# Maintaining Triconnected Components under Node Expansion

**Simon D. Fink**
Faculty of Informatics and Mathematics, University of Passau, Germany
Algorithms and Complexity Group, Technische Universität Wien, Austria

**Ignaz Rutter**
Faculty of Informatics and Mathematics, University of Passau, Germany

## Abstract

SPQR-trees model the decomposition of a biconnected graph into triconnected components. In this paper, we study the problem of dynamically maintaining an SPQR-tree while expanding vertices into arbitrary biconnected graphs. This allows us to efficiently merge two SPQR-trees by identifying the edges incident to two vertices with each other. We do this working along an axiomatic definition lifting the SPQR-tree to a stand-alone data structure that can be modified independently from the graph it might have been derived from. Making changes to this structure, we can now observe how the graph represented by the SPQR-tree changes, instead of having to reason which updates to the SPQR-tree are necessary after a change to the represented graph.

Using efficient expansions and merges allows us to improve the runtime of the Synchronized Planarity algorithm by Bläsius et al. [4] from $O(m^2)$ to $O(m \cdot \Delta)$, where $\Delta$ is the maximum degree of a vertex with a synchronization constraint that requires two vertices to have the same rotation under a given bijection. This also reduces the time for solving several related constrained planarity problems, e.g. for Clustered Planarity from $O((n+d)^2)$ to $O(n + d \cdot \Delta)$, where $d$ is the total number of crossings between cluster borders and edges and $\Delta$ is the maximum number of edge crossings on a single cluster border.[1]

**Keywords and phrases** SPQR-Tree, Dynamic Algorithm, Clustered Planarity

**Digital Object Identifier** 10.57717/cgt.v3i2.44

**Related Version** A preliminary version of this paper has appeared at CIAC'23 [9]. This journal version provides full proofs for all statements together with the detailed changes made to the algorithm. It also contains additional figures together with an overview over the runtime improvements for various other constrained planarity variants.

**Funding** Funded by DFG-grant RU-1903/3-1.

## 1 Introduction

The SPQR-tree is a data structure that represents the decomposition of a graph at its *separation pairs*, that is the pairs of vertices whose removal disconnects the graph. The components obtained by this decomposition are called *skeletons*. SPQR-trees form a central component of many graph visualization techniques and are used for, e.g., planarity testing and variations thereof [7, 14] and for computing embeddings and layouts [12, 19]. Initially, SPQR-trees were devised by Di Battista and Tamassia for incremental planarity testing [7]. Their use was quickly expanded to other on-line problems [6] and to the fully-dynamic setting, that is allowing insertion and deletion of vertices and edges in $O(\sqrt{n})$ time [8], where $n$ is the number of vertices in the graph.

[1] There is a linear reduction from Clustered to Synchronized Planarity that converts edges crossing cluster boundaries to edges incident to synchronized vertices. Thereby, for an instance of the former problem, both definitions of $\Delta$ yield the same value.

In this paper, we consider an incremental setting where we allow a single operation that expands a vertex $v$ into an arbitrary biconnected graph $G_\nu$. The approach of Eppstein et al. [8] allows this in $O((\deg(v) + |G_\nu|) \cdot \sqrt{n})$ time by only representing parts of triconnected components.[2] We improve this to $O(\deg(v) + |G_\nu|)$ using an algorithm that is much simpler and explicitly yields full triconnected components, which will become important for our applications later. In addition, our approach also allows to efficiently merge two SPQR-trees as follows. Given two biconnected graphs $G_1, G_2$ containing vertices $v_1, v_2$, respectively, together with a bijection between their incident edges, we construct a new graph $G$ by replacing $v_1$ with $G_2 - v_2$ in $G_1$, identifying edges using the given bijection. Given the SPQR-trees of $G_1$ and $G_2$, we show that the SPQR-tree of $G$ can be found in $O(\deg(v_1))$ time. More specifically, we present a data structure that supports the following operations: $\texttt{InsertGraph}_{\texttt{SPQR}}$ expands a single vertex in time linear in the size of the expanded subgraph, $\texttt{Merge}_{\texttt{SPQR}}$ merges two SPQR-trees in time linear in the degree of the replaced vertices, `IsPlanar` indicates whether the currently represented graph is planar in constant time, and `Rotation` yields one of the two possible planar *rotations* (cyclic orders of incident edges) of a vertex in a triconnected skeleton in constant time. Furthermore, our data structure can be adapted to yield consistent planar embeddings for all triconnected skeletons and to test for the existence of three distinct paths between two arbitrary vertices with an additional factor of $\alpha(n)$ for all operations, where $\alpha$ is the inverse Ackermann function.

The main idea of our approach is that the subtree of the SPQR-tree affected by expanding a vertex $v$ has size linear in the degree of $v$, but may contain arbitrarily large skeletons. In a "non-normalized" version of an SPQR-tree, the affected cycle ('S') skeletons can easily be split to have a constant size, while we develop a custom splitting operation to limit the size of triconnected ('R') skeletons. This limits the size of the affected structure to be linear in the degree of $v$ and allows us to perform the expansion efficiently.

In addition to the description of this data structure, the technical contribution of this paper is twofold: First, we develop an axiomatic definition of the decomposition at separation pairs, putting the SPQR-tree as "mechanical" data structure into focus instead of relying on and working along a given graph structure. As a result, we can deduce the represented graph from the data structure instead of computing the data structure from the graph. This allows us to make more or less arbitrary changes to the data structure (respecting its consistency criteria) and observe how the graph changes, instead of having to reason which changes to the graph require which updates to the data structure.

Second, we explain how our data structure can be used to improve the runtime of the algorithm by Bläsius et al. [4] for solving the SYNCHRONIZED PLANARITY problem from $O(m^2)$ to $O(m \cdot \Delta)$, where $\Delta$ is the maximum *pipe* degree. Synchronization constraints in the form of pipes are a core component of this problem: we seek a planar embedding of a graph such that for each pipe matching up two distinct vertices, their rotation lines up under a given bijection. SYNCHRONIZED PLANARITY can be used to model and solve a vast class of different kinds of constrained planarity; see Table 1 for an overview of problems benefiting from our speedup. Among them is the notorious CLUSTERED PLANARITY, whose complexity was open for 30 years before Fulek and Tóth gave an algorithm with runtime $O((n + d)^8)$ in 2019 [11], where $d$ is the total number of crossings between cluster borders and edges. Shortly thereafter, Bläsius et al. [4] gave a solution in $O((n + d)^2)$ time. We improve this to $O(n + d \cdot \Delta)$, where $\Delta$ is the maximum number of edge crossings on a single cluster border.

[2] Unfortunately, the recent improvements by Holm and Rotenberg are not applicable here, as they maintain triconnectivity in an only incremental setting [15], while maintaining only planarity information in the fully-dynamic setting [14].

| Problem | Running Times | | |
|---|---|---|---|
| | before [4] | using [4] | with this paper |
| ATOMIC EMBEDDABILITY / SYNCHRONIZED PLANARITY | $O(m^8)$ [11] | $O(m^2)$ | $O(m \cdot \Delta)$ |
| CLUSTERED PLANARITY | $O((n+d)^8)$ [11] | $O((n+d)^2)$ | $O(n + d \cdot \Delta)$ |
| CONNECTED SEFE | $O(n^{16})$ [11] bicon: $O(n^2)$ [3] | $O(n^2)$ | $O(n \cdot \Delta)$ |
| PARTIALLY PQ-CONSTRAINED PLANARITY | bicon: $O(m)$ [3] | $O(m^2)$ | $O(m \cdot \Delta)$ |
| ROW-COLUMN INDEPENDENT NODETRIX PLANARITY | bicon: $O(n^2)$ [17] | $O(n^2)$ | $O(n \cdot \Delta)$ |
| STRIP PLANARITY | $O(n^8)$ [2, 11] fixed emb: $O(n^2)$ [2] | $O(n^2)$ | $O(n \cdot \Delta)$ |

**Table 1** The best known running times for various constrained planarity problems before SYNCHRONIZED PLANARITY [4] was published; using it as described in [4]; and using it together with the speed-up from this paper. Running times prefixed with "bicon" only apply for certain problem instances which expose some form of biconnectivity. The variables $n$ and $m$ refer to the number of vertices and edges of the problem instance, respectively. The variable $d$ refers to the number of edge-cluster boundary crossings in CLUSTERED PLANARITY instances, while $\Delta$ refers to the maximum pipe degree in the corresponding SYNCHRONIZED PLANARITY instances. This is bounded by the maximum number of edges crossing a single cluster border or the maximum vertex degree in the input instance, depending on the problem.

This work is structured as follows. After preliminaries in Section 2, we describe the skeleton decomposition and show how it relates to the SPQR-tree in Section 3. Section 4 extends this data structure by the capability of splitting triconnected components. In Section 5, we use this to ensure the affected part of the SPQR-tree is small when we replace a vertex with a new graph. Section 6 shows how our results can be used to reduce the time required for solving SYNCHRONIZED PLANARITY, CLUSTERED PLANARITY and related constrained planarity variants.

## 2 Preliminaries

In the context of this work, $G = (V, E)$ is a (usually biconnected and loop-free) multi-graph with $n$ vertices $V$ and $m$ (possibly parallel) edges $E$. For a vertex $v$, we denote its open neighborhood (excluding $v$ itself) by $N(v)$. For a bijection or matching $\phi$ we call $\phi(x)$ the *partner* of an element $x$. We use $A \uplus B$ to denote the union of two disjoint sets $A, B$.

A separating $k$-set is a set of $k$ vertices whose removal increases the number of connected components. Separating 1-sets are called *cutvertices*, while separating 2-sets are called *separation pairs*. A connected graph is *biconnected* if it does not have a cutvertex. A biconnected graph is *triconnected* if it does not have a separation pair. Maximal biconnected subgraphs are called *blocks*. Each separation pair divides the graph into *bridges*, the maximal subgraphs which cannot be disconnected by removing or splitting the vertices of the separation pair. A *bond* is a graph that consists solely of two *pole* vertices connected by multiple parallel edges, a *polygon* is a simple cycle, while a *rigid* is any simple triconnected graph. A *wheel* is a cycle with an additional central vertex connected to all other vertices.

Finally, the *expansion* that is central to this work is formally defined as follows. Let $G_\alpha, G_\beta$ be two graphs where $G_\alpha$ contains a vertex $u$ and $G_\beta$ contains $|N(u)|$ marked vertices, together with a bijection $\phi$ between the neighbors of $u$ and the marked vertices in $G_\beta$. With $G_\alpha[u \rightarrow_\phi G_\beta]$ we denote the graph that is obtained from the disjoint union of $G_\alpha, G_\beta$ by identifying each neighbor $x$ of $u$ with its respective marked vertex $\phi(x)$ in $G_\beta$ and removing $u$, i.e. the graph $G_\alpha$ where the vertex $u$ was expanded into $G_\beta$; see Figure 4 for an example.

## 3 Skeleton decompositions

Classically, SPQR-trees are defined as a tree derived from the decomposition of a given graph at separation pairs. Each node of the SPQR-tree is associated with a skeleton graph that represents a subgraph of the original graph that cannot be decomposed further. For our purposes, we invert this approach and define a skeleton structure that starts from arbitrary skeleton graphs equipped with explicit mappings and conditions that ensure that, together, they define a graph. The SPQR-tree for a given graph $G$ is then simply the skeleton structure that happens to define $G$. This has significant advantages in a dynamic context. Instead of tracking how changes in $G$ affect its decomposition and thus require updates to its SPQR-tree, we can directly apply modifications to the skeletons that preserve all mappings and conditions. It then suffices to check that the resulting skeletons define the desired modification of $G$.

A *skeleton structure* $\mathcal{S} = (\mathcal{G}, \mathrm{origV}, \mathrm{origE}, \mathrm{twinE})$ that *represents* a graph $G_\mathcal{S} = (V, E)$ consists of a set $\mathcal{G}$ of disjoint *skeleton* graphs together with three total, surjective mappings $\mathrm{twinE}, \mathrm{origE}$, and $\mathrm{origV}$ that satisfy the following conditions:

- Each skeleton $G_\mu = (V_\mu, E_\mu^{\mathrm{real}} \uplus E_\mu^{\mathrm{virt}})$ in $\mathcal{G}$ is a multi-graph where each edge is either in $E_\mu^{\mathrm{real}}$ and thus called *real* or in $E_\mu^{\mathrm{virt}}$ and thus called *virtual*.
- Bijection $\mathrm{twinE} : E^{\mathrm{virt}} \rightarrow E^{\mathrm{virt}}$ matches all virtual edges $E^{\mathrm{virt}} = \bigcup_\mu E_\mu^{\mathrm{virt}}$ such that $\mathrm{twinE}(e) \neq e$ and $\mathrm{twinE}^2 = \mathrm{id}$.
- Surjection $\mathrm{origV} : \bigcup_\mu V_\mu \rightarrow V$ maps all skeleton vertices to graph vertices.
- Bijection $\mathrm{origE} : \bigcup_\mu E_\mu^{\mathrm{real}} \rightarrow E$ maps all real edges to the graph edge set $E$.

Note that each vertex and each edge of each skeleton is in the domain of exactly one of the three mappings. As the mappings are surjective, $V$ and $E$ are exactly the images of origV and origE. For each vertex $v \in G_\mathcal{S}$, the skeletons that contain a vertex $v'$ with $\mathrm{origV}(v') = v$ are called the *allocation skeletons* of $v$. We call $v'$ an *allocation vertex* of $v$. Furthermore, let $T_\mathcal{S}$ be a graph such that each node $\mu$ of $T_\mathcal{S}$ corresponds to a skeleton $G_\mu$ of $\mathcal{G}$ and two nodes of $T_\mathcal{S}$ are adjacent if their skeletons contain a pair of virtual edges matched with each other by twinE. We call a skeleton structure a *skeleton decomposition* if it satisfies the following conditions:

**1 (bicon)** Each skeleton is biconnected.
**2 (tree)** Graph $T_\mathcal{S}$ is simple, loop-free, connected and acyclic, i.e., a tree.
**3 (orig-inj)** For each skeleton $G_\mu$, the restriction $\mathrm{origV}|_{V_\mu}$ is injective.
**4 (orig-real)** For each real edge $uv$, the endpoints of $\mathrm{origE}(uv)$ are $\mathrm{origV}(u)$ and $\mathrm{origV}(v)$.
**5 (orig-virt)** Let $uv$ and $u'v'$ be two virtual edges with $uv = \mathrm{twinE}(u'v')$. For their respective skeletons $G_\mu$ and $G'_\mu$ (where $\mu$ and $\mu'$ are adjacent in $T_\mathcal{S}$), it is $\mathrm{origV}(V_\mu) \cap \mathrm{origV}(V_{\mu'}) = \mathrm{origV}(\{u, v\}) = \mathrm{origV}(\{u', v'\})$.
**6 (subgraph)** The allocation skeletons of any vertex of $G_\mathcal{S}$ form a connected subgraph of $T_\mathcal{S}$.

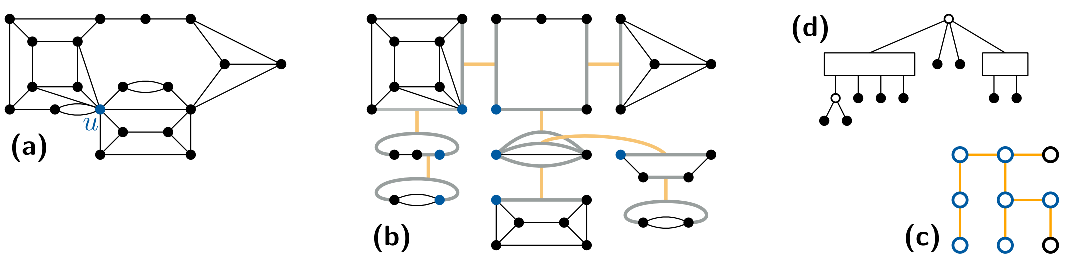

**Figure 1** Different views on the skeleton decomposition $\mathcal{S}$. **(a)** The graph $G_{\mathcal{S}}$ with a vertex $u$ marked in blue. **(b)** The skeletons of $\mathcal{G}$. Virtual edges are drawn in gray with their matching twinE being shown in orange. The allocation vertices of $u$ are marked in blue. **(c)** The tree $T_{\mathcal{S}}$. The allocation skeletons of $u$ are marked in blue. **(d)** The embedding tree of vertex $u$ as described in Section 6.1. P-nodes are shown as white disks, Q-nodes are shown as large rectangles. The leaves of the embedding tree correspond to the edges incident to $u$.

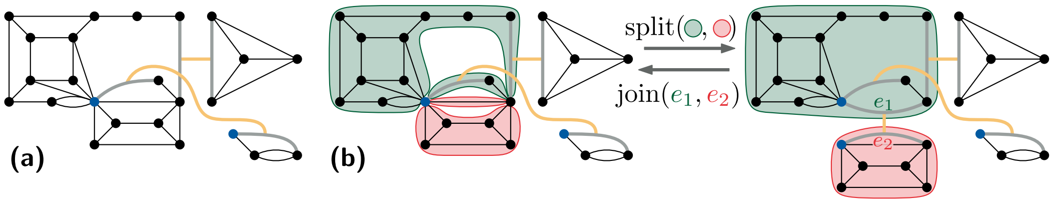

**Figure 2** **(a)** A skeleton decomposition that represents graph $G_{\mathcal{S}}$ from Figure 1a with the two allocation vertices of graph vertex $u$ marked in blue. **(b)** The result of applying `SplitSeparationPair` to separate the bridges highlighted in green from those in red. Applying the converse operation `JoinSeparationPair` on the resulting edges yields the original skeleton.

Figure 1 shows an example of $\mathcal{S}$, $G_{\mathcal{S}}$, and $T_{\mathcal{S}}$. We call a skeleton decomposition with only one skeleton $G_\mu$ *trivial*. In this case, $G_\mu$ is isomorphic to $G_{\mathcal{S}}$, and origE and origV are actually bijections between the edges and vertices of both graphs.

To model the decomposition into triconnected components, we define the operations `SplitSeparationPair` and its converse, `JoinSeparationPair`, on a skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE})$; see also Figure 2. For `SplitSeparationPair`, let $u, v$ be a separation pair of skeleton $G_\mu$ and let $(A, B)$ be a non-trivial bipartition of the bridges between $u$ and $v$ in $G_\mu$.[3] Applying `SplitSeparationPair`$(\mathcal{S}, (u, v), (A, B))$ yields skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$ as follows. In $\mathcal{G}'$, we replace $G_\mu$ by two skeletons $G_\alpha, G_\beta$, where $G_\alpha$ is obtained from $G_\mu[A]$ by adding a new virtual edge $e_\alpha$ between $u$ and $v$. The same respectively applies to $G_\beta$ with $G_\mu[B]$ and $e_\beta$. We set $\text{twinE}'(e_\alpha) = e_\beta$ and $\text{twinE}'(e_\beta) = e_\alpha$. Note that origV maps the endpoints of $e_\alpha$ and $e_\beta$ to the same vertices. All other skeletons and their mappings remain unchanged.

For `JoinSeparationPair`, consider virtual edges $e_\alpha, e_\beta$ with $\text{twinE}(e_\alpha) = e_\beta$ and let $G_\beta \neq G_\alpha$ be their respective skeletons. Applying `JoinSeparationPair`$(\mathcal{S}, e_\alpha)$ yields a skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$ as follows. In $\mathcal{G}'$, we merge $G_\alpha$ with $G_\beta$ to form a new skeleton $G_\mu$ by identifying the endpoints of $e_\alpha$ and $e_\beta$ that map to the same vertex of $G_{\mathcal{S}}$. Additionally, we remove $e_\alpha$ and $e_\beta$. All other skeletons and their mappings remain unchanged.

[3] Note that a bridge may consist of a single edge between $u$ and $v$ and that each bridge includes the vertices $u$ and $v$.

The main feature of both operations is that they leave the graph represented by the skeleton decomposition unaffected while splitting a node or contracting an edge in $T_{\mathcal{S}}$, which can be verified by checking the individual conditions.

▶ **Lemma 1.** *Applying* `SplitSeparationPair` *or* `JoinSeparationPair` *on a skeleton decomposition $\mathcal{S}$ yields a skeleton decomposition $\mathcal{S}'$ with an unchanged represented graph $G_{\mathcal{S}'} = G_{\mathcal{S}}$.*

**Proof.** We first check that all conditions still hold in the skeleton decomposition $\mathcal{S}'$ returned by `SplitSeparationPair`. As $(A, B)$ is a non-trivial bipartition, each set contains at least one bridge. Together with $e_\alpha$ (and $e_\beta$), this bridge ensures that $G_\alpha$ (and $G_\beta$) remain biconnected, satisfying condition 1 (bicon). The operation splits a node $\mu$ of $T_{\mathcal{S}}$ into two adjacent nodes $\alpha, \beta$, whose neighbors are defined exactly by the virtual edges in $A, B$, respectively. Thus, condition 2 (tree) remains satisfied. The mappings $\mathrm{origV}', \mathrm{origE}'$ and $\mathrm{twinE}'$ obviously still satisfy conditions 3 (orig-inj) and 4 (orig-real). We duplicated exactly two nodes, $u$ and $v$ of adjacent skeletons $G_\alpha$ and $G_\beta$. Because 3 (orig-inj) holds for $G_\mu$, $G_\alpha$ and $G_\beta$ share no other vertices that map to the same vertex of $G_{\mathcal{S}'}$. Thus, condition 5 (orig-virt) remains satisfied.

Condition 6 (subgraph) could only be violated if the subgraph of $T_{\mathcal{S}'}$ formed by the allocation skeletons of some vertex $z \in G_{\mathcal{S}'}$ was no longer connected. This could only happen if only one of $G_\alpha$ and $G_\beta$ were an allocation skeleton of $z$, while the other has a further neighbor that is also an allocation skeleton of $z$. Assume without loss of generality that $G_\alpha$ and the neighbor $G_\nu$ of $G_\beta$, but not $G_\beta$ itself, were allocation skeletons of $z$. Because $G_\nu$ and $G_\beta$ are adjacent in $T_{\mathcal{S}'}$ there are virtual edges $xy = \mathrm{twinE}'(x'y')$ with $xy \in G_\beta$ and $x'y' \in G_\nu$. The same virtual edges are also present in the input instance, only with the difference that $xy \in G_\mu$ and $\mu$ (instead of $\beta$) and $\nu$ are adjacent in $T_{\mathcal{S}}$. As the input instance satisfies condition 5 (orig-virt), it is $z \in \mathrm{origV}(V_\nu) \cap \mathrm{origV}(V_\mu) = \mathrm{origV}(\{x, y\}) = \mathrm{origV}(\{x', y'\})$. As $\mathrm{origV}(\{x, y\}) = \mathrm{origV}'(\{x, y\})$, this is a contradiction to $G_\beta$ not being an allocation skeleton of $z$.

Finally, the mapping origE remains unchanged and the only change to origV is to include two new vertices mapping to already existing vertices. Due to condition 4 (orig-real) holding for both the input and the output instance, this cannot affect the represented graph $G_{\mathcal{S}'}$.

Now consider the skeleton decomposition $\mathcal{S}'$ returned by `JoinSeparationPair`. Identifying distinct vertices of distinct connected components does not affect their biconnectivity, thus condition 1 (bicon) remains satisfied. The operation effectively contracts and removes an edge in $T_{\mathcal{S}}$, which does not affect $T_{\mathcal{S}'}$ being a tree satisfying condition 2 (tree). Note that condition 2 (tree) holding for the input instance also ensures that $G_\alpha$ and $G_\beta$ are two distinct skeletons. As the input instance also satisfies condition 5 (orig-virt), there are exactly two vertices in each of the two adjacent skeletons $G_\alpha$ and $G_\beta$, where origV maps to the same vertex of $G_{\mathcal{S}}$. These two vertices must be part of the twinE pair making the two skeletons adjacent, thus they are exactly the two pairs of vertices we identify with each other. Thus, $\mathrm{origV}|_{V_\mu}$ is still injective, satisfying condition 3 (orig-inj). As we modify no real edges and no other virtual edges, the mappings $\mathrm{origV}'$ and $\mathrm{origE}'$ obviously still satisfy condition 4 (orig-real). As the allocation skeletons of each graph vertex form a connected subgraph, joining two skeletons cannot change the intersection with any of their neighbors, leaving 5 (orig-virt) satisfied. Finally, contracting a tree edge cannot lead to any of the subgraphs of 6 (subgraph) becoming disconnected, thus the condition also remains satisfied. Again, no changes were made to origE, while condition 5 (orig-virt) makes sure that origV mapped the two pairs of merged vertices to the same vertex of $G_{\mathcal{S}}$. Thus, the represented graph $G_{\mathcal{S}'}$ remains unchanged. ◀

This gives us a second way of finding the represented graph by exhaustively joining all skeletons until there is only one left, obtaining the unique trivial skeleton decomposition:

▶ **Lemma 2.** *Exhaustively applying* `JoinSeparationPair` *to a skeleton decomposition* $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE})$ *yields a trivial skeleton decomposition* $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$ *where* $\text{origE}'$ *and* $\text{origV}'$ *define an isomorphism between* $G'_\mu$ *and* $G_{\mathcal{S}'}$*.*

**Proof.** As all virtual edges are matched, and the matched virtual edge always belongs to a different skeleton (condition 2 (tree) ensures that $T_\mathcal{S}$ is loop-free), we can always apply `JoinSeparationPair` on a virtual edge until there are none left. As $T_\mathcal{S}$ is connected, this means that we always obtain a tree with a single node, that is, an instance with only a single skeleton. As a single application of `JoinSeparationPair` preserves the represented graph, any chain of multiple applications also does. Note that $\text{origE}'$ is a bijection and the surjective $\text{origV}'$ is also injective on the single remaining skeleton due to condition 3 (orig-inj), thus it also globally is a bijection. Together with condition 4 (orig-real), this ensures that any two vertices $u$ and $v$ of $G'_\mu$ are adjacent if and only if $\text{origV}'(u)$ and $\text{origV}'(v)$ are adjacent in $G_{\mathcal{S}'}$. Thus $\text{origV}'$ is an edge-preserving bijection, that is an isomorphism. ◀

A key point about the skeleton decomposition and especially the operation `SplitSeparationPair` is that they model the decomposition of a graph at separation pairs. This decomposition was formalized as *SPQR-tree* by Di Battista and Tamassia [6, 7] and is unique for a given graph [16, 18]. Angelini et al. [1] describe a decomposition tree that is conceptually equivalent to our skeleton decomposition. They also present an alternative definition for the SPQR-tree as a decomposition tree satisfying further properties. We adopt this definition as follows, not requiring planarity of triconnected components and allowing virtual edges and real edges to appear within one skeleton (i.e., not including the so-called Q-nodes but merging them into their parents instead).

▶ **Definition 3.** *A skeleton decomposition* $\mathcal{S}$ *where any skeleton in* $\mathcal{G}$ *is either a polygon, a bond, or triconnected ("rigid"), and two skeletons adjacent in* $T_\mathcal{S}$ *are never both polygons or both bonds, is the unique SPQR-tree of* $G_\mathcal{S}$*.*

The main difference between the well-known ideas behind decomposition trees and our skeleton decomposition is that the latter allow an axiomatic access to the decomposition at separation pairs. For the skeleton decomposition, we employ a purely functional, "mechanical" data structure instead of relying on and working along a given graph structure. In our case, the represented graph is deduced from the data structure (i.e. from the SPQR-tree) instead of computing the data structure from the graph.

## 4 Extended skeleton decompositions

To modify (i.e., replace with another graph) a vertex $v$ in a graph $G$ while maintaining its SPQR-tree in the form of a skeleton decomposition, a naïve approach would be as follows: First join all allocation skeletons of $v$ to obtain a single skeleton $G_\mu$ that directly represents the structure of the neighborhood of $v$, second perform the replacement on the single allocation vertex of $v$ in $G_\mu$, third exhaustively decompose the resulting $G'_\mu$ to again obtain an SPQR-tree. The problem of this approach is that $G_\mu$ can be arbitrarily large; its size is the sum of the sizes of all allocation skeletons of $v$. Fortunately, polygons and bonds that contain an allocation vertex of $v$ can easily be decomposed along separation pairs into smaller parts, such that their total size is linear in the degree of $v$.

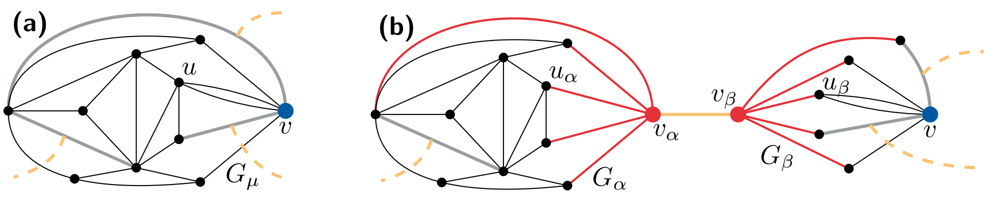

**Figure 3 (a)** A triconnected skeleton $G_\mu$ with a highlighted vertex $v$ incident to two gray virtual edges. **(b)** The result of applying `IsolateVertex` to isolate $v$ out of the skeleton. The red occupied edges in the old skeleton $G_\alpha$ form a star with center $v_\alpha$, while the red occupied edges in $G_\beta$ connect all neighbors of $v$ to form a star with center $v_\beta \neq v$. The centers $v_\alpha$ and $v_\beta$ are virtual and matched with each other. Neighbor $u$ of $v$ was split into vertices $u_\alpha$ and $u_\beta$. Conversely, applying `Integrate` on $(v_\alpha, v_\beta)$ in **(b)** again yields skeleton $G_\mu$ as in **(a)**.

The exception to this are triconnected skeletons, which cannot be split further using the operations we defined up to now. We thus define a further set of operations which allow us to efficiently isolate an allocation vertex $u$ of $v$ out of an arbitrary triconnected component by replacing it with a (“virtual”) placeholder vertex. This placeholder then points to a smaller component that contains the actual allocation vertex $u$; see Figure 3. Modification of the edges incident to the placeholder is disallowed, which is why we call them “occupied”. In the end, applying these splits guarantees that the allocation skeletons of $v$ have, in total, a size linear in the degree of $v$.

Formally, the structures needed to keep track of the triconnected components split in this way in an *extended* skeleton decomposition $\mathcal{S} = (\mathcal{G}, \mathrm{origV}, \mathrm{origE}, \mathrm{twinE}, \mathrm{twinV})$ are defined as follows. Skeletons now have the form $G_\mu = (V_\mu \cup\!\!\!\cdot\; V_\mu^{\mathrm{virt}},\ E_\mu^{\mathrm{real}} \cup\!\!\!\cdot\; E_\mu^{\mathrm{virt}} \cup\!\!\!\cdot\; E_\mu^{\mathrm{occ}})$. Bijection $\mathrm{twinV} : V^{\mathrm{virt}} \to V^{\mathrm{virt}}$ matches all *virtual vertices* $V^{\mathrm{virt}} = \bigcup_\mu V_\mu^{\mathrm{virt}}$, such that $\mathrm{twinV}(v) \neq v$, $\mathrm{twinV}^2 = \mathrm{id}$. The edges incident to virtual vertices are contained in $E_\mu^{\mathrm{occ}}$ and thus considered *occupied*; see Figure 3b. Similar to the virtual edges matched by twinE, any two virtual vertices matched by twinV induce an edge between their skeletons in $T_\mathcal{S}$. Condition 2 (tree) also equally applies to those edges induced by twinV, which in particular ensures that there are no parallel twinE and twinV tree edges in $T_\mathcal{S}$. Similarly, the connected subgraphs of condition 6 (subgraph) can also contain tree edges induced by twinV. All other conditions remain unchanged, but we add two further conditions to ensure that twinV is consistent:

**7 (stars)** For each $v_\alpha, v_\beta$ with $\mathrm{twinV}(v_\alpha) = v_\beta$, it is $\deg(v_\alpha) = \deg(v_\beta)$. All edges incident to $v_\alpha$ and $v_\beta$ are occupied and have distinct endpoints. Each occupied edge is adjacent to exactly one virtual vertex.

**8 (orig-stars)** Let $v_\alpha$ and $v_\beta$ again be two virtual vertices matched with each other by twinV. For their respective skeletons $G_\alpha$ and $G_\beta$ (where $\alpha$ and $\beta$ are adjacent in $T_\mathcal{S}$), it is $\mathrm{origV}(V_\alpha) \cap \mathrm{origV}(V_\beta) = \mathrm{origV}(N(v_\alpha)) = \mathrm{origV}(N(v_\beta))$.

Both conditions together yield a bijection $\gamma_{v_\alpha v_\beta}$ between the neighbors of $v_\alpha$ and $v_\beta$, as origV is injective when restricted to a single skeleton (condition 3 (orig-inj)) and $\deg(v_\alpha) = \deg(v_\beta)$. Operations `SplitSeparationPair` and `JoinSeparationPair` can also be applied to an extended skeleton decomposition, yielding an extended skeleton decomposition without modifying twinV. To ensure that conditions 7 (stars) and 8 (orig-stars) remain unaffected by both operations, `SplitSeparationPair` can only be applied to non-virtual vertices.

The operations `IsolateVertex` and `Integrate` now allow us to isolate vertices out of triconnected components and integrate them back in, respectively. For `IsolateVertex`, let $v$ be a non-virtual vertex of skeleton $G_\mu$, such that $v$ has no incident occupied edges. Applying

`IsolateVertex`$(\mathcal{S}, v)$ on an extended skeleton decomposition $\mathcal{S}$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ as follows. Each neighbor $u$ of $v$ is split into two non-adjacent vertices $u_\alpha$ and $u_\beta$, where $u_\beta$ is incident to all edges connecting $u$ with $v$, while $u_\alpha$ keeps all other edges of $u$. We set $\text{origV}'(u_\alpha) = \text{origV}'(u_\beta) = \text{origV}(u)$. This creates an independent, star-shaped component with center $v$, which we move to skeleton $G_\beta$, while we rename skeleton $G_\mu$ to $G_\alpha$. We connect all $u_\alpha$ to a single new virtual vertex $v_\alpha \in V_\alpha^{\text{virt}}$ using occupied edges, and all $u_\beta$ to a single new virtual vertex $v_\beta \in V_\beta^{\text{virt}}$ using occupied edges; see Figure 3. Finally, we set $\text{twinV}'(v_\alpha) = v_\beta$, $\text{twinV}'(v_\beta) = v_\alpha$, and add $G_\beta$ to $\mathcal{G}'$. All other mappings and skeletons remain unchanged.

For the converse operation `Integrate`, consider two virtual vertices $v_\alpha, v_\beta$ with $\text{twinV}(v_\alpha) = v_\beta$ and the bijection $\gamma_{v_\alpha v_\beta}$ between the neighbors of $v_\alpha$ and $v_\beta$; see Figure 3b. An application of `Integrate`$(\mathcal{S}, (v_\alpha, v_\beta))$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ as follows. We merge both skeletons into a skeleton $G_\mu$ (also replacing both in $\mathcal{G}'$) by identifying the neighbors of $v_\alpha$ and $v_\beta$ according to $\gamma_{v_\alpha v_\beta}$. Furthermore, we remove $v_\alpha$ and $v_\beta$ together with their incident occupied edges. All other mappings and skeletons remain unchanged.

▶ **Lemma 4.** *Applying* `IsolateVertex` *or* `Integrate` *on an extended skeleton decomposition* $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE}, \text{twinV})$ *yields an extended skeleton decomposition* $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ *with* $G_{\mathcal{S}'} = G_{\mathcal{S}}$.

**Proof.** We first check that all conditions still hold in the extended skeleton decomposition $\mathcal{S}'$ returned by `IsolateVertex`. Condition 1 (bicon) remains satisfied, as the structure of $G_\alpha$ remains unchanged compared to $G_\mu$ and the skeleton $G_\beta$ is a bond. As we are again splitting a node of $T_{\mathcal{S}}$, condition 2 (tree) also remains satisfied. Due to the neighbors of $v_\beta$ and $v_\alpha$ mapping to the same vertices of $G_{\mathcal{S}'}$, conditions 3 (orig-inj), 4 (orig-real), and 5 (orig-virt) remain satisfied. Conditions 7 (stars) and 8 (orig-stars) are satisfied by construction.

Lastly, condition 6 (subgraph) could only be violated if the subgraph of $T_{\mathcal{S}'}$ formed by the allocation skeletons of some vertex $z \in G_{\mathcal{S}'}$ was no longer connected. This could only happen if only one of $G_\alpha$ and $G_\beta$ were an allocation skeleton of $z$, while the other has a further neighbor $G_\nu$ that is also an allocation skeleton of $z$. Note that in any case, $\nu$ is adjacent to $\mu$ in $T_{\mathcal{S}}$ and $\mu$ must be an allocation skeleton of $z$, thus it is $z \in \text{origV}(G_\nu) \cap \text{origV}(G_\mu)$. Depending on the adjacency of $\nu$, it is either $\text{origV}(G_\nu) \cap \text{origV}(G_\mu) = \text{origV}'(G_\nu) \cap \text{origV}(G_\alpha)$ or $\text{origV}(G_\nu) \cap \text{origV}(G_\mu) = \text{origV}'(G_\nu) \cap \text{origV}(G_\beta)$, as $\nu$ is not modified by the operation and both $\mathcal{S}$ and $\mathcal{S}'$ satisfy 5 (orig-virt) and 8 (orig-stars). This immediately contradicts the skeleton of $\{\alpha, \beta\}$, that is adjacent to $\nu$, not being an allocation skeleton of $z$.

Finally, the mapping origE remains unchanged and the only change to origV is to include some duplicated vertices mapping to already existing vertices. As condition 4 (orig-real) holds for both the input and the output instance, this cannot affect the represented graph $G_{\mathcal{S}'}$.

Now consider the extended skeleton decomposition $\mathcal{S}'$ returned by `Integrate`. The merged skeleton is biconnected, as we are effectively replacing a single vertex by a connected subgraph, satisfying 1 (bicon). The operation effectively contracts and removes an edge in $T_{\mathcal{S}}$, which does not affect $T_{\mathcal{S}'}$ being a tree, satisfying condition 2 (tree). Note that condition 2 (tree) holding for the input instance also ensures that $v_\alpha$ and $v_\beta$ belong to two distinct skeletons. As the input instance satisfies condition 5 (orig-virt), the vertices in each of the two adjacent skeletons where origV maps to the same vertex of $G_{\mathcal{S}}$ are exactly the neighbors of the matched $v_\alpha$ and $v_\beta$. Thus, $\text{origV}\,|_{V_\alpha}$ is still injective, satisfying condition 3 (orig-inj). As we modify no real or virtual edges, the mappings $\text{origV}', \text{origE}'$ and $\text{twinE}'$ obviously still satisfy conditions 4 (orig-real) and 5 (orig-virt). Finally, contracting a tree edge cannot lead to any of the subgraphs of 6 (subgraph) becoming disconnected,

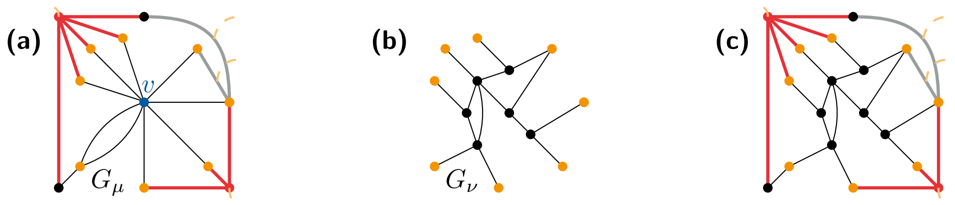

**Figure 4** Expanding a skeleton vertex $v$ into a graph $G_\nu$ in the SPQR-tree of Figure 5b. **(a)** The single allocation skeleton $G_\mu$ of $u$ with the single allocation vertex $v$ of $u$ from Figure 5b. The neighbors of $v$ are marked in orange. **(b)** The inserted graph $G_\nu$ with orange marked vertices. Note that the graph is biconnected when all marked vertices are collapsed into a single vertex. **(c)** The result of applying `InsertGraph`$(\mathcal{S}, u, G_\nu, \phi)$ followed by an application of `Integrate` on the generated virtual vertices $v$ and $v'$.

thus the condition also remains satisfied. Conditions 7 (stars) and 8 (orig-stars) also remain unaffected, as we simply remove an entry from twinV.

Again, no changes were made to origE, while condition 8 (orig-stars) makes sure that origV mapped each pair of merged vertices to the same vertex of $G_\mathcal{S}$. Thus, the represented graph $G_{\mathcal{S}'}$ remains unchanged. ◀

Furthermore, as `Integrate` is the converse of `IsolateVertex` and has no preconditions, any changes made by `IsolateVertex` can be undone at any time to obtain a (non-extended) skeleton decomposition, and thus, possibly, the SPQR-tree of the represented graph.

▶ Remark 5. Exhaustively applying `Integrate` to an extended skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE}, \text{twinV})$ yields a extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ where $\text{twinV}' = \emptyset$. Thus, $\mathcal{S}'$ is equivalent to a (non-extended) skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$.

## 5 Node expansion in extended skeleton decompositions

We now introduce the dynamic operation that changes the represented graph by expanding a single vertex $u$ into an arbitrary connected graph $G_\nu$. This is done by identifying $|N(u)|$ marked vertices in $G_\nu$ with the neighbors of $u$ via a bijection $\phi$ and then removing $u$ and its incident edges. We use the "occupied stars" from the previous section to model the identification of these vertices, allowing us to defer the actual insertion to an application of `Integrate`. We need to ensure that the inserted graph makes the same "guarantees" to the surrounding graph in terms of connectivity as the vertex it replaces, that is all neighbors of $u$ (i.e. all marked vertices in $G_\nu$) need to be pairwise connected via paths in $G_\nu$ not using any other neighbor of $u$ (i.e. any other marked vertex). Without this requirement, a single vertex could e.g. also be split into two non-adjacent halves, which could break a triconnected component apart. Thus, we require $G_\nu$ to be biconnected when all marked vertices are collapsed into a single vertex. Note that this also ensures that the old graph can be restored by contracting the vertices of the inserted graph. For the sake of simplicity, we require vertex $u$ from the represented graph to have a single allocation vertex $v \in G_\mu$ with $\text{origV}^{-1}(u) = \{v\}$ so that we only need to change a single allocation skeleton $G_\mu$ in the skeleton decomposition. As we will make clear later on, this condition can be satisfied easily.

Formally, let $u \in G_\mathcal{S}$ be a vertex that only has a single allocation vertex $v \in G_\mu$ (and thus only a single allocation skeleton $G_\mu$). Let $G_\nu$ be an arbitrary, new graph containing $|N(u)|$ marked vertices, together with a bijection $\phi$ between the marked vertices in $G_\nu$ and the neighbors of $v$ in $G_\mu$. We require $G_\nu$ to be biconnected when all marked vertices are *collapsed* into a single vertex, that is, when all marked vertices are indentified with each other. Operation `InsertGraph`$(\mathcal{S}, u, G_\nu, \phi)$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ as follows, see also Figure 4. We interpret $G_\nu$ as skeleton and add it to $\mathcal{G}'$. For each marked vertex $x$ in $G_\nu$, we set $\text{origV}'(x) = \text{origV}(\phi(x))$. For all other vertices and edges in $G_\nu$, we set $\text{origV}'$ and $\text{origE}'$ to point to new vertices and edges forming a copy of $G_\nu$ in $\mathcal{G}_{\mathcal{S}'}$. We connect every marked vertex in $G_\nu$ to a new virtual vertex $v' \in G_\nu$ using occupied edges. We also convert $v$ to a virtual vertex, converting its incident edges to occupied edges while removing parallel edges. Finally, we set $\text{twinV}'(v) = v'$ and $\text{twinV}'(v') = v$.

▶ **Lemma 6.** *Applying* `InsertGraph`$(\mathcal{S}, u, G_\nu, \phi)$ *on an extended skeleton decomposition* $\mathcal{S}$ *yields an extended skeleton decomposition* $\mathcal{S}'$ *with* $G_{\mathcal{S}'}$ *isomorphic to* $G_{\mathcal{S}}[u \rightarrow_\phi G_\nu]$.

**Proof.** Condition 1 (bicon) remains satisfied, as the structure of $G_\mu$ remains unchanged and the resulting $G_\nu$ is biconnected by precondition. Regarding $T_{\mathcal{S}}$, we are attaching a degree-1 node $\nu$ to an existing node $\mu$, thus condition 2 (tree) also remains satisfied. As all vertices of $G_\nu$ except for the vertices in $N(v')$ got their new, unique copy assigned by $\text{origV}'$ and $\text{origV}'(N(v')) = \text{origV}(N(v))$, condition 3 (orig-inj) is also satisfied for the new $G_\nu$. As we updated origE alongside origV and $G_\nu$ contains no virtual edges, conditions 4 (orig-real) and 5 (orig-virt) remain satisfied. As $\nu$ is a leaf of $T_{\mathcal{S}}$ with $\mu$ being its only neighbor, $\text{origV}'(N(v')) \subset \text{origV}(V_\mu)$, and $G_\nu$ is the only allocation skeleton for all vertices in $G_\nu \setminus N(v')$, condition 6 (subgraph) remains satisfied. Conditions 7 (stars) and 8 (orig-stars) are satisfied by construction. Finally, the mappings $\text{origE}'$ and $\text{origV}'$ are by construction updated to correctly reproduce the structure of $G_\nu$ in $G_{\mathcal{S}'}$. ◀

On its own, this operation is not of much use though, as graph vertices only rarely have a single allocation skeleton. Furthermore, our goal is to dynamically maintain SPQR-trees, while this operation on its own will in most cases not yield an SPQR-tree. To fix this, we introduce the full procedure $\texttt{InsertGraph}_{\texttt{SPQR}}(\mathcal{S}, u, G_\nu, \phi)$ that can be applied to any graph vertex $u$ and that, given an SPQR-tree $\mathcal{S}$, yields the SPQR-tree of $G_{\mathcal{S}}[u \rightarrow_\phi G_\nu]$. It consists of three preparations steps, the insertion of $G_\nu$, and two further clean-up steps:

1. We apply `SplitSeparationPair` to each polygon allocation skeleton of $u$ with more than three vertices, using the neighbors of the allocation vertex of $u$ as separation pair.
2. For each rigid allocation skeleton of $u$, we move the contained allocation vertex $v$ of $u$ to its own skeleton by applying `IsolateVertex`$(\mathcal{S}, v)$.
3. We exhaustively apply `JoinSeparationPair` to any pair of allocation skeletons of $u$ that are adjacent in $T_{\mathcal{S}}$. Due to condition 6 (subgraph), this yields a single component $G_\mu$ that is the sole allocation skeleton of $u$ with the single allocation vertex $v$ of $u$. Furthermore, the size of $G_\mu$ is linear in $\deg(u)$.
4. We apply `InsertGraph` to insert $G_\nu$ as skeleton, followed by an application of `Integrate` to the virtual vertices $\{v, v'\}$ introduced by the insertion, thus integrating $G_\nu$ into $G_\mu$.
5. We apply `SplitSeparationPair` to all separation pairs in $G_\mu$ that do not involve a virtual vertex. These pairs can be found in linear time, e.g. by temporarily duplicating all virtual vertices and their incident edges and then computing the SPQR-tree.[4]
6. Finally, we exhaustively apply `Integrate` and also apply `JoinSeparationPair` to any two adjacent polygons and to any two adjacent bonds to obtain the SPQR-tree of the updated graph.

[4] The wheels replacing virtual vertices in the proof of Theorem 10 also ensure this.

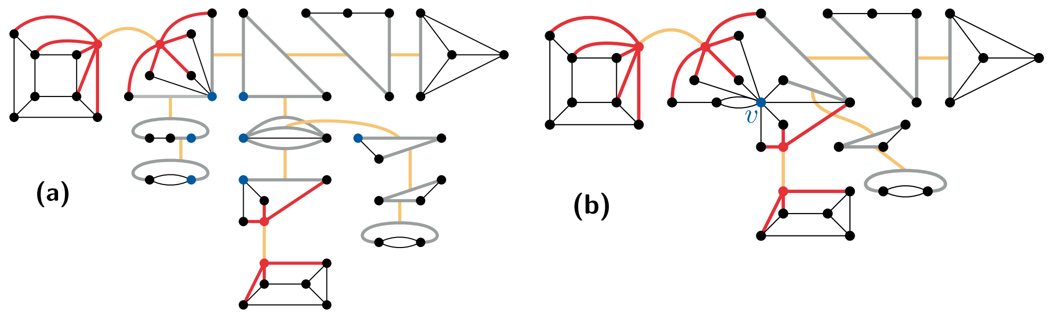

**Figure 5** The preprocessing steps of `InsertGraph`$_{\texttt{SPQR}}$ being applied to the SPQR-tree of Figure 1b. **(a)** The state after Step 2, after all allocation skeletons of $u$ have been split. **(b)** The state after Step 3, after all allocation skeletons of $u$ have been merged into a single one.

The basic idea behind the correctness of this procedure is that splitting the newly inserted component according to its SPQR-tree in Step 5 yields biconnected components that are each either a polygon, a bond, or “almost” triconnected. The latter (and only those) might still contain virtual vertices and all their remaining separation pairs, which were not split in Step 5, contain one of these virtual vertices. This, together with the fact that there still may be pairs of adjacent skeletons where both are polygons or both are bonds, prevents the instance from being an SPQR-tree. Both issues are resolved in Step 6: The adjacent skeletons are obviously fixed by the `JoinSeparationPair` applications. To show that the virtual vertices are removed by the `Integrate` applications, making the remaining components triconnected, we need the following lemma.

▶ **Lemma 7.** *Let $G_\alpha$ be a triconnected skeleton containing a virtual vertex $v_\alpha$ matched with a virtual vertex $v_\beta$ of a biconnected skeleton $G_\beta$. Furthermore, let $P \subseteq \binom{V(G_\beta)}{2}$ be the set of all separation pairs in $G_\beta$. An application of* `Integrate`$(\mathcal{S}, (v_\alpha, v_\beta))$ *yields a biconnected skeleton $G_\mu$ with separation pairs $P' = \{\{u, v\} \in P \mid v_\beta \notin \{u, v\}\}$.*

**Proof.** We partition the vertices of $G_\mu$ into sets $A, B$, and $N$ depending on whether the vertex stems from $G_\alpha$, $G_\beta$, or both, respectively. The set $N$ thus contains the neighbors of $v_\alpha$, which were identified with the neighbors of $v_\beta$. We will show by contradiction that $G_\mu$ contains no separation pairs except for those in $P'$. Thus, consider a separation pair $u, v \in G_\mu$ not in $P'$. First, consider the case where $u, v \in A \cup N$. Observe that removing $u, v$ in this case leaves $B$ connected. Thus, we can contract all vertices of $B$ into a single vertex, reobtain $G_\alpha$ and see that $u, v$ is a separation pair in $G_\alpha$. This contradicts the precondition that $G_\alpha$ is triconnected. Now consider the case where $u, v \in B \cup N$. Analogously to above, we find that $u, v$ is a separation pair in $G_\beta$ that does not contain $v_\beta$, a contradiction to $\{u, v\} \notin P'$. Finally, consider the remaining case where, without loss of generality, $u \in A, v \in B$. Since $\{u, v\}$ is a separation pair, $u$ has two neighbors $x, y$ that lie in different connected components of $G_\mu - \{u, v\}$ and therefore also in different components of $(G_\mu - \{u, v\}) - B$ which is isomorphic to $G_\alpha - \{u, v_\alpha\}$. This again contradicts $G_\alpha$ being triconnected. ◀

▶ **Theorem 8.** *Applying* `InsertGraph`$_{\texttt{SPQR}}(\mathcal{S}, u, G_\nu, \phi)$ *to an SPQR-tree $\mathcal{S}$ yields an SPQR-tree $\mathcal{S}'$ in $O(|G_\nu|)$ time with $G_{\mathcal{S}'}$ isomorphic to $G_{\mathcal{S}}[u \rightarrow_\phi G_\nu]$.*

**Proof.** As all applied operations leave the extended skeleton decomposition valid, the final extended skeleton decomposition $\mathcal{S}'$ is also valid. Observe that the purpose of the preprocessing Steps 1–3 is to ensure that the precondition of `InsertGraph` is satisfied (that there is only a single allocation vertex) and the affected component is not too large, that is, it has size linear in $\deg(u)$. This ensures that all further steps only need to consider skeletons that have, in total, a size linear in $\deg(u)$ plus the size of the inserted graph $G_\nu$. Note that all rigids split in Step 2 remain structurally unmodified in the sense that edges only changed their type, but the graph and especially its triconnectedness remains unchanged. Step 4 performs the actual insertion and yields the desired represented graph according to Lemma 6. For correctness, it thus remains to show that the clean-up Steps 5 and 6 turn the obtained extended skeleton decomposition into an SPQR-tree. Afterwards, we show that the operation can be executed in the claimed linear running time.

Applying `Integrate` exhaustively in Step 6 ensures that the extended skeleton decomposition is equivalent to a non-extended one (Remark 5). Recall that a non-extended skeleton decomposition is an SPQR-tree if all skeletons are either polygons, bonds or triconnected and two adjacent skeletons are never both polygons or both bonds (Definition 3). Step 6 ensures that the second half holds, as joining two polygons (or two bonds) with `JoinSeparationPair` yields a bigger polygon (or bond, respectively). Before Step 6, all skeletons that are not an allocation skeleton of $u$ are still unmodified and thus already have a suitable structure, i.e., they are either polygons, bonds or triconnected. Furthermore, the allocation skeletons of $u$ not containing virtual vertices also have a suitable structure, as their splits were made according to the SPQR-tree in Step 5.

It remains to show that the remaining skeletons, that is those resulting from the `Integrate` applications in Step 6, are triconnected. Note that in these skeletons, Step 5 ensures that every separation pair consists of at least one virtual vertex, as otherwise the computed SPQR-tree would have split the skeleton further. Further note that, for each of these virtual vertices, the matched partner vertex is part of a structurally unmodified triconnected skeleton that was split in Step 2. Lemma 7 shows that applying `Integrate` does not introduce new separation pairs while removing two virtual vertices if one of the two sides is triconnected. We can thus exhaustively apply `Integrate` and thereby remove all virtual vertices and thus also all separation pairs, obtaining triconnected components. This shows that the criteria for being an SPQR-tree are satisfied and, as `InsertGraph` expanded $u$ to $G_\nu$ in the represented graph, we now have the unique SPQR-tree of $G_\mathcal{S}[u \to_\phi G_\nu]$.

**Runtime.** All operations we used can be performed in time linear in the degree of the vertices they are applied on. For the bipartition of bridges input to `SplitSeparationPair`, it is sufficient to describe each bridge via its edges incident to the separation pair instead of explicitly enumerating all vertices in the bridge. Thus, the applications of `SplitSeparationPair` and `IsolateVertex` in Steps 1 and 2 touch every edge incident to $u$ at most once and thus take $O(\deg(u))$ time. Furthermore, they yield skeletons that have a size linear in the degree of their respective allocation vertex of $u$. As the subtree of $u$'s allocation skeletons has size at most $\deg(u)$, the `JoinSeparationPair` applications of Step 3 also take at most $O(\deg(u))$ time. It follows that the resulting single allocation skeleton of $u$ has size $O(\deg(u))$.

The applications of `InsertGraph` and `Integrate` in Step 4 take time linear in the number of identified neighbors, which is $O(\deg(u))$. Generating the SPQR-tree of the inserted graph in Step 5 (where all virtual vertices were replaced by wheels) can be done in time linear in the size of the inserted graph [13, 16], that is $O(|G_\nu|)$. Applying `SplitSeparationPair` according to all separation pairs identified by this SPQR-tree can also be done in $O(|G_\nu|)$ time in total.

Note that there are at most $\deg(u)$ edges between the skeletons that existed before Step 4 and those that were created or modified in Steps 4 and 5, and these are the only edges that might now connect two polygons or two bonds. Thus, the applications of `Integrate` and `JoinSeparationPair` in Step 6 run in $O(\deg(u))$ time in total. Furthermore, they remove all pairs of adjacent polygons and all pairs of adjacent bonds. This shows that all steps take $O(\deg(u))$ time, except for Step 5, which takes $O(|G_\nu|)$ time. As the inserted graph contains at least one vertex for each neighbor of $u$, the total runtime is in $O(|G_\nu|)$. ◀

▶ **Corollary 9.** *Let $\mathcal{S}_1, \mathcal{S}_2$ be two SPQR-trees together with vertices $u_1 \in G_{\mathcal{S}_1}$, $u_2 \in G_{\mathcal{S}_2}$, and let $\phi$ be a bijection between the edges incident to $u_1$ and the edges incident to $u_2$. Operation* $\texttt{Merge}_{\texttt{SPQR}}(\mathcal{S}_1, \mathcal{S}_2, u_1, u_2, \phi)$ *yields the SPQR-tree of the graph $G_{\mathcal{S}_1}[u_1 \to_\phi G_{\mathcal{S}_2} - u_2]$, i.e. the union of both graphs where the edges incident to $u_1, u_2$ were identified according to $\phi$ and $u_1, u_2$ removed, in time $O(\deg(u_1)) = O(\deg(u_2))$.*

**Proof.** Operation $\texttt{Merge}_{\texttt{SPQR}}$ works similar to the more general $\texttt{InsertGraph}_{\texttt{SPQR}}$, although the running time is better because we already know the SPQR-tree for the graph being inserted. We apply Steps 1–3 to ensure that both $u_1$ and $u_2$ have sole allocation vertices $v_1$ and $v_2$, respectively. To properly handle parallel edges, we subdivide all edges incident to $u_1, u_2$ (and thus also the corresponding real edges incident to $v_1, v_2$) and then identify the subdivision vertices of each pair of edges matched by $\phi$. By deleting vertices $v_1$ and $v_2$ and suppressing the subdivision vertices (that is, removing them and identifying each pair of incident edges) we obtain a skeleton $G_\mu$ that has size $O(\deg(u_1)) = O(\deg(u_2))$. Finally, we apply Steps 5 and 6 to $G_\mu$ to obtain the final SPQR-tree. Again, as the partner vertex of every virtual vertex in the allocation skeletons of $u$ is part of a triconnected skeleton, applying `Integrate` exhaustively in Step 6 yields triconnected skeletons. As previously discussed, the preprocessing and clean-up steps run in time linear in degree of the affected vertices, thus the overall runtime is $O(\deg(u_1)) = O(\deg(u_2))$ in this case. ◀

## 5.1 Maintaining planarity and vertex rotations

Note that expanding a vertex of a planar graph using another planar graph using $\texttt{Insert-}$ $\texttt{Graph}_{\texttt{SPQR}}$ (or merging two SPQR-trees of planar graphs using Corollary 9) might actually yield a non-planar graph. This is, e.g., because the rigids of both graphs might require incompatible orders for the neighbors of the replaced vertex. The aim of this section is to efficiently detect this case, that is a planar graph turning non-planar. To check a general graph for planarity, it suffices to check the rigids in its SPQR-tree for planarity and each rigid allows exactly two planar embeddings, where one is the reverse of the other [7]. Thus, if a graph becomes non-planar through an application of $\texttt{InsertGraph}_{\texttt{SPQR}}$, this will be noticeable from the triconnected allocation skeletons of the replaced vertex. To be able to immediately report if the instance became non-planar, we need to maintain a rotation, that is a cyclic order of all incident edges, for each vertex in any triconnected skeleton. Note that we do not track the direction of the orders, that is we only store the order up to reversal. As discussed later, the exact orders can also be maintained with a slight overhead.

▶ **Theorem 10.** *SPQR-trees support the following operations:*

- $\texttt{InsertGraph}_{\texttt{SPQR}}(\mathcal{S}, u, G_\nu, \phi)$*: expansion of a single vertex $u$ in time $O(|G_\nu|)$,*
- $\texttt{Merge}_{\texttt{SPQR}}(\mathcal{S}_1, \mathcal{S}_2, u_1, u_2, \phi)$*: merging of two SPQR-trees in time $O(\deg(u_1))$,*
- `IsPlanar`*: queries whether the represented graph is planar in time $O(1)$, and*
- `Rotation`$(u)$*: queries for one of the two possible rotations of vertices $u$ in planar triconnected skeletons in time $O(1)$.*

**Proof.** Note that the flag `IsPlanar` together with the `Rotation` information can be computed in linear time when creating a new SPQR-tree and that expanding a vertex or merging two SPQR-trees cannot turn a non-planar graph planar. We make the following changes to the operations $\texttt{InsertGraph}_{\texttt{SPQR}}$ and $\texttt{Merge}_{\texttt{SPQR}}$ to maintain the new information. After a triconnected component is split in Step 2 we now introduce further structure to ensure that the embedding is maintained on both sides. The occupied edges generated around the split-off vertex $v$ (and those around its copy $v'$) are subdivided and connected cyclically according to $\texttt{Rotation}(v)$. Instead of “stars”, we thus now generate occupied “wheels” that encode the edge ordering in the embedding of the triconnected component. When generating the SPQR-tree of the modified subgraph in Step 5, we also generate a planar embedding for all its triconnected skeletons. If no planar embedding can be found for at least one skeleton, we report that the resulting instance is non-planar by setting `IsPlanar` to false. Otherwise, after performing all splits indicated by the SPQR-tree, we assign `Rotation` by generating embeddings for all new rigids. Note that for all skeletons with virtual vertices, the generated embedding will be compatible with the one of the neighboring triconnected component, that is, the rotation of each virtual vertex will line up with that of its matched partner vertex, thanks to the inserted wheel. Finally, before applying `Integrate` in Step 6, we contract each occupied wheel into a single vertex to re-obtain occupied stars. The creation and contraction of wheels adds an overhead that is at most linear in the degree of the expanded vertex and the generation of embeddings for the rigids can be done in time linear in the size of the rigid. Thus, this does not affect the asymptotic runtime of both operations. ◀

▶ **Corollary 11.** *The data structure from Theorem 10 can be adapted to also provide the exact rotations with matching direction for every vertex in a rigid. Furthermore, it can support queries whether two vertices $v_1, v_2$ are connected by at least 3 different vertex-disjoint paths via 3Paths$(v_1, v_2)$ in $O((\deg(v_1)+\deg(v_2))\cdot\alpha(n))$ time. These adaptions change the runtime of $\texttt{InsertGraph}_{\texttt{SPQR}}$ to $O(\deg(u)\cdot\alpha(n)+|G_\nu|)$, that of $\texttt{Merge}_{\texttt{SPQR}}$ to $O(\deg(u_1)\cdot\alpha(n))$, and that of Rotation$(u)$ to $O(\alpha(n))$.*

**Proof.** The exact rotation information for `Rotation` can be maintained by using union-find to keep track of the rigid a vertex belongs to and synchronizing the reversal of all vertices within one rigid when two rigids are merged by `Integrate` as follows. We create a union-find set for every vertex in a triconnected component and apply `Union` to all vertices in the same rigid. Next to the pointer indicating the representative in the union-find structure, we store a boolean flag indicating whether the rotation information for the current vertex is reversed with regard to rotation of its direct representative. To find whether a `Rotation` needs to be flipped, we accumulate all flags along the path to the actual representative of a vertex by using an exclusive-or. As $\texttt{Rotation}(u)$ thus relies on the `Find` operation, its amortized runtime is $O(\alpha(n))$. When merging two rigids with `Integrate`, we also perform a `Union` on their respective representatives (which we need to `Find` first), making $\texttt{Integrate}(\mathcal{S}, (v_\alpha, v_\beta))$ run in $O(\deg(v_\alpha)+\alpha(n))$. We also compare the `Rotation` of the replaced vertices and flip the flag stored with the vertex that does not end up as the representative if they do not match. In total, this makes $\texttt{InsertGraph}_{\texttt{SPQR}}$ run in $O(\deg(u)\cdot\alpha(n)+|G_\nu|)$ time as there can be up to $\deg(u)$ split rigids. Furthermore, $\texttt{Merge}_{\texttt{SPQR}}$ now runs in $O(\deg(u_1)\cdot\alpha(n))$ time.

Maintaining the information in which rigid a skeleton vertex is contained in can then also be used to answer queries whether two arbitrary vertices are connected by three disjoint paths. This is exactly the case if they are part of the same rigid, appear as poles of the same bond or are connected by a virtual edge in a polygon. This can be checked by enumerating all allocation skeletons of both vertices, which can be done in time linear in their degree. As finding each of the skeletons may require a `Find` call, the total runtime for this is bounded by $O((\deg(v_1)+\deg(v_2))\cdot\alpha(n))$. ◀

## 6 Applications

In this section, we show how extended skeleton decompositions and their dynamic operation $\texttt{InsertGraph}_{\texttt{SPQR}}$ can be used to improve the runtime of the algorithm by Bläsius et al. [4] for solving Synchronized Planarity and how this transfers to other constrained planarity variants. Formally, the problem itself is defined as follows.

▶ **Problem 1.** Synchronized Planarity[5]

**Given** graph $G$ and a set $\mathcal{P}$, where each *pipe* $\rho \in \mathcal{P}$ consists of two distinct vertices $v_1, v_2 \in V(G)$ and a bijection $\varphi_\rho$ between the edges incident to $v_1$ and those incident to $v_2$, and each vertex is part of at most one pipe

**Question** Is there a drawing of $G$ where for each pipe $\rho = (v_1, v_2, \varphi_\rho)$, the cyclic order of edges incident to $v_1$ lines up with the order of edges incident to $v_2$ under the bijection $\varphi_\rho$?

The algorithm for solving Synchronized Planarity works by removing an arbitrary pipe each step, using one of three operations depending on the graphs around the matched vertices [4]; see also Figure 6. A more elaborate summary of the operations is also given in [10]. Some of these operations require so-called embedding trees, which describe all possible rotations of a single vertex in a planar graph and are used to communicate embedding restrictions between vertices with synchronized rotation. Without our optimizations, computing an embedding tree requires time linear in the size of the concerned biconnected component, that is $O(m)$. Once their embedding trees are available, each of the at most $O(m)$ executed operations runs in time linear in the degree of the pipe it is applied on, that is in $O(\Delta) \subset O(m)$ [4]. Thus, being able to generate these embedding trees without an overhead over the operation that uses them, by maintaining the SPQR-trees they can be derived from is our main contribution towards the speedup of the Synchronized Planarity algorithm.

### 6.1 Embedding trees

An *embedding tree* $\mathcal{T}_v$ for a vertex $v$ of a biconnected graph $G$ describes the possible cyclic orderings or *rotations* of the edges incident to $v$ in all planar embeddings of $G$ [5]. The leaves of $\mathcal{T}_v$ are the edges incident to $v$, while its inner nodes are partitioned into two categories: *Q-nodes* define an up-to-reversal fixed rotation of their incident tree edges, while *P-nodes* allow arbitrary rotation; see Figure 1d. To generate the embedding tree we use the observation about the relationship of SPQR-trees and embedding trees described by Bläsius and Rutter [3, Section 2.5]: there is a bijection between the P- and Q-nodes in the embedding tree of $v$ and the bond and triconnected allocation skeletons of $v$ in the SPQR-tree of $G$, respectively.

▶ **Lemma 12.** *Let $\mathcal{S}$ be an SPQR-tree with a planar represented graph $G_\mathcal{S}$. The embedding tree for a vertex $v \in G_\mathcal{S}$ can be found in time $O(\deg(v))$.*

**Proof.** We use the rotation information from Theorem 10 and furthermore maintain an (arbitrary) allocation vertex for each vertex in $G_\mathcal{S}$. To compute the embedding tree of a vertex $v$ starting at the allocation vertex $u$ of $v$, we will explore the SPQR-tree by using twinE on one of the edges incident to $u$ and then finding the next allocation vertex of $v$ as one endpoint of the obtained edge. If $u$ has degree 2, it is part of a polygon skeleton

[5] We simplify the definition and disregard the originally included Q-vertices as they can be modeled using pipes [4, Section 5].

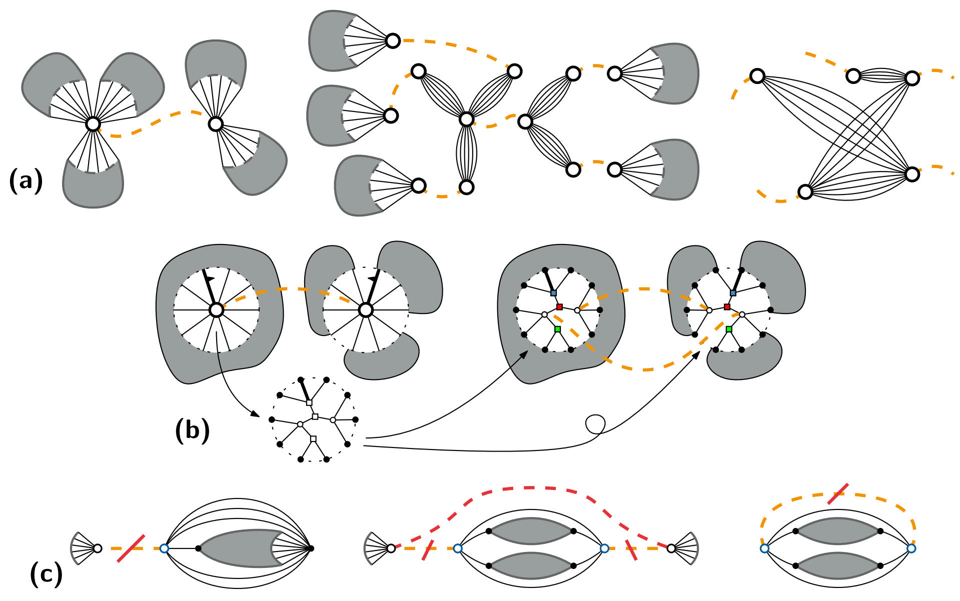

**Figure 6** Schematic representation of the three operations used by Bläsius et al. [4] for solving Synchronized Planarity. Matched vertices are shown as bigger disks, the matching (i.e., the pipes) is indicated by the orange dotted lines. **Top:** Two cutvertices matched with each other (left), the result of splitting off ("encapsulating") their incident blocks (middle) and the bipartite graph resulting from joining both cutvertices (right). **Middle:** A matched non-cutvertex with a non-trivial embedding tree (left) that is propagated to replace both the vertex and its partner (right). Constraints that only synchronize a binary decision (e.g. because they correspond to a Q-node in the embedding tree) are shown as same-colored squares. **Bottom:** Three different cases of matched vertices with trivial embedding trees (blue) and how their pipes can be removed or replaced (red).

that does not induce a node in the embedding tree. We thus move on to its neighboring allocation skeletons and will also similarly skip over any other polygon skeleton we encounter. If $u$ has degree 3 or greater, we inspect two arbitrary incident edges: if they lead to the same vertex, $u$ is the pole of a bond, and we generate a P-node. Otherwise it is part of a triconnected component, and we generate a Q-node. We now iterate over the edges incident to $u$, in the case of a triconnected component using the order given by the rotation of $u$. For each real edge, we attach a corresponding leaf to the newly generated node. The graph edge corresponding to the leaf can be obtained from origE. For each virtual edge, we recurse on the respective neighboring skeleton and attach the recursively generated node to the current node. As $u$ can only be part of $\deg(u)$ many skeletons, which form a subtree of $T_{\mathcal{S}}$, and the allocation vertices of $u$ in total only have $O(\deg(u))$ many virtual and real edges incident, this procedure yields the embedding tree of $u$ in time linear in its degree. ◀

## 6.2 Synchronized planarity

We now show how we reduce the runtime of solving Synchronized Planarity. We do so by generating an SPQR-tree upfront, maintaining it throughout all applied operations, and deriving any needed embedding tree from the SPQR-tree.

▶ **Theorem 13.** Synchronized Planarity *can be solved in time in* $O(m \cdot \Delta)$*, where* $m$ *is the number of edges and* $\Delta$ *is the maximum degree of a pipe.*

**Proof.** The algorithm works by splitting (i.e., removing and replacing by smaller ones) the pipes representing synchronization constraints until they are small enough to be trivial. It does so by exhaustively applying the three operations `EncapsulateAndJoin`, `PropagatePQ` and `SimplifyMatching` depending on the graph structure around the pairs of synchronized vertices. As mentioned by Bläsius et al., all operations run in time linear in the degree of the pipe they are applied on if the used embedding trees are known, and $O(m)$ operations are sufficient to solve a given instance [4]. Our modification is that we maintain an SPQR-tree for each biconnected component and then generate the needed embedding trees on-demand using Lemma 12.

Operation `EncapsulateAndJoin` generates a new bipartite component representing how the edges of the blocks incident to two synchronized cutvertices are matched with each other. The size of this component is linear in the degree of the synchronized vertices. Thus, we can freshly compute the SPQR-tree for the generated component in linear time, which also does not negatively impact the running time. The only other change made by this operation is that both cutvertices are split up according to their incident blocks; see Figure 6a. As this does not affect the SPQR-trees of the blocks, there are no further updates necessary.

`PropagatePQ` takes the non-trivial embedding tree of one synchronized vertex $v$ and inserts copies of the tree in place of $v$ and its partner, respectively. Synchronization constraints on the inner vertices of the inserted trees are used to ensure that the trees are embedded in the same way; see Figure 6b. We use $\texttt{InsertGraph}_{\texttt{SPQR}}$ to also insert the embedding tree into the respective SPQR trees, representing Q-nodes using wheels. When propagating into a cutvertex we also need to check whether two or more incident blocks merge. We form equivalence classes on the incident blocks, where two blocks are in the same class if 1) the two subtrees induced by their respective edges share at least two nodes 2) both induced subtrees share a Q-node that has degree at least 2 in both subtrees. Blocks in the same equivalence class will end up in the same biconnected component as follows: We construct the subtree induced by all edges in the equivalence class and add a single further node for each block in the class, connecting all leaves to the node of the block the edges they represent lead to. We calculate the SPQR-tree for this biconnected graph and then merge the SPQR-trees of the individual blocks into it by applying Corollary 9. As $\texttt{InsertGraph}_{\texttt{SPQR}}$ (and similarly all $\texttt{Merge}_{\texttt{SPQR}}$ applications) runs in time linear in the size of the inserted embedding tree, which is limited by the degree of the vertex it represents, this does not negatively impact the running time of the operation.

Operation `SimplifyMatching` can be applied if the graph around a synchronized vertex $v$ allows arbitrary rotations of $v$, that is the embedding tree of $v$ is trivial. In this case, the pipe can be removed without modifying the graph structure; see Figure 6c. As this operation makes no changes to the graph, no updates to the SPQR-trees are necessary.

Furthermore, as we now no longer need to iterate over whole biconnected components to generate the embedding trees, we are also no longer required to ensure those components do not grow too large. We can thus also directly contract pipes between two distinct biconnected components using Corollary 9 instead of having to insert embedding trees using `PropagatePQ`. This may improve the practical runtime, as `PropagatePQ` might require further operations to clean up the generated pipes, while the direct contraction entirely removes a pipe without generating new ones. ◀

## 6.3 Other constrained planarity variants

The speed-ups for Connected SEFE, Partially PQ-constrained Planarity, Row-Column Independent NodeTrix Planarity, and Strip Planarity in Table 1 follow directly by combining their linear reduction to Synchronized Planarity by Bläsius et al. [4] with the improved runtime for Synchronized Planarity from Theorem 13. For Clustered Planarity, we provide a more detailed analysis. Formally, this problem is defined as follows.

▶ **Problem 2.** Clustered Planarity
**Given** graph $G$, rooted tree $T$, cluster assignment function $\gamma : V(G) \to V(T)$
**Question** Is there a planar drawing where, for each cluster $c \in V(T)$, we can add a simple closed region that

1. encloses exactly the vertices mapped to $c$ or one of its descendants in $T$, and
2. has a border that crosses each edge that connects a vertex within its interior to a vertex on its outside exactly once, but no other edge or cluster region border?

▶ **Corollary 14.** Clustered Planarity *can be solved in time in* $O(n + d \cdot \Delta)$*, where* $d$ *is the total number of crossings between cluster borders and edges and* $\Delta$ *is the maximum number of edge crossings on a single cluster border.*

**Proof.** Note that for a simple graph to be planar, the number of edges has to be linear in the number of vertices. We apply the reduction from Clustered Planarity to Synchronized Planarity as described by Bläsius et al. [4]. We then generate an SPQR-tree for every component of the obtained instance with size in $O(n+d)$ in linear time. The instance contains one pipe for every cluster boundary, where the degree of a pipe corresponds to the number of edges crossing the respective cluster boundary. Thus, the potential described by Bläsius et al. [4], which sums up the degrees of all pipes with a constant factor depending on the endpoints of each pipe, is in $O(d)$. Each operation applied when solving the Synchronized Planarity instance runs in time $O(\Delta)$ (the maximum degree of a pipe) and reduces the potential by at least 1. Thus, a reduced instance without pipes, which can be solved in linear time, can be reached in $O(d \cdot \Delta)$ time. ◀

## References

1 P. Angelini, T. Bläsius, and I. Rutter. Testing mutual duality of planar graphs. *International Journal of Computational Geometry & Applications*, 24(4):325–346, 2014. `arXiv:1303.1640`, `doi:10.1142/S0218195914600103`.

2 P. Angelini, G. D. Lozzo, G. Di Battista, and F. Frati. Strip planarity testing for embedded planar graphs. *Algorithmica*, 77(4):1022–1059, 2016. `doi:10.1007/s00453-016-0128-9`.

3 T. Bläsius and I. Rutter. Simultaneous PQ-ordering with applications to constrained embedding problems. *ACM Transactions on Algorithms*, 12(2):16:1–16:46, 2016. `doi:10.1145/2738054`.

4 T. Bläsius, S. D. Fink, and I. Rutter. Synchronized planarity with applications to constrained planarity problems. *ACM Transactions on Algorithms*, 19(4):1–23, Sept. 2023. `arXiv:2007.15362`, `doi:10.1145/3607474`.

5 K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976. `doi:10.1016/s0022-0000(76)80045-1`.

6 G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996. `doi:10.1007/bf01961541`.

7 G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996. `doi:10.1137/s0097539794280736`.

8 D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification. *Journal of Computer and System Sciences*, 52(1):3–27, 1996. `doi:10.1006/jcss.1996.0002`.

9 S. D. Fink and I. Rutter. Maintaining triconnected components under node expansion. In M. Mavronicolas, editor, *Proceedings of the 13th International Conference on Algorithms and Complexity (CIAC'23)*, volume 13898 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2023. doi:10.1007/978-3-031-30448-4_15.

10 S. D. Fink and I. Rutter. *Constrained Planarity in Practice: Engineering the Synchronized Planarity Algorithm*, pages 1–14. Society for Industrial and Applied Mathematics, 2024. doi:10.1137/1.9781611977929.1.

11 R. Fulek and C. D. Tóth. Atomic embeddability, clustered planarity, and thickenability. *Journal of the ACM*, 69(2):13:1–13:34, 2022. arXiv:1907.13086v1, doi:10.1145/3502264.

12 C. Gutwenger. *Application of SPQR-trees in the planarization approach for drawing graphs.* PhD thesis, 2010. URL: https://eldorado.tu-dortmund.de/bitstream/2003/27430/1/diss_gutwenger.pdf.

13 C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'02)*, pages 77–90. Springer, 2001. doi:10.1007/3-540-44541-2_8.

14 J. Holm and E. Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC'20)*, pages 167–180. ACM, 2020. arXiv:1911.03449, doi:10.1145/3357713.3384249.

15 J. Holm and E. Rotenberg. Worst-case polylog incremental SPQR-trees: Embeddings, planarity, and triconnectivity. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'20)*, pages 2378–2397. SIAM, 2020. doi:10.1137/1.9781611975994.146.

16 J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973. doi:10.1137/0202012.

17 G. Liotta, I. Rutter, and A. Tappini. Simultaneous FPQ-ordering and hybrid planarity testing. *Theoretical Computer Science*, 874:59–79, June 2021. doi:10.1016/j.tcs.2021.05.012.

18 S. Mac Lane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460–472, 1937. doi:10.1215/S0012-7094-37-00336-3.

19 R. Weiskircher. *New applications of SPQR-trees in graph drawing.* PhD thesis, Universität des Saarlandes, 2002. doi:10.22028/D291-25752.