SPQR-tree Construction via Ear Decompositions Theory and Experiments

Master Thesis of

Marcel Posch

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

Reviewers:	Prof. Dr. Ignaz Rutter
	Prof. Dr. Christian Bachmaier
Advisors:	Prof. Dr. Ignaz Rutter
	Simon Dominik Fink, M. Sc.

Time Period: 06.05.2022 – 14.11.2022

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Passau, November 11, 2022

Abstract

We consider the problem of finding and maintaining the triconnected components of a graph. The problem of decomposing a graph into triconnected components was first described by Mac Lane in 1937 [Lan37] and later refined by Tutte in 1966 [Tut66]. Tutte introduced the concept of *virtual edges* and devised a decomposition of a biconnected graph into three fundamental types of components polygons, bonds and simple triconnected components. The polygons describe simple circles, whereas bonds describe pairs of vertices, that are connected by multiple parallel edges. The virtual edges represent a contraction of a path into a single edge. This allows the splitting of a biconnected graph into multiple biconnected split graphs. For the problem of finding and maintaining the triconnected components, we consider the SPQR-tree, a dynamic data structure used for efficiently maintaining the triconnected components of a graph, based on the Tutte [Tut66] components. The SPQR-tree mainly finds applications in planarity testing [BT89, DBT96] and graph drawing [BHR19, Gut10, DL98], but also finds applications in other areas such as lithography [LH10]. We examine the constructing of SPQR-trees using ear-decompositions. An ear-decomposition is another graph decomposition, which decomposes the graph into a series of edge-disjoint paths such that each path is only connected to a former path through its endpoints. Ear-decomposition may also be used to identify the triconnected components of a graph and test for planarity. However, unlike current the SPQR-tree, ear-decomposition can be easily computed using parallelism.

Deutsche Zusammenfassung

Wir betrachten das Problem der Verwaltung und Erkennung von 3-Zusammenhangskomponenten. Die Zerlegung eines Graphen in 3-Zusammenhangskomponenten wurde erstmals von Mac Lane [Lan37] in 1937 beschrieben und später durch Tutte 1966 [Tut66] weiter entwickelt. Tutte führte das Prinzip der *virtuellen Kanten* ein und entwickelte somit die Zerlegung in drei fundamentale Typen der Komponenten. Tutte [Tut66] nannte diese Komponenten *polygons* (Polygone), *bonds* (Bündel) und *simple triconnected components* (einfache 3-Zusammenhangskomponenten). Die Polygone bezeichen einfache Kreisgraphen, wohingegen die Bündel aus zwei Knoten mit mehreren parallelen Kanten bestehen. Die virtuellen Kanten repräsentieren die Kontraktion eines Pfades zu einer einzelnen Kante. Dies erlaubt es einen zweifach verbundenen Graphen in mehrere ebenfalls zweifach verbundene Graphen aufzuspalten. Für die Verwaltung der 3-Zusammenhangskomponenten verwenden wir den SQPR-Baum, eine dynamische Datenstruktur, zur Verwaltung der 3-Zusammenhangskomponenten. Der SPQR-Baum basiert auf den Tutte-Komponenten [Tut66]. Anwendungen für SPQR-Bäume finden sich hauptsächlich in Planaritätstests [BT89, DBT96] und im Graphenzeichnen [BHR19, Gut10, DL98], jedoch auch in anderen Bereichen wie z.B. in der Lithographie [LH10]. Wir betrachen die Konstruktion von SPQR-Bäumen mittels Ohrenzerlegungen. Die Ohrenzerlegung ist eine weitere Graphenzerlegung in eine Folge von kantendisjunkten Pfaden, die nur über ihre Endpunkte mit vorherigen Pfaden verbunden sind. Ähnlich wie der SPQR-Baum kann eine Ohrenzerlegung zum Finden von 3-Zusammenhangskomponenten und für Planaritätstests verwendet werden. Anders als der SPQR-Baum lässt sich die Ohrenzerlegung jedoch leicht parallelisieren.

Contents

1. Introduction

Connectivity represents one of the fundamental properties of a graph. The degree of connectivity describes how many nodes or edges have to be removed to disconnect any two nodes of the graph. The lowest degree of connectivity is a disjoint graph, the simplest of which is an independent set. The maximum degree of connectivity in a graph is bounded by the number of nodes, a simple example of a highly connected graph is a clique. The degree of connectivity is particularly important for network reliability, but is also relevant for planarity testing and graph drawing. One important problem is finding and maintaining components of higher connectivity in a graph of lower connectivity, such as connected components in a disjoint graph, biconnected components in a connected graph or triconnected components in a biconected graph.

We consider the problem of finding and maintaining the triconnected components of a graph. The problem of decomposing a graph into triconnected components was first described by Mac Lane in 1937 [Lan37] and later refined by Tutte in 1966 [Tut66] (c.f. [Gut10]). Tutte introduced the concept of *virtual edges* and devised a decomposition of a biconnected graph into three fundamental types of components, which he referred to as polygons, bonds and simple triconnected components. The polygons describe simple circles, whereas bonds describe pairs of vertices that are connected by multiple parallel paths. The two aforementioned types are similar to the serial and parallel components in series-parallel graphs. The virtual edges represent a contraction of a path into a single edge. This allows the splitting of a graph into multiple biconnected split graphs, which Tutte [Tut66] referred to as *cleavage graphs*.

For the problem of finding and maintaining the triconnected components, we consider the SPQR-tree, a dynamic data structure used for efficiently maintaining the triconnected components of a graph, based on the Tutte [Tut66] components (c.f. [BT96]). The SPQR-tree mainly finds applications in planarity testing [BT89, DBT96] and graph drawing [BHR19, Gut10, DL98], but also finds applications in other areas such as lithography [LH10]. We examine the construction of SPQR-trees using ear-decompositions and compare it to the linear time construction of Gutwenger [Gut10]. An ear-decomposition is another graph decomposition, which decomposes the graph into a series of edge-disjoint paths such that each path is only connected to a former path through its endpoints. Ear-decomposition may also be used to identify the triconnected components of a graph and test for planarity. However, unlike the SPQR-tree proposed by Di Battista and Tamassia [BT96], ear-decomposition can be easily computed using parallelism (see e.g.[Ram92]).

1.1 SPQR-tree Background

The concept of the graph decomposition into its triconnected components originates from the components three types of triconnected components described by Tutte in 1966 [Tut66]. As mentioned before, the three types of components in a biconnected graph are: *polygons*, *bonds* and *simple triconnected components*. The components have direct counter parts in the SPQR-tree. The polygons correspond to the S-nodes, bonds correspond to P-nodes and simple triconnected components correspond to R-nodes (c.f. [BT96, Gut10]). Connecting the polygons, bonds and simple triconnected components with their virtual edges, such that two components are adjacent if they share a virtual edge, induces an an acyclic connected graph, i.e., a tree. Adding a Q-node for each edge and rooting the tree at one of the Q-nodes yields an SPQR-tree.

In 1973 Hopcroft and Tarjan [HT73] published a linear-time algorithm for finding the triconnected components of a graph. The algorithm did contain some mistakes, which were only later corrected by Gutwenger and Mutzel [GM01] in 2001. The algorithm was used to create a linear-time algorithm for the construction of SPQR-trees.

The SPQR-tree was first proposed by Di Battista and Tamassia in 1989 [BT89] for the purpose of on-line planarity testing. As a result, the original definition was based on planar biconnected graphs, but the definition was soon after extended to general biconnected graphs and also disjoint graphs using BC-trees. The definition of the SPQR-tree is simpler than the implicit structure given by the Tutte components and allows for vertex and edge insertions. Di Battista and Tammasia suggested that the Hopcroft and Tarjan algorithm [HT73] could be used to construct the SPQR-tree in linear time (see [BT90, BT96]).

Even before Di Battista and Tamassia published their journal version of the SPQR-tree in 1996 [BT96], La Poutré [Pou92] published an optimized structure for maintaining the triconnected components of a graph in 1992 based on the SPQR-tree. The structure he proposed achieves a running time in $\mathcal{O}(k \cdot \alpha(n, k))$, for k operations on a graph with n nodes. For comparison, the SPQR-tree proposed by Di Battista and Tamassia requires $\mathcal{O}(k \cdot \log k)$ time for k operations (c.f. [BT96]).

The first implementation of SPQR-trees and BC-trees was published in 1997 [gdt] as part of the GDToolkit. Di Battista and Tamassia suggested that the SPQR-tree could be constructed in linear time by adapting the Hopcroft and Tarjan [HT73] algorithm (see [BT96]). However, the implementation of SPQR-trees in the GDToolkit did not run in linear time. This is likely due to the fact that no linear time implementation of the Hopcroft and Tarjan algorithm existed at the time and the fact that the algorithm contained several mistakes (c.f. [GM01, Gut10]). An actual linear time algorithm for the construction of the SPQR-tree was published by Gutwenger and Mutzel in 2001 [GM01], who discovered and corrected major mistakes in the Hopcroft and Tarjan algorithm [HT73]. The linear time implementation was made available through the graph drawing library AGD [MGB+98], which later became the graph drawing framework OGDF [CGJ+14].

In 2018 Holm et. al. [HIK+18] proposed a decremental SPQR-tree which supports both edge deletion and edge contraction in $\mathcal{O}(\log^2(n))$ amortized time. Holm and Rotenberg [HR20] further improved the decremental SPQR-tree in 2020, turning it into a fully-dynamic SPQR-tree with an amortized running time $\mathcal{O}(\log^3(n))$ per edge deletion or insertion.

1.2 Ear-Decomposition Background

The ear-decomposition was originally proposed by Hetyei in 1964 [Het64] and later refined by Lovász and Plummer in 1975 [LP75] (c.f. [Lov83]). The definition of the ear-decomposition has changed significantly throughout the years. In its original form, the ear-decomposition

did not require the initial ear to be a path, but rather the ear-decomposition allowed an arbitrary subgraph of the input graph to be the starting point of the decomposition, which was not considered to be an ear itself.

The term *strong ear-decomposition* was used to describe an ear-decomposition starting with a single edge, which is more closely related to the later ear-decompositions. The single edge starting point was eventually combined with the following ear to form a cycle, which is the most commonly used definition ear decomposition as of today.

Ear-decompositions are used for finding maximal matching covers [Lov83], finding trin-connected components [MR87, Ram92, FRT93], st-numbering [MSV86, Bra02, SS19] and recognition of series-parallel graphs [Epp92].

2. Preliminaries

We call a graph *simple*, if it has no parallel edges. Conversely, a graph that may contain parallel edges is called a *multigraph*. In the following we simply use *graph* to refer to a multigraph.

Let $G = (V, E)$ be a graph. A *path* of length $n \in \mathbb{N}$ is a sequence of edges $e_1, e_2, ..., e_{n-1}, e_n$, such that $e_i = (v_i, v_{i+1})$ holds for $1 \leq i < n$, where $v_i \in V$. We refer to the first and last vertices, that is v_1 and v_n, as the *endpoints* of the path.

If the graph is simple we use the sequence of vertices to refer to the unique path induced by the sequence.

Let $G = (V, E)$ be a graph. We say G is *connected* if for any two vertices $v_1, v_2 \in V$, there exists a path with v_1 and v_2 as its endpoints. We call a graph that is not connected is a *disjoint* graph. A *cutvertex* $v \in V$ is a vertex which upon deletion separates the graph into multiple non connected components, that is the graph induced by $V \setminus \{v\}$ is not connected.

Let $G = (V, E)$ be a graph. We call two paths $e_1, ..., e_n \in V$ and $e'_1, ..., e'_m \in V$ *edge-disjoint*, if $e_i \neq e'_j$ for all $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$ (see Figure 2.1).

Let $G = (V, E)$ be a graph. We call two paths $v_1, ..., v_n \in V$ and $v'_1, ..., v'_m \in V$ *vertex-disjoint*, if $v_i \neq v'_j$ holds for all $2 \leq i \leq n - 1$ and $2 \leq j \leq m - 1$ (see Figure 2.1).

Two non-edge-disjoint paths with at least two edges are also not vertex disjoint, as the sharing of an edge means the endpoints of the edge are also shared, that is there is a shared vertex that is not an endpoint of the path.

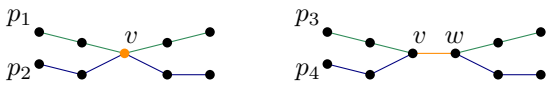


Figure 2.1: Two edge-disjoint paths p_1, p_2 to the left and two non edge-disjoint paths p_3, p_4 to the right. The paths p_1, p_2 are not vertex-disjoint as both contain v as an inner vertex. The paths p_3, p_4 are not edge-disjoint due to the shared edge $\{v, w\}$. The two paths are also not vertex-disjoint due to the endpoints of the shared edge being inner vertices.

Let $G = (V, E)$ be a graph and $k \in \mathbb{N}_1$. We consider G to be *k-vertex-connected* if for all subsets $C = \{v_1, ..., v_{k-1}\} \subseteq V$, the subgraph induced by $V \setminus C$ is connected.

Let $G = (V, E)$ be a graph and $k \in \mathbb{N}_1$. We consider G to be *k-edge-connected* if for all subsets $C = \{e_1, ..., e_{k-1}\} \subseteq E\}$, the Graph $G' = (V, E \setminus C)$ is connected.

We can also describe k-connectivity in terms of vertex disjoint paths (or edge disjoint paths for k-edge-connectivity), that is two vertices are k-vertex-connected, if there are k vertex disjoint paths connecting the two vertices. The two definitions are in fact equivalent as per Menger's theorem [Men27].

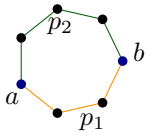


Figure 2.2: The two paths between the biconnected vertices a and b.

We call a graph *biconnected* if it is 2-vertex-connected (see Figure 2.2). Likewise, a 3-vertex-connected graph is called *triconnected*.

Let $G = (V, E)$ be a graph, we call a k-vertex connected maximal subgraph G' a *k-vertex connected component* of G.

That is G' is a k-vertex connected component if no set of vertices and edges of G may be added to G' without breaking the k-connectivity. As with the connectivity of the graph itself we call a component which is 1-vertex connected, simply a *connected component*. Similarly, we call the 2-vertex and 3-vertex connected components *biconnected components* and *triconnected components* respectively (see Figure 2.2).

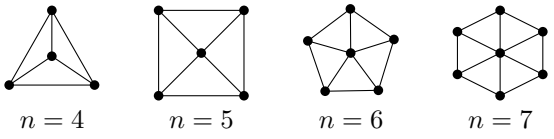


Figure 2.3: Different sizes of wheel graphs.

A simple category of triconnected components are the *wheel graphs* identified by Tutte [Tut66] in 1966 (2.3). A wheel graph of size $n \geq 4$ consists of a cycle graph with $n - 1$ vertices and a single vertex, which is connected to all other vertices, similar to a star graph. The wheel graph of size n is the smallest triconnected graph with n nodes. Since each wheel graph has $n - 1$ edges for the polygon and $n - 1$ edges for the star graph each triconnected graph with n nodes, has at least $2n - 2$ edges.

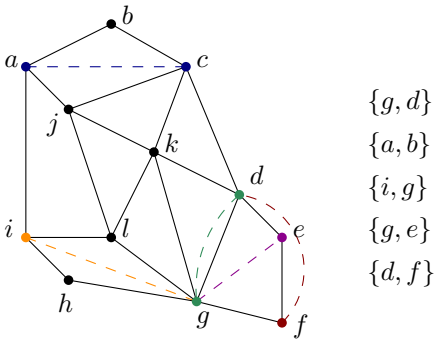


Figure 2.4: A biconnected graph with its separation pair vertices in matching colors. Note that a vertex may belong to multiple separation pairs. Each vertex that is not part of a separation pair is only adjacent to triconnected vertices.

Let G be a biconnected graph. A *separation-pair* $\{s, t\}$ of G is pair of vertices which when removed separate the graph into multiple *split components* $C_1, ..., C_n$, where $n \in \mathbb{N}_1$ (see Figure 2.4).

For $k \in \mathbb{N}_0$ and $j \in \mathbb{N}_1$, the Ackermann function is defined as follows:

$$\mathrm{Ack}_k(j) = \begin{cases} j+1, & \text{if } k = 0 \\ \mathrm{Ack}_{k-1}^{(j+1)}(j), & \text{if } k \geq 1 \end{cases}$$

The Ackermann function grows extremely quickly in k, going from 2047 at $\mathrm{Ack}_3(1)$, to vastly exceeding 10^{80} at $\mathrm{Ack}_4(1)$.

The inverse α of the Ackermann function is given by $\alpha(m, n) = \min\{k \in \mathbb{N}_0 \mid \mathrm{Ack}(m, k) \geq n\}$, where $m, n \in \mathbb{N}_0$.

Since the Ackermann function grows extremely quickly, its inverse function α grows extremely slowly, to the point, that it is often considered a constant in practical applications.

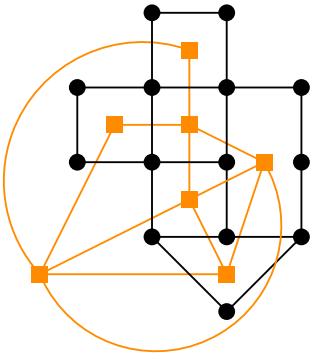


Figure 2.5: A graph G (drawn in black) with its corresponding vertex face graph $G^\diamond$ (drawn in orange).

Let G be a plane embedded graph. We call the regions enclosed by the edges of G, the *faces* of G. The outer plane, which is not enclosed by any edge is called the *outer face* of G.

Let G be a plane embedded graph. The *dual* graph $G^\diamond$ of G is the graph obtained by placing a vertex for each face of G and connecting the vertices of adjacent faces (see Figure 2.5). The dual graph is also called the *vertex-face* graph.

The vertex face graph of a canonically embedded wheel graph (see Figure 2.3) is itself a wheel graph, as the inner faces form a cycle and each inner face is connected to the outer face.

3. Data Structures and Algorithms

In the following we consider useful data structures and algorithms for the SPQR-tree, such as Union-Find, Split-Find, ear-decompositions and the BC-tree. Union-Find and Split-Find present useful data structures for the maintenance of the SPQR-tree, as different nodes types in the SPQR-tree require the different operations. The R-nodes only require unions and S-nodes only require splits, but none of the different node types requires both operations. This allows the application of both structures to different parts of the SPQR-tree without having to use a combined, less efficient structure for the entire SPQR-tree.

3.1 Union-Find

The Union-Find structure is a data structure for maintaining disjoint sets, under union operations. The Union-Find structure supports three operations

- `makeSet`(v), which creates a new set containing only v,
- `find`(v), which returns the representative of v,
- `union`(v, w), which combines the sets of v and w into a single set, with the representative $v' \in \{\texttt{find}(v), \texttt{find}(w)\}$.

The structure is generally implemented as a tree, where each element except the representative has another element of the same set as its parent. To find the representative of a set, we simply traverse the path to the root. To combine two sets we attach the root of the second set to the root of the first set (see Figure 3.1).

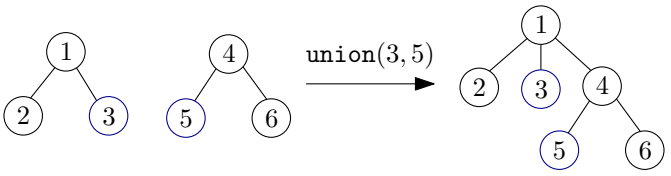


Figure 3.1: A union operation in a simple Union-Find structure.

Naturally, the tree may overtime grow in depth which negatively affects the performance of the find operation, due to the longer paths to the root. To remedy this problem, we simply *compress* the path traversed during a find operation, that is we remove the path and attach all nodes on the path to the root instead. This technique is called *path compression*.

Similarly, we may also improve the union operation by attaching the tree with lower height to the tree of greater height, this technique is referred to as *weighted union*.

Using these techniques the running time for m operations on the Union-Find structure is $\mathcal{O}(m \cdot \alpha(m, n))$ time, where n is the number of `makeSet` operations (see also [LP90]). The structure matches well with the R-nodes of the SPQR-tree as R-nodes are merged through edge insertions.

3.2 Split-Find

Th Split-Find structure is a similar structure to the Union-Find. Though, its main operation `split` is the counterpart to the `union` operation. Split-Find maintains a disjoint collection of integer intervals, with integers $1, ..., n$, under set splitting operations. Unlike Union-Find, Split-Find requires the elements to be ordered in order for splits to be well defined and efficient. Split-Find offers the following operations.

- `find`(i), which returns the representative of i,
- `split`(i), which splits the interval $S = \{a, ..., i, ..., b\}$ into two disjoint intervals $S_1 = \{a, ..., i\}$ and $S_2 = \{i + 1, ..., b\}$, where a and b are the lowest and highest integers in the interval S respectively.

Similar to Union-Find, the Split-Find structure supports the operation `find`(i) but uses `split` instead of `union`. Unlike Union-Find, the Split-Find structure generally does not support an incremental operation like `makeSet`. The structure is instead instantiated with a single set containing all elements, i.e., the interval $\{1, ..., n\}$.

Like Union-Find, Split-Find can be implemented as an ordered tree or rather a forest structure, with each set as its own tree. Though unlike Union-Find, all elements are present as leaves and the root stores the representative of the set. The nodes in the tree are connected via bidirectional pointers, that is, each node knows its children as well as its parent. A simple initialization for the tree structure consists of a single tree with a single inner node connected to the n elements. More sophisticated implementations use multiple inner nodes to pre-split the set into multiple subsets which get bigger towards the root and smaller towards the leaves, e.g., the subset of a parent is the union of the sets of its children (see figure for an example 3.3). To find the representative of an element i we simply traverse the path from the leaf i to the root of its set, the root stores the identity of the representative. The representative in Split-Find is generally chosen to be the highest numbered element of the set, though it is not strictly required. Using the highest element as the representative of the set is useful in structures which use pre-split sets, as it makes it easy to detect if a set needs to be split or is already correctly (pre-)split. This differs from Union-Find where the number of the number of the representative does not matter, as long as the representative is part of the same set. The reason for this divergence lies in the splitting, which usually requires knowledge of the highest numbered element in the set and pre-split subsets The time required to obtain the representative depends on the length of the path.

To split the set at an element i, we traverse the path from the leaf i to the root v_r. Each node v on the path $v_i, ..., v_r$ is replaced by two nodes v_1, v_2 one for lower half of the children and one for the upper half. The last child of v_1 is the ancestor of i, which was a child of v. Note, that the upper half may be empty, e.g., when we call `split` on the last element of a set or when v is a leaf. In this case no splitting is necessary, as v already contains the correct children. Splitting a vertex v requires us to update its children to point to their new parents, to avoid having to move all children, we only update the smaller portion of the split, i.e., keeping the original vertex v as the parent of the larger portion (see Figure

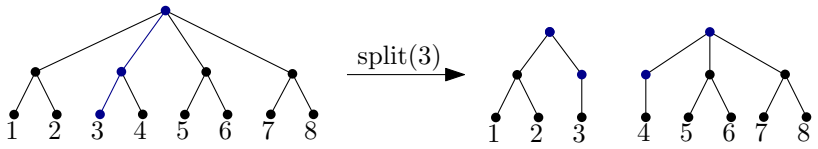


Figure 3.2: Splitting of the set $S = \{1, ..., 8\}$ into sets $S_1 = \{1, 2, 3\}$ and $S_2 = \{4, ..., 8\}$. The path to the root with the split nodes is marked in blue.

3.2). We can figure out which is the smaller portion in time proportional to the smaller portion, by simultaneously counting the two split parts.

The following recursive algorithm describes the splitting of a node v at an element i, starting from the root node.

1. First, we obtain the unique child c of v which is an ancestor of the leaf containing i.
2. If i is not the highest numbered element of c, we recursively split c into c_1 and c_2, where the highest numbered element of c_1 is i.
3. Then, we split v into two new nodes v_1 and v_2 with v_1 containing the children of v up to and including c_1 and v_2 contains c_2 and the following children, if c was not split we simply use c for c_1 and the next child for c_2.

To easily check if i is the highest numbered of a node v, we store highest numbered vertex, i.e., the representative of each subset at the corresponding vertex. The time required for each split of a set depends on the number of elements that have to be moved.

Efficient Split-Find Implementations

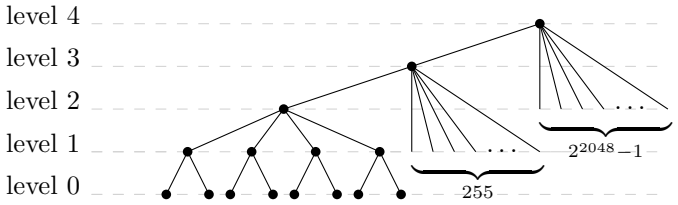


Figure 3.3: The tree structure proposed by Hopcroft and Ullman [HU73]. Each inner node has at most $2^{A(i-1)}$ children, where i is the level of the node.

Hopcroft and Ullman [HU73] have proposed a Split-Find tree structure with subset sizes based on the values Ackermann function, which yields an upper bound of $\mathcal{O}(n \cdot \alpha(n))$ on the cost of moving elements.

We call *level* the longest path from a vertex v to a leaf, i.e., the leaves are at level 0, the parents of the leaves are at level 1 and so forth, with the root at the highest level. A node n at level i is complete if one of the following conditions is satisfied.

- Node n is a leaf, which means its level is zero.
- Node n has $2^{\text{Ack}(i-1)}$ children and its children are complete, where i is the level of n.

This ensures that processing each additional level requires less than a fraction of the time of its children. The structure was originally designed for the Union-Find problem and adjusted for Split-Find.

The structure requires $\mathcal{O}(n \cdot \alpha(n))$ amortized time for all split operations according to Hopcroft and Ullman [HU73]. Later publications from other authors such as Gabow and Tarjan [GT85], La Poutré [LP90] and Imai and Asano [IA84], however, state that the

running time of the structure is $\mathcal{O}(n + m \cdot \log^* n)$, where $\log^* n$ is number of repeated log applications required to get below 1, i.e.

$$\log^*(n) = 0 \qquad \text{if } n < 1,$$
$$\log^*(n) = 1 + \log^*(\log n) \qquad \text{if } n \geq 1.$$

Though, to the extent of our knowledge none of the other authors provides an analysis of the Hopcroft and Ullman [HU73] structure.

The time bound of $\mathcal{O}(n \cdot \alpha(n))$ is optimal for pointer machines, since La Poutré [Pou94] has shown that $\Omega(n + m \cdot \alpha(m, n))$ is a lower bound for the split-find problem on pointer machines. Random access machines however, do admit faster solutions.

Improvements for Split-Find

Gabow and Tarjan [GT85] have proposed a similar structure based on the layering of Hoprcroft and Ullman [HU73]. The structure uses only three layers called *macrosets*, *mezzosets* and *microsets*. Each layer has a logarithmic number of elements of its parent set, that is each mezzoset has $\lfloor \log n \rfloor$ elements and each microset has $\lfloor \log \log n \rfloor$, elements, with the macrosets representing the actual sets which are maintained, i.e., the initial macroset consists of n elements. Furthermore, Gabow and Tarjan [GT85] store the name of the containing set with each integer, similar to the technique we mentioned earlier, where we stored the highest numbered element at the inner nodes of the tree. Like the Split-Find structure of Hopcroft and Ullman [HU73] it was originally designed for Union-Find and modified for Split-Find.

With this structure Gabow and Tarjan have achieved an overall running time of $\mathcal{O}(m + n)$ on a random acess machine, for m operations and hence amortized constant time for a single operation, which is an improvement over the structure of Hopcroft and Ullman [HU73]. However, this is only possible through the use of random access machines as shown by La Poutré [Pou94].

Element Insertion

Both the structure of Gabow and Tarjan [GT85] and the structure of Hopcroft and Ullman [HU73] require the first set to be initialized with all elements. Imai and Asano [IA84] have proposed an incremental extensions for both data structures which allow elements to be added. Extending the operations by $\texttt{add}(v, v^*)$, which adds the element v^* to the set $S(v)$, where $v^* > u \in S(v)$. The elements are effectively appended to the end of the corresponding set and given an id that is higher than all other elements even those that are in a different set.

This incremental principle can be applied to both the Hopcroft and Ullman [HU73] structure as well, as the structure of Gabow and Tarjan [GT85]. The running times of the extensions are equivalent to the non incremental running times. The extension of the Hopcroft and Ullman [HU73] structure runs in $\mathcal{O}((k+m) \log^* k)$, starting from the empty set, where k is the number of add operations and m is the number of split and find operations. Note, that Imai and Asano [IA84] consider, as well as other authors consider the running time of the Hopcroft and Ullman [HU73] to be $\mathcal{O}((k+m) \log^* k)$. As with the non incremental variants, the extension of the Gabow and Tarjan [GT85] structure admits a better running time of $\mathcal{O}(k + m)$, where k is the number of add operations and m is the number of split and find operations. Again the bound is achieved due to the use of a random access machine.

Circular Split-Find

The *circular Split-Find* is an extension of the *regular Split-Find* (non-circular Split-Find), where sets are considered to be cyclic rather than regular intervals, i.e., the first element is also adjacent to the last element in the set. Due to this cyclic nature the sets are split according to two split points i and j rather than a single one, as a single split point would not disconnect the cycle set. The new split operation is defined as follows: `split`(i, j) splits the set $S = \{a, ..., b, i, ..., j, c, ..., d\}$ into two disjoint sets $S_1 = \{a, ..., b, c, ..., d\}$ and $S_2 = \{i, ..., j\}$, where $a \leq b < i \leq j < c \leq d$. If $a = i$ or $j = d$, the set is split in the same way as the regular split, resulting in the sets $S_1 = \{i, ..., j\}$, $S_2 = \{c, ..., d\}$ for the first case and $S_1 = \{a, ..., b\}$, $S_2 = \{i, ..., j\}$ for the second case.

This new split operation slices the set into parts an outer and an inner portion rather than a lower and an upper portion, as is the case in the regular Split-Find. The start and end of the inner portion is given by the two split points i and j, with the outer portion consisting of the remaining elements. The Circular Split-Find presents a generalizaton of the regular Split-Find, as we may replicate the behaviour of the regular Split-Find by choosing the first or last element of the set as one of the split points, which is why the Circular Split-Find is also referred to as the *Generalized Split-Find Structure* by La Poutré [LP91]. As the name suggests, the Circular Split-Find is particularly useful for cases where the elements are cyclic, such as the vertices and edges in a cycle graph.

3.3 BC-Tree

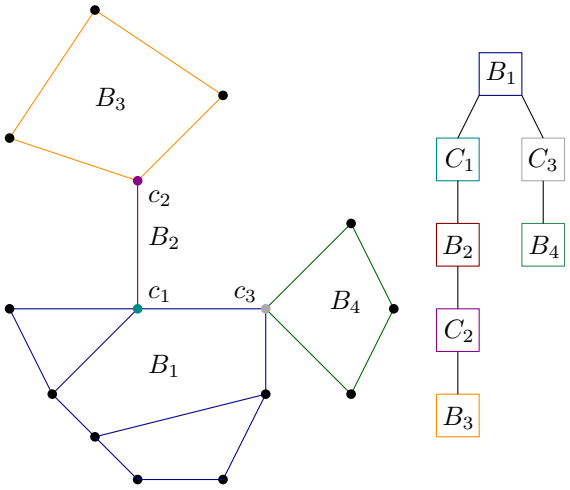


Figure 3.4: A BC-tree with the B-nodes and C-nodes colored according to their corresponding part of the graph.

A *block-cut-tree* or *BC-tree* is a graph decomposition, which decomposes a connected graph into cutvertices and biconnected components. The BC-tree consists of two node types: B-nodes, C-nodes. The B-nodes represent the biconnected components of the graph, whereas each C-node represents a distinct *cutvertex*, i.e., a vertex which upon deletion separates the graph into multiple connected components. The C-nodes are adjacent to the B-nodes corresponding to the split components resulting from the cutvertex of the respective C-node (see Figure 3.4).

Definition 3.1. *Let G be a connected Graph, we call T a BC-tree of G if it satisfies the following conditions.*

- *Each C-node corresponds to a unique cutvertex in G*

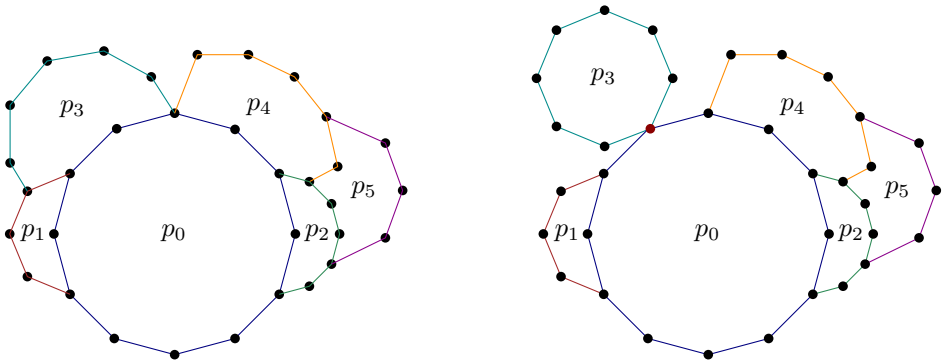
- *Each B-node corresponds either to a unique biconnected component or unique edge in G*
- *A B-node and a C-node are adjacent if they share a vertex.*
- *No two B-nodes are adjacent.*
- *No two C-nodes are adjacent.*

3.4 Ear-Decomposition

Definition 3.2. *Let $G = (V, E)$ be an undirected Graph. An* ear-decomposition *E of G is an edge partition into vertex-disjoint paths $p_0, p_1, ..., p_n$, called* ears*, such that p_0 is a cycle and each following path p_k is connected to previous paths p_i, p_j only with its endpoints, where $k > j \geq i \geq 0$ and $n \in \mathbb{N}_0$.*

A variant of the ear-decomposition that is better suited to our needs is the *open ear-decomposition*. The open ear-decomposition requires the ears, excluding the initial ear, to have distinct endpoints (see Figure 3.5). The open ear-decomposition has a one to one correspondence with biconnected graphs, that is, a graph G is biconnected if and only if it has an open ear-decomposition (see e.g. [Sch13]). We call an ear-decomposition that is not an open ear-decomposition a *closed ear-decomposition* (see Figure 3.5 for a comparison). The closed ear-decomposition have a similar correspondence with connectivity, though with the weaker 2-edge-connectivity rather than the stronger 2-vertex connectivity, i.e., biconnectivity (see e.g. [Sch13]).

Let $G = (V, E)$ be an undirected Graph. An *open ear-decomposition* is an ear-decomposition of G, with ears $p_0, p_1, ..., p_n$, such that for each $i \in \{1, ..., n\}$, the nodes of the path p_i are distinct.



Open ear-decomposition (closed) ear-decomposition

Figure 3.5: The two types of ear-decomposition. The right ear decomposition is not an open ear decomposition due to the cycle ear p_3. The closed ear-decomposition is not biconnected due to the cut vertex (marked in red) connecting p_3 to its previous ear p_0.

3.5 Ear Decomposition Algorithms

In the following we present different techniques for the computation of ear decompositions. Ear-decompositions can be computed in linear time using either depth-first search or a spanning tree. The time bound can be improved to logarithmic time, through parallelization with a linear number of processors (see e.g.[Ram92]).

3.5.1 Chain Decomposition

A simple greedy algorithm using depth-first search was proposed by Jens. M. Schmidt in 2013 [Sch13]. The algorithm was developed to be more easily understandable than earlier algorithms.

First, we perform a depth-first-search on the graph, where we partition the edges into *tree edges* and *back edges* and assign *depth-first-indices* to the nodes. Edges that are traversed during the depth-first-search form the tree edges, whereas the remaining edges are called back edges. The depth-first-search doubles as a check for connectivity, as constructing an ear-decomposition is not possible if the graph is not connected. We consider the tree edges to be directed towards the node and the back edges to be directed away from the node, that is an edge is directed toward the root if traversing the edge brings us to a node with a lower index.

We then traverse the back edge of each node in ascending order of the depth-first-indices, marking each passed node as *visited*. When we encounter a visited node we stop the traversal and proceed with the next back edge (see Figure 3.5.1). The path of each traversal is called a chain. These chains form the ears of an ear-decomposition if the graph is 2-edge-connected. We can easily check if the graph is 2-edge-connected, by keeping track of traversed edges, if there is an untraversed edge, the graph is not biconnected as removing the edge disconnects the graph. The resulting ear-decomposition is an open ear decomposition if the graph is biconnected (see [Sch13]).

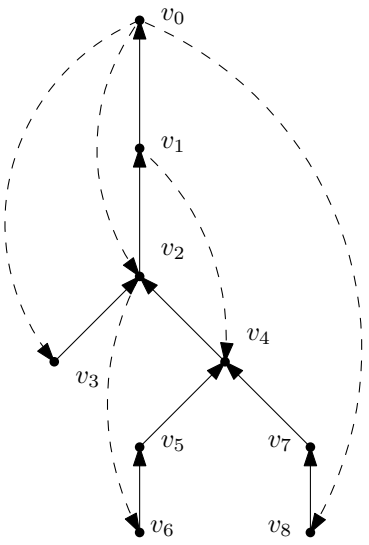


Figure 3.6: The resulting edge orientation of the chain decomposition, with the back edges drawn as dashed lines.

3.5.2 Ear Decomposition using Spanning Tree

In the following we present an earlier algorithm which uses a spanning tree rather than a depth-first search. The algorithm was proposed by Lovasz [Lov85] in 1985 and later improved for parallel efficiency by Miller and Ramachandran [MR87, Ram92].

First, we find a spanning tree, then we pick a vertex as the root and number the vertices of the tree in preorder starting from the root vertex with 0. The edges which are not part of the spanning tree are called *nontree edges*, similar to the back edges in the algorithm of Schmidt. In the next step, we assign ear numbers to the nontree edges, by labeling each nontree edge by its nearest common ancestor in the spanning tree. Once the nontree edges have been labeled, we sort the nontree edges in ascending order. Now, we extend

the numbering to the remaining edges, i.e., the *tree edges*. To extend the labeling we number each tree edge e by the smallest label of the nontree edge e' containing e in its fundamental cycle, that is the smallest simple cycle containing e'. The labeling can be achieved by, first, labeling each vertex v with the minimum label of all nontree edges which are incident incident to v and then assigning each tree edge $e = (a, b)$, the minimum label of all descendants of b. Lastly, the nontree edges labeled with 1 are relabeled with 0 instead. The resulting labeling indicates the ears to which each edge belongs.

Note, that the first ear, of the computed ear decomposition, is not a circle but rather a single edge or path, this is due to the earlier definitions of ear-decompositions which start with a single edge rather than a circle. To obtain an ear decomposition starting with a circle we can simply merge the first and second ear, i.e., edges labeled 0 and 1, which yields an ear-decomposition starting with a cycle, if the ear-decomposition is an open ear-decomposition. The algorithm runs in logarithmic time using a linear number of processors [Ram92].

4. SPQR-Tree Structure

An SPQR-tree is a decomposition of a biconnected graph, which decomposes the graph into its triconnected components. The tree consists of four different types of nodes: *S-nodes*, *P-nodes*, *Q-nodes* and *R-nodes*. Each node in the tree is associated with a biconnected multigraph, called a *skeleton* (see Figure 4.1). Two nodes in the SPQR-Tree are adjacent if and only if they share a *virtual edge*. The virtual edges represent a connection between two vertices, that exists in the original graph, but is not necessarily an actual edge in the graph. As such the virtual edge presents a contraction of a connected subgraph.

- Q-nodes are the simplest of the four node types. The skeleton of a Q-node consists of two edges, one edge representing the actual edge in the graph and one virtual edge.
- P-nodes represent pairs of triconnected vertices. The skeleton of a P-node consists of two vertices a, b and $k \geq 3$ parallel edges e_1, e_2, ..., e_k with endpoints a and b. P-nodes are sometimes called *bonds*.
- S-nodes represent *serial* parts of the graph. The skeletons correspond to cycles, which are sometimes referred to as *polygons*.
- R-nodes represent a simple triconnected or *rigid* component, their skeletons correspond to a simple triconnected subgraph.

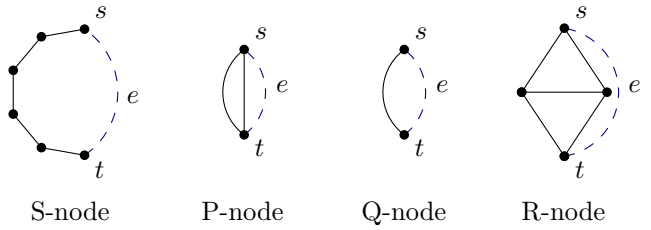


Figure 4.1: The skeletons of the four node types in the SPQR-tree. The reference edge e is shown as a dashed line.

The skeletons are obtained from the original graph by splitting the graph at a separation pair $\{a, b\}$ and adding a virtual edge $e = \{a, b\}$ to each of the resulting split graphs. The union of the skeletons, without the virtual edges yields the underlying graph. The virtual edges induce an acyclic graph, which forms the structure of the SPQR-tree. The root of the SPQR-tree is determined by the starting edge and the corresponding Q-node.

The SPQR-tree may be rooted at any Q-node, as the resulting trees are isomorphic and can be obtained by starting with a different edge [BT96]. For the recursive construction, we choose the first node as the root of the SPQR-tree. Rooting the SPQR-tree at a Q-node rather than at an S-node, P-node or R-node ensures the root node does not change with edge insertions, which is not necessarily the case for the other node types.

Omitting Q-nodes

The Q-nodes can easily be omitted from the SPQR-tree by distinguishing between virtual and *real edges* in the skeleton graphs. The real edges are the virtual edges corresponding to a Q-node. Gutwenger and Mutzel [GM01] have shown this notion to be equivalent. In the following we shall use the SPQR-tree without Q-nodes and use real edges instead (see Figure 4.2 for an SPQR-tree without Q-nodes).

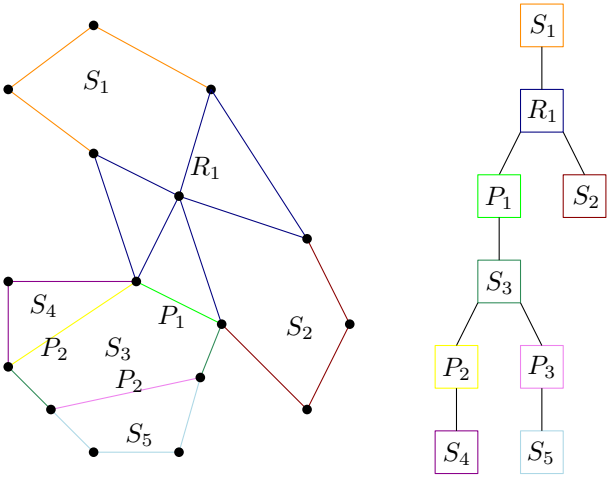


Figure 4.2: A graph with its corresponding SPQR-tree. The tree consists of five S-nodes $S_1, ..., S_5$, two P-nodes P_1, P_2, P_3 and a single R-node R_1. The edges of the graph are colored according to the component containing the edge.

Supported Operations

The SPQR-tree as described by Di Battista and Tamassia [BT96] supports the following operations:

- `triconnected`(a, b), which checks if the vertices a and b are triconnected.
- `insertEdge`(a, b), which inserts an edge between a and b.
- `splitEdge`(e, v), which splits the edge $e = \{a, b\}$ into two edges $e_1 = \{a, v\}$, $e_2 = \{v, b\}$ sharing the vertex v.

Definition 4.1. *Let $G = (V, E)$ be a biconnected graph. We call* (V_{skeleton}, E_{real}, E_{virtual}, type)*, with*

$$\begin{aligned} V_{\text{skeleton}} &\subseteq V, \\ E_{\text{real}} &\subseteq E, \\ E_{\text{virtual}} &\subseteq \{\{a, b\} \mid a, b \in V\}, \\ \text{type} &\in \{S, P, R\} \end{aligned}$$

a skeleton *of G if the* skeleton graph $G' = (V_{\text{skeleton}}, E_{\text{real}} \cup E_{\text{virtual}})$ *is biconnected.*

Each skeleton is itself a graph consisting of the skeleton vertices V_{skeleton} and the skeleton edges $E_{\text{real}} \cup E_{\text{virtual}}$, which distinguished into real edges, edges that exist in the original graph, and virtual edges, edges that are not part of the original graph, but represent a contracted part of the original graph.

Definition 4.2. *Let $G = (V, E)$ be a biconnected graph, N a set of skeletons of G and*

$$\begin{aligned} \text{realVertex} &: \bigcup_{\lambda \in N} V_{\text{skeleton}}(\lambda) \to V, \\ \text{realEdge} &: \bigcup_{\lambda \in N} E_{\text{real}}(\lambda) \to E, \\ \text{twinEdge} &: \bigcup_{\lambda \in N} E_{\text{virtual}}(\lambda) \to \bigcup_{\lambda \in N} E_{\text{virtual}}(\lambda) \end{aligned}$$

we call $T = (G, N)$ an SPQR-tree *of G if it satisfies all of the following conditions:*

1. *Vertex complete:* $\bigcup_{\lambda \in N} V_{\text{skeleton}}(\lambda) = V$.
2. *Edge complete:* $\bigcup_{\lambda \in N} E_{real}(\lambda) = E$
3. *Same endpoints for twins:* $\forall e, e' \in E' : \text{twinEdge}(e) = e' \Rightarrow \text{realVertex}(\{a, b\}) = \text{realVertex}(\{a', b'\})$, *where* $e = \{a, b\}, e' = \{a', b'\}$ *and* $E' = \bigcup_{\lambda \in N} E_{\text{virtual}}(\lambda)$
4. *Virtual edge pairs:* $\forall e = \{a, b\} \in E' : \text{twinEdge}(\text{twinEdge}(e)) = e$, $\text{twinEdge}(e) \neq e$
5. *P-node skeleton:* $\forall \lambda \in N : \text{type}(\lambda) = P \Rightarrow |V_{\text{skeleton}}(\lambda)| = 2$ *and* $|E_{virtual}(\lambda) \cup E_{\text{real}}(\lambda)| \geq 3$
6. *R-node skeleton:* $\forall \lambda \in N : \text{type}(\lambda) = R \Rightarrow G' = (V_{\text{skeleton}}(\lambda), E_{\text{virtual}}(\lambda) \cup E_{\text{real}}(\lambda))$ *is a simple triconnected graph.*
7. *S-node skeleton:* $\forall \lambda \in N : \text{type}(\lambda) = S \Rightarrow G' = (V_{\text{skeleton}}(\lambda), E_{\text{virtual}}(\lambda) \cup E_{\text{real}}(\lambda))$ *is a cycle.*
8. *The Graph* $G' = (N, E')$ *is a tree graph, where* $E' = \{\{\lambda, \mu\} \mid \lambda, \mu \in N, \exists e_1 \in \lambda(E_{\text{virtual}}), e_2 \in \mu(E_{\text{virtual}}) : \text{twinEdge}(e_1) = e_2$.
9. *No two P-nodes are adjacent.*
10. *No two S-nodes are adjacent.*

The conditions 1 and 2 are fairly simple. An SPQR-tree contains all vertices and edges of the original graph and no other vertices or edges. Conditions 3 and 4 give the requirements for the tree edges. A pair of two distinct virtual edges forms a tree edge. The tree edges are represented by distinct pairs of virtual edges. Each pair consists of two virtual edges e_1 and e_2 which correspond to each other, i.e., $\text{twinEdge}(e1) = e2$ and vice versa. Two edges in such a pair share the same endpoints in the original graph. To avoid self edges we require the two edges to be distinct. The three conditions 5, 6, and 7 give the requirements for the skeletons of the different node types as outlined at the beginning of the Chapter 4. The two conditions 9 and 10 the requirement for R-node skeletons to not contain parallel edges (see condition 6) are not strictly necessary for the structure, but ensure that the SPQR-trees of a graph are unique up to isomorphism (see [HR19] or Section 6.3.2 for a relaxed SPQR-tree without the adjacency conditions).

4.1 Recursive Construction

The SPQR-tree was originally defined by Di Battista and Tamassia [BT96] via a recursive construction starting with an edge of an adjacent separation pair, which forms the root Q-node of the resulting SPQR-tree. The authors use a slightly different notion of the the separation pair, called a *split pair* which allows a single edge to be split off.

Definition 4.3. *Let G be a biconnected graph. A* split-pair $\{s, t\}$ *of G is either a separation pair or a pair of adjacent vertices, that is there exists an edge $e = \{s, t\} \in E$.*

For the consruction of the SPQR-tree, let $G = (V, E)$ be a biconnected graph, $s, t \in V$ be a split pair with $e = \{s, t\}$. We call this e the reference edge. The *SPQR-tree* T of the graph G with respect to the reference edge e is recursively defined as follows:

1. Trivial Case: If G consists of exactly two parallel edges between s and t, then T consists of a single Q-node, with G as the associated skeleton.
2. Parallel Case: If the Split pair $\{s, t\}$ has at least three split components $C_1, C_2, ..., C_k$, with $k \geq 3$, then the root r of T is a P-node, whose skeleton consists of k parallel edges $e_1, e_2, ..., e_k$ between s and t, with e being the first edge, i.e., $e = e_1$.
3. Series Case: If the split pair $\{s, t\}$ has exactly two split components, one of the components is the reference edge e and we refer to the other component as G'. If the component G' has cutvertices $c_1, ..., c_{k-1}$, that partition G into blocks $G_1, ..., G_k$, with $k \geq 2$, then the root of T is an S-node with the cycle $e_0, e_1, ..., e_k$, where $e_0 = e$ and e_i connecting the nodes c_{i-1} and c_i, with $c_0 = s$, $c_k = t$ and $i \in \{1, ..., k\}$.
4. Rigid Case: If none of the previous cases applies let $\{s_1, t_1\}, ..., \{s_k, s_t\}$ be the maximal split pairs of G with respect to $\{s, t\}$, $i \in \{1, ..., k\}$ and let G_i be the union of all split components of the pair $\{s_i, t_i\}$, except the component containing the reference edge e. The root of T is an R-node. We obtain the skeleton by replacing the subgraph G_i with the edge e_i between s_i and t_i.

4.2 SPQR-Trees on General Graphs

While the SPQR-tree requires the underlying graph to be biconnected, the structure can easily be extended to support general graphs, i.e., graphs that are not biconnected. Fundamentally, to maintain the triconnected components of a connected graph, we maintain the biconnected components of the graph and maintain the triconnected components within each biconnected component using an SPQR-tree. To maintain the biconnected components of a connected graph we use a BC-tree (see 3.3) and a SPQR-tree to maintain the triconnected components of each B-node.

For non-connected graphs we simply use multiple BC-trees, i.e., a BC-forest, where each connected component is represented by a BC-tree (see [BT96]). Using a BC-tree extends the SPQR-tree operations, allowing vertices to be inserted with a single edge rather than by splitting an existing edge, as the BC-tree only requires simple connectivity. We call this new operation `attachVertex`(a, v), which inserts the new vertex v and the edge $\{a, v\}$. Similarly, using a BC-forest allows to further extend the operations of the SPQR-tree with the operation `insertVertex`(v), which inserts the isolated vertex v.

4.3 Cycle Trees

A similar structure to the SPQR-tree is the *cycle tree* proposed by J.A. La Poutré in 1992 [Pou92, Pou94]. The nodes of the cycle tree are called *cycles*, *bars* and *3vc-classes*. The 3vc-classes represent the triconnected components of the graph, like the R-nodes in the SPQR-tree. The cylce nodes represent the biconnected components of the graph, similar to the S-nodes in the SPQR-tree. The bars represent the separation-pairs of the graph and are further divided into *3vc-bars* and *cycle bars*. The 3vc-bars are the separation-pairs that are triconnected. The 3vc-bars are similar to the P-nodes in the SPQR-tree. Each 3vc-bar is part of exactly one 3vc-class in the cycle tree. The cycle bars represent real edges of

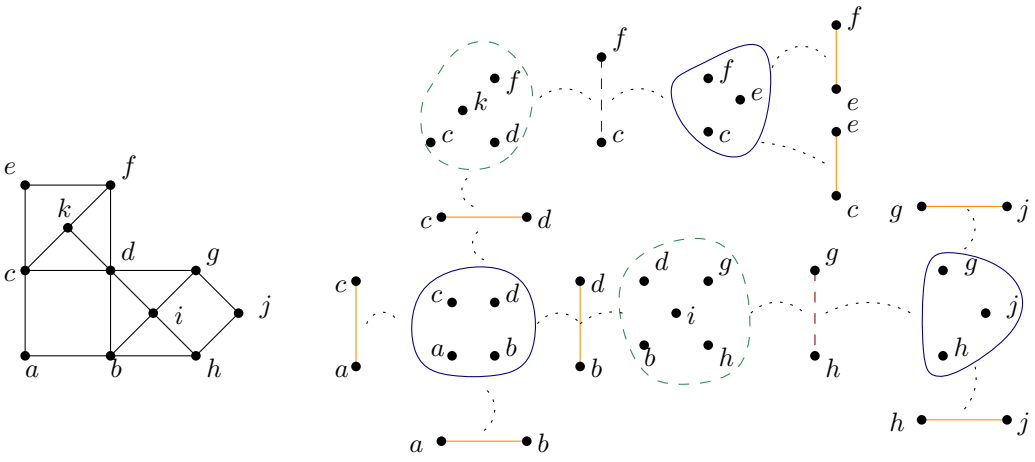


Figure 4.3: A biconnected graph with its coresponding cycle tree. The 3vc-classes (in green) and 3vc-bars (in red) are drawn with dashed lines and the cycle nodes (in blue) and cycle bars (in orange) are drawn with solid lines. The links between the bars and classes are drawn using dotted lines like the tree edges in the SPQR-tree.

the cycle nodes, similar to the Q-nodes in the SPQR-tree. However, unlike the Q-nodes, the cycle bars also serve as links between the adjacent cycle nodes and 3vc-classes, that is, cycle bars as well as 3vc-bars may be adjacent to multiple cycle node and 3vc-classes. As such the cycle bars may be seen as P-nodes with a real edge or adjacent Q-node (see Figure 4.3 for an example of a cycle tree).

Like the SPQR-tree, the construction of the cycle tree uses recursion by splitting the graph on the separation-pairs. However, unlike the SPQR-tree, the cycle tree is only split on triconnected separation-pairs instead of the biconnected separation-pairs.

If all nodes in the graph are triconnected, then the corresponding cycle tree consists of a single 3vc-class. Likewise, if the graph is a simple cycle, then the corresponding cycle tree consists of a single cycle node with the edges as cycle bars. If the graph is neither fully triconnected, nor a simple cycle, then there is a triconnected separation-pair $\{a, b\}$.

Deleting a and b from the graph induces multiple connected subgraphs $H_1, ..., H_k$, where $k \in \mathbb{N}$. Let $H'_i = (V_i \cup \{a, b\}, E_i \cup \{a, b\})$, for $i \in \{1, ..., k\}$. The cycle trees are recursively constructed for each of these subgraphs.

Edge insertions on cycle trees, between nodes that are not triconnected, are processed in a similar way to the edge insertions in the SPQR-tree (which are presented in Chapter 6). To insert an edge between two non triconnected nodes a and b, we obtain the tree path between the vertices a and b. The 3vc-classes and bars on the path and the vertices a and b are merged into a new 3vc-class. The cycle nodes on the path, like the S-nodes in the SPQR-tree, are split into new cycles by the nodes in the bars on the path. Unlike the SPQR-tree, the cycle tree does not support edge splits or any other way of inserting a new vertex. Similarly, the edges inside the 3vc-classes are also not maintained.

Extension to General Graphs

To extend the cycle trees to general graphs, La Poutré [Pou92] proposed further structures, such as the *cluster partition* the *rooted cluster tree* and the *dynamic microset* (see [LP93] for more information on the dynamic microset). The rooted cluster tree with the cluster partition restructures the components of the cycle tree, whereas the dynamic microset is an independent structure based on the *microset* introduced by Gabow and Tarjan [GT85]

(see Section 3.2). The dynamic microset is used to effiently join clusters similar to the microset in the Union-Find and Split-Find structure of Gabow and Tarjan [GT85]. La Poutré suggested that the running time of $\mathcal{O}(n + m \cdot \alpha(n, m))$ for the rooted cluster tree is optimal for pointer machines, where n is the number of vertices and m is the number of edges.

A cluster partition of a biconnected graph is a non-disjoint partition of the vertex set of the graph into *clusters*, such that each cluster contains at least three nodes of the graph and shares at most two nodes with every other cluster. For a cluster C, the set of triconnected node pairs is referred to as not3bar(C). Furthermore, for a cluster C the subgraph induced by its vertices has to be connected and for any node $a \in C$ there are at most two pairs with a in not3bar(C), that is for $\{a, b\}, \{a, c\}, \{a, d\} \in$ not3bar$(C), b = c$ or $b = d$. A 3vc-class is considered to occur in a cluster if the cluster contains at least three nodes of the 3vc-class. A 3vc-class that occurs in multiple clusters is called a *multiple class*. Similarly, nodes that occur in multiple clusters are called *multiple nodes*. The nodes of a cluster induce a subtree of the cycle tree, this subtree is referred to as the *local tree* of the cluster. The cluster partition satisfies the following four constraints (c.f. [Pou92]):

- For every 3vc-class K and nodes $a, b, c \in K$, there is a cluster C, such that $a, b, c \in C$.
- For every separation-pair $\{a, b\}$ in the graph, there is some cluster C, such that a and b are part of the subgraph induced by C.
- For every Cluster C and triconnected node pair $\{a, b\} \in$ not3bar(C), $\{a, b\}$ is not a 3vc-bar of the subgraph induced by C.
- For every node pair $\{a, b\}$ in the graph, there is at most one cluster C, with $a, b \in C$ and $\{a, b\} \notin$ not3bar(C).

The *rooted cluster tree* combines the concept of the cycle tree with the cluster partition in a tree structure, where the nodes of the rooted cluster tree are the clusters, multiple 3vc-classes and the 3vc-bars of the cycle tree.

For the general case La Poutré maintains the triconnected components of each biconnected component in the graph, similar to how BC-forests are used to maintain the multiple biconnected components of a general graph. For the maintenance of the biconnected components La Poutré distinguishes between *small* and *large* components. The small components are components of size $\mathcal{O}(\log \log n)$ and the large components have size $\mathcal{O}(\log n)$, where n is the number of nodes in the graph. The idea behind this separation is similar to the subdivision Gabow and Tarjan [GT85] used for Union-Find and Split-Find (c.f. Section 3.2). For the small components *dynamic microsets* are used. The larger components are maintained using the cluster trees, where Union-Find structures are used for the 3vc-classes and Split-Find structures for the cycles in the local cycle trees of each cluster. To join biconnected components of the same size class auxiliary edges are inserted between the *connection nodes*, i.e., common nodes, of the joining clusters, creating a new cycle tree consisting of the old cycle trees and the newly created cycle, with the clusters combined into a new cluster partition.

The overall structure maintains the triconnected components of a general graph in $\mathcal{O}(n + k \cdot \alpha(n, k))$ time, where n is the number of nodes in the graph and k is the number of insertions and query operations [Pou92].

While the Cycle trees were not developed further, some of the improvements of La Poutré [Pou92] can be integrated into the SPQR-tree, by using Union-Find structures for the R-nodes and Split-Find structures for the S-nodes. Di Battista and Tamassia [BT96] suggested that by using Split-Find and Union-Find the running time of k operations on an SPQR-tree starting from a triangle graph could be reduced to $\mathcal{O}(k \cdot \alpha(n, k))$. The authors

argue that this is possible due to the lack of interaction between the Union-Find and the Split-Find the SPQR-tree and suggest the use of a Split-Find structure which supports insertions, finds and splits in amortized $\mathcal{O}(1)$.

However, the Split-Find structure of Imai and Asanao [IA84] referenced by Di Battista and Tammasia [DBT96] only allows element insertions at the end of a set. To the extent of our knowledge no Split-Find structures which allows arbitrary insertions in amortized $\mathcal{O}(1)$ exists. As such insertion of an arbitrary vertex inside an S-node without breaking the corresponding Split-Find structure, like the one prroposed by Imai and Asano [IA84] is non trivial. It is possible remedy this problem by reassigning the Split-Find ids of the S-node links, but doing so would likely break the $\mathcal{O}(k \cdot \alpha(n, k))$ bound. Alternatively, it is possible to bypass the problem using a construction which only inserts vertices at the end of an S-node, like the construction using ear-decomposition (see Section 5.1). Furthermore, the cycle trees and cluster trees of La Poutré [Pou92] only allow edge insertions, as such it is unclear whether the $\mathcal{O}(k \cdot \alpha(n, k))$ bound could still be achieved if edge splits are allowed.

5. Efficient SPQR-Tree Construction

The SPQR-tree of a graph can be constructed in various different ways all of which result in isomorphic SPQR-trees, that is, the resulting SPQR-trees may be rooted at different SPQR-nodes, but may be rerooted to obtain the same SPQR-tree. In the following we describe the incremental construction of the SPQR-tree using ear-decomposition and the linear-time construction proposed by Gutwenger and Mutzel [GM01, Gut10] in 2001.

5.1 Construction using Ear-Decomposition

Ear decompositions provide a natural, incremental construction of the SPQR-tree, using only the SPQR-tree operations `insertEdge` and `splitEdge` starting from a single S-node with three vertices. The idea was proposed by Di Battista and Tamassia [DBT96], who used it to show that the SPQR-tree of any biconnected graph $G = (V, E)$ can be constructed using $|E| - 3$ `insertEdge` and $|V| - 3$ `insertVertex` operations starting from a triangle graph.

To construct the SPQR-tree from an open ear-decomposition $E = e_0, e_1, ..., e_k$, we first construct the S-node for the first ear. Since the first ear is a cycle the S-node is a perfect match for the ear. The real edges of the S-node are given by the edges of the ear. For the following ears, we first insert an edge covering the whole ear and then split the edge for each inner vertex on the ear (see Figure 5.1).

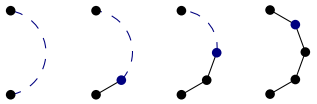


Figure 5.1: Insertion of an ear, an edge is inserted between the two endpoints and then continually split until all vertices of the ear have been inserted.

Ear decompositions are not necessarily unique, as such it is possible that the first ear only consists of two edges. In this case using the ear for the starting node is not possible, as S-nodes require at least a three vertices. There are two cases where this can occur

1. The underlying graph consists only of two vertices with parallel edges between them. In this case no corresponding SPQR-tree can exists, as the graph is not biconnected.
2. If the graph has more than two vertices, then there is at least one additional vertex v and an ear e_i connecting v with the two vertices of the first ear. In this case we can

move the last edge of the first ear e_0 to the ear e_i, turning the ear e_i into a cycle and the ear e_0 into a simple ear. Swapping the two ears e_0 and e_i yields an equivalent ear decomposition, which starts with a cycle and satisfies the ear-decomposition requirements, as the requirements of any ear which depends on the original ear e_0 is also satisfies by the swapped ear e_i, since all vertices of e_0 are also contained in e_i.

Alternatively, we may run the ear-decomposition on the corresponding simple graph. Each parallel edge forms a simple ear consisting solely of a single edge. We may place these parallel edge ears at the end of the ear list obtained from the simple graph, as the vertices of the parallel edges are already present in previous ears.

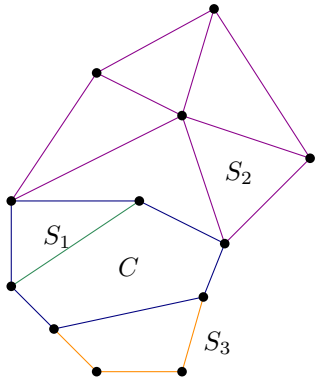
The construction requires k edge insertions, one for each ear after the first, and up to $n-3$ edge splits, one for each unique vertex in an ear.

In practice it may be better to construct the complete S-node of an ear, rather than starting from a triangle graph and splitting the edge to insert the remaining vertices. Similarly, the first ear may also be constructed as a completed S-node. This not only reduces the number of operations necessary to insert an ear, but also allows the use of a faster non incremental Split-Find for further operations, as Split-Find generally does not alllow arbitrary vertex insertions. However, the number of operations required is still dependent on the size of the ear.

5.2 Linear-Time Construction

Gutwenger and Mutzel [GM01, Gut10] devised an efficient algorithm to construct the SPQR-tree of a biconnected graph in linear time. The algorithm is based on and corrects the mistakes of the Hoprcoft and Tarjan [HT73] algorithm for finding triconnected components.

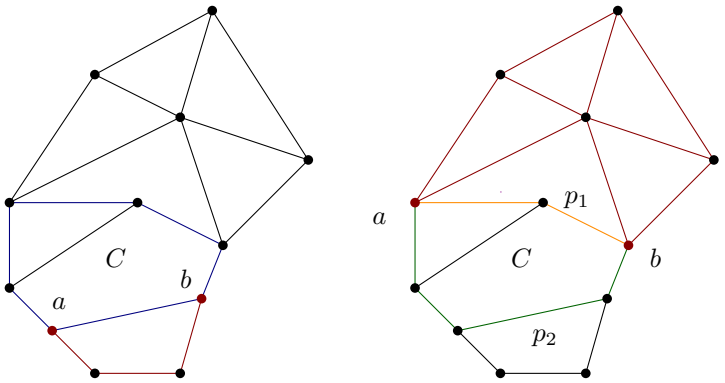
5.2.1 Overview



Cycle C with segments S_1, S_2, S_3

Figure 5.2: The cycle C with its relative segments S_1, S_2, S_3. The cycle is shown in blue.

The main idea behind the algorithm is to find the separation pairs of the graph to compute the split components. Gutwenger and Mutzel [GM01] use the idea of Hopcroft and Tarjan [HT73] to obtain the separation pairs of the graph by finding a cycle and its segments (see Figure 5.2). The idea is similar to the computation of ear-decompositions, with the cycle as the initial ear and the split components as the ears (see 3.5.1). The difference lies in the handling of the segments, which are split off similar to the original definition given by Tutte [Tut66]. The following theorem shows the relation between the separation pairs and the segments of cycles.



type-1 separation pair type-2 separation pair

Figure 5.3: The two types of separation pairs, with the split off segment and the two separation vertices a and b shown in red. The cycle C is shown in blue for the type-1 separation pair. For the type-2 separation pair the Cycle is divided into two paths p_1 and p_2 which are shown in orange and green respectively.

Theorem 5.1. *(Gutwenger, Hopcroft and Tarjan [Gut10, HT73]) Let $G = (V, E)$ be a biconnected graph, C a cycle in G and $S_1, ..., S_n$ the connected components of $G \setminus C$, called* segments *relative to C and let $a, b \in V$. Then each pair $\{a, b\}$ is a separation if and only if it satisfies all of the following conditions.*

- *a and b lie on the cycle C or a and b are part of the same segment S_i, where $i \in \{1, ..., n\}$.*
- *If a and b lie on the cycle c, let p_1, p_2 be the two paths in C connecting a and b (see Figure 5.3). Then*
  - *Type 1: there exists a segment $S_i \in \{S_1, ..., S_n\}$, such that S_i has at least two edges, $S_i \cap C = \{a, b\}$ and $S_i \setminus \{a, b\} \neq \emptyset$.*
  - *Type 2: $\forall S_i \in \{S_1, ..., S_n\} : S_i \cap p_1 \subseteq \{a, b\}$, $S_i \cap p_2 \subseteq \{a, b\}$ and there exist vertices $c \in p_1$ and $d \in p_2$ such that $\{c, d\} \cap \{a, b\} = \emptyset$.*
- *If a and b do not lie on the cycle C, then there exists some other cycle C' in G such that the above case holds.*

Recursively subdividing the graph into a cycle and the relative segments allows us to find the separation pairs. The idea originates from Auslander and Parter [AP61] as well as Goldstein [Gol63] (c.f. [Gut10]).

The algorithm requires a biconnected Graph G, without self loops, as self loops are not supported by the SPQR-tree. The entire algorithm consists of six major steps, which are as follows.

1. Remove parallel edges by replacing them with virtual edges.
2. Perform a depth-first-search and partition the edges into tree edges and back edges.
3. Compute a set of paths covering the dfs-tree, consisting of multiple tree edges followed by a single back edge, similar to the ears in the ear-decomposition (see 3.5.1).
4. Check potential separation pairs and construct the split components with tree edges between adjacent components.
5. Merge adjacent polygons and merge adjacent bonds.

6. Direct the SPQR-tree by rooting it at the SPQR-node containing the reference edge

The components are created by the function `newComponent`$(e_1, ..., e_k)$, which assigns the edges to a new component C and removes the edges from the graph. The type of the component may be detected from the edges, i.e., a polygon only has degree 2 vertices, as such each edge in shares its endpoints with exactly two other edges. The bonds consists of parallel edges with the same endpoint. The simple triconnected component has at least degree 3 at each of its vertices and has at least four vertices. As such the type of a component may be determined by examining only a constant number of vertices or edges. To add an edge e to a component C we use the notation $C := C \cup \{e\}$, as with the creation of components we remove the add edge from the graph.

5.2.2 Preparation

Before we can compute the separation pairs we first have to prepare the graph and pre-compute several values. To prepare the graph for the split component computation, we split off the parallel edges. Once the parallel edges have been removed we perform a depth-first-search to calculate an ordering which will help us in the detection of separation pairs.

Replacing Parallel Edges

Algorithm 5.1: Simplification of the Graph by splitting off parallel edges.

Data: A biconnected Graph $G = (V, E)$
Result: The simple Graph $G' = (V, E')$, where parallel edges are replaced by a single edge

```
1 splitBonds
2     sort edges such that parallel edges are grouped together
3     for each maximal bundle of parallel edges e_1, ..., e_k ≥ 2 do
4         let e_1, ..., e_k be edges incident to v and w
5         Replace e_1, ..., e_k by a new edge e' := (v, w, l), where l is a new label.
6         C := newComponent(e_1, ..., e_k, e')
```

The first step in the linear time construction algorithm is to simplify the Graph $G = (V, E)$ by replacing parallel edges with a single edge (see Algorithm 5.1). We call the simple Graph $G' = (V, E')$. To find the parallel edges, we sort the adjacency lists of G, such that parallel edges are grouped together. To achieve the grouping we first sort the edges according to the index of the lower endpoint and then use a stable sort according to the index of the higher endpoint. If the vertices are not numbered we simply label the vertices in some order from $1, ..., |V|$ (c.f. [Gut10]).

Depth First Searches

For the linear-time construction Gutwenger [Gut10] uses multiple depth-first searches. The depth-first search ranks the vertices in the order of traversal from 1 to $n = |V|$. We call the edges, which are traversed during the dept-first-search *tree edges* if they lead to an unranked vertex and *back edges* if they lead back to an already ranked vertex, similar to the chain decomposition algorithm (c.f. Section 3.5.1). Gutwenger and Mutzel [GM01, Gut10] use the notation of Hoprcoft and Tarjan [HT73], calling the tree edges *tree-arcs* or *T-arcs* and the back edges *B-arcs*.

The first procedure *DFS1* (see Algorithm 5.2) computes two values lowpt1 and lowpt2 for each vertex $v \in V$, using a simple depth first search (see Figure 5.4 for an example of the

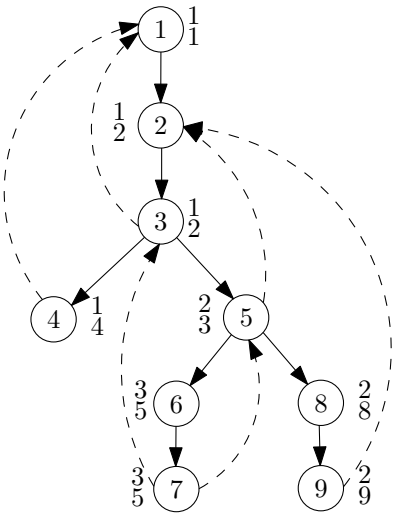


Figure 5.4: The DFS-tree with the back edges drawn as dashed lines and the lowpt values next to each vertex, with lowpt1 at the top and lowpt2 at the bottom. Notice the similarity to the DFS-tree of the chain decomposition 3.5.1. In fact, ordering the paths found by the algorithm in the order they are found, satisfies the requirements of an open ear-decomposition.

resulting dfs-tree). The value lowpt1(v) is the rank of the lowest ranked vertex reachable from v by traversing any number of tree edges followed by at most one back edge. This includes v itself. Note, that traversing a tree edge always leads to a higher ranked vertex whereas traversing a back edge always leads to a lower ranked vertex, which is why the traversal allows an arbitrary number of tree edges but only one back edge.

The value of lowpt2(v) is the second lowest ranked vertex reachable that is different from lowpt1(v) or rank(v) if no lower ranked vertex is reachable. Hence, the values of lowpt1(v) and lowpt2(v) are only the same if v is the lowest vertex reachable. This is always the case for the starting vertex as no other vertex may be lower ranked. The lowpt1 and lowpt2 value of each vertex v, is at least the respective lowpt value of its parent vertex in the dfs-tree (c.f. [Gut10]).

More formally, the values of lowpt1 and lowpt2 of a vertex $v \in V$ are defined as follows.

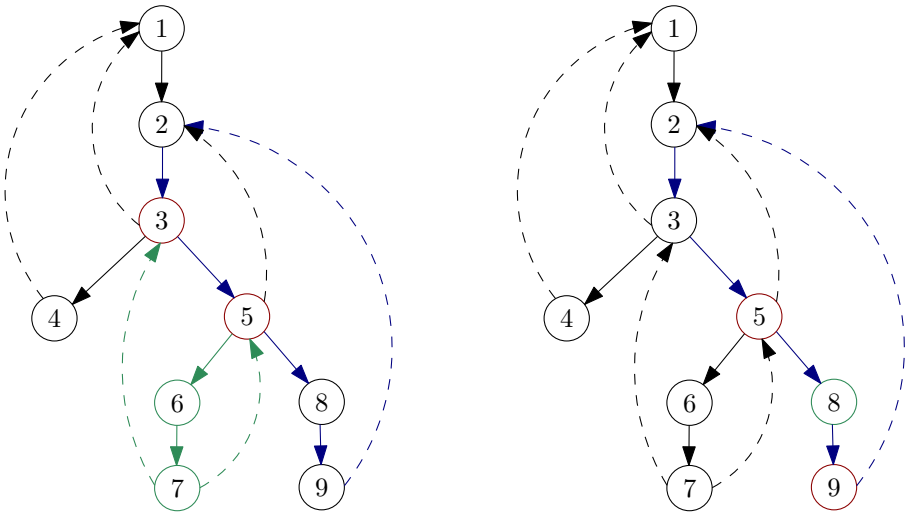
$$\text{lowpt1}(v) := \min(\{\text{rank}(w) \mid v \overset{*}{\rightarrow}\hookrightarrow w\} \cup \{\text{rank}(v)\}),$$
$$\text{lowpt2}(v) := \min(\{\text{rank}(w) \mid v \overset{*}{\rightarrow}\hookrightarrow w\} \setminus \text{lowpt1}(v) \cup \{\text{rank}(v)\}),$$

where $v \overset{*}{\rightarrow}\hookrightarrow w$ signifies a path of zero or more tree edges starting from v followed by a single back edge ending at w (see [Gut10]).

The lowpoints allow us to find the type-1 and type-2 separation pairs (see Figure 5.5). The following lemma shows the conditions with regards to the lowpoints, satisfied by the different types of separation pairs type-1, type-2 and parallel edge separation pairs.

Lemma 5.2. *(Gutwenger, Hopcroft and Tarjan [Gut10, HT73]) Let $G = (V, E)$ be a bicconected graph, $a, b \in V$ with* rank(a) < rank(b)*, where* rank(v) *is the rank of v in a depth first search. Then, $\{a, b\}$ is a separation pair if and only if it satisfies one of the following three conditions.*

- *Type-1: There exist distinct vertices $v, w \in V \setminus \{a, b\}$, such that v is a direct descendant of b, with* lowpoint1(v) = a, lowpoint2(v) $\geq b$ *and w is also a (not necessarily direct) descendent of b, but is not a descendant of v.*



type-1 separation pair type-2 separation pair

Figure 5.5: The two separation pair types, with the type-1 separation pair to the left and the type-2 separation pair to the right. The cycle is shown in blue, with the separation pair nodes in red and the split off segment in green.

- *Type-2: a is not the root vertex and there exists a vertex $v \in V \setminus \{b\}$, such that v is a direct descendant of a, i.e., $a \to v$, and b is a (not necessarily direct) descendant of v, i.e., $v \overset{*}{\to} b$.*
  - *For every back edge $x \hookrightarrow y$ with $\text{rank}(v) \leq \text{rank}(x) < \text{rank}(b)$, $\text{rank}(a) \leq$ y holds.*
  - *For every back edge $x \hookrightarrow y$ with $\text{rank}(a) < \text{rank}(y) < \text{rank}(b)$ and every vertex v in the tree edge path $b \to v \overset{*}{\to} x$, $\text{w} \geq \text{rank}(a)$ holds.*
- *Parallel Edge Case: There are parallel edges between a and b in G and G consists of at least four edges.*

Furthermore, the values lowpt1 and lowp2 are used to define an ordering $\phi : E' \to \mathbb{N}$ on the edges $e \in G'$,

$$e = (v, w) \mapsto \begin{cases} 3 \cdot \text{rank}(w) + 1, & \text{if } e \text{ is a back edge} \\ 3 \cdot \text{lowpt1}(w), & \text{if } e \text{ is a tree edge and } \text{lowpt2}(w) < \text{rank}(v) \\ 3 \cdot \text{lowpt1}(w) + 2, & \text{if } e \text{ is a tree edge and } \text{lowpt2}(w) \geq \text{rank}(v) \end{cases}$$

The ordering is used to sort the adjacency entries for each vertex, which is required for the later separation pair detection. Similarly, the vertices are also reordered such that for each vertex v with children $w_1, ..., w_n$ in the dfs tree, $\text{rank}(w_i) = v + \text{descendants}(w_{i+1}) + ... + \text{descendants}(w_n) + 1$. For the root we keep the rank as 1. The new ordering is computed in a second dfs (see Algorithm 5.3).

The second depth-first-search traversal (see Algorithm 5.3) identifies the paths in the dfs-tree, similar to the identification of the ears in the chain decomposition algorithm (see Section 3.5.1). Furthermore, the second dfs renumbers the vertices and computes the highpoints of each vertex. Unlike the name suggests, the highpoint of a vertex v is the list of descendants of v which have a back edge leading to v, rather than just a single vertex. This is a remnant of the original Hopcroft and Tarjan algorithm [HT73] which incorrectly used a single vertex rather than maintaining the list of vertices. The old highpoint of

Hopcroft and Tarjan has been replaced by the corresponding list and a function `high` which returns the first element of the list (see Algorithm 5.4).

The function `makeTreeEdge`$(e, v \to w)$ makes e a new tree edge in the dfs-tree leading from v to w.

5.2.3 Split Component Computation

The split component computation is the most intensive step in the algorithm. The split components are computed in the path search procedure (see 5.4) and the type-1 (see Algorithm 5.6) and type-2 checks (see Algorithm 5.5). The path search uses two stacks, TStack and EStack. The latter contains the edges that have been visited but have yet to be assigned to a component. The former contains triples of vertices (h, a, b) where $\{a, b\}$ is a potential type-2 separation pair and h is the highest numbered vertex in the component to be split off. Neither Gutwenger and Mutzel [GM01, Gut10] nor Hopcroft and Tarjan [HT73] give an algorithm for the calculation of the triples.

The triples may be obtained in yet another depth first search similar to the second one 5.3, where we traverse the path and add the triple (h, a, b) to the stack, where a and b are two distinct vertices of the path and h is the highest vertex to be split off. We can obtain this highest numbered vertex in the split component through the highpoints of the path vertices, which where computed earlier in the second depth first search 5.3. We use an end of stack symbol $\perp$ to separate the different paths on the TStack, the technique is mentioned in Hopcroft and Tarjan [HT73], using the three letters *eos* as the end of stack symbol.

For the sake of readability in the pathsearch and type-1 and type-2 checks, we use the vertices by their rank, e.g., lowpt1$[v]$ may refer to the vertex w with rank$[w]$ = lowpt1$[v]$ as well as the numerical value of lowpt1$[v]$.

5.2.4 Merging Components

The last step of the algorithm is to merge the adjacent bonds as well as the adjacent polygons. This directly corresponds to the requirement of the SPQR-tree not to have any adjacent P-nodes or S-nodes (c.f. Chapter 4). We consider two components adjacent if they share a virtual edge (see Algorithm 5.7). The type of a component is implicitly given by its skeleton. The polygons are cycles with at least three vertices and no parallel edges, whereas the bonds have exactly two vertices and at least three edges.

Combining the previous sub-algorithms we obtain the linear time construction (see Algorithm 5.8). The path search algorithm (see Algorithm 5.4) creates the majority of the components, but leaves the edges of the last component on the stack. This is a remnant of the original Hopcroft and Tarjan algorithm [HT73], where the last component was overlooked. The end of stack symbol $\perp$ is pushed on the triple stack for consistency as the different paths are all separated by an end of stack symbol, which is removed each time a new path is visited.

Algorithm 5.2: DFS calculating rank, lowpt values and the number of descendants for each vertex.

```
Data: biconnected simple Graph G' = (V, E'), initial vertex s ∈ V
Result: The lowpoints, rank and number of descendants of each vertex.
1  DFS1(vertex v, vertex parent)
2      nextNum := nextNum + 1    // the rank counter which increases with
        each recursive step
3      rank[v] := nextNum
4      lowpt1[v] := rank[v]
5      lowpt2[v] := rank[v]
6      descendants[v] := 1  // the number of descendants including v itself
7      forall e = (v, w) ∈ E' do
8          if e is marked then
9              continue
10         if rank[w] = 0 then
11             mark e as tree edge  // if w has not been visited yet, we mark
                the edge leading to w as tree edge
12             DFS1(w, v)
13             if lowpt1[w] < lowpt1[v] then
14                 lowpt2[v] := min{lowpt1[v], lowpt2[w]}
15                 lowpt1[v] = lowpt1[w]
16             else if lowpt1[w] = lowpt1[v] then
17                 lowpt2[v] := min{lowpt2[v], lowpt2[w]}
18             else
19                 lowpt2[v] := min{lowpt2[v], lowpt1[w]}
20             descendants[v] := descendants[v] + descendants[w]
21         else
22             mark e as back edge          // if w has been visited the edge
                leading to w is a back edge
23             if rank[w] < lowpt1[v] then
24                 lowpt2[v] := lowpt1[v]  lowpt1[v] := rank[w]
25             else if rank[w] > lowpt1[v] then
26                 lowpt2[v] := min{lowpt2[v], rank[w]}
```

Algorithm 5.3: DFS2 computing new numbering and paths.

```
Data: biconnected simple Graph G' = (V, E'), initial vertex s ∈ V, ordered
      adjacency lists Adj(v) for each v ∈ V
1  DFS2(vertex v, vertex parent)
2      numCount := |V|
3      newPath := true
4      forall e ∈ E' do
5          startsPath[e] = false
6          PathFinder(v)
7      forall v ∈ V do
8          old2new[rank[v]] := newnum[v]
9      forall v ∈ V do
10         lowpt1[v] := old2new[lowpt1[v]]
11         lowpt2[v] := old2new[lowpt2[v]]
12 PathFinder(vertex v, vertex parent)
13     newNum[v] := numCount − descendants[v] + 1
14     e_1, ..., e_k := A(v)
15     forall i ∈ {1, ..., k} do
16         e_i = (v, w)
17         if newPath then
18             newPath := false
19             startsPath[e] := true
20         if eisatreeedge then
21             PathFinder(w)
22             numCount := numCount − 1
23         else
24             highpt[w].pushBack(newNum[v])
25             newPath := true
```

Algorithm 5.4: The computation of the split components.

```
   Data: Result of DFS1 and DFS2
 1 pathSearch(vertex v)
 2     outv := |Adj(v)|
 3     forall e ∈ Adj(v) do
 4         if e is a tree edge then
 5             if startsPath[e] then
 6                 pop all (h, a, b) with a > lowpt1[w] from TStack
 7                 if no triples were deleted then
 8                     TStack.push(w + descendants[w] − 1, lowpt1[w], v)
 9                 else
10                     y := max{h | (h, a, b) deleted from TStack}
11                     let (h, a, b) be the last triple deleted.
12                     TStack.push(y, lowpt1[w], b)
13             pathSearch(w)
14             EStack.push({v, w})
15             typeTwoCheck
16             typeOneCheck
17             if startsPath[e] then
18                 remove all triples upto and including the first ⊥ on TStack.
19             while (h, a, b) on TStack has b ≠ v and high(v) > h do
20                 TStack.pop()
21         else
22             let e = {v, w} be a back edge
23             if startsPath[e] then
24                 deletions := 0
25                 lastDeleted := null
26                 forall (h, a, b) ∈ TStack with a > w do
27                     lastDeleted := (h, a, b)
28                     pop (h, a, b)
29                     deletions := deletions + 1
30                 if deletions = 0 then
31                     TStack.push(v, w, v)
32                 else
33                     y := max{h | (h, a, b) deleted from TStack}
34                     (h, a, b) := lastDeleted
35                     TStack.push(y, w, b)
36             EStack.push(e)
37         outv := outv − 1
38 high(vertex v)
39     if highpoint[v] is empty then
40         return 0
41     else
42         return highpoint[v].front()
```

Algorithm 5.5: The type-2 separation pair check. Note, the function checks for multiple type-2 separation pairs

```
   Data:
 1 type2Check
 2   while v ≠ 1 and (((h, a, b) on TStack has a = v) or (deg(w) = 2 and
       firstChild(w) > w)) do
 3     if a = v and parent(b) = a then
 4       TStack.pop()
 5     else
 6       e_{a,b} := null
 7       if deg(w) = 2 and firstChild(w) > w then
 8         C := newComponent()
 9         e_{v,w} := fromEStack.pop()
10         e_{w,x} := fromEStack.pop()
11         add (v, w) and (w, x) to C
12         C := C ∪ {e_{v,w}, e_{w,x}}
13         e' := newVirtualEdge(v, x, C)
14         if EStack.top() = (x, v) then
15           e_{a,b} := EStack.pop()
16           delHigh(e_{a,b})
17       else
18         (h, a, b) := TStack.pop()
19         C := newComponent()
20         while (x, y) on EStack has a ≤ x ≤ h and a ≤ y ≤ h do
21           if (x, y) = (a, b) then
22             e_{a,b} = EStack.pop()
23             delHigh(e_{a,b})
24           else
25             e_{x,y} = EStack.pop()
26             delHigh(e_{x,y})
27             C := C ∪ {e_{x,y}}
28         e' := newVirtualEdge(a, b, C)
29         if e_{a,b} ≠ null then
30           C := newComponent(e_{a,b}, e')
31           e' := newVirtualEdge(v, b, C)
32         EStack.push(e')
33         makeTreeEdge(e', v → b)
34         w := b
```

Algorithm 5.6: The type-1 separation pair check.

```
Data: The lowpoints, descendants, parent and the rank of each vertex, The edge
      stack EStack.
1  type1Check
2      if lowpt2[w] ≥ v and lowpt1 ≤ v and (parent(v) ≠ 1 or outv ≥ 2) then
3          C := newComponent()
4          while (x, y) on EStack has rank[w] ≤ rank[x] < rank[w] + descendants[w]
            or rank[w] ≤ y < w + descendants[w] do
5              e_{x,y} := EStack.pop()
6              delHigh(e_{x,y})
7              C := C ∪ {e_{x,y}}
8          e' := newVirtualEdge(v, lowpt1[w], C)
9          if EStack.top() = (v, lowpt1[w]) then
10             e'' := EStack.pop()
11             C := newComponent(e'', e')
12         if lowpt1[w] ≠ parent(v) then
13             EStack.push(e')
14         else
15             C := newComponent(e', lowpt1[w] → v)
16             e' := newVirtualEdge(lowpt1[w], v, C)
17             makeTreeEdge(e', lowpt1[w] → v)
```

Algorithm 5.7: Merging adjacent serial and parallel components.

```
Data: A set of components C_1, ..., C_k consisting of bonds, polygons and simple
      triconnected components
1  mergeAdjacentComponents
2      for i ∈ {1, ..., k} do
3          if C_i ≠ ∅ and C_i is a bond or a polygon then
4              forall e = (u, v, l) ∈ C_i do
5                  if ∃j ≠ i such that e ∈ C_j and type(C_i) = type(C_j) then
6                      C_i := (C_i ∪ C_j) \ {e} C_j := ∅
```

Algorithm 5.8: The linear-time algorithm. First the lowpoints are calculated in DFS1 and updated in DFS2. Next, the recursive path search computes the split components, excluding the last. The last split component is created from the leftover edges on the stack. Lastly, adjacent bonds and polygons are merged.

Data: A biconnected Graph $G = (V, E)$, the start vertex $s \in V$
Result: The Tutte components of G.

```
1  splitBonds(G)
2  nextNum := 0                                        // the dfs rank counter
3  DFS1(s, null)          // the dfs starts at s which has no previous node
4  DFS2(s)
5  TStack.push(⊥)
6  calculate triples for TStack
7  pathSearch(s)
8  let e_1, ..., e_k be the remaining edges on EStack
9  C_n := newComponent(e_1, ..., e_k)     // pathSearch does not create the last
     component
10 mergeAdjacentComponents(C_1, ..., C_n)
```

6. Dynamic SPQR-Tree Operations

In the following we give a detailed description of the SPQR-tree operations and the operations of its extension to general graphs. The SPQR-tree fundamentally supports two different kinds of operations, triconnectivity queries, which do not change the structure of the SPQR-tree and incremental updates, such as edge and vertex insertions, which change the structure of the SPQR-tree.

6.1 Triconnectivity Queries

One of the major functions of the SPQR-tree is to check whether two given vertices are *triconnected*. Two vertices a and b are triconnected, if they belong to the same triconnected component. In the SPQR-tree, the triconnected components are represented by P-nodes and R-nodes, as such a and b are triconnected, if there is a P-node or an R-node, which contains both a and b in its skeleton. We call these SPQR-nodes the *common allocation nodes* of a and b or simply an *allocation node* of a if the SPQR-node contains a. A simple way to find the common allocation nodes is to iterate through all the SPQR-nodes in the SPQR-tree. However, this simple solution can be quite costly, requiring $\mathcal{O}(n)$ time in the worst case, where n is the number of nodes in the SPQR-tree.

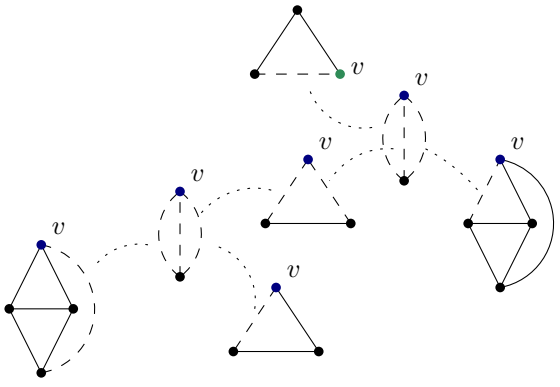


Figure 6.1: The tree induced by the allocation nodes of a vertex v, with the proper allocation node as the root (shown in green).

The allocation nodes form a subtree of the SPQR-tree, as allocation nodes are adjacent due to the virtual edges. The lowest common ancestor of the allocation nodes of a vertex v is called the *proper allocation node* of v (see Figure 6.1). To check if an SPQR-tree node is

an allocation node of a vertex v, we only need the proper allocation node of v and the *poles* of each SPQR-tree node, i.e., the endpoints of the reference edge. This is due to the fact, that a node in the SPQR-tree is an allocation node if and only if it is the proper allocation node or if it contains the vertex v as a pole.

Now that we have the proper allocation nodes and the poles, we only need to check three cases to test whether a and b are triconnected (c.f. [DBT96]). In the following we use prop(v) to denote the proper allocation node of a vertex v and parent(λ) to denote the parent node of an SPQR-tree node λ.

1. prop(a) = prop(b) is an R-node or a P-node (see Figure 6.2 case 1).
2. prop(a) is an R-node with b as a pole or vice versa ((see Figure 6.2 case 2)).
3. a and b are poles of a P- or R-node λ, which is the child of an S-node μ, i.e., parent(λ) = μ (see Figure 6.2 case 3)).

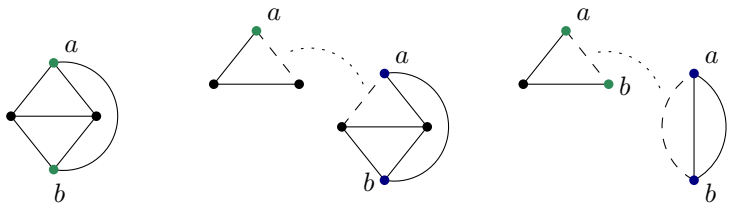


Figure 6.2: The three cases of triconnectivity from left to right: 1. shared proper allocation R-node or P-node, 2. b is a pole of the proper allocation R-node of a, 3. a and b are poles of a P-node, which is a child of their proper allocation S-node. The proper allocation nodes are shown in green, with the other allocation nodes shown in blue.

Note, the case where the proper allocation node of a vertex v is a P-node only occurs if that P-node is the root node, this is due to the fact, that P-nodes only contain two nodes, the poles, which are also allocated at the parent SPQR-node. Furthermore, in the last case the S-node is the proper allocation node of at least one of the two vertices a and b, as only one of the two may be a pole of the S-node due to the lack of parallel edges in S-nodes.

By maintaining the poles of each SPQR-tree node and the proper allocation nodes for each vertex, the triconnectivity of two vertices can be tested in $\mathcal{O}(1)$ (see [BT96]).

6.2 Incremental Updates

SPQR-trees can be updated to reflect incremental changes of the underlying graph without having to recompute the whole tree. The SPQR-tree supports edge splits and edge insertion. In fact, the entire SPQR-tree can be constructed starting from the triangle graph using only edge splits and insertions [BT96].

6.2.1 Vertex Insertion

To maintain biconnectivity in a graph, a vertex may only be inserted by splitting an edge, as inserting an isolated vertex or a vertex, which has only one edge breaks the biconnectivity of the graph.

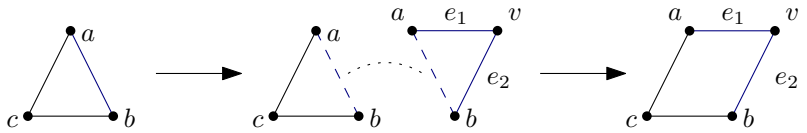


Figure 6.3: A vertex insertion via edge split inside an S-node, creating a new S-node with the new edges and vertex, which is then merged with the adjacent S-node. Dashed lines indicate virtual edges and dotted lines indicate tree edges.

Edge Splitting

To split an edge we replace the unique real edge e with a virtual edge. This virtual edge corresponds to a new S-node containing the new edges e_1, e_2 and the virtual edge e. If the new S-node is a child of another S-node we simplify the structure by absorbing the new S-node into its parent, i.e., the real edges of the new S-node become part of the parent S-node instead and the corresponding virtual edge is removed. If the edge e was the reference edge of the SPQR-tree, the new reference edge of the SPQR-tree becomes e_1 (see Figure 6.3). The entire process can be computed in $\mathcal{O}(1)$ time (see [BT96]).

6.2.2 Edge Insertion

Edge insertions are more complex than edge splits, as inserting edges increases the connectivity of the two endpoints and potentially of further vertices. This manifests itself in the SPQR-tree in the form of splits for affected S-nodes and merges for R-nodes, as affected biconnected components are split and triconnected components grow. The requirement for the splitting of S-nodes derives from the fact that affected vertices that were previously biconnected are now triconnected and hence need to be part of a triconnected component, i.e., an R-node or P-node. For the edge insertion we present two algorithm first the original edge insertion algorithm 6.2.2 described by Di Battista and Tamssia [BT96] and second a simplified edge insertion algorithm 6.2.2, which reduces the required case distinctions.

Merging Adjacent SPQR-nodes

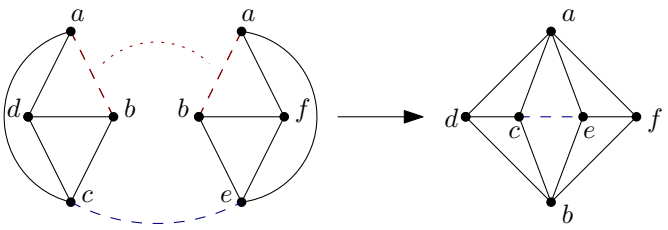


Figure 6.4: The merge of two adjacent R-nodes, the virtual edge $\{a, b\}$ (shown in red) is removed as it is no longer needed. Note, the resulting skeleton not necessarily triconnected as only two vertices may be shared between the two SPQR-nodes. For the triconnectivity an additional edge such as $\{c, e\}$ (shown in blue) or $\{d, f\}$ is necessary.

To merge two adjacent SPQR-nodes λ and μ of the same type, we create a new SPQR-node ν with the union of the vertex and edge sets as its skeleton. The virtual edge shared between λ and μ is removed from ν as it is no longer needed.

The SPQR-nodes λ and μ are removed and the neighbors of ν are given by the merged virtual edges, i.e., ν is adjacent to the union of neighbors, without the nodes λ and μ (see Figure 6.4). The merging of two SPQR-nodes is similar to the merging of vertices in a

Graph, where the resulting self loops are removed. The self loops in our case are the virtual edges which are shared. The resulting skeleton of ν is defined as follows.

$$\begin{aligned}\nu(V') &= \lambda(V') \cup \mu(V') \\ \nu(E_{\text{real}}) &= \lambda(E_{\text{real}}) \cup \mu(E_{\text{real}}) \\ \nu(E_{\text{virtual}}) &= \lambda(E_{\text{virtual}}) \cup \mu(E_{\text{virtual}}) \setminus (\lambda(E_{\text{virtual}}) \cap \mu(E_{\text{virtual}}))\end{aligned}$$

For the merge between an R-node λ and another adjacent SPQR-node μ of a different type we create a new R-node ν and assign the skeleton as above.

Note, merging two R-nodes without the insertion of an edge between distinct, i.e., not shared vertices and virtual edges will not result in a triconnected skeleton, as two distinct R-nodes are only biconnected. This may be observed in the example Figure 6.4, where the resulting skeleton is only triconnected due to insertion of the edge $\{c, e\}$.

Splitting S-nodes

To split an S-node S, along an edge $e = \{a, b\}$, we create two new S-nodes. The node S is split into two edge-disjoint paths between the vertices a and b. To complete the cycle again we add a virtual between a and b to the two S-nodes.

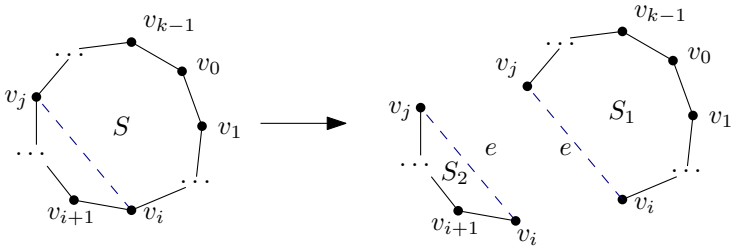


Figure 6.5: Splitting the S-node S at the edge $e = \{v_i, v_j\}$.

Let S be an S-node, with k edges, $V_{\text{skeleton}}(S) = \{v_0, ..., v_{k-1}\}$ and $E = E_{\text{real}}(S) \cup E_{\text{virtual}}(S) = \{e_i, ..., e_{k-1}\}$ ordered such that $e_i = \{v_i, v_{(i+1) \mod k}\}$, for all $i \in \{0, ..., k-1\}$, To split S at an edge $e = \{v_i, v_j\}$ with $1 \leq i \leq j \leq k$, we create two new S-nodes S_1 with the edges $E_1 = \{e_0, ..., e_{i-1}, e_j, ..., v_{k-1}\}$ and S_2 with the edges $E_2 = \{e_i, e_{i+1}, ..., e_{j-1}\}$ and add the edge $e = \{v_i, v_j\}$ to both as a virtual edge (see Figure 6.5). The resulting skeletons of S_1 and S_2 are

$$\begin{aligned}V_{\text{skeleton}}(S_1) &= \{v_0, ..., v_i, v_j, v_{j+1}, ..., v_{k-1}\} \\ E_{\text{real}}(S_1) &= E_{\text{real}}(S) \cap E_1 \\ E_{\text{virtual}}(S_1) &= E_{\text{virtual}}(S) \cap E_1 \cup \{e\} \\ V_{\text{skeleton}}(S_2) &= \{v_i, v_{i+1}, ..., v_{j-1}, v_j\} \\ E_{\text{real}}(S_2) &= E_{\text{real}}(S) \cap E_2 \\ E_{\text{virtual}}(S_2) &= E_{\text{virtual}}(S) \cap E_2 \cup \{e\}.\end{aligned}$$

Finding the shortest Path between two Allocation Nodes

Let $a, b \in V$, to obtain the shortest path between the allocation nodes of a and the allocation nodes of b we again use the proper allocation nodes, we introduced earlier. Due to the allocation nodes forming a subtree, one of the endpoints of the path will always be a proper allocation node (see Figure 6.6). Let A be the proper allocation node of a and let B be the proper allocation node of b. Either the trees induced by the allocation nodes of a

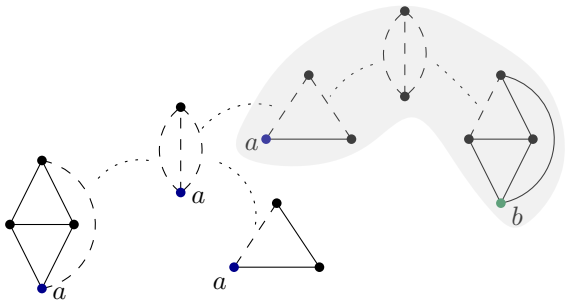


Figure 6.6: The allocation nodes of a (shown in blue) and b (shown in green), which do not have a common allocation node. The shortest path between the allocation nodes a and b (shown in grey) ends at the two proper allocation nodes and passes through the common ancestor P-node.

and b are disjoint or there is some overlap between the two. If the two subtrees overlap, then at least one of the two proper allocation nodes is also an allocation node of the other vertex, i.e., the proper allocation node of a is an allocation node of b or vice versa. If the allocation node trees of a and b are disjoint, there are two cases:

1. B is a descendant of A or vice versa. Without loss of generality, let B be a descendant of A. Then the shortest path between two allocation nodes can be obtained by following the parent path of B until we find an allocation node of a or we arrive at A.
2. There is some common ancestor C of A and B. Then, we follow the parent path of both A and B and mark the visited nodes until we arrive at an already visited node. The common ancestor C is the first and only node that is visited by both paths. The shortest path between A and B is the combination of the paths from A to C and from B to C, that is the path $A, \ldots, C, \ldots, B$.

Let p be the length of the shortest path. To obtain the shortest path in $\mathcal{O}(p)$ time, we traverse both paths simultaneously and stop when we find an allocation node of the other or an already visited node.

Original Edge Insertion Algorithm

In the following we present the edge insertion algorithm proposed by Di Battista and Tamassia [BT96]. The algorithm has been extended by an additional case for the insertion of parallel edges in an R-node, which was not considered by the authors (see Case 1b). The change ensures that the SPQR-tree is unique, up to isomorphism with regards to the root node, regardless of the construction. The distinction is not necessary for simple graphs, as the case is already covered by the Case 4.

Let $e = \{a, b\}$ be the inserted edge. For the restructuring of the SPQR-tree, Di Battista and Tamassia [BT96] use the common allocation nodes to distinguish five cases. The common allocation nodes of a and b are the nodes, which have both a and b as part of their skeletons.

1. Shared R-node:

   If a and b have exactly one common allocation node R which is an R-node, there are two possible cases:

   a) The edge e does not yet exist in R. Then, we simply add e to R (see Figure 6.7).

   b) An equivalent e' already exists in R. Then, we instead create a new P-node with a virtual edge e_R connecting it to the r-node and the two edges, e and e'.

The place of e' in the R-node is taken by a new virtual edge e_R which connects the R-node to the P-node (see Figure 6.8).

2. Shared S-node:

   If a and b have exactly one common allocation node S which is an S-node, we split the node S between the two vertices a and b creating two new S-nodes S_a and S_b. We replace the old S-node S with the two nodes S_a, S_b separated by a new P-node P containing the edge e (see Figure 6.12).

3. Shared P-node:

   If a and b have one common allocation P-node p, we simply add the edge e to the P-node P (see Figure 6.9).

4. Adjacent Case: If a and b have exactly two common allocation nodes N_1 and N_2. The two nodes are adjacent in the SPQR-tree, due to the allocation nodes forming a subtree. In this case, we replace the tree edge between the two nodes with a new P-node P, which is adjacent to the two nodes N_1 and N_2 and contains the new edge $\{a, b\}$ (see Figure 6.10). There are two cases in which we find two common allocation nodes without a shared P-node.

   a) Two adjacent R-nodes.

   b) An S-node adjacent to an R-node.

   Two S-nodes cannot be adjacent due to the merging of adjacent S-nodes created during the intermediate steps of edge insertion and edge splits. The same applies to P-nodes as well.

5. Path Case:

   If a and b do not have any common allocation nodes, let p be the shortest path between two allocation nodes of a and b. Let λ and μ be the SPQR-node endpoints of the path, which are allocation nodes of a and b respectively (see Figure 6.11 for an example of the Path Case).

   a) First, we remove all the tree edges on the path p.

   b) Then, we split every S-node S on the path p between its neighbors on the path. For the endpoints of the path λ and μ we split S between its unique neighbor on the path and the corresponding vertex a or b, respectively.

   c) Next, we create a new R-node R containing a, b, e and merge the skeletons of R-nodes on the path into it. The skeletons of this new R-node then contains the union of the skeletons of the R-nodes on the path p, the two edge endpoints a, b and the edge e.

   d) Then, we connect the P-nodes on the path p and the new nodes resulting from the splits to the newly created R-node r, adding the corresponding virtual edges to r for each node.

   e) Lastly, we absorb 2-edge S-node and P-node neighbors into R, that is we remove S-nodes and P-nodes which only have two edges, one virtual and one real edge and add the real edge to R, replacing the corresponding real edge in R.

The SPQR-tree changes the most in the path case. The additional edge increases the connectivity of a formerly biconnected path, collapsing the path in the SPQR-tree into a single R-node (see Figure 6.11).

A sequence of $k \in \mathbb{N}$ intermixed triconnectivity queries and insert edge operations on an SPQR-tree can be computed in $\mathcal{O}(n \cdot \log n + k)$ time, where n is the number of vertices (see [BT96]).

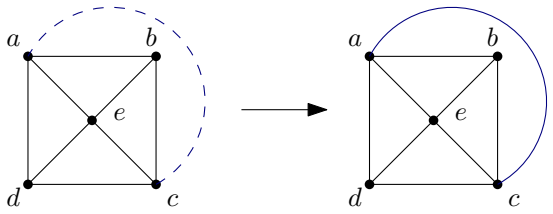


Figure 6.7: Insertion of the edge $\{a, b\}$ inside an R-node, with the new edge shown in blue.

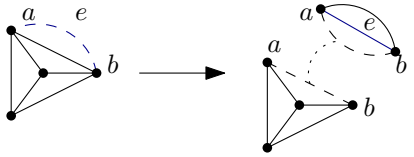


Figure 6.8: Insertion of an edge $e = \{a, b\}$ into an R-node which already has an equivalent real edge in its skeleton. A new P-node is created containing both real edges.

Split types

The splits of S-nodes can be categorized into three types: *inner splits*, *path splits* and *endpoint splits*. The inner split describes the split which occurs if both endpoints of the inserted edge $e = \{a, b\}$ are properly allocated at the S-node. In this case the S-node is split along the virtual edge e resulting in two new S-nodes with sharing the virtual edge e (see Figure 6.12). Note, that if a and b are already adjacent before the insertion of e, then the split results in an S-node consisting only of the singular edge e. This S-node is then absorbed into to the newly created P-node.

The path split occurs if the S-node s is an inner node of the tree path. In this case the S-node is split along the edges induced by the connecting reference edges of its neighbors on the path, i.e., its own poles and the poles of its path child c. The resulting S-nodes have distinct reference edges, comprised of one pole of s and one pole of c (c.f. Figure 6.13).

Lastly, the endpoint split occurs if the S-node is at one end of the path, this is the case if one of the endpoints is properly allocated at the S-node. Here the S-nodes are split among the poles and the edge endpoint, creating virtual edges from each of the two poles to the endpoint. The resulting S-nodes share the endpoint as a pole (c.f. Figure 6.14).

The different types of splits may be generalized into a single type based on the path split by using auxiliary self loops instead of the endpoint vertices. For the endpoint split we use an auxiliary self loop at the endpoint. Similarly, for the inner split we use two auxiliary self loop at each endpoint of the inserted edge.

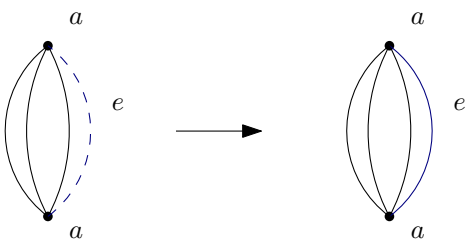


Figure 6.9: Insertion of the edge $e = \{a, b\}$ inside a P-node, with the new edge shown in blue.

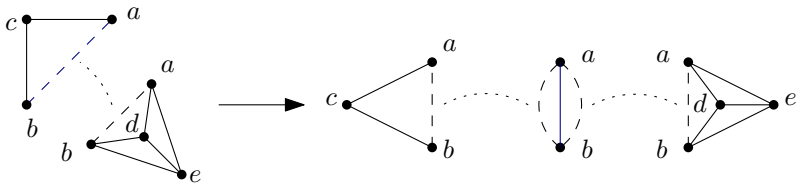


Figure 6.10: Insertion of the edge $\{a, b\}$, with two common allocation nodes. A new P-node is added adjacent to the two allocation nodes and the edge is added to the new P-node.

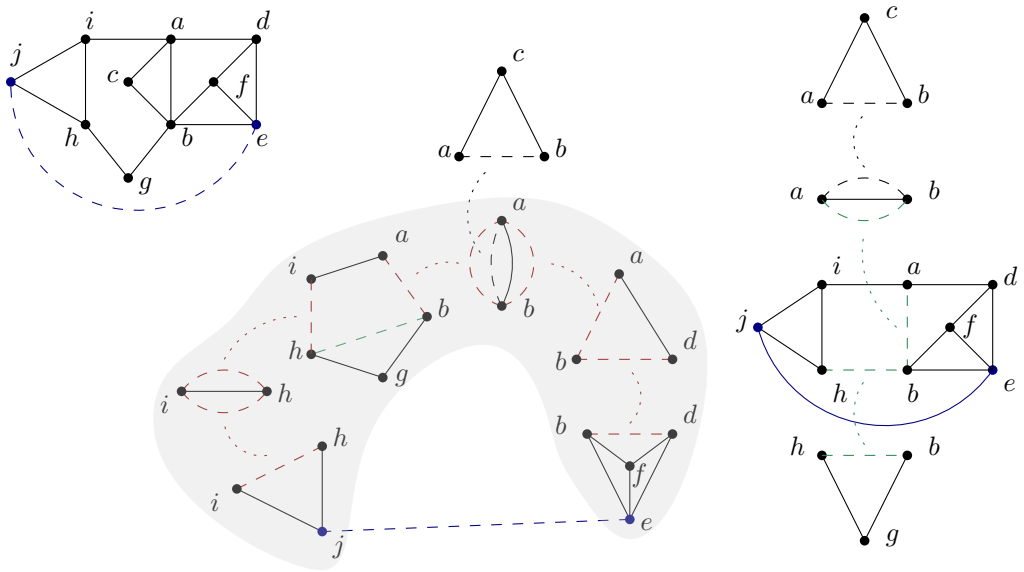


Figure 6.11: Insertion of the edge $\{e, j\}$ (shown in blue) and the resulting contraction of the path in the SPQR-tree. The SPQR-tree before the insertion is shown in the middle, with the resulting SPQR-tree to the right and the graph in the top left corner. The edges which are removed are shown in red and the added edges are shown in green.

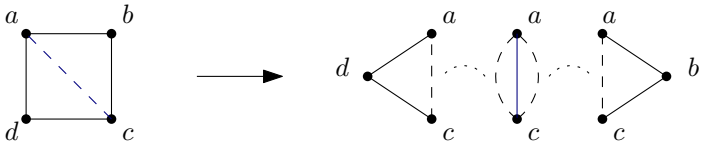


Figure 6.12: Insertion of the edge $\{a, c\}$ and the subsequent split of the S-node into two smaller S-nodes, with the virtual edges drawn as dashed lines and tree edges drawn as dotted lines.

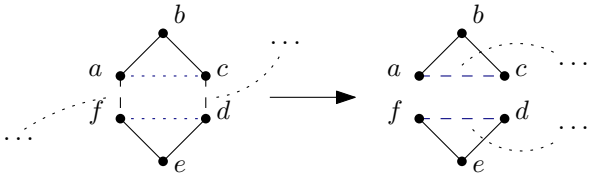


Figure 6.13: The split that occurs inside an S-node which is an inner node of the path. The S-node is split along the virtual edges of the path.

Simplified Edge Insertion

The edge insertion algorithm presented by Di Battista and Tamassia [BT96] requires many case distinctions. Despite the many cases, the algorithm largely follows the same pattern in each case, that is splitting some set of S-nodes, inserting a P-nodes for parallel edges and combining adjacent nodes of the same type.

In the following we present a simpler edge insertion algorithm, by reducing the case distinction to this core. For this, we use the shortest path between two allocation nodes and treat the common allocation nodes as paths of length 1. To avoid the ambiguity caused by multiple common allocation nodes, we use the proper allocation nodes of the inserted edge endpoints and obtain the path. Note, that the path between the two proper allocation nodes is not necessarily the shortest path between two allocation nodes. This is the case if one proper allocation node is an ancestor of the other. In this case we simply use the first allocation node which is also an ancestor instead.

We break down the insertion of an edge $e = \{a, b\}$ into the following steps (see Figure 6.15):

1. First, we obtain the shortest path between two allocation nodes of a and b. Note the path consists of a single allocation node if a and b share a common allocation node.
2. Next, we split each S-node of the path along the endpoints of its virtual edges on the path, creating new virtual edges for the split parts.
3. Similarly, for each P-node with more than three edges, we split off all edges except the virtual edges on the path.
4. Then, we merge the nodes of the path into a new SPQR-node λ and remove the virtual edges forming the path.
5. Next, we insert the edge e into λ.
    - If an equivalent edge e' already exists in λ, we first add a new P-node μ. If the existing edge e' is a real edge we move it to μ and add virtual edges connecting λ and μ (see Figure 6.8). If e' is a virtual edge, then there is another SPQR-node ν which contains the twin edge $e'' = \text{twinEdge}(e')$. In this case we add two virtual edges e_λ and e_ν to μ such that $\text{twinEdge}(e') = e_\lambda$ and $\text{twinEdge}(e'') = e_\nu$. and vice versa.

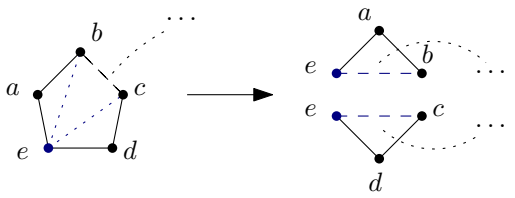


Figure 6.14: The split that occurs in an S-node that is at one end of the path.

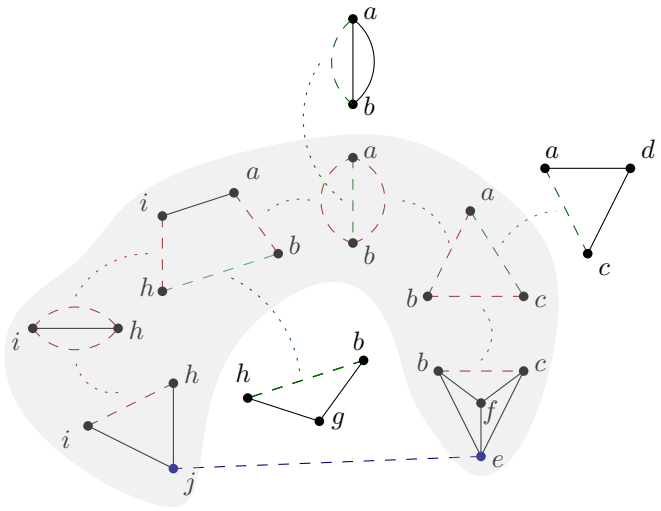


Figure 6.15: The splitting of S-nodes and P-nodes of the path. Edges resulting from the splitting are shown in green. The virtual and tree edges that are removed as part of the merging are shown in red.

6. Lastly, we merge adjacent S-nodes and P-nodes.

The newly created SPQR-node λ is an R-node, if the length of a path is at least 2 and a P-node if the length is less than 2. This is due to the fact, that a path of length 2 consists of at least four distinct nodes, the two endpoints of the inserted edge a and b and the poles of the proper allocation nodes of a and b. For paths of length 1, there are two cases, either a parallel edge is inserted or an S-node is split. In both cases we create a P-node.

For the S-nodes at the endpoints of the path we use an auxiliary self loop at the corresponding proper allocation node instead, as there is only one virtual edge on the path. After the splitting each S-node of the path is either a triangle or a square and each P-node has exactly three edges. This derives from the fact that each S-node of the path shares at most four nodes with its neighbors.

For comparison the original edge insertion algorithm by Di Battista and Tamassia [BT96] distinguishes five cases, whereas the only separate case in our algorithm is the insertion of parallel edges in an R-node, which was not considered by Di Battista and Tamassia [BT96].

6.2.3 BC-Tree Operations

In the following we describe the operations which are unique to the BC-tree and not covered by the SPQR-tree, i.e., the insertion of non biconnected vertices and the insertion of an edge between two B-nodes.

Triconnectivity

Similar to the SPQR-tree, we also maintain the proper allocation node in the BC-tree or BC-forest. This allows us to easily answer queries of the form triconnected(a, b) queries

by first checking if a and b are biconnected, by checking if a and b are part of the same B-node. If a and b are not part of the same B-node they are not biconnected and hence cannot be triconnected. If however, a and b are part of the same B-node we simply pass the query to the corresponding SPQR-tree.

Vertex Insertion

To insert a new vertex a, connected by an edge $e = \{a, b\}$, can be attached to a BC-tree by adding a new C-node for the vertex b and a new B-node containing the two endpoints a, b and the edge e. If there already is a C-node containing b, we do not need to add the C-node. To insert an isolated vertex in a BC-Forest, we simply add a new BC-tree.

Edge Insertion

For edge insertions, we obtain the proper allocation nodes of the two edge endpoints. If two endpoints have the same B-node as proper allocation node, we simply use the underlying SPQR-tree to insert the edge, as the BC-tree does not change. If however, the two endpoints do not have any common allocation nodes in the BC-tree, we obtain the shortest path p between two allocation nodes of a and b in the BC-tree and merge the B-nodes of the path p into a new B-node, similar to the path contraction in the SPQR-tree.

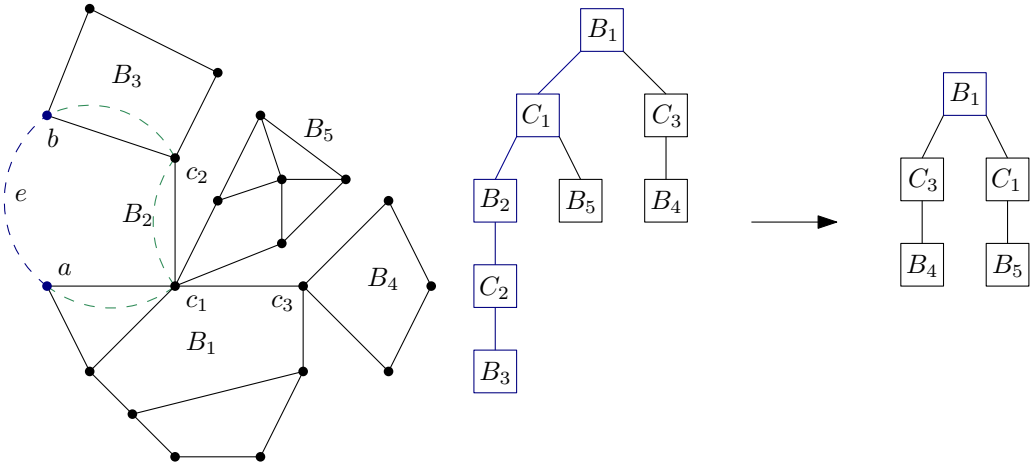


Figure 6.16: Insertion of an edge $e = \{a, b\}$ in a BC-tree. The B-nodes of the path are merged and the inner C-nodes disappear, as their vertices are no longer cutvertices. The edge and the resulting path in the BC-tree are shown in blue with the auxiliary edges drawn in green. The combination of the inserted edge and the auxiliary edges forms the cycle and thus the S-node.

The edge $e = \{a, b\}$ is then inserted as follows (see Figure 6.16)

1. First, we add an auxiliary edge e_i to the SPQR-tree of each B-node b_i of the path p. Let c_{i-1} and c_{i+1} be the vertices of the C-node neighbors b_i on the path p, then the auxiliary edge $e_i = \{c_{i-1}, c_{i+1}\}$ connects the the two vertices. For the B-nodes which only have one C-node neighbor, i.e., the endpoints of the path we use the vertices a or b instead.

2. Let b' be the B-node with the most vertices and re-root all other B-nodes of the path at the auxiliary edges we inserted earlier.

3. Then, we merge the B-nodes into b'. This is achieved by connecting the SPQR-trees with a new S-node s' containing the edge e and the auxiliary edges, keeping the path order for the S-node edges.

4. Lastly, merge S-nodes that are adjacent to s' and remove the C-nodes which no longer correspond to a cutvertex, which are the C-nodes that were only adjacent to path nodes (similar to the degree 3 P-nodes in the SPQR-tree).

The S-node forms the cycle given by the path and the endpoint a, b and connects the former B-nodes of the path to a single B-node.

To insert an edge $e = \{a, b\}$ between two BC-trees we simply add C-nodes for a and b to their respective BC-trees and a new B-node B' which connects the two C-nodes. If a or b already have corresponding C-nodes, we simply reuse them instead of creating new C-nodes.

A sequence of $k \in \mathbb{N}$ operations on a BC-forest takes $\mathcal{O}(n \log n + k)$ time, starting from a single vertex, where n is the number of vertices in the graph and k is the total number of operations (c.f. [BT96]).

6.3 Dynamic Extension of SPQR-Trees

While the SPQR-tree and BC-tree only allow edge and node insertions, it is possible to use the structure of the SPQR-tree to maintain the triconnected components under edge deletions. One such variant was proposed by Holm et al.[HIK+18] in 2018. The decremental SPQR-tree proposed by Holm et al.[HIK+18] supports edge deletions and vertex contractions, instead of edge insertions and splits. In the following year Holm and Rotenberg [HR19] published a fully dynamic SPQR-tree, supporting both edge insertions and deletions.

6.3.1 Edge Removal in SPQR-Trees

Removing an edge from the SPQR-tree is significantly more involved than inserting and edge, as removing an edge may increase the size of the SPQR-tree, by collapsing R-nodes into multiple S- and P-nodes, which negatively impacts the running time of decremental SPQR-trees. Holm et al.[HIK+18] managed to efficiently support edge deletions and contractions in $\mathcal{O}(\log^2(n))$ amortized time, by maintaining the separating 4-cycles in the dual graph. The separating 4-cycles in the dual graph have a one to one correspondence to the separation pairs. Maintaining the separating 4-cycles makes it easier to maintain the separation pairs, as deleting an edge in an embedded graph results in the contraction of an edge in the dual graph, as the edge separating the two faces is removed. This ensures the dual graph shrinks while the SPQR-tree expands, when triconnected components fall apart.

6.3.2 Dynamic SPQR-Trees

In 2019 Holm and Rotenberg [HR19] published a fully dynamic SPQR-tree, that runs in $\mathcal{O}(\log^3 n)$ time and supports both insertions and undo operations. The authors use a *heavy path decomposition* to identify paths in the SPQR-tree, containing the majority of the SPQR-nodes and pre-split the S- and P-nodes, that lie on a *heavy* path.

The heavy path decomposition partitions the edges of a tree into *heavy* edges and *light* edges. An edge $e = (v, \text{parent}(v))$ is heavy if $\text{size}(\text{parent}(v)) \leq 2 \cdot \text{size}(v)$, where $\text{size}(v)$ is the number of *descendants* of v including v itself. The set of descendants of v is the union of its children with the descendants of each of its children. The concept of the heavy path decomposition was originally proposed by Sleator and Tarjan [ST83] in 1983.

For the pre-splitting, the S-nodes lying on heavy paths are split into a primary and up to two secondary child nodes, with the primary child node containing the heavy path.

Similarly, the P-nodes are split into a primary child and at most one secondary child node. The primary child inherits the heavy path, whereas the secondary child node inherits the remaining nodes. The secondary children of S-nodes and P-nodes may again be split according to their own heavy paths. Since the original definition of the SPQR-tree does not allow S-nodes to be adjacent to other S-nodes and P-nodes to be adjacent to other P-nodes, the authors use a relaxed definition of the SPQR-tree, which allows S-nodes and P-nodes to be adjacent to their respective node types. This change does not break the SPQR-tree, as the requirement in the regular SPQR-tree simply minimizes the number of S- and P-nodes, by merging adjacent S- and P-nodes with their respective neighbors of the same type. However, since the relaxed SPQR-tree by Holm and Rotenberg does not minimize the number of nodes, it is not unique, unlike the regular SPQR-tree.

6.4 SPQR-Tree with Union-Find and Split-Find

Di Battista and Tamassia [BT96] suggested that the running time of the SPQR-tree could be improved to $\mathcal{O}(k \cdot \alpha(n))$ using Union-Find and Split-Find, where k is the number of edge insertions and query operations. Though, no in depth description on how to apply the structures to the SPQR-tree was given.

We maintain the SPQR-tree as a single graph called the *skeleton graph* with the different skeletons as disjoint components. In addition, we store the proper allocation node for each vertex of the original graph and the SPQR-node of each vertex in a skeleton and keep a map to identify the corresponding virtual or real edge of each edge in the skeleton and the children of each SPQR-node. The parent of each SPQR-node may be obtained via the corresponding edge of its reference edge. Lastly, we maintain a pointer to the root SPQR-node. This is not strictly necessary as the root may be identified by its non virtual reference edge. In the following we describe how to apply both Split-Find and Union-Find to the SPQR-tree.

Merging R-nodes using Union-Find

We use a single Union-Find for the vertices of the skeleton graph. For this we simply create a new set in the Union-Find structure whenever a new vertex is added to the skeleton graph. This allows us to merge two vertices by performing a union between the two.

When two R-nodes are merged we simply perform a union operation for the shared vertices. Since only the path vertices may be shared, at most two vertices may be shared between two R-nodes. The children of the merged R-node are appended to the parent R-node. This allows us to merge two R-nodes in $\mathcal{O}(\alpha(n))$ amortized time.

Splitting S-nodes using Split-Find

Using Split-Find brings with it some limitations as Split-Find generally does not allow adding new elements. Even incremental variants such as the one of Imai and Asanao [IA84] only allows insertions at the end of the ordered sets. To the extent of our knowledge no other incremental variants with a similar time bound exist. To alleviate this problem we insert entire S-nodes rather than splitting edges. This still allows the SPQR-tree to be constructed incrementally by inserting entire ears of an ear-decomposition rather than single vertices. To insert an entire S-node in the SPQR-tree, we simply insert a virtual edge and attach the new S-node to it. The change works well with the construction via ear-decomposition, as S-nodes can be inserted as an entire ear without needing to split edges.

Similar to the Union-Find, we use a single circular Split-Find to maintain the vertices of S-nodes. This differs from the original proposal of La Poutré [Pou92], who used separate

Split-Find structures for each of the cycle trees. To avoid having to use separate Split-Find structures for each S-node, we use an incremental Split-Find structure, which allows us to add entire sets as well as single elements. This allows us to use the same Split-Find structure for multiple S-nodes and allows us to add the split vertices to the resulting S-nodes. Such a structure can be created using a structure similar to the one proposed by Hopcroft and Ullman [HU73] or the one proposed by Gabow and Tarjan [GT85] and adding new layers as needed, effectively appending the new set or element to the structure. Using Split-Find allows us to split an S-node in $\mathcal{O}(\alpha(n))$ amortized time.

Running Time Improvements

Using both structures we may obtain an amortized running time of $\mathcal{O}(\alpha(n))$, where n is the number of vertices. The insertion of k vertices through the insertion of an ear requires $\mathcal{O}(k)$, resulting in an amortized time of $\mathcal{O}(1)$, per vertex, which is equivalent to the vertex insertion via an edge split. For the insertion of an edge $e = \{a, b\}$ we need to find the shortest path between allocation nodes of a and b in the SPQR-tree. This path may be obtained in $\mathcal{O}(p)$, where p is the length of the path. Each node of the path is either an S-node, P-node or R-node. For the S-nodes we may perform the splits using the split find structure in $\mathcal{O}(\alpha(n))$ amortized time. Adding the corresponding virtual edge to the resulting R-node or P-node only requires constant time. Similarly, merging the R-nodes may be done in $\mathcal{O}(\alpha(n))$ amortized time. For the P-nodes we modify at most three edges, as such we may also handle the P-nodes in constant time. Combining the path computation, splitting and merging we obtain an amortized running time of $\mathcal{O}(p \cdot \alpha(n))$, where p is the length of the path. Intuitively, the actual cost is lower as once an edge has been inserted the path is contracted and therefore the cost of obtaining, splitting and merging the path nodes is only payed once.

Theorem 6.1. *Let T be an SPQR-tree of a biconnected graph $G = (V, E)$ using Split-Find and Union-Find. The amortized running time of* ***insertEdge*** *is in $\mathcal{O}(\alpha(n))$.*

Proof. Let $e = \{a, b\} \in E$ be the edge to be inserted and let Π be the set of SPQR-nodes forming the shortest path between allocation nodes of a and b. This path can be computed in $\mathcal{O}(|\Pi|)$ as we have outlined earlier. Fundamentally, there are two distinct cases which have to be handled differently. The first is the trivial case, where $|\Pi| = 1$. In this case the SPQR-tree may be updated in $\mathcal{O}(\alpha(n))$ if the unique node $\lambda \in \Pi$ is an S-node and $\mathcal{O}(1) \leq (\alpha(n))$ for P-nodes and R-nodes.

The second case $|\Pi| \geq 2$ is the more complex than the first. Let R be the set of R-nodes, P the set of P-nodes and S the set of S-nodes in T. Let $\Phi(T) = 2 \cdot |R| + 2 \cdot \sum_{\lambda \in P} \deg(\lambda)$ be the potential of the SPQR-tree T.

Since, S-nodes must not be adjacent the number of S-nodes in the path is $|S_\Pi| \leq \lceil \frac{|\Pi|}{2} \rceil$ since $|S_\Pi| \in \mathbb{N}_0$, $\lceil \frac{|\Pi|}{2} \rceil \leq \frac{|\Pi|}{2} + \frac{1}{2}$ The remaining SPQR-nodes of the path are either R-nodes or P-nodes, hence there are $|R_\Pi| + |P_\Pi| \geq \lfloor \frac{|\Pi|}{2} \rfloor \geq \frac{|\Pi|}{2} - \frac{1}{2} = \frac{|\Pi|-1}{2}$ R-nodes and P-nodes of the path. Let R_π and P_Π be the sets of R-nodes and P-nodes of the path respectively, then $|R_\Pi| + |P_\Pi| \geq \frac{|\Pi|}{2} - 1$. The merging changes the number of R-nodes from $|R|$ to $|R| - |R_\Pi| + 1$, as the R-nodes of the path are merged into a single R-node μ or the new R-node μ is created if $R_\Pi = \emptyset$. Similarly, the degree of each P-node $\lambda \in P_\Pi$ reduces by 1 as

the two path edges of λ are removed and a single edge is added to connect λ to μ. Hence, the potential of the updated SPQR-tree T' is as follows.

$$\begin{aligned}
\Phi(T') &= 2 \cdot (|R| - |R_\Pi| + 1) + 2 \cdot \sum_{\lambda \in P} \deg(\lambda) - 2 \cdot |P_\Pi| \\
&= 2 \cdot (|R| + 1) + 2 \cdot \sum_{\lambda \in P} \deg(\lambda) - 2 \cdot (|P_\Pi| + |R_\Pi|) \\
&\leq 2 \cdot (|R| + 1) + 2 \cdot \sum_{\lambda \in P} \deg(\lambda) - 2 \cdot \frac{|\Pi| - 1}{2} \\
&= 2 \cdot (|R| + 1) + 2 \cdot \sum_{\lambda \in P} \deg(\lambda) - |\Pi| + 1 \\
&= 2 \cdot |R| + 2 \cdot \sum_{\lambda \in P} \deg(\lambda) - |\Pi| + 3 \\
&= \Phi(T) - |\Pi| + 3
\end{aligned}$$

From the change in the potential we obtain an amortized running time in $\mathcal{O}(\alpha(n) + 3) = \mathcal{O}(\alpha(n))$ for the `insertEdge` operation. □

6.5 SPQR-Tree Implementations

The Open Graph Drawing Framework [CGJ+14], or OGDF for short, is an open source C++ library providing data structures and algorithms for creating, maintaining and processing graphs. The framework provides algorithms for automated graph drawing, but also includes an implementation of SPQR-trees and BC-trees with the linear-time construction of Gutwenger and Mutzel. The project originated from the AGD library (Algorithms for Graph Drawing) [MGB+98], which was developed in 1996. After a complete redesign and rewrite of the AGD library in 1999 the project was renamed OGDF and has since been continuously updated and expanded (see [CGJ+14]).

As part of our study of the SPQR-tree, we created our own implementation of the SPQR-tree in C++. Our implementation consists of two variants a regular SPQR-tree and an improved version using Split-Find and Union-Find. Both variants make use of the graph structures provided by the OGDF library. The improved version also uses the Union-Find structure provided by OGDF, for the Split-Find we created our own incremental variant based on the structure described by Hopcroft and Ullman [HU73] which was extended to allow incrementally adding new sets and elements as described by Imai and Asano [IA84] (see Section 3.2). Our implementations use the ear-decomposition construction algorithm to construct the SPQR-tree, unlike the OGDF implementation, which uses the linear time algorithm of Gutwenger [Gut10] (see Section 5.2).

Running Time Comparisons

We tested our two implementations against each other and against the OGDF implementation on different sized graphs ranging from 100 vertices to 5000 vertices in increments of 100. For the number of edges we chose twice the number of vertices, e.g. 200 edges for the graph with 100 vertices and 400 edges for the graph with 200 vertices and so on. We believe that this ratio of edges to vertices results in larger SPQR-trees, as more dense graphs will likely result in few large R-nodes and sparse graph will result few in large S-nodes. The graph were generated randomly using OGDFs `randomBiconnectedGraph` function. The test system was an Intel Xeon E5 CPU, clocked at 3.0 GHz with 10 cores and 64 GB RAM. For greater accuracy of the results, each measurement were repeated ten times for each datapoint. The measurements were taken in microseconds using the C++ Chrono library.

The test results show a noticeable improvement from the simple SPQR-tree to the improved SPQR-tree. Though, as expected the ear-decomposition is drastically slower than the linear time construction used by OGDF (see Figure 6.17). A closer comparison of the linear-time and the ear-decomposition reveals that the running time of linear-time algorithm is only slightly slower than the ear-decomposition algorithm, roughly by a factor of 2 (see Figure 6.18). As such it is highly unlikely that an incremental construction using ear-decomposition could be faster than the linear-time construction.

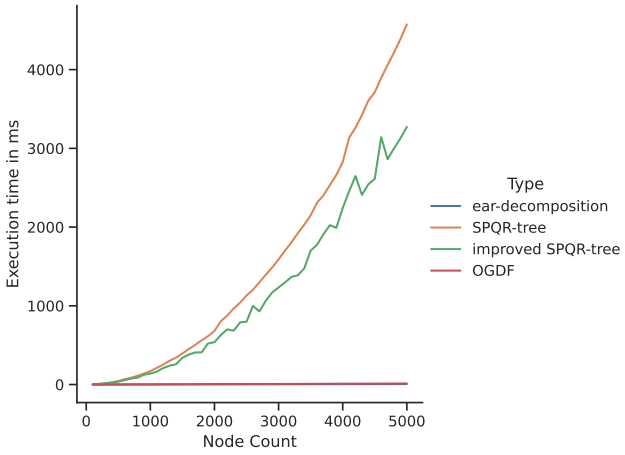


Figure 6.17: Comparison of the ear-decomposition construction using both out implementations and the linear time construction used by the OGDF implementation.

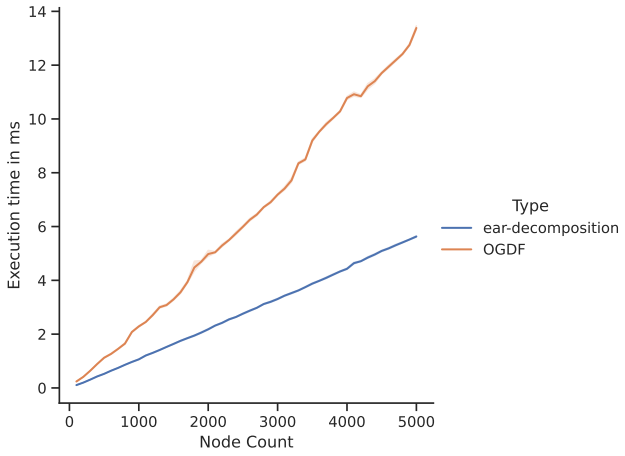


Figure 6.18: A closer comparison of the linear time construction and a simple ear-decomposition algorithm

Problems with OGDF

The implementation of SPQR-trees in OGDF is generally very efficient, however, we have discovered that the OGDF implementation struggles with the insertion of an edges into large P-nodes, yet has no problem when the edge is inserted on a path containing the very same large P-node, if the P-node is an inner node of the path. In the following we present a minimal example which demonstrates the problem.

Let $n \in \mathbb{N}$ and consider the complete bipartite graph $K_{2,n}$ with the two vertex sets $V_1 = \{s, t\}$ and $V_2 = \{v_1, ..., v_n\}$ (see Figure 6.19). The SPQR-tree of a bipartite graph consists of a single P-node P with $V_{\text{skeleton}}(P) = \{s, t\}$ connected to n S-nodes $S_1, \ldots,$

S_n, with $V_{\text{skeleton}}(S_i) = \{s, v_i, t\}$ for each $i \in \{1, ..., n\}$. Inserting an edge $e = \{s, t\}$ simply consists of adding e to $E_{\text{real}}(P)$, i.e., $E_{\text{real}}(P) \cup \{e\}$.

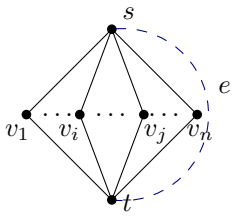


Figure 6.19: Insertion of the edge $e = \{s, t\}$ in the complete bipartite graph $K_{2,n}$.

We tested the OGDF implementation on different sizes of n ranging from 100 to 5000 in increments of 100 with ten measurements for each n and each variant. The different insertions are s, t, a, b and s, a, where a and b refer to randomly selected but distinct $v_i, v_j \in \{v_1, ..., v_n\}$.

Adding the edge e to the SPQR-tree in OGDF requires time linear to the number of vertices. In contrast inserting an edge between two vertices $v_i, v_j \in V_2$ is much faster even though the operation is more complicated in theory (see Figures 6.20 and 6.21). A similar trend may be observed when adding edges with only one endpoint in the P-node, though the required running time is not as drastic and more varied, requiring roughly half the time as the insertion in the P-node. This suggests a link between the running time and the degree of the endpoints of the inserted edge. The tests were conducted using the current version of OGDF, Dogwood (released February 2nd, 2022), using a `DynamicSPQRTree` constructed from the complete bipartite graph $K_{2,n}$.

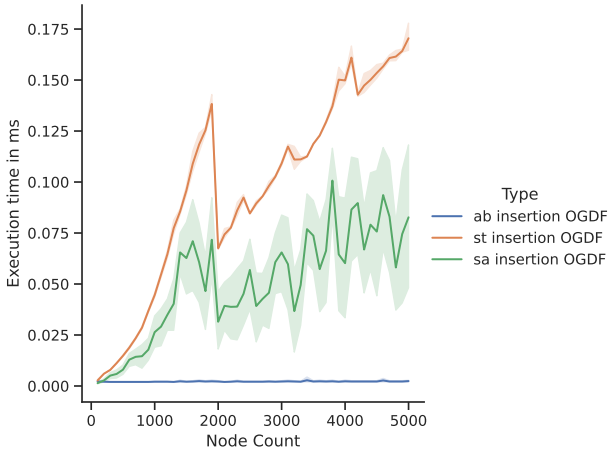


Figure 6.20: Test results from the different edge insertion for the complete bipartite graphs.

We also tested our own implementation on the very same graphs. The results are more in line with the with the expected results, with the s, t insertion being the fastest and the a, b insertion being the slowest (see Figure 6.22). However, our implementation is by no means faster than the OGDF version, which is to be expected. The improved SPQR-tree shows a similar trend with slightly better running times (see Figure 6.23).

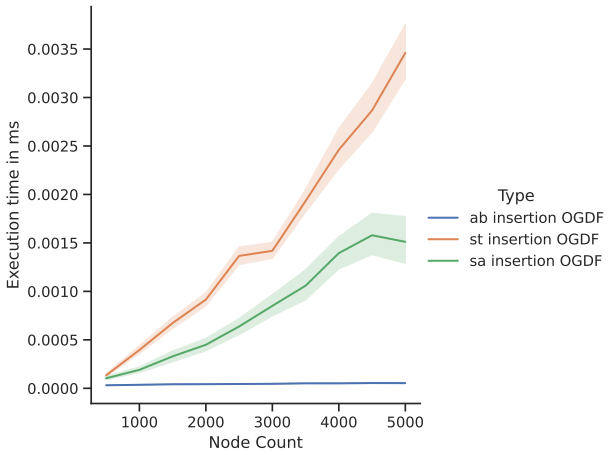


Figure 6.21: Test results obtained from a slower test system and less accuracy which better illustrates the trend (i3-7100U CPU@2.4 GHz with 2 cores and 8 GB RAM)

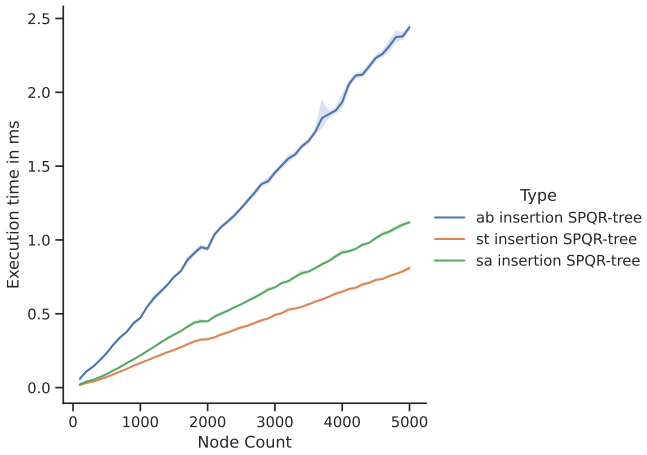


Figure 6.22: Test results of the different edge insertion on the bipartite graph using our implementation of the SPQR-tree.

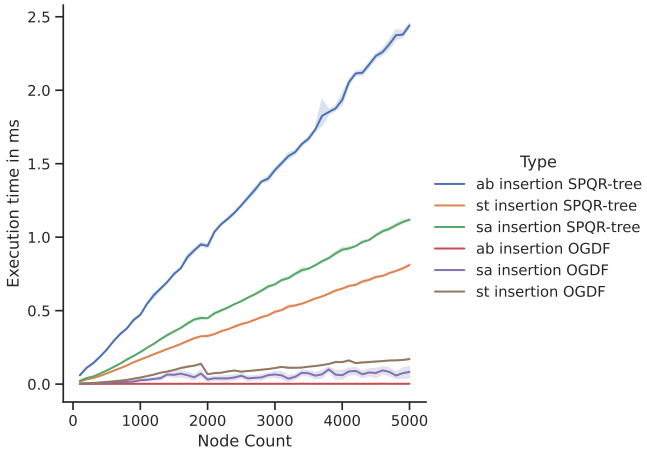


Figure 6.23: Comparison of the test results using the different SPQR-tree implementations.

7. Conclusion

The SPQR-tree remains an important data structure in the field of graph drawing and planarity testing. We have presented two different approaches for the construction of the SPQR-tree. A complex linear time construction which detects the separation pairs in the input graph and a simple incremental construction using an ear-decomposition. We compared the two approaches in a randomized experiment. Naturally, the linear time algorithm remains faster, which is to be expected as the ear-decomposition construction does not offer a theoretical improvement over the linear time algorithm. However, the simplicity of the ear-decomposition construction provides an advantage over the linear time construction particularly in the implementation of the SPQR-tree.

Furthermore, we have presented an alternative edge insertion algorithm for the SPQR-tree, which reduces the number of case distinctions of the original algorithm This makes the algorithm more concise and therefore easier to comprehend. We also extended the original algorithm by an additional case which was overlooked by the authors

We also created our own implementation of the SPQR-tree applying both Union-Find and Split-Find, which had been suggested before by the original authors, but to the extent of our knowledge had yet to be tested. The application of both structures shows a noticeable improvement in the running time of the SPQR-tree. There is some overhead to the two structures which may be observed in smaller graphs.

Additionally, we discovered a potential issue with the SPQR-tree implementation in t he open graph drawing framework OGDF. The OGDF implementation struggles with the insertion of some edges in the complete bipartite graph $K_{2,n}$, yet handles the theoretically more involved case easily. This suggests that the problem may be alleviated easily without any major changes.

Further research may examine the impact of different ear-decompositions on the running time of the ear-decomposition construction. It is likely that some ear-decomposition are better suited for the ear-decomposition construction than others. We conjecture that it is best to insert parallel edges last, as these can be inserted easily if the corresponding P-node has already been created. Such an ear-decomposition may easily be obtained from a regular ear-decomposition by moving the simple ears to the end of the path list.

Bibliography

[AP61] Louis Auslander and SV Parter. On imbedding graphs in the sphere. *Journal of Mathematics and Mechanics*, pages 517–523, 1961.

[BHR19] Guido Brückner, Markus Himmel, and Ignaz Rutter. *An SPQR-Tree-Like Embedding Representation for Upward Planarity*, chapter Graph Drawing and Network Visualization, pages 517–531. Springer, 2019.

[Bra02] Ulrik Brandes. Eager st-ordering. In *Algorithms — ESA 2002*, pages 247–256. Springer Berlin Heidelberg, 2002.

[BT89] G. Di Battista and R. Tamassia. Incremental planarity testing. In *30th Annual Symposium on Foundations of Computer Science*. IEEE, 1989.

[BT90] Giuseppe Di Battista and Roberto Tamassia. On-line graph algorithms with SPQR-trees. In *Automata, Languages and Programming*, pages 598–611. Springer Berlin Heidelberg, 1990.

[BT96] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, apr 1996.

[CGJ+14] M. Chimani, C. Gutwenger, M. Jünger, G.W. Klau, K. Klein, and P. Mutzel. *Handbook of Graph Drawing and Visualization*, chapter 17 The Open Graph Drawing Framework (OGDF). CRC Press, 2014.

[DBT96] Giuseppe Di Battista and Roberto Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.

[DL98] Walter Didimo and Giuseppe Liotta. Computing orthogonal drawings in a variable embedding setting. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *Algorithms and Computation*, pages 80–89, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Epp92] David Eppstein. Parallel recognition of series-parallel graphs. *Elsevier*, 98(1):41–55, 1992.

[FRT93] Donald Fussell, Vijaya Ramachandran, and Ramakrishna Thurimella. Finding triconnected components by local replacement. *SIAM Journal on Computing*, 22(3):587–616, 1993.

[gdt] GDToolkit: An object oriented C++ library for handling and drawing Graphs. `http://www.dia.uniroma3.it/~gdt`.

[GM01] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In *Graph Drawing*, pages 77–90. Springer Berlin Heidelberg, 2001.

[Gol63] AJ Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conference, Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dept. of Mathematics, Princeton University, May 16-18*, 1963.

[GT85] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[Gut10] Carsten Gutwenger. *Application of SPQR-trees in the planarization approach for drawing graphs.* PhD thesis, Dortmund University of Technology, 2010.

[Het64] G Hetyei. On covering by 2x1 rectangles, pécsi tanárképző főisk, 1964.

[HIK+18] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Lacki, and Eva Rotenberg. Decremental SPQR-trees for Planar Graphs. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46:1–46:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[HR19] Jacob Holm and Eva Rotenberg. Worst-case polylog incremental spqr-trees: Embeddings, planarity, and triconnectivity. 2019.

[HR20] 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, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, volume abs/1911.03449, pages 167–180. ACM, 2020.

[HT73] J.E. Hopcroft and R.E Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[HU73] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[IA84] H. Imai and T. Asano. Dynamic segment intersection search with applications. In *25th Annual Symposium onFoundations of Computer Science, 1984.*, pages 393–402, 1984.

[Lan37] Saunders Mac Lane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460 – 472, 1937.

[LH10] Wai-Shing Luk and Huiping Huang. Fast and lossless graph division method for layout decomposition using spqr-tree. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 112–115, 2010.

[Lov83] L. Lovász. Ear-decompositions of matching-covered graphs. *Combinatorica*, 3(1):105–117, Mar 1983.

[Lov85] L. Lovász. Computing ears and branchings in parallel. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 464–467, 1985.

[LP75] L. Lovász and M. D. Plummer. *Infinite and Finite sets*, chapter On bicritical graphs, pages 1051–1979. North-Holland, 1975.

[LP90] J. A. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 34–44, New York, NY, USA, 1990. Association for Computing Machinery.

[LP91] Han La Poutré. *Dynamic Graph Algorithms and Data Structures.* PhD thesis, 1991.

[LP93] Johannes Antonius La Poutré. *Dynamic microsets for RAMs.* Department of Computer Science, Utrecht University, 1993.

[Men27] Karl Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–1159, 1927.

[MGB+98] Petra Mutzel, Carsten Gutwenger, Ralf Brockenauer, Sergej Fialko, Gunnar Klau, Michael Krüger, Thomas Ziegler, Stefan Näher, David Alberts, Dirk Ambras, Gunter Koch, Michael Jünger, Christoph Buchheim, and Sebastian Leipert. A library of algorithms for graph drawing. In Sue H. Whitesides, editor, *Graph Drawing*, pages 456–457, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[MR87] G. Miller and V. Ramachandran. A new graphy triconnectivity algorithm and its parallelization. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 335–344, New York, NY, USA, 1987. Association for Computing Machinery.

[MSV86] Yael Maon, Baruch Schieber, and Uzi Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[Pou92] J. A. La Poutré. Maintenance of triconnected components of graphs. In *Automata, Languages and Programming*, pages 354–365. Springer Berlin Heidelberg, 1992.

[Pou94] Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing - STOC '94*. ACM Press, 1994.

[Ram92] Vijaya Ramachandran. *Synthesis of Parallel Algorithms*, chapter Parallel Open Ear Decomposition with Applications to Graph Biconnectivity and Triconnectivity. Morgan-Kaufmann, 1992.

[Sch13] Jens M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241–244, apr 2013.

[SS19] Lena Schlipf and Jens M. Schmidt. Simple computation of st-edge- and st-numberings from ear decompositions. *Information Processing Letters*, 145:58–63, may 2019.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, jun 1983.

[Tut66] W.T. Tutte. *Connectivity in Graphs*. University of Toronto Press, 1966.