A Linear Time Implementation of SPQR-Trees*

Carsten Gutwenger[1] and Petra Mutzel[2]

[1] Max-Planck-Institut für Informatik
Saarbrücken, Germany, gutwenge@mpi-sb.mpg.de
[2] Technische Universität Wien, Austria, mutzel@ads.tuwien.ac.at

Abstract. The data structure SPQR-tree represents the decomposition of a biconnected graph with respect to its triconnected components. SPQR-trees have been introduced by Di Battista and Tamassia [8] and, since then, became quite important in the field of graph algorithms. Theoretical papers using SPQR-trees claim that they can be implemented in linear time using a modification of the algorithm by Hopcroft and Tarjan [15] for decomposing a graph into its triconnected components. So far no correct linear time implementation of either triconnectivity decomposition or SPQR-trees is known to us. Here, we show the incorrectness of the Hopcroft and Tarjan algorithm [15], and correct the faulty parts. We describe the relationship between SPQR-trees and triconnected components and apply the resulting algorithm to the computation of SPQR-trees. Our implementation is publically available in AGD [1].

1 Introduction

The data structure SPQR-tree represents the decomposition of a biconnected graph with respect to its triconnected components. SPQR-trees have been introduced by Di Battista and Tamassia [8] in a static and in a dynamic environment. In [8,10], the authors use SPQR-trees in order to represent the set of all planar embeddings of a planar biconnected graph.

Since then, SPQR-trees evolved to an important data structure in the field of graph algorithms. Many linear time algorithms that work for triconnected graphs only can be extended to work for biconnected graphs using SPQR-trees (e.g., [4,17]). Often it is essential to represent the set of all planar embeddings of a planar graph, e.g. in order to optimize a specific criteria over all planar embeddings [14,21,3,5], or for testing cluster planarity [18,6]. In a dynamic environment, SPQR-trees are useful for a variety of on-line graph algorithms dealing with triconnectivity, transitive closure, minimum spanning tree, and planarity testing [2]. Here, we restrict our attention to the static environment.

In the theoretical papers (e.g., [8,9,10]), the authors suggest to construct the data structure SPQR-tree in linear time "using a variation of the algorithm of [15] for finding the triconnected components of a graph...[10]". So far, to our knowledge, no correct linear time implementation is publically available. The only correct implementation of SPQR-trees we are aware of is part of GDToolkit [12],

* Partially supported by DFG-Grant Mu 1129/3-1, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen".

where SPQR-trees are used in connection with a branch-and-bound algorithm to compute an orthogonal drawing of a biconnected planar graph with the minimum number of bends. However, this implementation does not run in linear time [11].

Here, we present a linear time implementation of the data structure SPQR-tree. We show the relationship between SPQR-trees and triconnected components, and show the incorrectness of the algorithm presented in [15] for decomposing a graph into its triconnected components. We develop a correct algorithm for triconnectivity decomposition by correcting and replacing the faulty parts in [15], and apply it to the computation of SPQR-trees. Our implementation (in a re-usable form) is publically available in AGD [1] (see Section 6).

The paper is structured as follows. The basics of SPQR-trees and triconnected components are described in Section 3. The algorithm for computing SPQR-trees and triconnectivity decomposition is described in Section 4, and the faulty parts of the Hopcroft and Tarjan algorithm are shown in Section 5, where we also point out the corrections we have made. We have carefully tested our implementation. Computational results concerning running time are described in Section 6.

2 Preliminaries

Let $G = (V, E)$ be an *undirected multi-graph*, that is, V is a set of vertices and E is a multi-set of unordered pairs (u, v) with $u, v \in V$. An edge (v, v) is called a *self-loop*. If an edge $(u, v) \in E$ occurs more than once in E, it is called a *multiple edge*. G is called *simple*, if it contains neither self-loops nor multiple edges. If E' is a set of edges, $V(E')$ denotes the set of all vertices incident to at least one edge in E'. A *path* $p : v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w. A path is *simple* if all its vertices are distinct. If $p : v \overset{*}{\Rightarrow} w$ is a simple path, then p plus the edge (w, v) is a *cycle*.

An undirected multi-graph $G = (V, E)$ is *connected* if every pair $v, w \in V$ of vertices in G is connected by a path. A connected multi-graph G is *biconnected* if for each triple of distinct vertices v, w, a, there is a path $p : v \overset{*}{\Rightarrow} w$ such that a is not on p. Let $G = (V, E)$ be a biconnected multi-graph and $a, b \in V$. E can be divided into equivalence classes $E_1, \ldots, E_k$ such that two edges which lie on a common path not containing any vertex of $\{a, b\}$ except as an endpoint are in the same class. The classes E_i are called the *separation classes* of G with respect to $\{a, b\}$. If there are at least two separation classes, then $\{a, b\}$ is a *separation pair* of G unless (i) there are exactly two separation classes, and one class consists of a single edge, or (ii) there are exactly three classes, each consisting of a single edge. If G contains no separation pair, G is called *triconnected*.

A *tree* T is a directed graph whose underlying undirected graph is connected, such that there is exactly one vertex (called the *root*) having no incoming edges and every other vertex has exactly one incoming edge. An edge in T from v to w is denoted with $v \rightarrow w$. If there is a (directed) path from v to w, we write $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the *parent* of w, and w a *child* of v. If $v \overset{*}{\rightarrow} w$, v is an *ancestor* of w, and w a *descendant* of v. Every vertex is an ancestor and a descendant of itself. If G is a directed multi-graph, a tree T is a *spanning tree* of G if T is a subgraph of G and T contains all vertices in G.

A *palm tree* P is a directed multi-graph such that each edge in P is a either a tree arc (denoted with $v \rightarrow w$) or a frond (denoted with $v \hookrightarrow w$) satisfying the following properties:

(i) The subgraph T consisting of all tree arcs is a spanning tree of P.
(ii) If $v \hookrightarrow w$, then $w \overset{*}{\rightarrow} v$.

3 SPQR-Trees and Triconnected Components

Let $G = (V, E)$ be a biconnected multi-graph, $\{a, b\}$ a separation pair of G, and $E_1, \ldots, E_k$ the separation classes of G with respect to $\{a, b\}$. Let $E' = \bigcup_{i=1}^{\ell} E_i$ and $E'' = \bigcup_{i=\ell+1}^{k} E_i$ be such that $|E'| \geq 2$ and $|E''| \geq 2$. The two graphs $G' = (V(E'), E' \cup \{e\})$ and $G'' = (V(E''), E'' \cup \{e\})$ are called *split graphs* of G with respect to $\{a, b\}$, where $e = (a, b)$ is a new edge. Replacing a multi-graph G by two split graphs is called splitting G. Each split graph is again biconnected. The edge e is called *virtual edge* and identifies the split operation.

Suppose G is split, the split graphs are split, and so on, until no more split operations are possible. The resulting graphs are called the *split components* of G. Each of them is a set of three multiple edges (*triple bond*), or a cycle of length three (*triangle*), or a triconnected simple graph. The split components are not necessarily unique.

Lemma 1. *Let $G = (V, E)$ be a multi-graph.*

(i) *Each edge in E is contained in exactly one, and each virtual edge in exactly two split components.*
(ii) [15] *The total number of edges in all split components is at most* $3|E| - 6$.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two split components containing the same virtual edge e. The graph $G' = (V_1 \cup V_2, (E_1 \cup E_2) \setminus \{e\})$ is called a *merge graph* of G_1 and G_2. Replacing two components G_1 and G_2 by a merge graph of G_1 and G_2 is called *merging* G_1 and G_2. The *triconnected components* of G are obtained from its split components by merging the triple bonds into maximal sets of multiple edges (*bonds*) and the triangles into maximal simple cycles (*polygons*).

Lemma 2. [19,15] *The triconnected components of G are unique.*

Triconnected components of graphs are closely related to SPQR-trees. SPQR-trees were originally defined in [8] for planar graphs only. Here, we cite the more general definition given in [9], that also applies to not necessarily planar graphs.

Let G be a biconnected graph. A *split pair* of G is either a separation pair or a pair of adjacent vertices. A *split component* of a split pair $\{u, v\}$ is either an edge (u, v) or a maximal subgraph C of G such that $\{u, v\}$ is not a split pair of C. Let $\{s, t\}$ be a split pair of G. A *maximal split pair* $\{u, v\}$ of G with respect to $\{s, t\}$ is such that, for any other split pair $\{u', v'\}$, vertices u, v, s, and t are in the same split component.

Let $e = (s, t)$ be an edge of G, called the *reference edge*. The SPQR-tree $\mathcal{T}$ of G with respect to e is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of $\mathcal{T}$ has an associated biconnected multi-graph, called the *skeleton* of μ. Tree $\mathcal{T}$ is recursively defined as follows:

Trivial Case: If G consists of exactly two parallel edges between s and t, then $\mathcal{T}$ consists of a single Q-node whose skeleton is G itself.

Parallel Case: If the split pair $\{s, t\}$ has at least three split components $G_1, \ldots, G_k$, the root of $\mathcal{T}$ is a P-node μ, whose skeleton consists of k parallel edges $e = e_1, \ldots, e_k$ between s and t.

Series Case: Otherwise, the split pair $\{s, t\}$ has exactly two split components, one of them is e, and the other one is denoted with G'. If G' has cutvertices $c_1, \ldots, c_{k-1}$ $(k \geq 2)$ that partition G into its blocks $G_1, \ldots, G_k$, in this order from s to t, the root of $\mathcal{T}$ is an S-node μ, whose skeleton is the cycle $e_0, e_1, \ldots, e_k$, where $e_0 = e$, $c_0 = s$, $c_k = t$, and $e_i = (c_{i-1}, c_i)$ $(i = 1, \ldots, k)$.

Rigid Case: If none of the above cases applies, let $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ be the maximal split pairs of G with respect to $\{s, t\}$ $(k \geq 1)$, and, for $i = 1, \ldots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$ but the one containing e. The root of $\mathcal{T}$ is an R-node, whose skeleton is obtained from G by replacing each subgraph G_i with the edge $e_i = (s_i, t_i)$.

Except for the trivial case, μ has children $\mu_1, \ldots, \mu_k$, such that μ_i is the root of the SPQR-tree of $G_i \cup e_i$ with respect to e_i $(i = 1, \ldots, k)$. The endpoints of edge e_i are called the *poles* of node μ_i. The virtual edge of node μ_i is edge e_i of skeleton of μ. Tree $\mathcal{T}$ is completed by adding a Q-node, representing the reference edge e, and making it the parent of μ so that it becomes the root.

Each edge in G is associated with a Q-node in $\mathcal{T}$. Each edge e_i in skeleton of μ is associated with the child μ_i of μ. It is possible to root $\mathcal{T}$ at an arbitrary Q-node μ', resulting in an SPQR-tree with respect to the edge associated with μ' [9]. In our implementation, we use a slightly different, but equivalent, definition of SPQR-tree. We omit Q-nodes and distinguish between *real edges* and *virtual edges* in the skeleton graphs instead. An edge in the skeleton of μ which is associated with a Q-node in the original definition is a real edge that is not associated with a child of μ, all other skeleton edges are virtual edges associated with a P-, S-, or R-node. Using this modified definition, we can show that the skeleton graphs are the unique triconnected components of G:

Theorem 1. *Let G be a biconnected multi-graph and $\mathcal{T}$ its SPQR-tree.*

(i) *The skeleton graphs of $\mathcal{T}$ are the triconnected components of G. P-nodes correspond to bonds, S-nodes to polygons, and R-nodes to triconnected simple graphs.*
(ii) *There is an edge between two nodes $\mu, \nu \in \mathcal{T}$ if and only if the two corresponding triconnected components share a common virtual edge.*
(iii) *The size of $\mathcal{T}$, including all skeleton graphs, is linear in the size of G.*

Proof. (sketch) We remark that if $\{u, v\}$ is a separation pair, the split components of $\{u, v\}$ are the separation classes with respect to $\{u, v\}$. In the parallel, series, and rigid case of the definition of SPQR-tree, subgraphs $G_1, \ldots, G_k$ are considered. Assume that $G_1, \ldots, G_\ell$ contains more than one edge, and $G_{\ell+1}, \ldots, G_k$ contains exactly one edge. In each of the three cases, the recursive decomposition step can be realized by performing ℓ split operations, each splitting off one G_i, $1 \leq i \leq \ell$ and introducing a new virtual edge e' in the skeleton of node μ and the skeleton of a child μ_i of μ. since e' remains in the skeleton of μ_i in subsequent steps, part (ii) of the theorem follows.

The final skeleton graphs are each either a polygon, a bond, or a simple triconnected graph, and no two polygons, and no two bonds share a common virtual edge. Thus, the skeleton graphs are the unique triconnected components of G proving part (i). The last part of the theorem follows directly from (i) and Lemma 1. □

4 The Algorithm

Let G be a biconnected multi-graph without self-loops. According to Theorem 1, it suffices to compute the triconnected components of G, which give us enough information to build the SPQR-tree of G. We correct the faulty parts in the algorithm by Hopcroft and Tarjan [15] and apply this modified algorithm for computing the triconnected components. We focus on the computation of split pairs, because the description of this part in [15] is not only confusing but contains also severe errors. For an overview of the Hopcroft and Tarjan algorithm, please refer to [15] or [13].

4.1 Computing SPQR-Trees

Input to the algorithm is a biconnected multi-graph $G = (V, E)$ and a reference edge e_r. In the first step, bundles of multiple edges are replaced by a new virtual edge as shown in Alg. 1. This creates a set of bonds $C_1, \ldots, C_k$ and results in a simple graph G'. The required sorting of the edges in line **1.1** can be done in $\mathcal{O}(|V| + |E|)$ time using bucket sort two times. Firstly according to the endpoint with lower index, and secondly to the one with higher index, where we assume that vertices have unique indices in the range $1, \ldots, |V|$. The for-loop in line **1.2** iterates over all edges, so Alg. 1 has running time $\mathcal{O}(|V| + |E|)$.

Algorithm 1: Split off multiple edges

```
1.1 Sort edges such that all multiple edges come after each other
1.2 for each maximal bundle of multiple edges $e_1, \ldots, e_\ell$ with $\ell \geq 2$ do
        let $e_1, \ldots, e_\ell$ be edges between $v$ and $w$
        replace $e_1, \ldots, e_\ell$ by a new edge $e' = (v, w)$
        create a new component $C = \{e_1, \ldots, e_\ell, e'\}$
    end
```

The second step finds the split components $C_{k+1}, \ldots, C_m$ of G'. The procedure is presented in detail in the next subsection. The triconnected components of the input graph G are created by partially reassembling the components $C_1, \ldots, C_m$. As long as two bonds or two polygons C_i and C_j containing the same virtual edge exist, C_i and C_j are merged. This is shown in Alg. 2. Removed components are marked as empty. The forall-loop in line **2.1** steps over *all* edges in C_i, i.e. those added to C_i during the loop. The test in line **2.1** can be done in constant time by precomputing for each virtual edge e the two components to which e belongs. We represent the edges in a component C_i by a list of edges, which allows to implement the set operations in lines **2.3** and **2.4** in constant time. According to Lemma 1, the total number of edges in all components is $\mathcal{O}(|E|)$, so Alg. 2 can also be implemented in time $\mathcal{O}(|V| + |E|)$.

Algorithm 2: Build triconnected components

```
         for $i := 1$ to $m$ do
             if $C_i \neq \emptyset$ and $C_i$ is a bond or a polygon then
2.1              forall $e \in C_i$ do
2.2                  if there exists $j \neq i$ with $e \in C_j$ and $type(C_i) = type(C_j)$ then
2.3                      $C_i := (C_i \cup C_j) \setminus \{e\}$
2.4                      $C_j := \emptyset$
                     end
                 od
             end
         end
```

The preceding steps give enough information to build the SPQR-tree $\mathcal{T}$ of G. Applying Theorem 1, it is easy to construct the unrooted version of $\mathcal{T}$. Since we omit Q-nodes in our representation, we root $\mathcal{T}$ at the node whose skeleton contains the reference edge e_r. During the construction, we also create cross links between each tree edge $\mu \to \nu$ in $\mathcal{T}$ and the two corresponding virtual edges in skeleton of μ and skeleton of ν.

4.2 Finding Separation Pairs

Suppose we have a palm tree P for the simple, biconnected graph $G' = (V, E')$, and the vertices of G' are numbered $1, \ldots, |V|$. In the following, we identify vertices with their numbers. We introduce the following notation:

$$lowpt1(v) = \min\Big(\{v\} \cup \{w \mid v \xrightarrow{*}\hookrightarrow w\}\Big)$$
$$lowpt2(v) = \min\Big(\{v\} \cup \Big(\{w \mid v \xrightarrow{*}\hookrightarrow w\} \setminus \{lowpt1(v)\}\Big)\Big)$$

That is, $lowpt1(v)$ is the *lowest* vertex reachable by traversing zero or more tree arcs followed by one frond of P (or v if no such vertex exists), and $lowpt2(v)$ is the *second lowest* vertex reachable this way (or v if no such vertex exists).

We denote with $Adj(v)$ the ordered (non-cyclic) adjacency list of a vertex v, and with $D(v)$ the set of descendants of v. We seek for a numbering of the vertices and ordering of the edges in the adjacency lists satisfying the following properties:

(P1) the root of P is 1.
(P2) if $v \in V$ and $w_1, \ldots, w_n$ are the children of v in P according to the ordering in $Adj(v)$, then $w_i = w + |D(w_{i+1}) \cup \ldots \cup D(w_n)| + 1$.
(P3) the edges e in $Adj(v)$ are in ascending order according to $lowpt1(w)$ if $e = v \rightarrow w$, or w if $e = v \hookrightarrow w$, respectively.
Let $w_1, \ldots, w_n$ be the children of v with $lowpt1(w_i) = u$ in the order given by $Adj(v)$. Then there exists an i_0 such that $lowpt2(w_i) < v$ for $1 \le i \le i_0$, and $lowpt2(w_j) \ge v$ for $i_0 < j \le n$. If $v \hookrightarrow u \in E'$, then $v \hookrightarrow u$ comes in $Adj(v)$ between $v \rightarrow w_{i_0}$ and $v \rightarrow w_{i_0+1}$.

It is shown in [15], how to compute such a numbering of the vertices and ordering of the adjacency lists in linear time. Unlike [15], we demand that a frond $v \hookrightarrow w$, if contained in E', must come between $v \rightarrow w_{i_0}$ and $v \rightarrow w_{i_0+1}$ in $Adj(v)$. This can easily be done by adapting the sorting function ϕ used in [15]:

$$\phi(e) = \begin{cases} 3lowpt1(w) & \text{if } e = v \rightarrow w \text{ and } lowpt2(w) < v \\ 3w + 1 & \text{if } e = v \hookrightarrow w \\ 3lowpt1(w) + 2 & \text{if } e = v \rightarrow w \text{ and } lowpt2(w) \ge v \end{cases}$$

The required ordering can be obtained by sorting the edges according to their ϕ-values using bucket sort. Using ordering ϕ and procedure PATHSEARCH as suggested in [15] will not recognize all multiple edges and thus not correctly compute the split components of G'.

Suppose we perform a depth-first-search on G' using the ordering of the edges in the adjacency lists. This divides G' into a set of paths consisting of zero or more tree arcs followed by one frond. The first path starts at vertex 1 and a path ends, when the first frond on the path is reached (see Fig. 1). Each path ends at the lowest possible vertex, and has only its initial and terminal vertex in common with previously traversed paths. From each such path $p : v \overset{*}{\Rightarrow} w$, we can form a cycle by adding the tree path from $w \overset{*}{\rightarrow} v$ to p (compare [15,16]).

Example 1. Fig. 1 shows a palm tree with a numbering that satisfies (P1)-(P3). The edges are numbered according to the generated paths. The generated paths are

1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 13 \hookrightarrow 1$	7: $12 \hookrightarrow 9$
2: $13 \hookrightarrow 2$	8: $10 \rightarrow 11 \hookrightarrow 8$
3: $3 \rightarrow 4 \hookrightarrow 1$	9: $11 \hookrightarrow 9$
4: $4 \rightarrow 5 \rightarrow 8 \hookrightarrow 1$	10: $5 \rightarrow 6 \rightarrow 7 \hookrightarrow 4$
5: $8 \rightarrow 9 \rightarrow 10 \rightarrow 12 \hookrightarrow 1$	11: $7 \hookrightarrow 5$
6: $12 \hookrightarrow 8$	12: $6 \hookrightarrow 4$

We need one more definition: u_n is a *first descendant* of u_0 if $u_0 \rightarrow \cdots \rightarrow u_n$ and each $u_i \rightarrow u_{i+1}$ is the first edge in $Adj(u_i)$. In the sequel, we consider a palm

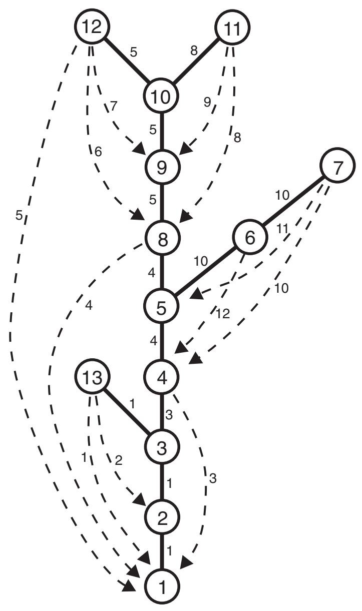

Fig. 1. Palm tree with numbered vertices and generated paths.

tree P satisfying (P1)-(P3). The following lemma gives us three easy-to-check conditions for separation pairs.

Lemma 3. (Lemma 13 in [15]) *Let $G = (V, E)$ be a biconnected graph and a, b be two vertices in G with $a < b$. Then $\{a, b\}$ is a separation pair if and only if one of the following conditions holds.*

Type-1 Case: There are distinct vertices $r \neq a, b$ and $s \neq a, b$ such that $b \to r$, $lowpt1(r) = a$, $lowpt2(r) \geq b$, and s is not a descendant of r.

Type-2 Case: There is a vertex $r \neq b$ such that $a \to r \overset{*}{\to} b$, b is a first descendant of r, $a \neq 1$, every frond $x \hookrightarrow y$ with $r \leq x < b$ has $a \leq y$, and every frond $x \hookrightarrow y$ with $a < y < b$ and $b \to w \overset{*}{\to} x$ has $lowpt1(w) \geq a$.

Multiple Edge Case: (a, b) is a multiple edge of G and G contains at least four edges.

Example 2. Consider the palm tree from Fig. 1. We have the following separation pairs:

type-1 pairs: $(1, 4), (1, 5), (4, 5), (1, 8), (1, 3)$
type-2 pairs: $(4, 8), (8, 12)$

4.3 Finding Split Components

During the algorithm, we maintain a graph G_c and a palm tree P_c of G_c. We denote with $deg(v)$ the degree of v in G_c, with $v \to w$ a tree arc in P_c, with $v \hookrightarrow w$ a frond in P_c, with $parent(v)$ the parent of v in P_c, and with $ND(v)$ the number of descendants of v in P_c. Each time we identify a split component C, we split it off, and G_c and P_c are updated. We use the following update functions:

$C :=$ `new_component`$(e_1, \ldots, e_\ell)$: a new component $C = \{e_1, \ldots, e_\ell\}$ is created, and $e_1, \ldots, e_\ell$ are removed from G_c.
$C := C \cup \{e_1, \ldots, e_\ell\}$: the edges $e_1, \ldots, e_\ell$ are added to C and removed from G_c.
$e' :=$ `new_virtual_edge`(v, w, C): a new virtual edge $e' = (v, w)$ is created and added to component C and G_c.
`make_tree_edge`$(e, v \to w)$: makes edge $e = (v, w)$ a new tree edge in P_c.

Moreover, we define the access functions

$$firstChild(v) = \text{first child of } v \text{ in } P_c \text{ according to } Adj(v).$$

$$high(w) = \begin{cases} 0 & \text{if } F(w) = \emptyset \\ \text{source vertex of first visited edge in } F(w) & \text{otherwise} \end{cases}$$

where $F(w) = \{v \mid v \hookrightarrow w \in E_c\}$, and we use two stacks for which the usual functions `push`, `pop`, and `top` are defined:

ESTACK contains already visited edges that are not yet assigned to a split component.
TSTACK contains triples (h, a, b) (or a special end-of-stack marker EOS), such that $\{a, b\}$ is a potential type-2 separation pair, and h is the highest numbered vertex in the component that would be split off.

The algorithm starts by calling the recursive procedure `PathSearch` for vertex 1, the root vertex of P (see Alg. 3). When returning from the call, the edges belonging to the last split component are on ESTACK.

Algorithm 3: Find split components

```
     TSTACK.push($EOS$)
     PathSearch(1)
     let $e_1, \ldots, e_\ell$ be the edges on ESTACK
3.1  $C :=$ new_component($e_1, \ldots, e_\ell$)
```

Procedure `PathSearch` is shown in Alg. 4. The testing for separation pairs applying Lemma 3 is depicted separately in Alg. 5 for type-2 and in Alg. 6 for type-1 separation pairs[1]. For a detailed description of the algorithm, please refer to [15,13]. In order to achieve linear running time, we set up the following data structures:

[1] The algorithm will not find *all* separation pairs, but only the separation pairs needed for dividing the graph into its split components

Algorithm 4: PathSearch(v)

```
    forall $e \in Adj(v)$ do
        if $e = v \to w$ then
            if $e$ starts a path then
                pop all $(h, a, b)$ with $a > lowpt1(w)$ from TSTACK
                if no triples deleted then
                    TSTACK.push($w + ND(w) - 1, lowpt1(w), v$)
                else
                    $y := \max\{h \mid (h, a, b)$ deleted from TSTACK $\}$
                    let $(h, a, b)$ be last triple deleted
                    TSTACK.push($\max(y, w + ND(w) - 1), lowpt1(w), b$)
                end
                TSTACK.push($EOS$)
            end

            PathSearch($w$)
            ESTACK.push($v \to w$)

            check for type-2 pairs
            check for a type-1 pair

            if $e$ starts a path then
                remove all triples on TSTACK down to and including EOS
            end
4.1         while $(h, a, b)$ on TSTACK has $a \neq v$ and $b \neq v$ and $high(v) > h$ do
                TSTACK.pop()
            od
        else
            let $e = v \hookrightarrow w$
            if $e$ starts a path then
                pop all $(h, a, b)$ with $a > w$ from TSTACK
                if no triples deleted then
                    TSTACK.push($v, w, v$)
                else
                    $y := \max\{h \mid (h, a, b)$ deleted from TSTACK $\}$
                    let $(h, a, b)$ be last triple deleted
                    TSTACK.push($y, w, b$)
                end
            end
            if $w = parent(v)$ then
                $C$ := new_component($e, w \to v$)
                $e'$ := new_virtual_edge($w, v, C$)
                make_tree_edge($e', w \to v$)
            else
                ESTACK.push($e$)
            end
        end
    od
```

Algorithm 5: check for type-2 pairs

```
while $v \neq 1$ and $(((h,a,b)$ on TSTACK has $a = v)$ or $(deg(w) = 2$ and
firstChild$(w) > w))$ do
    if $a = v$ and $parent(b) = a$ then
        TSTACK.pop()
    else
        $e_{ab} := nil$
        if $deg(w) = 2$ and $firstChild(w) > w$ then
            $C :=$ new_component()
            remove top edges $(v,w)$ and $(w,b)$ from ESTACK and add to $C$
            $e' :=$ new_virtual_edge($v, x, C$)
            if ESTACK.top() $= (v,b)$ then $e_{ab} :=$ ESTACK.pop()
        else
            $(h,a,b) :=$ TSTACK.pop()
            $C :=$ new_component()
            while $(x,y)$ on ESTACK has $a \leq x \leq h$ and $a \leq y \leq h$ do
                if $(x,y) = (a,b)$ then $e_{ab} :=$ ESTACK.pop()
                else $C := C \cup \{$ ESTACK.pop() $\}$
            od
            $e' :=$ new_virtual_edge($a, b, C$)
        end
        if $e_{ab} \neq nil$ then
            $C :=$ new_component($e_{ab}, e'$)
            $e' :=$ new_virtual_edge($v, b, C$)
        end
        ESTACK.push($e'$); make_tree_edge($e', v \rightarrow b$); $w := b$
    end
od
```

Algorithm 6: check for a type-1 pair

```
6.1 if $lowpt2(w) \geq v$ and $lowpt1(w) < v$ and $(parent(v) \neq 1$ or $v$ is adjacent to
    a not yet visited tree arc$)$ then
        $C :=$ new_component()
        while $(x,y)$ on ESTACK has $w \leq x < w+ND(w)$ or $w \leq y < w+ND(w)$
        do
            $C := C \cup \{$ ESTACK.pop() $\}$
        od
        $e' :=$ new_virtual_edge($v, lowpt1(w), C$)
        if ESTACK.top() $= (v, lowpt1(w))$ then
            $C :=$ new_component(ESTACK.pop(),$e'$)
            $e' :=$ new_virtual_edge($v, lowpt1(w), C$)
        end
        if $lowpt1(w) \neq parent(v)$ then
            ESTACK.push($e'$)
            make_tree_edge($e', lowpt1(w) \rightarrow v$)
        else
            $C :=$ new_component($e', lowpt1(w) \rightarrow v$)
            $e' :=$ new_virtual_edge($lowpt1(w), v, C$)
            make_tree_edge($e', lowpt1(w) \rightarrow v$)
        end
    end
```

– The palm tree P is represented by arrays PARENT[v], TREE_ARC[v] (the tree arc entering v), and TYPE[e] (tree arc or frond).
– The values $lowpt1(v)$, $lowpt2(v)$, and $ND(v)$ are precomputed. It is not necessary to update them.
– An array DEGREE[v] contains the degree of $v \in G_c$. It is updated each time an edge is added to or removed from G_c.
– In order to compute $firstChild(v)$, we update the adjacency lists each time an edge is added to or removed from G_c.
– In order to compute $high(v)$, we precompute the list of fronds $v_i \hookrightarrow w$ ending at w in the order they are visited. When a frond is removed from or added to G_c, the respective list is updated.
– We precompute an array START[e] which is true iff e starts a path.
– The test if "v is adjacent to a not yet visited tree arc" in line **6.1** can be done by simply counting the visited adjacent tree arcs.

5 Corrections on the Hopcroft and Tarjan Algorithm

Procedure SPLIT in [15] does not correctly split a graph into its split components. We summarize the important changes we have made in our algorithm:

– The sorting function ϕ had to be modified as described in subsection 4.2 in order to identify all multiple edges.
– The creation of the last split component (line **3.1**) was missing.
– The condition in line **4.1** was changed. The original condition could remove triples from TSTACK corresponding to real type-2 separation pairs. Such a separation pair could not be recognized by the original SPLIT procedure.
– The condition in line **6.1** was changed. The original condition could incorrectly identify separation pairs after the graph had been modified.
– The updates for $firstChild(v)$ (which is $A1(v)$ in [15]) and DEGREE(v) were not sufficient.
– $high(w)$ (which is HIGHPT(w) in [15]) was not updated, which is not correct. It is necessary to update HIGHPT dynamically, when G_c is modified. We replaced HIGHPT(w) by a list of fronds ending at w, which is updated as G_c changes.

6 Computational Experiments

Our implementation is based on LEDA [20] and made publically available in AGD [1]. We tested our implementation with generated planar and non-planar biconnected graphs, and the benchmark graphs collected by Di Battista et al. [7] ranging from 10 to 100 vertices. A planar biconnected graph with n vertices and m edges is generated by n randomly chosen split-edge and $m - n$ split face operations. A general biconnected graph is generated by creating a random graph G and making G biconnected by augmenting edges. The computed SPQR-trees are automatically checked with several consistency and plausibility tests. The average running times are depicted in Fig. 2. The x-axis shows the number of vertices, and the y-axis the running time in seconds. The left side shows the results for generated graphs with n vertices and m edges applying the algorithm to 100 test instances each. Even very large instances with 40000 edges could be solved within less than 4 seconds. The right side shows the results for the benchmark graphs applying the algorithm to each biconnected component containing at least three edges. Graphs with 100 vertices took about 0.02 seconds.

Recently, Gutwenger, Mutzel, and Weiskircher [14] have presented a linear time algorithm for solving the one edge insertion problem optimally (minimum number of crossings) over all combinatorial embeddings. Their algorithm requires to compute SPQR-trees for some biconnected components of a planar input graph. In their tests, they also used the benchmark graphs from [7] and applied our implementation of SPQR-trees. The longest running time for a graph with 100 vertices was 1.15 seconds, where 38 edges had to be inserted.

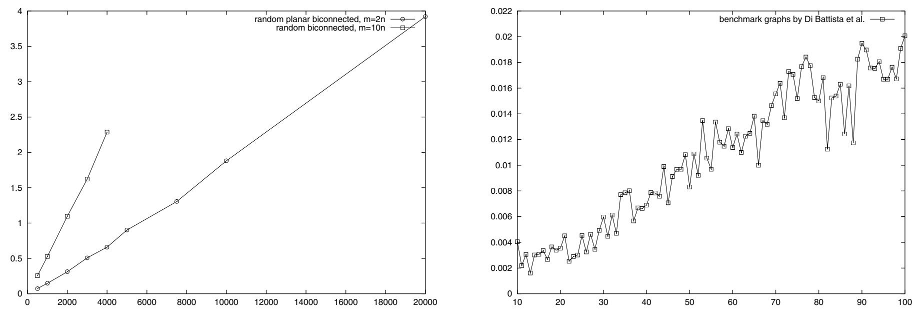

Fig. 2. Average running times (system configuration: Pentium II, 400 MHz, 128MB RAM)

References

[1] *AGD User Manual (Version 1.1.2)*, Feb. 2000. Max-Planck-Institut Saarbrücken, Technische Universität Wien, Universität zu Köln, Universität Trier. See also `http://www.mpi-sb.mpg.de/AGD/`.

[2] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In M. S. Paterson, editor, *Proc. of the 17th International Colloqium on Automata, Languages and Prog ramming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 598–611. Springer-Verlag, 1990.

[3] P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. In *Proc. 5th Workshop Algorithms, Data Struct.*, volume 1272 of *Lecture Notes in Computer Science*, pages 331–344, 1998.

[4] P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino. Optimal upward planarity testing of single-source digraphs. *SIAM J. Comput.*, 27(1):132–169, 1998.

[5] D. Bienstock and C. L. Monma. On the complexity of embedding planar graphs to minimize certain distance measures. *Algorithmica*, 5(1):93–109, 1990.

[6] Elias Dahlhaus. A linear time algorithm to recognize clustered planar graphs and its parallelization. In C. L. Lucchesi and A. V. Moura, editors, *LATIN '98: Theoretical Informatics, Third Latin American Symposium*, volume 1380 of *Lecture Notes in Computer Science*, pages 239–248. Springer-Verlag, 1998.

[7] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, and F. Vargiu. An experimental comparision of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–326, 1997.

[8] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 436–441, 1989.

[9] G. Di Battista and R. Tamassia. On-line maintanance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.

[10] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997, 1996.

[11] W. Didimo. Dipartimento di Informatica e Automazione, Università di Roma Tre, Rome, Italy, Personal Communication.

[12] *GDToolkit Online Documentation*. See `http://www.dia.uniroma3.it/~gdt`.

[13] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. Technical report, Technische Universität Wien, 2000. To appear.

[14] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*. ACM Press, 2001. To appear.

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

[16] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.

[17] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica, Special Issue on Graph Drawing*, 16(1):4–32, 1996.

[18] T. Lengauer. Hierarchical planarity testing. *Journal of the ACM*, 36:474–509, 1989.

[19] S. MacLaine. A structural characterization of planar combinatorial graphs. *Duke Math. J.*, 3:460–472, 1937.

[20] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. to appear.

[21] P. Mutzel and R. Weiskircher. Optimizing over all combinatorial embeddings of a planar graph. In G. Cornuéjols, R. Burkard, and G. Woeginger, editors, *Proceedings of the Seventh Conference on Integer Programming and Combinatorial Optimization (IPCO)*, volume 1610 of *LNCS*, pages 361–376. Springer Verlag, 1999.