant of a *top tree* $\mathcal{T}_G^{op}$ over the graph G [2, 41, 23]. This data structure (Figure 2) is a hierarchical decomposition of a planar, embedded, graph G based on a spanning tree T_G of G. Formally, for every connected subgraph S of T_G we define the *boundary vertices* of S as the vertices incident to an edge in $T_G \setminus S$. A *cluster* is a connected subgraph of T_G with at most 2 boundary vertices. A cluster with one boundary vertex is a *point cluster*; otherwise a *path cluster*. A top tree $\mathcal{T}_G^{op}$ is a hierarchical decomposition of G (with depth $O(\log n)$) into point and path clusters that is structured as follows: the leaves of $\mathcal{T}_G^{op}$ are the path and point clusters for each edge (u, v) in T_G (a leaf in $\mathcal{T}_G^{op}$ is a point cluster if and only if the corresponding edge (u, v) is a leaf in T_G). Each inner node $\nu \in \mathcal{T}_G^{op}$ *merges* a constant number of *child* clusters sharing a single vertex into a new point or path cluster. The vertex set of ν is the union of those corresponding to its children. We refer to combining a constant number of nodes into a new inner node as a *merge*. We refer to its inverse as a *split*. Furthermore, for planar embedded graphs, we restrict our attention to *embedding-respecting* top trees; that is, given for each vertex a circular ordering of its incident edges, top trees that only allow merges of neighbouring clusters according to this ordering. In other words, if two clusters ν and μ share a boundary vertex b, and are mergable according to the usual rules of top trees, we only allow them to merge if furthermore they contain a pair of neighbouring edges $e_\mu \in \mu$ and $e_\nu \in \nu$ where e_μ is a neighbour of e_ν around b. Holm and Rotenberg [23] show how to dynamically maintain $\mathcal{T}_G^{op}$ (and the spanning tree and cotree) with the following property:

▶ **Property 1.** *Let $\nu \in \mathcal{T}_G^{op}$ be a point cluster with boundary vertex u. The graph $G[\nu]$ is a contiguous segment of the extended Euler tour of T_G.*

▶ **Corollary 1.** *Let $\nu \in \mathcal{T}_G^{op}$ be a point cluster with boundary vertex u. The edges of $G[\nu]$ that are incident to u form a connected interval in the clockwise order around u.*

Edge division. For ease of exposition, we perform the trick of subdividing edges into paths of length three. We refer to G^o as the original and G as this edge-divided graph. Since G^o is planar, this does not asymptotically increase the number of vertices. We note:

1. Edge subdivision respects biconnectivity (since edge subdivision preserves the cycles in the graph; it preserves biconnectivity).

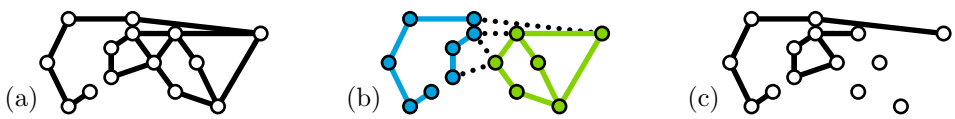


Figure 1 (a) A planar graph G. (b) A set $\beta \subset V$ and $\gamma \subset V$. We show $G[\beta]$ and $G[\gamma]$. The set $G[\beta] \cup G[\gamma]$ contains no black edges. (c) We show $G \setminus G[\gamma]$.

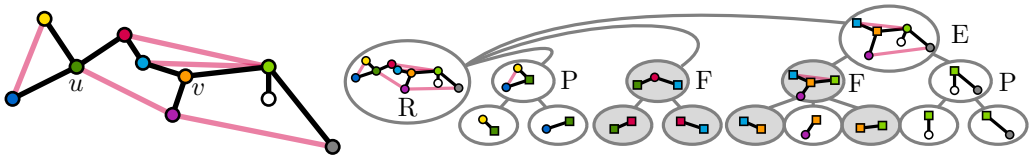


Figure 2 We recursively decompose G based on a spanning tree T_G. Square vertices are boundary vertices. We highlight path clusters. The root node has three children, where one is a path cluster that exposes $\{u, v\}$. The letters indicate the later defined merge type.

2. Any spanning tree of G^o can be transformed into a spanning tree of G where all non-tree edges have end points of degree two: for each non-tree edge in G, include exactly the first and last edge on its corresponding path in the spanning tree. This property can easily be maintained by any dynamic tree algorithm.
3. Dynamic operations in G^o easily transform to constantly many operations in G.

With this in place, our top tree structure automatically maintains more information about the endpoints of non-tree edges and their ordering around each endpoint.

Paper notation. We refer to vertices in G with Latin letters. We refer to nodes in the top tree $\mathcal{T}_G^{op}$ with Greek letters. We refer indistinguishably to nodes $\nu \in \mathcal{T}_G^{op}$ and their associated vertex set. Vertices u and v are boundary vertices. For a path cluster $\nu \in \mathcal{T}_G^{op}$ with boundary vertices $\{u, v\}$ we call its *spine* $\pi(\nu)$ the path in T_G that connects u and v. For any path, its *internal vertices* exclude the two endpoints. For a point cluster with boundary vertex u, its spine $\pi(\nu)$ is u. We denote by $\mathcal{T}_G^{op}(\nu)$ the subtree rooted at ν.

Slim-path top trees over G. We use a variant of the top tree called a *slim-path top tree* by Holm and Rotenberg [23]. This variant of top trees upholds the *slim-path invariant*: for any path-cluster ν, all edges (of the spanning tree T_G) in the cluster that are incident to a boundary vertex belong to the spine. In other words: for every path cluster $\nu \in \mathcal{T}_G^{op}$, for each boundary vertex u, there is exactly one edge in the induced subgraph $G[\nu]$ that is connected to u. The root of this top tree is the merge between a path cluster with boundary vertices u and v, with at most two point clusters λ, μ, with $\pi(\lambda) = \{u\}$ and $\pi(\mu) = \{v\}$.[1] Holm and Rotenberg show how to obtain (and dynamically maintain) this top tree with four types of merges between clusters, illustrated by Figure 3 and 4. Our operations merge:

(Root merge) at most two point clusters and a path cluster to create the root node,

(Point merge) two point clusters μ, ν with $\pi(\mu) = \pi(\nu)$.

(End merge) a point and a path cluster that results in a point cluster, and

(Four-way merge) two path clusters μ, ν and at most two point clusters α, β, where their common intersection is one *central* vertex m. If there are two point clusters, they are not adjacent around m. This merge creates a path cluster.

Holm and Rotenberg [23] dynamically maintain the above data structure with at most $O(\log n)$ merges and splits per graph operation (where each merge or split requires $O(\log n)$ additional operations). Their data structure supports two additional critical operations:

Expose(u, v) selects two vertices u, v of G and ensures that for the unique path cluster ν of the root node, u and v are the two endpoints of $\pi(\nu)$; ($O(\log n)$ splits/merges).

[1] In the degenerate case where the graph is a star, we add one dummy edge to G to create a path cluster.

Meet(u, v, w) selects three vertices u, v, w of G and returns their meet in T_G, defined as the unique common vertex on all 3 paths between the vertices. Moreover, they also support this operation on the cotree T_G^Δ; ($O(\log n)$ time).

When ν is a node in an embedding-respecting slim-path toptree, and G is formed from G^o via edge-subdivisions, note that $G[\nu]$ has the following properties:

- For a point cluster ν with boundary vertex b that encompasses the tree-edges e_1 to e_l in the circular ordering around b, $G[\nu]$ corresponds to the sub-graph in G^o induced by all vertices in ν, except edges to b that are not in $e_1, \ldots, e_l$.
- When ν is a path cluster, $G[\nu]$ corresponds to the sub-graph in G^o induced by all vertices in ν except non-tree edges incident to either of the two boundary vertices.

3 Dynamic biconnectivity queries and data structure

We want to maintain a slim-path top tree, subject to the aforementioned operations, that additionally supports biconnectivity queries in $O(\log^2 n)$ time. Our data structure consists of the slim-path top tree from [23] with three invariants which we formalise later:

1. For each cluster $\nu \in \mathcal{T}_G^{op}$, we store the biconnected components in $G[\nu]$ which are relevant for the exposed vertices (u, v).
2. For each cluster ν, we store the information required to navigate through the top tree.
3. For each stored biconnected component, we store its "border" along the spine $\pi(\nu)$.

We show the technical details for our invariants. The full version contains the proofs and larger figures. We specify for each $\nu \in \mathcal{T}_G^{op}$, for each endpoint u of the spine, a *designated face*:

▶ **Lemma 1.** *Let $\nu \in \mathcal{T}_G^{op}$ be a path cluster. Each boundary vertex u (resp. v) is incident to a unique face $f_u^{\text{des}}(\nu)$ (resp. $f_v^{\text{des}}(\nu)$) of $G[\nu]$. Moreover, all edges in G that are not in $G[\nu]$ are contained in either $f_u^{\text{des}}(\nu)$ or $f_v^{\text{des}}(\nu)$.*[2]

▶ **Lemma 2.** *Let $\nu \in \mathcal{T}_G^{op}$ be a point cluster with boundary vertex u. The subgraph $G[\nu]$ has a unique face $f_u^{\text{des}}(\nu)$ such that all edges in G that are not in $G[\nu]$ are contained in $f_u^{\text{des}}(\nu)$.*

▶ **Corollary 2.** *Let $\nu \in \mathcal{T}_G^{op}$ have a boundary vertex u, let μ be a descendent of ν and x be the boundary vertex of μ closest to u in T. Then $f_u^{\text{des}}(\nu) \subseteq f_x^{\text{des}}(\mu)$.*

[2] Formally, we can say that an edge, vertex, or face in G is contained in face f of a subgraph G', if it is contained in f in any drawing of G that is consistent with the current combinatorial embedding.

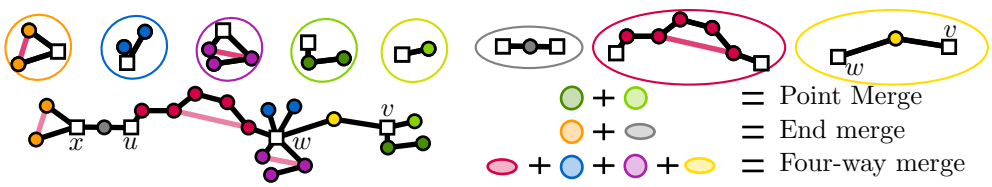


Figure 3 A graph G which we already split into five point clusters (circles) and three path clusters (ovals). We show the combinations of clusters that create three merge types. To obtain the root: execute the three suggested merges and merge the remaining components.

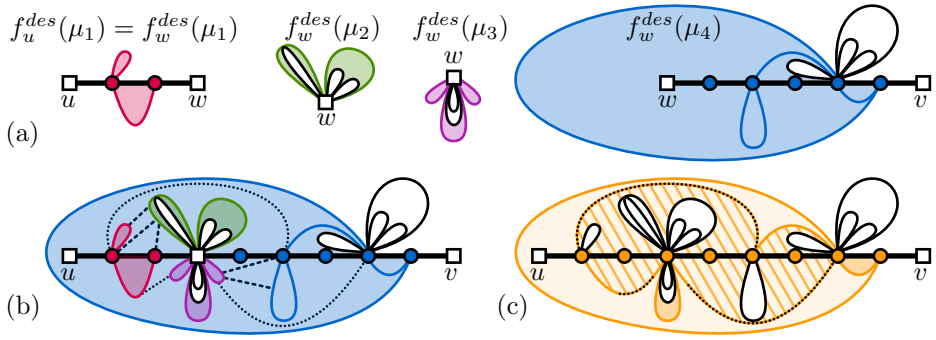


Figure 4 (a) Nodes μ_i with faces $f_u^{des}(\mu_1)$ and $f_w^{des}(\mu_i)$. We color $BC_u(\mu_1, f_u^{des}(\mu_1))$ and $BC_w(\mu_i, f_w^{des}(\mu_i))$. (b) The four-way merge introduces a new node ν. Edges in $G[\nu]$ that are not the induced subgraph of $G[\mu_j]$ for $j \in \{1, 2, 3, 4\}$ are dashed/dotted. (c) Every dotted edge is part of a new biconnected component $B \in BC_u^*(\nu)$ which we show as tiled.

Lemmas 1 and 2 inspire the following definition: for all $\nu \in \mathcal{T}_G^{op}$, for each boundary vertex u (or v) of ν, there exists a unique face f which we call its *designated face* $f_u^{des}(\nu)$ (or $f_v^{des}(\nu)$). For biconnectivity between the exposed vertices, we are only interested in biconnected components that are edge-incident to $f_u^{des}(\nu)$ or $f_v^{des}(\nu)$. Let for a node ν with boundary vertex u, a biconnected component B be edge-incident to $f_u^{des}(\nu)$. Let μ be a descendant of ν and $B \subseteq G[\mu]$ then, by Corollary 2, B must be edge-incident to $f_x^{des}(\mu)$ where x is the boundary vertex of μ closest to u. This relation inspires us to define the *projected face* of u in μ as $\hat{f}_u^{des}(\mu) = f_x^{des}(\mu)$.

Relevant and alive biconnected components. Consider for a cluster ν, a biconnected component B of the induced subgraph $G[\nu]$. We say that B is *relevant* with respect to ν if B is vertex-incident to the spine $\pi(\nu)$. We say that B is *alive* with respect to a face f in $G[\nu]$ if B is edge-incident to f. We denote by $BC(\nu, f)$ the set of biconnected components in the induced subgraph $G[\nu]$ that are relevant with respect to ν and alive with respect to f. Intuitively, we want to keep track of the relevant and alive components (with respect $f_u^{des}(\nu)$ or $f_v^{des}(\nu)$). To save space, we store only the relevant biconnected components of ν that are not in its children. Formally (Figure 4), we define an invariant:

▶ **Invariant 1.** *For each cluster $\nu \in \mathcal{T}_G^{op}$ (apart from the root) with children $\mu_1, \mu_2, \ldots \mu_s$ where u is a boundary vertex of ν, we store a unique object for each element in:*

$$BC_u^*(\nu) := BC(\nu, f_u^{\text{des}}(\nu)) \setminus \bigcup_{i=1}^{s} BC(\mu_i, \hat{f}_u^{\text{des}}(\mu_i)).$$

Storing biconnected components in this way does not make us lose information:

▶ **Lemma 3.** *Let $\nu \in \mathcal{T}_G^{op}$ with boundary vertex u and $B \in BC(\nu, f_u^{\text{des}}(\nu))$. There exists a unique node μ in $\mathcal{T}_G^{op}(\nu)$ where: $B \in BC_x^*(\mu)$ and x is the closest boundary vertex of μ to u.*

In the remainder of this paper, we show that for each cluster $\nu \in \mathcal{T}_G^{op}$ with boundary vertex v, the set $BC_v^*(\nu)$ contains constantly many elements. Moreover, the root vertices (u, v) are biconnected at the root μ if and only if they share a biconnected components in $BC_u^*(\mu)$. In this section, we define two additional invariants so that we can maintain Invariant 1 in $O(\log n)$ additional time per split and merge. This will imply Theorem 1.

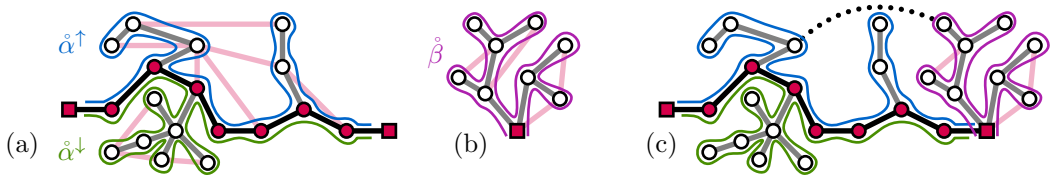


Figure 5 (a) A path cluster α with $\pi(\alpha)$ in black and edges in T_G in black or grey. We show the tourpaths $\mathring{\alpha}^{\uparrow}$ and $\mathring{\alpha}^{\downarrow}$ in blue and green. (b) A point cluster β with the path $\mathring{\beta}$. (c) Any edge in $G[\alpha \cup \beta] \setminus (G[\alpha] \cup G[\beta])$ must intersect one of $\{\mathring{\alpha}^{\uparrow}, \mathring{\alpha}^{\downarrow}\}$ and $\mathring{\beta}$.

The core of our data structure is a slim-path top tree $\mathcal{T}_G^{op}$ on G, that supports the expose operation in $O(\log n)$ splits and merges. In addition, it supports the meet operation in both the spanning tree T_G and its cotree T_G^{Δ} in $O(\log n)$ time. To maintain Invariant 1 in $O(\log n)$ time per split and merge, we add the following invariant for $\mathcal{T}_G^{op}$ where:

▶ **Invariant 2.**
(a) *each node ν has pointers to its boundary vertices and parent node.*
(b) *each path cluster ν stores: the length of $\pi(\nu)$ and its outermost spine edges.*
(c) *each point cluster ν stores: the number of tree edges in ν incident to the boundary vertex.*
(d) *each $x \in V$ points to the lowest common ancestor in $\mathcal{T}_G^{op}$ where x is a boundary vertex.*

Finally we add one final invariant which uses three additional concepts: slices of biconnected components, index orderings on the spine and an orientation on edges incident to a spine.

Biconnected component slices. For any node $\nu \in \mathcal{T}_G^{op}$ and any biconnected component $B \subseteq G[\nu]$, its *slice* is the interval $B \cap \pi(\nu)$ (which may be empty, or one vertex).

▶ **Lemma 4.** *Let $\nu \in \mathcal{T}_G^{op}$. For all (maximal) biconnected components B in $G[\nu]$, if $B \cap \pi(\nu)$ is not empty, it is a path (possibly consisting of a single vertex).*

Index orderings. For any $\nu \in \mathcal{T}_G^{op}$ and $w \in \pi(\nu)$, let w be the i'th vertex on $\pi(\nu)$. We say that i is the *index* of w in $\pi(\nu)$. We can use Invariant 2 to obtain this index:

▶ **Lemma 5.** *Given Invariant 2, a path cluster ν and a vertex $w \in \pi(\nu)$, we can compute for every path cluster β with $\pi(\beta) \subseteq \pi(\nu)$ the index of w in $\pi(\beta)$ in $O(\log n)$ total time.*

Similarly in a point cluster ν, for each tree edge in ν incident to the boundary vertex u we define its *clockwise index* in ν as its index in the clockwise ordering of the edges around u.

▶ **Lemma 6.** *Given Invariant 2, and an edge $e = (u, x) \in T_G$, we can compute the clockwise index of e in every point cluster $\nu \in \mathcal{T}_G^{op}$ that contains e and has u as boundary vertex, simultaneously, in worst case $O(\log n)$ total time.*

Euler tour paths and endpoint orientations. Consider the Euler tour of T_G and an embedding of that Euler tour such that the Euler tour is arbitrarily close to the edges in T_G. Each edge e in G that is not in T_G must intersect the Euler tour twice. We classify each endpoint of e based on where it intersects this Euler tour. Formally, we define (Figure 5):

▶ **Definition 3.** *For a point cluster ν with boundary vertex u, we denote by $\mathring{\nu}$ its **tourpath** (the segment of the Euler tour in T_G from u to u that is incident to edges in $G[\nu]$).*

▶ **Definition 4.** *For a path cluster ν with boundary vertices u and v. We denote by $\mathring{\nu}^{\uparrow}$ and $\mathring{\nu}^{\downarrow}$ its two* ***tourpaths*** *(the two paths in the Euler tour in T_G from u to v).*

▶ **Definition 5** (Figure 6)**.** *For any tourpath $\mathring{\alpha}$ let e_1 be the first and e_2 be the last edge of T_G incident to $\mathring{\alpha}$. We denote by $f^{\text{first}}(\mathring{\alpha})$ the unique face in G (incident to e_1) whose interior contains the start of $\mathring{\alpha}$. The face $f^{\text{last}}(\mathring{\alpha})$ is defined analogously using e_2.*

▶ **Observation 6.** *Let ν be a path cluster with the slim-path property. Any edge in $G[\nu]$ is either an edge in T_G or it must intersect one of $\{\mathring{\nu}^{\uparrow}, \mathring{\nu}^{\downarrow}\}$.*

We introduce one last concept. Let $e = (x, y)$ be an edge of G not in T_G where e is an edge with an endpoint in $G[\alpha]$ and in $G[\beta]$ (for two clusters $\alpha, \beta \in \mathcal{T}_G^{op}$). We intuitively refer for an endpoint x of e, to the tourpath intersected by e "near" x. Let α be a path cluster. We say that the endpoint $x \in \alpha$ of e is a *northern* endpoint if e intersects $\mathring{\alpha}^{\uparrow}$ near x, and a *southern* endpoint if e intersects $\mathring{\alpha}^{\downarrow}$ near x. This distinction between north and south endpoints inspires the notion of biconnected component borders (Figure 7):

▶ **Definition 7** (Biconnected component borders)**.** *Let B be a subset of the edges in $G[\nu]$ that induces a biconnected subgraph such that $B \cap \pi(\nu)$ is a path (or singleton vertex).*

- *Let ν be a point cluster. Consider the clockwise ordering of edges in B incident to its boundary vertex u, starting from $f_u^{\text{des}}(\nu)$. The* ***border*** *of B is:*
  - *the vertex u together with its* ***eastern border****: the first edge of B in this ordering, and its* ***western border****: the last edge of B in this ordering.*
- *Let ν be a path cluster. Denote by a the "eastmost" vertex of $B \cap \pi(\nu)$ and by b its "westmost" vertex. If $a = b$ then the border is empty. Otherwise:*
  - *the* ***eastern border*** *of B is a, together with the first northern and last southern edge of B that is incident to a.*
  - *the* ***western border*** *of of B is b, together with the last northern and first southern edge of B that is incident to b.*

▶ **Invariant 3.** *For any $\nu \in \mathcal{T}_G^{op}$ with boundary vertex u, for all $B \in BC_u^*(\nu)$ we store the* ***borders*** *of B in ν and their indices and clockwise indices in ν.*

Using invariants for biconnectivity. Invariant 2 allows us to not only obtain the meet between vertices, but also the edges of their path to this meet (incident to the meet):

▶ **Theorem 2.** *Given Invariant 2, let ν be a path cluster with boundary vertices u and v and $w \notin \pi(\nu)$ be a vertex in $G[\nu]$. We can obtain the meet $m = meet(u, v, w)$ and the last edge e^* in the path from w to m (in T_G) in $O(\log n)$ time.*

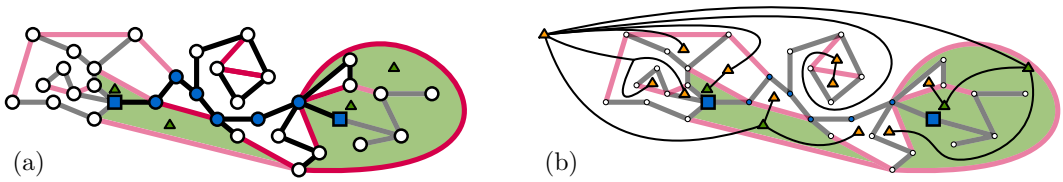


Figure 6 (a) A graph with the spine of a node ν shown in blue. We show the faces $(f^{first}(\mathring{\alpha}^{\uparrow}), f^{last}(\mathring{\alpha}^{\uparrow}), f^{first}(\mathring{\alpha}^{\downarrow}), f^{last}(\mathring{\alpha}^{\downarrow})$ in green. (b) The cotree T_G^{Δ}. Vertices are triangles.

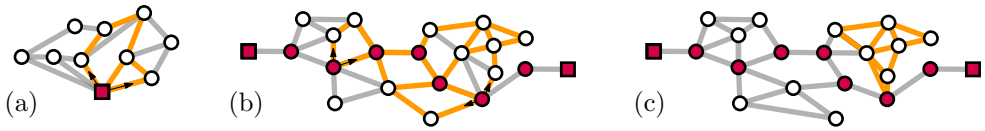


Figure 7 Three times a cluster ν with $\pi(\nu)$ as red vertices and a yellow (not maximal) set of biconnected edges in $G[\nu]$. We show: (a) a border in a point cluster (b) a border in a path cluster and (c) a set of biconnected edges in $G[\nu]$ that has an empty border.

In our later analysis, we show that for each merge, the only edges e° which can be part of new relevant and alive biconnected components are the edges incident to some convenient meets in the dual graph. Given such an edge e°, we identify a convenient edge e^* of the newly formed biconnected component B. We identify the already stored biconnected components $B^* \in BC_u(\nu)$ which contain e^* (these components B^* get "absorbed" into B). We use Invariant 3 to identify all such B^* that contain e^*:

▶ **Theorem 3.** *Let $e^* \in T_G$ be an edge incident to a vertex u. Let k be the maximum over all u and ν of the number of elements in $BC^*_u(\nu)$. In $O(k \log n)$ total time we can, for all of the $O(\log n)$ nodes $\nu \in \mathcal{T}^{op}_G$ that contain e^*, for each $B^* \in BC^*_u(\nu)$, determine if e^* is in between the border of B^* in ν.*

Finally, Invariants 2 + 3 will suffice to maintain Invariant 1 in $O(\log n)$ time per split and merge (and thus, in $O(\log^2 n)$ time per update operation) which will prove Theorem 1.

4 Summary of the remainder of this paper

We present a high-level overview of how Theorem 1 is obtained in the remainder of this paper. At all times, we dynamically maintain a (combinatorial) embedding of some edge-divided graph G. We maintain the top tree $\mathcal{T}^{op}_G$ by Holm and Rotenberg from [23] augmented with three aforementioned invariants. All updates the combinatorial embedding in Theorem 1 can be realized by $O(\log n)$ split and merge operations on the top tree (and co-tree). On a high level, we maintain all three invariants with $O(\log n)$ additional time per split and merge in the top tree (and co-tree). Thus, we have $O(\log^2 n)$ total update time.

4.1 Invariant 2: pointers in the top tree

Invariant 2 specifies that we want to store for each cluster ν in the top tree some "metadata". This metadata can be stored using $O(1)$ space and $O(\log n)$ additional time per split and merge. Indeed, per split we simply delete the constant-complexity data. It may occur that for a vertex $x \in V$, we delete the lowest common ancestor in $\mathcal{T}^{op}_G$ where x is the boundary vertex: Since a top tree is a balanced tree with $O(1)$ boundary vertices per node, we find the new lowest common ancestor in $O(\log n)$ time. For a merge, we simply compute components $(a), (b)$ and (c) in $O(1)$ additional time. For the $O(1)$ boundary vertices, we test if there exists a vertex $x \in V$ for which Invariant 2(d) changes in $O(\log n)$ additional time.

4.2 Invariant 1: maintaining $BC^*_u(\nu)$

We define k as the maximum over all vertices u and clusters ν, of the size of $BC^*_u(\nu)$. During each split, a cluster ν with boundary vertex u is destroyed and we simply delete $BC^*_u(\nu)$ in $O(k)$ time. What remains is to show that when we merge clusters in our top trees to create a vertex ν with boundary vertex u, we can identify the new $BC^*_u(\nu)$.

Suppose we merge clusters α and β to create a cluster ν with boundary vertex u (Figure 8 (a)). We want to construct the set $BC^*_u(\nu)$ of "newly formed" relevant and alive biconnected components. We assumed that every such set contains at most k elements. Any $B \in BC^*_u(\nu)$ contains at least one edge e° in $G[\alpha \cup \beta] \backslash (G[\alpha] \cup G[\beta])$. Assume, for now, that there exist at most $O(k)$ such edges e°.

Any edge e° in $G[\alpha \cup \beta] \backslash (G[\alpha] \cup G[\beta])$ must be part of some new biconnected component B in $G[\nu]$. Indeed: e° has one endpoint x in $G[\alpha]$ and y in $G[\beta]$. The edge e° together with the path π° in the spanning tree connecting x and y must form a cycle. To determine whether B is a relevant and alive biconnected component, we want to (implicitly) compute the out-most cycle bounding B. This is not straightforward: as the biconnected component B contains the aforementioned cycle, but may also absorb biconnected components B_α in $G[\alpha]$ and B_β in $G[\beta]$ (Figure 8 (b)).

Testing whether B is relevant. The new biconnected component B is in $BC^*_u(\nu)$ whenever it (partially) coincides with at least one edge from the spine $\pi(\nu)$. We show that this occurs if and only if π° (partially) coincides with $\pi(\nu)$ and we test this in $O(\log n)$ time.

Testing whether B is alive. The newly formed biconnected component B is alive whenever it is incident to the face $f^{des}_u(\nu)$ in the graph $G[\nu]$. We want for a given pair (e°, B) test whether B is alive in $O(k \log n)$ time. In the full version we show how to do this efficiently. The core idea of this proof is (Figure 11 (a)) that B is alive if and only if one of two things is true:

1. $\{e^\circ\} \cup \pi^\circ$ incident to $f^{des}_u(\nu)$, or
2. B contains some (maximal) pre-stored biconnected B^* incident to $f^{des}_u(\nu)$.

We test case 1 with conventional methods in $O(\log n)$ time. For case 2, we identify a special edge e^* on π°. We show that such a pre-stored biconnected component B^* exists only if $e^* \in B^*$. Since the top tree has height $O(\log n)$, e^* may be contained in $O(k \log n)$ pre-stored biconnected components. We find these in $O(1)$ amortized constant time per biconnected component. I.e., our invariants allow us to apply Theorem 3 to identify such a B^* in $O(k \log n)$ *worst case total time*. Given the maximal B^* that contains e^*, we test whether B^* is incident to $f^{des}_u(\nu)$ using an additional $O(\log n)$ time (searching over its boundary).

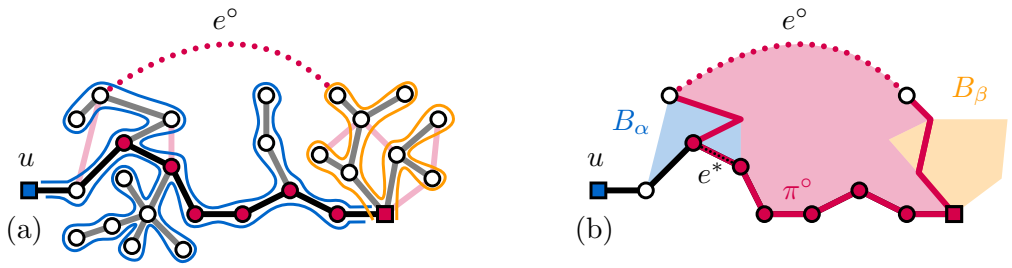


Figure 8 (a) Suppose that we merge a path cluster α (blue) with a point cluster β (yellow) to create a new point cluster ν (we call this an end merge). We are interested in all "new" biconnected components in $G[\nu]$. Every such new biconnected component must contain an edge e° in $G[\alpha \cup \beta] \backslash (G[\alpha] \cup G[\beta])$. (b) Consider the path π° along the spanning tree that connects the two endpoints of e° (red). This creates a cycle, and thus a new biconnected component B in the graph $G[\nu]$. The component B consists of this cycle, but may additionally "absorb" biconnected components B_α in $G[\alpha]$ and B_β in $G[\beta]$.

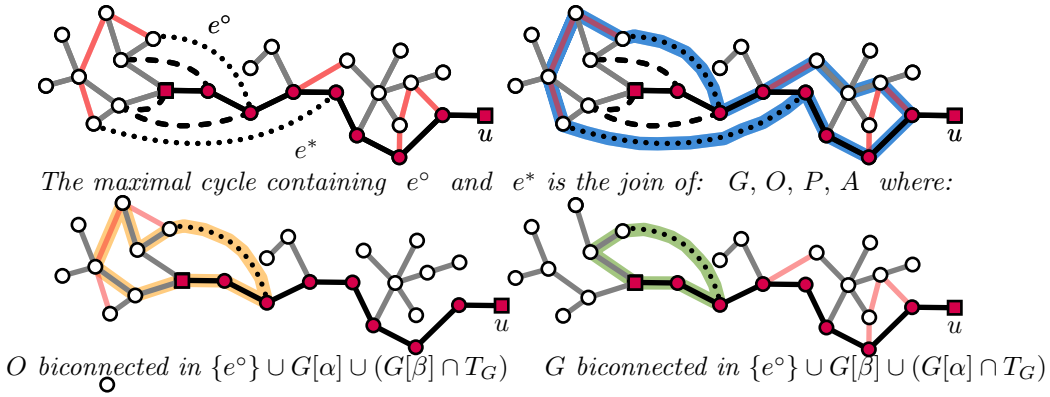


Figure 9 A cluster ν with as children a point cluster α and a path cluster β. There may be many edges in $G[\alpha \cup \beta]$. These edges are all contained in some maximal cycle which we show in blue. For the edge e°, we show the biconnected component G in $\{e^\circ\} \cup G[\beta] \cup (T_G \cap G[\alpha])$ and O in $\{e^\circ\} \cup G[\alpha] \cup (T_G \cap G[\beta])$. Similarly, for the edge e^* we show the biconnected components P and A. On an intuitive level, the maximal blue cycle is their "join".

Thus, we identify for every such edge e° in $O(k \log n)$ time whether it created a new relevant and alive biconnected component $B \in BC^*_u(\nu)$. If so, we create an object representing $B \in BC^*_u(\nu)$ in $O(1)$ additional time.

4.3 Invariant 3: storing the border of B

For each $B \in BC^*_u(\nu)$, we show in the full version that we can not only compute B but also its border to store in Invariant 3. The core idea (Figure 11 (b)) is that when merging two clusters α and β we can "project" e° onto $G[\alpha]$ and $G[\beta]$ to find the border of B in the respective graphs. However, two complications arise:

Firstly, we observed earlier that B may be the result of combining the cycle $e^\circ \cup \pi^\circ$ with a biconnected component B_α in $G[\alpha]$ (additionally, some biconnected component B_β in $G[\beta]$). Whenever that is the case, the eastern border of B is not the border of the path $\pi^\circ \cap \pi(\nu)$, but it rather gets "extended" to be the eastern border of B_α. This complication is relatively easy to solve: In the previous subsection we explained that we can find B_α. Since we merge the trees bottom-up, we have already restored Invariant 3 for the node α. Thus, we obtain the eastern border of B_α in $O(1)$ additional time and set it to be the eastern border of B.

Secondly, a merge can contain up to four clusters, not only two. We perform an extensive case analysis where we show that we can construct the border of B in $G[\nu]$ by pairwise joining projected borders (for an example, see Figure 9).

4.4 Finalising our argument

Up to this point, we showed that we can maintain our data structure and its invariants in $O(k^2 \log^2 n)$ time per operation in G. The integer k has two functions: first, it upper bounds the number of elements in $BC^*_u(\nu)$ for any $u \in V$ and $\nu \in \mathcal{T}^{op}_G$. Second we assumed that to maintain $BC^*_u(\nu)$, for each merge we need to inspect at most $O(k)$ edges $e^\circ \in G[\alpha \cup \beta] \backslash (G[\alpha] \cup G[\beta])$ in $O(k \log n)$ time each. In the full version we prove Theorem 1 by proving that such a k exists and that it is constant.

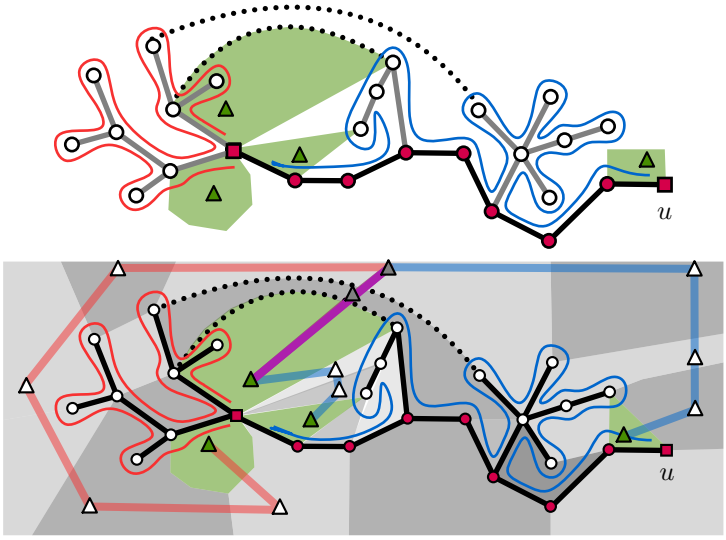


Figure 10 An End merge between a point cluster α and a path cluster β to create a new path cluster ν. We show two Euler tours $\mathring{\alpha}$ and $\mathring{\beta}^{\uparrow}$ in blue and red. The tour $\mathring{\alpha}$ corresponds to the red path in the dual between two faces. The tour $\mathring{\beta}^{\uparrow}$ to the blue path. The purple path is their meet. Any edge $e^{\circ} \in G[\alpha \cup \beta] \backslash (G[\alpha] \cup G[\beta])$ intersects both $\mathring{\alpha}$ and $\mathring{\beta}^{\uparrow}$ (or $\mathring{\alpha}$ and $\mathring{\beta}^{\downarrow}$) and must thus lie on the purple path (or an alternative meet in the dual). The first edge on this path is $e^{\star}$, as any further edge cannot be incident to the face $f_u^{des}(\nu)$.

Proving k exists and that it is a constant. Consider any edge $e^{\circ} \in G[\alpha \cup \beta] \backslash (G[\alpha] \cup G[\beta])$. We observe that e° must intersect a tourpath of α and a tourpath of β. For any fixed pair of tourpaths $(\mathring{\alpha}, \mathring{\beta})$ we consider our co-tree (i.e., the spanning tree on the dual of G). The Euler tour around $\mathring{\alpha}$ is a path in the dual. Similarly, the Euler tour around $\mathring{\beta}$ is a path and their common intersection is a meet in the dual (Figure 10). All edges that intersect both $\mathring{\alpha}$ and $\mathring{\beta}$ must lie on this meet (this concept is similar to the *edge bundles* by Laporte et al. in [30]). Thus, by Theorem 2, we can obtain for each pair $(\mathring{\alpha}, \mathring{\beta})$ this meet in $O(\log n)$ time.

We show that we may restrict our attention to the first edge $e^{\star}$ of this bundle (i.e. the first edge encountered on the meet). Indeed, the cycle formed by T_G and $e^{\star}$ encloses all other edges e° of the bundle in a face $f^{\star}$. We are only interested in biconnected components that are alive (incident to the face $f_u^{des}(\nu)$). For all other edges e° their respective biconnected components B either include $e^{\star}$, or are contained in $f^{\star}$ and can therefore not be incident to $f_u^{des}(\nu)$. It follows that whenever we create a new node ν with boundary vertex u, for every pair of tourpaths of its children, there is a unique edge $e^{\star}$ which can form a new biconnected component in $BC_u^*(\nu)$. Each merge involves at most 4 children that collectively have at most 6 tourpaths, and thus k is upper bound by 6 choose 2 (which is 15).

One special case. The above proof strategy applies to almost all our merge types. There exists however, one special case. During a four-way merge, whenever α and β are path clusters around a central vertex m, there exists no such "maximal" edge $e^{\star}$ (Figure 11 (c)). Thus, we cannot identify the biconnected components created by the edge bundle between $G[\alpha]$ and $G[\beta]$. We observe that any such component is only useful for answering biconnectivity queries between u and v if it connects the edges e_1 and e_2 of $\pi(\nu)$ incident to m. Indeed, if removing m separates e_1 and e_2 then it must also separate u and v (which are the boundary vertices of the root). We test if removing m separates e_1 and e_2 in G in $O(\log n)$ time. Note

that testing if removing a vertex separates two other vertices is already possible in $O(\log^2 n)$ time using [23] (by splitting m along the right corners and testing for connectivity). However to have $O(\log^2 n)$ total update time, we want $O(\log n)$ update time per merge. In the full version we open their black box slightly to test this in $O(\log n)$ time instead.

4.5 Conclusion

We have presented an efficient data structure for 2-vertex connectivity in dynamic planar graphs subject to edge-insertions, edge-deletions, contractions, splits, and local changes to the embedding in the form of flips. In this process, and with this result, we may have taken a first step towards worst-case deterministic fully-dynamic planarity testing in polylogarithmic time; one of the fundamental research questions in dynamic graph drawing.

Our technique is to consider the planar top-tree, and our contribution includes insights into important features and information to store in the top-tree clusters. Top-tree clusters can be seen as sketches of subgraphs, and thus, these insights about subgraph features for this computational problem, may have independent interest. Indeed, the concepts of *designated face* and *alive*, may be useful when constructing a dynamic data structure for fully-dynamic planar 3-vertex connectivity, or even for higher vertex connectivity in dynamic planar graphs.

Looking forward, there is a multitude of planar graph problems which would be interesting to examine in the worst-case deterministic fully-dynamic setting. Examples of such problems include $k \geq 3$-vertex connectivity, k-edge connectivity, arboricity decomposition, and various questions about constrained planar drawings of directed graphs. Approaching any of these related questions would require new ideas and techniques.

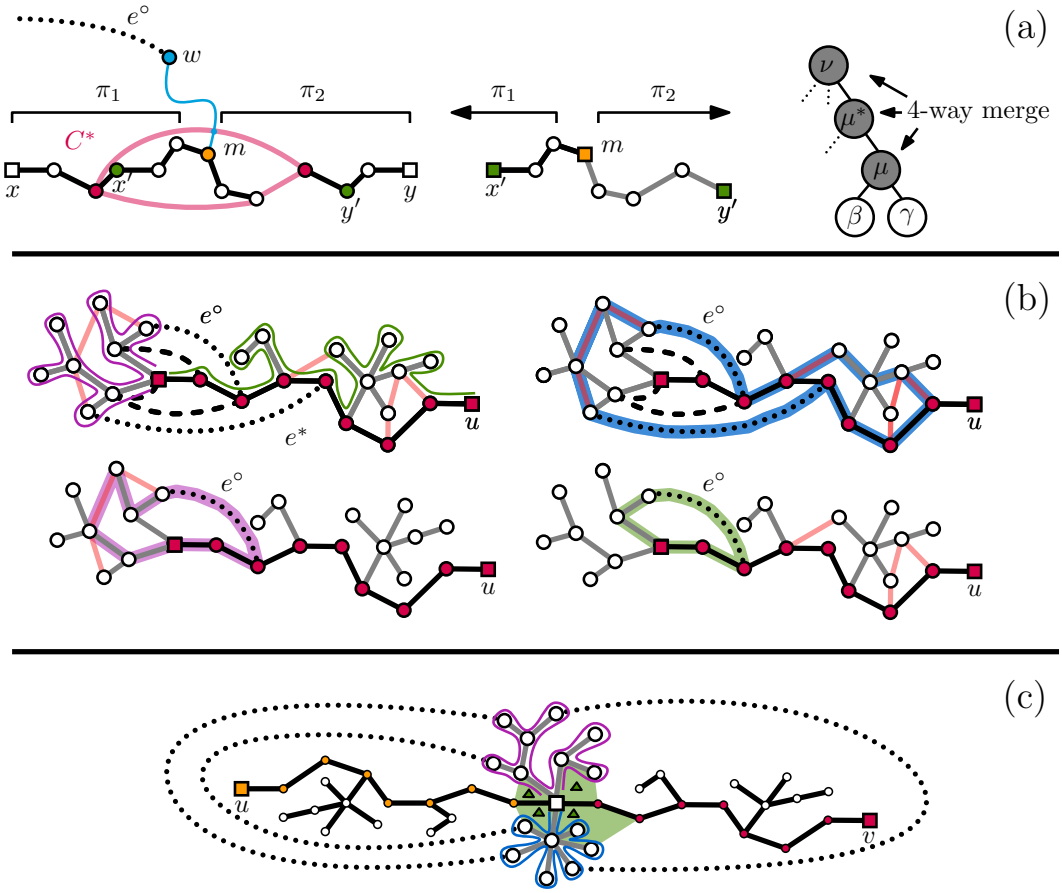


Figure 11 Three challenges that are encountered in our paper and described in our overview.
(a) Let e° have some endpoint w and consider the path in T_G to some vertex m. Let B be the biconnected component of $G[\nu]$ that contains e°. There exists some child μ of ν where m is the central vertex of the merge. If B contains some pre-stored biconnected component B^* (red) then B includes either the edge e_1 or e_2 in $G[\mu]$ incident to m.
(b) Consider an End Merge and an edge e° intersecting the purple and green Euler tours. The edge e° is part of a biconnected component with the blue cycle as its outer cycle. We find for e°, however, only the purple and green cycles in $G[\alpha]$ and $G[\beta]$ separately. We smartly join these cycles together with the cycles for e^* to get the out-most cycle bounding the new biconnected component B^*.
(c) In a four-way merge, the edges incident to the outer face of the embedding may be arbitrary edges in the edge bundle between the two path clusters. Neither edges incident to the outer face are incident to $f_u^{des}(\nu)$ or $f_v^{des}(\nu)$. Since we have no techniques for finding these edges, we instead test if the central vertex separates (u, v).

References

1 Anders Aamand, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. *ArXiV*, 2021.

2 Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms (TALG)*, 2005. doi:10.1145/1103963.1103966.

3 Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 1996. doi:10.1007/BF01961541.

4 David Eppstein. Dynamic generators of topologically embedded graphs. In *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2003. doi:10.5555/644108.644208.

5 David Eppstein, Zvi Galil, Giuseppe Italiano, and Thomas Spencer. Separator-based sparsification ii: Edge and vertex connectivity. *SIAM Journal on Computing*, 1999. doi:10.1137/S0097539794269072.

6 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 1997. doi:10.1145/265910.265914.

7 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. i. planary testing and minimum spanning trees. *Journal of Computer and System Sciences (JCSS)*, 1996. doi:10.1006/jcss.1996.0002.

8 David Eppstein, Giuseppe F Italiano, Roberto Tamassia, Robert Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *Journal of Algorithms*, 1992. doi:10.1016/0196-6774(92)90004-V.

9 David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery R. Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *Journal of Algorithms*, 1992.

10 Greg Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 1985.

11 Greg Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 1997. doi:10.1137/S0097539792226825.

12 Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *Journal of the ACM (JACM)*, 1999. doi:10.1145/300515.300517.

13 Dora Giammarresi and Giuseppe F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 1996. doi:10.1007/BF01955676.

14 Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In Dániel Marx, editor, *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2021. doi:10.1137/1.9781611976465.132.

15 Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theoretical Computer Science (TCS)*, 1998. doi:10.1016/S0304-3975(97)00291-0.

16 Monika R. Henzinger and Han La Poutré. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. In *European Symposium on Algorithms (ESA)*, 1995.

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

18 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 1999. doi:10.1145/320211.320215.

19 Monika Rauch Henzinger and Mikkel Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures and Algorithms*, 1997. doi:10.1002/(SICI)1098-2418(199712)11:4<369::AID-RSA5>3.0.CO;2-X.

20 John Hershberger, Monika Rauch, and Subhash Suri. Data structures for two-edge connectivity in planar graphs. *Theoretical Computer Science (TCS)*, 1994. doi:10.1016/0304-3975(94)90156-2.

21 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 2001. doi:10.1145/502090.502095.

22 Jacob Holm, Giuseppe Italiano, Adam Karczmarz, Jakub Lacki, and Eva Rotenberg. Decremental SPQR-trees for Planar Graphs. In *European Symposium on Algorithms (ESA)*, 2018. doi:10.4230/LIPIcs.ESA.2018.46.

23 Jacob Holm and Eva Rotenberg. Dynamic planar embeddings of dynamic graphs. *Theory of Computing Systems (TCS)*, 2017. doi:10.1007/s00224-017-9768-7.

24 Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *PACM Symposium on Theory of Computing (STOC)*, 2020. doi:10.1145/3357713.3384249.

25 Jacob Holm and Eva Rotenberg. Worst-case polylog incremental SPQR-trees: Embeddings, planarity, and triconnectivity. In Shuchi Chawla, editor, *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2020. doi:10.1137/1.9781611975994.146.

26 Jacob Holm and Eva Rotenberg. Good r-divisions imply optimal amortised decremental biconnectivity. *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2021. doi:10.4230/LIPIcs.STACS.2021.42.

27 Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic bridge-finding in $\widetilde{O}(\log^2 n)$ amortized time. In *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2018. doi:10.1137/1.9781611975031.3.

28 John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 1974. doi:10.1145/321850.321852.

29 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2017. doi:10.1137/1.9781611974782.32.

30 Giuseppe F. Italiano, Johannes A. La Poutré, and Monika Rauch. Fully dynamic planarity testing in planar embedded graphs (extended abstract). In Thomas Lengauer, editor, *European Symposium on Algorithms (ESA*, 1993. doi:10.1007/3-540-57273-2_57.

31 Bruce Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013. doi:10.1137/1.9781611973105.81.

32 Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster Worst Case Deterministic Dynamic Connectivity. In *European Symposium on Algorithms (ESA)*, 2016. doi:10.4230/LIPIcs.ESA.2016.53.

33 Jakub Łacki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n\log\log n)$ time. In *European Symposium on Algorithms (ESA)*, 2011. doi:10.1007/978-3-642-23719-5_14.

34 Jakub Łacki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. In *Symposium on Theoretical Aspects of Computer Science, (STACS)*, 2015. doi:10.4230/LIPIcs.STACS.2015.608.

35 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Symposium on Foundations of Computer Science (FOCS)*, 2017. doi:10.1109/FOCS.2017.92.

36 Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In Frank Thomson Leighton and Michael T. Goodrich, editors, *ACM Symposium on Theory of Computing (STOC)*, 1994. doi:10.1145/195058.195439.

37 Johannes A. La Poutré. Maintenance of 2- and 3-edge-connected components of graphs II. *SIAM Journal of Computing*, 2000. doi:10.1137/S0097539793257770.

38 Johannes A. La Poutré, Jan van Leeuwen, and Mark H. Overmars. Maintenance of 2- and 3-edge- connected components of graphs I. *Discrete Mathematics*, 1993. doi:10.1016/0012-365X(93)90376-5.

39 Johannes A. La Poutré and Jeffery R. Westbrook. Dynamic 2-connectivity with backtracking. *SIAM Journal of Computing*, 1998. doi:10.1137/S0097539794272582.

40 Mihai Pătraşcu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 2006. doi:10.1137/S0097539705447256.

41 Robert Endre Tarjan and Renato Fonseca F Werneck. Self-adjusting top trees. In *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2005. doi:10.5555/1070432.1070547.

42 Mikkel Thorup. Decremental dynamic connectivity. In *ACM-SIAM Symposium on Discrete algorithms (SODA)*, 1997.

43 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *ACM Symposium on Theory of Computing (STOC)*, 2000. doi:10.1145/335305.335345.

44 Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*. Springer Berlin Heidelberg, 2016. doi:10.1137/1.9781611973105.126.