Maintenance of 2- and 3-Connected Components of Graphs, Part I: 2- and 3-Edge-Connected Components

J.A. La Poutré, J. van Leeuwen and M.H. Overmars

RUU-CS-90-26
July 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Maintenance of 2- and 3-Connected Components of Graphs, Part I: 2- and 3-Edge-Connected Components

J.A. La Poutré, J. van Leeuwen and M.H. Overmars

Technical Report RUU-CS-90-26
July 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

The results of this paper were presented at the French-Israeli Conference on Combinatorics and Algorithms, November 1988, Jerusalem (Israel).

A preliminary version of this paper was completed in Februari 1989. The final version of this paper was released in June 1991.

ISSN:0924-3275

Maintenance of 2- and 3-connected components of graphs, Part I: 2- and 3-edge-connected components*

J.A. La Poutré, J. van Leeuwen and M.H. Overmars§

Abstract

In this paper a data structure is presented to efficiently maintain the 2- and 3-edge-connected components of a graph, under insertions of edges in the graph. Starting from an "empty" graph of n nodes, the insertion of e edges takes $O(n \log n + e)$ time in total. The data structure allows for insertions of nodes also (in the same time bounds, taking n as the final number of nodes). Moreover, at any moment, the data structure can answer the following type of query in $O(1)$ time: given two nodes in the graph, are these nodes 2- or 3-edge-connected.

1 Introduction

Recently there has been a growing interest in dynamic or on-line graph algorithms (see e.g. [3, 8, 9, 10, 18]). A graph algorithm is called dynamic or on-line if it maintains some information related to a graph while the graph is being changed (e.g. by inserting or deleting a node or an edge). A dynamic algorithm exploits a suitable data representation for a graph and uses information of the old graph to compute the required information for the new updated graph. It is anticipated that a dynamic algorithm does not need to compute a new solution for the new graph from scratch, i.e., by using the new graph as input only, and a better performance may be expected compared to an algorithm that simply "recomputes". Dynamic algorithms are known for e.g. computing transitive closures (cf. [8, 9, 10], or cf. [17] for planar graphs), minimal spanning trees (cf. [3]), incremental planarity testing (cf. [2]) and maintaining shortest paths (cf. [18]). One sometimes uses the term "on-line" algorithm when only insertions (of nodes or edges) are allowed.

*This research was partially supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM).

§Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

In this paper we consider the problem of maintaining 2- and 3-edge-connected components of a graph under insertions of edges (and vertices), where k-edge-connectivity is defined as follows. Let G be an undirected graph. Two nodes x and y are called k-edge-connected in G ($k \geq 1$) if after the removal of any set of at most $k-1$ edge(s) x and y are (still) connected (i.e., there is a path between x and y).

We present a data structure with algorithms for maintaining the 2- and 3-edge-connectivity relation of a graph. The algorithm starts from an "empty" graph of n nodes (i.e., a graph with no edges) in which edges are inserted one by one and where at any time for any two nodes the query whether these nodes are 2- or 3-edge-connected can be answered in $O(1)$ time. Moreover, the 2- and 3-edge-connected components are maintained. The insertion of e edges takes $O(n \log n + e)$ time altogether. By using additional data structuring techniques, the time bounds for 2- and 3-edge-connectivity can be improved, as will be demonstrated in [14]. The algorithms have an improved running time of $O(n + m.\alpha(m,n))$ where m is the number of edge insertions and queries and where $\alpha(m,n)$ denotes the inverse Ackermann function. Recently, Westbrook and Tarjan [21] independently obtained the same time bounds for 2-edge-connectivity. The methods though are quite different. Moreover, our method for obtaining the results on 2-edge-connectivity can be used for 3-edge-connectivity as well, as will be shown.

The paper is organized as follows. In Section 2 we introduce some terminology and state properties on connectivity. In Section 3 we consider the 2-edge-connectivity problem and in Section 4 we consider the 3-edge-connectivity problem: first we consider 3-edge-connectivity in 2-edge-connected graphs and then we extend this to general graphs.

2 Preliminaries

2.1 Graphs and terminology

Let $G =< V, E >$ be an undirected graph with V the set of vertices and E the set of edges. The edge set E consists of edges with the incidence relation in the following form: an edge is a triple (e, x, y), where e is the edge name and x and y are the end nodes of the edge. The order of the end nodes x and y of an edge is not relevant (hence, $(e, x, y) = (e, y, x)$). Moreover, all edge names are required to be distinct. Therefore we can denote an edge by its name only.

We use the following notions (see also [7]). A *path* between two nodes x and y is

an alternating sequence of nodes and edges such that x and y are at the end of this sequence and each edge is bracketed by its *end nodes* x and y. However, we often

consider a path to consist of the (sub)sequence of the nodes only. A path is *nontrivial* if it contains at least 2 distinct nodes. A path is *simple* if no node occurs twice in it. Two paths are called *edge disjoint* if they do not have a common edge. Two nodes are called *connected* if there exists a path between them. A (elementary) *cycle* is a path where the end nodes are equal and where no edge occurs twice. A cycle containing just one distinct node is called *trivial*, otherwise it is called nontrivial. A cycle is *simple* if there is no node that occurs twice in the sequence except for the end nodes.

Definition 2.1 *Nodes x and y are k-edge-connected ($k \geq 1$) if after the removal of any set of at most $k-1$ edge(s) x and y are (still) connected. If the removal of a set of edges separates the vertices x and y (i.e., x and y are not connected), then that set is called a cut edge set for x and y.*

It is easily seen that if two nodes are k-edge-connected, then they are k'-edge-connected for any k' with $1 \leq k' \leq k$. We state some lemmas. The following lemma of Menger (cf. [16]) characterizes k-edge-connected vertices.

Lemma 2.2 [Menger] *Two nodes x and y are k-edge-connected iff there exist k edge-disjoint paths between x and y.*

A special case of this lemma occurs for $k = 2$: two nodes are 2-edge-connected iff they lie on a common elementary cycle.

Lemma 2.3 *k-edge-connectivity is an equivalence relation on the set of nodes of a graph.*

The 2-edge-connected components of a graph G are subgraphs of G that are induced by equivalence classes of nodes w.r.t. 2-edge-connectivity. To be precise, 2-edge-connected components are defined as follows.

Definition 2.4 *Let $G =< V, E >$ be a graph. Let $C \subseteq V$ be an equivalence class w.r.t. to 2-edge-connectivity. Then $< C, \{(e,x,y) \in E | x,y \in C\} >$ is called a 2-edge-connected component of G (induced by C).*

In this paper we will represent the 2/3-edge-connected components in a graph by means of a "super" graph. To this end we introduce the notion of a class node.

Definition 2.5 *Let $G =< V, E >$ be a graph. Let V be partitioned in classes and let some (new) distinct node be related to each class, called the class node of that class. Let $cc(x)$ be the class node of the class in which x is contained ($x \in V$). Then*

the induced node set $cc(V)$*, the induced edge set* $cc(E')$ *of a set of edges* $E' \subseteq E$ *and the induced graph* $cc(G)$ *are given by*

$$\begin{aligned} cc(V) &:= \{cc(x) | x \in V\} \\ cc(E') &:= \{(e, cc(x), cc(y)) | (e, x, y) \in E' \wedge cc(x) \neq cc(y)\} \\ cc(G) &:= < cc(V), cc(E) > \end{aligned}$$

We state some lemmas.

Lemma 2.6 *Let* $G =< V, E >$ *be a graph and let* k *be a positive integer. Let* V *be partitioned in classes and let some (new) distinct node be related to each class. Suppose that any two nodes* x *and* y *that are in the same class are* k*-edge-connected. Let* $cc(x)$ *be the class node of the class in which* x *is contained* ($x \in V$)*. Then the following holds.*

1. *A collection* $E' \subseteq E$ *of at most* $k-1$ *edges is a cut edge set for* $x, y \in V$ *in* G *iff the induced edge set* $cc(E')$ *is a cut edge set of* $cc(x)$ *and* $cc(y)$ *in* $cc(G)$*.*
2. *If* E' *is a cut edge set for nodes* x *and* y *of at most* $k-1$ *edges and if* $(e, u, v) \in E$ *such that* $cc(u) = cc(v)$*, then* $E' \backslash \{(e, u, v)\}$ *is a cut edge set for* x *and* y *too.*
3. *For all* $x, y \in V$ *and* $1 \leq k' \leq k$*,* x *and* y *are* k'*-edge-connected in* G *iff* $cc(x)$ *and* $cc(y)$ *are* k'*-edge-connected in* $cc(G)$*.*

Proof. Let $E' \subseteq E$ be a set of at most $k-1$ edges.

If E' is not a cut edge set for x and y, then there exists a path P in G between x and y that does not use an edge of E'. The corresponding path of P in $cc(G)$ is a path between $cc(x)$ and $cc(y)$ that does not use an edge of $cc(E')$. Hence $cc(E')$ is not a cut edge set in $cc(G)$.

Suppose $cc(E')$ is not a cut edge set for $cc(x)$ and $cc(y)$ in $cc(G)$. Then there exists a simple path CP between $cc(x)$ and $cc(y)$ in $cc(G)$ that does not use edges of $cc(E')$. Let P be the path in G constructed from CP as follows. Each edge $(e, cc(u), cc(v))$ in CP is replaced by the (unique) edge (e, u, v) in G. Moreover, the vertices u and v bracket this edge in P in the proper order (i.e., if $cc(u)$ occurs before $cc(v)$ in CP, then u occurs before v in P). Finally bracket the obtained sequence with the nodes x and y. Now we have a sequence of nodes and edges in G such that each edge is bracketed by its end nodes and such that two consecutive nodes u, v in P without an edge in between are in the same class with class representative $cc(u)$ $(= cc(v))$. Since a class is k-edge-connected and since E' contains at most $k-1$ edges, there exists a path between such u and v that does not use an edge of E'. Now we can obtain the path P from the above sequence by inserting these paths between these nodes. Hence, P is a path in G that does not contain nodes of E'. Therefore, E' is not a cut edge set for x and y in G. This concludes the proof of the first statement.

If E' is a cut edge set for nodes x and y of at most $k-1$ edges and if E' contains an edge (e,u,v) such that $cc(u) = cc(v)$, then $(e, cc(x), cc(y)) \notin cc(E')$ while $cc(E')$ is a cut edge set for $cc(x)$ and $cc(y)$ in $cc(G)$. Hence, by the first statement, $E' \backslash \{(e,x,y)\}$ is a cut edge set for x and y too. This proves the second statement.

The third statement now follows since we only have to consider cut edge sets E' with $|E'| = |cc(E')|$. □

In other words: "internal" edges of classes of k-edge-connected nodes are not relevant for cut edge sets up to size $k-1$. The following lemma is based on the observation that for two nodes that are k-edge-connected, there exist k edge-disjoint paths between them, and hence, all the nodes on these paths are 2-edge-connected .

Lemma 2.7 *Let $G =< V, E >$ be a graph. Let H be a 2-edge-connected component of G. Then H is a 2-edge-connected graph. Moreover, nodes $x, y \in H$ are k-edge-connected in H iff they are k-edge-connected in G ($k \geq 1$).*

Proof. Let x and y be two nodes of H. Suppose there are k edge disjoint paths in G between x and y, for some $k \geq 2$. (For $k = 1$ the lemma is trivial.) Let P_1 and P_2 be any two of these paths. Now between x and a node a on P_1 there are 2 edge disjoint paths: they can be obtained by splitting P_1 at a and by concatenating P_2 with the appropriate part of P_1 in reversed order. Hence, all nodes on P_1 are in H and therefore P_1 is a path in H. Hence, G and H contain the same paths between x and y. □

Since a definition of 3-edge-connected components similar to Definition 2.4 does not yield 3-edge-connected graphs, we introduce the following definition, that can described informally as: a 3-edge-connected component J of a graph G is a subgraph that is induced by an equivalence class C of nodes of G w.r.t. the 3-edge-connectivity relation and that is extended with a collection of additional (new) edges such that two nodes x and y are k-edge-connected in G iff they are k-edge-connected in J.

Definition 2.8 *Let $G =< V, E >$ be a graph. Let C be an equivalence class of V w.r.t. to 3-edge-connectivity. For each pair x, y of nodes in C let $f(\{x,y\})$ be the maximal number of non-trivial edge disjoint paths between x and y that intersect with C at x and y only and that intersect with $V \backslash C$. Let $E(\{x,y\})$ be a set of $f(\{x,y\})$ new edges with end nodes x and y, called auxiliary edges. Let $E(C)$ be the set*

$$E(C) := \{(e,x,y) \in E | x, y \in C\} \cup \bigcup_{x,y \in C} E(\{x,y\})$$

Then the graph $< C, E(C) >$ is called a 3-edge-connected component of G (induced by C).

Lemma 2.9 *Let $G =< V, E >$ be a graph. Let C be an equivalence class of V w.r.t. 3-edge-connectivity. Let J be a 3-edge-connected component of G induced*

by C. Then J is a 3-edge-connected graph. Moreover, nodes $x, y \in C$ are k-edge-connected in J iff they are k-edge-connected in G.

Proof. Let $x, y \in C$. Suppose there exist k edge-disjoint paths in G between x and y. W.l.o.g. these paths are simple. Now replace each part of a path between two nodes a and b of C that consist of nodes in $V \backslash C$ by an edge of $E(\{a, b\})$ that is not used in some other path already. Since the k paths do not intersect in edges and by the definition of $f(\{a, b\})$, $E(\{a, b\})$ and $E(C)$ it follows that this procedure yields k edge-disjoint paths between x and y within J.

On the other hand, suppose there are k edge disjoint paths in J between x and y (w.l.o.g. these paths are simple). By the definition of $E(C)$, the edges in $E(C) \backslash E$ with end nodes a and b can be replaced by an equal number of edge disjoint simple paths in G between a and b such that each path does not contain other nodes of C except for their end nodes a and b and such that each path intersects with $V \backslash C$. Moreover, two such collections of paths, say, between a and b and between c and d, do not intersect in any node outside C if $\{a, b\} \neq \{c, d\}$. (This is seen as follows. Suppose $a \notin \{c, d\}$ and suppose that such a path P_1 from a to b and such a path P_2 from c to d intersect outside C. Then follow path P_1 starting at a until it intersects with P_2 at some node h, $h \notin C$. Then this path part together with the two paths obtained by splitting P_2 at h yield three edge-disjoint paths from h to nodes of C. Since C is 3-edge-connected it follows that h is 3-edge-connected with the nodes of C too, and hence $h \in C$, which yields a contradiction.) Hence there exist k edge disjoint paths between x and y in G. □

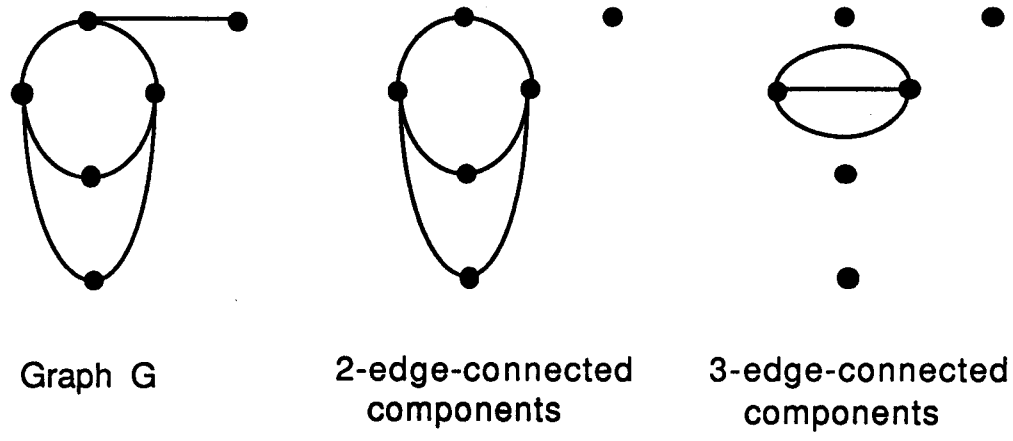
Note that the 3-edge-connected component of graph G with vertex set C defined in Lemma 2.9 is unique apart from the names of the new edges. In the sequel we will not bother about these exact names and just call such a graph *the* component of G with vertex set C. An example is given in Figure 1.

By means of Lemma 2.7 and Lemma 2.9 the following lemma easily follows.

Lemma 2.10 *Let G be a graph. Let C_2 be an equivalence class of V w.r.t. 2-edge-connectivity, and let C_3 be an equivalence class of V w.r.t. 3-edge-connectivity. Then either $C_2 \cap C_3 = \emptyset$ or $C_3 \subseteq C_2$. Let H be the 2-edge-connected component of G induced by C_2 and let J be the 3-edge-connected component of G induced by C_3. If $C_3 \subseteq C_2$ then C_3 is a 3-edge-connected equivalence class of H and moreover J is a 3-edge-connected component of H induced by C_3 (up to edge names).*

Stated differently, each 3-edge-connected component of G is a 3-edge-connected component of some 2-edge-connected component of G and reversely.

Figure 1: Two and three edge-connected components of graph G.



2.2 Representation and algorithms

In order to deal with the maintenance problem we represent a graph as follows. All nodes and edges of a graph are represented in memory by records, which we will consider to be the actual nodes and edges. I.e., we do not distinguish between a vertex (or an edge) and the record that represents it. Each vertex has an incidence list, that consist of pointers to all edges that are incident with that vertex. Also, each edge contains pointers to its two end nodes. (Hence, the vertices that are *adjacent* to some vertex v can be obtained by the incidence list of v and by the pointers from edges to their end nodes.) Finally, an edge that has to be inserted is given by its record with the pointers to its end nodes (according to the above representation) as input for the algorithms.

When we consider classes (sets) of nodes in a graph, we often refer to a class of nodes that is represented by a node c by "class c". Moreover, we will often not distinguish between a pointer to a record and the record itself.

Lemma 2.3 states that k-edge-connectivity is an equivalence relation. In our algorithms we need operations on equivalence classes like joining classes and determining in which class an element is contained. This problem is condensed in the *Union-Find problem*, which is given as follows. Let U be a universe of n nodes, called elements. Suppose U is partitioned into a collection of singleton sets and to each set some *node* is related as its name. Suppose we want to perform the following operations: Union(A,B), i.e., join the two sets named A and B into a new set and relate a node to the resulting set as its name, and Find(x), i.e., return the name of

the set (= the node related to the set) in which element x is contained. The thus occurring set names must satisfy the condition, that at every moment, the names of the existing sets are distinct. Many solutions have been proposed for the Union-Find problem (cf. [11, 19, 20]): these solutions all take $O(n + m.\alpha(m, n))$ time for all Unions and m Finds on n elements, which is optimal [4, 12]. However, in the most part of this paper we only use a simple algorithm [1] taking $O(n \log n)$ time for all Unions together and $O(1)$ time per Find, since additional computations already take $O(n \log n)$ time.

In the sequel, the Union-Find structure is used to maintain the equivalence classes for connectivity, 2-edge-connectivity and 3-edge-connectivity, where the Unions and Finds on the different kind of sets are denoted by $Union_c$, $Find_c$, $Union_{2ec}$, $Find_{2ec}$, $Union_{3ec}$ and $Find_{3ec}$, respectively. Note that this can easily be implemented by reserving a dedicated field for each type of (equivalence) set in each of the considered nodes, where this field either contains the (sub)field(s) for the corresponding Union-Find structure, or where it contains a pointer to a representative record of the node for the considered Union-Find structure. We often denote the above three types of Finds just by c, $2ec$ and $3ec$, respectively.

We consider the connectivity problem for edge insertions. Let $G =< V, E >$ be a graph. Suppose a sequence of edge insertions in G and queries whether two nodes are connected is performed. The equivalence classes of connected nodes are represented by a Union-Find structure on these nodes. The class to which node x belongs has $c(x)$ as its name. Hence, nodes x and y are connected iff $c(x) = c(y)$. If an edge (e, x, y) is inserted, there are two cases. If $c(x) = c(y)$, then nothing needs to be done. Otherwise, if $c(x) \neq c(y)$ then x and y are not connected yet and the (old) equivalence classes $c(x)$ and $c(y)$ need to be joined. This is performed by $Union_c(c(x), c(y))$. Since apart from these Unions each insertion takes $O(1)$ time, it follows that all insertions and queries can be performed in $O(|E|)$ time plus the time need for the Union and Find operations. In the sequel, we use this algorithm for maintaining connectivity.

For maintaining 3-edge-connected we also need a structure for a problem that is closely related to the Split-Find problem [5, 13]: the *Circular Split-Find problem* [13], which is given as follows. Let U be a collection of nodes, called elements. Suppose U is partitioned into a collection of cyclic lists and suppose to each list a (new) unique node is related, called set name. We want to be able to perform the following operations: Find(x) and Split(x, y) (where x and y are in the same list and $x \neq y$), i.e., given (pointers to) elements x and y, split the cyclic list that contains x and y into two cyclic lists, viz. the part starting from x up to but excluding y and the part from y up to but excluding x and relate set names to the two newly arisen cyclic lists. The occurring set names must satisfy the condition that, at every moment, the names of the existing cyclic lists are distinct. A solution for the Split-Find problem is as follows: at any moment, each cyclic list is implemented as a

doubly linked cyclic list and each element has a pointer to its set name. Hence, a Find can be performed in $O(1)$ time. A Split(x, y) is performed as follows: first split the list at these two points into the two sublists as described above (which can be done in $O(1)$ time since the lists are doubly linked). Then determine the smallest of these lists as follows: traverse both lists by performing a step of each traversal in an alternating way, until one of the traversals has been completed: that lists is the smallest list. (Note that this takes time linear to the size of the smallest of the two resulting lists.) Finally, for all nodes in the smallest list, adapt the pointer to point to a new set name. It is easily seen that all Splits take $O(n.\log n)$ time altogether, since a Split takes time proportional to the size of the smallest resulting list (also cf. the Union-Find algorithms in [1]). In [13] faster solutions for the Circular Split-Find problem are given which are optimal on pointer machines [12]. These solutions closely correspond to the solutions in [5] for the ordinary Split-Find problem. The solutions take $O(n + m.\alpha(m, n))$ time for all Circular Splits and m Finds on n elements.

3 Two-edge-connectivity

3.1 Graph observations

Let $G =< V, E >$ be a graph. The set V can be partitioned into 2-edge-connected equivalence classes. Let each 2-edge-connected equivalence class C be represented by a new (distinct) node c, called the *class node* of C. Let $2ec(x)$ be the class node of the 2-edge-connected class in which the node x is contained. We define the graph $2ec(G)$ as follows (according to Definition 2.5):

$$2ec(G) =< 2ec(V), \{(e, 2ec(x), 2ec(y)) | (e, x, y) \in E \wedge 2ec(x) \neq 2ec(y)\} > .$$

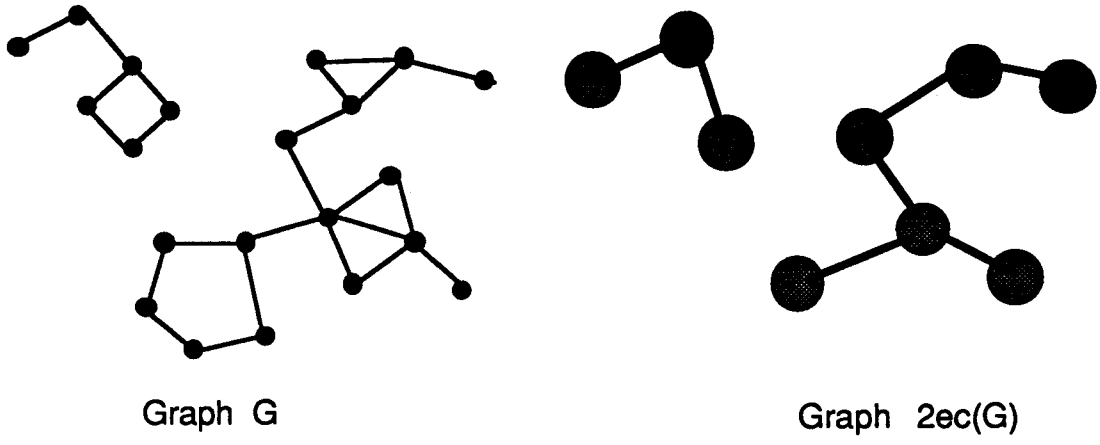
Hence, $2ec(G)$ is the graph that is obtained if we contract each 2-edge-connected component into one (representing) class node. Since $2ec(V)$ represents the set of equivalence classes of G, it follows by Lemma 2.6 (sub 3) that $2ec(G)$ is a forest (cf. Figure 2). We maintain the 2-edge-connectivity relation under edge insertions by means of the graph $2ec(G)$.

Edge insertions can be handled as follows. Suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G =< V, E >$. We distinguish three cases.

1. $c(x) \neq c(y)$. Then by Lemma 2.6 (sub 3) $2ec(x)$ and $2ec(y)$ are not connected in $2ec(G)$. Hence, $(e, 2ec(x), 2ec(y))$ connects two trees in $2ec(G)$ that have to be joined into one tree.

2. $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$. Then the edge $(e, 2ec(x), 2ec(y))$ arises as an inserted edge in $2ec(G)$. Edge $(e, 2ec(x), 2ec(y))$ connects the class nodes

Figure 2: Graph G and the corresponding graph $2ec(G)$.



$2ec(x)$ and $2ec(y)$ in a tree of $2ec(G)$ and a cycle arises. Hence, all class nodes on the tree path from $2ec(x)$ to $2ec(y)$ become 2-edge-connected in $2ec(G)$. By Lemma 2.6 (sub 3) all nodes in V that are contained in the corresponding classes become 2-edge-connected too. The update can now be performed in the following way.

- obtain the tree path in $2ec(G)$ between $2ec(x)$ and $2ec(y)$.
- join all the classes "on" this tree path into one new class C' and adapt the related information.

3. $2ec(x) = 2ec(y) \wedge c(x) = c(y)$. Then the edge (e, x, y) connects two nodes that are 2-edge-connected in G, and, hence, insertion of this node will not affect the 2-edge-connectivity relation (cf. Lemma 2.6, sub 3).

3.2 The algorithms

We will now describe the different steps in more detail.

In our algorithms we represent each of the collections of connected classes and 2-edge-connected classes of a graph G by a Union-Find structure (cf. Subsection 2.2), where the name of each class is the class node of that class (i.e., a Find on an element of a class outputs the class node related to that class). Therefore we (may) denote $Find_c(x)$ or $Find_{2ec}(x)$ by $c(x)$ or $2ec(x)$ too (cf. Subsection 2.2). We represent the forest $2ec(G)$ by means of rooted trees in our algorithms. We denote a rooted forest by $2ec(G)^R$ without making the roots explicit in our description.

For each class node c we have a field $father(c)$ that is *nil* or that contains a pointer to the edge (e, x, y) such that $2ec(x) = c$ (i.e., x is contained in class c) and $2ec(y)$ is the father of $2ec(x)$ in the (rooted) forest $2ec(G)^R$. Edge (e, x, y) is called the *interconnection edge* between (classes) $2ec(x)$ and $2ec(y)$, or it is called the *father edge* of (class) $2ec(x)$. (Note that the father of $2ec(x)$ in $2ec(G)^R$ can be obtained by the father edge of $2ec(x)$.)

Initially, there are no edges, each node forms both a connected class and a 2-edge-connected class and for all class nodes c, $father(c) = nil$.

Now, suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G =< V, E >$. Then after inserting edge (e, x, y) in the proper adjacency lists, procedure $insert_2$ given in Figure 4 updates the structure as follows. (The sub-procedures of $insert_2$ are given in Figure 5 and Figure 6.) We distinguish the three previous cases.

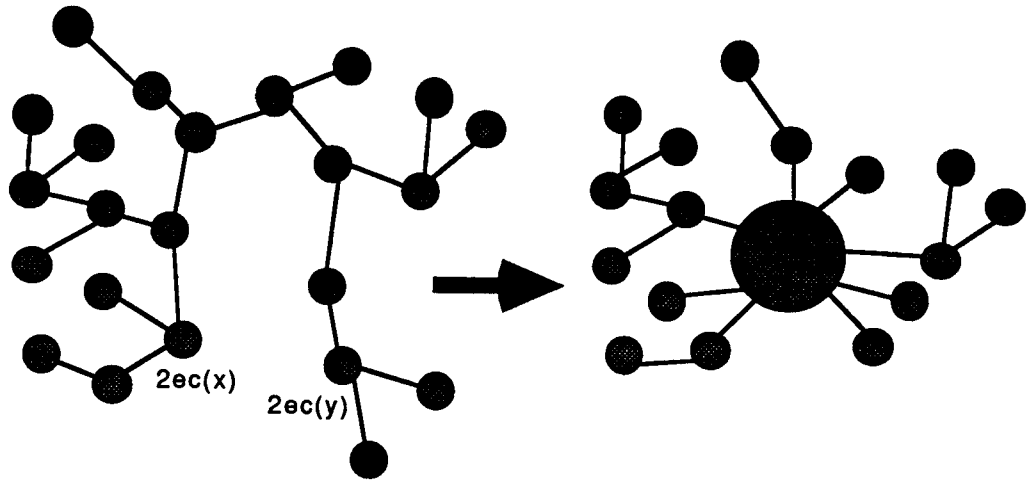
1. $c(x) \neq c(y)$ (line 2-8). Then $(e, 2ec(x), 2ec(y))$ connects two trees in $2ec(G)$ that have to be joined to one tree. Since the trees are represented by rooted trees this means that one of the two trees has to be redirected w.r.t. the father relation of classes. We take the tree with the smallest size, i.e., the tree that has the least number of nodes that are contained in the classes in that tree. (This can be determined by means of a parameter in the Union-Find structure for connected components.) W.l.o.g. this is the tree containing $2ec(y)$. It suffices to "reverse" the father pointers for the nodes on the root path of the class node $2ec(y)$ (i.e., the path from node $2ec(y)$ to the root of its tree). This is performed by procedure *ReverseRootPath* that is given in Figure 5.

2. $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$ (line 9-12). All classes on the tree path in $2ec(G)$ between $2ec(x)$ to $2ec(y)$ become 2-edge-connected and must be joined. This is done as follows. First of all, the tree path P between $2ec(x)$ and $2ec(y)$ is obtained by means of procedure call $TreePath_2$ that outputs tree path P together with a pointer $fath$ to the father edge of the nearest common ancestor *top* of $2ec(x)$ and $2ec(y)$ in $2ec(G)^R$ (this pointer is *nil* if this edge does not exist). These are obtained by stepwise traversing the root paths from $2ec(x)$ and $2ec(y)$ in an alternating way (cf. Figure 6) until a node *top* has been visited by both traversals. This class node *top* is the nearest common ancestor of $2ec(x)$ and $2ec(y)$. Then the path between $2ec(x)$ and $2ec(y)$ consists of the two parts of these root paths up to and including this "first mutual class node".

   The joining of the classes on P is done by means of Union_{2ec} operations. Note that the father edge of the resulting class is the father edge of the (old) class *top*. (Cf. Figure 3.)

3. $2ec(x) = 2ec(y) \wedge c(x) = c(y)$ (line 13-14). Then nothing needs to be done.

Figure 3: Joining the classes of the tree path from $2ec(x)$ to $2ec(y)$.



For the Union-Find structures we take the basic Union-Find structure that takes $O(n.\log n)$ time for all Unions on n elements and $O(1)$ time per Find.

3.3 Time bounds

We consider the time complexity of the algorithm. All insert operations can be performed in $O(n \log n+e)$ time for e edge insertions together (where n is the number of nodes). This is seen as follows. All redirections of trees are performed in the basic Union-Find way, i.e., always only the *father* values in the smallest tree are adapted. Since the redirection of a tree of size *size* is performed in $O(size)$ time and since after the linking the resulting tree has to be at least twice as large as the smallest of the previous two trees, the total time for all these adaptations is $O(n \log n)$. Furthermore, all Unions take $O(n \log n)$ time altogether too. A computation of a tree path P (line 10) is done in $O(|P|)$ time, since the traversed part P_1 of one of the two root paths contains class nodes of P only, while the traversed part P_2 of the other root path contains at most as many class nodes as P_1 : hence at most $2.|P|$ class nodes are encountered in these traversals. Since the number of classes decreases by $|P|-1$ (>0), since initially there are n classes and since the number of classes never increases, all tree path computations take $O(n)$ time altogether. Finally, each insertion takes $O(1)$ time apart from the cost considered above.

Combining the above time bounds yields that all e insertions take altogether $O(n \log n+e)$ time.

We consider the space complexity. Note that all edges that do not become intercon-

Figure 4: Procedure $insert_2(e, x, y)$.

```
(1)  procedure insert_2(e, x, y);
(2)  if c(x) ≠ c(y)
(3)     ⟶ if size(c(x)) ≥ size(c(y))
(4)          ⟶ ReverseRootPath(2ec(y)); father(2ec(y)) := (e, x, y)
(5)       [] size(c(x)) < size(c(y))
(6)          ⟶ ReverseRootPath(2ec(x)); father(2ec(x)) := (e, x, y)
(7)       fi;
(8)       Union_c(c(x), c(y))
(9)  [] c(x) = c(y) ∧ 2ec(x) ≠ 2ec(y)
(10)    ⟶ Treepath(2ec(x), 2ec(y), P, fath);
(11)      for all C ∈ P\{2ec(x)} ⟶ Union_2ec(C, 2ec(x)) rof;
(12)      father(2ec(x)) := fath
(13) [] c(x) = c(y) ∧ 2ec(x) = 2ec(y)
(14)    ⟶ skip
(15) fi
```

Figure 5: Procedure $ReverseRootPath(C)$.

```
(1)  procedure ReverseRootPath(C);
(2)  if father(C) ≠ nil
(3)     ⟶ (e, u, v) := father(C); father(C) := nil;
(4)       w.l.o.g., 2ec(u) = C ∧ 2ec(v) ≠ C (otherwise, interchange u and v);
(5)       ReverseRootPath(2ec(v)) ;
(6)       father(2ec(v)) := (e, u, v)
(7)  [] father(C) = nil
(8)     ⟶ skip
(9)  fi
```

Figure 6: Procedure $TreePath_2(C, D,$ **output** $P, fath)$.

---

procedure $TreePath_2(C, D,$ **output** $P, fath)$;

- stepwise traverse the root paths from C and D alternatively, i.e., by performing steps of the traversals of these root paths in an alternating way. During this traversals, mark the class nodes encountered and stop the traversals if one of the two path traversals encounters a class node *top* that has been marked by the other traversal;
- path P between C and D consists of the two parts of these root paths up to and including *top*;
- $fath := father(top)$;
- remove the marks

---

nection edge at the moment of insertion, are not used by the algorithm and hence do not need to be stored in memory. We show that there exist at most $n-1$ interconnection edges during all edge insertions. An edge that is inserted becomes an interconnection edge if its end nodes are in two distinct connected components just before its insertion, while these connected components are joined. Since initially (in the "empty" graph) there are n connected components, at most $n-1$ joinings of components occur and hence there exist at most $n-1$ such edges during the entire sequence of insertions. Therefore, it follows that the space complexity is $O(n)$.

A query whether two nodes are in the same 2-edge-connected class is simply done by performing $Find_{2ec}$ queries on these nodes, which takes $O(1)$ time.

3.4 Maintaining components

Although the above method maintains the 2-edge-connectivity relation, it does not actually maintain the components themselves. If the 2-edge-connected components have to be maintained, then this can be done as follows: for each node, we have two incidence list, viz. the list $list2$ containing the edges within its 2-edge-connected component and a list $list1$ containing the edges outside of it. Each edge not only contains pointers to its end nodes, but to its occurrences in the incidence lists as well. The incidence lists are doubly linked. Since all edges that are incident with node x and that are within its 2-edge-connected component are stored in list $list2$, a 2-edge-connected component can be traversed by means of these lists. (Moreover, all

nodes that are in this component can be enumerated by means of a list in the Union-Find structure that represents the set corresponding to the class node $2ec(x)$.) Now again consider the insertion of the new edge (e, x, y). In addition to the previous algorithm the following steps must be performed:

- $c(x) \neq c(y)$. Then insert the edge in the lists $list1$ of the nodes.
- $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. Then all interconnection edges encountered on the tree path of $2ec(x)$ and $2ec(y)$ become edges inside a 2-edge-connected component instead of outside it. Therefore, remove all these edge from the related lists $list1$ and insert them in the lists $list2$ of their end nodes. This can be done in $O(1)$ time per edge. Moreover, edge (e, x, y) is inserted in the lists $list2$ of the nodes x and y.
- $2ec(x) = 2ec(y)$. Then insert the new edge in the lists $list2$ of nodes x and y.

Since the additional operations increase the time with only $O(1)$ time per encountered edge, it follows that this does not increase the time complexity in order of magnitude.

Theorem 3.1 *Given a graph G, there exists a data structure such that the query whether two nodes are 2-edge-connected can be answered in $O(1)$ time and that maintains the 2-edge-connected components of G when edges are inserted. Starting from the empty graph $G =< V, \emptyset >$ (i.e., a graph with no edges), the insertion of e edges take $O(n \log n + e)$ time altogether, if n is the number of nodes in G. Finally, the data structure can be initialised in $O(n)$ time and it uses $O(n + e)$ space when the 2-edge-connected components are maintained and $O(n)$ space otherwise.*

It is easily seen that besides edges, new nodes can be inserted in the graph in $O(1)$ time (where each inserted node forms a 2-edge-connected class in its own at the moment of insertion). Therefore, the statement in the above theorem can be extended with node insertions, where n is the final number of nodes in the graph.

4 Three-edge-connectivity

We will now extend the results to the maintenance of 3-edge-connected components in a graph. We first introduce some notions and prove some properties for them. In subsection 4.1 we consider maintaining the 3-edge-connectivity relation within 2-edge-connected graphs and subsequently in Subsection 4.2 we consider the problem for general graphs. In Subsection 4.3 we consider the maintenance of complete 3-edge-connected components.

Let $G =< V, E >$ be a graph. The set V can be partitioned into 3-edge-connected equivalence classes. Each 3-edge-connected class C is represented by a new (distinct) node c, called the *class node* of C. Let $3ec(x)$ be the class node of the 3-edge-connected class in which the vertex x is contained. We define the graph $3ec(G)$ as follows:

$$3ec(G) =< 3ec(V), \{(e, 3ec(x), 3ec(y))|(e, x, y) \in E \wedge 3ec(x) \neq 3ec(y)\} > .$$

Hence, $3ec(G)$ is the graph that is obtained if we contract each 3-edge-connected component into one representing (class) node (see Figure 7 if G is 2-edge-connected). By Lemma 2.6 (sub 3) it follows that $3ec(G)$ does not contain pairs of distinct class nodes that are 3-edge-connected in $3ec(G)$.

4.1 Two-edge-connected graphs

Throughout this subsection, we suppose that the graph G is 2-edge-connected. By Lemma 2.6 (sub 3) for 2-edge-connectivity, every two distinct class nodes must lie on a common elementary cycle in $3ec(G)$. On the other hand, simple cycles cannot intersect in more than one class node, since $3ec(G)$ does not contain pairs of distinct class nodes that are 3-edge-connected. (The proof is similar to the proof of Lemma 2.9. If two different simple cycles S_1 and S_2 intersect in two different nodes, then take a maximal part P of cycle S_1 that consists of at least three nodes and that has no nodes in common with S_2 except for both its end nodes. Then P and the two paths between these nodes in S_2 yield 3 edge-disjoint paths.) Therefore, it follows that each edge in $3ec(G)$ is on exactly one simple cycle in $3ec(G)$.

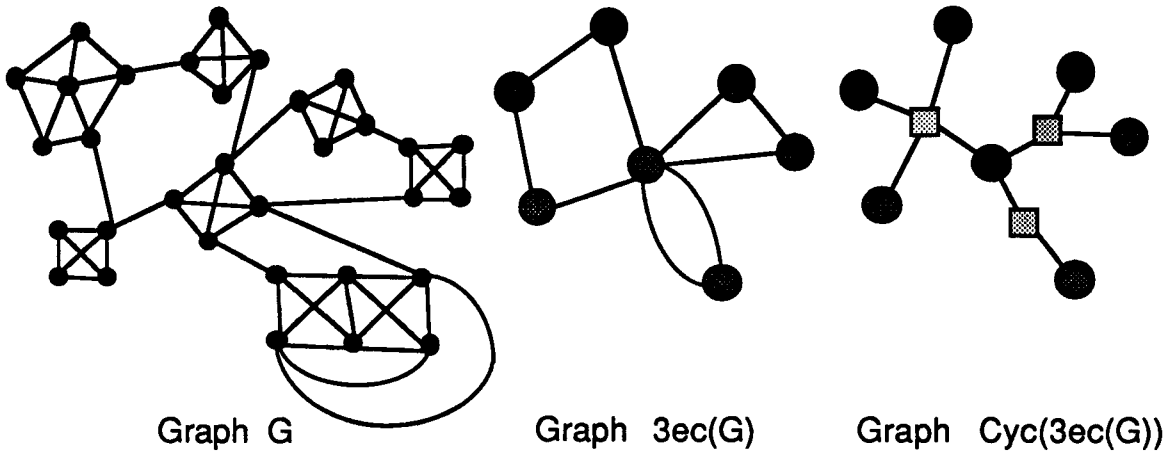
Let $Cyc(3ec(G))$ be the graph that is constructed from $3ec(G)$ as follows. Each non-trivial simple cycle (i.e., consisting of at least two distinct class nodes) is represented by a distinct node, called *cycle node*. Let $cn(3ec(G))$ be the set of cycle nodes. For a cycle node s let $cycle(s)$ be the set of all class nodes that are on the cycle s. Then the graph $Cyc(3ec(G))$ is defined uniquely up to the choice of (distinct) edge names by

$$Cyc(3ec(G)) =$$
$$< 3ec(V) \cup cn(3ec(G)), \{(e, c, s)|c \in 3ec(G) \wedge s \in cn(3ec(G)) \wedge c \in cycle(s)\} > .$$

Hence, $Cyc(3ec(G))$ consists of the class nodes and cycle nodes of $3ec(G)$, where a class node c is adjacent to a cycle node s in $Cyc(3ec(G))$ iff c lies on cycle s in $3ec(G)$ (i.e., c is "incident" with cycle s). Therefore, graph $Cyc(3ec(G))$ shows the incidence relation for class nodes and cycles. The structure of $Cyc(3ec(G))$ is illustrated in Figure 7, where the cycle nodes are drawn as boxes.

Below we will show that $Cyc(3ec(G))$ is a tree. Therefore we call graph $Cyc(3ec(G))$ the *cycle tree* of G.

Figure 7: A 2-edge-connected graph G and the related graphs $3ec(G)$ and $Cyc(3ec(G))$.



Lemma 4.1 *Let G be a 2-edge-connected graph. Let $c, d \in 3ec(G)$. Let P be a path between c and d in $Cyc(3ec(G))$. Then there are 2 edge disjoint paths in $3ec(G)$ between c and d that only consist of edges from the cycles represented by the cycle nodes on P.*

Proof. Between any two distinct class nodes on a simple cycle, there are precisely two edge disjoint paths within that cycle. On the other hand, each edge is contained in exactly one simple cycle. Now the lemma easily follows. □

Lemma 4.2 *Let G be a 2-edge-connected graph. Then $Cyc(3ec(G))$ is a tree.*

Proof. Let c and d be two class nodes in $Cyc(3ec(G))$. By Lemma 2.6 (sub 3) for connectivity, graph $3ec(G)$ is connected. Hence, there is a simple path P in $3ec(G)$ between class nodes c and d. We can construct a path in $Cyc(3ec(G))$ between c and d by the observation that each edge (e, f, g) on P is in some simple cycle s and hence that there are edges between f and s and between g and s in $Cyc(3ec(G))$. Hence, all class nodes are connected in $Cyc(3ec(G))$. On the other hand, each cycle node is adjacent to at least one class node. Hence, $Cyc(3ec(G))$ is connected.

On the other hand, suppose there is a nontrivial simple cycle in $Cyc(3ec(G))$. Hence, it consists of at least distinct 2 class nodes c and d and at least 2 cycle nodes. Lemma 4.1 yields that there are at least 4 edge disjoint paths between c and d in $3ec(G)$, since an edge in $3ec(G)$ is contained in precisely one simple cycle. Hence, by Lemma 2.2 c and d are 3-edge-connected in $3ec(G)$. Contradiction by Lemma 2.6 (sub 3). □

4.1.1 Graph observations

We maintain the 3-edge-connectivity relation under insertions of edges by means of the graph $Cyc(3ec(G))$.

Suppose a new edge (e, x, y) is inserted in the 2-edge-connected graph $G =< V, E >$ ($(e, x, y) \notin E$). Because G is 2-edge-connected, we have two cases. If $3ec(x) = 3ec(y)$ then the edge connects two nodes that are 3-edge-connected in G, and, hence (by Lemma 2.6, sub 3), insertion of this edge does not affect the $3ec$-relation and the graphs $3ec(G)$ and $Cyc(3ec(G))$ remain unchanged. So we can assume that $3ec(x) \neq 3ec(y) \wedge 2ec(x) = 2ec(y)$. Then edge $(e, 3ec(x), 3ec(y))$ arises as an inserted edge in $3ec(G)$ and it connects two class nodes $3ec(x)$ and $3ec(y)$ in $3ec(G)$.

Lemma 4.3 *Let G be a 2-edge-connected graph. Suppose edge $(e, 3ec(x), 3ec(y))$ is inserted to the graph $3ec(G)$. Then all the class nodes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ become 3-edge-connected in $3ec(G)$, while the other pairs of distinct class nodes in $3ec(G)$ stay only 2-edge-connected.*

Proof. Let P be the tree path in $Cyc(3ec(G))$ between the class nodes $3ec(x)$ and $3ec(y)$. Let c and d be any two class nodes on P. Now split P into 3 disjoint parts: part P_1 from $3ec(x)$ to c, part P_2 from c to d and part P_3 from d to $3ec(y)$. By Lemma 4.1 it follows that there exist 2 edge disjoint paths Q_1 and Q_2 in $3ec(G)$ between c and d that only consist of edges from cycles represented by cycle nodes on P_2. Similarly, it follows from Lemma 4.1 that there exists a path R_1 from c to $3ec(x)$ that only uses edges from the cycles represented by cycle nodes on P_1, and a path R_2 from $3ec(y)$ to d only using edges from cycles represented by cycle nodes on P_3. Let Q_3 be the path $R_1, (e, 3ec(x), 3ec(y)), R_2$ from c to d. Then it follows that Q_3 has no edges in common with Q_1 and Q_2. Hence, by Lemma 2.2 c and d are 3-edge-connected.

On the other hand, let c and d be 2 distinct class nodes in $3ec(G)$ such that c is not on P. Consider a cycle node r that is adjacent to class node c in $Cyc(3ec(G))$ such that r separates c from $3ec(x)$ and $3ec(y)$. (This node exists because r is not on P.) The deletion of the two edges in $3ec(G)$ that are incident with c and that belong to the simple cycle r, separates c from all class nodes on the other side of r in the tree $Cyc(3ec(G))$. (For, otherwise there would be a distinct class node on cycle r that was 3-edge-connected with c.) Hence, the removal of these edges either separates c from both d, $3ec(x)$ and $3ec(y)$, or c and d are "on the same side of r" in $Cyc(3ec(G))$. In the first case it follows that insertion of $(e, 3ec(x), 3ec(y))$ does not make c and d 3-edge-connected, in the latter case we can make the same construction for d, yielding the required result. □

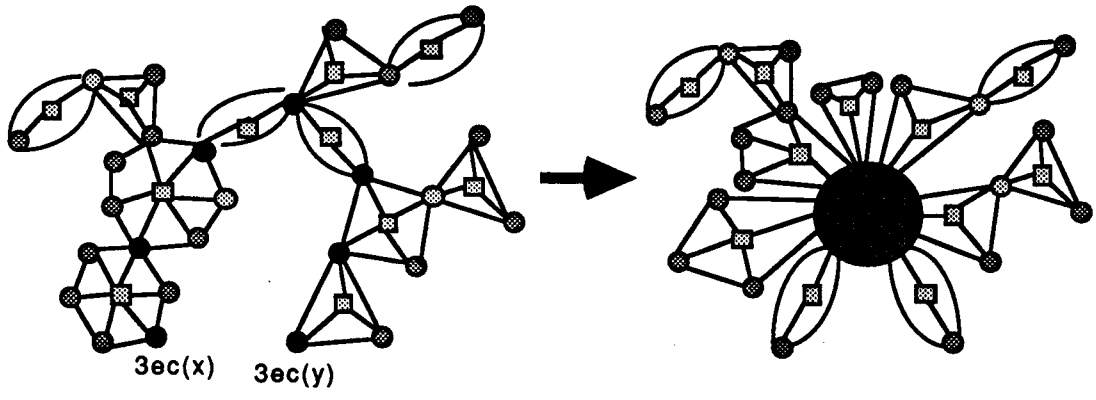
By Lemma 4.3 all class nodes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ become 3-edge-connected in $3ec(G)$ and, hence, by Lemma 2.6 (sub 3) all the

corresponding classes form a new class. The update can now be performed in the following way.

- obtain the tree path in $Cyc(3ec(G))$ between $3ec(x)$ and $3ec(y)$
- join all the classes "on" this tree path into one new class C' and adapt the cycle tree $Cyc(3ec(G))$ into $Cyc(3ec(G'))$ accordingly (where G' is the result graph after the insertion of the edge).

The update is illustrated in Figure 8. The cycle tree changes as follows. Consider

Figure 8: Adapting the tree path between $3ec(x)$ and $3ec(y)$.

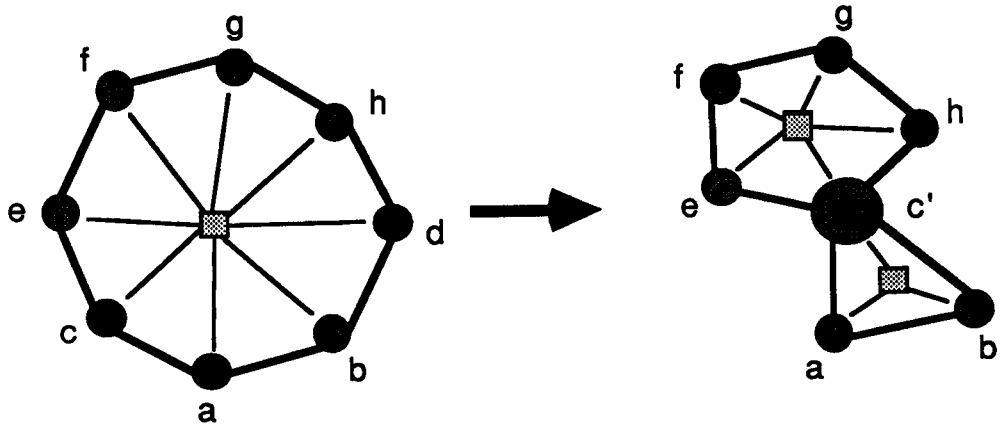


the simple cycle s and the class nodes c and d ($c \neq d$) such that s, c and d are on P and $c, d \in cycle(s)$. Then classes c and d are joined into the new class c'. The original simple cycle s splits into two "smaller" simple cycles, each one consisting of the class node c' for the new class and of the class nodes of one of the two parts of the cycle *between* c and d, in the same cyclic order (cf. Figure 9). One or both of these two new cycles may be a trivial cycle: i.e., consisting of class node c' only (which is the case if one of the parts mentioned above of the cycle is empty).

4.1.2 The algorithms

In our algorithms we represent the collection of 3-edge-connected classes of a graph G by a Union-Find structure (cf. Subsection 2.2), where the name of each class is the class node of that class (i.e., a Find on an element of a class returns the class node related to that class). Therefore we (may) denote $Find_{3ec}(x)$ by $3ec(x)$ too (cf. Subsection 2.2).

Figure 9: Splitting cycles.



We represent the cycle tree $Cyc(3ec(G))$ as a rooted tree in a way which will be described in the sequel, where the root of the tree is some *class node*. We will denote this rooted tree by $Cyc(3ec(G))^R$ in the descriptions below without making the root explicit.

The edges in $Cyc(3ec(G))$ are represented as follows. The data structure is extended with a (variable) collection of *class representatives*, which are new records. Each class representative represents some edge in $Cyc(3ec(G))$ between a class node and a cycle node. (If a cycle tree changes because of an edge insertion, a class representative may represent another edge of the resulting cycle tree.) We *denote* the class representative that is related to the edge between class node c and cycle node s (in $Cyc(3ec(G))$) by $repr(c, s)$.

To implement the relation between a class representative $repr(c, s)$ and the corresponding edge between c and s in $Cyc(3ec(G))$, we use a Circular Split-Find and a Union Find structure, from which the end nodes c and s of that edge can be obtained. (Hence, in contrary to the representation of ordinary edges in the graph G, a class representative $repr(c, s)$ that represents an edge between c and s in $Cyc(3ec(G))$ does not have direct pointers to the end nodes c and s of that edge.) These structures are used in the following way. A class representative $repr(c, s)$ is an element of the so-called *cycle list* for cycle node s and of the so-called *representative set* for class node c, which are given as follows. The *cycle list for a cycle node* s contains the class representatives $repr(c, s)$ of all class nodes c in cycle s in $3ec(G)$ in the order in which these class nodes occur in cycle s. The collection of cycle lists is implemented as a Circular Split-Find structure (cf. Subsection 2.2), where the name of a cycle list for cycle node s is s itself. (Hence, a Find on an element of that list

returns node s.) We denote a Circular Split or a Find in this structure by $Split_{cyc}$ or $Find_{cyc}$ respectively. The *representative set for a class node* c is the set that contains the representatives $repr(c, s)$ for all cycle nodes s for which $repr(c, s)$ exists. The collection of representative sets is implemented as a Union-Find structure, where the name of the representative set for class node c is c itself. (Hence, a Find on an element of that set returns the node c.) In the algorithms we perform a Union on two representative sets (of class representatives) for two class nodes c and d iff the corresponding classes c and d (of ordinary nodes) in the graph are joined. Therefore we will not make these joinings explicit in our algorithms. We denote a Find in this structure by $Find_{class}$. Hence, the operations $Find_{class}$ and $Find_{cyc}$ on a class representative yield the end nodes of the edge that is related to it.

The father relation in $Cyc(3ec(G))^R$ is implemented as follows. If h is the father of g in $Cyc(3ec(G))^R$, then $father(g)$ is a pointer to the class representative $repr(g, h)$ or $repr(h, g)$, depending on which of the nodes g or h is the class node. Then the father of g in $Cyc(3ec(G))^R$ can be obtained by means of $Find_{cyc}(father(g))$ or $Find_{class}(father(g))$ respectively.

We assume that initially the 2-edge-connected graph G is represented as described above, where the father relation satisfies the orientation in $Cyc(3ec(G))^R$ for some root.

Now, edge insertions can be handled as follows. Suppose edge (e, x, y) is inserted in graph $G =< V, E >$ with $(e, x, y) \notin E$. Then after inserting this edge in the incidence lists, procedure $insert_3$ (given in Figure 10) performs the updates as follows. We distinguish the two cases. If $3ec(x) = 3ec(y) \wedge 2ec(x) = 2ec(y)$ (line 2), then nothing needs to be done. Otherwise, we have $3ec(x) \neq 3ec(y) \wedge 2ec(x) = 2ec(y)$ (line 3-7). All class nodes on the tree path in $Cyc(3ec(G))$ between $3ec(x)$ and $3ec(y)$ become 3-edge-connected in $3ec(G)$. The procedure first determines this tree path (line 4) and then adapts the cycle tree accordingly by first splitting all cycles on P (line 5) and then joining all classes on P (line 6). This is done as follows.

1. *The computation of the tree path (line 4).* In line 4, the tree path P between $3ec(x)$ and $3ec(y)$ is obtained by traversing the root paths of $3ec(x)$ and $3ec(y)$ alternatively like in Section 2. This is performed by the call of procedure $TreePath_3$ which is given in Figure 11. This procedure returns the tree path P. Moreover, it detects whether the nearest common ancestor *top* of $3ec(x)$ and $3ec(y)$ in $Cyc(3ec(G))^R$ is a cycle node (if this is the case, $topcyc = true$ is returned) and it returns the class representative $father(top)$ in the parameter *toprepr*.

2. *The splitting of cycles* (line 5) is performed by procedure *AdjustCycles*, which is given in Figure 12. The strategy is as follows: let c, s and d be three consecutive nodes on P,where s is a cycle node. Note that, since $Cyc(3ec(G))^R$ is a rooted tree, either c or s contains a pointer to $repr(c, s)$ and that the same

holds for d, s and $repr(d, s)$. Therefore, these records can be obtained by using these pointers. The cycle list for cycle node s is split into two parts: the part from $repr(c, s)$ up to but excluding $repr(d, s)$ and the part from $repr(d, s)$ up to but excluding $repr(c, s)$. This is done by a Circular Split operation at those two elements. (Each part forms a new cyclic list, for which a new cycle node is generated.) If one (or both) of these two lists appears to correspond to a trivial cycle (i.e., it contains only one element), then that list is deleted. Note that if s' and s'' are the cycle nodes resulting from the Circular Split, then the class representatives denoted by $repr(c, s')$ and $repr(d, s'')$ (after the Circular Split) actually are the class representatives formerly denoted by $repr(c, s)$ and $repr(d, s)$. Each resulting cycle node s' or s'' (which can be obtained by the Find_{cyc} operation) gets a father pointer to $repr(c, s')$ or $repr(d, s'')$ respectively.

3. The *joining of the classes* on P is done by joining the classes pairwise, resulting in a new class c' (line 6). Note that afterwards all cycle nodes s' that have resulted from the previous Circular Splits, now have a father pointer to the class representative denoted by $repr(c', s')$.

4. Finally, the $father(c')$ value for the newly formed class c' is assigned by procedure $AdjustFathers$ (line 7). Procedure $AdjustFathers$ is given in Figure 13. The father values are updated according to the following observations.

   Consider the old graph $Cyc(3ec(G))^R$. Let *top* be the nearest common ancestor of $3ec(x)$ and $3ec(y)$ in $Cyc(3ec(G))^R$. Recall that *toprepr* is the class representative corresponding to the edge between *top* and its father in $Cyc(3ec(G))$ (if any). We have the following cases:

   - *top* is a class node that is the root. Then the new class node c' will be the root of the new tree.
   - *top* is a class node that is not the root. Then the father of *top* in $Cyc(3ec(G))$ is a cycle node s for which no Circular Split is performed on its cycle list. Then s must be the father of the new class c'.
   - *top* is a cycle node (that is not the root). Then let a be the class node that is the father of *top* in $Cyc(3ec(G))^R$. Note that the father of the new class c' must be the cycle node s that contains both a and c': this cycle node s may be different from *top* since a Circular Split just generates two new cyclic lists with two names (being the resulting cycle nodes). Hence, the father of c' is s and the father of s is a. Note that s can be obtained by $Find_{cyc}(toprepr)$. Then $repr(c', s)$ can be obtained by $father(s)$, since all involved cycle nodes have their father pointers to the representative of class c'. (See Figure 14 that shows the results of the four successive parts as distinguished above for this case (*top* is a cycle node), where a class representative together with the father pointer to that class representative is indicated as a directed edge as follows: e.g.,

if the father pointer of node t points to $repr(d,t)$, then this is indicated as the directed edge from t to d).

Hence, c' becomes father of all the cycle nodes that are on P or that are created in the cycle splittings, except for the cycle node s of the third case given above.

Figure 10: Procedure $insert_3((e,x,y))$.

---

(1) **procedure** $insert_3((e,x,y))$;
(2) **if** $3ec(x) = 3ec(y) \longrightarrow$ skip
(3) [] $3ec(x) \neq 3ec(y)$
(4) $\longrightarrow TreePath_3(3ec(x), 3ec(y), P, toprepr, topcyc)$;
(5) $AdjustCycles(P)$;
(6) **for all** *class nodes* $c \in P \backslash \{3ec(x)\} \longrightarrow Union_{3ec}(c, 3ec(x))$ **rof**;
(7) $AdjustFathers(3ec(x), toprepr, topcyc)$
(8) **fi**

Figure 11: Procedure $TreePath_3(c,d,$ **output** $P, top)$.

---

procedure $TreePath_3(c,d,$ **output** $P, toprepr, topcyc)$;

- traverse the root paths from c and d alternatively, i.e., by performing steps of the traversals of these root paths in an alternating way. During this traversals, mark the nodes encountered and stop the traversals if one of the two path traversals encounters a node *top* that has been marked by the other traversal;
- path P between c and d consists of the two parts of these root paths up to and including *top*;
- $topcyc$:=[*top* is a cycle node];
  $toprepr := father(top)$;
- remove the marks

Figure 12: Procedure $AdjustCycles(P)$.

---

procedure $AdjustCycles(P)$;
traverse P and for all three consecutive nodes c,s, d on the path where s is a cycle node, perform the following.

- obtain the class representatives $repr(c,s)$ and $repr(d,s)$ by means of the fields $father(c)$, $father(d)$ and $father(s)$.
- split the cycle list for cycle node s into two parts by a Circular Split $Split_{cyc}(repr(c,s), repr(d,s))$: the part from $repr(c,s)$ up to but excluding $repr(d,s)$ and the remainder, (while (new) cycle nodes are related to these cycles as names); dispose such a cycle list if it contains only one element
- as far as the considered class representatives $repr(c,s)$ and $repr(d,s)$ are not disposed:

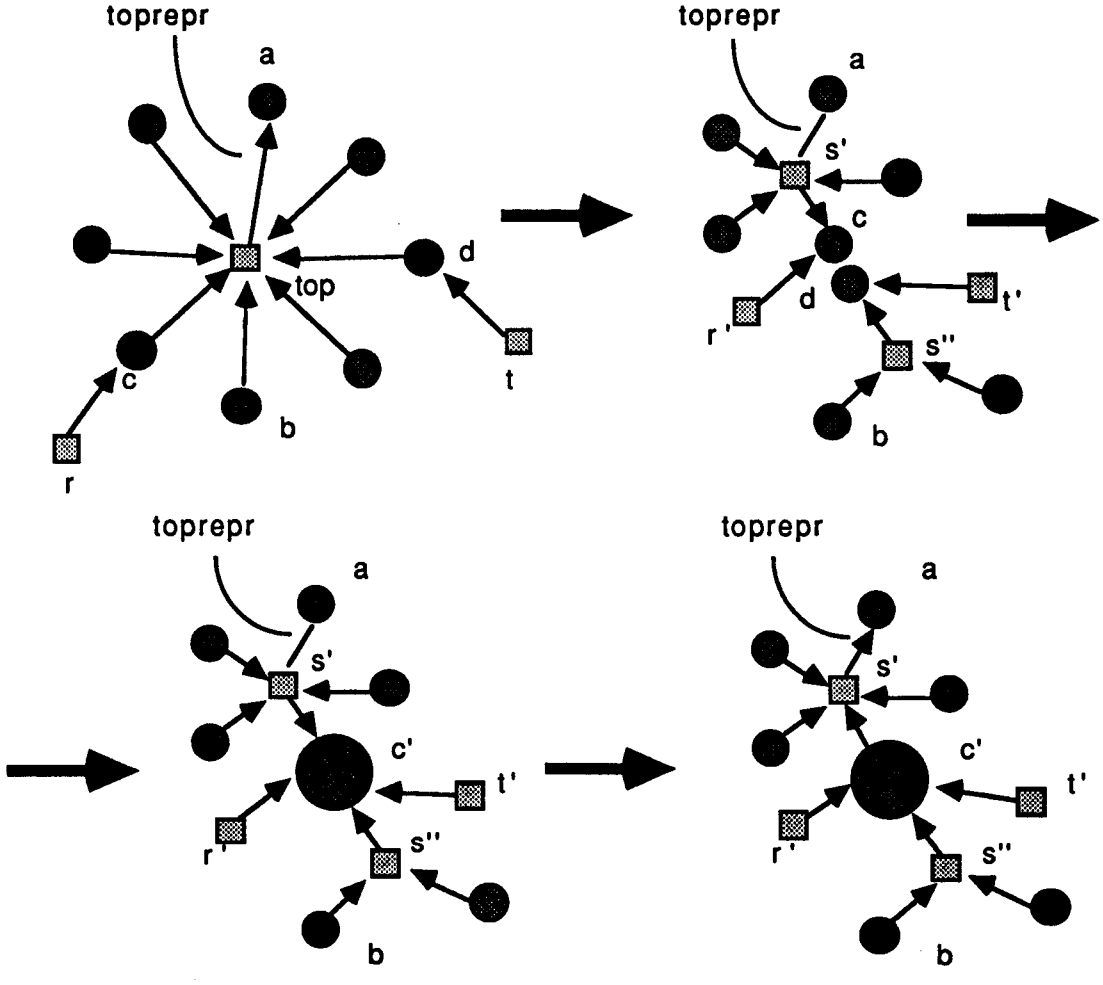
$$father(Find_{cyc}(repr(c,s))) := repr(c,s);$$
$$father(Find_{cyc}(repr(d,s))) := repr(d,s);$$

Figure 13: Procedure $AdjustFathers(3ec(x), toprepr, topcyc)$.

---

procedure $AdjustFathers(3ec(x), toprepr, topcyc)$;
(1) **if** $topcyc \longrightarrow$ $s := Find_{cyc}(toprepr)$;
(2) $father(3ec(x)) := father(s)$; $father(s) := toprepr$
(3) $[]$ $\neg topcyc \longrightarrow$ $father(3ec(x)) := toprepr$
(4) **fi**

Figure 14: Changes in the father relation.



4.1.3 Complexity

We study the time complexity of the method. We express the time complexity of an execution of procedure $insert_3$ in the number of computational steps that are executed, where a Find operation (a.o. for obtaining fathers in a tree) is considered to be one step. Consider some insertion. Apart from $O(1)$ steps for lines 1-3 we have the following cost (which is only the case if $3ec(x) \neq 3ec(y)$). Let the number of classes decrease by d ($d \geq 1$). By a similar argument as for procedure call $TreePath_2$ it follows that a call of $TreePath_3$ (line 4) takes $O(d)$ steps of computation. It is easily seen that the call of procedure *AdjustCycles* (line 5) takes $O(d)$ steps plus the time needed for the Circular Split operations. Finally, line 6-7 take $O(d)$ steps apart from the time needed for the Unions.

Concluding the above observations we obtain the following property.

Property 4.4 *A call of procedure $insert_3$ in a 2-edge-connected graph takes $O(1+d)$ steps plus the time needed to join 3-edge-connected classes and to perform Circular Splits, where d is the number by which the amount of classes decreases.*

Observe that there exist $O(n)$ different classes during all insertions. Moreover, initially for the 2-edge-connected graph from which is started, there are at most $2(n-1)$ class representatives, since each class representative corresponds to an edge in $Cyc(3ec(G))$, the tree $Cyc(3ec(G))$ contains at most n class nodes, leaves of the tree are class nodes and edges connect cycle nodes and class nodes only. Hence, we obtain the following lemma, which a.o. can be used in [14].

Lemma 4.5 *Given a 2-edge-connected graph G of n nodes with a cycle tree, there exists a data structure that allows insertions of edges in G and that can answer queries of the following type at any moment: given two nodes in G, are these nodes 3-edge-connected. The total time for m insertions and queries is $O(m+n)$ plus the time needed to perform $O(m+n)$ Finds and $O(n)$ Unions and Splits in a Union-Find or a Circular Split-Find structure for n elements.*

By using a Union-Find structure and a Circular Split-Find structure with time complexity $O(n + m.\alpha(m,n))$ time for all Union/Splits on n elements and for m Finds (cf. [11, 19, 20] and [5, 13]) we obtain the following result.

Lemma 4.6 *Given a 2-edge-connected graph G of n nodes with a cycle tree, there exists a data structure that allows insertions of edges in G and that can answer queries of the following type at any moment: given two nodes in G, are these nodes 3-edge-connected. The total time for m insertions and queries is $O((m+n).\alpha(m,n))$ time.*

4.2 General Graphs

We now extend the solution of the previous section to general graphs.

Note that for detecting the 3-edge-connected classes it suffices to detect the 3-edge-connected components inside the 2-edge-connected components (cf. Lemma 2.10). Therefore, our algorithms for general graphs maintain the 2-edge-connected classes by using the previous solutions for 2-edge-connectivity (Section 3) and maintain the 3-edge-connected classes by using the previous solutions for 3-edge-connectivity within 2-edge-connected components (=graphs) (Subsection 4.1).

The representation of a graph consists of the representations and the data structures of both Section 3 and Subsection 4.1 (for 2-edge-connectivity and 3-edge-connectivity respectively). Hence, there is a cycle tree (of 3-edge-connected class nodes) for each 2-edge-connected component.

Initially, there are n nodes and no edges in the graph. Each node forms a connected, a 2-edge-connected and a 3-edge-connected class on its own. For each class a distinct class node with the data as described in the previous (sub)sections is present. (Of course no cycle nodes are present yet.) Note that the initialisation can be performed in $O(n)$ time.

Suppose edge (e, x, y) is inserted in graph G yielding graph G'. Then the updates are performed by procedure $INSERT$ (given in Figure 16), that is based on procedure $insert_2$ (cf. Figure 4). The procedure works as follows. Three cases are considered (cf. Figure 16).

If $c(x) \neq c(y)$ (line 2-3), then the 2-edge-connected classes do not change. Therefore the computations performed in $insert_2$ for this case (i.e. line 2-8 of Figure 4) suffice here.

Otherwise, if $2ec(x) = 2ec(y)$ (line 23-24) then the edge is inserted inside a 2-edge-connected component. Therefore procedure $insert_3$ (Figure 10) is performed, that deals with 3-edge-connected classes within a 2-edge-connected component.

Otherwise, we have $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$ (line 4-22). Then consider $2ec(G)$. Let P_2 be the tree path between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ (consisting of the class nodes only) and let CS_2 be the cyclic list obtained from P_2 by inserting the interconnection edges between consecutive class nodes of P_2 and by inserting the edge (e, x, y) between class nodes $2ec(x)$ and $2ec(y)$. Then the major changes are the following:

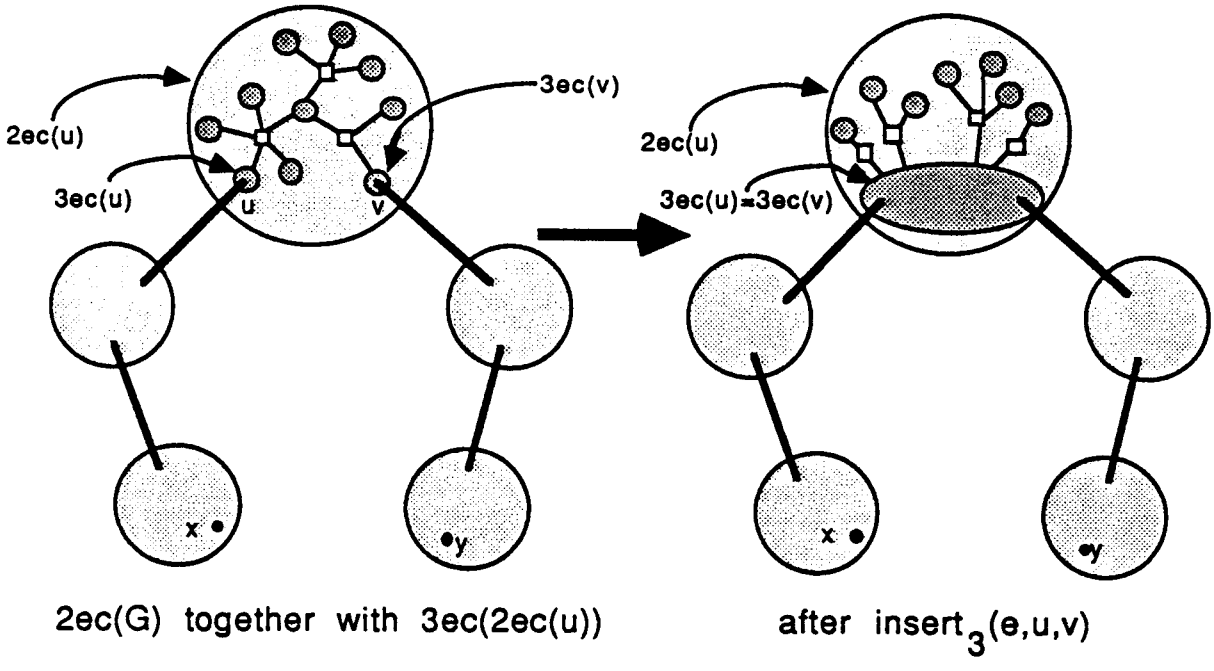
- all 2-edge-connected classes corresponding to class nodes on P_2 form one new 2-edge-connected class
- for each 2-edge-connected class C on P_2, the 3-edge-connected classes inside C (and hence the corresponding cycle tree) are changed: several 3-edge-connected classes may form one new 3-edge-connected class

- a new cycle of 3-edge-connected classes arises that links the (updated) cycle trees that correspond to the 2-edge-connected classes on CS_2

We consider the updates more precisely.

Consider the changes of the 3-edge-connected components that occur in 2-edge-connected classes on P_2. Consider a particular 2-edge-connected class C on P_2 in $2ec(G)$. Let u and v be the two nodes in C that are end nodes of interconnection edges on CS_2. Then there is a new path between u and v in G' that does not intersect with C except for u and v, where such a path did not exist in G before. Hence, considered within C only, this corresponds to inserting a temporary edge between the nodes u and v (cf. Figure 15). Therefore, we can first insert a temporary edge between u and v to update the 3-edge-connected classes (and hence the cycle tree) inside C (causing u and v to be in the same 3-edge-connected class) and then perform all remaining updates w.r.t. the insertion of (e, x, y).

Figure 15: Tree path versus temporary edges.



Now suppose all these "local" insertions are performed in the 2-edge-connected classes on P_2. Then the two edges in CS_2 that are incident with one 2-edge-connected class C on P_2 have their end nodes in the same (updated) 3-edge-connected class in C. Call such a 3-edge-connected class the interconnection class. Then all these interconnection classes form a new cycle r. Hence, all the updated

cycle trees in the 2-edge-connected classes on P_2 (that result from the local insertions of temporary edges) must be linked to the new cycle node r. All these cycle trees now form one new tree together.

According to the above observations the following is performed in procedure $INSERT$ (cf. line 5-21).

First, the tree path P_2 in $2ec(G)$ is computed together with the corresponding sequence CS_2 that also contains the interconnection edges and the edge (e, x, y). Note that these interconnection edges can easily be obtained from the father fields of all class nodes that are on P_2. (In fact this sequence can be obtained in $TreePath_2$ instead of P_2.) Then for each pair of nodes u and v that are in a 2-edge-connected class on P_2 and that are end nodes of two consecutive edges in CS_2 (where u and v may be equal), procedure $insert_3((e', u, v))$ is executed to adapt the 3-edge-connected classes inside class $2ec(u)$ by means of a temporary edge (e', u, v) that only exists during this execution (cf. Figure 15). Moreover, the cyclic list CS_3 of the 3-edge-connected interconnection classes is extended with the (updated) class node $3ec(u)$ $(= 3ec(v))$. Finally, if the 2-edge-connected class $2ec(u)$ is not the largest class c_0 "on" P_2 (i.e., it does not contain the largest number of nodes), then class node $3ec(u)$ is made to be the new root of the (updated) cycle tree in which it is contained (inside the 2-edge-connected class $2ec(u)$) by reversing the root path of node $3ec(u)$. This is done by procedure $ReverseRootPath_3$, which works similar to procedure $ReverseRootPath_2$ with obvious adaptations.

Afterwards all 2-edge-connected classes are joined (while the father of the resulting 2-edge-connected class $2ec(x)$ is adapted) and a new cycle node r for the new cycle corresponding to CS_3 is created. For each 3-edge-connected class $c \in CS_3$, that is on cycle r, a class representative $repr(c, r)$ is created and is inserted in the representative set of class node c. (This can be done for the Union-Find structure as follows: first make a singleton set of $repr(c, r)$ and then join that set with the representative set of c.) Then the father pointers w.r.t. this new cycle node r are adapted: the father of r will be the class node cc_0 (the 3-edge-connected class cc_0 is the interconnection class that was contained in the largest 2-edge-connected class c_0), while all other 3-edge-connected class nodes in CS_3 have r as their father. Note that now each 3-edge-connected class node occurring in the new cycle has at most one father pointer, since all class nodes in CS_3 except for cc_0 were the roots of the cycle trees in the 2-edge-connected classes on CS_2. Therefore, all father pointers implement a rooted tree.

The Union-Find and the Circular Split-Find structures that we use here are the basic structures that take $O(n. \log n)$ time altogether for all the Unions/Splits on n elements and that take $O(1)$ time for each Find (cf. Subsection 2.2).

Figure 16: Procedure $INSERT((e,x,y))$.

(1) **procedure** $INSERT((e,x,y))$;
(2) **if** $c(x) \neq c(y)$
(3) $\longrightarrow$ $insert_2((e,x,y))$
(4) [] $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$
(5) $\longrightarrow$ $Treepath_2(2ec(x), 2ec(y), P_2, fath)$;
(6) let c_0 be the class node on P_2 with the largest number of nodes in its class;
(7) construct the cyclic sequence CS_2 from P_2 by inserting the
(8) interconnection edges between consecutive class nodes on P_2 and by
(9) inserting edge (e,x,y) between class nodes $2ec(x)$ and $2ec(y)$;
(10) $CS_3 := \emptyset$;
(11) **for all** end nodes u, v of consecutive edges in CS_2 with $2ec(u) = 2ec(v)$
(12) $\longrightarrow$ $insert_3((e',u,v))$, for a temporary edge (e',u,v);
(13) insert $3ec(u)$ in CS_3;
(14) **if** $2ec(u) \neq c_0 \longrightarrow$ $ReverseRootPath_3(3ec(u))$
(15) [] $2ec(u) = c_0 \longrightarrow$ $cc_0 := 3ec(u)$
(16) **fi**
(17) **rof**;
(18) **for all** $C \in P \backslash \{2ec(x)\} \longrightarrow$ $Union_{2ec}(C, 2ec(x))$ **rof**;
(19) $father(2ec(x)) := fath$;
(20) make a new cycle r of the class nodes in CS_3 in the same cyclic order
(21) in which they appear in CS_3, where cc_0 is the father of the new cycle
(22) node r and the father of the class nodes in $CS_3 \backslash \{cc_0\}$ is r
(23) [] $c(x) = c(y) \wedge 2ec(x) = 2ec(y)$
(24) $\longrightarrow$ $insert_3((e,x,y))$
(25) **fi**

4.2.1 Complexity

Let us look at the time complexity. Note that procedure $INSERT$ operates similar to procedure $insert_2$ apart from the computations made because of 3-edge-connectivity. Therefore we only have to consider these extra computations.

First we show that the total number of class representatives that exists during the entire process of insertions is at most $2n-1$ if n is the number of nodes in the graph. Note that class representatives (in cycles) are only created when 2-edge-connected classes are joined. In particular, one class representative arises per 2-edge-connected class that is joined with another class. Since initially in the "empty" graph (with no edges) there are n 2-edge-connected classes, it follows that there exist at most $2n-1$ different 2-edge-connected classes throughout all operations. Hence, there exist at most $2n-1$ different class representatives.

We now compute the time complexity of procedure $INSERT$ for all edge insertions.

All computations in lines 1-5, 18, 19, 23 and 25 correspond to computations of procedure $insert_2$ and hence take altogether $O(n \log n + e)$ time for e edge-insertions. Moreover, lines 6-10 and 20-22 can be performed within the same time complexity as line 5 since they all need time linear to the length of P_2. Therefore we charge this cost to line 5, what does not increase the order of time complexity of that line. Hence, the only computations we need to consider now are those performed in lines 11-17 and line 24.

By Property 4.4 the execution of $insert_3$ takes $O(1+d)$ time apart from the time needed to join 3-edge-connected classes and to perform Circular Splits, where the number of 3-edge-connected classes decreases with d. Firstly note that the number of calls of procedure $insert_3$ in line 12 of procedure $INSERT$ is $O(|CS_2|)$. Therefore we can charge $O(1)$ time per call to line 5 without increasing the order of time complexity too. A similar remark can be made for the call of procedure $insert_3$ in line 24: $O(1)$ time can be charged to procedure call $INSERT$. Hence, we only need to consider the remaining part $O(d)$ of the cost $O(1+d)$ of a call of $insert_3$. Since initially there are n 3-edge-connected classes and since the number of classes only decreases and never increases, it follows that the remaining time $O(d)$ spent by procedure $insert_3$ (where d is the decrease in the number of 3-edge-connected classes) adds up to $O(n)$ for all calls together. The Union-Find structure and the Circular Split-Find structure take $O(n.\log n)$ time for all Unions and Splits, since there are $O(n)$ elements occurring in these structures.

Adding all the above time complexities yields a total time complexity of $O(n \log n + e)$ altogether for the insertions of e edges. Moreover, only the edges that become interconnection edges between 2-edge-connected classes at the time of insertion (and hence, for which their end nodes are in two distinct connected components just before the insertion) need to be stored. Hence, since there exist at most $n-1$ such edges during the entire sequence of insertions, the space complexity is $O(n)$.

We have proved the following theorem.

Theorem 4.7 *Given a graph G, there exists a data structure such that the query whether two nodes are 2-edge-connected or 3-edge-connected can be answered in $O(1)$ time. Starting from the empty graph $G =< V, \emptyset >$ (i.e., a graph with no edges), the insertion of e edges take $O(n \log n + e)$ time altogether, if n is the number of nodes in G. Finally, the data structure can be initialised in $O(n)$ time and it uses $O(n)$ space.*

4.3 Maintaining complete 3-edge-connected components

In this subsection we study the maintenance of the actual 3-edge-connected components.

We first state a lemma and a corollary on the relation between the auxiliary edges of a 3-edge-connected component H of a graph G (cf. Def. 2.8) and the cycles in which (the class of nodes of) H is contained in graph $3ec(G)$. We only need to consider 2-edge-connected graphs, since by Lemma 2.10 we may restrict ourselves to determining 3-edge-connected components inside 2-edge-connected components only.

Lemma 4.8 *Let $G =< V, E >$ be a 2-edge-connected graph. Let class c be an equivalence class of V w.r.t. 3-edge-connectivity. Let x and y be nodes in class c. Then the maximal number of edge-disjoint paths between x and y that intersect with class c at x and y only and that contain nodes outside class c, equals the number of cycle nodes s in $Cyc(3ec(G))$ for which there exist 2 edges (e, x, u) and (e', y, v) in G such that $(e, 3ec(x), 3ec(u))$ and $(e', 3ec(y), 3ec(v))$ are in cycle s in $3ec(G)$.*

Proof. Consider a maximal set of edge-disjoint paths between x and y in G, that intersect with class c in x and y only and that contains nodes outside class c. W.l.o.g. these paths are simple. Each such path P contains two unique edges (e, x, u) and (e', y, v) that are at the end of it. Hence $u, v \notin C$ and therefore $(e, 3ec(x), 3ec(u))$ and $(e', 3ec(y), 3ec(v))$ are edges of a simple cycle in $3ec(G)$. Because every edge in $3ec(G)$ has exactly one original in G, it follows that each such a path uniquely corresponds to a simple cycle.

On the other hand, consider a cycle s in $3ec(G)$ with distinct edges (e, c, a) and (e', c, b), for which there are two edges (e, x, u) and (e', y, v) in G such that $(e, c, a) = (e, 3ec(x), 3ec(u))$ and $(e', c, b) = (e', 3ec(y), 3ec(v))$. Then there is a simple path in G that starts in x and ends in y, with edges (e, x, u) and (e', y, v) at the end of these paths. For, edges (e, c, a) and (e', c, b) form a cut edge set in $3ec(G)$ and hence by Lemma 2.6 (sub 1 and 2), (e, x, u) and (e', y, v) are a cut edge set in G with x and y on the one side and (since a cut edge set of only one edge does not exist in G) u

and v on the other side: therefore there is a simple path from u to v not using these edges. Because of such sets being cut edge sets it follows that all paths constructed in this way from the different cycle nodes, are edge disjoint. □

Corollary 4.9 *Let G be a 2-edge-connected graph. Each edge between a cycle node s and class node c in $Cyc(3ec(G))$ can uniquely be related to an auxiliary edge (e,x,y) in class c for which there exist 2 so-called interconnection edges (e_1,x,u) and (e_2,y,v) in G such that $(e_1,3ec(x),3ec(u))$ and $(e_2,3ec(y),3ec(v))$ are in cycle s in $3ec(G)$.*

We base our strategy on Corollary 4.9. Each node x contains 4 adjacency lists consisting of pointers to edges that have x as an end node as follows:

- $list3.aux$ containing the auxiliary edges of the 3-edge-connected component in which x is contained
- $list3$ containing the edges of the 3-edge-connected component in which x is contained except for those in $list3.aux$
- $list2$ containing the edges of the 2-edge-connected component in which x is contained except for those in $list3$ (and $list3.aux$)
- $list1$ containing the remaining edges

Moreover, each existing edge contain pointers to its occurrences in adjacency lists. It is easily seen how these list can be used to traverse or enumerate components.

In our algorithms we have the following extensions for the class representatives. A class representative $repr(c,s)$ for a class node c and a cycle node s in graph $3ec(G)$ contains in addition pointers to the two interconnection edges in cycle s that are incident with class node c and to the auxiliary edge in class c related to cycle s (cf. Corollary 4.9).

The algorithms for insertion of an edge (e,x,y) are adapted as follows. Firstly, the edge (e,x,y) is inserted in the proper adjacency list of its end nodes: viz., if at the moment of insertion x and y are 2-edge-connected (or 3-edge-connected), then (e,x,y) is inserted in the lists $list3$ (because x and y will be 3-edge-connected after the insertion), if they are connected but not 2-edge-connected, then (e,x,y) is inserted in $list2$ and otherwise (e,x,y) is inserted in $list1$. We have the following cases for additional computations for the insertion.

If $3ec(x) \neq 3ec(y) \wedge 2ec(x) = 2ec(y)$, then the splitting of a cycle s, where two classes c and d on it are joined into one new class, has two consequences. The auxiliary edges in the classes c and d that correspond to the old cycle s disappear. If a resulting cycle appears to be a trivial cycle, then the interconnection edge(s) of the

original cycle that have both end nodes in the new class, become internal edges of that class: these edges are moved from *list*2 to *list*3. Otherwise, new auxiliary edges arise in the new class c' according to the resulting nontrivial cycle(s). The end nodes of these auxiliary edges can be obtained from the end nodes of the interconnection edges in these cycles that are incident with class c'. All these adaptations can easily be integrated in procedure *AdjustCycles* by using the appropriate pointers stored in the class representatives on which a Circular Split is performed, without increasing the order of time complexity.

If $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$ then a new cycle is created (cf. Figure 16). The creation of the new cycle (line 20-22) can easily be extended with the computation of the pointers in the class representatives to interconnection edges of the cycle. Moreover, note that each 3-edge-connected class that is in the new cycle s must now have a new auxiliary edge corresponding to s (cf. Corollary 4.9). Therefore, for each such 3-edge-connected class, a new auxiliary edge needs to be created. Note that in fact the collection of temporary edges created in line 12 already satisfies the constraints of Corollary 4.9 and hence we may take these edges as auxiliary edges. It is easily seen that all these additional computations can be performed without increasing the order of time complexity.

Theorem 4.10 *Given a graph G, there exists a data structure such that the query whether two nodes are 2-edge-connected or 3-edge-connected can be answered in $O(1)$ time and that maintains the 2-edge-connected and 3-edge-connected components of G when edges are inserted. Starting from the empty graph $G =< V, \emptyset >$ (i.e., a graph with no edges), the insertion of e edges take $O(n \log n + e)$ time altogether, if n is the number of nodes in G. Finally, the data structure can be initialised in $O(n)$ time and it uses $O(n + e)$ space when the 2-edge-connected or 3-edge-connected components are maintained and it uses $O(n)$ space otherwise.*

It is easily seen that besides edges, new nodes can be inserted in the graph in $O(1)$ time (each inserted node forms a 2-edge-connected and a 3-edge-connected class in its own at the moment of insertion). Therefore, the statement in the above theorem can be extended with node insertions, where n is the final number of nodes in the graph (and n is the initial number regarding the time needed for initialisation).

5 Conclusion

In this paper we have presented algorithms for maintaining the 2- and 3-edge-connected components in a graph under the insertion of edges and vertices. The insertion of e edges costs $O(n. \log n + e)$ time in total, while at any moment connectivity queries can be answered in time $O(1)$. The time bounds can be improved to $O(n + m.\alpha(m, n))$ where m is the total number of queries and edge insertions and

n is the number of nodes, using a number of sophisticated data structuring techniques. These results will be presented in the accompanying paper [14] since the additional data structures are rather complicated. Moreover, the same time bounds can be achieved for 2- and 3-vertex-connectivity. We refer to [14, 15]. In this way in [14] and [21] optimal algorithms are obtained for 2-edge/vertex-connectivity (with different methods) while [14, 15] also present optimal algorithms for 3-edge/vertex-connectivity.[1]

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley Publ. Comp., Reading, Massachusets, 1974.

[2] G. Di Battista and R. Tamassia, Incremental Planarity Testing, Proc. 30^{th} Ann. Symp. on Found. of Comp. Sci. (FOCS) 1989, 436-441.

[3] G.N. Frederickson, Data structures for on-line updating of minimum spanning trees, with applications, SIAM J. Computing 14 (1985), pp.781-798.

[4] M.L. Fredman and M.E. Saks, The Cell-Probe Complexity of Dynamic Data Structures, Proc. 21^{th} Ann. ACM Symp. on Theory of Comput. (STOC) 1989, 345-354

[5] H.N. Gabow, A Scaling Algorithm for Weighted Matching on General Graphs, Proc. 26^{th} Ann. Symp. on Found. of Comp. Sci. (FOCS) 1985, 90-100.

[6] H.N. Gabow, Data Structures for Weighted Matching and Nearest Common Ancestors with Linking, Proc. 1^{st} Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA) 1990, 434-443.

[7] F. Harary, Graph Theory, Addison-Wesley Publishing Company, Reading, Massachusets, 1969.

[8] G.F. Italiano, Amortized efficiency of a path retrieval data structure, Theoretical Computer Science, 48, (1986), pp. 273-281.

[9] G.F. Italiano, Finding paths and deleting edges in directed acyclic graphs, Information Processing Letters, 28, (1988), pp. 5-11.

[10] J.A. La Poutré and J. van Leeuwen, Maintenance of Transitive Closures and Transitive Reductions of Graphs, In: H. Göttler, H.J. Schneider (Eds.), Graph-Theoretic Concepts in Computer Science 1987, Lecture Notes in Computer Science Vol. 314, Springer-Verlag, Berlin, pp. 106-120.

[1] Very recently, Galil and Italiano independently obtained such time bounds for a special case of 3-edge-connectivity, viz., if the initial graph is connected.

[11] J.A. La Poutré, New Techniques for the Union-Find Problem, Proc. 1^{st} Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA) 1990, 54-63.

[12] J.A. La Poutré, Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines, Proc. 22^{th} Ann. ACM Symp. on Theory of Comput. (STOC) 1990, 34-44.

[13] J.A. La Poutré, A Fast and Optimal Algorithm for an Extension of the Split-Find Problem on Pointer Machines, Tech. Rep. RUU-CS-89-20, Utrecht University, 1989.

[14] J.A. La Poutré, Maintenance of 2- and 3-connected components of graphs, Part II: 2- and 3-edge-connected components and 2-vertex-connected components, Tech. Rep., Utrecht University, to appear.

[15] J.A. La Poutré, Maintenance of 2- and 3-connected components of graphs, Part III: 3-vertex-connected components, in preparation.

[16] K. Mehlhorn, Graph Algorithms and NP-completeness, Springer-Verlag, Berlin, 1984.

[17] F.P. Preparata and R. Tamassia, Fully Dynamic Techniques for Point Location and Transitive Closure in Planar Structures, Proc. 29th Ann. Symp. on Found. of Comp. Sci. (FOCS) 1988, pp. 558-567.

[18] H. Rohnert, A dynamization of the all pairs least cost path problem, In: K. Mehlhorn (ed.), 2nd Annual Symposium on Theoretical Aspects of Computer Science 1985, Lecture Notes in Computer Science Vol. 182, Springer-Verlag, Berlin, pp. 279-286.

[19] R.E. Tarjan, Efficiency of a Good but Not Linear Set Union Algorithm, J. ACM 22, No. 2, April 1975, pp 215-225.

[20] R.E. Tarjan and J. van Leeuwen, Worst case analysis of set union algorithms, J. ACM, 31, (1984), pp. 245-281.

[21] J. Westbrook and R.E. Tarjan, Maintaining Bridge-Connected and Biconnected Components On-Line, Tech. rep. CS-TR-228-89, Princeton University, 1989.