SIAM J. COMPUT.
Vol. 29, No. 5, pp. 1521–1549

MAINTENANCE OF 2- AND 3-EDGE-CONNECTED COMPONENTS OF GRAPHS II*

HAN LA POUTRÉ†

Abstract. Data structures and algorithms are presented to efficiently maintain the 2- and 3-edge-connected components of a general graph, under insertions of edges and nodes in the graph. At any moment, the data structure can answer whether two nodes are 2- or 3-edge-connected. The algorithms run in $O(n+m.\alpha(m,n))$ time, where m is the total number of queries and edge insertions. Furthermore, a linear-time algorithm is presented for maintaining the 2-edge-connected components in case the initial graph is connected. Finally, a new solution is presented for the 2-vertex-connected components of a graph.

Key words. analysis of algorithms, dynamic data structures, edge connectivity, vertex connectivity

AMS subject classifications. 68P05, 68Q20, 68R10

PII. S0097539793257770

1. Introduction.

A graph algorithm is called dynamic if it maintains some information related to a graph while the graph is being changed. Dynamic algorithms are known for several graph problems. Examples are, e.g., maintenance of transitive closures [16, 17, 18, 27], minimal spanning trees [7, 8], planarity testing [3, 4, 6, 25, 33], shortest paths [1, 2, 29], k-connectivity [4, 5, 8, 9, 14, 21, 23, 24, 28, 32, 34, 35], and nearest common ancestors in trees [12].

The general problem of maintaining the k-edge- or k-vertex-connected components of a graph ($k \geq 1$) starts with an "empty" graph of n nodes[1] (i.e., a graph with no edges) and allows subsequent edge insertions and queries that ask whether two nodes are k-edge/vertex-connected.[2] For these problems, a lower bound of $\Omega(n + m.\alpha(m,n))$ [32] exists,[3] which is induced by lower bounds for set merging algorithms [10, 20]. Here m is the number of insertions and queries. So it is important to know whether there exist algorithms that actually run in this time. For $k = 2$, Westbrook and Tarjan [32] obtained the optimal running time of $O(n + m.\alpha(m,n))$. For 3-edge-connectivity, however, only combinatorial and special-case results exist. In our companion paper [21], we developed combinatorial and special-case results for 2- and 3-edge-connectivity; we will further use these in this paper. These special-case results concern maintaining the 3-edge-connectivity relation in 2-edge-connected graphs and give an implementation in $O((n + m).\alpha(m,n))$ time. In [14], Galil and Italiano obtained comparable results with this complexity for maintaining the 3-edge-connectivity relation in connected graphs.

*Received by the editors October 27, 1993; accepted for publication (in revised form) October 4, 1996; published electronically March 15, 2000. This work was done when the author was a researcher at Utrecht University and Leiden University, The Netherlands. The work was partially supported by the ESPRIT Basic Research Actions 3075 and 7141 of the EC, and partially by the Royal Netherlands Academy of Sciences and Arts (KNAW). Earlier versions of this paper occurred in Tech. report RUU-CS-90-27, Utrecht University, 1990; *Dynamic Graph Algorithms and Data Structures*, Ph.D. thesis, Utrecht University, The Netherlands, 1991.

http://www.siam.org/journals/sicomp/29-5/25777.html

†CWI (Centre for Mathematics and Computer Science), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (hlp@cwi.nl).

[1]In this paper, n is the number of nodes.

[2]For definitions, see subsection 2.5.

[3]$\alpha(m,n)$ is the inverse Ackermann function.

Although [21] presents a solution for 3-edge-connectivity which works in time $O(n \log n + m)$, it still leaves the problem whether the α-bound is achievable for the general case. The general problem appears to be substantially more difficult than the special-case problems, which can make use of the preset combinatorial structure of the special graphs. This is also sustained by a coresult of this paper, which shows that maintaining 2-edge-connected components in connected graphs can be done in linear time, while in general graphs this cannot. Therefore, the important issue remains whether the (actually more involved) 3-edge-connectivity relation can be maintained within the α-bound.

The main objectives of this paper are presenting algorithms and data structures that maintain the 3-edge-connectivity relation in general graphs with a running time of $O(n + m.\alpha(m, n))$. To achieve this, we develop extended combinatorial structures (augmented cycle forests and basic cluster trees), and we present new data structures (fractionally rooted trees). We thus construct a solution consisting of different data structure layers to maintain the 3-edge-connectivity relation. For practical applications, however, the structures seem very well suited for implementation. Furthermore, we also present a linear-time solution for maintaining 2-edge-connected components in connected graphs. Since there is a nonlinear lower bound of $\Omega(n + m.\alpha(m, n))$ for maintaining 2-*vertex*-connectivity in connected graphs, this seems to be the first result that reveals a difference in computational complexity between 2-edge- and 2-vertex-connectivity. Finally, we also give new solutions[4] for maintaining the 2-edge- and 2-vertex-connected components of a graph, which also makes use of the above data structures and with a similar running time of $O(n + m.\alpha(m, n))$. This integrates the approaches for 2- and 3-edge-connectivity and also connects those for 2- and 3-vertex-connectivity (see [24]). We remark that all our results allow the insertion of nodes as well.

The paper is organized as follows. Section 2 contains the preliminaries. In section 3, the specifications of the operations on a new data structure, called fractionally rooted trees, are given. In section 4, the maintenance of 2-edge-connected components is considered, including the special case for connected graphs. In section 5–7, the fractionally rooted tree is presented. To be precise, observations and ideas are given in section 5; the building elements for fractionally rooted trees, called division trees, are described in section 6; and the fractionally rooted trees themselves are presented in section 7. Their complexity is considered in section 8. The final results for fractionally rooted trees are in section 9. In section 10, the optimal solution for maintaining the 3-edge-connected components is presented. Furthermore, in section 11, the maintenance of 2-vertex-connected components is briefly considered. Finally, section 12 contains concluding remarks. Readers interested in the main outlines of the paper can skip sections 6.2, 8.1, and 9 (except for the theorems in section 9). Readers interested in 3-edge-connectivity only can skip subsection 4.2.

2. Preliminaries.

2.1. Graphs and terminology.

Let $G = \langle V, E \rangle$ be an undirected graph with V the set of vertices and E the set of edges. We denote an edge as a triple (e, x, y), where e is a unique edge name and x and y are the end nodes of the edge. A graph is called *empty* if it consists of nodes without edges. We use the standard terminology (see also [15]). 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

[4] Obtained independently from [32].

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. An (elementary) *cycle* is a path of which the end nodes are equal and in which no edge occurs twice. A cycle is *simple* if no node occurs twice 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; this is similar for a set of edges. Suppose the vertex set of T is partitioned into disjoint subsets, where each set induces a subtree of T. Let each induced subtree of T be contracted to a new node, called *contraction node*. For an edge (e, x, y), where x and y are contracted to p and q, $p \neq q$, the edge (e, p, q) is called the *contraction (edge)* of (e, x, y), and (e, x, y) is called the *original* of (e, p, q). (Both edges are given the same name.)

The tree CT consisting of the contraction nodes and the contraction edges is called *a contraction tree of* T. For a class D of edges in T, the class of edges in CT *inherited from* D consists of the contractions of edges in D. When we consider classes of nodes in a graph, we often refer to a class that is represented by a node c by "class c." A *singleton* class, set, or tree is a class, set, or tree that consists of one element or node, respectively. For a set or list L, $|L|$ denotes the number of elements in L. (If a sublist is attached to each element in L, then these sublists are not considered for $|L|$.)

Consider a tree T that is rooted at node r. (This just means that node r is a distinguished node.) 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 between x and 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.

2.2. Connectivity. Two nodes x and y are *k-edge-connected* ($k \geq 1$) iff there exist k edge-disjoint paths between x and y, and x and y are *k-vertex-connected* iff there exist k different vertex-disjoint paths between x and y (Menger; see [26]). It is well known that k-edge-connectivity is an equivalence relation on the set of nodes of a graph.

Henceforth, we will usually call an equivalence class for 2-edge-connectivity a 2*ec-class*, and an equivalence class for 3-edge-connectivity a 3*ec-class*. The 2-edge-connected components of a graph $G = \langle V, E\rangle$ are the subgraphs of G that are induced by the 2ec-classes, i.e., subgraph $\langle C, \{(e, x, y) \in E | x, y \in C\}\rangle$ for each 2ec-class C.

LEMMA 2.1 (see [21]). *Let $G = \langle V, E\rangle$ 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 ($k \geq 1$).*

The notion of a 3-edge-connected component can be defined such that Lemma 2.1 holds for 3-edge-connectivity too. We refer to [21]. In our observations, we will represent the 2ec-classes and the 3ec-classes of a graph by means of a "super" graph. To this end, we use the notion of a *class node*, which is a new node (or "name") that represents a class.

LEMMA 2.2 (see [21]). *Let $G = \langle V, E\rangle$ be a graph and let $k \geq 1$. Let V be partitioned into classes, where any two nodes in the same class are k-edge-connected. Let a new class node be related to each class. Let k' satisfy $1 \leq k' \leq k$. Then two nodes are k'-edge-connected in G iff the class nodes of their classes are k'-edge-connected in the graph obtained from G by contracting each class to its class node.*

We call a set S of at least two nodes a 2*vc-class* if the nodes are 2-vertex-connected, and if there does not exist a node outside 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.)

In the sequel, we will often denote 2-edge-connectivity by "2ec-," etc., when we consider components or relations. For example, 3ec-components denotes 3-edge-connected components, and 2vc-relation denotes 2-vertex-connectivity relation.

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

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

2*ec-comp*(x). Output the name of the 2ec-component (2ec-class) which contains x.

3*ec-comp*(x). Output the name of the 3ec-component (3ec-class) which contains x.

Is2*vc*(x,y). Output whether x and y are two nodes in the graph that are 2-vertex-connected and output the name of the 2vc-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. In these problems, we normally start with an empty graph with n nodes (unless stated otherwise). 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 2ec-relation changes, or, for the 3ec-problem, the 2ec- or 3ec-relation changes, or, for the 2vc-problem, the 2vc-relation changes. An insertion is called *nonessential* otherwise. Note that nonessential insertions can be omitted, which is known after a proper couple of queries. (Thus such an insertion does not need to take more than the time for those queries.)

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

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

The *row inverse* a of A and the *functional inverse* α of A are defined in correspondence to [11, 12, 19, 23] by

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

$$\alpha(m,n) = \min\{i \geq 1 | a(i,n) \leq 4.\lceil m/n \rceil\} \quad (m \geq 0,\ n \geq 1). \tag{3}$$

Here we take $\lceil 0 \rceil = 1$. For more technical insight on these functions, we refer to [19]. Here we quote that

$$a(i, A(i,x)) = x \quad (i \geq 0,\ x \geq 0), \tag{4}$$

and also that for *any* practical n, we have $\alpha(m,n) \leq 3$. Also, $A(i,1) = 2$ and $A(i,2) = 4$ $(i \geq 0)$, and $A(0,x) = 2x$, $A(1,x) = 2^x$, and $A(2,x) = \left. 2^{2^{2^{\cdot^{\cdot^{2}}}}} \right\}^{x \text{ 2s}}$.
Similarly we have

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

where the superscript (j) denotes the j consecutive applications. For simplicity, we extend the Ackermann function by $A(i,-1) = 0$ for all $i \geq 0$.

2.5. 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. Nodes and edges of a graph are represented in memory by records, which we will consider to be the actual nodes and edges. Each vertex has an incidence list consisting of pointers to the incident edges. Also, each edge contains pointers to its two end nodes. If we consider a tree T rooted at some node r, then for each node in T, its father node and father edge are related to it by appropriate pointers. An edge that has to be inserted is given by its record with the pointers to its end nodes as input for the algorithms.

In the following, the Union-Find structure is used to maintain the equivalence classes for connectivity, 2-edge-, 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. Many solutions have been proposed for the Union-Find problem [19, 30, 31]: these solutions all take $O(n + m.\alpha(m,n))$ time for all Unions and m Finds on n elements, which is optimal [10, 20]. The solution of [19] ensures that, in addition, the fth Find can be done in $O(\alpha(f,n))$ worst-case time. We call such structures *α-UF structures.* In this paper, we will also make use of a class of structures UF(i) $(i \geq 1)$, as defined in [19].

THEOREM 2.3 (see [19]). *Structure UF(i) takes $O(n.a(i,n))$ time for all Unions on n elements, where a Find takes $O(i)$ worst-case time $(i \geq 1)$.*

We consider the *connectivity problem* for edge insertions. Let $G = \langle V, E \rangle$ be a graph. Suppose a sequence of edge insertions in G and queries about whether two nodes are connected are performed. 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 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, but we will not make the above computations explicit any more.

3. Fractionally rooted trees: Concept and operations. We give a formal description of the operations supported by the data structure called *fractionally rooted tree*, 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.*

We first define some notions. Let x and y be two nodes in the same tree of F, and let P be the tree path between x and y. By "edge classes on P," we mean the edge classes of which an edge is on P. An edge class is *incident* with node x if it contains an edge with x as end node. We call a node x on P a *boundary node* of P if it is incident with two classes on P or if it is one of the end nodes of P. We call a node of P an *internal node* otherwise. A *boundary edge set* for a boundary node z on P is a set of (0, 1, or 2) edges incident with z: one from each class that is incident with z and that is on P. (See Figure 1, 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 the possible 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 a boundary edge set for it on P. (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.) 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.)

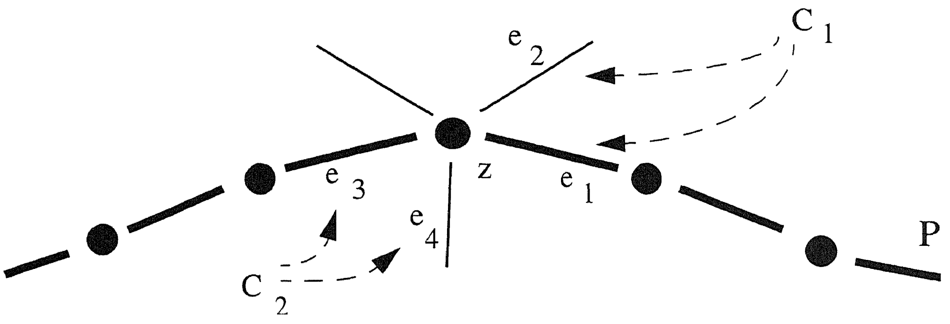

FIG. 1. *Boundary edge sets.*

We say that an edge class *occurs* in a list consisting of sublists of edges if an edge of it occurs in some sublist. A *joining list* J is a list of nodes with sublists of edges such that the union of the classes occurring in J induces some subtree in F. (Hence it yields a new admissible partition of the edge set.) In addition, the nodes in J must be the nodes incident with at least two classes occurring in J, and the sublist for each node contains an edge for each class in J incident with the node.

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), where (e, x, y) forms a new singleton class.

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.

candidates(x, y). Return an edge incident with x and return an edge incident with y; these edges (the candidates) are in the same class if such edges exist (i.e., $x \sim y$). Return the names of the edge classes in which the edges are contained.

Finally, we define some notions that will be used in the sequel. We say that a call *boundary*(x, y) is *essential* if $\neg(x \sim y)$ and it is *nonessential* if $x \sim y$. An *essential sequence* is a sequence of *link*, *boundary*, and *joinclasses*, where every call of *boundary* is essential and is followed by a call *joinclasses*(J) such that all the edge classes occurring in the output of *boundary* also occur in J. A *matching sequence* is a sequence 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.

In this section, we consider the problem of maintaining the 2ec-components in a graph, and we will present algorithms that run in $O(n + m.\alpha(m, n))$ time for n nodes and m queries and insertions using fractionally rooted trees. Thus we present a solution that is different from that given in [32] but whose approach is closer to the approach for maintaining the 3ec-relation in general graphs (section 10), and that we will use there too. We also present a linear-time solution for maintaining the 2ec-components in case the initial graph is connected.

4.1. Graph observations.

In this subsection, we recall from the companion paper [21] the observations for inserting edges in a graph $G = \langle V, E\rangle$. The set V can be partitioned into equivalence classes for 2-edge-connectivity: the 2ec-classes. Let each 2ec-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 2ec-class in which the node x is contained. We define the contracted graph $2ec(G)$ as follows:

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

For example, $2ec(G)$ is the graph that is obtained if we contract each 2ec-class into one class node. By Lemma 2.2, $2ec(G)$ is a forest (for a figure, see [21]). An edge (e, x, y) in G is called an *interconnection edge* between (classes) $2ec(x)$ and $2ec(y)$ if $2ec(x) \neq 2ec(y)$.

We consider the 2ec-relation under edge insertions by means of the graph $2ec(G)$. Suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G = \langle V, E\rangle$. We distinguish three cases and apply Lemma 2.2.

1. $c(x) \neq c(y)$. Then $(e, 2ec(x), 2ec(y))$ 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 edge $(e, 2ec(x), 2ec(y))$ connects $2ec(x)$ and $2ec(y)$ in a tree of $2ec(G)$, and all class nodes on the tree path P from $2ec(x)$ to $2ec(y)$ become 2-edge-connected in $2ec(G)$. Thus all the classes "on" P must be joined.
3. $2ec(x) = 2ec(y)$. Then nothing happens.

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 $ST(G)$. Note that a 2ec-class induces a subtree in $ST(G)$. Since the tree $ST(G)$ can be constructed in advance, we can use the Union-Find algorithms of [13] to maintain the 2-edge-connected classes: this algorithm runs in $O(n + m)$ time for m Finds for this special case. (It runs on a RAM but not on a pointer machine.) Moreover, as remarked in [19], a Find can be performed in $O(1)$ worst-case time.

We give the algorithms in case the graph G initially is a tree. We implement the tree as a rooted tree and initialize the Union-Find structure of [13] accordingly. We recall from [13] 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. If $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$, then the tree path between $2ec(x)$ and $2ec(y)$ is obtained as in [21] by traversing the root paths of $2ec(x)$ and $2ec(y)$ in $2ec(G)$ stepwise in an alternating way, where we use $ST(G)$ with the Union-Find structure as representation for $2ec(G)$. This is stopped if a class name *top* has been visited by both traversals; path P between $2ec(x)$ and $2ec(x)$ consists of the two parts of these root paths up to and including *top*.

We consider the time complexity. A computation of a tree path P is done in $O(|P|)$ time, since one of the two traversals contains nodes of P only. Since the number of classes decreases by $|P| - 1$, all path computations take $O(n)$ time altogether. All Unions and m Finds take $O(n+m)$ time. Finally, each insertion takes two Finds and $O(1)$ time, apart from the above cost.

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 initialize 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.

THEOREM 4.1. *The 2ec-problem for graphs that are initially connected can be solved such that a sequence of m insert operations takes $O(n + m)$ time, where a query takes $O(1)$ time. The structure can be initialized $O(e_0)$ time and takes $O(n)$ space, where e_0 is the number of edges in the initial graph.*

The above theorem can be augmented to allow attachment of a single new node by an edge connecting it with an existing node in the graph, within the same time complexity. (Thus, the graph remains connected.) This can be done by [13, section 3].

4.3. Algorithms and data structures for general graphs. 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. We denote the forest together with additional information (defined below) by $SF(G)$. $SF(G)$ is augmented with *edge classes* induced by the 2ec-relation.

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, which we call a *quasi class*.

An edge class that is not a quasi class is called a *real class*. Note that interconnection edges form quasi classes and vice versa.

As observed in subsection 4.2, a 2ec-class (of nodes) induces some subtree in $SF(G)$. Therefore, each edge class induces a subtree in $SF(G)$. Also, if each subtree in $SF(G)$ induced by a real edge class is contracted to some node, then we obtain the forest $2ec(G)$, where the quasi edge classes in $SF(G)$ correspond to the edges in $2ec(G)$.

We consider the insertion of edge (e,x,y). If x and y are in different trees of $SF(G)$, then these trees need to be linked. Now suppose x and y are in the same tree T of $SF(G)$. 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 neighbors on P are not both in the same 2ec-class as itself. The two neighbors 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 need to be joined.

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 the 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 (FRT) structure.

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 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.

The initialization and the queries are straightforward. The insertion of edge (e, x, y) in graph G is done by procedure $insert_{2ec}((e, x, y))$ as follows.

1. If $c(x) \neq c(y)$, then $link((e, x, y))$ is performed, and the two connected components $c(x)$ and $c(y)$ are joined (in UF_c).
2. If $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$, then the following is done. First, operation $boundary(x, y)$ is performed, returning boundary list BL. All the (node) classes in which the boundary nodes are contained are joined in UF_{2ec}. For each node z in BL, the associated edge e of z (if any) is obtained. If edge e is not in an edge class occurring in the sublist of z in BL, then e is inserted in its sublist. (This ensures that $2ec(e) = 2ec(z)$ remains true after subsequent joinings.) The end nodes of BL are removed in case their sublists contain one edge only. Then, if $BL \neq \emptyset$, operation $joinclasses(BL)$ is performed. Finally, for each node z in BL without an associated edge, an edge in its (old) sublist is made its associated edge.
3. If $2ec(x) = 2ec(y)$, nothing is done.

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 2ec-classes are joined.

OBSERVATION 4.2. *In a 2EC structure, the time needed for a sequence of essential insertions is linear to the time for a matching sequence of $O(n)$ operations on n nodes in FRT_{2ec} and for $O(n)$ Unions and Finds in UF_c and UF_{2ec}. Each nonessential insertion takes time linear to $\theta(1)$ Finds in UF_c and UF_{2ec}.*

A $2EC(i)$ structure is the structure described above, where $FRT_{2ec} = FRT(i)$ (see section 9), $UF_{2ec} = UF(i)$, and $UF_c = UF(i)$.

THEOREM 4.3. *A $2EC(i)$ structure solves the 2ec-problem such that the total time for all essential insertions is $O(n.i.a(i, n))$, where a query and a nonessential insertion can be performed in $O(i)$ time, and where the data structure can be initialized in $O(n)$ time and takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. Theorem 4.3 can be proved by Observation 4.2, Theorem 2.3, and Theorem 9.1. □

We denote the Union-Find structures UF_{2ec} and UF_c together by UF. We consider the UF structures to be one 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 [19], and take for UF the α-UF structure (see subsection 2.5). Then we obtain the following.

THEOREM 4.4. *The 2ec-problem can be solved such that the time is $O(m.\alpha(m, n))$ in total (where m is the number of edge insertions and queries), where the fth query takes $O(\alpha(f, n))$ time. The data structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

Proof. Each query and nonessential insertion corresponds to $O(1)$ Finds in the

UF structures. Moreover, all essential insertions take at most $O(n)$ Finds. Hence by [19] the fth operation is performed in $O(\alpha(f,n))$ time if it is a query or nonessential insertion. The remaining statements follow by Theorem 9.1 (with $n' \leq \min\{2m, n\}$), by (3), and by subsection 2.5. $\square$

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 9).

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 (see section 3).

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. 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 *inherited from* the edge classes of 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. Suppose $c \neq d$. Let P be the tree path between x and y in T. Let P' be the tree path between c and d in T'. Since an edge class induces a subtree, it follows that each boundary node of P' contains a boundary node of P, and the other way around. For a boundary node b on P', let P_b be the part of P inside b, and let s and t be its end nodes. Then, obviously, for a node $z \notin \{s,t\}$ contained in b, z is a boundary node of P_b iff it is one of P. If we extend P_b to P_{bb} with the other edges e_s and e_t on P incident with s and t, respectively (if they exist), then it follows that *a node contained in b is a boundary node of P_{bb} iff it is one of P.*

Now suppose that e_s exists. Then the boundary set for b contains an edge that is in the same edge class as the contraction of e_s. Let f_s be the original of this edge. Then e_s and f_s are in the same class, and, hence, the tree path connecting them consists of edges in this class only. Therefore, if we change P_b by replacing the "end edge" e_s by f_s, the boundary nodes contained inside b remain unchanged. We can do the same for t. Hence, *the boundary nodes contained in b are those contained in the local tree path between the originals of the edges in the boundary edge set of b or x or y (if x or y are contained in b).* (See Figure 2 for an illustration within T, where the subtree of T that is contracted to b in T' is surrounded by an ellipsoid.)

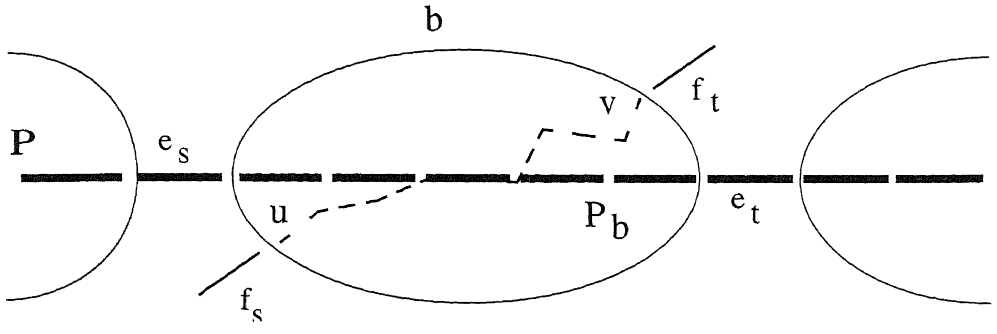

FIG. 2. *Boundary nodes in b.*

Hence we can *compute* a *boundary list B* for x and y as follows. 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 nodes u and v in b that are x, y, or end nodes of the originals of the edges in the boundary edge set of b. Subsequently, we compute the "local" boundary list $bl(b)$ for u and v. Finally, we extend the sublists of the end nodes u and v with the appropriate originals of the edges in the boundary edge set of b; if u (or v) is not a boundary node for x and y after all, then it is removed from $bl(b)$. Then these local boundary lists $bl(b)$ together form B.

6. Division trees.

6.1. Description. Division trees form the base 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, distinguished as *global edge classes*. Let T be a tree in F. Let $CT(T)$ be a contraction tree of T, where $CN(T)$ is the collection of contraction nodes and for each $b \in CN(T)$, $tree(b)$ is the subtree of T that is contracted to b. Then T together with set $CN(T)$ and with subtrees $tree(b)$ is called a *division tree*. An edge is called *internal* if it is contained in some $tree(b)$, and *external* otherwise. For a contraction node b, the *extended tree* $extree(b)$ is $tree(b)$ extended with the external edges incident with $tree(b)$. The edge set of a $extree(b)$ is partitioned into *local edge classes* induced by the global edge classes of T. This yields an admissible partition. Tree $extree(b)$ is rooted at some node.

An external edge (e, x, y) may contain different information pertaining to the two extended subtrees in which it is contained. Therefore, we distinguish two representatives called *(edge) sides*, one for each of its end nodes $(e, x, y)_x$ and $(e, x, y)_y$. For external edge (e, x, y), $(e, x, y)_x$ is the representative for $extree(contr(x))$, where $contr(x)$ is the node to which x is contracted. For internal edges, both sides are considered to be identical. We often omit referring to the proper side, however.

The class of edge e in $extree(b)$ is denoted by $class(e)$. Every edge class contains at most one edge that is marked by a so-called *d-mark*, which must be an external edge and which contains a direct pointer $d(e)$ to the name of the edge class. (So, the class name can be obtained fast.) For each edge class C in $extree(b)$, the following edges are distinguished (with direct pointers to them):

- $\max(C)$ is a maximal edge of C in the rooted tree $extree(b)$. Such an edge is then called *the* maximal edge of that class and is marked by an *m-mark* (which is done implicitly).
- $ext(C)$ is an external edge in it (if there exists any).
- $direct(C)$ is the d-marked edge in it (if it exists).

For a node x in $extree(b)$, the father edge of x or an m-marked edge incident with x is called a *preferred edge for x*. Note that for node x and a class C incident with x, there is *exactly one* preferred edge for x in C.

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 $b \in CN(T)$, $tree(b)$ is not affected by the operation. The new edge (e, x, y) forms a new singleton class on its own.

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 $tree(c)$ is extended with edge (e, x, y) and tree T_y. For example, $CN(T) = CN(T_x)$, and $tree(b)$ remain unchanged for $b \in CN(T_x) \backslash \{c\}$. The new edge (e, x, y) forms a new singleton class on its own. The edges in T_y are called *affected*.

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 just one subtree, with contraction node f. For example, afterward $CN(T) = \{f\}$. The edges in T are called *affected.*

basic-boundary(x, y). Let *contr*(x) = *contr*(y). Then return a boundary list BL for x and y, where the edges in BL are preferred.

basic-joinclasses(J). Let J be a joining list with one node such that there is at most one edge class occurring in J that contains a d-marked edge. Then join the edge classes occurring in J.

Note that for affected edges, the father relations and m-marks of these edges (edge sides) may change during these calls.

6.2. Implementation. We implement the structures as follows. A tree T in F is implemented in the common way. Each node x in T contains a pointer $contr(x)$ to the contraction node in which it is contained, and, conversely, for each contraction node b, the list *nodes*(b) consists of the nodes in *tree*(b). Similarly, each external edge has a pointer to its contraction edge and vice versa. An edge is marked external or internal. The edge classes in *extree*(b) are represented by a Union-Find structure, called the *local class Union-Find structure.* The initialization of a division tree with one contraction node is straightforward.

The operations are implemented as follows. We omit straightforward implementation details regarding, e.g., handling marks, (special) pointers, lists, etc. Note that converting an edge from external to internal may have consequences for classes, marks, and pointers.

basic-external-link$((e, x, y))$ *and basic-integrate*(x, f). The implementation of these operations is obvious. Note that maximal edges can be found by checking for each edge whether its father edge is in the same class.

basic-internal-link$((e, x, y), y)$. Let c = *contr*(x). First, *basic-integrate*(y, c) is performed, edge (e, x, y) is inserted, and x is made the father of y. Then $\max(class(e))$ is set to (e, x, y).

basic-boundary(x, y). If $x = y$, then return the boundary list BL consisting of node x with empty sublist. Otherwise, the following is done. First, two boundary lists $s(x)$ and $s(y)$ for the root paths of x and y are stepwisely computed in an alternating way, until a node *top* has been visited by both computations. This is as follows. List $s(x)$ starts with visiting node x, and a step for $s(x)$ is as follows: obtain the father edge (e, z, z') of the node z that is visited (if any), obtain the edge $\max(class(e)) = (e', u, v)$, and visit the father node of e'. Shorten the lists $s(x)$ and $s(y)$ such that they are boundary lists for x and *top* and for y and *top* respectively. Boundary list BL is created from $s(x)$ and $s(y)$, where if *top* $\notin \{x, y\}$ and the two edges related to *top* are in the same edge class, then *top* is removed from the list (since it cannot be a boundary node of P).

basic-joinclasses(J). First a list CJ is created consisting of all (names of) edge classes occurring in J. Then the classes in CJ are joined, and the edges e_d, e_m, and e_{ex} (given below) are related to this new class appropriately. Edge e_m is the maximum edge of the class of the father edge of x if this class occurs in CJ, and of any class in CJ otherwise. Edge e_d is the (unique) d-marked edge in one of the classes in CJ (if it exists), and e_{ex} is the external edge of some class in CJ (if any).

7. Fractionally rooted trees: The data structure. We present the recursive data structure called the fractionally rooted trees. We consider a dynamic forest F_0 with an admissible partition of its edge set.

Let $i \geq 1$. Let F_i consist of contractions of a number of trees in F_0. The edge set of forest F_i is partitioned into the edge classes that are inherited from the edge 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. For each tree in F_i, its data structure contains its 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 consists of a division tree, denoted by $tree(s,j)$. For layer i, $tree(s,i)$ is the tree in F_i with name s. For existing layer $j < i$, $tree(s,j)$ is the contraction of $tree(s,j+1)$, and its global edge classes are inherited from those of $tree(s,j+1)$. Finally, tree name s forms the contraction tree of $tree(s, down(s))$. (The above number $down(s)$ is only used in the description.) We denote by $tree_0(s)$ the original in F_0 of $tree(s,i)$ in F_i. To each tree name some parameters are associated, which will be given in the following sections. The structure FRT(i) allows the operations on F_i as described in section 3, where we add the parameter i to easily allow recursion. Thus we have (with the following modifications) the operations $link((e,x,y),s,t,i)$, where $x \in tree(s,i)$ and $y \in tree(t,i)$; $boundary(x,y,i)$, where the returned boundary list consists of *preferred* edges; $joinclasses(J,i)$; and $candidates(x,y,i)$, which does not return the names of edges classes, and where the returned edges are *preferred.* In addition, we have an operation $treename(x)$ that trivially outputs the name s of the tree in which a node x occurs.

For implementation purposes, we mention that the edge classes in F_0 are represented by a Union-Find structure UF_0. If FRT(i) is used as a complete structure, directly on F_0 (i.e., $F_i = F_0$), then $UF_0 = UF(i)$ (see subsection 2.5), and each operation $joinclasses(J,i)$ also joins all classes in F_0 occurring in J (in UF_0).

The structures FRT(i) are defined inductively (in terms of divisions trees). The method of induction has relations to those in [11, 12, 19, 23]. 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. For each tree name s, we have a parameter $weight(s,1)$ that contains the number of nodes in $tree(s,1)$. The local class Union-Find structure for F_1 is UF(1). FRT(1) is initialized as a forest of division trees with one contraction node each. The algorithms for the operations are as follows.

$link((e,x,y),s,t,1)$. W.l.o.g. suppose that $weight(s,1) \leq weight(t,1)$. Then $basic\text{-}internal\text{-}link((e,x,y),x)$ is performed.

$boundary(x,y,1)$. Boundary list BL is obtained by a call $basic\text{-}boundary(x,y)$.

$joinclasses(J,1)$. The joining of classes is performed by calls $basic\text{-}joinclasses(J_x)$ for each node x in J, where J_x consists of x and its sublist in J.

$candidates(x,y,1)$. Let e_x and e_y be the father edges of x and y, respectively (if they exist). Obtain the edges $m_x := \max(class(e_x))$ and $m_y := \max(class(e_y))$. If m_x is incident with y, then $e'_y := m_x$ (now e'_y is m-marked for y), otherwise $e'_y := e_y$; this is similar for e'_x. Output e'_x and e'_y. (Now e'_x and e'_y are preferred.)

We remark that procedure $candidates(x,y,1)$ yields a correct pair of edges, since if x and y are incident with the same edge class C, at least one of the father edges of x and y must be in C, and if the father edge of, say, x is not in C, then the m-marked edge of C is incident with x.

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. For each tree name

s, we keep a parameter $weight(s,i)$ that contains the number of nodes of $tree(s,i)$. Also, we have a parameter $lowindex(s,i)$ which is an integer ≥ -1 that satisfies

$$2.A(i, lowindex(s,i)) \leq weight(s,i). \tag{5}$$

(The parameter *lowindex* is incremented from time to time by the algorithms.) The Union-Find structure for local classes in F_i is UF(i).

Two cases are distinguished.

- If $weight(s,i) = 1$, then $down(s) = i$. Hence $CN(tree(s,1)) = \{s\}$.
- Otherwise, if $weight(s,i) > 1$, then $down(s) < i$. A contraction node $b \in CN(tree(s,i))$ satisfies (besides $\mid nodes(b) \mid \geq 2$)

  $$\mid nodes(b) \mid \geq 2.A(i, lowindex(s,i)). \tag{6}$$

  If layer i is removed, then the remaining part, starting from $tree(s, i-1)$ in layer $i-1$, is an FRT$(i-1)$-structure. For an external edge (e,x,y) in $tree(s,i)$, side $(e,x,y)_x$ is *d-marked* if its contraction edge is preferred for $contr(x)$.

Note that every edge class C in $extree(b)$ for some $b \in CN(tree(s,i))$ contains at most one d-marked edge, since every edge class in $tree(s,i-1)$ contains at most one preferred edge incident with b.

7.2.1. Implementation. The initialization is done by initializing a forest of division trees with one contraction node each. For singleton trees, the contraction node is the tree name, where for nonsingleton trees, new tree names are recursively related to them in the next layer. All the corresponding *lowindex*-values are set to -1.

We give the algorithms for the operations. Note that, by (5), $lowindex(s,i) \geq 0$ implies that $down(s) < i$.

$link((e,x,y),s,t,i)$. W.l.o.g., we assume that $lowindex(s,i) \geq lowindex(t,i)$.

Let $newweight := weight(s,i) + weight(t,i)$ and let $ls := lowindex(s,i)$. There are three cases. (For more intuition behind this operation, we refer to the comments and figures in [19].)

- $lowindex(s,i) > lowindex(t,i)$. A call $basic\text{-}internal\text{-}link((e,x,y),y)$ is performed. (Now, $tree(t,i)$ is contracted to $contr(x)$.) Then t and its related layers j with $j < i$ are disposed.
- $lowindex(s,i) = lowindex(t,i) \wedge newweight \geq 2.A(i, ls+1)$. Then a *new* contraction node f is created in layer $i-1$. Then $basic\text{-}external\text{-}link(e,x,y)$ and $basic - integrate(x,f)$ are called, and $contr(f) := s$. (Now, $tree(f)$ consists of the former $tree(s,i)$, $tree(t,i)$, and (e,x,y).) The old existing layers j related to s and t with $j < i$ are disposed, including tree name t. Finally, $lowindex(s,i) := lowindex(s,i) + 1$ and $lowindex(s,i-1) := -1$.
- $lowindex(s,i) = lowindex(t,i) \wedge newweight < 2.A(i, ls+1)$. Then we want to do the actual linking on a lower layer. Therefore, first $basic\text{-}external\text{-}link((e,x,y))$ is executed. Then the contraction edge (e,c,d) of (e,x,y) is created, and a recursive call $link((e,c,d),s,t,i-1)$ is performed, where all the affected edges in layer $i-1$ are obtained. For each edge (e',u,v) in layer i that is (e,x,y) or the original of an affected edge in layer $i-1$, the d-marks are updated: if its contraction edge is preferred for $contr(u)$, then $(e',u,v)_u$ is d-marked; otherwise, $(e',u,v)_u$ is un-d-marked. The same is done for v.

boundary(x, y, i). Perform *candidates*(x, y, i) yielding edges e_x and e_y. If e_x and e_y are in the same edge classes in F_0, then the desired boundary list consists of x and y with e_x and e_y in their sublists. Otherwise, let $c = contr(x)$ and $d = contr(y)$. Then the boundary list BB for c and d in layer $i-1$ is (recursively) computed: if $c = d$, then BB contains just c, and otherwise a recursive call *boundary*$(c, d, i-1)$ is performed, returning BB. For each boundary node b in BB, we obtain the nodes u and v in *tree*(b) that are x, y, or end nodes of the originals of the edges in the boundary edge set of b. Then a "local" boundary list for u and v in *tree*(b) is computed by *basic-boundary*(u, v), where the sublists of the end nodes u and v are extended with the originals of the (at most 2) appropriate edges in BB: if u (or v) is not a boundary node for x and y after all, then it is removed from the local list. These local boundary lists are concatenated, yielding the desired boundary list.

joinclasses(J, i). First a list JJ for layer $i-1$ is made, consisting of the nodes $contr(x)$ for nodes $x \in J$, where the sublist for c is the concatenation of all sublists for $x \in J$ with $contr(x) = c$. Then, for each node $c \in JJ$, the classes occurring in its sublist are obtained, and its sublist is replaced by a sublist that contains for each such class one external edge (if any). All nodes of JJ with a sublist containing at most one edge are removed. If $JJ \neq \emptyset$, then *joinclasses*$(JJ, i-1)$ is called. All the original edge sides of the edges that have been un-m-marked in layer $i-1$ are un-d-marked in layer i. Finally, for each node x in J, *basic-joinclasses*(J_x) is executed, where J_x contains x and its sublist in J.

candidates(x, y, i). Let $c = contr(x)$ and $d = contr(y)$. If $c = d$, then do the same as for $i = 1$. Otherwise, perform candidates$(c, d, i-1)$ that returns the (preferred) edges e_c and e_d. Let edge $e_1 \in extree(c)$ be the (d-marked) original of e_c. Let $e_2 := \max(d(e_1))$. If e_2 is incident with x, then $e_x := e_2$ (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. This is a correct pair of edges, which follows by the specification of *candidates*$(i-1)$ and similar observations as for $i = 1$. Note that by using $d(e_1)$ instead of $class(e_1)$, we need to follow *one* pointer only, instead of performing a Find.

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 for s. For further details and for the problem of how to obtain Ackermann values for all the structures (viz., by means of one "Ackermann net" for $2n$), we refer to [19].

8. Complexity of FRT(i).

We consider the time and space complexity of FRT(i) structures ($i \geq 1$). In the notation, we omit the procedure parameters except for the layer number i. Operations *treename* and *candidates*(i) can be performed in $O(i)$ time, and a nonessential call *boundary*(i) can be done in $O(i)$ time plus $O(1)$ Finds in UF_0. For *candidates*, this is seen as follows. If $contr(x) = contr(y)$, it takes one Find in UF(i), which is $O(i)$ time. Otherwise, all instructions except for the recursive call can be done in constant time (because of the d-marks and preferred edges), giving $O(i)$ time by induction. For a nonessential call *boundary*, we see that if $i = 1$ then $x \sim y$, and thus *basic-boundary*(x, y) is similar to *candidates*$(x, y, 1)$, while if $i > 1$, then *candidates*(x, y, i) is executed together with two Finds.

In the sequel, we consider the complexity of essential sequences (see section 3). 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. The proof is given in subsection 8.1 (which can be skipped at first reading). Note that if $F_i = F_0$, then $UF_0 = UF(i)$ and the set unions take $O(n.a(i,n))$ time (see Theorem 2.3).

LEMMA 8.1. *An essential sequence in an FRT(i) structure with n nodes needs a total of $O(n.a(i,n))$ steps, except for joining edge classes in F_0 ($i \geq 1$, $n \geq 2$).*

8.1. Proof of Lemma 8.1. Lemma 8.1 is proved by induction in a way similar to the proof in [19]. We first consider the *net cost* of the basic operations, i.e., the cost of the operations *except* for the cost of set unions. Then *basic-integrate*(y, f) and *basic-internal-link*$((e,x,y),y)$ take net $O(|T_y|)$ steps, where T_y is the tree containing y; *basic-external-link*$((e,x,y))$ takes net $O(1)$ steps. A call *basic-boundary*(x,y) takes $O(|BL|)$ steps if BL is the resulting boundary list, and *basic-joinclasses*(J) takes $O(|E_J|)$ net steps, where E_J is the number of edges in J.

We now consider the complexity of the structures FRT(i). As in [19], we do not need to consider the complexity of storing and obtaining the information for each layer related to a tree name, since this can be charged easily to other operations. We show that an essential sequence in FRT(i) takes $O(n.a(i,n))$ steps on n nodes (except for the cost on F_0). Moreover, we show that the number of times that an edge becomes affected (see section 6) is at most $a(i,n)$. We prove all this by considering the procedures *link*(i), (essential) *boundary*(i), and *joinclasses*(i), where the cost of set union or essential recursive calls is considered separately. Here an *essential recursive call* is any recursive call of these procedures with the restriction that recursive *boundary* calls are essential.

8.1.1. FRT(1). We consider the cost of an essential sequence on n nodes ($n > 1$) in FRT(1) by determining for each procedure the cost of all its calls.

Procedure *link*$((e,x,y),s,t,1)$ takes at most $O(|weight(t,1)|)$ steps, where, w.l.o.g. *tree*(t,i) is the smallest of the two sets to be joined. Charge each node in *tree*$(t,1)$ for $O(1)$ cost. Since the nodes in *tree*$(t,1)$ become elements of a tree with at least double size, all calls take at most $\lfloor \log n \rfloor \leq a(1,n)$ steps together. Similarly, the number of times that an edge is affected is at most $a(1,n)$.

A call *boundary*$(x,y,1)$ takes $O(|BL|)$ steps. Note that at least $|BL|-1$ different classes occur in BL. Charge $O(1)$ cost to the encountered classes. In the essential sequence, all these classes are subsequently joined by a call *joinclasses*. This gives at most $O(n)$ steps.

Procedure call *joinclasses*$(J,1)$ takes $O(1)$ steps for each class that is joined; thus the total amount of steps is $O(n)$ steps apart from the joinings.

Finally, since there are at most $2n$ edge sides, the time for set unions in UF(1) is $O(n.a(1,n))$ (Theorem 2.3).

Therefore, 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 $a(1,n)$.

8.1.2. FRT(i) for $i > 1$. We consider the cost for an essential sequence on n nodes ($n > 1$) 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 in an essential sequence on k nodes ($k > 1$), 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

$a(i-1,k)$. For each procedure or specific part of the computation, we determine the cost of all its calls.

For an essential call of *boundary*(i), we have the following. First, the calls of *candidates*(i) and the recursive call *boundary*$(i-1)$ take $O(1)$ net steps. (The call *boundary*$(i-1)$ takes net $O(1)$ steps if it is nonessential and it takes no steps if it is essential.) Then local boundary lists are computed and manipulated (but do not become empty, see section 5). Hence the net cost is $O(|BL|)$ steps. Since afterward all classes occurring in BL must be joined by a call of *joinclasses* in the essential sequence, it follows that the total net amount of steps is $O(n)$.

Procedure *joinclasses*(i) takes a net number of steps linear to the number of classes that will be joined, apart from the recursive call. Hence this is $O(n)$ in total.

We divide *link* $((e,x,y),s,t,i)$ into several parts and compute the net cost of each of these parts for all executions together. First, the removal of parts of structures can be charged to their creation. Second, the calls of procedure *basic-internal-link* and *basic-integrate* take at most $O(\textit{the number of processed nodes})$ steps. Therefore, we charge these steps to the processed nodes. Note that in both cases the processed nodes will (henceforth) be contained in a new tree with higher *lowindex* value and that there are at most $a(i,\lceil\frac{n+1}{2}\rceil)+2 \le 3.a(i,n)$ different *lowindex* values (cf. (5)). Therefore, the total cost of these calls is $O(n.a(i,n))$ steps. Similarly, it follows that an edge is affected is at most $a(i,n)$ times. Third, the cost for changing d-marks of edges in procedure *link*(i) is linear to the number of times that contraction edges are affected in the recursive call *link*$(i-1)$. In Observation 8.4 we will show that this is at most $\frac{1}{2}.n.a(i,n)$. Hence this takes $O(n.a(i,n))$ steps altogether. Last, the rest of the procedure requires $O(1)$ net time per call of link(i), which gives $O(n)$ time altogether. In conclusion, all calls of *link* take at most $O(n.a(i,n))$ steps net and affect an edge at most $a(i,n)$ times.

The required time for set unions in UF(i) is $O(n.a(i,n))$ (Theorem 2.3), since there are at most $2n$ edge sides.

Finally, we consider the essential recursive calls (performed on contraction nodes). We first have two observations (the latter can be proved as in [19]).

OBSERVATION 8.2. *The operations on contraction trees (for layer i) by procedure link$((e,x,y),i)$ are the creation of a singleton tree and the linking and removal of trees; procedures joinclasses(i) and boundary(i) only change edge classes in contraction trees.*

OBSERVATION 8.3. *In link$((e,x,y),s,t,i)$, a recursive call is performed only if*

$$\begin{aligned}&1 < lowindex(s,i) = lowindex(t,i) \le a(i,n)\\ &\wedge weight(s,i) + weight(t,i) < 2.A(i, lowindex(s,i)+1).\end{aligned}$$

For a contraction node $c \in CN(tree(s,i))$, we denote by *lowindex*(c) the value *lowindex*(s,i), which is fixed during its existence. We call c an l-contraction node if *lowindex*$(c) = l$. Similarly, we say that a 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*$(i-1)$ or *joinclasses*$(i-1)$ 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 vice versa. We compute the cost of all l-calls for fixed value l, $-1 \le l \le a(i,n)$. Note that any tree of l-contraction nodes with $l \le 0$ consists of one contraction node. Hence an l-call of *boundary*$(i-1)$ and *joinclasses*$(i-1)$ occurs only if $l \ge 1$. By Observation 8.3 and since $|nodes(b)| \ge 2$ for each contraction node b, it follows in case of an l-call *link* $(s,t,i-1)$ that $l > 1$, and that $weight(s,i-1)+$

$weight(t, i-1) < A(i, l+1)$. By this and by Observation 8.2, the maximal size of any tree of l-contraction nodes is $< A(i, l+1)$.

Now let l be fixed number with $1 \leq l \leq a(i, n)$. Partition the total collection of all l-contraction nodes involved in l-calls into the existing maximal sets. Then the size of such a maximal set is at most $A(i, l+1)$. It easily follows that 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 set of k contraction nodes, the cost of all (previous) 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-cluster nodes*)$.\ a(i-1, A(i, l+1))$. By (6), there are at most $n/(2.A(i,l))$ l-contraction nodes. Therefore, this cost is at most $c.\frac{n}{2.A(i,l)} \cdot a(i-1, A(i, l+1))$, which is at most $\frac{1}{2}c.n$ by using $i > 1$ and equations (1) (on $A(i, l+1)$) and (4), respectively.

Since at most $a(i, n)$ values l of *lowindex* occur, the cost of all these FRT$(i-1)$-calls is at most $\frac{1}{2}c.n.a(i.n)$.

Similar to the above, by the induction hypothesis, the number of times that l-contraction edges are affected in the l-calls *link*$(i-1)$ is at most $\frac{1}{2}.n$ for fixed l.

OBSERVATION 8.4. *The number of times that contraction edges are affected in recursive calls at most link*$(i-1)$ *is* $\frac{1}{2}.n.a(i, n)$.

Combining all the above results yields that the total number of steps is at most $c_1.n + c_2.n.a(i, n) + \frac{1}{2}c.n.a(i, n)$ for some c_1 and c_2 (independent of c). By taking $c = \max\{d, 2.(c_1 + c_2)\}$, it follows by induction that an essential sequence in FRT(i) takes at most $c.n.a(i, n)$ steps and affects an edge at most $a(i, n)$ times. This concludes the proof of Lemma 8.1.

9. FRT structures.

We consider FRT(i)-structures with $F_i = F_0$ and express the operations of section 3 in terms of section 7. It is easily seen how to use the latter for the former; we may only need an additional call of *treename* or $O(1)$ Finds in UF$_0$ for the proper result. Thus Lemma 8.1 remains valid for the operations in section 3 (in order of magnitude). Note that by Theorem 2.3, a step, as defined in the previous subsection, is $O(i)$ time.

THEOREM 9.1. *An essential sequence in FRT(i) on n nodes needs a total time of* $O(n.i.a(i, n))$ $(i \geq 1,\ n \geq 2)$. *Each candidates operation and each nonessential call boundary takes* $O(i)$ *time. The structure can be initialized in* $O(n)$ *time and takes* $O(n)$ *space.*

Note that if n' is the number of nodes that are not still contained in singleton trees after the execution of the above sequence (thus $n' \leq n$), then the total time is even $O(n'.i.a(i, n'))$. Also, the theorem 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 (see also [19]).

We define an α-FRT structure as follows. Initially, FRT$(\alpha(n, n))$ is used. From time to time, a transformation is performed, replacing an FRT(i) structure by an FRT$(i-1)$ structure, viz., each time that $\alpha(q, n)$ decreases by one, where at any moment q is the number of queries *candidates* performed until then. This is performed in a way similar to the proof of Theorem 5.2 of [19] (full paper), where now the query *candidates* plays the role of the Finds, and where *link* and *joinclasses* play the role of the Union operations. The building of a new FRT$(i-1)$ is done similar to Theorem 5.2 in [19], but instead of building just parts of FRT$(i-1)$ during *candidates* operations, we have for all pointers in F_0 two versions, and we build and handle FRT$(i-1)$ with the unused pointer version. (This duplication is only relevant in case we want a single *candidates* query to have $O(\alpha(q,n))$ worst-case time.) Then we obtain the following result.

THEOREM 9.2. *Let an α-FRT structure for an "empty" forest with n nodes be given. Then a matching sequence in α-FRT needs a total of $O((n+m).\alpha(m,n))$ time (where m is the number of operations candidates and boundary that is performed), where the qth call of candidates takes $O(\alpha(q,n))$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

Proof. The proof is related to the proof of Theorem 5.2 in [19]. We leave this as an exercise for the reader (and refer to [22, 23]). □

Note that if n' is defined as before (and, hence, the essential subsequence of the matching sequence consists of $\theta(n')$ operations), then the total time is even $O((n'+m).\alpha(m,n))$ time. Also, by using the same transformation techniques as in Theorem 6.2 in [19], the above theorem can be extended with the insertion of new (isolated) nodes in the structure with the corresponding complexity bound $O((n+m).\alpha(m,n))$ (where m and n denote the current number at the time of consideration), where the insertion of a new node takes $O(1)$ time. We will not give details here but refer to [22, 23]. (We want to remark that if at any time $m = O(n)$, as for the 2ec-and the 3ec-problem, then only rebuildings from FRT(i) to FRT($i+1$) are needed.)

In practice there is no need to perform transformations of FRT-structures or to compute Ackermann values [19]. This is because $\alpha(m,n) \leq 3$ for any practical n. Thus, structures FRT(i) with $i \in \{2,3\}$ are suited for all practical situations and only need the nontrivial Ackermann values $A(2,3) = 16$ and $A(2,4) = A(3,3) = 65536$. An essential sequence in FRT(2) takes $\leq c.2.n.a(2,n) = 2cn.\log^* n$ time, which is $\leq 8cn$ for $n \leq 2^{16}$ and $\leq 10cn$ for $n \leq 2^{65536}$, where c is not too large a constant (see section 8). Therefore, we conjecture that FRT(2) can be implemented as a fast structure for all *practical* situations, with constant-time queries.

10. 3-edge-connectivity. We will now extend the results to the maintenance of 3ec-components in a graph, with a time complexity of $O(n + m.\alpha(m,n))$ for n nodes and m queries and insertions. In subsection 10.1 we consider maintaining the 3ec-relation within 2-edge-connected graphs and, subsequently, in subsection 10.2 we consider the problem for general graphs.

Let $G = \langle V, E\rangle$ be a graph. The set V can be partitioned into equivalence classes for 3-edge-connectivity, called *3ec-classes*. Each 3ec-class C is represented by a new 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.

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

Hence, $3ec(G)$ is the graph that is obtained if we contract each 3ec-class into one class node (see Figure 3 if G is 2-edge-connected). No two nodes in $3ec(G)$ are 3-edge-connected (by Lemma 2.2).

10.1. 2-edge-connected graphs. In this subsection, we suppose that graph G is 2-edge-connected, and we state results from the companion paper [21]. Every two distinct class nodes must lie on a common elementary cycle in $3ec(G)$, while simple cycles in $3ec(G)$ cannot intersect in more than one class node.

Let $Cyc(3ec(G))$ be the graph that is constructed from $3ec(G)$ as follows. Each nontrivial simple cycle is represented by a distinct node called a *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. The graph $Cyc(3ec(G))$ 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 $3ec(G)$. Therefore, graph $Cyc(3ec(G))$ shows the incidence relation for class nodes and cycles. Moreover, graph $Cyc(3ec(G))$ is a tree called the *cycle tree* of G. The structure of $Cyc(3ec(G))$ is illustrated in Figure 3, where the cycle nodes are drawn as boxes.

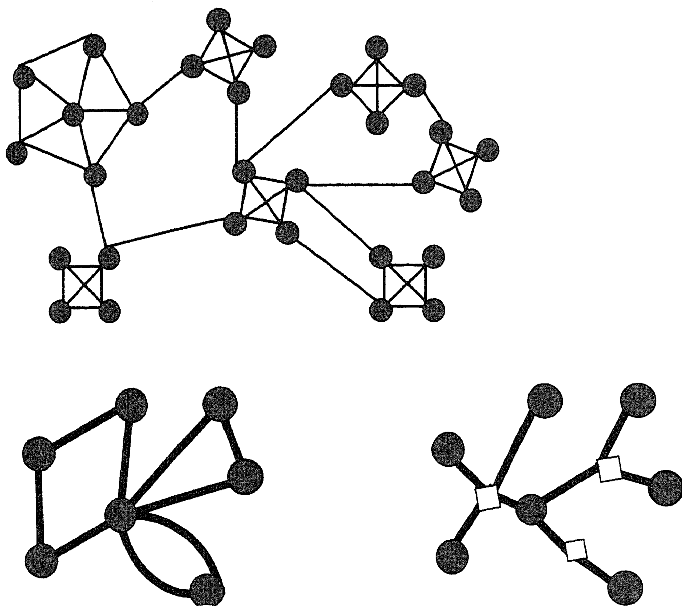
FIG. 3. *A* 2*-edge-connected graph G and the related graphs* $3ec(G)$ *and* $Cyc(3ec(G))$.

10.1.1. Edge insertions. We maintain the 3ec-relation under edge insertions by means of $Cyc(3ec(G))$. Suppose a new edge (e,x,y) is inserted in G. If $3ec(x) = 3ec(y)$, then by Lemma 2.2, the 3ec-relation, $3ec(G)$, 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 a new edge in $3ec(G)$.

LEMMA 10.1 (see [21]). *Let G be a* 2*-edge-connected graph. Suppose that an edge* $(e, 3ec(x), 3ec(y))$ *is inserted to the graph* $3ec(G)$. *Then all the class nodes on the tree path P 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.*

Thus, for all class nodes on P, all the corresponding classes form a new class (by Lemma 2.2). The update can now be performed as follows:

- obtain the tree path P between $3ec(x)$ and $3ec(y)$ in $Cyc(3ec(G))$,
- join all the classes "on" P into one new class C', and
- adapt the cycle tree $Cyc(3ec(G))$ accordingly.

The update heavily changes the structure of $Cyc(3ec(G))$. (For illustrations, we refer to [21].) 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' and of one of the two parts of the former cycle *between* c and d (we refer to [21]).

LEMMA 10.2 (see [21]). *Given a* 2*-edge-connected graph G of n nodes with a cycle tree, there exists a data structure for the* 3*ec-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.*

Here, the *Circular Split-Find problem* [23] is a problem closely related to the Split-Find problem [11]. It deals with splitting cyclic lists into two new cyclic lists, determined by two splitting nodes. In [23], solutions for this problem are given with similar complexities as the UF-structures (cf. section 2.5). (They are closely related to [11].) We denote such structures similarly by GSF(i) and α-GSF structures. Later, we choose appropriate structures when applying Lemma 10.2.

10.2. General graphs.

10.2.1. Observations. We extend the solution of the previous section to general graphs. We first state observations of [21] (to which we refer for further details and figures). For detecting the 3ec-classes it suffices to detect the 3ec-classes inside the 2ec-components. Therefore, our algorithms for general graphs maintain the 2ec-classes (as in section 4), and they maintain the 3ec-classes within 2ec-components. We consider the forest of all cycle trees for the 2ec-components, called the *cycle forest* $Cyc(3ec(G))$ of G.

Suppose edge (e, x, y) is inserted in graph G yielding graph G'. 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 2ec-component and the changes as described in subsection 10.1.1 occur. Otherwise, we have $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$. Consider $2ec(G)$. Let P_2 be the tree path between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ (see subsection 4.1). Then the major changes are that

1. all 2ec-classes corresponding to class nodes on P_2 form one new 2ec-class,
2. for each 2ec-class C on P_2, the 3ec-classes inside C are changed, and
3. a new cycle s of 3ec-classes arises; the new cycle node s links the (updated) cycle trees that are contained in the 2ec-classes on P_2.

We consider the changes more precisely. The first part is identical to subsection 4.1. For the second part, we consider the changes of the 3ec-classes that occur in 2ec-classes on P_2. Consider a 2ec-class C on P_2 in $2ec(G)$. Let u and v be the two nodes in C that are x, y, or end nodes of interconnection edges between C and other classes on P. (We call u and v the *interconnection nodes* for C.) 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 previously exist in G. Hence, considered within C only, this corresponds to inserting a temporary edge between the nodes u and v, since the 3ec-classes are completely determined by the 2ec-components in which they are contained (see Lemma 2.1). For the third part, now suppose all these "local" insertions are performed in the 2ec-classes on P_2. Then the two interconnection nodes in a 2ec-class C on P_2 are in the same (updated) 3ec-class in C, called the *interconnection* 3*ec-class* in C. All these interconnection 3ec-classes form a new cycle s. Then in the cycle forest s must be linked to these interconnection 3ec-classes, and thus it links the corresponding cycle trees.

10.2.2. Data structures and approach. We observe that when an edge (e, x, y) is inserted in a 2ec-component H, the changes in the 3ec-relation and $Cyc(3ec(H))$ are fully determined by just the 3ec-classes in which x and y are contained.

Consider a graph $G = \langle V, E\rangle$. We change the cycle forest $Cyc(3ec(G))$ by augmenting the collection of nodes of G and 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 (conceivably by means of artificial edges). 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 called the *subclass node*. We call these the subclass nodes for C. The subclass node of the subclass to which x belongs is denoted by $sub(x)$. Now an *augmented cycle forest* AF_G for G is a forest on the subclass nodes and the cycle nodes of $Cyc(3ec(G))$ such that for each 3ec-class C of G the subclass nodes for C induce a subtree of AF_G and such that $Cyc(3ec(G))$ is obtained if for each 3ec-class C its subclass nodes are contracted into one node. We call an edge that links two subclass nodes of a 3ec-class C a *connector* for 3ec-class C. The set of all the connectors for C is called the *connector class* for 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 4, where cycle nodes are drawn as boxes and (sub)class nodes as dots.

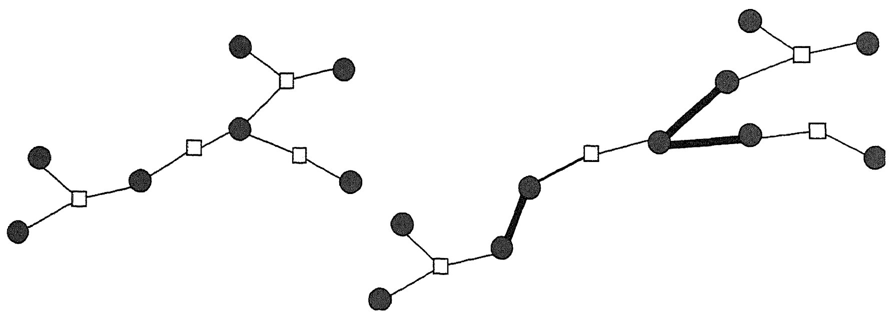
FIG. 4. *Graphs* $Cyc(3ec(G))$ *and* AF_G.

We consider the insertion of an edge (e, x, y) in a 2-edge-connected graph G in terms of AF_G. Let $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. The 3ec-classes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ correspond to the 3ec-classes that have at least one subclass on the tree path P between $sub(x)$ and $sub(y)$ in AF_G. Hence we can update the structure according to the following observations (also cf. subsection 10.1.1).

- Two successive subclass nodes on P (without a cycle node in between) correspond to the same class. Hence it suffices to obtain just 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. Hence, all subclass nodes for C' must form a (sub)tree. This can be done by splitting each cycle s occurring on P and by joining the two subclasses that are the neighbors of s on P. (These updates can be performed locally as for cycle trees for each part of P without adjacent subclass nodes.)

Our goal structure is now as follows. To a graph G we relate a forest $bc(G)$ and an augmented cycle forest AF_G that satisfy the following. The graph $G = \langle V, E\rangle$ is extended with an (incremental) collection of auxiliary nodes, which are 3-edge-connected to least one original node. The (thus extended) vertex set is partitioned into disjoint sets, called *basic-clusters.* Each basic-cluster has a (new) unique node 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 by intersecting C with the basic clusters. Then AF_G is an augmented cycle forest for G, based on this partition into subclasses.
- 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 $tree(b)$.
- Every connector in AF_G corresponds to exactly one edge in $bc(G)$ (and vice versa).

It follows that for a cluster b, $tree(b)$ does not have two adjacent subclass nodes. Therefore, $tree(b)$ is a cycle tree of some 2-edge-connected graph that has the nodes of basic-cluster b as its nodes together with a number of appropriate edges that induce the 3ec-relation as represented by $tree(b)$.

We observe that $bc(G)$ can be obtained from AF_G by contracting all subclass nodes in a basic-cluster b to cluster node b. Thus $bc(G)$ is an other contraction of AF_G (different from $Cyc(G)$). See Figure 5 for the example of Figure 4. We now define edge classes on $bc(G)$ as the classes inherited from the connector classes in AF_G (see section 2.1). Note that if two bc-edges incident with a cluster node b are in the same bc-edge class, then their originals in AF_G must have the same end node (a subclass node) in cluster b.

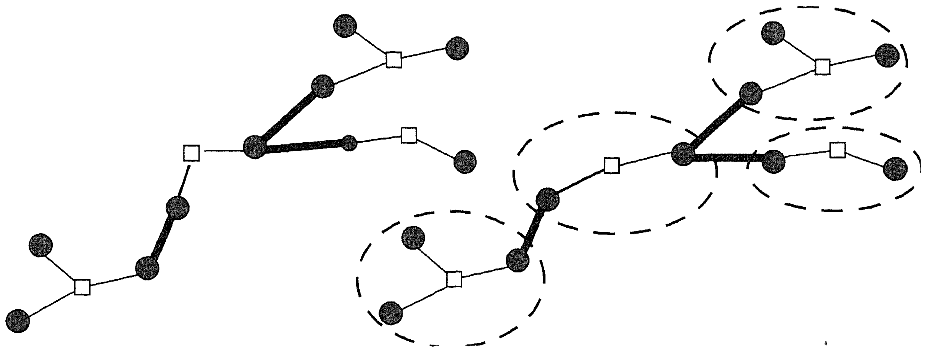
FIG. 5. *Forests AF_G and $bc(G)$.*

Now the strategy for inserting an edge (e, x, y) in 2-edge-connected graph G can be put in terms of $bc(G)$ as follows. Let c and d be the basic clusters containing x and y, respectively. 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 $sub(y)$. To obtain the relevant parts of P, it suffices to obtain a boundary list BL for c and d in $bc(G)$. This is seen as follows. The two incident bc-edges of an internal node b on P' are in the same bc-edge class. Hence their originals (which lie on P) have the same end node sb in b, which, therefore, is not adjacent to a cycle node on P. Note that for a boundary node $b \notin \{c, d\}$ of P', its two incident bc-edges on P' are not in the same bc-edge class, and thus cluster b contains at least two subclass nodes and one cycle node of P.

- 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. The end nodes sb_1 and sb_2 of P_b are $sub(x)$, $sub(y)$, or the end nodes in cluster b of the originals of the bc-edges on P'. The latter can also be obtained from the bc-edges in the sublist of b in BL. The update thus corresponds to the update for inserting a temporary edge between any two nodes of G that are contained in subclasses sb_1 and sb_2.

10.2.3. Data structures and algorithms. In this section, we describe a data structure for the 3ec-problem, called *3EC structure*. We distinguish between the different layers of *representation*.

The representation of graph G is as follows. The vertex set of G may be extended from time to time with auxiliary nodes. 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 a "global" Union-Find structure UF_{3ec} for implementing the 3ec-classes of nodes of G. We recall that in the $2EC$ structure, there are Union-Find structures UF_c and UF_{2ec}.

Each 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}.

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. Instead of a subclass node s as the end node of a connector, we take a node in subclass s as an end node. This is because subclasses are joined from time to time. Then the subclasses that are the ends of a connector (e, x, y) are $sub(x)$ and $sub(y)$. For a basic-cluster b, $tree(b)$ is implemented and maintained as a cycle tree as in Lemma 10.2. We refer to this as the *local structure*. The Union-Find and Circular Split-Find structures used in the local structure are denoted by UF_{loc} and GSF_{loc}. To each subclass, we relate a connector that has one of its end nodes in that subclass (if it exists) its *associated connector*.

The initialization for an empty graph is straightforward: each basic cluster contains one node. Also, a query corresponds to a Find in UF_{3ec}.

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 $3ec(x) \neq 3ec(y)$.

1. $c(x) \neq c(y)$. Then $insert_2((e, x, y))$ is performed
2. $c(x) = c(y) \wedge 2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. If $c = d$, then list 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 following is done. First, the nodes u and v in $tree(b)$ are obtained that are x, y, or end nodes of the originals of the bc-edges in the sublist of b (if any). If $3ec(u) \neq 3ec(v)$, then the following is done. A local insertion (Lemma 10.2) of a temporary edge (e', u, v) in basic-cluster b is performed to update $tree(b)$. Then an associated connector is obtained for each of the subclasses that are joined in cluster b, and the corresponding bc-edges are put in the sublist for b in J (since all classes must be joined later in the global structure). One of these connectors (if any) is assigned to the resulting subclass as its associated edge.

All the 3ec-classes of which a subclass was involved in the joinings are joined in UF_{3ec} (e.g., by taking a node from each subclass). Finally, the FRT_{3ec} structure is updated by means of call $joinclasses(J)$ (where if the sublist of node c or d is empty, then this node is removed from J.)

3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. First, 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 only the first part of the call $insert_2((e, x, y))$). Subsequently, the names of the 2ec-classes are obtained. Then a linear list L is constructed that consists of these names and of those edges in BL with end nodes in different 2ec-classes: the names and edges alternate in L such that an edge is bracketed by the two corresponding 2ec-classes. Now L contains the tree path between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ in the proper order.
For each 2ec-class C in L the following is done. We obtain the nodes u and v in C that are x, y, or the end nodes of the surrounding edges in L. If $3ec(u) \neq 3ec(v)$ then a temporary edge between u and v in C is inserted by a call $insert_3((e', u, v))$. Afterward, a new, auxiliary node z_C is created, and it is inserted in the interconnection 3ec-class $3ec(u)$. A connector (e'_C, z_C, z'_C) is created between z_C and some node z'_C of 3ec-class $3ec(u)$.
The nodes z_C $(C \in L)$ together form a new basic cluster b. Thus a cycle tree corresponding to the cycle of the new subclasses $\{z_C\}$ $(C \in L)$ is initialized in cluster b (in the same order as the 2ec-components C in L).
Cluster node b is linked with the involved trees in $bc(G)$ by means of new bc-edges as follows. For each z_C and connector (e'_C, z_C, z'_C), let $b' = clus(z'_C)$. Then a bc-edge (e'_C, b, b') is created for this connector, and a call $link((e'_C, b, b'))$ is performed in $bc(G)$. Edge (e'_C, z_C, z'_C) is associated with $sub(z_C)$ and $sub(z'_C)$. If $sub(z'_C)$ already had an associated edge (e'', z'', z'''), then (e'_C, b, b') is put in the class containing $(e'', clus(z''), clus(z'''))$ by a call of $joinclasses$.
The 2ec-classes in L are joined by performing a real call $insert_2((e, x, y))$ in $2EC$.

10.2.4. Complexity. We consider the complexity of the above algorithm. Regarding the creation of auxiliary nodes, suppose the initial graph G_0 has n (regular) nodes. The total number of new nodes created by the algorithm is at most $2n-1$, since a new node is created for each 2ec-class that is joined. Similarly, the total number of created clusters is at most $n-1$. Hence we only need FRT and Union-Find structures for $O(n)$ nodes. We denote all the Union-Find structures used independently in 3EC (not as part of FRT_{3ec} or FRT_{2ec}) by UF. We consider the UF structures to be one structure, on $O(n)$ elements. Obviously, a nonessential insertion takes time linear to $O(1)$ Finds.

LEMMA 10.3. *In a 3EC structure, the time for a sequence of essential insertions is at most linear to the time for a matching sequence of $O(n)$ operations on $O(n)$ nodes in FRT_{3ec} and FRT_{2ec} and for $O(n)$ Unions, Splits, and Finds in the UF and GSF_{loc} structures.*

Proof. We define a step to 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, including the $insert_3$ calls during the execution of an $insert_3$ itself. Therefore, we do not consider the cost of an essential call $insert_3$ inside $insert_3$. Obviously, there are at most $O(n)$ essential insertions possible. So the essential $insert_3$ operations yield a matching sequence of $O(n)$ operations in FRT_{3ec}. Also, all calls $insert_2$

in the calls $insert_3$ are essential. Therefore, by Observation 4.2, the lemma holds for the operations in $2EC$.

We consider the *net cost* of the sequence of essential $insert_3$ calls: 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 and Splits in the UF and GSF_{loc} structures.

1. Case $c(x) \neq c(y)$. Then there is no net cost.
2. Case $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. We consider the net cost of a call $insert_3$. One part corresponds to the cost of essential local insertions inside clusters. This takes $O(1)$ steps for each such cluster and for each subclass that is joined. These $O(1)$ steps are considered to be included in the cost for joining two subclasses by a local insertion (at most 2 of these clusters have no subclasses that are joined).
   Since, in total, at most $O(n)$ essential local insertions can occur, the *net* cost is linear to $O(n)$ Finds in these structures (by Lemma 10.2).
3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. The computation of a boundary list in 2EC is a part of an essential call $insert_2$. The remainder takes $O(|L|)$ steps, plus a number of (other) $insert_3$ calls (this latter cost is included in case 2). Since the 2ec-classes in L are joined, the total net cost is $O(n)$ steps. □

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 10.4. *A $3EC(i)$ structure solves the $3ec$-problem such that the following holds. The total time that is needed for all essential insertions is $O(n.i.a(i,n))$, whereas a query and nonessential insertion can be performed in $O(i)$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. By Lemma 10.3, Theorems 2.3 and 9.1, and [23] (for GSF(i)), the theorem follows. □

The α-$3EC$ structure is a 3EC structure with $FRT_{3ec} = FRT(\alpha(n,n))$, $FRT_{2ec} = FRT(\alpha(n,n))$, $UF = \alpha$-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, similarly as for Theorem 4.4, we obtain the following.

THEOREM 10.5. *The $3ec$-problem can be solved in $O(m.\alpha(m,n))$ total time (where m is the number of edge insertions and queries), where the fth query can be performed in $O(\alpha(f,n))$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

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 9).

11. A solution for 2-vertex-connectivity.

We 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 using fractionally rooted trees. Similar to 2-edge-connectivity, we thus present a solution that is different from that given in [32] but whose approach is closer to the approach for maintaining the 3-vertex-connectivity relation in general graphs [24].

11.1. Graph observations.

Let $G = \langle V, E \rangle$ be a graph. We define the graph $2vc(G)$ as follows. For each 2vc-class or quasi class, there is a unique 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 in 2vc-class c. (Thus we obtain a collection of trees corresponding to so-called block trees.) Hence $2vc(G)$ is a forest, where each tree in $2vc(G)$ corresponds to a connected component in G. For the insertion of an edge (e, x, y) in G, we have that all the classes of which the class node is on the tree path P between x and y in $2vc(G)$ form one new 2vc-class together (if P exists), while the other 2vc-classes and quasi classes remain unchanged.

We represent $2vc(G)$ by means of 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 a *real* edge class otherwise. Hence, a class of edges together with the end nodes of these edges induces a subtree in $SF(G)$, since for two 2-vertex-connected nodes x and y, all nodes on the tree path between x and y are 2-vertex-connected with them too. Also, two nodes x and y are 2-vertex-connected iff x and y are incident with 2 edges of the same real edge class. We use names of edge classes as the names of the corresponding 2vc-classes and quasi classes. Note that if each edge (e, x, y) in $SF(G)$ is replaced by two edges connecting class node C with x and y, where C is the edge class containing (e, x, y), then we obtain $2vc(G)$.

For the insertion of a new edge $(e, x, y) \notin E$ in G, we now have the following. If $c(x) \neq c(y)$, then (e, x, y) connects two connected components, and it thus connects two trees in $SF(G)$. If $\neg Is2vc(x, y) \wedge c(x) = c(y)$, then all edge classes occurring on the tree path between x and y in $SF(G)$ must be joined. Otherwise, we have $Is2vc(x, y) \wedge c(x) = c(y)$, and the insertion of (e, x, y) will not affect the 2vc-relation.

11.2. Algorithms. We use a fractionally rooted tree structure FRT on forest $SF(G)$, denoted by FRT_{2vc}. All quasi edge classes are marked as being quasi. All other classes are not marked. There is a Union-Find structure for connected components denoted by UF_c. The initialization for an empty graph is straightforward. A query $Is2vc(x, y)$ is performed by first performing a call *candidates*; 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. For the insertion of a new edge (e, x, y) in G, we distinguish the two relevant cases.

1. $c(x) \neq c(y)$. Then $link((e, x, y))$ is performed, and the two connected components $c(x)$ and $c(y)$ are joined (in UF_c).
2. $\neg Is2vc(x, y) \wedge c(x) = c(y)$. Then a boundary list BL for x and y in $SF(G)$ is obtained by $boundary(x, y)$. If BL contains nodes x and y only, then x and y form a quasi class; then the edge class obtained in the call $Is2vc(x, y)$ is unmarked, reflecting that the edge class is real now. Otherwise, nodes x and y are deleted from BL (their sublists contain one edge only), and $joinclasses(BL)$ is called.

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 11.1. *A $2VC(i)$ structure solves the 2vc-problem such that the following holds. The total time that is needed for all essential insertions is $O(n.i.a(i, n))$, where a query and a nonessential insertion can be performed in $O(i)$ time. The structure can be initialized in $O(n)$ time and 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 9.2 is used instead of Theorem 9.1.

THEOREM 11.2. *The 2vc-problem can be solved in $O(m.\alpha(m, n))$ total time (where m is the number of edge insertions and queries), where the fth query can* *be performed in $O(\alpha(f, n))$ time. The structure can be initialized in $O(n)$ time and 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 9).

12. Concluding remarks.

We have presented solutions for maintaining the 2-edge and the 3-edge-connected components of graphs under insertion of edges and vertices and for the 2-vertex-connected components. The solutions take $O(n + m.\alpha(m, n))$ time in total and are optimal on pointer machines and cell probe machines. The optimality follows from the $\Omega(n + m.\alpha(m, n))$ lower bound for k-edge/vertex-connectivity ($k \geq 1$) in general graphs [32]. Also, for all practical problem sizes, there is no need to perform transformations of FRT-structures; we recall that $a(2, n) = \log^* n$ and refer to section 9. Therefore, we conjecture that FRT(2), 2EC(2), 3EC(2), and 2VC(2) can be implemented as fast and easy structures in *practical* situations as well, with constant-time queries.

We have also presented linear-time algorithms for maintaining 2-*edge*-connectivity in a connected graph on a RAM. Since there is a nonlinear lower bound of $\Omega(n + m.\alpha(m, n))$ for maintaining 2-*vertex*-connectivity in connected graphs on a RAM [32], this shows an interesting difference in computational complexity between 2-edge-connectivity and 2-vertex-connectivity.

Finally, we remark that the problem of maintaining the 3-vertex-connected components of general graphs can be solved with the optimal complexity of $O(n + m.\alpha(m, n))$ time for m insertions and queries. This generalizes the special-case result in [4] for maintaining the 3-vertex-connectivity relation inside 2-vertex-connected graphs with such a time bound. We refer to [24].

REFERENCES

[1] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni, *Incremental algorithms for minimal length paths*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 12–21.

[2] R. F. Cohen and R. Tamassia, *Dynamic expression trees and their applications*, in Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1991, pp. 52–61.

[3] G. Di Battista and R. Tamassia, *Incremental planarity testing*, in Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS), 1989, pp. 436–441.

[4] G. Di Battista and R. Tamassia, *On-line algorithms with SPQR-trees*, in Proc. 17th Int. Colloq. on Automata, Languages, and Programming (ICALP), 1990, pp. 598–611.

[5] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, *Sparsification—A technique for speeding up dynamic graph algorithms*, J. ACM, 44 (1997), pp. 669–696.

[6] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer, *Separator based sparsification for dynamic planar graph algorithms*, in Proc. 25th Annual ACM Symposium on Theory of Computing (STOC), 1993, pp. 208–217.

[7] 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*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 1–11.

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

[9] G. N. Frederickson, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees*, SIAM J. Comput., 26 (1997), pp. 484–538.

[10] M. L. Fredman and M. E. Saks, *The cell-probe complexity of dynamic data structures*, in Proc. 21st Annual ACM Symposium on Theory of Computing (STOC), 1989, pp. 345–354

[11] H. N. Gabow, *A scaling algorithm for weighted matching on general graphs*, in Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS), 1985, pp. 90–100.

[12] H. N. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 434–443.

[13] H. N. Gabow and R. E. Tarjan, *A linear time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209–221.

[14] Z. Galil and G. F. Italiano, *Maintaining the* 3*-edge-connected components of a graph on-line*, SIAM J. Comput., 22 (1993), pp. 11–28.

[15] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, Reading, MA, 1969.

[16] G. F. Italiano, *Amortized efficiency of a path retrieval data structure*, Theoret. Comput. Sci., 48 (1986), pp. 273–281.

[17] G. F. Italiano, *Finding paths and deleting edges in directed acyclic graphs*, Inform. Process. Lett., 28 (1988), pp. 5–11.

[18] J. A. La Poutré and J. van Leeuwen, *Maintenance of transitive closures and transitive reductions of graphs*, in Graph-Theoretic Concepts in Computer Science, H. Göttler and H. J. Schneider, eds., Lecture Notes in Comput. Sci. 314, Springer-Verlag, Berlin, 1987, pp. 106–120.

[19] J. A. La Poutré, *New techniques for the union-find problem*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 54–63.

[20] J. A. La Poutré, *Lower bounds for the union-find and the split-find problem on pointer machines*, J. Comput. System Sci., 52 (1996), pp. 87–99.

[21] J. A. La Poutré, J. van Leeuwen, and M. H. Overmars, *Maintenance of* 2*- and* 3*-edge-connected components of graphs* I, Discrete Math., 114 (1993), pp. 329–359.

[22] J. A. La Poutré, *Maintenance of* 2*- and* 3*-Connected Components of Graphs, Part* II: 2*- and* 3*-Edge-Connected Components and* 2*-Vertex-Connected Components*, Tech. report RUU-CS-90-27, Utrecht University, The Netherlands, 1990.

[23] J. A. La Poutré, *Dynamic Graph Algorithms and Data Structures*, Ph.D. thesis, Utrecht University, The Netherlands, 1991.

[24] J.A. La Poutré, *Maintenance of triconnected components of graphs*, in Proc. 19th International Colloq. on Automata, Languages, and Programming (ICALP), 1992, pp. 354–365.

[25] J. A. La Poutré, *Alpha-algorithms for incremental planarity testing*, in Proc. 26th Annual ACM Symposium on Theory of Computing (STOC), 1994, pp. 706–715.

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

[27] F. P. Preparata and R. Tamassia, *Fully dynamic techniques for point location and transitive closure in planar structures*, in Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS), 1988, pp. 558–567.

[28] M. H. Rauch, *Fully dynamic biconnectivity in graphs*, Algorithmica, 13 (1995), pp. 503–538.

[29] H. Rohnert, *A dynamization of the all pairs least cost path problem*, in 2nd Annual Symposium on Theoretical Aspects of Computer Science, K. Mehlhorn, ed., Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, 1985, pp. 279–286.

[30] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[31] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

[32] J. Westbrook and R. E. Tarjan, *Maintaining bridge-connected and biconnected components on-line*, Algorithmica, 7 (1992), pp. 433–464.

[33] J. Westbrook, *Fast incremental planarity testing*, in Proc. 19th International Colloq. on Automata, Languages, and Programming (ICALP), 1992, pp. 342–353.

[34] Ye. Dinitz and J. Westbrook, *Maintaining the classes of y-edge-connectivity in a graph on-line*, Algorithmica, 20 (1998), pp. 242–276.

[35] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen, *On-line maintenance of the four-connected components of a graph*, in Proc. 32nd Annual Symp. on Foundations of Computer Science (FOCS), 1991, pp. 793–801.