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

J.A. La Poutré

RUU-CS-90-27
October 1990

Utrecht University

Department of Computer Science

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

A preliminary version of this paper was completed in October 1990. The final version of this paper was released in August 1991.

ISSN:0924-3275

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

J.A. La Poutré
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

In this paper data structures and algorithms are presented to efficiently maintain the 2- and 3-edge-connected components and the 2-vertex-connected components of a graph, under insertions of edges in the graph. At any moment, the data structure can answer the following type of query: given two nodes in the graph, are these nodes 2- or 3-edge-connected or 2-vertex-connected. Starting from an "empty" graph of n nodes, the algorithms run in $O(n+m.\alpha(m,n))$ time, where m is the total number of queries and edge insertions. The data structure allows for insertions of nodes also. Besides, a linear time algorithm is presented for maintaining the 2-edge-connected components in case the initial graph is connected.

1 Introduction

Dynamic algorithms are known for several graph problem. Examples are e.g. computing transitive closures (cf. [12, 13, 14], or cf. [22] for planar graphs), minimal spanning trees (cf. [4, 5]), incremental planarity testing (cf. [3]), maintaining shortest paths (cf. [23]) and nearest common ancestors in trees ([8]).

In this paper we consider the problem of maintaining the 2- and 3-edge-connected components and the 2-vertex-connected components of a graph under insertions of edges (and vertices), where k-edge-connectivity and k-vertex-connectivity are defined as follows. Let G be an undirected graph. Two nodes x and y are called

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

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). Nodes x and y are k-vertex-connected if either there is no edge between x and y and after the removal of any set of at most $k-1$ node(s) x and y are (still) connected or there are $l \geq 1$ edges between x and y and after the removal of any set of at most $m \leq min\{k-1, l\}$ edges and $k-m-1$ vertices, x and y are (still) connected.

In [18] a data structure with algorithms is presented 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. The insertion of e edges takes $O(n \log n + e)$ time altogether. In this paper we present a data structure, called fractionally rooted tree. We show that by means of this data structure the above time bounds can be improved for maintaining the 2- and 3-edge-connected components of a general graph, i.e., starting from an "empty" graph of n nodes. The solution has a 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. We also give a solution for 2-vertex connectivity with the same time bounds. Recently, Westbrook and Tarjan [28] independently obtained the same time bounds for 2-edge/vertex-connectivity. The methods though are quite different. Very recently, Galil and Italiano [10] independently obtained results with these time bounds for a special case of the problem of maintaining 3-edge-connected components of graphs, viz., in which the initial graph is connected. Finally, we also present a linear time solution for maintaining 2-edge-connected components in case the initial graph is connected. This solution uses the linear RAM-algorithm of [9] for Union-Find problems.

The paper is organized as follows. Section 2 contains the preliminaries. In Section 3 the specifications of the operations on fractionally rooted trees are given. In Section 4 the maintenance of 2-edge-connected components is considered. In Sections 5, 6 and 7 the fractionally rooted tree is presented. To be precise, in Section 5 observations and ideas are given, in Section 6 division trees are described, being the building elements for fractionally rooted trees, and in Section 7 the fractionally rooted trees are presented. The complexity is considered in Sections 8 and 9. The results are presented in Section 10. In Section 11 the maintenance of 3-edge-connected components is considered. Finally, in Section 12 the maintenance of 2-vertex-connected components is considered.

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. A graph is called *empty* if it consists of nodes without edges.

We use the following notions (see also [11]). Two nodes are called *adjacent* if there is an edge with these nodes as end nodes. 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 (different) paths are called *vertex disjoint* if they do not have a common vertex except for their end vertices. Two nodes are called *connected* if there exists a path between them. A (elementary) *cycle* is a path of which the end nodes are equal and in which 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.

We extend the terminology. Consider a tree T. A set of *nodes* of T *induces a subtree* of T if these nodes are the nodes of a subtree of T. A set of *edges* of T *induces a subtree* of T if these edges and their end nodes are the edges and nodes of a subtree of T.

Suppose the vertex set of T is partitioned into disjoint subsets, where each set induces a subtree of T. Suppose each induced subtree of T is contracted to a new node, called *contraction node*. We say that the subset that induces that subtree is contracted to that contraction node. We say that a node (or an edge) in such a subtree is *contracted to (or is contained in)* that contraction node. For an edge (e, x, y) that connects two different induced subtrees that are contracted to the nodes p and q, the edge (e, p, q) is called the *contraction edge* of (e, x, y). Edge (e, x, y) is called the *original* (in T) of (e, p, q). (We give an edge and its induced edge the same name e.) A tree CT consisting of the contraction nodes and the contraction edges is called *a contraction tree of* T and T is called to be contracted to CT. For a class D of edges in T, the class of edges in CT induced by D consists of the contraction edges in CT that have their originals in D.

If the tree T is contracted several times resulting in a tree CCT, then we say that a node $x \in T$ is contracted to (or is contained in) contraction node $c \in CCT$ if the consecutive contractions result in node c is we start from node x (i.e., we make the relation transitive). Similarly we make the relation contraction edge a transitive relation.

Now let tree T be rooted. The *father node of an edge* is the end node of the edge that is closest to the root. Then *father edge of a node* x is the edge incident with x and with the father node of x. The *father edge of an edge* is the father edge of the father node of that edge. For a subtree S of T the *maximal* node of that subtree is the (unique) node that is nearest to the root. We call an edge of subtree S a *maximal edge* if it is incident with the maximal node of S.

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".

A *singleton* class or set or a singleton tree is a class, set or tree that consists of one element or node respectively.

Notation 2.1 *For a set S, $|S|$ denotes the number of elements in the set. For a tree T, $|T|$ denotes the number of nodes in the tree. For a list L, $|L|$ denotes the number of elements in the list. If to each element in a list L a sublists is attached, then (still) $|L|$ denotes the number of elements in the list (without considering the sublists).*

2.2 Connectivity

We give the definitions for k-edge/vertex-connectivity.

Definition 2.2 *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.*

Definition 2.3 *Two non-adjacent nodes x and y are k-vertex-connected ($k \geq 1$) if after the removal of any set of at most $k-1$ vertices x and y are (still) connected. Two adjacent nodes x and y with l edges that have x and y as end nodes are k-vertex-connected ($k \geq 1$) if after the removal of any set of at most $m \leq min\{k-1, l\}$ edges and $k-m-1$ vertices, x and y are are (still) connected. If the removal of a set of vertices separates the vertices x and y (as described in the cases above), then that set is called a cut set for x and y.*

The following lemma due to Menger (cf. [21]) characterizes pairs of k-edge-connected vertices.

Lemma 2.4 [Menger [21]] *Two nodes x and y are k-edge-connected iff there exist k edge-disjoint paths between x and y. Two nodes x and y are k-vertex-connected iff there exist k different vertex-disjoint paths between x and y.*

It is easily seen that we may require that the paths referred to in the lemma are simple paths, without affecting the lemma. Furthermore, it is easily seen that if two nodes are k-edge-connected or k-vertex-connected, then they are k'-edge-connected or k'-vertex connected for any k' with $1 \leq k' \leq k$ respectively. Note that for $k = 2$, two nodes are 2-edge-connected iff they lie on a common elementary cycle, and two nodes are 2-vertex-connected iff they lie on a common simple cycle.

Lemma 2.5 *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 the subgraphs of G that are induced by the equivalence classes of nodes w.r.t. 2-edge-connectivity. To be precise, 2-edge-connected components are defined as follows.

Definition 2.6 *Let $G =< V, E >$ be a graph. Let $C \subseteq V$ be an equivalence class w.r.t. 2-edge-connectivity. Then $< C, \{(e, x, y) \in E | x, y \in C\} >$ is a 2-edge-connected component of G (induced by C).*

In the sequel, we quote some lemmas from [18].

The following lemma is based on the observation that for two nodes that are k-edge-connected ($k \geq 2$), there exist k edge-disjoint simple 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 ($k \geq 1$).*

For an equivalence class C of nodes w.r.t. 3-edge-connectivity, we can define the notion of a 3-edge-connected component (induced by that class) such that Lemma 2.7 holds for 3-edge-connectivity too. We will not give the formal definition here, but only state that it contains the edges of the graph that have both end nodes in C, together with for each pair of nodes x and y in C, a number of new edges between them that equals the maximal number of nontrivial edge-disjoint paths between x and y that intersect with C at x and y only, and that intersect with $V \backslash C$.

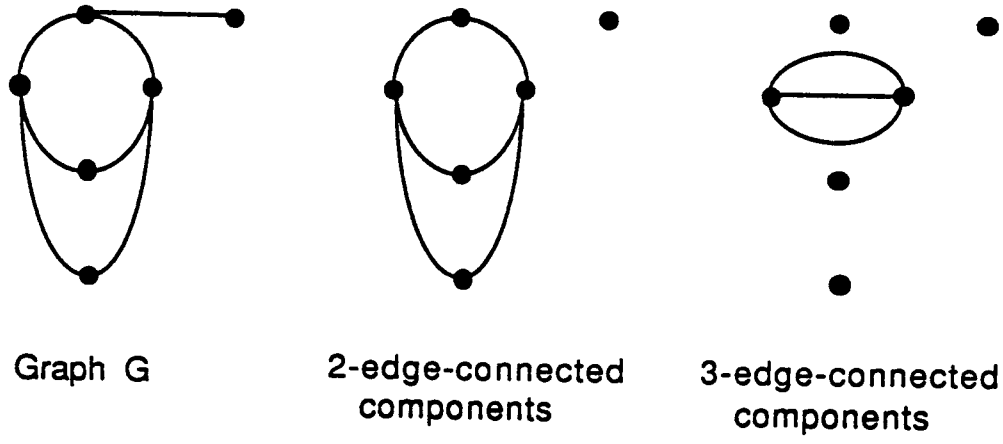
Henceforth, we will usually call an equivalence class for 2-edge-connectivity a *2ec-class*, and an equivalence class for 3-edge-connectivity a *3ec-class*.

By means of Lemma 2.7 the following corollary easily follows.

Corollary 2.8 *Let G be a graph. Let C_2 be a 2ec-class and let C_3 be a 3ec-class of G. 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. If $C_3 \subseteq C_2$ then C_3 is a 3ec-class of H.*

Stated differently, each 3ec-class of G is a 3ec-class of some 2-edge-connected component of G and vice versa.

Figure 1: The 2- and 3-edge-connected components of graph G.



In the observations in this paper, like in [18], we will represent the 2ec-classes and the 3ec-classes of a graph by means of a "super" graph. To this end, we introduce the notion of a *class node.*

Definition 2.9 *Let $G =< V, E >$ be a graph. Let V be partitioned into classes and let some new node be related to each class, where each such node is called the class node of the class which it represents. Let $cc(x)$ be the class node of the class containing node x ($x \in V$). 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*

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

Lemma 2.10 *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 for*

all $x, y \in V$ and for all k' with $1 \leq k' \leq k$, x and y are k'-edge-connected in G iff $cc(x)$ and $cc(y)$ are k'-edge-connected in $cc(G)$.

In other words: "internal" edges of classes of k-edge-connected nodes are not relevant for cut edge sets up to size $k-1$.

We call a set S of at least 2 nodes a *2vc-class* if the nodes are 2-vertex-connected and if there does not exist a node not in S that is 2-vertex-connected with the nodes of S (i.e., the class is maximal). Furthermore we define a *quasi class* to be any set of two nodes that are the end nodes of a cut edge.

The *2-vertex-connected components* of a graph G are the subgraphs of G that are induced by the 2vc-classes of nodes. (Note that the 2-vertex-connected components and the subgraphs induced by quasi classes as we defined them are usually called the blocks of a graph.)

2.3 The Ackermann Function

The Ackermann function A is defined as follows. For $i, x \geq 0$ function A is given by

$$\begin{array}{rcll} A(0,x) & = & 2x & \text{for } x \geq 0 \\ A(i,0) & = & 1 & \text{for } i \geq 1 \\ A(i,x) & = & A(i-1, A(i,x-1)) & \text{for } i \geq 1,\ x \geq 1. \end{array} \tag{1}$$

The row inverse a of A and the functional inverse α of A are defined by

$$a(i,n) = min\{x \geq 0 | A(i,x) \geq n\} \qquad (i \geq 0,\ n \geq 0) \tag{2}$$

$$\alpha(m,n) = min\{i \geq 1 | A(i, 4\lceil m/n \rceil) \geq n\} \qquad (m \geq 0,\ n \geq 1) \tag{3}$$

Here we take $\lceil 0 \rceil = 1$. The above two definitions correspond to those given in [7, 8, 15, 17]. It is easily shown that the differences with the definitions given in [25, 27] are bounded by some additive constants (except for $a(0,n)$ and $a(1,n)$). We quote some results from [15].

Firstly $A(i,1) = 2$, $A(i,2) = 4$, $A(i+1,3) = A(i,4)$ and $A(i+1,4) = A(i, A(i,4))$ for $i \geq 0$.

Lemma 2.11 $a(i, A(i,x)) = x \quad (i \geq 0,\ x \geq 0)$

Lemma 2.12 *Let $A^{(0)}(i,y) := y$ and $A^{(x+1)}(i,y) := A(i, A^{(x)}(i,y))$ for $i, x, y \geq 0$. Then $A(i,x) = A^{(x)}(i-1,1)$ for $i \geq 1,\ x \geq 0$.*

Let $a^{(0)}(i,n) := n$ and $a^{(j+1)}(i,n) := a(i, a^{(j)}(i,n))$ for $i, j \geq 0,\ n \geq 1$. Then $a(i,n) = min\{j | a^{(j)}(i-1,n) = 1\}$ for $i, n \geq 1$.

By Lemma 2.12 it follows that for every i, $A(i+1,x)$ is the result of x recurrent applications of function $A(i,.)$. Hence we have

$$\begin{aligned}
A(0,x) &= 2x \\
A(1,x) &= 2^x \\
A(2,x) &= 2^{2^{\cdot^{\cdot^{2}}}} \Big\} x \text{ two's} \\
A(3,x) &= \underbrace{2^{2^{2^{\cdot^{\cdot^{2}}}}} \Big\} 2^{2^{2^{\cdot^{\cdot^{2}}}}} \Big\} \cdots 2^{2\}2\}1 \text{ two}} \begin{matrix} \text{two's} \\ \text{two's} \end{matrix}}_{x \text{ braces}}.
\end{aligned}$$

and

$$\begin{array}{lllll}
a(0,n) & = & \lceil \frac{n}{2} \rceil & & \\
a(1,n) & = & \lceil \log n \rceil & = & min\{j | \lceil \frac{n}{2^j} \rceil = 1\} \\
a(2,n) & = & \log^* n & = & min\{j | \lceil \log^{(j)} n \rceil = 1\} \\
a(3,n) & = & & & min\{j | \log^{*(j)} n = 1\}
\end{array}$$

where as usual, the superscript (j) denotes the function obtained by j consecutive applications.

Lemma 2.13 $\alpha(m,n) = min\{i \geq 1 | a(i,n) \leq 4.\lceil m/n \rceil\}$.

Finally, note that $\alpha(m,n) \leq 3$ for $n \leq 2^{2^{\cdot^{\cdot^{2}}}} \Big\} 65536$ two's (which will be the case for all practical values of n).

For simplicity, we extend the Ackermann function as follows:

$$A(i,-1) = 0 \text{ for all } i \geq 0.$$

2.4 Representation and Data Structures

The algorithms and data structures that we present (except for the algorithm in Subsection 4.2) can be implemented on both a Pointer Machine and a Random Access Machine (RAM) with the same complexity. We first informally describe the main aspects of these models of computation. (For a detailed description we refer to [24, 26] and [20], respectively.) A *Pointer Machine* is a machine of which the memory consists of (equal) records. A record in memory is only accessible by means of a pointer to that record, which is a specification of that record (e.g., a pointer can be seen as the internal address of the record in memory). Fields of a record may contain either data values or pointer values. However, no arithmetic on pointers is possible: the only operations on pointers are assignment and testing for equality.

Therefore, a record can not be obtained by calculation of its address but only by means of following a sequence of pointers. A *Random Access Machine (RAM)* is a machine of which the memory consists of storage locations that are numbered 0,1,2, ... W.l.o.g. we interpret the locations as records also, where the values in the fields are integers. A record in memory is accessible by means of its number. However, the integer that can be stored in a field is limited to a size $O(\log n)$ (i.e., it consists of $O(\log n)$ bits), where n is the problem size (i.e., the number of items considered in the problem, like the number of nodes and edges in a graph).

A mentioned above, our algorithms and data structures (except in Subsection 4.2) can be implemented on both a Pointer Machine and a RAM with the same complexity. To be precise, the memory that is used by the implementation consists of records that can only be accessed by means of pointers on which no arithmetic is performed, where each record contains a bounded number of fields (that may contain pointers), and where each field contains $O(\log n)$ bits. This kind of implementation of an algorithm and its associated data structures is called a *pointer/* $\log n$ *solution.*

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.) An edge that has to be inserted is given by its record with the pointers to its end nodes as input for the algorithms. Moreover, we will often not distinguish between a pointer to a record and the record itself.

Lemma 2.5 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. Many solutions have been proposed for the Union-Find problem (cf. [15, 25, 27]): these solutions all take $O(n + m.\alpha(m, n))$ time for all Unions and m Finds on n elements, which is optimal [6, 16]. We call such a structure an α-*UF structure.*

In the sequel, the Union-Find structure is used to maintain the equivalence classes for connectivity, 2-edge-connectivity and 3-edge-connectivity. These structures are denoted by UF_c, UF_{2ec} and UF_{3ec} respectively, where the corresponding Finds on elements x are denoted by $c(x)$, $2ec(x)$ and $3ec(x)$ 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 (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. (This is not made explicit in the algorithms.)

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 $c(y)$ are joined. 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 needed for the Union and Find operations. In the sequel, we use this algorithm for maintaining connectivity. However, we will not make the above computations explicit in the future algorithms for maintaining 2/3-edge-connectivity and 2-vertex-connectivity.

For maintaining 3-edge-connectivity we also need a structure for a problem that is closely related to the Split-Find problem [7, 17], viz. the *Circular Split-Find problem* [17], 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. In [17] fast solutions for the Generalized Split-Find problem, especially the Circular Split-Find Problem, are given which are optimal on pointer machines [16]. These solutions closely correspond to the solutions in [7] for the ordinary Split-Find problem. The solutions take $O(n + m.\alpha(m, n))$ time for all Generalized (or Circular) Splits and m Finds on n elements. We call such a structure an *α-GSF structure*. The Circular Split-Find structure is used in [18]. It will not be used explicitly in this paper, but we only chose appropriate Circular Split-Find structures when we apply the results of [18].

2.5 Problem Description

The problems that we consider in this paper are as follows. Let be given a graph. Then the following operations may be applied on the graph.

insert$((e, x, y))$: insert the edge (e, x, y) in the graph

2ec-comp(x): output the name of the 2-edge-connected component (2ec-class) which contains x

3ec-comp(x): output the name of the 3-edge-connected component (3ec-class) which contains x

Is2vc(x, y): output whether x and y are two nodes in the graph that are 2-vertex-connected and output the name of the 2-vertex-connected component (2vc-class) in which they both are contained (if any)

We call a problem the *2ec-problem* if operations *insert* and *2ec-comp* are considered, the *3ec-problem* if operations *insert*, *2ec-comp* and *3ec-comp* are considered and the *2vc-problem* if operations *insert* and *Is2vc* are considered. Note that the query whether 2 nodes are 2-edge-connected (or 3-edge-connected) can be performed by means of two calls of *2ec-comp* (or *3ec-comp*), namely, one call for each node.

In addition, the above collection of operations can be extended with the insertion of a new (isolated) node in the graph. We will consider this operation only in the last steps of our solutions.

We call the insertion of an edge an *essential insertion* for a given problem, if in the graph either the connectivity relation changes, or for the 2ec-problem, the 2-edge-connectivity relation changes, or for the 3ec-problem, the 2-edge-connectivity relation or 3-edge-connectivity relation changes, or for the 2vc-problem, the 2-vertex-connectivity relation changes. An insertion is called *nonessential* otherwise.

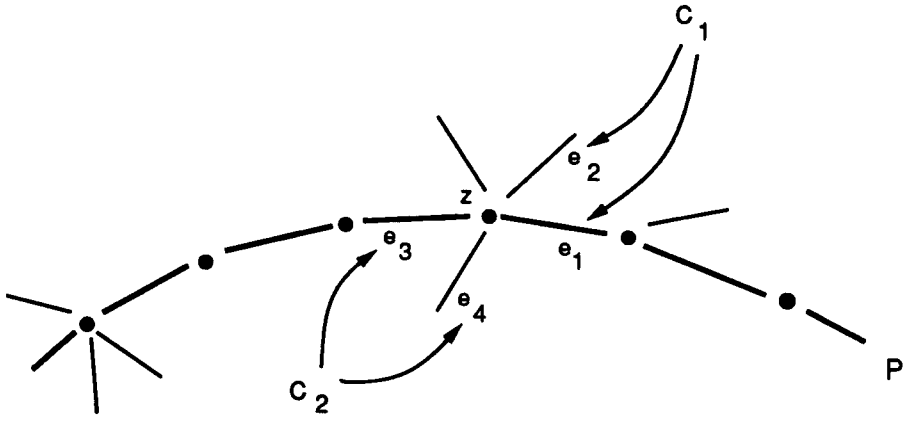
3 Fractionally Rooted Trees: the Operations

We give a formal description of the operations supported by fractionally rooted trees, without considering the data structure itself yet.

Let a forest F be given. Suppose the collection of edges is partitioned into disjoint classes such that each class induces some subtree of F. Such a partition is called an *admissible partition.*

Let x and y be two nodes in the same tree of F. Let P be the tree path between x and y. We call a node x on tree path P an *internal node* of P if it is incident with two edges of P that are in the same edge class. We call a node of P a *boundary node* otherwise. Hence, a boundary node is either one of the end nodes x or y of P, or it is a node for which its two incident edges on P are in different classes. A *boundary edge set* for a boundary node z on P is a set of (0, 1 or 2) edges that contains for each edge e of P that is incident with z, exactly one edge e' which is incident with z and which is in the same edge class as e. (See Figure 2, where path P is drawn with heavy lines, C_1 and C_2 are two different edge classes, $\{e_1, e_2\} \subseteq C_1$ and $\{e_3, e_4\} \subseteq C_2$, and where $\{e_1, e_3\}$, $\{e_1, e_4\}$, $\{e_2, e_3\}$, and $\{e_2, e_4\}$ are boundary edge sets for z on P.) A *boundary list* for the two nodes x and y is a list consisting of the boundary nodes of P, where each boundary node has a sublist that contains

Figure 2: Boundary edge sets



a boundary edge set for it on P. An edge class *occurs* in a boundary list if an edge of it occurs in a sublist in it. (Note that in a boundary list for x and y with $x \neq y$, all nodes have a sublist with two edges except for nodes x and y that each have one edge in their sublist. The boundary list for x and y with $x = y$ consists of node x with an empty sublist.) We say that x and y are *related nodes*, denoted by $x \sim y$, if $x = y$ or if all the edges on P are in the same edge class. (Hence, $x \sim y$ iff x and y are the only nodes in a boundary list for x and y.)

A *joining list* J is a list of nodes with sublists of edges as follows. An edge class *occurs* in list J if an edge of it occurs in a sublist of J. Let CJ be the collection of edge classes occurring in J. It is required that the union of the classes in CJ induces some subtree in F (and hence yields a new admissible partition of the edge set.) Moreover, the nodes in list J must be the nodes that are incident with edges of at least two classes in CJ. For each node z in J, the sublist of z must contain an edge for each class in CJ that contains an edge incident with z.

The following operations, called *FRT-operations*, may be performed on a forest F.

link$((e, x, y))$: Let x and y be nodes in different trees of forest F. Then link the two trees containing x and y by inserting the edge (e, x, y).

boundary(x, y): Let x and y be in the same tree of F, with $x \neq y$. Then output a boundary list for x and y.

joinclasses(J): Let J be a joining list. Then join all the edge classes of which an edge occurs in the list.

equal-class-edges(x, y) : Return an edge incident with x and return an edge incident with y; these edges are in the same class if such edges exist. Return the

names of the edge classes in which the edges are contained.

A call $boundary(x, y)$ is *essential* if $\neg(x \sim y)$ and it is *nonessential* if $x \sim y$. (Note that an essential call *boundary* outputs a boundary list with at least three nodes and at least two edge classes occurring in it, and it outputs a boundary list with two nodes and one edge class otherwise.)

An *essential sequence* is a sequence of calls of *link*, essential calls of *boundary* and calls of *joinclasses* where every (essential) call *boundary*, returning a list BL, is followed by the call $joinclasses(J)$ such that the edge classes occurring in BL also occur in J. (Note that by the definition of joining list this means that J consists at least of the nodes in the boundary list BL that is output by *boundary* except possibly for the end nodes in BL, where for each edge e in the sublist of x in BL there is an edge e in the sublist of node x in J that is in the same edge class as e.)

A *matching sequence* is a sequence of calls of FRT-operations where the subsequence of calls of *link*, essential calls of *boundary* and calls of *joinclasses* forms an essential sequence.

4 Two-Edge-Connectivity

4.1 Graph observations

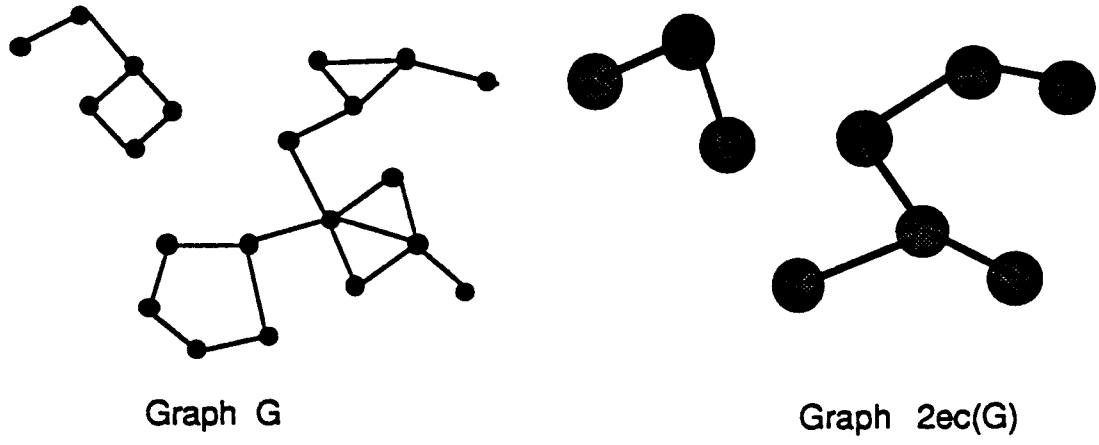
In this subsection we recall from [18] the observations for inserting edges in a graph. Let $G =< V, E >$ be a graph. The set V can be partitioned into equivalence classes for 2-edge-connectivity . Recall that an equivalence class for 2-edge-connectivity is called a 2ec-class. Let each 2-edge-connected 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 2.9):

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

Hence, $2ec(G)$ is the graph that is obtained if we contract each 2ec-class into one (representing) class node. Since $2ec(V)$ represents the set of equivalence classes for 2-edge-connectivity in G, it follows by Lemma 2.10 that $2ec(G)$ is a forest (cf. Figure 3). An edge (e, x, y) in G is called an *interconnection edge* between (classes) $2ec(x)$ and $2ec(y)$ if $2ec(x) \neq 2ec(y)$. (Hence, the edges in $2ec(G)$ are the contraction edges of the interconnection edges in G.)

We consider the 2-edge-connectivity relation under edge insertions by means of the graph $2ec(G)$. Suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G =< V, E >$. We distinguish three cases.

Figure 3: Graph G and the corresponding graph $2ec(G)$.



1. $c(x) \neq c(y)$. Then by Lemma 2.10 $2ec(x)$ 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 $2ec(x)$ and $2ec(y)$ in a tree of $2ec(G)$ and a cycle arises. Hence, all class nodes on the tree path P from $2ec(x)$ to $2ec(y)$ become 2-edge-connected in $2ec(G)$. By Lemma 2.10 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 insertion of this edge does not affect the 2-edge-connectivity relation (cf. Lemma 2.10).

4.2 Algorithms for Initially Connected Graphs

We consider the 2ec-problem in case the initial graph is connected.

We represent the graph $2ec(G)$ by means of a spanning tree of G, denoted by $ST(G)$. Now, a 2ec-class induces a subtree in $ST(G)$. This is seen as follows. Let the two nodes x and y be 2-edge-connected. Then every node z that is on the tree path between x and y is 2-edge-connected with x and y too. For, suppose that an edge is removed from G. Then at least one of the tree paths between x and z or between y and z is not affected. Moreover, there still exists a path between x and y in G since x and y are 2-edge-connected. This yields that there still exists a path between z and x and between z and y in G.

Since at any time, every 2ec-class induces a subtree of $ST(G)$, and since the tree $ST(G)$ can be constructed in advance (i.e., the tree is not built on-line), we can use the Union-Find algorithms of [9] to maintain these classes: this algorithm runs in $O(n + m)$ time for m Finds and n nodes for this special case of the Union-Find problem. (It runs on a RAM but not on a pointer machine with this time complexity.) Moreover, as remarked in [15], a Find can be performed in $O(1)$ worst-case time.

We give the algorithms in case the initial graph is a tree. Consider graph G that initially is a tree (without additional edges). The initialisation of the data structure we use is as follows: implement the tree as a rooted tree and initialise the Union-Find structure of [9] accordingly. We recall from [9] that the name of a set in the Union-Find structure is the (unique) node in the set that is closest to the root.

Suppose an edge (e, x, y) is inserted. Then we have two cases, according to the above observations. If $2ec(x) = 2ec(y)$ then nothing needs to be done. Otherwise, $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. The tree path between $2ec(G)$ and $2ec(G)$ is obtained like in [18] by traversing the root paths of $2ec(x)$ and $2ec(y)$ in $2ec(G)$ stepwisely in an alternating way, i.e., by performing steps of the traversals of these root paths in an alternating way, where a step consists of: obtain the node in $ST(G)$ that is in the current class and that is closets to the root, obtain its father and obtain the name of the class in which it is contained. During this traversals, mark the encountered class names and stop the traversals if one of the two path traversals encounters a class name *top* that has been marked by the other traversal; path P between $2ec(x)$ and $2ec(x)$ consists of the two parts of these root paths up to and including *top*. Remove the marks. Then the classes on P are joined in the order of P (to meet the conditions in [9]).

We consider the time complexity of the method as described for trees. A computation of a tree path P 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. Moreover, for each encountered class name $O(1)$ time is spent. 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. Furthermore, all Unions and m

Finds only take $O(n+m)$ time. Finally, each insertion takes two Finds and $O(1)$ time, apart from the cost of path computations and Unions.

In case the initial graph is connected but it is not a tree, then we do the following. First obtain a spanning tree of the graph, and initialise the structure for this tree. Then insert the edges of the graph that are not in the tree by means of the above algorithm. Then the actual insertions can be performed. Obviously, this initialisation can be done in $O(e_0)$ time, if e_0 is the number of edges in the initial graph. (Note that $e_0 \geq n$.)

Hence, we have the following theorem.

Theorem 4.1 *There exists a structure and algorithms that solve the 2ec-problem for graphs G that are initially connected, and that can be implemented on a RAM, such that the following holds. Starting from a connected graph G, m insert operations take $O(n+m)$ time, if n is the number of nodes in G. Any 2ec-comp(x) query and any nonessential insertion can be performed in $O(1)$ time. The initialisation can be performed in $O(e_0)$ time and the entire structure takes $O(n)$ space, where e_0 is the number of edges in the initial graph.*

4.3 Algorithms and data structures

In this subsection, we will give a solution for the general 2ec-problem with a time complexity of $O(n + m.\alpha(m,n))$ for n nodes and m queries and insertions.

We represent the structure $2ec(G)$ by means of a forest of spanning trees of G. (Hence, each connected component is represented by a tree.) We denote the forest together with additional information (defined below) by $SF(G)$.

We follow a strategy based on observations for $2ec(G)$ in Subsection 4.1. We give the further observations that lead to our algorithm.

Consider $SF(G)$. We augment $SF(G)$ with *edge classes.*

> Let (e,x,y) be an edge in $SF(G)$. If $2ec(x) = 2ec(y)$, then (e,x,y) is in the edge class named $2ec(x)$. Otherwise, edge (e,x,y) forms a singleton class on its own.

An edge class that is a singleton edge class consisting of one edge (e,x,y) with $2ec(x) \neq 2ec(y)$ is called a *quasi class*; otherwise it is called a *real class.* Hence, interconnection edges form quasi classes and vice versa.

As observed in Subsection 4.2, a 2ec-class (of nodes) induces some subtree in $SF(G)$. Hence, in particular a non-singleton 2ec-class (i.e., with at least 2 nodes) induces some subtree in $SF(G)$. The set of the edges in that subtree is a real edge class. Therefore, if each subtree in $SF(G)$ that is induced by a real edge class is contracted

to some node, then we obtain the forest $2ec(G)$ (up to edge names and node names). (Note that the edges in forest $2ec(G)$ correspond to the edges in $SF(G)$ that are in quasi edge classes.)

From the above observations it follows that each edge class induces a subtree in $SF(G)$.

We consider the insertion of edge (e, x, y). We distinguish the two relevant cases of Subsection 4.2.

If x and y are in different trees of $SF(G)$ (and, hence, are in different components), then these trees need to be linked (corresponding to linking the spanning trees of two connected components if these are joined).

Now suppose x and y are in the same tree T of $SF(G)$ (and hence classes $2ec(x)$ and $2ec(y)$ are in the same tree of $2ec(G)$). Let P be the tree path in T between x and y. We use the terminology of Section 3. By the definition of edge classes, a boundary node of P is either one of the end nodes x or y, or it is a node for which its two neighbours on P are not both in the same 2ec-class as itself. The two neighbours of an internal node z on P are inside class $2ec(z)$ too. Therefore, if we compute the boundary nodes of P only, then we obtain one or two nodes of each 2ec-class (of nodes) that needs to be joined because of inserting (e, x, y).

We need some tree representation to compute boundary sequences efficiently while trees are linked from time to time. One solution is to use rooted trees and, in case of linkings of trees, to redirect the smallest one of the two trees that are linked. However, this takes $O(n. \log n)$ for the linkings. To improve the time complexity, we use the *fractionally rooted trees* structure FRT.

We solve the 2ec-problem by the so-called *2EC structure*, which is given as follows. We use the above forest $SF(G)$ with the 2ec-classes and the above edge classes. A node x in $SF(G)$ that is not in a singleton 2ec-class, has a pointer *assoc* to an edge (in $SF(G)$) that is incident with x and that is in the class named $2ec(x)$. (Such an edge exists.) We call such an edge an *associated edge* for x. Forest $SF(G)$ is implemented as a FRT structure, denoted by FRT_{2ec}. Moreover, all 2ec-classes of nodes (in $SF(G)$) are implemented by a Union-Find structure, denoted by UF_{2ec}. All connected components of nodes are implemented by a Union-Find structure, denoted by UF_c.

A query $2ec\text{-}comp(x)$ now corresponds to a Find call $2ec(x)$.

The initialisation is as follows. For an empty graph consisting of n nodes, the corresponding spanning forest SF is just the collection of nodes. For each node, its pointer *assoc* is set to *nil*. Moreover, each node forms a singleton set in UF_{2ec} and UF_c.

Procedure $insert_{2ec}$ that implements the insert operation for the 2ec-problem is as follows. A call $insert_{2ec}((e, x, y))$ for the insertion of edge (e, x, y) in graph G does

the following. Three cases are distinguished.

1. $c(x) \neq c(y)$. Then the operation $link((e,x,y))$ is performed. Moreover, the two connected components $c(x)$ and $c(y)$ are joined (in UF_c).

2. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. Then $boundary(x,y)$ is performed, returning boundary list BL. All the classes in which the boundary nodes (in BL) are contained, are joined in UF_{2ec}. For each node z in BL the associated edge of z (if any) is obtained by means of pointer *assoc*. Then for each node z in BL that does not have an associated edge yet (i.e. $assoc = nil$), an edge of its sublist (in BL) is related to it as its associated edge (i.e., its pointer *assoc* is set to it). Otherwise, if its (existing) associated edge is not in the same edge class as the edge(s) in the sublist of z (which is tested by means of Finds), the associated edge is inserted in its sublist. The end nodes of BL are removed in case their sublists contain one edge only. Finally, if $BL \neq \emptyset$ then operation $joinclasses(BL)$ is performed.

3. $2ec(x) = 2ec(y)$. Then nothing is done.

In case the initial graph G is not empty at the beginning, the "initial" situation can be obtained e.g. by starting from the empty graph and by inserting all edges of G one at a time by procedure $insert_{2ec}$.

Note that starting from a graph with n nodes, there are at most $2(n-1)$ essential insertions, since in each essential insertion at least two connected components or at least two 2ec-classes are joined, and since initially there are at most n connected components and n 2ec-classes.

Lemma 4.2 *In a 2EC structure for a graph with n nodes, the time needed for a sequence of essential insertions consists of the time for an essential sequence on n nodes in FRT_{2ec}, the time for $O(n)$ Unions in UF_c and UF_{2ec}, the time for at most $O(n)$ nonessential calls boundary in FRT_{2ec}, the time for at most $O(n)$ Finds in UF_{2ec} and UF_c, together with an additional amount of $O(n)$ time. Each nonessential insertion takes $O(1)$ time together with $\theta(1)$ Finds in UF_c and UF_{2ec}.*

Proof. Obviously an essential call $insert_{2ec}$ takes 4 Finds in the Union-Find structures for connected classes and 2ec-classes, together with the time needed for calls *link*, *boundary* and *joinclasses* and for the Unions in UF_{1ec} and UF_{2ec}.

The subsequence of *link*, *joinclasses* and essential *boundary* calls of a sequence of calls of procedure $insert_{2ec}$ yields an essential sequence of operations in FRT_{2ec}, which is seen as follows. Each essential call of procedure $boundary(x,y)$ with output BL is followed by a call $joinclasses(J)$. The list J contains all nodes and edges of BL except for possibly the end nodes x and y, in case their sublists contain one edge

only. Hence, all classes occurring in BL occur in J too if at least 2 classes occur in BL. □

A $2EC(i)$ structure is the above structure where $FRT_{2ec} = FRT(i)$ and where $UF_{2ec} = UF(i)$ and $UF_c = UF(i)$.

Theorem 4.3 *There exists a data structure and algorithms that solve the 2ec-problem and that can be implemented as a pointer/$\log n$ solution such that the following holds. Starting from an empty graph with n nodes, the total time that is needed for all essential insertions is $O(n.i.a(i,n))$, whereas a query and a nonessential insertion can be performed in $O(i)$ time. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. By Theorem 10.1 (for FRT(i)) and [15] (for the complexity of UF(i)), it follows that the initialisation can be performed in $O(n)$ time.

Each nonessential call of *boundary* takes $O(i)$ time. Each Find operation in UF(i) takes $O(i)$ time too. Hence, a query can be performed in $O(i)$ time. By Lemma 4.2, Theorem 10.1 and by the complexity of UF(i) [15], the lemma follows. □

We denote the Union-Find structures UF_{2ec} and UF_c together by UF. We consider the UF structures to be one structure; hence, it is a structure on $O(n)$ elements.

Now take FRT($\alpha(n,n)$) as FRT_{2ec} for a graph with n nodes, where $\alpha(n,n)$ is obtained as in [15], and take for UF the α-UF structure. Then we obtain the following.

Theorem 4.4 *There exists a data structure and algorithms that solve the 2ec-problem and that can be implemented as a pointer/$\log n$ solution such that the following holds. The total time that is needed starting from an empty graph with n nodes is $O(m.\alpha(m,n))$ (where m is the number of edge insertions and queries), whereas the f^{th} operation is performed in $O(\alpha(f,n))$ time if that operation is query or a nonessential insertion. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space.*

Proof. Each query and nonessential insertion corresponds to $\theta(1)$ Finds in the UF structures. Moreover, all essential insertions take at most $O(n)$ Finds. Hence, by [15] the f^{th} operation is performed in $O(\alpha(f',n)) = O(\alpha(f,n))$ time (where $f' = \theta(f)+O(n)$ by Lemma 4.2) if that operation is query or a nonessential insertion. The remaining statements follow by Theorem 10.1 (with $n_e \leq min\{2m,n\}$, where $n_e \leq \{2m,n\}$ is implied by Lemma 4.2, since the part of the graph that is operated on contains at most $min\{2m,n\}$ nodes), Lemma 2.13 (w.r.t FRT($\alpha(n,n)$)), and by [15]. □

The above theorem can be augmented to allow insertion of new nodes in the graph with a time complexity of $O(n + m.\alpha(m,n))$: then α-FRT is used instead of FRT($\alpha(n,n)$) (cf. Section 10). Then n, m, and f in the theorem denote the

current number at the moment of consideration. (Note that only $O(min\{n,m\})$ operations are performed in the α-FRT structure, since these operations are only performed in essential calls of $insert_{2ec}$.)

5 Fractionally Rooted Trees: Observations and Ideas

We give some of the ideas and observations regarding *fractionally rooted trees.* We consider a forest F, with an admissible partition of the edge set.

A tree T in F is partitioned into subtrees that all are (locally) rooted, i.e., each subtree has its own root independent of the remainder of the tree and subtrees. (The subtrees are independent of the admissible partition of the edge set.) Each subtree is contracted to a new node, which yields a contracted tree T'. The collection of edges of T' is partitioned into edge classes induced by the edge classes of T, where an induced edge class in T' consists of the contraction edges of the edges in a certain edge class in T.

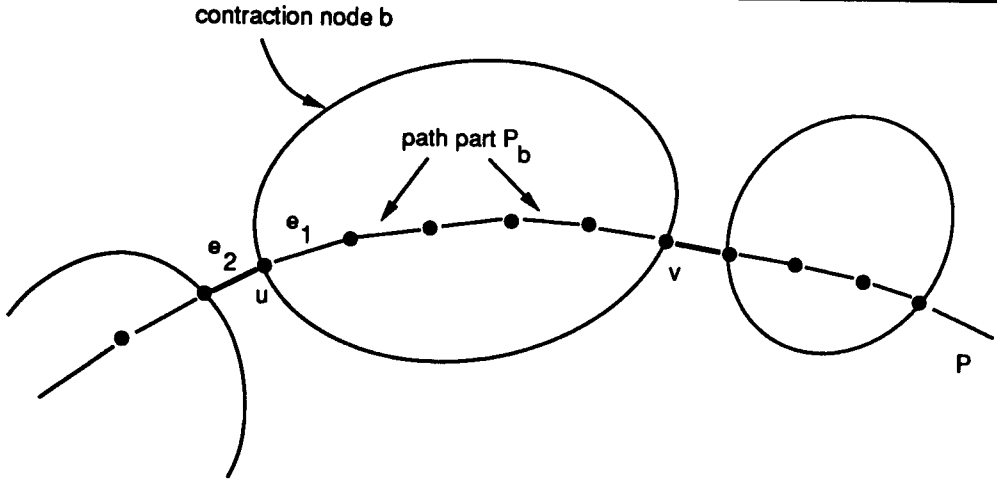
A boundary list B between two nodes x and y in T can now be obtained as follows.

Let c and d be the nodes in T' to which x and y are contracted respectively. If $c = d$, then the tree path between x and y in T is entirely inside contraction node c. Therefore, we assume $c \neq d$. Let P be the tree path between x and y in T. Let P' be the tree path between c and d in T'. Consider an internal node b of P'. Then the edges of P' that are incident with b are in the same class. Hence, the originals of these edges (in T) are in the same edge class and, since an edge class induces a subtree, all edges on P that are contained in contraction node b are in that edge class too. Hence, all the nodes on P that are contained in b are internal nodes of P. On the other hand, for each boundary node b of P', there is a boundary node of P that is contained in b. For, either an end node of P is contained in b or the edges of P' (in T') that are incident with b are in the different classes. In the latter case, the originals of these two edges (on P) are in different edge classes, and hence there is at least one node of P contained in b of which the two incident edges of P are in different classes. Therefore, each boundary node of P is contained in a boundary node of P' and each boundary node of P' contains a boundary node of P.

Now, suppose that b is a boundary node of P'. Consider the part P_b of P inside contraction node b. We consider the relation between boundary nodes in P_b and P (see Figure 4). Trivially, a boundary node of P that is contained in P_b is a boundary node of P_b too. Now, let z be a boundary node of P_b. Let the end nodes of P_b be u and v. If $z \notin \{u,v\}$, then z is a boundary node of P too, and a boundary edge set for z on P_b is a boundary edge set for z on P. If $z \in \{u,v\}$ and $u \neq v$, then z is an end node of P_b and hence a boundary edge set for z on P_b contains only one

edge, say edge e_1. Let e_2 be the original of the edge in a boundary edge set for b on P' that is incident with z, if e_2 exists (i.e., if $z \notin \{x, y\}$). If e_2 exists, and if e_1 and e_2 are in the same edge class, then z is an internal node of P. Otherwise, z is a boundary node; then a boundary edge set for z on P_b extended with e_2 (if e_2 exists) is a boundary edge set of z for P. Finally, if $z = u = v$ then P_b consists of node z only. Hence, a boundary edge set for z $(= u = v)$ on P_b consists of z with an empty sublist. In that case the original(s) of the edge(s) in a boundary edge set for b on P' form a boundary edge set for z on P. (Since otherwise, b would not be a boundary node of P'.)

Figure 4: Considering a part P_b of P.



Hence, we can follow the following strategy. First we compute a boundary list B' in T' for the nodes c and d. Then, for each boundary node b in B', we obtain the above nodes u and v as follows: if $b \notin \{c, d\}$ then u and v are the nodes that are contained in b and that are end nodes of the originals of the two edges in the sublist of b in B'; otherwise, if $b = c$, then $u = x$ and v is the node that is contained in b and that is an end node of the original of the edge in the sublist of b; if $b = d$, then we have the same situation for y. Subsequently we compute the "local" boundary list $bl(b)$ for u and v. (Note that this can be computed inside the subtree that is contracted to b only.) Finally, we consider the end nodes u and v like above: if u (or v) is not a boundary node after all, then it is removed from $bl(b)$, and otherwise, its sublist, containing boundary edge sets, is adapted like above. Then these local boundary lists $bl(b)$ for $b \in B'$ are concatenated in the same order as that the corresponding contraction nodes b occur in B'.

We will present fractionally rooted trees and algorithms for maintaining them that

are based on the above observations.

6 Division Trees

6.1 Description of the Data Structure and the Operations

Division trees form an essential part of the fractionally rooted trees. For the terminology regarding contractions we refer to Section 2.1.

Let F be a forest with an admissible partition of the edge set into edge classes. Henceforth we call these edge classes *global edge classes* (to distinguish them from other, local, edge classes that will be defined below).

Let T be a tree in F. Then T together with a set $CN(T)$ of new nodes, called contraction nodes, and with a set $nodes(b)$ of nodes in T for each $b \in CN(T)$ is called a *division tree* if the sets $nodes(b)$ for $b \in CN(T)$ partition the node set of T into disjoint subsets and if each set $nodes(b)$ induces a subtree of T, denoted by $tree(b)$. A subtree $tree(b)$ is called an *elementary subtree* of T. (Hence, each elementary subtree can be considered to be contracted to a unique contraction node in $CN(T)$.)

The *contraction tree* $CT(T)$ *of a division tree* T (with the sets as described above) is the tree with node set $CN(T)$ and with the edge set being the set of corresponding contraction edges (hence, consisting of the edges (e, c, d) such that $c \neq d$ and there exists an edge (e, x, y) with $x \in nodes(c)$ and $y \in nodes(d)$).

For a subtree $tree(b)$ of T we define the set of *external edges of* $tree(b)$ as the edges of T that are incident with exactly one node of $tree(b)$. (Note that if $tree(b) = T$, then there are no external edges.) We define the *extended tree of* $tree(b)$, denoted by $extree(b)$, as the tree $tree(b)$ extended with its external edges. Note that an extended tree is therefore not a tree in the usual sense, since only one of the end nodes of an external edge is in the tree. However, we will still apply the regular tree notions on the nodes and edges in an extended tree, where if necessary the lacking end nodes can be thought to be present in the extended tree, though. E.g., if $tree(b)$ is rooted, then the father relation is extended to $extree(b)$ by taking for the father node of an external edge its unique end node that is in $tree(b)$. Moreover, note that each node is contained in exactly one extended elementary subtree.

An edge that is contained in some elementary subtree $tree(b)$ is called an *internal edge of* T. An edge in tree T that is not contained in any elementary subtree is an external edge of two elementary subtrees and, hence, is contained in two extended subtrees (namely, in the two extended subtrees corresponding to the two contraction nodes in which the end nodes of that edge are contained). These edges are called the *external edges* of T.

Let S be an extended elementary subtree of T. The edge set of S is partitioned into the edge classes as follows. The edge classes of S are the nonempty intersections of the global edge classes of T with the set of edges of S. It is easily seen that the edge classes of S form an admissible partition for S. We sometimes call these edge classes local edge classes, in particular if we consider these classes in general (i.e., not in the context of some extended subtree).

We describe some further aspects of the division trees.

A tree T in F is implemented in the common way: each node has an incidence list, consisting of (pointers to) the edges of which it is an end node. Each node x in T contains a pointer $contr(x)$ to the contraction node $b \in CN(T)$ in which it is contained (i.e., for which $x \in nodes(b)$), and, conversely, for each contraction node $b \in CN(T)$, the set $nodes(b)$ is implemented as a list (which we denote by $nodes(b)$ too). An edge contains a status field indicating whether it is external or internal. (Note that it can also be determined without status field whether an edge is external or not, viz., by checking whether the end nodes of the edge are contained in the same contraction node by comparing the *contr* pointers of these nodes.)

Note that each internal edge is in exactly one (extended) subtree, while each external edge is contained in exactly two subtrees. The operations that may be applied on division trees (as described in the sequel) may change edges from external to internal, but not the other way around. Moreover, an external edge may contain different information pertaining to the two extended subtrees in which it is contained. This is implemented as follows. Each edge has two representatives called *edge sides* (or just: sides), one for each of its end nodes (e.g. implemented as two records pointed at from the edge, or two dedicated blocks of fields in the edge). The side of edge (e, x, y) for end node x is denoted by $(e, x, y)_x$ (and similarly for y). For an external edge (e, x, y), the side for end node x is the representative of e in the extended subtree in which x is contained. Hence, if (e, x, y) is considered inside $extree(b)$, then the appropriate side is the side for the end node $z \in \{x, y\}$ with $contr(z) = b$, and hence it can be obtained by comparing $contr(x)$ and $contr(y)$ with b. For internal edges, both the sides are considered to be identical representations for the same subtree (hence, $(e, x, y)_x = (e, x, y)_y$), where only one of them is taken (and distinguished) to be the actual (and active) representative. Instead of speaking of "side $(e, x, y)_x$" we will also speak of "edge (e, x, y) w.r.t. x".

In the sequel the (local) edge classes for each extended subtree are implemented by Union-Find structures (which is described below). For an external edge, the appropriate side for the extended tree is used. An internal edge, according to the above implementation, actually "occurs" two times in a Union-Find structure, namely by both its sides, of which one is a dummy and is not used explicitly in the structure. (This avoids the presence of a "remove" operation in a set in the Union-Find structure.)

The extended subtrees in a division tree have the following additional implementa-

tion.

Let b be a contraction node. We consider $extree(b)$. Extended tree $extree(b)$ is rooted at some node. Each node in the extended tree has a pointer to its father node (if any) and to its father edge (if any).

Every edge class contains at most one edge that is marked by a so-called *c-mark*, which is an external edge. The edge classes in $extree(b)$ are represented by a Union-Find structure (according to the above representation method with sides), called the *local class Union-Find structure*. The class of edge e in $extree(b)$ is denoted by $class(e)$ (which corresponds to a Find). (Note that actually we have to give the appropriate side of e w.r.t. $extree(b)$ as parameter of *class*. We will often omit this if it is clear for which extended tree the edge is considered.) There are the following pointers w.r.t. classes.

- For each edge class C in $extree(b)$ there are the following pointers:
  - pointer $max(C)$ to a maximal edge of C in the rooted tree $extree(b)$ Such an edge is called the maximal edge of that class. It is marked by an *m-mark*.
  - pointer $ext(C)$ to an external edge in it (if there exists any).
  - pointer $edge(C)$ to the c-marked edge in it (if it exists).

  These pointer are stored in (the record representing) the name of the class C.
- every c-marked edge $e \in extree(b)$ contains a pointer $c(e)$ to the name of the class in which it occurs.

Note that for a node x in $extree(b)$ and for an edge class C in $extree(b)$ that contains an edge incident with x, the father edge of x is in C, or the (unique) m-marked edge in C (which is the edge to which $max(C)$ points) is incident with x.

Also, note that the global edge classes of forest F are not implemented and therefore only conceptually exist in a division tree: i.e., there is no Union-Find structure present for the global edge classes in a division tree. (However, note the global edge classes can be obtained from the local edge classes if all local edge classes that contain a common edge are joined.)

We describe the operations that we want to perform on F.

basic-external-link((e,x,y)): Let x and y be nodes in two different trees T_x and T_y. Then link these trees by the edge (e, x, y), yielding tree T, where the partition of the node set remains unchanged. This means that $CN(T) = CN(T_x) \cup CN(T_y)$ and for each $b \in CN(T)$, the set $nodes(b)$ is not affected by the operation. The new edge (e, x, y) (which is hence an external edge) forms a new singleton class on its own in the extended trees in which it is contained.

basic-internal-link((e,x,y),y): Let x and y be nodes in two different trees T_x and T_y. Let $c = contr(x)$. Then link these trees by the edge (e, x, y), yielding tree T, where the elementary subtree $tree(c)$ is extended with the (internal) edge (e, x, y) and with the tree T_y. I.e., $CN(T) = CN(T_x)$ and all sets $nodes(b)$ for $b \in CN(T_x)$ remain unchanged except for $nodes(c)$ that is augmented with the nodes of T_y. The new edge (e, x, y) (which is hence an internal edge) forms a new singleton class on its own in the extended tree in which it is contained.

basic-integrate(x, f): Let x be a node in tree T and let f be a (possibly new) contraction node not occurring in $CN(T)$. Then change the partition of T such that it consists of precisely one elementary subtree with contraction node f (hence, T itself). I.e., afterwards $CN(T) = \{f\}$ and $nodes(f)$ contains (at least) all the nodes of T.

basic-boundary(x, y): Let x and y be in the same elementary subtree S. Then return a boundary list BL for nodes x and y in S, where each edge in the sublist of a node in BL either is the father edge of that node or it is m-marked in S.

basic-joinclasses(J): Let J be a joining list containing precisely one node and such that there is at most one edge class occurring in J that contains a c-marked edge. Then join the edge classes of which an edge occurs in the list.

Note that elementary subtrees are changed in case of a call *basic-integrate* or *basic-internal-link*. (For, the partition of the node set in subsets $nodes(b)$ is altered.) Therefore we call an edge *affected* by an operation, if for the extended trees $extree(b)$ and $extree(b')$ in which it is contained before and after such a procedure call respectively, $b \neq b'$ holds. (Note that all edges in the tree on which *basic-integrate* is performed, are affected then. For affected edges, the father relations and m-marks of these edges (edge sides) may change during these calls.)

We show how to initialize a forest with an admissible partition on its edge set as a forest of division trees, where each division tree contains exactly one elementary subtree, namely the tree itself. We suppose that a list of nodes in the forest is present and a collection of lists, one for each edge class, where a list contains exactly the edges in that edge class. First for each tree T create a new contraction node c, create two sides for each edge occurring in the tree, make the tree rooted (together with the father relation on the nodes). The contraction pointers of all the nodes are set to c and the nodes are put in the list $nodes(c)$. All edges are un-m-marked and un-c-marked. Then for each list do the following. Initialise a set in the Union-Find structure consisting of all the edges in the list. Then detect an edge in the set that is maximal in the tree in which it is contained: first mark all edges in the set, then detect for each edge whether its father edge (if any) is in the set too, i.e., it is marked too, and if not (i.e., the father edge does not exist or it is not present), then

that edge is a maximal edge. In this way a maximal edge for the set is selected and m-marked and a pointer to it is stored in field max of the name of the class. The other fields ext and $edge$ in the set name are set to nil. Note that all this can be performed in linear time provided that the sets can be initialised in linear time in the Union-Find structure that is used.

We summarize the pointers and representations. For a tree T in F we have the following.

- Each node x in a tree T contains the following information, where $b = contr(x)$ $(b \in CN(T))$:
  - an incidence list, consisting of (pointers to) the edges of which it is an end node.
  - a pointer $contr(x)$ to node b
  - a pointer to its father node (if any) and to its father edge (if any) in $extree(b)$
- For each contraction node $b \in CN(T)$, the set $nodes(b)$ is implemented as a list (denoted by $nodes(b)$ too).
- An edge (e, x, y) in T contains the following:
  - a status field indicating whether it is external or internal.
  - two edge sides (being its representatives), one for each of its end nodes: $(e, x, y)_x$ and $(e, x, y)_y$.
  - side $(e, x, y)_x$ contains a field for the local-class Union-Find structure representing the edge classes in $extree(contr(x))$. (Similarly for y.)
  - if (e, x, y) is c-marked in $extree(contr(x))$, then $(e, x, y)_x$ contains a pointer $c(e)$ to the name of the class in which it occurs. (Similarly for y.)
- For each edge class C in $extree(b)$ for some $b \in CN(T)$, there are the following pointers:
  - pointer $max(C)$ to the m-marked edge of C.
  - pointer $ext(C)$ to an external edge in it (if there exists any).
  - pointer $edge(C)$ to the c-marked edge in it (if it exists).

  These pointer are stored in (the record representing) the name of the class C.

6.2 Implementation of the Operations

The operations are implemented as follows. We give the computations and we intermix it with comments. (This subsection may be skipped at first reading.)

basic-external-link((e,x,y)): First edge (e, x, y) is inserted as an edge between x and y: i.e., the edge in inserted in the incidence list of both its end nodes x and y. Then the two sides of the new edge (e, x, y) are both inserted as singleton sets in the local class Union-Find structure. For both the sides, the pointers max and ext are set to edge (e, x, y) itself. The sides are m-marked.

basic-internal-link((e,x,y),y): Let $c = contr(x)$. First the operation *basic-integrate*(y, c) is performed. (Note that now y is the root of the tree in which it is contained.) Then the operation *basic-external-link*(e, x, y) is performed. The two singleton classes consisting of the edge sides of (e, x, y) are joined, yielding class C (by performing a Union on the output of the Finds on the sides). Then edge (e, x, y) is converted to internal and it is made the father edge of node y. Make x the father node of y. (Note that since y is the root of its tree, converting (e, x, y) to internal and making x the father of y yields that the new resulting tree $tree(c)$ is rooted again.) The pointers $max(C)$ and $ext(C)$ are set to (e, x, y) and to nil respectively. Finally, the pointer $edge(C)$ is set to nil and edge (e, x, y) is m-marked.

basic-boundary(x, y): Note that x and y are in the same elementary subtree. If $x = y$ then return the boundary list BL consisting of node x with an empty sublist. Otherwise, $x \neq y$ and the following is done. The boundary list BL is obtained as follows. First the boundary nodes (together with boundary edge sets) for the root paths of x and y are partially computed: viz., two boundary lists $s(x)$ and $s(y)$ are computed as follows. The two lists $s(x)$ and $s(y)$ start with x and y with empty sublists respectively. Then the two lists $s(x)$ and $s(y)$ are stepwisely computed in an alternating way until a node *top* has been visited by both computations. A computation step for sequence $s(x)$ (or $s(y)$) is as follows: obtain the father edge (e, z, z') of the last node z in the sequence, (if any, otherwise skip the rest of the step), insert the edge in the sublist of z, obtain the edge $max(class(e)) = (e', u, v)$ and obtain the father node of e' (being u or v); then insert the father node at the end of the list and insert edge (e', u, v) in its sublist. (The stop condition can be checked by marking all nodes that are visited: it becomes true if a node is visited that is already marked. After the traversals the nodes are unmarked. Cf. [18].) (Note that now $s(x)$ and $s(y)$ are boundary lists for their end nodes.) (It follows that each edge in the sublist of a node is either its father edge or it is an m-marked edge.)

Adapt the lists as follows: remove all nodes in the lists occurring after *top* and remove the father edge of *top* from its sublists (if present).

Now $s(x)$ and $s(y)$ are boundary lists for x and *top* and for y and *top* respectively. Hence, both $s(x)$ and $s(y)$ contain the boundary nodes (together with boundary edge sets) for the paths between x and *top* and between y and *top* respectively. Moreover, note that all their nodes, except for possibly node *top*, are on the the path P between x and y and hence are boundary nodes for P. So it is left to verify whether *top* is a boundary node for P. If $top \in \{x, y\}$ then *top* is a boundary node of P. Otherwise, each of the two sublists of *top* (in $s(x)$ and $s(y)$) contains exactly one edge. If the two edges in these sublists are in the same edge class, then *top* cannot be a boundary node of P. Otherwise, if they are not in the same edge class, then *top* is on P and hence it is a boundary node of P, where the two edges form a boundary edge set. This is the observation justifying the following part of the computations.

If each of the two sublists of *top* (in $s(x)$ and $s(y)$) contains exactly one edge, and if these two edges are in the same edge class, remove *top* from both its lists. Otherwise, extend the sublist of *top* in $s(x)$ with the sublist of *top* in $s(y)$ and remove *top* from $s(y)$. Then the boundary list BL is created by appending the reversed list $s(y)$ to the list $s(x)$.

basic-joinclasses(J): Let J be a joining list consisting of precisely one node for some subtree S. For all edges in J, the corresponding classes must be joined yielding one new class C.

First a list CJ is created consisting of all (names of) edge classes occurring in J. (This is done by performing a Find operation *class* on each edge in the list: for each edge (e, x, y) in the sublist of node x in J, obtain its class name $class((e, x, y)_x)$.)

We compute a maximal edge e_m of the (future) new class C as follows. For each class name c in CJ, obtain the maximal edge $max(c)$ in its class. Check whether the class of the father edge $(e, x, y,)$ of x occurs in CJ (which can be done by marking the class names occurring in CJ). If this is the case, then e_m is the maximal edge of that class. Otherwise, e_m is any of the maximal edges obtained above.

Subsequently, the (unique) c-marked edge e_c that is contained in one of the classes in CJ is selected (if it exists). Moreover, one of the external edges of the classes in CJ (if any) is selected as edge e_{ex}.

Join the classes in CJ, resulting in one new class. Edge e_c is related to c as c-marked edge: i.e., $c(e_c)$ is set to point to the name of the new class C (which is obtained by performing a Find operation $class(e_c)$) and $edge(C)$ is set to point to e_c. An external edge is related to the resulting class C by setting $ext(C)$ to e_{ex}.

The m-markings are updated as follows: all maximal edges obtained above are un-m-marked except for edge e_m. (Remark that a list containing these edges that are un-m-marked can easily be returned by the procedure, if wanted.) Then $max(C)$ is set to point to e_m.

basic-integrate(x,f): Let T denote the tree in which x is contained. First x is taken as the root of T and the father pointers of all nodes (to the resulting father nodes and father edges) are adapted accordingly. Moreover, the pointers *contr* of the nodes are set to f and the nodes are put in list $nodes(f)$. For each external edge e, the two classes in which its sides are contained are joined. The external edges of T are set to internal and are (hence) un-c-marked. Then $c(e) := nil$ for all the processed edges and $ext(C) := edge(C) := nil$ for all occurring classes C (since all edges in T are internal now). Moreover, all edges are un-m-marked and for all occurring classes the pointer max is set to nil. (All this can be performed during a tree traversal algorithm.) Next, maximal edges are related to the edge classes by checking for each edge e whether its father edge is in the same class too: if this is not the case, and if $max(class(e)) = nil$ then the pointer $max(class(e))$ is set to e and e is m-marked. (There may be several candidates for one class: then after the first candidate the max-pointer is not nil and hence no further changes occur.) (This can be performed during a tree traversal algorithm.)

Finally, note that because of the insertion of edges Union-Find structures must allow the insertions of elements. However, since the number of edges is less than the number n of nodes in the forest, this can be implemented by using $2(n-1)$ "free" records, where 2 such free records are associated to an inserted edge (or: its edge side) as its representatives w.r.t. the Union-Find structure. Then, (with a fixed number of nodes) no insertions in the Union-Find structures are needed.

7 Fractionally Rooted Trees: the Data Structure

We now present the data structure called the fractionally rooted tree.

We consider a dynamic forest F_0 with an admissible partition of its edge set into (global) edge classes. The edge classes in F_0 are represented by a Union-Find structure denoted by UF_0. A Find in UF_0 on a edge $e \in F_0$ (to obtain the name of its edge class) is denoted by $class_0(e)$.

Let $i \geq 1$. Let F_i be a forest consisting of trees that are contraction trees of trees in F_0, where each tree in F_0 has at most one contraction tree in F_i, but where not for all trees in F_0 a contraction tree needs to be present in F_i (already). (In that case F_i can be extended (from time to time) with a singleton tree being the contraction

of such a tree in F_0.) The edge set of forest F_i is partitioned into the edge classes that are induced by the edges classes of F_0

We introduce the structures FRT(i) for F_i for $i \geq 1$.

Each tree of F_i has a name in FRT(i), being some (new) unique node. We denote the tree in F_i that has the name s in FRT(i) by $tree_i(s)$ and we denote the corresponding original tree in F_0 by $tree_0(s)$. The FRT(i) structure consist of a collection of so-called tree structures , one for each occurring tree name (i.e., for each occurring tree in F_i). A tree structure consists of a tree name s and a collection of at most i layers, numbered from i in a decreasing order (say, down to $down(s)$). Each existing layer j ($down(s) \leq j \leq i$) consists of a division tree, denoted by $tree(s,j)$. For layer i, $tree(s,i)$ is the $tree_i(s)$ represented as a division tree. The tree $tree(s,j)$ in an existing layer j ($down(s) \leq j \leq i-1$) is the contraction of the division tree $tree(s,j+1)$ in layer $j+1$. (Hence, $tree(s,j)$, with $down(s) \leq j \leq i$, is a contraction tree of $tree_0(s)$ too.) Each edge in a tree $tree(s,j)$ has a pointer $orig_0$ to its original in F_0, which is called its 0-original. The tree name s forms the contraction tree of the division tree $tree(s, down(s))$ stored in layer $down(s)$. Tree name s contains a pointer *contr* being *nil*. (The above number $down(s)$ is only used in the above description and will not be used in the data structure itself.)

To each tree name some parameters are associated and the corresponding tree structure satisfies additional constraints w.r.t. these parameters, which will be given in the sequel.

The collection of tree structures is changed by operations that are given in the sequel.

Note that from the above description the following follows.

Firstly, the trees stored in layer i of FRT(i) (i.e., the trees $tree(s,i)$) form the forest F_i. Hence, two edges in a tree $tree(s,i)$ are in the same global edge class (in F_i) iff their 0-originals in F_0 are in the same global edge class.

Secondly, all the nodes in the tree structure for tree name s contain a pointer field *contr*. For tree name s pointer $contr(s)$ is *nil*. For an existing layer j ($down(s) \leq j \leq i$) a node x in layer j the pointer $contr(x)$ either points to a node in layer $j-1$ (the contraction node in which x is contained) if layer $j-1$ exists, or it points to tree name s otherwise. Moreover, for a node x, $nodes(x)$ is the list of the nodes y for which the pointer $contr(y)$ points to x (i.e., it represents the set of nodes that are contracted to x).

Consider a structure FRT(i) for forest F_i. The structure FRT(i) allows the following operations on the nodes and edges of F_i:

treename(x): x is a node. Then output the name s of the tree in which node x occurs (i.e., for which $x \in tree(s,i)$).

link$((e,x,y),s,t,i)$: s and t are tree names, $s \neq t$, $x \in tree(s,i)$ and $y \in tree(d,i)$. Then link $tree(c,i)$ and $tree(d,i)$ by the edge (e,x,y), where edge (e,x,y) forms a new singleton class. Update the structure.

boundary(x,y,i): Let $x \neq y$ and $x,y \in tree(s,i)$ for some tree name s. Then output a boundary list BL for nodes x and y in $tree(s,i)$.

joinclasses(J,i): Let J be a joining list. Then update the structure according to the joining of the global edge classes occurring in the list.

candidates(x,y,i): Let x and y be two nodes, $x \neq y$. Return an edge e_x incident with x and an edge e_y incident with y such that these edges are in the same global edge class if such edges exist. Moreover, e_x is the father edge of x, or e_x is m-marked w.r.t. x, and similar for e_y and y.

(Note that the above correspondence between x and s and between y and t in procedure *link* means that we can make distinction between the "first" node and the "second" end node of edge (e,x,y). We can formalize this by adding new parameters containing x and y in the procedure heading. However, we will not do this here.)

Operation $treename(x)$ is given by: if $contr(x) \neq nil$ then return $treename(contr(x))$, otherwise return x. Obviously (from the above description), $treename(x)$ outputs the name of the tree in which node c is contained. The other operations will be given in the sequel.

The structures FRT(i) are defined inductively in a way similar to [15]. We start from a base structure FRT(1) that corresponds to the idea using ordinary rooted trees. This structure takes $O(n.\log n)$ time for an essential sequence of operations.

7.1 The Structure FRT(1)

Structure FRT(1) is a structure for a forest F_1 that satisfies the following conditions. (Recall that a tree in F_1 with name s is denoted by $tree(s,1)$ and that $tree(s,1)$ is in layer 1, where $CN(tree(s,1)) = \{s\}$. (The entire tree $tree(s,1)$ is "contracted" to node s, the name of the tree.))

The Union-Find structure for local classes in F_1 is UF(1).

For each tree name s we have a parameter $weight(s,1)$ that contains the number of nodes in $tree(s,1)$: $weight(s,1) = |tree(s,1)|$ (Note that we count the nodes of $tree(s,1)$, cf. Notation 2.1.)

We give the algorithms for the operations.

link$((e,x,y),s,t,1)$: The trees $tree(s,1)$ and $tree(t,1)$ must be linked by edge (e,x,y). W.l.o.g. suppose that $weight(s,i) \leq weight(t,i)$. (Otherwise interchange s,x and t,y in the description below.) Then *basic-internal-link*$((e,x,y),x)$ is performed.

boundary$(x,y,1)$: The boundary list BL is obtained by a call *basic-boundary*(x,y).

joinclasses$(J,1)$: The joining of classes is performed by the calls *basicjoinclasses*(J_x) for each node x in J, where J_x consists of x and its sublist in J.

candidates$(x,y,1)$: Note that $x \neq y$. Let e_x be the father edge of x and let e_y be the father edge of y. Obtain the edges $m_x := max(class(e_x))$ and $m_y := max(class(e_y))$. If m_x is incident with y then $e_y := m_x$ (and then e_y is m-marked for y) and if m_y is incident with y then $e_x := m_y$ (and then e_x is m-marked for x). Output the edges e_x and e_y. (Now e_x is either the father edge of x or it is m-marked for x and similar for e_y and y.)

Procedure *candidates*$(x,y,1)$ yields a correct pair of edges, since if x and y are incident with edges of the same edge class C, then either the father edge of x is in C or the maximal edge of C is incident with x. The same holds for y. Moreover, at least one of the father edges of x and y must be in C (if $x \neq y$).

If FRT(1) is used directly on F_0 (i.e., $F_1 = F_0$ and hence $tree(s,i) = tree_0(s)$ for all tree names s), and hence inside an environment not being FRT(2), then $UF_0 = UF(1)$ (i.e., the global edge classes on F_0 are implemented by a Union-Find structure UF(1)).

If FRT(1) is used directly on F_0 (i.e., $F_1 = F_0$), then the initialisation for some (sub-)collection of nodes in singleton trees is as follows. Relate a tree name s to each singleton tree. For each node x with name s for the singleton tree consisting of x, the following initialisation is performed: $contr(x) = s$, $nodes(s) = \{x\}$, $weight(s,1) = 1$. (Note that the insertion of a singleton set consisting of a newly created node can easily be performed in this way too.) If we want to initialise the structure for some a forest F_0 not necessarily consisting of singleton trees, where there is a list of the names of the existing edge classes and for each name there is a sublist with the edges in the corresponding class, then this can be performed as follows. First the forest is initialised as a forest of division trees, where each division tree contains exactly one elementary subtree, viz. the tree itself. This is done in the way described in Section 6. Hence, for each tree there is exactly one (new) contraction node. Then the contraction node s is taken to be the tree name in FRT(1) and $weight(s,1) =$ [the number of nodes in the tree].

7.2 The Structure FRT(i) for i>1

Let $i > 1$. Structure FRT(i) is a structure for a forest F_i that satisfies the following conditions. (Recall that a tree in F_i with name s is denoted by $tree(s,i)$ and that $tree(s,i)$ is in layer i.)

The Union-Find structure for local classes in F_i is UF(i).

For each tree name s we have a parameter $weight(s,i)$ that contains the number of nodes of $tree(s,i)$: $weight(s,i) =\mid tree(s,i) \mid$. Also, we have a parameter $lowindex(s,i)$ which is an integer ≥ -1 that satisfies

$$2.A(i, lowindex(s,i)) \leq weight(s,i). \tag{4}$$

(The parameter *lowindex* is incremented from time to time by the algorithms.)

Two cases are distinguished.

- If $tree(s,i)$ consists of precisely one node x (i.e., $weight(s,i) = 1$) then $CN(tree(s,1)) = \{s\}$ (I.e., then $contr(x) = s$, $nodes(s) = \{x\}$.) (Hence, layer $i-1$ does not exist in tree structure s.)

- Otherwise, if $tree(s,i)$ contains more than one node (i.e., $weight(s,i) > 1$), then recall that $tree(s,i)$ is a division tree.

  A contraction node $b \in CN(tree(s,i))$ satisfies (besides $\mid cluster(b) \mid \geq 2$)

  $$\mid nodes(b) \mid \geq 2.A(i, lowindex(s,i)). \tag{5}$$

  The contraction tree of the division tree $tree(s,i)$ is tree $tree(s,i-1)$ in layer $i-1$. (Hence, for each external edge $(e,x,y) \in tree(s,i)$ there exists a contraction edge (e,c,d) in layer $i-1$ with $c = contr(x)$ and $d = contr(y)$.) The global edge classes in tree $tree(s,i-1)$ are the edge classes induced by the global edge classes of $tree(s,i)$ (and hence induced by the global edge classes of $tree_0(s)$.)

  If layer i is removed then the remaining part, starting from $tree(s,i-1)$ in layer $i-1$, is a FRT($i-1$)-structure. (Where hence $tree(s,i-1)$ is a division tree with edge classes induced by $tree_0(s)$.)

  For an external edge (e,x,y) in $tree(s,i)$ we have the following. Let $c = contr(x)$ and $d = contr(y)$. Then the contraction edge (e,c,d) contains a pointer *orig* to its original edge (e,x,y) in $tree(s,i)$ (besides the pointer that this edge contains to its 0-original in F_0). The side $(e,x,y)_x$ (i.e., the side for x) is *c-marked* if the edge (e,c,d) is the father edge of c or if the edge side $(e,c,d)_c$ is m-marked.

Note that every edge class in $extree(b)$ for some $b \in CN(tree(s,i))$ now contains at most one c-marked edge, which is seen as follows. Let (e,x,y) be a c-marked edge in $extree(b)$, where $contr(x) = b$ and $contr(y) = c$. Let (e,x,y) be contained in class C of $extree(b)$. Then either edge (e,c,d) is the father edge of contraction node c or the edge side $(e,c,d)_c$ is m-marked. By applying the observations of Section 6 to $tree(s,i-1)$, there is not another edge in the local edge class of $(e,c,d)_c$ in $tree(s,i)$ that is incident with c and that has one of these two properties. Hence, there is not another c-marked edge in class C.

We give the algorithms for the operations (intermixed with comments). Note that, by (4), $lowindex(s,i) \geq 0$ implies that $tree(s,i)$ consists of at least 2 nodes and hence there exists a contraction node c at layer $i-1$ (hence, $c \neq s$).

link$((e,x,y),s,t,i)$: The trees $tree(s,i)$ and $tree(t,i)$ must be linked by edge (e,x,y). W.l.o.g. we assume that $lowindex(s,i) \geq lowindex(t,i)$. (Otherwise interchange x,s and y,d in the description below.)

Let $newweight := weight(s,i)+weight(t,i)$ and let $ls := lowindex(s,i)$. Then set $weight(s,i) := weight(t,i) := newweight$. There are three cases.

- $lowindex(s,i) > lowindex(t,i)$. Let $c := contr(x)$. (Then $c \neq s$, since we have $lowindex(s,i) \geq 0$. Hence, c is a node on layer $i-1$.) The following is done. Then a call $basic\text{-}internal\text{-}insert((e,x,y),y)$ is performed (yielding the extension of $subtree(c)$ with edge (e,x,y) and with $tree(t,i)$ and where all nodes the contain a pointer $contr$ to c) and the old existing layers j with $j < i$ for tree structure t are disposed, together with name t itself.
- $lowindex(s,i) = lowindex(t,i) \wedge newweight \geq 2.A(i,ls+1)$. Then a *new* contraction node f is created in layer $i-1$. Then a call $basic\text{-}external\text{-}insert(e,x,y)$ is performed and subsequently a call $basicintegrate(r,f)$ for some arbitrary node in the tree (e.g. $r = x$). The old existing layers j of tree structures s and t with $j < i$ are disposed including tree name t. The tree name s is taken to be the name of the resulting tree: $contr(f) := s$. Finally, $lowindex(s,i) := lowindex(s,i)+1$, $weight(s,i-1) := 1$ and $lowindex(s,i-1) := -1$. (Note that now the subtree $subtree(f)$ consists of $tree(s,i)$ and $tree(t,i)$ together with linking edge (e,x,y).)
- $lowindex(s,i) = lowindex(t,i) \wedge newweight < 2.A(i,ls+1)$. Then $basic\text{-}external\text{-}insert((e,x,y))$ is executed. (Hence, edge (e,x,y) is inserted as an external edge between x and y.)

  Let $c = contr(x)$ and $d = contr(y)$. (Then $c \neq s$ and $d \neq t$ since $0 \leq newweight < 2.A(i,ls+1)$ implies $ls \geq 0$. Hence, c and d are nodes on layer $i-1$.) A new edge (e,c,d) is created. Then $orig(e,c,d) := (e,x,y)$ and $orig_0(e,c,d) := orig_0(e,x,y)$. Subsequently a recursive call

$link((e,c,d),s,t,i-1)$ is performed. (This is to link the contractions $tree(s,i-1)$ and $tree(t,i-1)$; then one of the above cases occurs on a layer j with $j < i$.) Then all the affected edges in layer $i-1$ are obtained (i.e., the edges processed by a call *basic-integrate* or *basic-internal-insert* on layer $i-1$, which hence may change the father relations and m-marks of these edges). (Note that these edges can easily be obtained by having the recursive call $link(i-1)$ returning a list of all these edges, where hence the same must be done by calls *basic-integrate* and *basic-internal-insert*.)

For each original edge in layer i of an affected edge in layer $i-1$ and for edge (e,x,y), the following is done to update the c-marks. Let (e',u,v) be the considered edge and let (e',a,b) be its contraction edge with $a = contr(u)$ and $b = contr(v)$. If $(e',a,b)_a$ is m-marked or if it is the father edge of node a, then c-mark the edge side $(e',u,v)_u$, obtain its edge class $k = class((e',u,v)_u)$ and set pointers $c((e',u,v)_u) := k$ and $edge(k) := (e',u,v)$. Otherwise, un-c-mark $(e',u,v)_u$. The same is done for edge side $(e',u,v)_v$. (Note that now an edge class k' cannot have an *edge*-pointer left to an ex-c-marked edge, since an edge class that contains an external edge always contains a c-marked edge and hence its *edge*-pointer is set to that edge.)

boundary(x,y,i): The boundary list BL is obtained as follows.

Perform $candidates(x,y,i)$ yielding edges e_x and e_y. If $class_0(orig_0(e_x)) = class_0(orig_0(e_y))$ (i.e., e_x and e_y are in the same global edge class and hence $x \sim y$), then the nodes x and y are put in BL with the edges e_x and e_y in their sublists.

Otherwise we have $\neg(x \sim y)$ and we do the following. Let $c = contr(y)$ and $d = contr(y)$.

If $c = d$, then x and y are both in the same tree $tree(c)$. Then *basic-boundary*(x,y) is performed that gives BL as its output.

Otherwise we have $c \neq d$ and the following is done (corresponding to the observations of Section 5). A recursive call $boundary(c,d,i-1)$ is performed that outputs a boundary list BB for c and d, consisting of nodes and edges of the contraction tree in layer $i-1$.

For each node f in BB a list $bl(f)$ is computed as follows (according to the observations in Section 5). First the original(s) in layer i of the edges in the sublist of f are obtained. Let these edge(s) be the edge (e_1,z_1,u) and (if $zz \notin \{c,d\}$) the edge (e_2,z_2,v), where z_1 and z_2 are the nodes in which these edges are incident with $tree(f)$. If $f = c$ or $f = d$ then let $z_2 = x$ or $z_2 = y$ respectively. Then in $subtree(f)$ a boundary list $bl(f)$ for z_1 and z_2 is computed by a call *basic-boundary*(z_1,z_2). The sublists of the nodes z_1 and z_2 in $bl(f)$ are extended with edge (e_1,z_1,u) and (if $f \notin \{c,d\}$) edge (e_2,z_2,v)

respectively. Finally, a node $z \in \{z_1, z_2\}$ for which the sublist of z in sequence $bl(f)$ consists of two edges that are in a same edge class, is deleted from the sequence (together with its sublist).

Then BL is obtained by concatenating the lists $bl(f)$ in the order in which the contraction nodes f occur in BB.

joinclasses(J, i): First a joining list JJ of nodes in layer $i-1$ is made as follows. The nodes in JJ consist of the nodes $contr(x)$ for nodes x occurring in JJ. For $c \in JJ$, the sublist for c is the concatenation of all sublists for $x \in J$ with $contr(x) = c$. (JJ is constructed such that no contraction node occurs more than once in JJ by having for each contraction node that is already in JJ a pointer to its occurrence in JJ.) Then, for each node c in JJ, the classes are determined in which the edges in its sublist are contained in, and its sublist is replaced by a sublist that contains for each of these classes one external edge (if any). Remove all nodes of JJ that have a sublist that is empty or that consist of one edge only. If $JJ \neq \emptyset$ then perform recursively a call $joinclasses(JJ, i-1)$. Delete list JJ. All the original edge sides of the edge sides that are un-m-marked in layer $i-1$ (and that hence are contained in the edge classes occurring in JJ), are un-c-marked in layer i (and the related pointers are deleted). (Note that these edge sides in layer $i-1$ can be obtained by either having the recursive call $joinclasses(JJ, i-1)$ return these edge sides or by obtaining all the m-marked edges in layer $i-1$ for the edge classes occurring in JJ before the recursive call and by checking which of these edges still are m-marked after the call.)

Now for each node x in J, execute $basicjoinclasses(J_x)$, where J_x contains x and its sublist in J. (Note that at most one of the old classes still contains a c-marked edge because of the previous un-c-marking).

candidates(x, y, i): Let $c = contr(x)$ and $d = contr(y)$. If $c = d$, then do the same as for $i = 1$ (we have the same situation now). Otherwise, perform candidates$(c, d, i-1)$ that returns the edges e_c and e_d (where e_c is either the father edge of c or it is m-marked w.r.t. c and similar for e_d and d). Let edge $e_1 \in extree(c)$ be the original (in layer i) of e_c. Hence, e_1 and x are in the same extended subtree and e_1 is c-marked in the extended tree.. Let $e_2 := max(c(e_1))$. If e_2 is incident with x, then $e_x := e_2$ (hence e_x is m-marked w.r.t. x), otherwise e_x is the father edge of x. The same is done for y yielding e_y. Return the edges e_x and e_y.

(Note that in this case *candidates* return a correct pair of edges indeed, which is ssen as follows. By the specification of $candidates(i-1)$ the originals of the edges e_c and e_d in $tree(s, i)$ are in the same global edge class in $tree(s, i)$, if such edges exist. Then the correctness follows by similar observations as those for $i = 1$.)

We are left with the problem of how to obtain and store the values *weight*, *lowindex* and the Ackermann values. All these values depend on both the tree name and the layer number. The values $lowindex(s,j)$ and $weight(s,j)$ for all relevant j are stored in a list of records: each records contains these values for some layer j. The tree name s contains a pointer to the begin and the end of the list of records. (The end of the list is the record for layer i if for the FRT(i) structure we have $F_i = F_0$, i.e., FRT(i) is used in some environment not being a part of a FRT($i+1$) structure.) For further details and for the problem of how to obtain Ackermann values we refer to [15]. The approach is similar, where the pointers *contr* in the structures FRT(i) correspond to the pointers *father* in the structures UF(i).

In the FRT(i) structure, UF(j) structures are used for $1 \leq j \leq i$. Since the size of the occurring sets of edges will not exceed $2n$, and since the only way in which the number of elements is relevant for the UF(j) algorithms, is in the size of the Ackermann net that is present (which must be an Ackermann net for at least the size of the largest set that ever exists), it follows that it suffices to use the UF(j) structures with one Ackermann net that is used for all structures, where the net is an Ackermann net for $2n$.

If FRT(i) is used on F_0 (i.e., $F_i = F_0$ and hence $tree(s,i) = tree_0(s)$ for all tree names s), and hence inside an environment not being FRT($i+1$), then $UF_0 = UF(i)$ (i.e., the edge classes on the original dynamic forest F_0 are represented as a Union-Find structure UF(i)).

If FRT(i) is used directly on F_0 (i.e., $F_i = F_0$), then the initialisation for some (sub-)collection of nodes in singleton trees is as follows. Relate a tree name s to each singleton tree. For each node x with name s for the singleton tree consisting of x, the following initialisation is performed: $contr(x) = s$, $nodes(s) = \{x\}$, $weight(s,i) = 1$, $lowindex(s,i) = -1$. (Note that the insertion of a singleton set consisting of a newly created node can easily be performed in this way too.) If we want to initialise the structure for some a forest F_0 not necessarily consisting of singleton trees, where there is a list of the names of the existing edge classes and for each name there is a sublist with the edges in the corresponding class, then this can be performed as follows. First the forest is initialised as a forest of division trees, where each division tree contains exactly one elementary subtree, viz. the tree itself. This is done in the way described in Section 6. Hence for each tree there is exactly one (new) contraction node. For a singleton tree, the contraction node is taken to be the tree name s in FRT(i) and then $weight(s,i) := 1$, $lowindex(s,i) := -1$. For a tree T that is not a singleton tree, let c be its (new) contraction node c created by the initialisation as division tree. Relate a tree name s to tree T. Then make $nodes(s) = \{c\}$, $contr(c) = s$, $weight(s,i-1) = 1$ $weight(s,i) =$ [the number of nodes in the tree] and $lowindex(s,i) = lowindex(s,i-1) = -1$.

8 Complexity of FRT(i)

We consider the time and space complexity of FRT(i) structures and their operations.

We denote the call of procedure *link*, *boundary*, *joinclasses* or *candidates* in layer j (i.e., in FRT(j)) by *link*(j), *boundary*(j), *joinclasses*(j) or *candidates* respectively (omitting the other arguments).

The execution of a call of *treename* in a FRT(i) structure ($i \geq 1$) takes at most $c_t.i$ time for some constant c_t, since starting from the nodes in layer i at most i pointers in the successive layers have to be traversed before the tree name is reached.

The execution of a call of *candidates*(i) in a FRT(i) structure ($i \geq 1$) takes at most $c_c.i$ time for some constant c_c. This is seen as follows. For FRT(1) it is easily seen that *candidates*(1) takes 1 Find operation, which takes at most d_c time since UF(1) is used. For FRT(i) ($i > 1$) consider call *candidates*(x, y, i). If $contr(x) = contr(y)$ then we have the same situation as for *candidates*(1). Hence, since UF(i) is used for the local edge classes, the time complexity is at most $d_f.i$ time. Otherwise, note that all instructions except for the recursive call *candidates*($i-1$) can be done in at most c_r time for some constant c_r. Therefore, by induction, a call takes at most $c_c.i$ time altogether, where $c_c \geq max\{d_c, d_f, c_r\}$.

The execution of a nonessential call *boundary*(x, y, i) in a FRT(i) structure ($i \geq 1$) takes at most $c_b.i$ time plus the time for at most two Finds in UF_0, for some constant c_b. This is seen as follows. If $i = 1$ then, since $x \sim y$, the computations in the call *basic-boundary*(x, y) are similar to those performed in *candidates*($x, y, 1$). If $i > 1$ then in the call only *candidates*(x, y, i) is executed together with 2 Finds (viz., the calls $class_0$) in UF_0. This gives the above bound.

We consider the complexity of the further operations, viz., the complexity of feasible sequences. We determine the time complexity in steps, where one *step* denotes a Find operation (in any involved Union-Find structure), a *candidates* operation, a nonessential *boundary* operation or one ordinary elementary computation step not included in these three operations. Hence, each *candidates* operation and each nonessential call of *boundary* takes 1 step.

We obtain the following result.

Lemma 8.1 *Let a FRT(i) structure for a forest with n nodes be given. The structure and the algorithms can be implemented as a pointer/$\log n$ solution such that the following holds. An essential sequence in FRT(i) (cf. Section 3) needs a total of $O(n_e.a(i, n_e))$ steps ($i \geq 1$, $n_e \geq 2$), where n_e is the number of nodes that are not contained in singleton trees after the execution of the sequence.*

Note in the lemma that $n_e \leq n$. The proof of the lemma is given in Section 9.

9 Proof of Lemma 8.1.

Lemma 8.1 is proved by induction in a way similar to the proof in [15]. We consider the *net cost* of the basic operations, i.e., the cost of the operations except for the cost of Union operations and creations of new singleton sets in Union-Find structures.

basic-integrate(x,f): Let T be the tree containing x. This operation takes a net cost of $O(|T|)$ steps, since all old subtrees of T can be integrated to one tree by a simple traversal, while the updates for the edge classes takes a number of Finds linear to the number of edges. Moreover, Unions occur on two different classes, viz. in which the two sides of an (old) external edge are contained.

basic-external-link((e,x,y)): This operation takes net $O(1)$ steps.

basic-internal-link((e,x,y),y): Firstly, *basic-integrate*(d) takes $|T_y|$ net steps, where T_y is the tree containing y. Then a *basic-external-link*$((e,x,y))$ and the remaining updates take $O(1)$ steps . Hence, the operation takes $O(|T_y|)$ steps.

basic-boundary(x,y): A call *basic-boundary*(x,y) takes $O(|BL|)$ steps if BL is the resulting boundary list for x and y. This is seen as follows. If $x = y$, this is obvious. Consider $x \neq y$. Then the computations take $O(|s(x)| + |s(y)| + |BL| + 1)$ steps. Moreover, $|s(y)| - 1 \leq |s(x)| \leq |s(y)| + 1$ and hence $|BL| \geq min\{|s(x)| - 1, |s(y)| - 1, 2\}$. Therefore $|s(x)| \leq 2|BL|$ and $|s(y)| \leq 2|BL|$. Hence all this takes $O(|BL|)$ steps.

basic-joinclasses(J): This takes $O(|E_J|)$ steps, where E_J is the number of edges in J. This follows since for each occurring edge class in J, $O(1)$ steps are performed.

We now consider the complexity of the structures FRT(i). Like in [15], we do not need to consider the complexity of storing and obtaining the information for each layer that exists for a tree name, since this can easily be charged to other operations by increasing their cost with $O(1)$ time per operation.

We show that an essential sequence in FRT(i) (of procedure *link*(i) *pathrep*(i) and *joinclasses*(i)) takes $O(n.a(i,n))$ steps on n nodes. Moreover, we show that the number of times that an edge becomes affected in FRT(i) (by procedure *basic-internal-insert* or *basic-integrate*, cf. Section 6) is at most $a(i,n)$.

We prove this by calculating the *net cost* of the procedures *link*(i), (essential) *boundary*(i) and *joinclasses*(i), the cost of unions and creations of singleton sets in layer i and the cost of essential recursive calls: for each call of the procedures *link*(i), (essential) *boundary*(i) and *joinclasses*(i) in layer i we do not account for

steps performed in an essential recursive call or steps regarding Unions or creations of new singleton sets. Here, an *essential recursive call* is any recursive call of these procedures with the restriction that recursive *boundary* calls are essential.

9.1 FRT(1)

We consider the cost of an essential sequence on n nodes ($n > 1$) in FRT(1).

We consider the *net cost* of each of the procedures and we consider the cost of unions and creations of singleton sets.

procedure link(1): Consider procedure *link*. The execution of a procedure call $link((e,x,y),s,t,1)$ takes at most $c_0.|weight(t,1)|$ steps (for some appropriate constant c_0), where w.l.o.g. $tree(t,i)$ is the smallest of the two sets to be joined. Now charge the cost of such a linking to the nodes in $tree(t,1)$ by charging to each node for at most c_0 steps. A node can only be charged to if it becomes an element of a new tree whose size is at least twice the size of the old tree it belonged to. Hence a node can be charged to at most $\lfloor \log n \rfloor \leq a(1,n)$ times. Therefore, all these operations take at most $d_l.n.\lfloor \log n \rfloor \leq d_l.n.a(1,n)$ steps together.

On the other hand it follows in the same way that the number of times that an edge is affected, is at most $a(1,n)$.

procedure boundary(1): By the above considerations for procedure *basic-boundary* a call $boundary(x,y,1)$ takes $O(|BL|)$ steps where BL is the resulting boundary list for x and y. Note that at least $|BL|-1$ different classes occur in BL, which is ≥ 1. Charge $O(1)$ cost to the encountered classes. After this procedure call, all classes occurring in BL are joined into one new class by a call of procedure joinclasses, since we are considering an essential sequence. Since during all operations there exist at most $2.(2n)-1$ different edge classes (since there are at most $2n$ edge sides), this gives that the total amount of steps is linear to the number of classes that have existed in FRT(1), which yields at most $d_b.n$ for some constant d_b.

procedure joinclasses(1): Procedure call $joinclasses(J,1)$ takes $O(1)$ steps for each class that is joined. Since during all operations there exist at most $2.(2n)-1$ different edge classes, the total amount of steps is at most $d_j.n$ steps for some constant d_j, apart from the time used for joinings.

Unions: There are at most $2n$ edge sides in layer 1. By [15], the time for the joinings and insertion of edges in layer 1 is at most $c_U.n.a(1,n)$ for some constant c_U.

Concluding the above observations, FRT(1) takes at most $d.n.a(1,n)$ steps for an essential sequence on n nodes ($n > 1$) for some constant d. Moreover, the number of times that a node is affected is at most $a(1,n)$.

9.2 FRT(i) for i>1

We now consider the complexity of the execution of an essential sequence in FRT(i) with $i > 1$. We perform the analysis by means of induction on i.

Suppose FRT($i-1$) takes at most $c.k.a(i-1,k)$ steps for all operations *link*, *boundary* and *joinclasses* on k nodes ($k > 1$) in an essential sequence, where c is some arbitrary constant. Moreover, suppose that the number of times that an edge in the FRT($i-1$) structure is affected, is at most $a(i-1,k)$.

We consider the cost for an essential sequence on n nodes ($n > 1$) in FRT(i). We do this by considering the *net cost* of each of the procedures and by considering the cost of unions and creations of singleton sets and the cost of essential recursive calls.

procedure boundary(i): Recall that the call must be essential. Firstly, the call of procedure *candidates*(i) and the check whether its output edges are in the same class and the recursive call *boundary*($i-1$) takes at most c_2 net steps (for some constant c_2). (For, the call *boundary*($i-1$) takes net $O(1)$ steps if it is nonessential and it takes no stpes if it is essential.)

Then for each node f in BB a call *boundary* is performed in *tree*(f) that returns $bl(f)$, which takes $O(|bl(f)|)$ steps. Note that then at most 2 nodes may be removed from $bl(f)$ in the subsequent computations, but still $bl(f)$ contains at least one node: since f is a boundary node in BB, there is at least one boundary node left in $bl(f)$ (cf. Subsection 5). Hence, the net cost of the entire computation of $bl(f)$ is at most $c_3.|bl(f)|$ steps for some constant c_3.

The remaining operations take at most c_4 steps.

Note that afterwards, all classes occurring in BL (which are at least 2 classes since the call is essential) are joined into one new class (b.m.o. procedure joinclasses). Note that each such (old) class has at most 2 edges in BL. Therefore, charge at most $2(c_2+c_3+c_4)$ steps to each encountered class. Since during all operations there exist at most $2.(2n)-1$ different edge classes (in layer i), it follows that the total amount of steps is at most $c_b.n$ for some constant c_b. Hence, the total net number of steps for all these calls is at most $c_b.n$.

procedure joinclasses(i): The procedure takes a net number of steps that is linear to the number of classes that will be joined, apart from the steps for the recursive call. Therefore, each step is charged to a current class that is joined

(i.e., that is joined with another class). Hence, the total net amount of steps is at most $c_j.n$ for some constant c_j.

procedure link(i): Consider procedure Link$((e,c,d),s,t,i)$. We divide this procedure into several parts.

1. The removal of parts of the structures.
2. The calls of procedure *basic-internal-link* and *basic-integrate*,
3. The recursive call $link(i-1)$ and resulting c-mark changes.
4. The rest of the procedure.

We compute the cost of each of the above parts for *all* executions of procedure Link$((e,x,y),s,t,i)$ together.

1. *The removal of parts of structures:* The removal of parts of structures can be performed in $O(1)$ time per item that must be removed. Therefore, we charge the cost of the removal of an item to its creation. This increases the cost of some operations by constant time only.
2. *The calls of procedure basic-internal-link and basic-integrate:* The execution of the calls of *basic-internal-insert* and *basic-integrate* take at most c_5.*(the number of processed nodes)* steps. Therefore, we charge the cost of the above statements to the processed nodes. Note that in both cases the processed nodes will be contained in a new set that has a higher *lowindex* value than the old set in which they were contained, and that a node will never be contained in a set with a lower *lowindex* value. Therefore the number of times that a node can be charged to is bounded by the number of different *lowindex* values. Since there are at most n (>1) elements in a set, there are by the definition of *lowindex* (cf. (4)) at most $a(i,\lceil\frac{n+1}{2}\rceil)+2 \leq 3.a(i,n)$ different values. Therefore, the total cost of the considered parts of the procedure is at most $c_6.n.a(i,n)$ steps for some constant c_6.

   On the other hand it follows in the same way that the number of times that an edge is affected, is at most $a(i,n)$.
3. *The recursive call Link($i-1$)* We consider the cost of a recursive call $link(i-1)$ in the recursive call part.

   The cost for changing c-marks of edges (not being the inserted edge) in procedure $link(i)$ (and for the related computations) is linear to the number of times that contraction edges are affected in the recursive call $link(i-1)$. Later, in the part considering the recursive calls, we will show that this is at most $\frac{1}{2}.n.a(i,n)$. (This is stated in Observation 9.4.) Hence, this takes altogether $c_7.n.a(i,n)$ steps for some constant c_7.

4. *The rest of the procedure:* The execution of all statements except form those considered above require at most c_8 time per call of Link(i). Since there are at most $n-1$ Links, this takes altogether at most $c_8.n$ time.

Hence, adding the above amounts, all calls of procedure *link* take net at most $c_l.n.a(i,n)$ steps for some constant c_l.

Unions: There are at most $2n$ edge sides in layer i. By [15] the time for the joinings and insertions of edges in layer i is at most $c_U.n.a(i,n)$ for some constant c_U.

essential recursive calls: The essential recursive calls are performed on contraction nodes. We first consider contraction nodes and the conditions for a recursive call Link($i-1$).

Observation 9.1 *The operations on contraction trees (in layer i) by procedure Link$((e,x,y),i)$ are:*

1. *the creation of a contraction node, resulting in a singleton tree*
2. *the linking of contraction trees of nodes by Link$((e,c,d),i-1)$*
3. *the removal of a complete contraction tree*

The operations joinclasses(i) and boundary(i) do not change contraction trees apart from joining edges classes inside a contraction tree (by operation joinclasses(i)).

Similar to the proof of Claim 4.2 in [15] we can prove the following claim.

Claim 9.2 *A recursive call Link$((e,c,d),s,t,i-1)$ inside Link$((e,x,y),s,t,i)$, with $c = contr(x)$ and $d = contr(y)$, is performed only if*

$$1 < lowindex(s,i) = lowindex(t,i) \leq a(i,n) \wedge$$
$$weight(s,i) + weight(t,i) < 2.A(i, lowindex(s,i)+1).$$

For a contraction node $c \in CN(tree(s,i))$, we denote by $lowindex(c)$ the value $lowindex(s,i)$. It is easily seen that a Link does not change the value $lowindex(c)$ for any contraction node c that is not disposed by it (since then the new tree name has the same *lowindex* value as the old one). Moreover, the other operations do not change the value $lowindex(c)$ either. Therefore for any contraction node c the value $lowindex(c)$ is fixed (i.e., c is a contraction node for trees with some fixed *lowindex* only). We call a contraction node c with $lowindex(c) = l$ an l-contraction node.

Similarly, we say that any recursive call $Link((e,c,d),s,t,i-1)$ is an l-call if $l = lowindex(s,i) = lowindex(t,i)$. A recursive call $boundary(c,d,i-1)$ or

joinclasses$(JJ, i-1)$ is an l-call if $l = lowindex(s,i)$, where s is the name of the tree on which the operation is applied. Obviously an l-call operates on l-contraction nodes only, and l-contraction nodes are only operated on by l-calls. We compute the cost of all l-calls for fixed value l.

Let l be a fixed number satisfying $-1 \leq l \leq a(i,n)$. We consider the cost of all recursive l-calls.

By Claim 9.2 and since $|nodes(b)| \geq 2$ for each contraction node b, it follows in case of an l-call Link$(s,t,i-1)$ that we have $l > 1$ and that the size of the set $CN(tree(s,i-1)) \cup CN(tree(t,i-1))$ is $< A(i,l+1)$. Therefore the maximal size of any tree of l-contraction nodes that results from such an l-call is $< A(i,l+1)$. By Observation 9.1 and since in an initialisation at most one contraction node per tree is created, it follows that the maximal size of any occurring tree of l-contraction nodes is $\leq max\{A(i,l+1),1\}$.

Note that any occurring tree of l-contraction nodes with $l \leq 0$ consists of one contraction node. Hence, an l-call of *boundary*$(i-1)$ and *joinclasses*$(i-1)$ occurs only if $l \geq 1$.

Now let l be fixed number with $1 \leq l \leq a(i,n)$. Now partition the total collection of all l-contraction nodes involved in l-calls into collections that correspond to the maximal sets that ever exist (which is possible because of Observation 9.1). Then the size of such a maximal collection is at most $A(i,l+1)$. We have the following observation (that will be proved further on).

Observation 9.3 *The sequence of essential recursive l-calls on the nodes of a maximal set in FRT$(i-1)$ is an essential sequence.*

For each such maximal collection of k contraction nodes, the cost of all essential l-calls on these nodes in FRT$(i-1)$ is at most $c.k.a(i-1,k) \leq c.k.\ a(i-1, A(i,l+1))$. Hence, the total cost of all essential l-calls in FRT$(i-1)$ on l-cluster nodes is at most $c.(number\ of\ l\text{-}cluster\ nodes).\ a(i-1, A(i,l+1))$. Since for each l-contraction node b we have $|nodes(b)| \geq 2.A(i,l)$ (cf. (5)), and since as long as a node is contained in tree structures with *lowindex* value l it has the same contraction node in which it is contained, there are at most $n/(2.A(i,l))$ l-contraction nodes. Therefore, the total number of steps for all essential l-calls is at most

$$\begin{aligned} & c.\frac{n}{2.A(i,l)}.\ a(i-1, A(i,l+1)) \\ = \ & \frac{1}{2}c.\frac{n}{A(i,l)}.\ a(i-1, A(i-1, A(i,l))) \\ \leq \ & \frac{1}{2}c.n \end{aligned}$$

by using $i > 1$, Equation (1) and Lemma 2.11 respectively.

Since there are at most $a(i,n)$ applicable values l of *lowindex* to be considered (viz. l with $1 \leq l \leq a(i,n)$), this yields that the total number of steps used for all these FRT$(i-1)$-calls is at most $\frac{1}{2}c.n.a(i.n)$.

We consider the number of times that contraction edges are affected, for use in the analysis of procedure *link*. Similarly as above, by the induction hypothesis, the number of times (for fixed $l \geq 1$) that l-contraction edges are affected in the l-calls $link(i-1)$ on a maximal set of l-contraction nodes, having size k, is $k.a(i-1,k)$, which yields again $\frac{1}{2}.n$ times for fixed l. Hence, we obtain the following observation.

Observation 9.4 *The number of times that contraction edges are affected in the recursive calls* $link(i-1)$ *is* $\frac{1}{2}.n.a(i,n)$ *altogether.*

We are left to prove Observation 9.3.

Proof of Observation 9.3. We are left to prove Observation 9.3. Suppose some essential operation $boundary(i-1)$ is executed inside operation $boundary(i)$, returning boundary list BB. For each node f in BB with edges e_1 and e_2 in its sublist, there are edges e_1' and e_2' in $bl(f)$ such that the originals of e_1' and e_1 are in the same edge set in F_0 and similarly for e_2 and e_2'. Since the operations in FRT(i) yield a feasible sequence in FRT(i), the call $boundary(i)$ is followed by a call $joinclasses(i)$ that joins the classes of e_1' and e_2' inside $extree(f)$. Since these two classes each have at least one external edge in FRT(i), viz., $orig(e_1)$ and $orig(e_2)$, there is a recursive call $joinclasses(i-1)$ with a joining list that contains node f together with two edges in its sublists that are in the same edge sets as e_1 and e_2 respectively. This proves that the sequence of essential recursive l-calls on the nodes of a maximal set in FRT$(i-1)$ is a feasible sequence. This concludes the proof of Observation 9.3. □

Combining the above results yields that the total number of steps is at most

$$c_b.n + c_j.n + c_l.n.a(i,n) + c_U.n.a(i,n) + \frac{1}{2}c.n.a(i,n).$$

Note that this is at most $c.n.a(i,n)$ steps if $c \geq max\{d, 2.(c_b + c_j + c_l + c_U)\}$.

Since the constant c was arbitrary and since c_b, c_j, c_l and c_U do not depend on c, we can take $c = max\{d, 2.(c_b + c_j + c_l + c_U)\}$. Then it follows by induction that an essential sequence in FRT(i) takes at most $c.n.a(i,n)$ steps.

9.3 FRT(i) for i$\geq$ 1

From subsections 9.1 and 9.2, it follows that an essential sequence in FRT(i) on n nodes takes at most $c.n.a(i,n)$ steps. By the observation, that all nodes that still are in singleton trees after executing the sequence are not involved in the algorithms, Lemma 8.1 follows.

10 FRT Structures

Starting from now on we only consider a FRT(i) structure to be used in some environment not being FRT($i+1$), i.e., $F_i = F_0$. We have the following aspects.

Firstly, we now consider the operations as described in Section 3. We express these operations in the operations described in Section 7. Note that the operations *boundary* and *joinclasses* match in both sections if the appropriate i is used. The operation $link((e,x,y))$ corresponds to the operation

$$link((e,x,y), treename(x), treename(y), i)$$

in the FRT(i) structure. Hence, the time needed for a *link* operation is now extended with two *treename* operations, being two steps. Hence , this does not increase the order of time complexity of this operation. The operations *equal-class-edges*(x,y) can be performed by a call *candidates*(x,y,i) returning two edges e_x and e_y and by performing the Find calls $class_0(e_x)$ and $class_0(e_y)$ (in UF_0). Hence, the time needed for such a call is the time for *candidates* and two Find operations in UF_0, which is $O(1)$ steps. Therefore, we can consider the operations as described in Section 3 with the same order of complexity. Thus, Lemma 8.1 remains valid for these operations (in order of magnitude).

Secondly, for UF_0 (that represents the edge classes in F_0) UF(i) is used. Now each operation *joinclasses*(J,i) performed in FRT(i) also joins all classes in F_0 occurring in J (in UF_0). (This obviously can be done in $O(|J|)$ steps apart from the time needed for performing the Union operations themselves. Hence, these steps do not increase the total time complexity of the FRT(i) structure).

Henceforth, we denote by an FRT(i) structure a thus adapted FRT(i) structure.

Note that all Union-Find structures used in FRT(i) are UF(j) structures with $1 \leq j \leq i$, and that UF_0 is UF(i). Therefore it follows that a step, as defined in the previous subsection, is $O(i)$ time.

By [15] an Ackermann net for n can be computed in $O(\log n)$ time and takes $O(\log n)$ space. Moreover, it is readily verified that the initialisation of FRT(i) can be performed in $O(n)$ time. Finally, by induction to i it easily follows that the total space

complexity of FRT(i) is $O(n)$, since layer $i-1$ has at most $\frac{1}{2}.n$ contraction nodes since for each contraction node b we have $|nodes(b)| \geq 2$.

By Lemma 8.1, by the above observations and since UF(i) takes $O(n_e.a(i,n_e))$ time for n_e elements, we obtain the following result.

Theorem 10.1 *Let a FRT(i) structure for a forest with n nodes be given. The structure and the algorithms can be implemented as a pointer/* $\log n$ *solution such that the following holds. An essential sequence (of the operations link, boundary and joinclasses) in FRT(i) needs a total of* $O(n_e.i.a(i,n_e))$ *time* ($i \geq 1$, $n_e \geq 2$), *where* n_e *is the number of nodes that are not contained in singleton trees after the execution of the sequence. (Of course,* $n_e \leq n$.*) Each equal-class-edges operation takes* $O(i)$ *time. Each nonessential call boundary takes* $O(i)$ *time. The initialisation can be performed in* $O(n)$ *time and the entire structure takes* $O(n)$ *space* ($i \geq 1$, $n \geq 2$).

By using the same solution as in Theorem 6.1 of [15] for the augmentation of the Ackermann net that is used, the above lemma can be extended with the insertion of new (isolated) nodes in the structure with the same complexity bounds, where the insertion of a new node takes $O(1)$ time.

We define an α-FRT structure (for n nodes) as follows. Initially, a FRT($\alpha(n,n)$) structure is used. From time to time, a transformation is performed, replacing a FRT(i) structure by a FRT($i-1$) structure, viz., each time that $\alpha(q,n)$ decreases by one, where at any moment q is the number of queries *equal-class-edges* and *boundary* performed until then. This is performed similar to the way in the proof of Theorem 5.2 of [15] (the full paper), where hence now the queries *equal-class-edges* and *boundary* play the role of the Find operations, and where *link* and *joinclasses* play the role of the Union operations. The building of the new structure FRT($i-1$) is done like in Theorem 5.2 in [15], but instead of building parts of FRT($i-1$) during *equal-class-edges* and *boundary* operations, and using parts of both FRT(i) and FRT($i-1$), we do the following. We have all pointers in the forest F_0 in duplicate, say version 1 and version 2, and we either use version 1 or version 2 of all the pointers. When for FRT(i) version 1 is used, then FRT($i-1$) is builded with version 2 and starting from the moment that FRT($i-1$) is completed the version 2 pointers are used (instead of version 1 pointers).

Then we obtain the following result.

Theorem 10.2 *Let an α-FRT structure for an "empty" forest with n nodes be given. The structure and the algorithms can be implemented as a pointer/* $\log n$ *solution such that the following holds. A matching sequence M of operations link, boundary, joinclasses and equal-class-edges in α-FRT needs a total of* $O((n_e+m).\alpha(m,n))$ *time, where m is the number of operations equal-class-edges and boundary that is performed, and where* n_e *is the number of nodes that are contained in non-singleton trees at the end (and, hence, the essential subsequence of M*

consists of $\theta(n_e)$ operations). The q^{th} call of the operations equal-class-edges and boundary takes $O(\alpha(q,n))$ time if it is a call of equal-class-edges or a nonessential call of boundary. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space.

The proof is similar to the proof of Theorem 5.2 in [15]. However, for the initial case, i.e., $i = \alpha(n,n)$, an essential sequence takes $O(n_e.i.a(i,n_e)) = O(n_e.i)$ time (by Theorem 10.1 and Lemma 2.13). For rebuilding a FRT(i) structure to a FRT($i-1$) structure, we now charge to each of the last $\lceil \frac{1}{2}f \rceil$ operations *equal-class-edges* and *boundary* for $O(i)$ time, based on Equation (13) in [15] (note that now $f \geq n$). This $O(i)$ is then included in the cost of these operations, hence augmenting their cost by a constant factor only. Thus, if $m \leq n$, we have $i = \alpha(n,n)$, and the total cost is $O((n_e + f).\alpha(m,n))$. Otherwise, charge the $O(n_e.i)$ cost for FRT($\alpha(n,n)$) to the first n operations *equal-class-edges* and *boundary*, hence augmenting their costs by a constant factor again. Then, the cost of these operations *equal-class-edges* and *boundary* is computed like in Theorem 5.2 in [15], yielding the required result.

Note that the number m in this lemma refers to the number of calls of operations *equal-class-edges* and *boundary* that are performed in the environment. I.e., a call *equal-class-edges* inside operation *boundary* is not relevant. (However, if these calls inside other operations are counted for too (but not the recursive calls), this still does not affect the above statement.)

Note that the adapted building strategy (where the building of a new structure is distributed over several operations) is important only if we want queries like *equal-class-edges* (or nonessential *boundary* calls) to have a $O(\alpha(q,n))$ worst-case time. Otherwise, the building can be done straightforward during one of the operations and the two versions of the pointers in F_0 are not needed.

By using the same techniques as in Theorem 6.2 in [15], the above theorem can be extended with the insertion of new (isolated) nodes in the structure with the corresponding complexity bound $O(n + (n_e + m).\alpha(m,n))$ (where m, n, n_e, and q denote the current number at the time of consideration). The strategy is again to start with a structure FRT($\alpha'(n,n)$) (where $\alpha'(m,n)$ is defined below, satisfying $\alpha'(m,n) = \theta(\alpha(m,n))$), and to replace FRT($i$) by FRT($i'$) (for some $i' \neq i$) in case $\alpha'(q,n)$ decreases or increases (with additional constraints), where at any moment q is the number of queries *equal-class-edges* and *boundary* performed until then, and n is the number of nodes actually present in the structure at that moment, while the insertion of a new node in the structure is deferred until that node is operated on (viz., by operation *link*: it then becomes part of a non-singleton tree). (The deferring of insertions in the actual data structure guarantees, that at any time, $n_{pres} = n_e$, where n_e is as before, and where n_{pres} is the number of nodes present in the thus adapted structure.) All this is performed similar to the method in the proof of Theorem 6.2 in [15] (the full paper), where now the queries *equal-class-edges* and *boundary* play the role of the Find operations, where *link* and *joinclasses* play the

role of the Union operations, and where the building of the new structure FRT(i') is done like in Theorem 6.2 in [15] with the previous adaptations. We want to remark that if at any time $m = O(n)$ (i.e., at any time the number of operations performed until then is at most linear in the number of nodes present at that time), then the above transformation techniques can be simplified by replacing m by n in the conditions; then only $\alpha(n,n)$ is used and maintained, and only rebuildings from FRT(i) to FRT($i+1$) are performed, viz., if $\alpha(n,n)$ increases. (This situation occurs in the 2ec-and the 3ec-problem.)

We describe further changes for the above situation w.r.t. the proof of Theorem 6.2 in [15]. Firstly, instead of the inverse Ackermann function $\alpha(m,n)$, a variant is taken, viz.,

$$\alpha'(m,n) = min\{i \geq 1 | i.(a(i,n) - 5) \leq 5.\lceil m/n \rceil\}.$$

We have $\alpha'(m,n) = \theta(\alpha(m,n))$. The checking of the transformation condition can be done in a way similar as in [15], and the only necessary arithmetic operations still are addition, subtraction and comparison. Then the complexity part of the proof of Theorem6.2 in [15] is changed as follows. Lemmas 2.10, 2.11 and 2.12 in [15] are adapted to deal with $i.a(i,n)$ instead of with $a(i,n)$. The cost function in the proof of Theorem 6.2 in [15] is slightly adapted (viz., its constants are changed, and n_{base} is replaced by $n_{base}.\alpha_b$). Since at any moment, the number n_{pres} of nodes actually present in the structure satisfies $n_{pres} = n_e$, and since an insertion takes $O(1)$ time, the resulting time complexity becomes $O(n + (n_e + m).\alpha'(m,n)) = O(n + (n_e + m).\alpha(m,n))$.

11 Three-Edge-Connectivity

We will now extend the results to the maintenance of 3-edge-connected components in a graph, with a time complexity of $O(n + m.\alpha(m,n))$ for n nodes and m queries and insertions. We first state the combinatorial observations of [18]. For details we refer to [18]. In Subsection 11.1 we consider maintaining the 3-edge-connectivity relation within 2-edge-connected graphs and subsequently in Subsection 11.2 we consider the problem for general graphs.

Let $G =< V, E >$ be a graph. The set V can be partitioned into equivalence classes for the 3-edge-connectivity relation, called *3ec-classes*. Each 3ec-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 3ec-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 3ec-class into one representing (class) node (see Figure 5 if G is 2-edge-connected). By Lemma 2.10

it follows that $3ec(G)$ does not contain pairs of distinct class nodes that are 3-edge-connected in $3ec(G)$.

11.1 Two-edge-connected graphs

Throughout this subsection, we suppose that the graph G is 2-edge-connected. We give some observations. By Lemma 2.10 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. 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

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

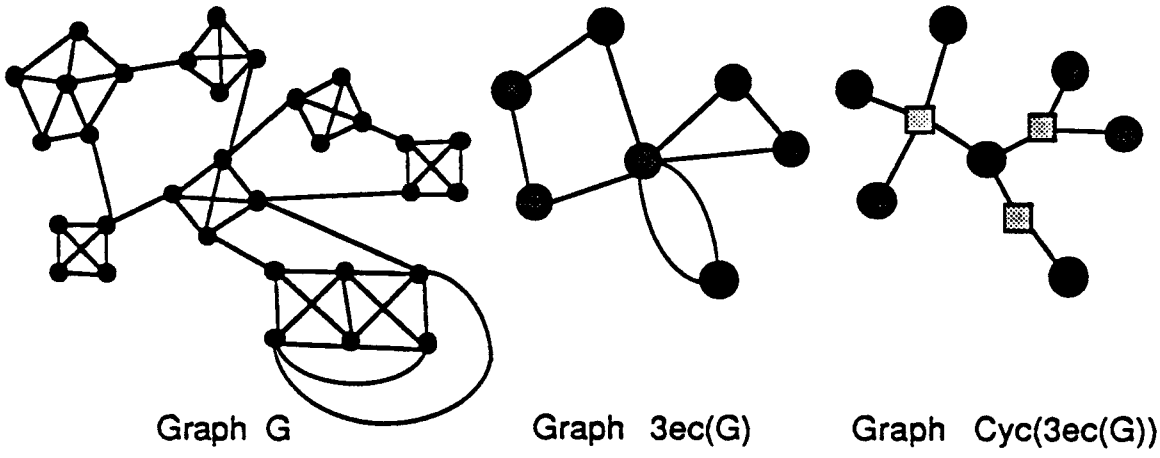
Hence, $Cyc(3ec(G))$ consists of the class nodes and cycle nodes of $3ec(G)$, where a class node c is adjacent to a cycle node s in $Cyc(3ec(G))$ iff c lies on cycle s in $3ec(G)$ (i.e., c is "incident" with cycle s). Therefore, graph $Cyc(3ec(G))$ shows the incidence relation for class nodes and cycles. Moreover, graph $Cyc(3ec(G))$ is a tree. We call graph $Cyc(3ec(G))$ the *cycle tree* of G. The structure of $Cyc(3ec(G))$ is illustrated in Figure 5, where the cycle nodes are drawn as boxes.

11.1.1 Edge insertions

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 2.10), 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 $3ec(G)$.

Figure 5: A 2-edge-connected graph G and the related graphs $3ec(G)$ and $Cyc(3ec(G))$.



Lemma 11.1 *[18] 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.*

By Lemma 11.1 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 2.10 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'
- 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 6. The cycle tree changes as follows. Consider 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 7). 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).

Figure 6: Adapting the tree path between $3ec(x)$ and $3ec(y)$.

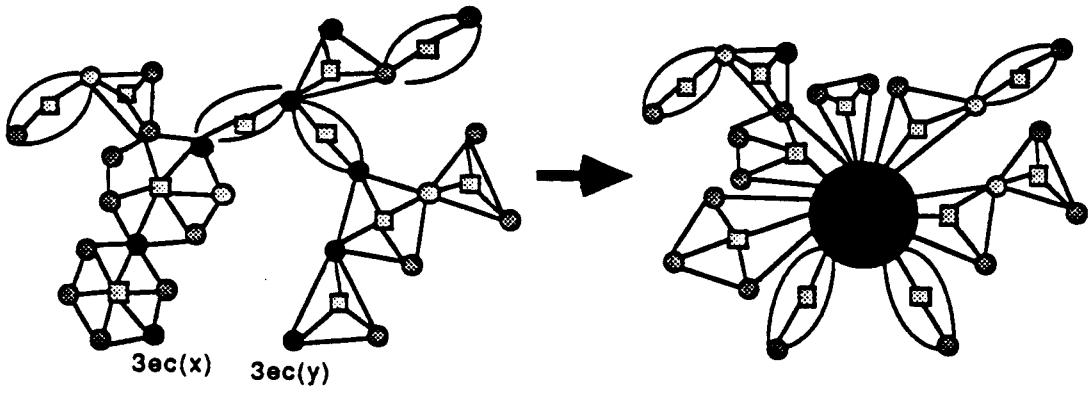
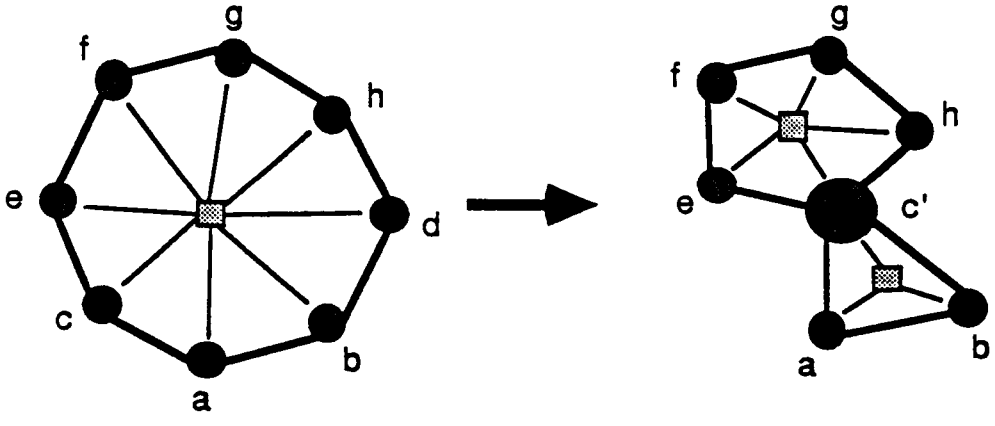


Figure 7: Splitting cycles.



Lemma 11.2 *[18] Given a 2-edge-connected graph G of n nodes with a cycle tree, there exists a data structure for the 3ec-problem (that also maintains a cycle tree) such that the following holds. The total time for m insertions and queries is $O(m+n)$ time 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 $O(n)$ elements. The data structure takes $O(n)$ space.*

11.2 General Graphs

11.2.1 Observations

We now extend the solution of the previous section to general graphs. We first state observations of [18]. For detecting the 3ec-classes it suffices to detect the 3ec-classes inside the 2-edge-connected components. Therefore, our algorithms for general graphs maintain the 2ec-classes (as in Section 4), and they maintain the 3ec-classes by using solutions for 3-edge-connectivity within 2-edge-connected components.

We denote the forest of all cycle trees for the 2-edge-connected components by $Cyc(3ec(G))$. We call $Cyc(3ec(G))$ a *cycle forest* of G.

Suppose edge (e,x,y) is inserted in graph G yielding graph G'. Then the following changes occur. We distinguish three cases.

If $c(x) \neq c(y)$, then the 2ec-classes and the 3ec-classes do not change.

Otherwise, if $2ec(x) = 2ec(y)$ then (e,x,y) is inserted inside a 2-edge-connected component and the changes as described in Subsection 11.1 occur.

Otherwise we have $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$. 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) (cf. Subsection 4.1) 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:

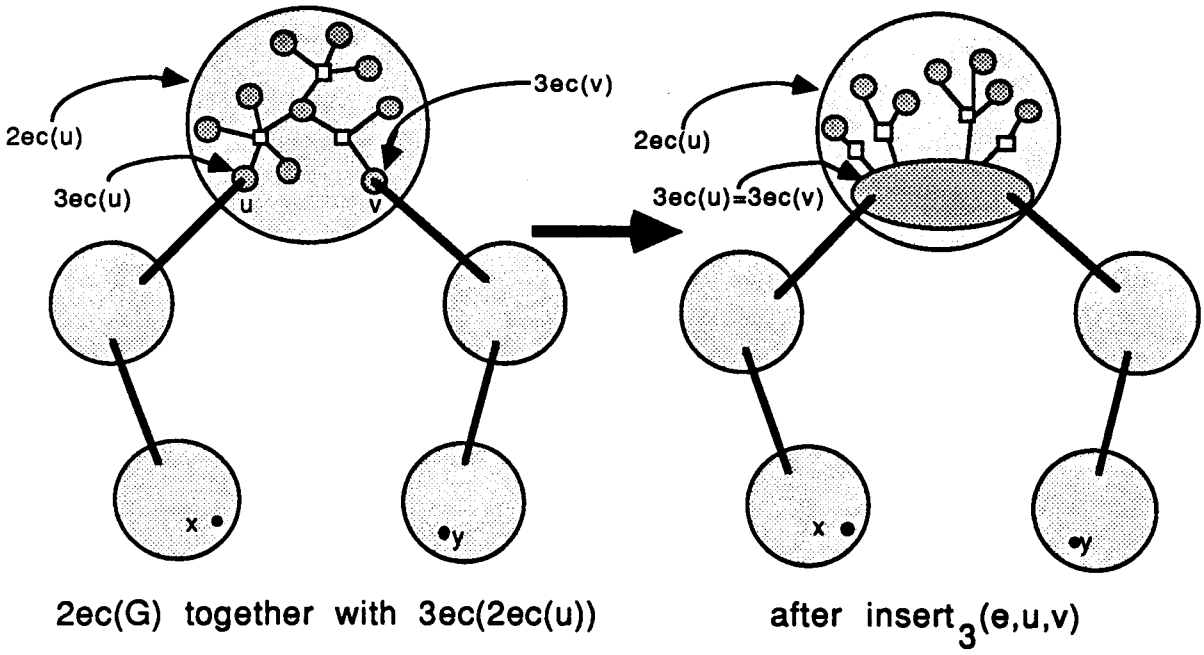
1. all 2ec-classes corresponding to class nodes on P_2 form one new 2ec-class (cf. Subsection 4.1)
2. for each 2ec-class C on P_2, the 3ec-classes inside C (and hence the corresponding cycle tree) are changed: several 3ec-classes may form one new 3ec-class
3. a new cycle s of 3ec-classes arises; the new cycle node s links the (updated) cycle trees that correspond to the 2ec-classes on CS_2

We consider the changes more precisely.

1. This part is identical to Subsection 11.1.

2. We consider the changes of the 3ec-classes that occur in 2ec-classes on P_2. Consider a particular 2ec-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 8), since the 3ec-classes are completely determined by the

Figure 8: Tree path versus temporary edges.



2-edge-connected components in which they are contained (and hence by the nodes in C together with edges of G that have both their end nodes in C. Cf. Corollary 2.8). The update of the 3ec-classes (and hence the cycle tree) can be performed in C by the insertion of a temporary edge in the 2-edge-connected component C.

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

component C on P_2 is linked to the new cycle node s by an edge between cycle node s and the class node of the interconnection 3ec-class in C. All these cycle trees are linked to s and hence now form one new tree together.

11.2.2 Algorithms

We have the following observation for inserting an edge in a 2-edge-connected graph G (or 2-edge-connected component). The changes in the 3-edge-connectivity relation and the change of $Cyc(3ec(G))$ are only determined by the 3ec-classes in which an inserted edge is contained. Therefore, only the 3ec-classes in which the end nodes of a new edge are contained are relevant, and not the actual end nodes themselves.

Consider some graph $G =< V, E >$. We change the cycle forest $Cyc(3ec(G))$ by on the one hand augmenting the collection of nodes of G and on the other hand partitioning the thus obtained 3ec-classes into subclasses. We do this as follows.

Each 3ec-class in G may be extended with an arbitrary number of new, auxiliary nodes that are considered to be nodes in that 3ec-class. The new additional edges that should make this 3-edge-connectivity relation true are not given explicitly (but of course linking such a new node with some other node by 3 edges will do). In the following, the auxiliary nodes are not distinguished from the original nodes.

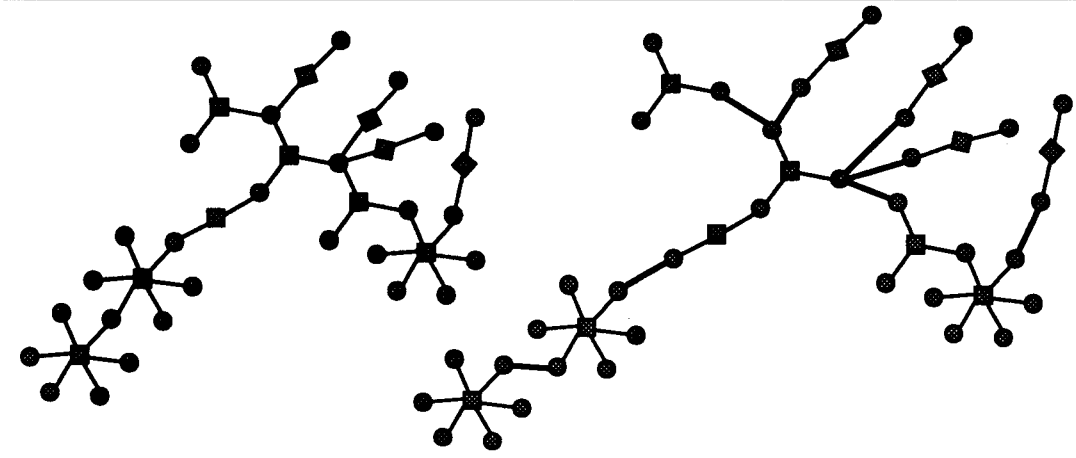
Each (extended) 3ec-class C of G is partitioned into subclasses of nodes. To each subclass a (new) distinct node is related as its name, called the *subclass node*. We call these subclass nodes the subclass nodes for C. The subclass node of the subclass to which a node x belongs is denoted by $sub(x)$. An *augmented cycle forest* AF_G for G for this collection of subclasses is a forest that has the subclass nodes and the cycle nodes of $Cyc(3ec(G))$ as its nodes such that

- for each 3ec-class C of G, the subclass nodes for C induce a subtree of AF_G
- $Cyc(3ec(G))$ is obtained (up to edge names) if for each 3ec-class C the subclass nodes for C are contracted into the corresponding class node of C.

Note that the edges between a cycle node and a subclass node in AF_G correspond to the edges in $Cyc(3ec(G))$, viz., for each edge in $Cyc(3ec(G))$ between class node c and cycle node s there is precisely one edge between s and some subclass node c' for class c. We call the (other) edges that connect two subclass nodes (that hence correspond to the same class) *connectors*. A connector that links two subclass nodes of a 3ec-class C is called a *connector for 3ec-class* C.

Stated informally, AF_G can be obtained by replacing each class node in $Cyc(3ec(G))$ by some tree of subclass nodes and connectors. See Figure 9.

Figure 9: Augmented cycle forest



We consider the insertion of an edge (e, x, y) in a 2-edge-connected graph in terms of an augmented cycle forest AF_G for G. Let $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. 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 the corresponding classes form one new class. Note that these classes are the classes that have at least one subclass on the tree path P in AF_G between $sub(x)$ and $sub(y)$. Hence, we can update the structure according to the following observations (also cf. Subsection 11.1).

- Two successive subclass nodes on P (without a cycle node in between) correspond to the same class. Hence, it suffices to obtain all the subclass nodes on P that are adjacent to a cycle node on P.
- All the classes of which a subclass node is "on" P must be joined into one new class C'.
- The augmented cycle tree AF_G must be adapted to be an augmented cycle tree for the resulting graph. Hence, all subclass nodes for C' must form a tree and no cycle node may occur in between. In particular, this can be done by joining for each cycle node s on P the two subclass nodes that are its neighbours on P and to split cycle s in AF_G accordingly.

  Therefore updates are locally performed in the way as for cycle trees, viz., for each maximal part of P that does not contain two adjacent subclass nodes (and hence that is locally similar to a cycle tree)

Note that we only join subclasses with subclass nodes that are adjacent to a cycle node and, hence, belong to different classes.

Our goal structure is now as follows. For a graph G, we have a forest $bc(G)$ (not being a forest inside G) and an augmented cycle forest AF_G that satisfy the following. The graph $G =< V, E >$ is extended with a collection of auxiliary nodes, which may be extended from time to time. Each auxiliary node is considered to be in some existing 3ec-class that consists of at least one original node (i.e., a node in V). The additional edges that should make this true are not given explicitly. The (thus extended) vertex set is partitioned into disjoint sets, called *basic-clusters*. Each basic-cluster has a (new) unique node as its name, called *cluster node*. The nodes of forest $bc(G)$ are these cluster nodes. We call the edges of $bc(G)$ *bc-edges*. The following constraints are satisfied.

- Each 3ec-class C is partitioned into subclasses obtained by intersecting C with the basic clusters. To each subclass, a unique node is related as the subclass node. The subclass nodes of G are the subclass nodes in AF_G. Then AF_G is an augmented cycle forest for G.
- Each subclass node is considered to be contained in the basic cluster that contains its subclass. Then for a basic-cluster b, the subclass nodes that are contained in b together with appropriate cycle nodes of AF_G induce a subtree of AF_G, denoted by $tree(b)$.
- The edges of AF_G of which the end nodes are in different basic-clusters are connectors.
- There is a connector with end nodes in the basic-clusters b_1 and b_2 iff there is bc-edge between b_1 and b_2.

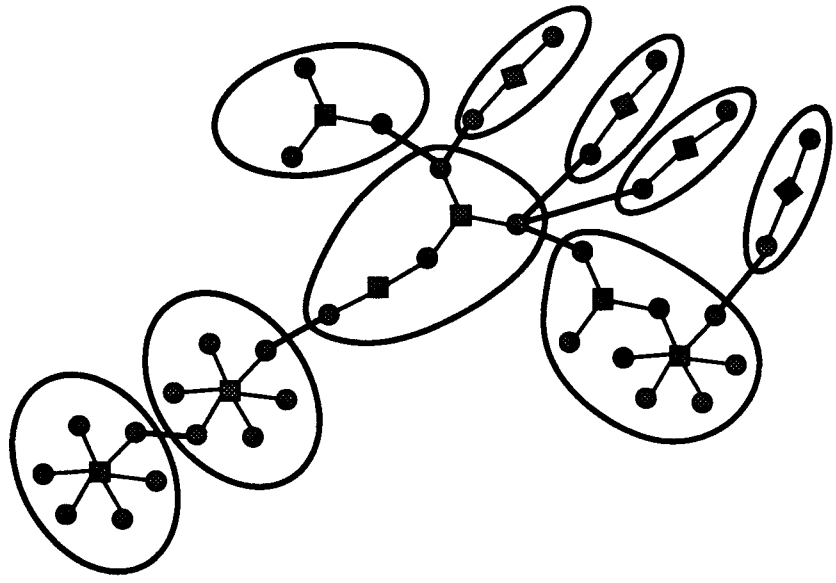
By the above constraints, it follows that for a cluster b, $tree(b)$ does not have two adjacent subclass nodes. Therefore, $tree(b)$ is a cycle tree of a 2-edge-connected graph that has the nodes of basic-cluster b as its nodes together with appropriate edges that induce the 3-edge-connectivity relation as represented by $tree(b)$. E.g. it has all edges of G with end nodes in basic-cluster b together with additional edges between each pair of nodes in basic-cluster b that are 3-edge-connected.

Note that $bc(G)$ can be obtained from AF_G by contracting all subclass nodes in a basic-cluster b to its cluster node b. Note that the only edges in AF_G with images in $bc(G)$ are the connectors.

We thus have a structure of clusters with bc-edges in between, where the original connector of such a bc-edge "connects" the occurrences in AF_G of some 3ec-class of G inside the two corresponding basic-clusters (viz., the 3ec-classes determined by the end nodes of the connector). See Figure 10 for the example of Figure 9.

We define edge classes on $bc(G)$ as follows:

Figure 10: A forest $bc(G)$



The *bc*-edge set of $bc(G)$ is partitioned into disjoint classes, where a *bc*-edge class consists of the *bc*-edges of which the original connectors in AF_G are connectors for the same 3ec-class.

Note that if two *bc*-edges are incident with a cluster node b and if they are in the same *bc*-edge class, then their original connectors in AF_G have the same subclass node as end node (in cluster b).

The above strategy (in terms of an augmented cycle forest) for inserting an edge (e, x, y) in a 2-edge-connected graph is transformed in terms of $bc(G)$ into the following. Let $c = clus(x)$ and let $d = clus(y)$. Suppose that $c \neq d$.

- Let P be the tree path in $bc(G)$ between c and d. Let P' be the tree path in AF_G between $sub(x)$ and $sub(y)$.

  The two incident *bc*-edges of an internal node b on P are in the same *bc*-edge class. Hence, the two connectors that are their originals both are connectors for some 3ec-class C. Moreover, these connectors are on P'. Hence, only one subclass node of P' is in cluster b. Since edges between subclass nodes and cycle nodes in AF_G occur inside clusters only, this gives that there is no cycle node on P that is in cluster b. Hence, we do not need the internal nodes of P.

  A boundary node b of P is either one of the end nodes c or d, or it is a node for which its two incident *bc*-edges e_1 and e_2 on P are not both in the same *bc*-

edge class. In the latter case this means that the two connectors that are their originals are connectors for different 3ec-classes. Moreover, these connectors are on P'. Hence, at least two different subclasses on P' and at least one cycle node on P' are in cluster b.

Hence, to obtain the relevant path parts of P', it suffices to obtain a boundary list BL for c and d and to consider the boundary nodes.

- For each such cluster b with $b \in BL$, a local update of the local cycle tree must be performed by joining all subclasses on the part P'_b of P' inside cluster b and by updating the local cycle tree correspondingly. Note that this update corresponds to the update for inserting a temporary edge between any two nodes of G that are contained in the two subclasses that correspond to the subclass nodes that are the ends of P'_b. The end nodes of P_b are the end nodes (in cluster b) of the originals of the bc-edges on P that are incident with b, where if there is only one such bc-edge, $sub(x)$ or $sub(y)$ is the other end node of P'_b. Note that by the definition of bc-edge classes we still obtain the same end nodes if we substitute these bc-edges by other bc-edges in the same bc-edge classes. Therefore we can use the bc-edges in the sublist of b in BL to obtain the end nodes of P'_b.

We describe a structure, called *3EC structure* that solves the 3ec-problem.

We distinguish between the different layers of *representation.*

The representation for the graph G itself is as follows. Firstly, there is a structure $2EC$ to maintain the 2ec-classes of G. This structure works on the regular nodes only and hence the additional nodes are not involved. There is Union-Find structure for implementing the 3ec-classes of nodes of G, called the global Union-Find structures and denoted by UF_{3ec}. Note that in the $2EC$ structure there are Union-Find structures for the connected components and the 2ec-classes of G, denoted by UF_c and UF_{2ec}.

A query $3ec\text{-}comp(x)$ now corresponds to a Find call $3ec(x)$.

The vertex set of G may be extended from time to time with auxiliary nodes.

Each (original or additional) node x has a pointer $clus(x)$ to the cluster node in which it is contained.

Forest $bc(G)$ is implemented as a fractionally rooted tree structure (FRT), denoted by FRT_{3ec}. (Also the forest $bc(G)$ has a regular implementation as a forest, i.e., with incidence lists for its nodes.)

The augmented cycle forest AF_G is not implemented as a whole. In fact, it is implemented in parts, viz. by cycle trees inside basic-clusters and by separate connectors. To be precise, we have the following implementation.

Note that AF_G has connectors (being the originals of *bc*-connectors) which have end nodes being subclass names. Note that subclasses are joined from time to time. Therefore, instead of having a subclass node as end node, a connector has a node of such a subclass as end node. Then the subclasses that are the ends of a connector (e, x, y) are $sub(x)$ and $sub(y)$.

Recall that for a basic-cluster b, the part of AF_G inside basic-cluster b, viz. $tree(B)$, is a cycle tree on the nodes of basic-cluster b. Then $tree(b)$ is implemented as a cycle tree independent of the rest of AF_G or G. It is implemented and maintained as the cycle tree in the former solution of Lemma 11.2. (for maintaining 3-edge-connectivity inside a 2-edge-connected graph). We refer to this solution as the *local structure*. The Union-Find and the Circular Split-Find structures used in the local structure are denoted by UF_{loc} and GSF_{loc}. The Find operation in UF_{loc} for a node x (returning the name of its subclass) is denoted by $sub(x)$. The insertion operation in a local structure is denoted by $insertloc_3$.

A *bc*-edge has a pointer to its original connector in AF_G as represented above (which actually is an artificial edge between to nodes of G), and, conversely, a connector in AF_G has a pointer to the *bc*-edge that is its contraction edge.

We relate to each subclass of nodes that occurs inside some basic cluster a connector that has one of its end nodes in that subclass (if such a connector exists). Such a connector is called an *associated connector* for that class. (Notice the similarity with the associated edges for nodes in the 2ec-problem.) A pointer *assoc* to that connector is stored in the subclass node.

Remark that the edge classes in $bc(G)$ can now be described as follows:

> Let (e, c, d) be a *bc*-edge. Let (e, x, y) be its original connector. Then (e, c, d) is in the edge class called $3ec(x)$ (this name is only used in the description, not in the algorithms).

(Recall that for a connector (e, x, y) we have $3ec(x) = 3ec(y)$.)

The initialisation for an empty graph is straightforward. (Note that each node in the graph forms a singleton basic-cluster on its own, and, hence, for each node, a cluster node is created representing the singleton basic-cluster that is formed by the node.)

Suppose some new edge (e, x, y) is inserted in G, resulting in graph G'. Let the corresponding clusters for x and y be c and d. Then procedure $insert_3((e, x, y))$ updates the structure as follows. If $3ec(x) \neq 3ec(y)$, then the following cases are considered.

1. $c(x) \neq c(y)$. Then an insertion is performed as for 2-edge-connectivity, viz., by a call $insert_2((e, x, y))$.

2. $c(x) = c(y) \wedge 2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. Let *glob* be an empty list. Let $c = clus(x)$ and $d = clus(y)$. If $c = d$ then BL is the list consisting of c with empty sublist; otherwise, $boundary(c, d)$ is performed in FRT_{3ec}, yielding boundary list BL in $bc(G)$. List BL is copied as list J, but with empty sublists.

   For each basic cluster b in BL, the original(s) of the bc-edge(s) in the sublist of b are obtained (if any). If $b = c = d$ then let $u = x$ and $v = y$. Otherwise, if $b = c$ or $b = d$ then let $v = x$ or $v = y$ respectively, and let node u the end node of the above original edge that is in basic-cluster b. Otherwise, let nodes u and v be the end nodes of the above original edges that are in basic-cluster b. (Note that if $3ec(u) = 3ec(v)$, then $v \in \{x, y\}$, since otherwise the two above bc-connectors in the sublist would be in the same edge class.) If $3ec(u) \neq 3ec(v)$, then the following is done. A call $insertloc_3((e', u, v))$ of a temporary edge (e', u, v) in basic-cluster b is performed (being an insertion in the local cycle tree for b, causing an update of it). Obtain an associated connector for each of the subclasses that are joined in cluster b. Put the corresponding bc-connectors in the sublist in J related to cluster node b. One of these connectors (if any) is assigned to the resulting subclass as its associated edge. For each subclass involved in the joining, obtain a node z of that subclass and put it in list *glob*.

   Note that J consist of the cluster nodes in BL, where the sublists contain the associated edges of the old subclasses that are joined in the clusters (and hence for each bc-edge e in the sublist for a node $b \notin \{c, d\}$ in BL, there is at least one bc-edge in the sublist for b in J that is in the same bc-edge class as e).

   All the classes in which the nodes in *glob* are contained are joined: on each node $x \in glob$ the Find call $3ec(x)$ is performed, all these outputs are put in a list such that every 3ec-class name occurs at most once in the list (which can be done by means of marking), and then Union operations are performed on these names in UF_{3ec}. If the sublist of c or d is empty, then that node is removed from J. Finally, the FRT_{3ec} structure is updated by means of call $joinclasses(J)$.

3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. Firstly, the 2ec-classes that will be joined into one new class are determined. This is done as follows. A boundary list BL for x and y is computed in $2EC$ (this is the first part of the call $insert_2((e, x, y))$). Subsequently the names of the 2ec-classes are obtained, where to each such name k a sublist is related that is the concatenation of all sublists for $z \in BL$ with $2ec(z) = k$. These names are stored in a temporary list TL. Then all edges in the sublists are removed from the sublists that have both their end nodes in the same 2ec-class. (Hence, we are left with a list TL where the sublist of each 2ec-class name k contains the interconnection edges that are incident with 2ec-class k, and with another class of which the name is in TL. Such a sublist consists of exactly two edges except for the sublists of $2ec(x)$ and $2ec(y)$.) Then a list L is constructed from TL consisting of the names and

their sublists in TL such that the (1 or 2) neighbours in L of each 2ec-class c in L are the 2ec-classes in which the end node(s) of the edges in its sublists are contained (apart from 2ec-class c). (Note that this can be done by obtaining each for 2ec-class C the other 2ec-classes in which the end nodes of the edges in its sublist are contained, and by setting pointers from C to these 2ec-class names.) (Now L contains the class nodes of the tree path P between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ in the proper order, where the sublist for each class node c contains the interconnection edges between class c and its neighbour(s) on P. Hence, the sublist of 2ec-class name c consists of the interconnection edges that are incident with class c and with the (one or two) neighbours classes in L.)

For each 2-edge-connected component C in L the following is done. If $C \notin \{2ec(x), 2ec(y)\}$ then u and v are the two nodes in C that are the end nodes of the edges in the sublists of C. If $C = 2ec(x)$ (or $C = 2ec(y)$) then u is the node in C that is the end node of the edge in the sublists of C and $v = x$ (or $v = y$). If $3ec(u) \neq 3ec(v)$ then a temporary edge between u and v in C is inserted by a call $insert_3((e', u, v))$. (Hence, then the local 3ec-classes are updated as above for the case $2ec(u) = 2ec(v)$.) Afterwards, create a new node, which we denote by z_C, and insert it in the (updated) 3ec-class $3ec(u)$ $(= 3ec(v))$ (the interconnection 3ec-class). A connector (e', z_C, z'_C) is created between z_C and some node z'_C of 3ec-class $3ec(z_C)$. Replace the sublist of C in L by the sublist consisting of z_C and the connector.

Then a new basic cluster with (new) cluster name b is created from these new nodes z_C for $C \in L$: each of the nodes z_C is provided with a pointer $clus(z_C)$ to b. Then the subclasses in b are initialised: each node z_C forms a singleton subclass in the cluster on its own. Subsequently a cycle tree corresponding to the (single) cycle of the new subclass nodes in B is initialised: these nodes $sub(z_C)$ occur in the same order as the 2-edge-connected components C in L. (The cycle of these subclass nodes correspond to the cycle of the interconnection 3ec-classes in the new graph $3ec(G')$.)

Then the 2ec-classes in L are joined by performing a call $insert_2((e, x, y))$ in $2EC$ (of which actually the first part was already executed in the beginning of the computations in this case, viz. the computation of a boundary list in $2EC$).

Cluster node b is linked with the involved trees in $bc(G)$ (corresponding to the 2-edge-connected components that are involved) by means of new bc-edges as follows. For each auxiliary node z_C (in L) together with connector (e', z_C, z'_C), let $b' = clus(z'_C)$. Then a new bc-connector (e', b, b') is created (with the appropriate pointer between (e', z_C, z'_C) and (e', b, b')), and the tree in $bc(G)$ containing b' is linked with b by means of call $link((e', b, b'))$. The edge (e', z_C, z'_C) is related to $sub(z_C)$ as its associated edge. If $sub(z'_C)$ does not have

an associated edge yet, then (e', z_C, z'_C) is related to $sub(z'_C)$ as its associated edge. Otherwise, the following is done. Let (e'', z'', z''') be the associated edge for $sub(z'_C)$. Then the operation $joinclasses(J)$ is performed, where J consists of the node b' with the bc-edges (e', b, b') and $(e'', clus(z'), clus(z''))$ in its sublist (to reflect that these two edges are in the same bc-edge class).

We consider some aspects of the above $insert_3$ algorithm.

Suppose the initial graph G_0 has n (regular) nodes. Note that G_0 contains at most n 2-edge-connected components. Then the total number of new (auxiliary) nodes (in the graph) that is created by the algorithm is at most $2n-1$, since a new node is created for each 2-edge-connected component that is joined with other 2-edge-connected components. Hence, the final number of nodes is at most $3n-1 = O(n)$, and the GSF_{loc} structure is a structure on $O(n)$ nodes. On the other hand, the total number of clusters created by the algorithm is at most $n-1$, since a new cluster is created only in case of the joining of 2-edge-connected components, and in that case, the total number of 2-edge-connected components decreases by at least one. Hence, we only need a FRT-structure for at most $2n-1$ cluster nodes. (Note that this can be done e.g. by initially having a collection of $n-1$ "free" ("isolated") nodes available that serve as the nodes to be taken as the new cluster nodes. (Hence, we do not need a structure for increasing number of nodes yet.)) The same holds for the Union-Find structures on nodes of G: we do not need to insert new elements in these structures from time to time if we start from a situation with $2n-1$ auxiliary "free" nodes.

We denote all the Union-Find structures used independently in 3EC (i.e., not as part of FRT_{3ec} etc.) by UF. We consider the UF structures to be one structure; hence, it is a structure on $O(n)$ elements.

We consider the complexity of the above algorithm. Note that there are at most $3(n-1)$ essential insertions possible in the 3ec-problem, since in each essential insert, at least two connected components, two 2ec-classes, or two 3ec-classes are joined.

Lemma 11.3 *In a 3EC structure for a graph with n nodes, a nonessential insertion takes $O(1)$ time together with the time for $\theta(1)$ Find operations in a UF structure. The time needed for a sequence of essential insertions in 3EC is at most linear to the time for an essential sequence on $O(n)$ nodes in FRT_{3ec} and an essential sequence on $O(n)$ nodes in FRT_{2ec}, the time for $O(n)$ Unions in the UF structures and $O(n)$ Circular Splits in the GSF_{loc} structure, the time for $O(n)$ nonessential calls boundary in FRT_{2ec} and FRT_{3ec}, the time for all $O(n)$ Finds in the UF structures and the GSF_{loc} structure, together with an additional amount of $O(n)$ time.*

Proof. We define a step be an ordinary computational step or a Find operation in any UF or GSF_{loc} structure. We consider a collection of essential $insert_3$ operations in the considered graph, including the (essential) $insert_3$ operations called in the

execution of operation $insert_3$ itself. Therefore, we do not consider the cost of an essential call $insert_3$ inside procedure $insert_3$: we already consider it in the above collection. (We may think of such an $insert_3$ call to occur just before the call $insert_3$ in which it was invoked.)

The sequence of calls *link*, *joinclasses* and essential calls of *boundary* in FRT_{3ec} as performed during the $insert_3$ operations yield an essential sequence in FRT_{3ec}, which is seen as follows. Procedure $boundary(c, d)$ is explicitly called in part 2 of procedure $insert_3$ only. Then an essential call $boundary(c, d)$ with output sequence BL is followed by $joinclasses(J)$, where all *bc*-edge classes occurring in BL also occur in J if the *boundary* call was essential.

Moreover, operation *boundary* in FRT_{3ec} is performed at most once in an essential $insert_3$ call. Hence, there are at most $O(n)$ nonessential *boundary* calls.

All calls $insert_{2ec}$ in the calls $insert_{3ec}$ are essential. Therefore, by Lemma 4.2 the claim regarding the operations present in $2EC$ is true.

We consider the *net cost* of the procedure calls of $insert_3$: i.e., the cost of the parts of the computations apart from the computations considered above, from $O(1)$ steps per call $insert_3$ and from the Unions in UF structures and the Circular Splits in the GSF_{loc} structure.

1. Case $c(x) \neq c(y)$. Then there is no net cost.

2. Case $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. Consider a call $insert_3$. Firstly a boundary list BL is computed, which does not contribute to the net cost. Then the basic-clusters in BL are handled as: for each such $b \in BL$ first $O(1)$ steps are performed, and then a call $insertloc_3$ may be performed in cluster b if it is an essential insertion in the local structure. Finally, for each subclass that is joined with at least another subclass (in any local insertion) $O(1)$ steps are performed in $insert_3$.

   Note that there are at most 2 basic-clusters $b \in BL$ in which no subclasses are joined: the $O(1)$ steps performed for these classes are charged to the procedure call $insert_3$, hence not contributing to the net cost. For each other basic-cluster $b \in BL$, the $O(1)$ steps are considered to be included in the $O(1)$ steps performed in one of its subclasses that are joined.

   We now add up all these costs for all calls $insert_3$ together.

   Since there are at most $3n - 1$ nodes present, and since these nodes are partitioned into disjoint clusters in which the local structures are applied, at most $O(n)$ essential calls $insertloc_3$ may occur. By Lemma 11.2 this takes time linear to the time for Unions and Splits in the structures UF_{loc} and GSF_{loc} respectively, which does not contribute to the net cost, together with $O(n)$ Finds in these structures.

Since there exist at most $O(n)$ different subclasses during the entire process, the total number of steps regarding the above $O(1)$ steps per joined subclass is $O(n)$. (There are $O(n)$ different subclasses, since initially there are at most n subclasses and since new nodes each yield one new subclass.)

Hence, the net cost of all calls in this case is $O(n)$ steps.

3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. Note that the computation of a boundary list in 2EC at the beginning of this case is a part of an essential call $insert_2$ (that is actually called later on in $insert_3$) and, hence, can be considered to be included in the above parts for 2EC. (Or observed in another way, this computation of the boundary list is executed twice: one time her and one time later in the "entire" execution of the insertion procedure. This increases the cost with a factor 2 at most.)

   The construction of L from BL takes $O(|L|)$ steps (note that $|L| = \theta(|BL|)$). Then $O(1)$ steps are performed for each 2-edge-connected $C \in L$. Subsequently for each 2-edge-connected component a temporary edge is inserted by a call $insert_3$ in case that that edge has end nodes in different 3ec-classes: hence such an insertion is essential and its cost is included in the previous case (case 2). Moreover, for each 2-edge-connected component a new node is created, together with a new connector. A new cluster consisting of these nodes is created and some additional computations are performed. All this can be done in O(the number of new nodes) steps. Since, the 2-edge-connected components occurring in L are joined, the net cost of all these computations can be seen as $O(1)$ steps per 2-edge-connected component that is joined.

   Since there are at most $2n-1$ 2-edge-connected components during the entire process, the net cost of all the calls in this case is $O(n)$ steps.

Hence, the lemma follows for the essential insertions. The lemma is obvious for nonessential insertions. □

A $3EC(i)$ *structure* is a 3EC structure where $FRT_{3ec} = FRT(i)$, $FRT_{2ec} = FRT(i)$, $UF = UF(i)$ and $GSF = GSF(i)$.

Theorem 11.4 *A 3EC structure with the algorithms solves the 3ec-problem and can be implemented as a pointer/$\log n$ solution such that the following holds. The total time that is needed for all essential insertions starting from an empty graph of n nodes is $O(n.i.a(i,n))$, whereas the queries and nonessential insertions can be performed in $O(i)$ time. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. It is easily seen that the initialisation can be done in $O(n)$ time. By Lemma 11.3, Theorem 10.1 and [15, 17] (for UF(i) and GSF(i)) the theorem follows. □

The α-$3EC$ structure is a 3EC structure for a graph with n nodes where $FRT_{3ec} = FRT(\alpha(n,n))$, $FRT_{2ec} = FRT(\alpha(n,n))$ (where $\alpha(n,n)$ can be obtained as in [15]), $UF = \alpha$-UF and $GSF = \alpha$-GSF, where in the latter structures the number of Finds is replaced by the number of *insert* operations and queries. Then we obtain the following.

Theorem 11.5 *There exists a structure and algorithms that solve the 3ec-problem and that can be implemented as a pointer/$\log n$ solution such that the following holds. The total time that is needed starting from an empty graph with n nodes is $O(m.\alpha(m,n))$ (where m is the number of edge insertions and queries), whereas the f^{th} operation is performed in $O(\alpha(f,n))$ time if that operation is a query or a nonessential insertion. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space.*

Proof. Like the proof of Theorem 4.4. □

By using the α-$3EC$ structure where $FRT_{3ec} = \alpha$-FRT and $FRT_{2ec} = \alpha$-FRT instead, the above theorem can be augmented to allow insertions of new nodes in the graph with a time complexity of $O(n + m.\alpha(m,n))$ (cf. Section 10). Then n, m and f denote the current number at the moment of consideration. (Note that at any time, at most $O(n)$ operations *boundary* are performed on one of the two FRT structures, which simplifies the insertions of new nodes in the FRT structure. Cf. Section 10. A similar remark holds for the structures GSF and UF_{loc}.)

12 Two-Vertex-Connectivity

We will now consider the problem of maintaining the 2-vertex-connected components in a graph, and we will present algorithms with a time complexity of $O(n + m.\alpha(m,n))$ for n nodes and m queries and insertions.

12.1 Graph Observations

Let $G =< V, E >$ be a graph. We define the graph $2vc(G)$ as follows. For each 2vc-class or quasi class there is a unique (new) node related to that class, called the class node. The vertices of $2vc(G)$ are the nodes of G together with these class nodes. For each node x there is an edge between x and each class node c such that x is contained 2vc-class c. (Thus we obtain a collection of trees corresponding to so-called block trees.)

Lemma 12.1 *Graph $2vc(G)$ is a forest, where each tree in $2vc(G)$ corresponds to a connected component in G, i.e., it consists of class nodes together with the nodes of a connected component in G.*

Hence, two distinct 2vc-classes have at most one node in common and, conversely, for any two nodes there exists at most one 2vc-class that contains them.

Lemma 12.2 *If edge (e, x, y) is inserted in graph G, then all the classes of which the class node is on the tree path in $2vc(G)$ between x and y form one new 2vc-class together, while the other 2vc-classes and quasi classes remain unchanged.*

Proof. Let G' be the graph G together with edge (e, x, y). Let P be the tree path between x and y in $2vc(G)$. Let u and v be any two nodes that are adjacent to a class node on P.

Suppose u and v are not adjacent in G. Suppose a node $w \notin \{u, v\}$ is deleted from G'. We show that there is a path from u to v in G'. Delete w in $2vc(G)$. Then there is a path P_1 between u and node x or node y in $2vc(G)$. Since each class node c on P_1 can be replaced by a path in G between any two nodes ($\neq w$) in class c such that it does not contain w (because the corresponding class is either a 2vc-class or it consist of two nodes with an edge in between), this gives that there exists a path between u and node x or node v in G that does not contain w. The same can be obtained for v. Since there exists an edge (e, x, y) in G' this yields that u and v are still connected. Hence u and v are 2-vertex-connected.

Now suppose u and v are adjacent in G. Then either u and v are in the same 2vc-class, in which case we are done, or they are in a quasi class c. In the latter case it follows that c, u and v are on P. Suppose the edge e' between x and y is deleted from G'. We show that there is a path from u to v in G'. There exists a path between u and x or y not using c and hence, like before, there exists a path between u and node x or node y in G that does not contain e'. The same can be obtained for v. Hence x and y are 2-vertex-connected.

On the other hand if u and v are not in the same class, and they are not both adjacent to class nodes on P, note that the removal of any node of G that is on the tree path P' in $2vc(G)$ between u and v separates u and v. Since u and v are not both adjacent to a class node on P, there is a node $w \in G$ on P' that is not on P. Then the deletion of w in G separates either u from v, x and y or x from u, x and y. Hence, after the insertion of edge (e, x, y) in G w is still a cut node.

Finally, if u and v are in the same quasi class and they are not both adjacent to a class node on P, a similar observation yields that the edge between u and v still is a cut edge in G'. □

We represent $2vc(G)$ by means of a spanning forest of G. Consider a spanning forest $SF(G)$ of G. We augment $SF(G)$ with edge classes on its set of edges. An edge class contains all the edges that connect two vertices that are in some 2vc-class or quasi class. An edge class consisting of a cut edge of G is called a quasi edge class (and hence the end nodes of the class form a quasi class). Otherwise the edge class is called a real class.

Now a class of edges together with the end nodes of these edges induces a subtree in $SF(G)$, which is seen as follows. For two nodes x and y that are 2-vertex-connected, all nodes on the tree path P between x and y are 2-vertex-connected with them too. Therefore, all these nodes are in the same 2vc-class and hence the edges on P are in the same edge class. This implies that each edge class induces a subtree in $SF(G)$.

Note that this implies that the collection of edge classes thus yields an admissible partition of $SF(G)$,

From the above observation it follows that two nodes x and y are 2-vertex-connected iff x and y are incident with 2 edges of the same real edge class.

On the other hand, a maximal class of 2-vertex-connected nodes induces some subtree in $SF(G)$ and the set of the edges in that subtree is an edge class. Hence, if we relate to each edge class a new unique node as its class node, if we extend $SF(G)$ with these class node and if each edge (e, x, y) in an edge class is replaced by two edges $(e'x, c)$ and (e'', y, c), then we obtain the forest $2vc(G)$ (up to the choice of the class names and the names of edges). Therefore, we use the names of edge classes as the names of the corresponding 2vc-classes and quasi classes.

We define the predicate $2vc(x, y)$ to be true iff nodes x and y are 2-vertex-connected.

We consider the insertion of an edge in a graph in terms of edge classes by means of Lemma 12.2. 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 x and y are not connected in $SF(G)$. Hence, (e, x, y) connects two trees in $SF(G)$ that have to be joined into one tree.

2. $\neg 2vc(x, y) \wedge c(x) = c(y)$. Edge (e, x, y) connects the nodes x and y in a tree of $SF(G)$ and a cycle arises. Then all edge classes of which an edge is on the tree path between x and y must be joined into one edge class.

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

12.2 Algorithms

We use a fractionally rooted tree structure FRT for the operations on the forest $SF(G)$, denoted by FRT_{2vc}. All quasi edge classes are marked as being quasi. All other classes are not marked (in particular, classes with at least 2 edges are automatically unmarked.) There is a Union-Find structure for connected components, denoted by UF_c. The initialisation for an empty graph is straightforward.

A query $Is2vc(x,y)$ is now performed by first performing a call *equal-class-edge*(x,y); then *false* is returned if the returned edge class names are distinct or correspond to a quasi edge class, while *true* and the (common) edge class name are returned otherwise.

We consider the insertion of an edge in a graph. Suppose a new edge $(e,x,y) \notin E$ is inserted in graph $G =< V, E >$. We distinguish three cases.

1. $c(x) \neq c(y)$. Perform the operation $link((e,x,y))$ to connect the two trees in $SF(G)$ containing x and y respectively. Moreover, the two connected components $c(x)$ and $c(y)$ are joined (in UF_c).

2. $\neg Is2vc(x,y) \wedge c(x) = c(y)$. We need to determine the edge classes that have an edge on the tree path between x and y and then join these classes:

   - obtain a boundary list BL for x and y in $SF(G)$ by a call $boundary(x,y)$.
   - If BL contains nodes x and y only, then x and y from a quasi class. Then unmark the edge class of the edge obtained in the call $Is2vc(x,y)$, reflecting that the edge class is real now.
   - Otherwise, if BL contains more than the 2 nodes x and y, delete the nodes x and y from BL (their sublists contain one edge only). Join all the edge classes occurring in BL by means of the call $joinclasses(BL)$.

3. $Is2vc(x,y) \wedge c(x) = c(y)$. Nothing is done.

A $2VC(i)$ structure is the above structure where $FRT_{2vc} = FRT(i)$ and where $UF_c = UF(i)$. Then we obtain the following result in a way similar to Subsection 4.3.

Theorem 12.3 *There exists a data structure and algorithms that solve the 2vc-problem and that can be implemented as a pointer/* $\log n$ *solution such that the following holds. The total time that is needed for all essential insertions starting from an empty graph of* n *nodes is* $O(n.i.a(i,n))$*, whereas a query and a nonessential insertion can be performed in* $O(i)$ *time. The initialisation can be performed in* $O(n)$ *time and the entire structure takes* $O(n)$ *space* ($i \geq 1$, $n \geq 2$)..

Now take α-FRT as FRT_{2vc} for a graph with n nodes, and take α-UF for UF_c. Then we obtain the following result in a way similar to Subsection 4, where now Theorem 10.2 is used instead of Theorem 10.1.

Theorem 12.4 *There exists a data structure and algorithms that solve the 2vc-problem and that can be implemented as a pointer/* $\log n$ *solution such that the following holds. The total time that is needed starting from an empty graph with* n *nodes is* $O(m.\alpha(m,n))$ *(where* m *is the number of edge insertions and queries),*

whereas the f^{th} operation can be performed in $O(\alpha(f,n))$ time if it is a query or a nonessential insertion. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space.

The above theorem can be augmented to allow insertion of new nodes in the graph with a time complexity of $O(n + m.\alpha(m,n))$ (cf. Section 10). Then n, m and f in the theorem denote the current number at the moment of consideration.

We can augment the 2vc-problem as follows. Note that a node x can be in several 2vc-classes. Suppose that x has a representative for each class in which it occurs. Then we can maintain this representative as follows. For a node x we partition the collection of edges incident with x in sets, so-called incidence sets, that are the intersections with the edge classes. (I.e., a set consists of the edges incident with x that all are in the same edge class.) These sets are implemented as a Union-Find structure. For each such set, its set name is the representative of the node for the corresponding 2vc-class. Note that thus an edge is element of two such Union-Find structures: one for each of its end nodes. This can be implemented by using one Union-Find structure on all the edge sides: each edge has a representative, called "side", for each of its end nodes.

A query $Is2vc(x,y)$ obtains two edges that are incident with these nodes and that are in the same edge class (if any). These edges can be used (b.m.o. the above Union-Find structure) to obtain the representatives for the common class.

The updates of the sets of edges related to a node can be done as follows. Consider the insertion of an edge (e,x,y). In case 1 of the procedure, edge (e,x,y) forms a set on its own for both x and y. In case 2 of the procedure, for each node u occurring in the joining sequence BL for procedure *joinclasses*, the incidence sets in which the edges in the sublist of u are contained, must be joined. Note that this takes only $O(1)$ additional Finds and other steps per edge in a sublist (apart from the time to join the incidence sets), yielding the same time bounds as before.

Note that we can also obtain the representative of a node for a given 2vc-class: for a class C and a node $x \in C$, the representative of x for C can be obtained by taking two nodes of C, say u and v, and then perform either $2vc(x,u)$ (if $x \neq u$) or $2vc(x,v)$ (if $x = u$). Finally, we want to remark that we do not really need the above Union-Find structure on the edge sides. For, the query $2vc(x,y)$ outputs two edges e_x and e_y incident with x and y respectively. Edge e_x is either the father edge of x in the FRT(i) structure that is used, or it is m-marked w.r.t. x. For some edge class C that has an edge incident with x, either the father edge of x is in C, or the m-marked edge in C that is in the extended subtree containing x, is incident with x. Hence a query always outputs the same edge for x with various y from a given 2vc-class C. We can take this edge as a reference to the representative of x in C. Then the only thing to do is updating these references in case of a joining of classes and in case the father and m-marks are changed. We will not give the details.

13 Concluding Remarks

We have presented solutions for the problem of maintaining the 2-edge-connected and the 3-edge-connected components of graphs and the 2-vertex-connected components of graphs under insertion of edges and vertices. The solutions take $O(n + m.\alpha(m,n))$ time, starting from the graph $< \emptyset, \emptyset >$, and are optimal on Pointer Machines and Cell Probe Machines. For 2-edge-connectivity and 2-vertex-connectivity, the optimality of solutions that run in $O(n + m.\alpha(m,n))$ time is proved in [28] (where for our results we use that the insertion of a node takes $\Omega(1)$ time). (Note that the complexity of the algorithms in [28] is $O(m'.\alpha(m',n))$, where $m' = m + n$, since we consider m to be the number of queries and edge insertions and n to be the final number of nodes, whereas m' in [28] includes both.) (Actually, the above proofs are for Pointer Machines with the Separation Condition, but by using the results of [16], the bounds follow for general Pointer Machines.) We give the proof for the 3ec-problem. Like in [28] we use reductions to the Union-Find problem. Consider the Union-Find problem for some collection of elements. For each element x there is a triple of nodes x_1, x_2 and x_3 with edges (x_1, x_2), (x_2, x_3) and (x_3, x_1). Then a query Find(x) is performed by a query $3ec\text{-}comp(x_1)$ in the graph. Moreover, the joining of two sets is as follows: for each set a triple of nodes for some element in that set is taken, say x_1, x_2 and x_3, and y_1, y_2, and y_3, and then the edges (x_1, y_1), (x_2, y_2) and (x_3, y_3) are inserted in the graph. This yields that every set corresponds to a 3ec-class in the graph. By the lower bounds for the Union-Find problem on both Pointer Machines and Cell Probe Machines [6, 16], the lower bound of $\Omega(n + m.\alpha(m,n))$ follows for the 3-edge-connectivity problem, if we use that the insertion of a node takes $\Omega(1)$ time.

Note that $\alpha(m,n) \leq 3$ for $n \leq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}65536 \text{ two's}}$. Therefore in practice there is no need to perform transformations of FRT structures like those occurring in Section 10: structures FRT(2) and FRT(3) are suited for all practical situations. The same remark holds for the UF and the GSF structures that we use: UF(i) and GSF(i) with $i \in \{2,3\}$ are suited for all practical situations too.

The time bound for an essential sequence in FRT(3) is $c.3.n.a(3,n) \leq 12.c.n$ for such n, where c is not too large a constant (cf. Sections 8 and 9 for its definition). An essential sequence on n nodes in FRT(2) takes $\leq c.2.n.a(2,n) \leq 8.n$ time for $n \leq 2^{16} = 65536$ and takes $\leq 10.c.n$ time for *very* large practical values $n \leq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}5 \text{ two's}} = 2^{65536}$. Similar remarks hold for UF(i) and GSF(i) with $i \in \{2,3\}$.

Moreover, in all practical situations for FRT(i), UF(i) and GSF(i) with $i \in \{2,3\}$ only the nontrivial Ackermann values 16 and 65536 need to available (being $A(2,3)$ and $A(2,4) = A(3,3)$) respectively), so there is no need to compute Ackermann values or to have Ackermann nets in practice (neither in the initialisation nor in case new elements are inserted like in Section 10).

Thus, in practice there is no need to perform transformations of UF, GSF, or FRT structures, and, moreover, in all practical situations there is no need to compute Ackermann values. Therefore, we conjecture that FRT(2), 2EC(2), 3EC(2) and 2VC(2) are fast structures (i.e., with practically linear time complexity) for all practical situations, with constant-time queries and constant-time nonessential insertions.

Finally we want to remark that the problem of maintaining the 3-vertex-connected components of general graphs can be solved with an (optimal) complexity of $O(n + m.\alpha(m,n))$ time for m insertions and queries for a graph with n nodes too. We refer to [19].

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. Ausiello, G.F. Italiano, A. Marchetti Spaccamela, and U. Nanni, Incremental Algorithms for Minimal Length Paths, Proc. 1^{st} Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA) 1990, 12-21.

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

[4] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook and M. Yung, Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph, Proc. 1^{st} Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA) 1990, 1-11.

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

[6] 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

[7] 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.

[8] 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.

[9] H.N. Gabow and R.E. Tarjan, A Linear Time Algorithm for a Special Case of Disjoint Set Union, J. Comput. Syst. Sci. 30 (1985), 209-221.

[10] Z. Galil and G.F.Italiano, Fully Dynamic Algorithms for Edge Connectivity Problems, Proc. 23^{th} Ann. ACM Symp. on the Theory of Computing (STOC) 1991.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] J.A. La Poutré, J. van Leeuwen and M.H. Overmars, Maintenance of 2- and 3-connected components of graphs, Part I: 2- and 3-edge-connected components, Tech. Rep. RUU-CS-26, Utrecht University.

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

[20] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, EATCS Monograph Series, Springer-Verlag, Berlin, 1984.

[21] K. Mehlhorn, Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness, EATCS Monograph Series, Springer-Verlag, Berlin, 1984.

[22] 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.

[23] 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.

[24] A. Schönhage, Storage Modification Machines, SIAM J. Comput. 9 (1980) 490-508.

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

[26] R.E. Tarjan, A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets, J. Comput. Syst. Sci. 18 (1979) 110-127.

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

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

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

J.A. La Poutré

RUU-CS-90-27
October 1990

Utrecht University

Department of Computer Science

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