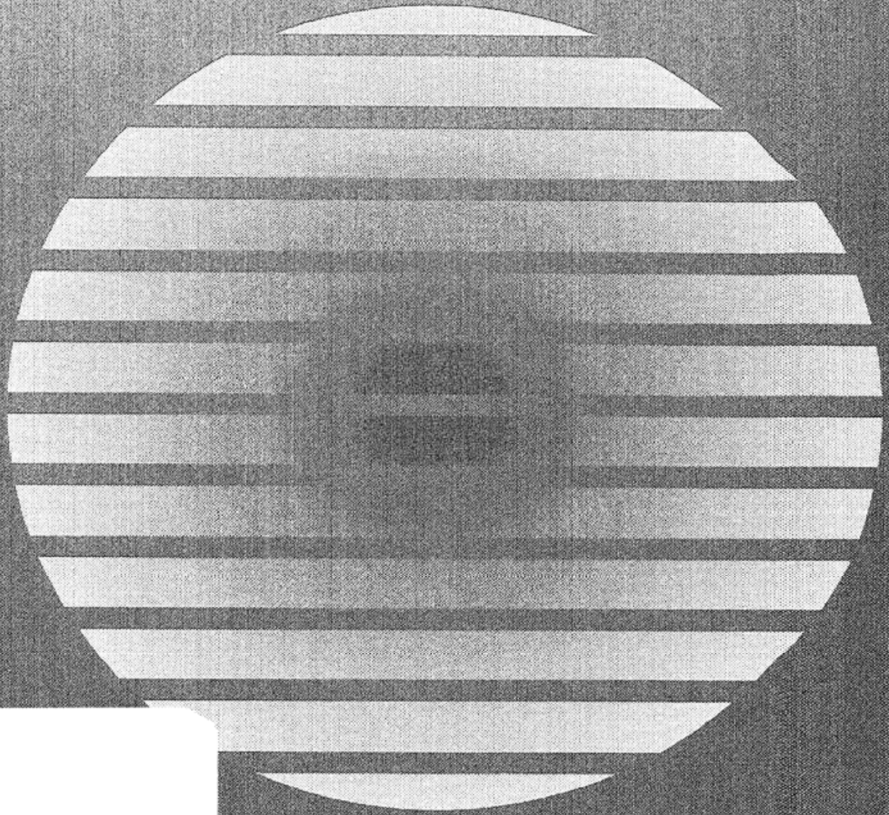
RUU
001

Dynamic Graph Algorithms and Data Structures



J.A. La Poutré

L4

Dynamic Graph Algorithms and Data Structures

Dynamische Graafalgoritmen en
Datastructuren

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE RIJKSUNIVERSITEIT TE UTRECHT OP
GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR.
J. A. VAN GINKEL, INGEVOLGE HET BESLUIT VAN
HET COLLEGE VAN DEKANEN IN HET OPENBAAR
TE VERDEDIGEN OP DONDERDAG 5 SEPTEMBER 1991
DES NAMIDDAGS TE 12.45 UUR

DOOR

JOHANNES ANTONIUS LA POUTRÉ

geboren op 15 juni 1962
te Breda

Promotoren: Prof. Dr. J. van Leeuwen
Prof. Dr. M.H. Overmars

Faculteit Wiskunde en Informatica

This research was partially supported by the ESPRIT Basic Research Actions of the EC under contract no. 3075 (project ALCOM).

Contents

Chapter 1

Introduction

A graph algorithm is called dynamic or on-line if it maintains some information related to a graph while the graph is being changed, e.g., by the insertion or deletion of an edge or a node. A dynamic data structure and its maintenance algorithms exploit a suitable data representation for a graph and use information of the old graph to compute the required information for the updated graph. It is anticipated that a dynamic algorithm does not need to compute a new solution from scratch, i.e., by using the new graph as input only, and a better performance may be expected compared to an algorithm that simply "recomputes". Dynamic algorithms are known for e.g. transitive closures [18, 19, 23, 28], incremental planarity testing [5], minimal spanning trees [8, 9], and maintaining shortest paths [2, 29]. One sometimes uses the term "on-line" or "semi on-line" when only insertions (of nodes or edges) are allowed.

A problem that is important for several dynamic graph problems is the Union-Find problem, which is given as follows. Let U be a universe of n elements. Suppose U is partitioned into a collection of (named) singleton sets and suppose we want to be able to perform the following operations: Union(A,B,C), i.e, join the two sets named A and B and call the result C, and Find(x), i.e., return the name of the set in which element x is contained. A well-known result of Tarjan [31] states that a sequence of up to $n-1$ Union and m Find instructions can be executed in $O(n + m.\alpha(m,n))$ time on a collection of n elements, where $\alpha(m,n)$ denotes the functional inverse of Ackermann's function. In Chapter 3, we develop a new approach to the problem and prove that the time for the k^{th} Find can be limited to $O(\alpha(k,n))$ worst-case, while the total cost for the program of Union's and m Finds remains bounded by $O(n + m.\alpha(m,n))$. These techniques appear to be closely related to the techniques in [11] used for the Split-Find problem. The new algorithm is important in all set-manipulation problems that require frequent Finds. Because $\alpha(m,n)$ is $O(1)$ in all practical cases, the new algorithm guarantees that Finds are essentially $O(1)$ worst case, within the optimal bound for the Union-Find problem as a whole. The algorithm can be implemented on a pointer machine and does not use any form of path compression. The dual problem of the Union-Find problem is the Split-Find problem, which is given as follows. Let U be a universe of n elements (listed in a fixed order). Suppose U is partitioned into a collection of ordered sets whose contents are given in order, and suppose we want to be able to perform the following operations: Find(x) and Split(x), i.e., split the set that contains element x into two new sets, viz, one set containing all elements smaller then x and one set containing all elements that are greater than or equal to x. In [11], Gabow presented a solution for the Split-Find problem that runs with a time complexity as above for the Union-Find problem, and that can be executed on a pointer machine. In Chapter 4, we consider a generalisation of the Split-Find problem: a generalised Split divides an ordered S into two ordered sets S_1 and S_2 such that S_1 is the concatenation of a bounded number of intervals of S, and S_2 is the remainder. (The usual Split is a special case.) The generalisation has applications in problems like maintaining the 3-edge-connected or 3-vertex-connected components of graphs. We present results that have similar complexity bounds as above for the Union-Find problem, and that can be run on a pointer machine.

In 1979, Tarjan [32] proved the well-known lower bound for the time complexity of the Union-Find problem on pointer machines that satisfy the separation condition: for all n and $m \geq n$, there exists a sequence of $n-1$ Union and m Find operations that needs at least $\Omega(m.\alpha(m,n))$ execution steps on a pointer machine satisfying the separation condition. (The separation condition says that, at any moment, the records in the data structure can be partitioned into disjoint sets that have no pointers to each other, where each set of records corresponds to exactly one set of elements.) In [3, 33] the bound was extended to $\Omega(n + m.\alpha(m,n))$ for all m and n. In Chapter 5, we prove that the lower bound of $\Omega(n + m.\alpha(m,n))$ holds on a general pointer machine without the separation condition, thus resolving Tarjan's conjecture. We prove that the same lower bound holds for the Split-Find problem as well.

Consider an undirected graph. Two nodes x and y are called k-edge-connected iff there exist k edge-disjoint paths between x and y, i.e., k paths that do not have an edge in common. Two nodes x and y are called k-vertex-connected iff there exist k different vertex-disjoint paths between x and y, i.e., k paths that do not have a common node except perhaps for their end nodes x and y. In Chapter 6, a data structure is presented to maintain the 2- and 3-edge-connected components of a graph under insertions of edges in the graph. Starting from an "empty" graph of n nodes, i.e., a graph with no edges, the insertion of e edges takes $O(n \log n + e)$ time in total. The data structure allows for insertions of nodes also (in the same time bounds, taking n as the final number of nodes). Moreover, at any moment, the data structure can answer the following type of query in $O(1)$ time: given two nodes in the graph, are these nodes 2- or 3-edge-connected. To obtain better time bounds for this problem, we develop a data structure, called fractionally rooted trees, which are presented in Chapter 7. By means of fractionally rooted trees, we obtain optimal solutions for the problems of maintaining 2-edge-connected and 3-edge-connected components of graphs in Chapter 8. The solutions have a time complexity of $O(n + m.\alpha(m, n))$ for m edge insertions and queries, starting from an "empty" graph with n nodes and no edges. The data structure allows for insertions of nodes also (in the same time bounds, taking n as the final number of nodes). In Chapter 9, we consider the problem of maintaining 2- and 3-vertex-connected components of graphs. Here, the following type of query is considered: given two nodes in the graph, are these nodes 2- or 3-vertex-connected. By means of fractionally rooted trees, we obtain optimal solutions for the problem of maintaining the 2-vertex-connected components of graphs, with the same time complexity as above. Finally, we briefly describe an optimal solution (with the same time complexity) for the problem of maintaining the 3-vertex-connected components of graphs.

Chapter 2

Preliminaries

2.1 Graphs and Terminology

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

We use the following notions (see also [15]). Two nodes are called *adjacent* if there is an edge with these nodes as end nodes. A *path* between two nodes x and y is an alternating sequence of nodes and edges such that x and y are at the end of this sequence and each edge in the sequence is bracketed by its end nodes. However, we often consider a path to consist of the (sub)sequence of the nodes only. A path is *nontrivial* if it contains at least 2 distinct nodes. A path is *simple* if no node occurs twice in it. Two paths are called *edge-disjoint* if they do not have an edge in common. Two (different) paths are called *vertex-disjoint* if they do not have a common vertex except perhaps for their end vertices. Two nodes are called *connected* if there exists a path between them. A (elementary) *cycle* is a path of which the end nodes are equal and in which no edge occurs twice. A cycle containing just one distinct node is called *trivial*, otherwise it is called *nontrivial*. A cycle is *simple* if there is no node that occurs twice in it except for the end nodes.

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

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

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

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

When we consider classes (sets) of nodes in a graph, we often refer to a class of nodes that is represented by a node c as "class c".

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

Notation 2.1.1 *For a set S, $|S|$ denotes the number of elements in the set. For a tree T, $|T|$ denotes the number of nodes in the tree. For a list L, $|L|$ denotes the number of elements in the list. (If to each element in a list L a sublists is attached, then $|L|$ denotes the number of elements in the list without the sublists.)*

2.2 Representation and Data Structures

We informally describe the main aspects of two models of computation: the Pointer Machine and the Random Access Machine. For a detailed description we refer to [31, 20, 21, 30] and [25], respectively. A *pointer machine* is a machine of which the memory consists of (equal) records. A record in memory is only accessible by means of a pointer to that record, which is a specification of that record (e.g., a pointer can be seen as the internal address of the record in memory). Fields of a record may contain either data values or pointer values. However, no arithmetic on pointers is the only operations on pointers are assignment and testing for equality. e, a record can not be obtained by calculation of its address but only by : following a sequence of pointers. A *Random Access Machine (RAM)* is a of which the memory consists of storage locations that are numbered 0,1,2, o.g. we interpret the locations as records also, where the values in the fields gers. A record in memory is accessible by means of its number. However, ger that can be stored in a field is limited to a size $O(\log n)$ (i.e., it consists n) bits), where n is the problem size (i.e., the number of items considered roblem, like the number of nodes and edges in a graph).

hesis we will specify data structures in terms of nodes (or equivalently: and pointers to nodes. As usual, a pointer is a specification of some node unique name or some other identifier) and a node may contain additional ion in some field(s) (e.g. a string, symbol, or pointer). Note that in the odel, a pointer matches with the number related to a storage location.

all the algorithms and data structures that we describe can be implemented a pointer machine and a RAM, with the same complexity. I.e., the memory sed by the implementation consists of records that can only be accessed by f pointers on which no arithmetic is performed, where each record contains a l number of fields (that may contain pointers), and where each field contains bits. This kind of implementation of an algorithm and its associated data es is called a *pointer/*$\log n$ *solution.*

to handle graph maintenance problems, we represent a graph as follows. es and edges of a graph are represented in memory by records, which we sider to be the actual nodes and edges. I.e., we do not distinguish between c (or an edge) and the record that represents it. Also, we will often not ish between a pointer to a record and the record itself. Each vertex has ence list, that consist of pointers to all edges in the representation that are with it. Also, each edge contains pointers to its two end nodes. (Hence, tices that are *adjacent* to some vertex v can be obtained by means of the ce list of v and the pointers from the edges to their end nodes.) An edge that inserted is given by its record, with the pointers to its end nodes.

algorithms we present, we use lists (or equivalently: sets) that can be ma-ed in the following way. We make use of lists of (pointers to) nodes that e following operations: two lists can be joined or a node can be inserted in oth in $O(1)$ time, and a list can be enumerated or removed in linear time. sly, an implementation of a list as a doubly-linked linear list of nodes with al pointers to its two end nodes will do (where the union operation can be ed just by concatenating the linear lists). We do not make these operations in our algorithms, but simply use the mathematical denotations like "$\cup$" for on of two lists and "$\{x\}$" for the list consisting of the element x only.

2.3 The Ackermann Function

The Ackermann function A plays an important role in our algorithms and complexity analyses. It is defined as follows. For all integers $i, x \geq 0$, function A is given by

$$\begin{array}{rcll} A(0,x) & = & 2x & \text{for } x \geq 0 \\ A(i,0) & = & 1 & \text{for } i \geq 1 \\ A(i,x) & = & A(i-1, A(i,x-1)) & \text{for } i \geq 1,\ x \geq 1. \end{array} \tag{2.1}$$

It is easily seen that $A(i,1) = 2$, $A(i,2) = 4$ and $A(i+1,3) = A(i,4)$ for $i \geq 0$. Moreover we have

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

In fact, for every i, $A(i+1,x)$ is the result of x recursive applications of the function $A(i,.)$ (compare e.g. $A(1,x)$, $A(2,x)$ and $A(3,x)$).

Lemma 2.3.1 *Let $A^{(0)}(i,y) := y$ and $A^{(x+1)}(i,y) := A(i, A^{(x)}(i,y))$ for $i, x, y \geq 0$. Then $A(i,x) = A^{(x)}(i-1,1)$ for $i \geq 1,\ x \geq 0$.*

Proof. Straightforward by induction on x. □

Lemma 2.3.2 *$A(i',x') \geq A(i,x)$ for all $i' \geq i,\ x' \geq x$.*

Proof. By induction it follows that for every i, $A(i,x)$ is strictly increasing in x and $A(i,x) \geq 2x$. Next it follows by induction that for every x, $A(i,x)$ is strictly increasing in i. This concludes the proof. (Also cf. [31].) □

Definition 2.3.3

1. *The row inverse a of the Ackermann function is defined by*

$$a(i,n) = min\{j \geq 0 | A(i,j) \geq n\} \tag{2.2}$$

for $i, n \geq 0$.

2. *The functional inverse α of the Ackermann function is defined by*

$$\alpha(m,n) = min\{i \geq 1 | A(i, 4\lceil m/n \rceil) \geq n\} \qquad (2.3)$$

for $m \geq 0$, $n \geq 1$. Here we take $\lceil 0 \rceil = 1$.

(The row inverse $a(i,n)$ for some fixed i is a slowly increasing function: the higher the index i is, the slower the function $a(i,n)$ grows in n. On the other hand, the inverse Ackermann function $\alpha(n,n)$ grows slower than any row inverse.)

Note that $\alpha(0,n) = \alpha(n,n)$. The above two definitions differ slightly from those appearing in [31, 32, 33]. However, it is easily shown that the differences are bounded by some additive constants (except for the functions $a(0,n)$ and $a(1,n)$).

Lemma 2.3.4

$$\begin{array}{lcll} a(i, A(i,x)) & = & x & (i \geq 0,\ x \geq 0) \\ a(i, A(i+1, x+1)) & = & A(i+1, x) & (i \geq 0,\ x \geq 0) \\ a(i,n) & = & a(i, a(i-1,n)) + 1 & (i \geq 1,\ n \geq 2) \end{array}$$

Proof. The first equality follows by definition. By (2.1) we have $a(i, A(i+1, x+1)) = a(i, A(i, A(i+1, x))) = A(i+1, x)$. Moreover, since $n \geq 2$ implies $a(i,n) \geq 1$ and by (2.2), (2.1) and $i \geq 1$ we have

$$\begin{array}{rl} a(i,n) = & \\ = & min\{j \geq 1 | A(i,j) \geq n\} \\ = & min\{j \geq 1 | A(i-1, A(i,j-1)) \geq n\} \\ = & min\{j \geq 1 | A(i,j-1) \geq a(i-1,n)\} \\ = & min\{j' \geq 0 | A(i,j')) \geq a(i-1,n)\} + 1 \\ = & a(i, a(i-1,n)) + 1. \end{array}$$

□

The following lemma shows the relationship between two successive row inverses.

Lemma 2.3.5 *Let $a^{(0)}(i,n) := n$ and $a^{(j+1)}(i,n) := a(i, a^{(j)}(i,n))$ for $i,j \geq 0$, $n \geq 1$. Then $a(i,n) = min\{j | a^{(j)}(i-1,n) = 1\}$ for $i,n \geq 1$.*

Proof. Note that $n \geq 2$ equals $a(i,n) \geq 1$. Hence, by Lemma 2.3.4 it follows by induction that $a(i,n) = a(i, a^{(j)}(i-1,n)) + j$ if $a^{(j)}(i-1,n) \geq 1$. As $a(i,1) = 0$, this yields the required result. □

Thus we have for $n \geq 1$:

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

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

Lemma 2.3.6 *$a(i,n) \leq a(i',n')$ for $i \geq i'$, $n' \geq n$.*

Proof. By Lemma 2.3.2. □

By means of the row inverse of the Ackermann function we can express the functional inverse α as follows.

Lemma 2.3.7 *$\alpha(m,n) = min\{i \geq 1 | a(i,n) \leq 4.\lceil m/n \rceil\}$.*

Corollary 2.3.8 *$\alpha(m',n') \leq \alpha(m,n)$ for $m' \geq m$ and $n' \leq n$.*

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

For simplicity, we extend the Ackermann function as follows:

$$A(i,-1) = 0 \text{ for all } i \geq 0. \tag{2.4}$$

Notation 2.3.9 *The set of all integers greater or equal to -1, is denoted by $\mathbb{N}_{-1}$.*

We state some more lemma's that we will need in the sequel. The proofs can be skipped at first reading.

Lemma 2.3.10 *Let $i \geq 2$, $n \geq 0$. Then $a(i,n) \geq 5 \Rightarrow a(i,n) < \frac{1}{3}.a(i-1,n)$.*

Proof. Suppose $a(i,n) \geq 5$ (and hence $n \geq 2$). Then Lemma 2.3.4 gives

$$a(i, a(i-1,n)) = a(i,n) - 1$$

and therefore by (2.2)

$$a(i-1,n) > A(i, a(i,n) - 2). \tag{2.5}$$

Since $A(2, 5-2) = 16 \geq 3.5$ and since $A(2, x+1-2) = 2^{A(2,x-2)}$, it follows that $A(2, x-2) \geq 3.x$ for $x \geq 5$. Applying this (by means of Lemma 2.3.2) in (2.5) yields

$$a(i-1,n) > 3.a(i,n) \tag{2.6}$$

□

Lemma 2.3.11 *Let $n \geq 1$, $f \geq 0$. Then*

$$\alpha(f,n) < \alpha(0,n) \Rightarrow f > n \wedge 8.f \geq n.a(\alpha(f,n),n)$$

Proof. Let $\alpha(f,n) < \alpha(0,n)$. Then by Lemma 2.3.7

$$4.\lceil\frac{0}{n}\rceil < a(\alpha(f,n),n) \leq 4.\lceil\frac{f}{n}\rceil.$$

Since $\lceil 0\rceil = 1$, this yields $f > n$ and $n.a(\alpha(f,n),n) \leq 8.f$. □

Lemma 2.3.12 *Let $n \geq 1$ and let f_1 and f_2 be such that*

$$\alpha(f_2+1,n) = \alpha(f_2,n) - 1 = \alpha(f_1+1,n) - 1 = \alpha(f_1,n) - 2.$$

(Hence, f_1 and f_2 are two consecutive values of f "after" which the value of $\alpha(f,n)$ decreases.) Then $f_2 \geq 3.f_1 \geq 3.n$.

Proof. Let f_1 and f_2 be as defined above.

First, for all f such that $\alpha(f+1,n) = \alpha(f,n) - 1$ we have by Lemma 2.3.7

$$4.\lceil f/n\rceil < a(\alpha(f+1,n),n) \leq 4.\lceil (f+1)/n\rceil.$$

Hence f/n must be a (positive) integer, which yields

$$\frac{f}{n} \in \mathbb{N} \;\wedge\; 4.\frac{f}{n} < a(\alpha(f+1,n),n) \leq 4.(\frac{f}{n}+1). \tag{2.7}$$

Note that f_1 and f_2 are two such consecutive values of f for which the value of $\alpha(f,n)$ decreases.

Let $\alpha_1 = \alpha(f_1+1,n)$. By $\alpha(f_1+1,n) = \alpha(f_2+1,n)+1$ it follows that $\alpha_1 \geq 2$. Because of $\alpha_1 = \alpha(f_1+1,n) = \alpha(f_1,n) - 1$ and Lemma 2.3.7 we must have $a(\alpha_1,n) \geq 5$. Applying Lemma 2.3.10 yields

$$a(\alpha_1 - 1,n) > 3.a(\alpha_1,n). \tag{2.8}$$

Combining (2.8) and (2.7) gives (by using $\alpha(f_2+1,n) = \alpha_1 - 1$)

$$3.4.\frac{f_1}{n} < 3.a(\alpha_1,n) < a(\alpha_1-1,n) \leq 4.(\frac{f_2}{n}+1).$$

Hence,

$$\frac{f_2}{n} > 3.\frac{f_1}{n} - 1$$

and since by (2.7) $\frac{f_1}{n}$ and $\frac{f_2}{n}$ are integers, this gives $f_2 \geq 3.f_1$. Lemma 2.3.11 provides the second inequality $f_1 + 1 > n$, since $\alpha(f_1+1,n) < \alpha(f_1,n) \leq \alpha(0,n)$. □

The next lemma's use the following:

$$n \geq 3 \wedge i \geq 1 \Rightarrow a(i,n) > A(i+1, a(i+1,n) - 2). \tag{2.9}$$

This follows by using Lemma 2.3.4 that gives $a(i+1, a(i,n)) = a(i+1,n) - 1$ and by using (2.2).

Lemma 2.3.13 *For n and c such that $\alpha(n,n) > \alpha(c,c)+1$ the following holds for i with $1 \leq i \leq \alpha(n,n)-3$:*

$$a(i,n) \geq 8.12^i.i.(c+1)^{i-1}.(\,2a(i+1,n)+c+1).$$

Proof. Let n and c satisfy $\alpha(n,n) \geq \alpha(c,c)+2$. (Hence $\alpha(n,n) \geq 3$.)

Claim 2.3.14 *$c+1 \leq a(i,n)$ for i with $1 \leq i \leq \alpha(n,n)-2$.*

Proof. By (2.3) we have $A(\alpha(n,n)-1,4) < n$ and $A(\alpha(c,c),4) \geq c$. By using $\alpha(n,n)-2 \geq \alpha(c,c)$ it follows that

$$n > A(\alpha(n,n)-1,4) = A(\alpha(n,n)-2, A(\alpha(n,n)-2,4))) \geq A(\alpha(n,n)-2,c).$$

Hence by (2.2) we obtain $c < a(\alpha(n,n)-2,n)$. By Lemma 2.3.2 it follows that $c < a(i,n)$ for i with $1 \leq i \leq \alpha(n,n)-2$. This proves the claim. □

Claim 2.3.15 *$A(i+1,x-2) \geq 2.12^{i+1}.i.x^i$ for $x \geq 6$ and $i \geq 1$.*

Proof. We prove the claim by induction to i and x. Firstly, For $i=1$ and $x=6$ we have $A(2,4) = A(1,A(1,4)) = 2^{16} \geq 2.12^2.1.6$. For $i>1$ and $x=6$ we have by induction

$$A(i+1,4) = A(i,A(i,4)) \geq 2^{A(i,4)} \geq 2^{2.12^i.(i-1).6^{i-1}} \geq 2.12^{i+1}.i.6^i.$$

Finally, for $i>1$ and $x>6$ we have by induction

$$A(i+1,x-2) = A(i,A(i+1,x-3)) \geq 2^{A(i+1,x-3)} \geq 2^{2.12^{i+1}.i.(x-1)^i} \geq 2.12^{i+1}.i.x^i.$$

This proves the claim. □

Let i be such that $1 \leq i \leq \alpha(n,n)-3$. Note that $i+2 \leq \alpha(n,n)-1$ implies $a(i+2,n) \geq 5$ and $n \geq 3$. By applying (2.9) for $i+1$ we obtain

$$a(i+1,n) > A(i+2,a(i+2,n)-2) \geq A(i+2,5-2) \geq A(3,3) \geq 6$$

Hence, Equation (2.9) and Claim 2.3.15 give that

$$a(i,n) > 2.12^{i+1}.i.(a(i+1,n))^i = 8.12^i.i.(a(i+1,n))^{i-1}(2a(i+1,n)+a(i+1,n)).$$

By Claim 2.3.14 the inequality of Lemma 2.3.13 follows. □

Lemma 2.3.16 *Let $n \geq 0$, $1 \leq i \leq \alpha(n,n)-2$. Then $a(i+1,n) < \frac{a(i,n)}{i}$.*

Proof. Since $i+2 \leq \alpha(n,n)$ we have $a(i+1,n) \geq 5$ and $n \geq 3$. Claim 2.3.15 gives that $A(i+1,x-2) \geq i.x$ for $x \geq 6$ and $i \geq 1$. Moreover, $A(i+1,5-2) = A(i,4) \geq i.5$ by Claim 2.3.15 or by $A(1,4)=16$. Applying this in (2.9) yields the required result. □

2.4 Obtaining Ackermann Values

First we consider the Ackermann function as an infinite matrix A with Ackermann values: $A(i,x)$ $(i \geq 1,\ x \geq -1)$. Note that $A(i,-1) = 0$, $A(i,0) = 1$, $A(i,1) = 2$ $A(i,2) = 4$ and that $A(i,3) = A(i-1,4)$ $(i > 1)$.

Let n be an integer, $n > 1$. Suppose we only need Ackermann values $A(i,x)$ to compare them with numbers n' for $0 < n' < n$, i.e. to evaluate a predicate $n' < A(i,x)$, for $i \geq 1$ and $-1 \leq x \leq a(i,n)$. (In our later algorithms we have this situation: for a universe of size n the size m of a set in it "grows" and from time to time $\frac{m}{2}$ is compared with some $A(i,x)$ for $x \leq a(i,n)$.) If all Ackermann values in the matrix that are at least n are replaced by the value ∞, then this does not influence the result of the above comparisons. Note that by (2.2) $A(i,a(i,n)) \geq n$ and $A(i,a(i,n)-1) < n$. By Equation (2.3) it follows that $A(\alpha(n,n),4) \geq n$. By $A(\alpha(n,n)+1,3) = A(\alpha(n,n),4)$ and by Lemma 2.3.2 this gives that $A(i,3) \geq n$ for $i > \alpha(n,n)$. Therefore, if all Ackermann values that are at least n are replaced by ∞ in the matrix, then all rows with row number at least $\alpha(n,n)+1$ become identical.

Therefore, it suffices to have a table that contains the values $A(i,x)$ for

$$1 \leq i \leq \alpha(n,n)+1 \quad \text{and} \quad -1 \leq x \leq a(i,n).$$

where all values $A(i,a(i,n))$ $(1 \leq i \leq \alpha(n,n)+1)$ are replaced by ∞. (In fact, we even can do with less.) In this case, row $\alpha(n,n)+1$ represents all rows i with $i > \alpha(n,n)$.

In correspondence with these observations we define the following notion.

Definition 2.4.1 *Let n_{ack} be an integer, $n_{ack} > 1$. An Ackermann table for n_{ack} with row number $\alpha_{ack} = \alpha(n_{ack},n_{ack})$ is a table A' that contains values $A'(i,x)$ for*

$$1 \leq i \leq \alpha(n_{ack},n_{ack})+1 \quad \textit{and} \quad -1 \leq x \leq a(i,n_{ack}) \tag{2.10}$$

with $A'(i,a(i,n_{ack})) = \infty$ and $A'(i,x) = A(i,x)$ otherwise.

Lemma 2.4.2 *Let A' be an Ackermann table for n_{ack} with row number α_{ack}. Then the following holds for $0 < n' < n_{ack}$, $i \geq 1$, $-1 \leq x \leq a(i,n_{ack})$:*

$$n' < A(i,x) \Leftrightarrow n' < A'(min\{i,\alpha_{ack}+1\},x).$$

In this Section we consider how to compute Ackermann tables for some n_{ack} and how to store them: first by a matrix array and then by a pointer structure. We note that the iterative technique that we employ for computing Ackermann values is well-known, but that the primary reason for this subsection is to define the representations.

2.4.1 Matrix Representation

Suppose we want to represent an Ackermann table for n_{ack} by a matrix. Then this matrix needs columns numbered from -1 up to $a(1, n_{ack})$ and rows numbered from 1 up to $\alpha_{ack} + 1 = \alpha(n_{ack}, n_{ack}) + 1$. Since α_{ack} cannot be computed directly, an upper bound like $\log n$ can be taken. Let A' be the matrix that represents the Ackermann table for n_{ack}. Then A' can be computed by means of the algorithm given in Figure 2.1, where n denotes n_{ack}. We consider the algorithm briefly.

Figure 2.1: The (matrix) computation of the Ackermann table A'.

```
(1)  $i := 0$; $A'(i+1,-1) := 0$; $A'(i+1,0) := 1$; $x := 0$;
(2)  do $A'(i+1,x) < n \longrightarrow A'(i+1,x+1) := 2.A'(i+1,x)$; $x := x+1$ od;
(3)  $A'(i+1,x) := \infty$; $i := i+1$; $ain := x$;
(4)  do $ain > 3 \vee i = 1$
(5)       $\longrightarrow A'(i+1,-1) := 0$; $A'(i+1,0) := 1$; $x := 0$;
(6)          do $A'(i+1,x) \neq \infty$
(7)              $\longrightarrow$if $A'(i+1,x) \geq ain \longrightarrow A'(i+1,x+1) := \infty$
(8)                 $[\!]$ $A'(i+1,x) < ain \longrightarrow A'(i+1,x+1) := A'(i, A'(i+1,x))$
(9)                 fi;
(10)                $x := x+1$
(11)         od;
(12)         $i := i+1$; $ain := x$
(13) od;
(14) $\alpha_{ack} := i-1$
```

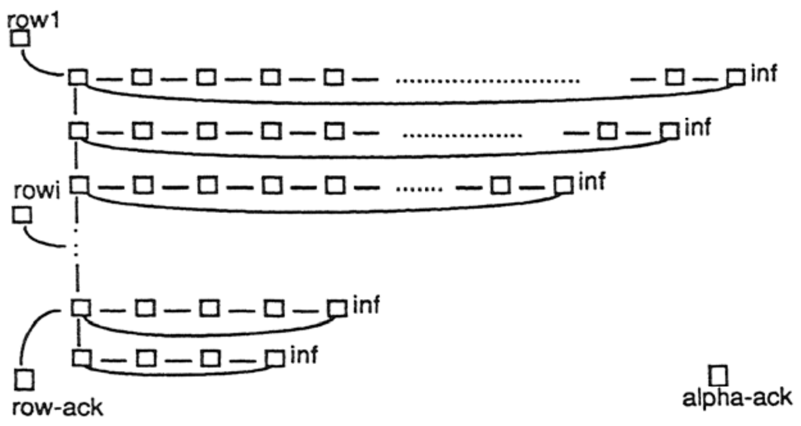
Note that in line 2, $i+1$ is used instead of just 1. This is for later convenience. In line 2 it is used that $A(1,x) = 2^x$ and that $2^{x+1} = 2^x + 2^x$. Also, when row $i+1$ is computed, then ain has the value $a(i,n)$ (i.e., the "last" element of row i and hence $A'(i, ain) = \infty$). The definition of the Ackermann function is used straightforwardly, and in line 7 it is used that if $A(i+1,x) \geq ain$ then $A(i+1,x+1) = A(i, A(i+1,x)) \geq A(i, ain) \geq n$. Therefore it easily follows that the algorithm computes the Ackermann table for n correctly. Note that the entire table can be computed by means of additions and comparisons only, since the expression $2.A'(i+1,x)$ occurring in line 2 can be replaced by $A'(i+1,x) + A'(i+1,x)$.

2.4.2 Node Net Representation

Since we want to run our algorithms on a pointer machine, we represent the Ackermann table for n_{ack} by a "node net", i.e., a data structure of nodes (records) that represent the entries $A'(i,x)$. Apart from n_{ack}, we assume that a positive integer i_0 is given too.

Each node contains a value of the Ackermann table for n_{ack}. The node net consists of (say, "horizontal") linear lists of nodes corresponding to rows of the matrix presented above, while the leading nodes of the lists themselves form a (say, "vertical") linear list corresponding to the column -1. In this description we will denote each node in the net by its coordinates in the corresponding Ackermann matrix. (Hence, node (i,x) represents the coefficient $A'(i,x)$.) Furthermore, the Ackermann net contains the parameter α_{ack}, the pointers row_{ack} and row_1 that point to the leading nodes of rows α_{ack} and 1 respectively (i.e., to nodes $(\alpha_{ack},-1)$ and $(1,-1)$ respectively) and an additional pointer *rowi* that points to the first node of row $(min\{i_0,\alpha_{ack}+1\},-1)$. Finally, in each node $(i,-1)$ the value $a(i,n)$ is stored together with a pointer to node $(i,a(i,n))$. The structure of the node net is illustrated in Figure 2.2. We call such a node net the Ackermann net for n_{ack}.

Figure 2.2: An Ackermann net.



In this representation, the Ackermann table A' can be computed in a similar way as in the case of a matrix representation. We only need to make the obvious adaptations to deal with records and pointers instead of array locations and to deal with the additional pointers that have to be present in the net. We describe the alterations that need to be incorporated in the algorithm given in Figure 2.1. Firstly, together with the variable i used in the algorithm a pointer is maintained that points to the node $(i,-1)$, except initially at line 1 ($i=0$), where pointer *rowi* is initialized to point to node $(1,-1)$. For each value $A'(i+1,-1)$ that is created in the algorithm of Figure 2.1 (line 1 and 5) a node is created and it is appended to the linear "vertical" list that corresponds to column -1. Moreover, this node is taken to be the first node of the "horizontal" list corresponding to row $i+1$. For every other value $A'(i+1,x)$ $(x \geq 0)$ that is computed in the algorithm of Figure 2.1, a node is created containing that value and is appended to the list representing row $i+1$. Together with the variable x used in the algorithm a pointer is maintained that points to the node $(i+1,x)$ (where row $i+1$ is the row that is computed). Then all comparisons w.r.t. values $A'(i+1,x)$ as performed in lines 2, 6, 7 or 8 are performed by means of this pointer. The only remaining question is how to obtain the value $A'(i, A'(i+1,x))$ in line 8. Recall that the function A is monotone (cf. Lemma 2.3.2). Since during the computation of row $i+1$ the computed values $A'(i+1,x)$ are computed for increasing values of x, the values $A'(i, A'(i+1,x))$ are values $A'(i,y)$ that are to be obtained for increasing y too. Therefore during the "traversal" of row $i+1$ in line 6-11 these values can be obtained by "simultaneously" traversing row i with variable y and an appropriate pointer corresponding to y. Note that in this way each row is traversed at most two times in the entire process ("by" x and "by" y). If a row i is "finished" (lines 3 and 12), then the value $a(i,n)$ $(= ain)$ can be stored in node $(i,-1)$ together with a pointer to node $(i, a(i,n))$.

Finally, pointer row_{ack} is initialised. Note that pointer $rowi$ can be initialised to point to node $(min\{i_0, \alpha_{ack}+1\}, -1)$ during the above computations.

In the above way the Ackermann net for n_{ack} is computed. By Lemma 2.3.10 and Lemma 2.3.7 it is easily seen that the Ackermann net contains $O(\log n)$ values and that it is computed in $O(\log n)$ time. Therefore we have proved the following lemma.

Lemma 2.4.3 *The Ackermann net for n_{ack} can be computed in $O(\log n_{ack})$ time.*

Finally, we consider the extension of an Ackermann net for increasing n_{ack}. We only consider values n_{ack} such that $n_{ack} = 2^{x_{ack}}$ for some x_{ack}.

Lemma 2.4.4 *Let the Ackermann net for $n_{ack} = 2^{x_{ack}}$ be given. Then the net can be extended to the Ackermann net for $2.n_{ack} = 2^{x_{ack}+1}$ in $O(\alpha(n_{ack}, n_{ack}))$ time.*

Proof. Suppose we have the Ackermann net for $n_{ack} = 2^{x_{ack}}$. Now extend the Ackermann net as follows. First consider row 1, that is reachable by means of the pointer row_1 pointing to node $(1,-1)$. Note that $x_{ack} = a(1, n_{ack})$. Now replace the value ∞ in node $(1, x_{ack})$ by $2^{x_{ack}}$ $(= n_{ack})$ and append a new node $(1, x_{ack}+1)$ with value ∞ to row 1. Moreover, adapt the relevant values and pointers in node $(1,-1)$.

Then the remaining rows are augmented as follows: if row i $(i \leq \alpha_{ack})$ is not extended, then we are ready, otherwise compare the value $a(i, n_{ack})$ with $A(i+1, a(i+1, n_{ack})-1)$ (both can be obtained in $O(1)$ time using the available pointers, since we can maintain a pointer to node $(i,-1)$): if $a(i, n_{ack}) = A(i+1, a(i+1, n_{ack})-1)$ then store the value n_{ack} $(= A(i, a(i, n_{ack}))$ in node $(i+1, a(i+1, n_{ack}))$ (cf. (2.1)) and append a node with value ∞ to it. Then modify the values and pointers accordingly in node $(i+1,-1)$. Otherwise, nothing needs to be done.

If row $\alpha(n_{ack}, n_{ack}) + 1$ is adapted too, and this row contains the new node $(\alpha(n_{ack}, n_{ack})+1, 4)$, then a new row $\alpha(n_{ack}, n_{ack})+2$ is created containing nodes for $-1 \leq x \leq 3$, where $(\alpha(n_{ack}, n_{ack}) + 2, 3)$ contains the value ∞. Afterwards increase the value α_{ack} by one and adapt the pointer row_{ack} accordingly. After this procedure is finished, set $n_{ack} := 2.n_{ack}$ and $x_{ack} := x_{ack} + 1$. Moreover, the pointer $rowi$ can easily be set to point to node $(min\{i_0, \alpha_{ack} + 1\}, -1)$. It is easily seen that all the above actions can be performed in $O(\alpha(n_{ack}, n_{ack}))$ time. □

Chapter 3

New Techniques for the Union-Find Problem

3.1 Introduction

Let U be a universe of n elements. Suppose U is partitioned into a collection of (named) singleton sets and suppose we want to be able to perform the following operations:

- Union(A,B,C): join the two sets named A and B and call the result C,
- Find(x): return the name of the set in which element x is contained.

The occurring set names must satisfy the condition that, at every moment, the names of the existing sets are distinct. The problem of efficiently implementing Union-Find programs is widely known as the disjoint set union problem or, simply, the "Union-Find problem".

Several data structures and algorithms for the Union-Find problem have been developed. In [16] two set union algorithms were presented (of which the second is the familiar set union algorithm using balancing and path compression), and it was shown that these take $O((n+m).\log^* n)$ time for $n-1$ Unions on n elements and for m Finds, where $\log^*$ denotes the "iterated log-function". In 1975, Tarjan [31] proved that the worst-case time bound for the set union algorithm with balancing and path compression is $O(m.\alpha(m,n))$ for $n-1$ Unions and $m \geq n$ Finds, where α is the inverse Ackermann function (see Section 2.3). The algorithm can be implemented on a pointer machine that satisfies the *separation condition* (i.e., at any moment the records in the data structure can be partitioned into disjoint sets that have no pointers to each other, where each set of records corresponds to exactly one set of elements) (cf. [32, 27] for a precise description). In [32] a lower bound was proved on the time complexity of any Union-Find program that runs on a pointer machine and that satisfies the separation condition: any program of $n - 1$ Unions and m Finds takes at least $\Omega(m.\alpha(m,n))$ time, if $m \geq n$. In [3] and [33] the lower bound was extended to $\Omega(n+m.\alpha(m,n))$ time for all n and m. Until now all known algorithms that run on a pointer machine satisfy the separation condition. Finally, in [22] the first algorithm presented in [16] was combined with path compression yielding a time bound of $O(n.\log^* n + m)$ time for all Unions on n elements and for m Finds.

In this chapter, we reconsider the Union-Find problem, and study the question of bounding the individual complexity of the Finds. We present a collection of Union-Find structures that each take $O(1)$ time per Find in the worst case. I.e., we present a collection of structures UF(i) ($i \geq 1$) that solve the Union-Find problem on a pointer machine, such that for UF(i) a Find operation takes $O(i)$ time and all Union operations together take $O(n.a(i,n))$ time for a universe of n elements ($i \geq 1$, $n \geq 2$), where $a(i,n)$ is the row inverse of the Ackermann function (see Section 2.3). Moreover, by means of these structures, a Union-Find structure is given that has a worst-case time of $O(\alpha(f,n))$ for the f^{th} Find, while the total time complexity for m Finds and $n-1$ Unions is $O(n+m.\alpha(m,n))$. This structure therefore differs from the structures that use path compaction (cf. [31, 33]) in the fact that the worst-case time bound of a Find is small instead of the worst-case time bound of a Union. Because $\alpha(m,n) \leq 3$ in all practical situations, the new algorithm guarantees that Finds are essentially $O(1)$ worst case, within the optimal time bound for the Union-Find problem as a whole. The techniques in the algorithms appear to be closely related to the techniques used in [11] for the Split-Find problem.

The results stated in this chapter have the following applications. The results can be used in cases in which a low worst-case time of a Find is more important than that of the Union(s) because of additional computations (e.g., cf. chapters 8 and 9). Furthermore, because $\alpha(m,n)$ is $O(1)$ in all practical cases, the new algorithm guarantees that Finds are essentially $O(1)$ worst case, within the optimal bound for the Union-Find problem as a whole. On the other hand, the techniques will be used to design an efficient generalized Split-Find structure that runs on a pointer machine (cf. Chapter 4) (a generalized Split divides an interval I into two intervals I_1 and I_2: I_1 is the concatenation of a bounded number of subintervals of I and I_2 is the remainder; the usual Split is a special case), and structures to maintain 2- and 3-edge/vertex-connected components (cf. chapters 6-9). This is because of the possibility of maintaining structural information by means of these techniques, whereas e.g. path compression destroys this kind of information.

As in [27, 31, 32, 33], we consider the Union-Find problem in terms of nodes in a pointer machine. We will not explicitly keep track of the 1-1 correspondence between these nodes and the elements and set names in the actual computing environment (if these are different). However, our procedures are such that the 1-1 correspondences can easily be maintained when necessary.

This chapter is organised as follows. In Section 3.2 we describe the Union-Find problem (on pointer machines) precisely. In Section 3.3 we present a collection of structures UF(i) for all integers $i \geq 1$ by means of an inductive construction starting from the structure UF(1), that in fact is equivalent to a well-known simple Union-Find structure that takes $O(n \log n)$ time for all Unions. Inductively we will describe UF($i+1$) by means of UF(i). The structure UF(i) will turn out to have a time complexity for a Find of $O(i)$ in the worst case and a time complexity for all Unions of $O(n.a(i, n))$: these time bounds are proved in Section 3.4. In Section 3.5 we consider how the transformation of UF(i) structures to other UF(i') structures can be employed to yield a time bound of $O(n + m.\alpha(m, n))$ for all Unions on n elements and for m Finds. In Section 3.6 we consider the problem of insertions of elements.

3.2 The Union-Find Problem and Pointer Machines

We formulate the Union-Find problem in terms of nodes as follows (also cf. [27, 31, 32, 33]). Let U be a collection of nodes, called elements. Suppose U is partitioned into a collection of singleton sets, and suppose to each singleton set a (new) unique node is related, called *set name*. We want to be able to perform the following operations:

- Union(s,t): given two (pointers to) set names s and t, join the corresponding sets into a new set, relate either s or t to it as set name and dispose the other one, and return (a pointer to) the resulting set name.

- Find(x): given (a pointer to) element x, return (a pointer to) the set name of the set in which element x is contained.

The occurring set names must satisfy the condition that, at every moment, the names of the existing sets are distinct.

The above description differs slightly from the description given in [31, 33] where nodes that represent elements represent set names too, but it easily seen that because the fields of nodes can be chosen arbitrary and because (a pointer to) the resulting set name is returned by Union(s, t), other descriptions can easily be simulated.

Now, in the Union-Find problem on a pointer machine each node is *identified* with some unique record in the memory of the pointer machine. Note that this means that nodes have a fixed number of fields and that nodes can only be reached by means of pointer values on which no arithmetic is possible. Therefore we will describe our

Union-Find structures in terms of nodes only and we will not distinguish between a node and the record that represents that node in a memory.

With respect to the 1-1 correspondence between the elements in a pointer machine model and the elements in a actual computing environment we only mention some well-known techniques (that also apply for [27, 31, 32, 33]): if these elements in the environment are records themselves, the 1-1 correspondence can be implemented by means of bidirectional pointers, where a bidirectional pointer between two records denotes that the two records have a pointer to each other. On the other hand, if the elements are represented by arrays, then the 1-1 correspondence can be implemented by means of pointers in the one direction and indices in the other direction. Moreover, note that records can be implemented by means of arrays, where pointers are, in fact, indices in the arrays. Finally, with respect to the correspondences of the set names in a pointer machine and the set names in the environment, similar remarks can be made. Then these 1-1 correspondences can be used and adapted (if necessary) just before and after the executions of Union(s,t) and Find(x) operations and in this way the operations Union and Find as described in Section 3.1 can be achieved.

3.3 The Union-Find Structure UF(i)

In this section we present a collection of structures UF(i) $(i \geq 1)$ that allow Union and Find operations as described in Section 3.2. Let $i \geq 1$. A UF(i) structure is a collection of rooted trees. The collection of trees is changed by Union operations. For each set name s let the set of elements corresponding to name s be denoted by $set(s,i)$. Each set name is the root of a tree. The leaves of the tree with root s are the elements in $set(s,i)$ and have an equal distance $\leq i$ to the root. The nodes of the tree without the root can be split into layers of nodes that have equal distance to the root. The layer that contains the elements of $set(s,i)$ is called layer i, the other occurring layers are numbered consecutively in a decreasing order starting from layer i.

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

Trees are represented as follows: for each node x the field $father(x)$ contains a pointer to its father if x is not a root and it contains the value nil otherwise. Moreover, $sons(x)$ is the list of the sons of x.

Structure UF(i) allows the operations UNION(s,t,i) and FIND(x) that satisfy the specification given in Section 3.2. Function UNION(s,t,i) will be given in the sequel. Function FIND(x) is given in Figure 3.1. Obviously, FIND(x) outputs the root of the tree in which node x is contained. From the above description of UF(i) it follows that for any element x, FIND(x) outputs the name of the set in which x is contained.

Figure 3.1: Procedure $FIND(x)$ in UF(i) ($i \geq 1$).

```
(1) procedure $FIND(x)$; return $< set\ name >$;
(2) if $father(x) = nil \longrightarrow FIND := x$
(3) [] $father(x) \neq nil \longrightarrow FIND := FIND(father(x))$
(4) fi
```

The structures UF(i) are defined inductively for $i > 1$, starting from a base structure UF(1) (that, in fact, is equivalent to a well-known simple Union-Find structure, which can be found in [1]). First we outline the structure of UF(1) in Subsection 3.3.1, and then we describe UF(i) for $i > 1$ in Subsection 3.3.2,

In the sequel we denote by "UF(i) elements" elements that are involved in the Union-Find problem to be solved by the UF(i) structure. Moreover, sometimes we will refer by "UF(i) structure" to the algorithms too.

3.3.1 The Union-Find Structure UF(1)

Structure UF(1) is the structure that underlies the straightforward set-merging algorithm. Recall that the set corresponding to set name s is denoted by $set(s, 1)$ and that the nodes in $set(s, 1)$ are in layer 1. According to the above constraints, for every element x $father(x)$ contains a pointer to the name of the set in which it is contained and for every set name s, $sons(s) = set(s, 1)$.

For each set name s we have a parameter $weight(s, 1)$ that contains the size of $set(s, 1)$: $weight(s, 1) = | set(s, 1) |$.

If UF(1) is used to solve the Union-Find problem, then the initialisation for some (sub-)collection of elements into sets is straightforward (for any initial collection of sets, but usually singleton sets).

The Union of two sets can now be performed by the algorithm UNION($s, t, 1$) given in Figure 3.2. The algorithm is based on changing the father pointers of the elements of the smallest of the two sets that are involved in the Union. The generation of all $e \in sons(t)$ that occurs in line 3 of the procedure can be performed by enumerating the list sons(t). Moreover, the joining of the two lists occurring in line 5 can be performed in $O(1)$ time (cf. Section 2.2).

Figure 3.2: The Union procedure in UF(1).

```
(1) procedure UNION($s$,$t$,1); return < set name >;
(2) {pre: weight($s$, 1) $\geq$ weight($t$, 1); otherwise interchange $s$ and $t$}
(3) for all $e \in$ sons($t$) $\longrightarrow$ father($e$) := $s$ rof;
(4) weight($s$, 1) := weight($s$, 1) + weight($t$, 1);
(5) sons($s$) := sons($s$) $\cup$ sons($t$);
(6) remove node $t$; return node $s$
```

3.3.2 The Union-Find Structure UF(i) for $i > 1$

Let $i > 1$. Structure UF(i) is a structure that satisfies the following conditions. Recall that the set corresponding to set name s is denoted by $set(s, i)$ and that nodes in $set(s, i)$ are in layer i.

For each set node s we have a parameter $weight(s, i)$ that contains the size of $set(s, i)$: $weight(s, i) = \mid set(s, i) \mid$. Moreover, we have a parameter $lowindex(s, i) \in \mathbb{N}_{-1}$ that satisfies

$$2.A(i, lowindex(s, i)) \leq weight(s, i). \tag{3.1}$$

Note that $lowindex(s, i)$ does not have to be the largest number that satisfies this inequality, and that the above restriction on $lowindex$ is equivalent to

$$lowindex(s, i) < a(i, \lceil \frac{weight(s, i) + 1}{2} \rceil). \tag{3.2}$$

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

Two cases are distinguished.

- If $set(s, i)$ contains more than one element (i.e., $weight(s, i) > 1$), then $set(s, i)$ is partitioned into clusters (subsets) of at least 2 elements. For each such cluster C there is a unique so-called cluster node c (which is not an element in $set(s, i)$); all nodes in cluster C have node c as their father and $sons(c) = C$. In this description we denote the set of these cluster nodes by $clusset(s, i)$.

  A cluster node $c \in clusset(s, i)$ satisfies (besides $\mid sons(c) \mid \geq 2$)

  $$\mid sons(c) \mid \geq 2.A(i, lowindex(s, i)). \tag{3.3}$$

  The subtree between s and $clusset(s, i)$ is a tree of a UF($i - 1$)-structure: the nodes of $clusset(s, i)$ are the elements of the set named s in a UF($i - 1$)

structure. Thus:

$$set(s, i-1) = clusset(s, i).$$

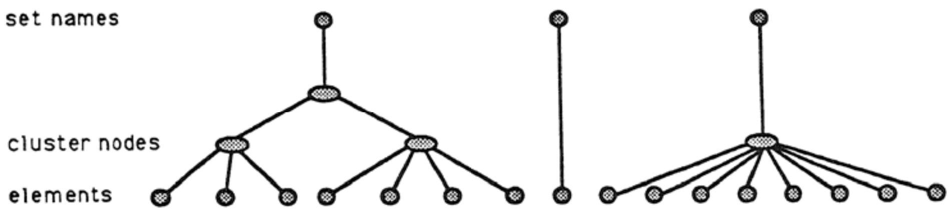
Finally, $clus(s, i)$ contains a pointer to an arbitrary cluster node in $clusset(s, i)$.

- If $set(s, i)$ consists of precisely one element e (i.e., $weight(s, i) = 1$) then $father(e) = s$, $sons(s) = \{e\}$ and $clus(s, i) = nil$.

(Note that in each of the above cases, the elements in a tree have the same distance $\leq i$ to the root, which easily follows by induction.)

By means of this recursive definition, the UF(i) structure consists of a collection of trees, one for each set. In each tree different layers can be distinguished starting from the elements at layer i via clusters nodes that are "elements" on layer $i-1$ etcetera, to some layer that consists of only one element or that is layer 1 (what depends on the considered set, cf. Figure 3.3). Alternatively stated, UF(i) consists of trees that have one leaf as the son of the root and of trees that are trees in some UF($i-1$) structures if the leaves are removed.

Figure 3.3: Set representations by trees in UF(i) ($i > 1$).



If UF(i) is used to solve the Union-Find problem, then the initialisation for some (sub-)collection of elements in singleton sets is as follows: for each element e with set name s for the singleton set $\{e\}$, the following initialisation is performed: $father(e) = s$, $sons(s) = \{e\}$, $weight(s, i) = 1$, $lowindex(s, i) = -1$ and $clus(s, i) = nil$. In this way the set names and the elements satisfy the conditions of UF(i) initially. (Note that afterwards, the insertion of an element in the collection of elements can easily be performed in this way too.) If we want to initialise the structure for some collection of elements into a collection of given, directly available sets (not necesarily singleton sets) then this can be performed as follows: for each set with set name s that contains more then one element, create a cluster node c, let the father pointers of all its elements point to c and put them in the list $sons(c)$. Then make $sons(s) = \{c\}$, $father(c) = s$, $clus(s, i) = c$, $weight(s, i) =$

[the number of elements in the set] and $clus(s, i-1) = nil$, $weight(s, i-1) = 1$. Finally, $lowindex(s, i) = lowindex(s, i-1) = -1$.

The Union of sets can now be performed by the algorithm UNION(s, t, i) given in Figure 3.4. We do not consider the problem of how to obtain and store the values *weight*, *lowindex*, and *clus* yet (note that all these values depend on both the set name and i, the layer number). The set $set(s, i)$ can be obtained by the function $generate(s, i)$ given in Figure 3.5. This function generates set $set(s, i)$ and removes all intermediate tree nodes between s and $set(s, i)$.

Procedure UNION(s, t, i) operates in the following way. W.l.o.g. we assume that $lowindex(s, i) \geq lowindex(t, i)$. The procedure is based on changing the father pointers of $set(t, i)$ towards cluster node $clus(s, i)$ if $set(t, i)$ has a lower value *lowindex* than $set(s, i)$ (lines 5-10), otherwise, if the union of both sets has a size that allows a larger *lowindex* than the actual (equal) values of the old sets then the father pointers of both sets are changed towards an entire new cluster node (lines 14-21), and otherwise a recursive call is performed (line 12-13).

We describe the procedure in more detail. First, both the weights of s and t are adapted. (For, at this moment, both s and t can be the name of the set resulting from the Union.) If the levels of s and t are distinct (line 5), then all elements of t are put beneath a cluster node c of s, and the necessary updates are performed (line 6-10, also cf. Figure 3.6). I.e., the elements of t in this layer are generated, while all "intermediate" nodes are removed (line 7), the father pointers of these elements are adapted, the list $sons(c)$ is augmented with these elements and finally t itself is removed. Otherwise, if the levels are equal (line 11), then two cases are possible. If the size of the union becomes "sufficiently large" (line 14), then all elements of the union are put beneath a new cluster node c while the necessary updates are performed (line 15-27, also cf. Figure 3.7), i.e., the elements of both s and t in this layer are generated in the list $sons(c)$ while all "intermediate" nodes are removed, their father pointers are adapted to the new cluster node c, node c is "put below" node s and the parameters for layer i and layer $i-1$ are updated. Finally, set node t is removed. Otherwise, if the size of the union does not become "sufficiently large" (line 12), there is a recursive call to join the cluster nodes for layer i of both sets (line 13). Then the above cases appear on a lower layer (cf. Fig. 3.8).

Note that at the moment of the recursive call (line 13), all parameters at layer i satisfy the UF(i) conditions for the ultimate joined set, whichever of the two set names s or t will be its name. It is readily verified that the procedure maintains the conditions of UF(i).

We want to state here that the value $lowindex(s, i)$ can also be defined as the largest value that satisfies (3.1). In this case the corresponding UNION procedure is obtained from the procedure give in Figure 3.4 by changing the guards of the if-statements: line 15-21 only gets the guard $newweight \geq 2A(i, ls+1)$, lines 6-10 and 13 get the guard $newweight < 2A(i, ls+1)$ while the distinction between lines 6-10

Figure 3.4: The Union procedure in UF(i) ($i > 1$).

```
(1)  procedure UNION($s$,$t$,$i$); return < $set\ name$ >;
(2)  {pre: lowindex($s,i$) $\geq$ lowindex($t,i$); otherwise interchange $s$ and $t$}
(3)  $ls$ := lowindex($s,i$); $lt$ := lowindex($t,i$);
(4)  $newweight$ := weight($s,i$) := weight($t,i$) := weight($s,i$) + weight($t,i$);
(5)  if $ls > lt$
(6)     $\longrightarrow$ {$ls \geq 0$, hence $clus(s,i) \neq nil$}
(7)        $c$ := clus($s,i$); $setT$ := generate($t,i$);
(8)        for all $e \in setT \longrightarrow$ father($e$) := $c$ rof;
(9)        sons($c$) := sons($c$) $\cup$ $setT$;
(10)       remove node $t$; return node $s$
(11) [] $ls = lt$
(12)    $\longrightarrow$ if $newweight < 2.A(i, ls+1)$
(13)          $\longrightarrow$ UNION($s,t,i-1$)
(14)       [] $newweight \geq 2.A(i, ls+1)$
(15)          $\longrightarrow$ create a new cluster node $c$;
(16)             sons($c$) := $generate(s,i) \cup generate(t,i)$;
(17)             for all $e \in$ sons($c$) $\longrightarrow$ father($e$) := $c$ rof;
(18)             sons($s$) := {$c$}; father($c$) := $s$;
(19)             lowindex($s,i$) := lowindex($s,i$) + 1; clus($s,i$) := $c$;
(20)             lowindex($s,i-1$) := $-1$; weight($s,i-1$) := 1; clus($s,i-1$) := $nil$;
(21)             remove node $t$; return node $s$
(22) fi      fi
```

Figure 3.5: Procedure generate(s,i) in UF(i) ($i \geq 1$).

```
(1)  function $generate(s,i)$; return < $set\ of\ nodes$ >;
(2)  {generate($s,i$) generates the nodes in set($s,i$) and removes all intermediate }
(3)  {tree nodes between $s$ and set($s,i$) }
(4)  if $i = 1 \vee$ weight($s,i$) = 1 $\longrightarrow$ $generate$ := sons($s$);
(5)  [] $i > 1 \wedge$ weight($s,i$) > 1 $\longrightarrow$ $clusset$ := generate($s,i-1$);
(6)                                   $generate$ := sons($clusset$);
(7)                                   dispose all nodes of $clusset$
(8)  fi
```

Figure 3.6:

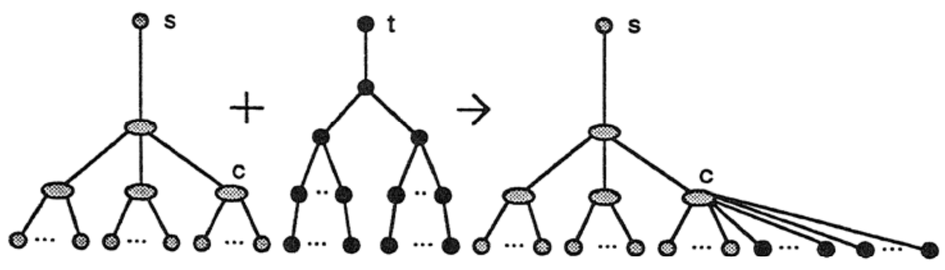


Figure 3.7:

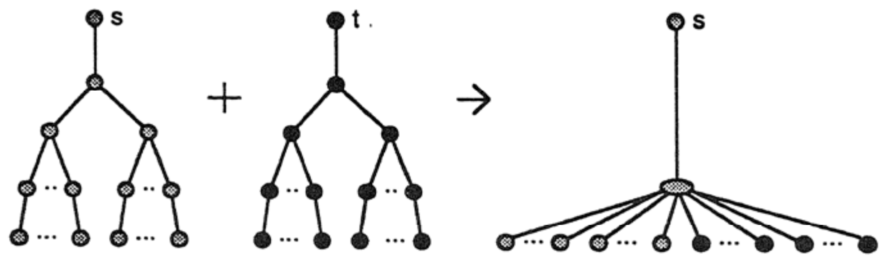
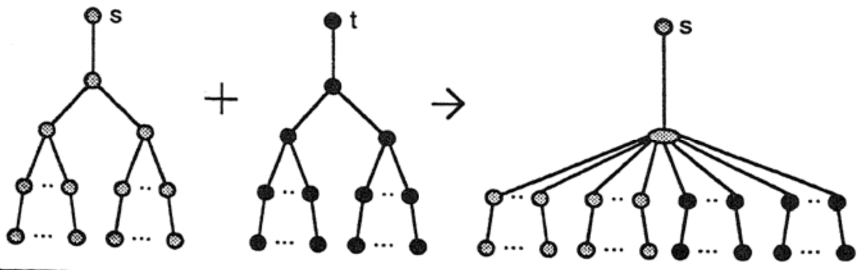


Figure 3.8:



and 13 is made by the guards $ls > lt$ and $ls = lt$ respectively. In that case the values *lowindex* need to be computed in initialisations of sets that contain more than one element (which can be performed without increasing the complexity).

3.3.3 Representations

First we consider the Ackermann values that are needed. Consider UF(i) for some $i \geq 1$. Suppose there are n elements in UF(i). Then for every occurring set name s we have $set(s, i) \leq n$. Since the leaves of the trees of UF(i) are UF(i) elements, it follows that every occurring set of UF(i') elements inside the UF(i) structure (for i' with $1 \leq i' \leq i$) has at most n elements too. Consider procedure UNION(s, t, i') for some i' with $2 \leq i' \leq i$. Since the parameter *newweight* in this procedure is the size of a set of UF(i') elements that is the result of a Union, it satisfies $newweight \leq n$. By Figure 3.4 (cf. lines 3, 4, 12 and 14) and by the definition of *lowindex* (cf. Section 3.3.2) it is easily seen that comparisons $newweight < 2.A(i', l+1)$ only occur if $2.A(i', l) < newweight$ and hence only if $2.A(i', l) < n$ which yields $l < a(i', n)$. Therefore comparisons $\frac{newweigth}{2} < A(i', x)$ are performed only for x and $\frac{newweight}{2}$ with $-1 \leq x \leq a(i', n)$ and $0 < \frac{newweight}{2} < n$. By Lemma 2.4.2 an Ackermann table for any $n_{ack} \geq n$ can be used for computing the comparisons.

We describe how to represent and how to obtain the information that is introduced in the previous subsection. Again we describe the representation inductively. Consider a UF(i) structure for some $i \geq 1$. Suppose there are n elements. For each set name s there is a distinct record $status_{s,i}$ for this layer i. Moreover, an Ackermann net for some n_{ack} with $n_{ack} \geq n$ is present (e.g., $n_{ack} = n$).

Record $status_{s,i}$ contains the fields *layer*, *weight*, *lowindex*, *clus*, *Ack*, *leftAck* and *up*. For $i = 1$, $layer(status_{s,1}) = 1$ and $weight(status_{s,1}) = weight(s, 1)$ and the other fields are irrelevant. For $i > 1$ the following holds.

- Field $layer(status_{s,i}) = i$, $weight(status_{s,i}) = weight(s, i)$, $lowindex(status_{s,i}) = lowindex(s, i)$ and $clus(status_{s,i}) = clus(s, i)$.
- Field $Ack(status_{s,i})$ contains a pointer into the Ackermann net that points to the node $(min\{i, \alpha_{ack} + 1\}, l)$ where $l = lowindex(s, i)$.
- Furthermore field $leftAck(status_{s,i})$ contains a pointer into the Ackermann net that points to the node $(min\{i, \alpha_{ack} + 1\}, -1))$.
- Finally, field $up(status_{s,i})$ contains a pointer to the record $status_{s,i-1}$ for s and layer $i - 1$, provided that $weight(s, i) > 1$ (i.e., $clusset(s, i) \neq \emptyset$).

Then, the Union procedures are adapted slightly in the following way. First of all, instead of UNION(s, t, i) we have UNION($status_{s,i}, status_{t,i}, i$). By means of the pointer $up(status_{s,i})$ the recursive call in line 13 of Figure 3.4 can be performed.

The statement "remove (set) node t" occurring in lines 10 and 21 has to be extended with the removal of the related chain of status records. A new status record $status_{s,i-1}$ has to be created in line 20 if it did not exist already and otherwise all status records $status_{s,j}$ for $j < i-1$ must be removed. (This can be done in $O(1)$ time per removal of a status node.)

Furthermore, when the value of $lowindex(s,i)$ is increased by one (in line 19), the corresponding pointer $Ack(status_{s,i})$ is adapted accordingly (obviously, this can be performed in $O(1)$ time too). When $lowindex(s,i-1)$ is put to -1 (in line 20), the corresponding node $(min\{i-1, \alpha_{ack}+1\}, -1)$ can be obtained by means of the pointer $leftAck(status_{s,i})$ that points to node $(min\{i, \alpha_{ack}+1\}, -1)$ and hence the pointers $Ack(status_{s,i-1})$ and $leftAck(status_{s,i-1})$ can be assigned in $O(1)$ time. (Note that we need the values α_{ack} and i to distinguish whether the above two nodes are equal or not.)

Now, the Ackermann values (lines 12 and 14) can be obtained by means of the pointers $Ack(status_{s,i})$ into the Ackermann net and the successor pointers of Ackermann nodes (with the convention that Ackermann values that are at least n are replaced by the value ∞).

As stated above, the call of a Union procedure in UF(i) is now performed by taking the appropriate *status* node at layer i. If the UF(i) structure is applied within some computing environment (i.e., it is not part of a UF($i+1$) structure), then each set name s contains a pointer to its status record $status_{s,i}$. Moreover, some Ackermann net for $n_{ack} \geq n$ is taken (e.g., $n_{ack} = n$), where pointer $rowi$ points to node $(min\{i, \alpha_{ack}+1\}, -1)$. Note that by means of pointer $rowi$ the initialisation of a UF(i) structure (for $i > 1$) as described in Subsection 3.3.2 can easily be augmented to initialize the pointers described above without increasing the total time in order of magnitude.

All the operations on status records as stated above (except for the removal of a chain of status records) can be done in $O(1)$ time each. Moreover, the removal of a status record can be charged to its creation. Therefore, all additional actions w.r.t. the status records that are performed in the way described above, do not increase the time order of the algorithms. Moreover, since for a set name s there only exists a status record for layer i' if $set(s,i') \neq \emptyset$, and since layers do not intersect, the status records do not increase the order of space used by the algorithms. We will therefore not consider the status records in the complexity analysis in Section 3.4.

3.4 Complexity of UF(i)

The execution of a Find in a UF(i) structure ($i \geq 1$) takes at most $O(i)$ time, since the elements in UF(i) have distance at most i to the corresponding roots.

As shown in [1], all Unions on n elements ($n > 1$) in structure UF(1) take at most $c_0.n.\log n$ time for some constant c_0, hence at most $c_0.n.a(1,n)$ time. We briefly recall the proof. Consider procedure UNION$(s,t,1)$. The execution of procedure UNION$(s,t,1)$ takes at most $c_0.|weight(t,1)|$ time (for some appropriate constant c_0), where $set(t,i)$ is the smallest of the two sets to be joined. Now charge the cost of such a Union to the nodes in $set(t,1)$ by charging to each node at most c_0 time. A node can only be charged to if it becomes an element of a new set whose size is at least twice the size of the set it belonged to. Hence a node can be charged to at most $\lfloor \log n \rfloor$ times. Therefore, all Unions take at most $c_0.n.\lfloor \log n \rfloor \leq c_0.n.a(1,n)$ time together.

We now consider the complexity for all Unions in UF(i) with $i > 1$. We perform the analysis by means of induction on i.

Suppose UF$(i-1)$ takes at most $c.k.a(i-1,k)$ time for all Unions on k elements ($k > 1$), where c is some arbitrary constant. We consider the cost of all Unions on n elements ($n > 1$) by means of UF(i). Therefore, consider procedure UNION(s,t,i). We divide this procedure into several parts.

1. The for-statements and the generate-statements (lines 7-8 and 16-17).
2. The recursive call UNION$(s,t,i-1)$ (line 13).
3. The removal of parts of the structures.
4. The rest of the procedure.

We compute the cost of each of the above parts for *all* executions of procedure UNION(s,t,i) together.

3.4.1 The For-Statements and the Generate-Statements

We consider the for-statements and the generate-statements (viz., lines 7-8 and 16-17). Firstly, it is easily seen by induction on i that the generation of $set(s,i)$ by means of procedure call $generate(s,i)$ takes time that is bounded by $c_1'.\mid set(s,i)\mid$ for some constant c_1', since the number of cluster nodes for layer i is at most half the number of elements at layer i (cf. Subsection 3.3.2). Moreover, the execution of the for-statements (lines 8 and 17) takes time bounded by c_1''.*(the number of processed elements)*. Therefore, we charge the cost of the above statements to the processed elements. Note that in both cases the processed elements will be contained in a new set that has a higher *lowindex* value than the old set (cf. line 5 and 19), and that an element will never be contained in a set with a lower *lowindex* value. Therefore the number of times that an element can be charged to is bounded by the number of different *lowindex* values. Since there are at most n (> 1) elements in a set, there are by the definition of *lowindex* (cf. (3.1) and (3.2)) at most $a(i, \lceil\frac{n+1}{2}\rceil) + 2 \leq 3.a(i,n)$ different values. Therefore, the total cost of the considered parts of the procedure is at most $c_1.n.a(i,n)$ for some constant c_1.

3.4.2 The Recursive Call UNION$(s,t,i-1)$

The recursive calls UNION$(s,t,i-1)$ are performed on cluster nodes. Therefore we first consider cluster nodes and the conditions for a recursive call UNION$(s,t,i-1)$.

Observation 3.4.1 *The operations on cluster nodes by procedure UNION(s,t,i) are:*

1. *the creation of a cluster node in a singleton set (viz. c and $clusset(s,i)$ in lines 15 and 18)*
2. *the Union of sets of cluster nodes by UNION$(s,t,i-1)$ (viz. $clusset(s,i)$ and $clusset(t,i)$ in line 13)*
3. *the removal of a complete set of cluster nodes (viz. $clusset(t,i)$ in line 7 or $clusset(s,i)$ and $clusset(t,i)$ in line 16).*

Claim 3.4.2 *A recursive call UNION$(s,t,i-1)$ inside UNION(s,t,i) is performed only if*

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

Proof. It follows directly from Figure 3.4 that the recursive call is performed only if

$$\begin{aligned} &lowindex(s,i) = lowindex(t,i) \wedge \\ &weight(s,i) + weight(t,i) < 2.A(i, lowindex(s,i)+1). \end{aligned} \tag{3.4}$$

Since $-1 \leq l \leq 1$ implies that $2.max\{2.A(i,l),1\} \geq 2.A(i,l+1)$, it follows by (3.4) and (3.1) that $lowindex(s,i) > 1$. By $n \geq weight(s,i) \geq 2.A(i, lowindex(s,i))$ it follows that $lowindex(s,i) \leq a(i,n)$. □

For a cluster node $c \in clusset(s,i)$ we denote by $lowindex(c)$ the value $lowindex(s,i)$. It is easily seen that a Union does not change the value $lowindex(c)$ for any cluster node c that is not removed by it (for in that case the new set name that corresponds to c has the same *lowindex* value as the old one). Therefore for any cluster node c the value $lowindex(c)$ is fixed (i.e., c is a cluster node for sets with some fixed *lowindex* only). We call a cluster node c with $lowindex(c) = l$ an l-cluster node.

Similarly, we say that a recursive call UNION$(s, t, i-1)$ is an l-call or an l-Union if $l = lowindex(s,i) = lowindex(t,i)$. Obviously an l-call operates on l-cluster nodes only and l-cluster nodes are only operated on by l-calls. We compute the cost of all l-calls for fixed value l.

Let l be a fixed number satisfying $1 < l \leq a(i,n)$. We consider the cost of all recursive l-calls UNION$(s,t,i-1)$. By Claim 3.4.2 and by Subsection 3.3.2 it follows that in case of an l-call UNION$(s,t,i-1)$ the size of the set $clusset(s,i-1) \cup clusset(t,i-1)$ is at most $A(i,l+1)$. Therefore the maximal size of any occurring set of l-cluster nodes is $A(i,l+1)$. Now partition the total collection of all l-cluster nodes involved in l-calls into collections that correspond to the maximal sets that ever exist (which is possible because of Observation 3.4.1). Then the size of such a maximal collection is at most $A(i,l+1)$. For each such maximal collection of k cluster nodes, the cost of all Unions on these nodes in UF$(i-1)$ 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 Unions in UF$(i-1)$ on l-cluster nodes is at most $c.(number\ of\ l\text{-}cluster\ nodes).\ a(i-1, A(i,l+1))$. Since each l-cluster node has at least $2.A(i,l)$ elements as its sons (cf. (3.3)), and since as long as an element is contained in sets with $lowindex$ value l it has the same cluster node as its father (cf. Subsection 3.4.1), there are at most $n/(2.A(i,l))$ l-cluster nodes. Therefore, the total cost for all l-Unions is at most

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

by using $i > 1$, equation (2.1) and Lemma 2.3.4 respectively.

Since there are less then $a(i,n)$ applicable values l of $lowindex$ to be considered (viz. l with $1 < l \leq a(i,n)$), this yields that the total time complexity of all UF$(i-1)$-Unions is at most $\frac{1}{2}c.n.a(i.n)$.

3.4.3 The Removal of Parts of Structures

The removal of parts of structures can be performed in $O(1)$ time per item that must be removed. Therefore, we charge the cost of the removal of an item to its creation. This increases the cost of some operations by constant time only.

3.4.4 The Rest of the Procedure

The execution of all statements together except those considered in subsections 3.4.1, 3.4.2 and 3.4.3, requires at most c_4 time per call of UNION(s,t,i). Since there are at most $n - 1$ Unions, this takes altogether at most $c_4.n$ time.

3.4.5 The Total Complexity of Unions

Combining the results of subsections 3.4.1 to 3.4.4 yields that the total time is at most

$$c_1.n.a(i,n) + \frac{1}{2}c.n.a(i,n) + c_4.n.$$

Note that this is at most $c.n.a(i,n)$ if $c \geq max\{c_0, 2.(c_1 + c_4)\}$.

Since the constant c was arbitrary and since c_1 and c_4 do not depend on c, we can take $c = max\{c_0, 2.(c_1 + c_4)\}$. Then it follows by induction that UF(i) takes at most $c.n.a(i,n)$ time for all Unions together.

By means of induction we have established the following result .

Lemma 3.4.3 *The total time that is needed for all Union operations in UF(i) for a universe with n elements is $O(n.a(i,n))$, whereas each Find operation takes $O(i)$ time ($i \geq 1$, $n \geq 2$).*

By Lemma 2.4.3 the Ackermann net for n can be computed in $O(\log n)$ time and takes $O(\log n)$ space. Moreover, it is readily verified that the initialisation of UF(i) as described in Subsection 3.3.1 (for $i = 1$) and Subsection 3.3.2 (for $i > 1$) can be performed in $O(n)$ time. Finally, by induction to i it easily follows that the total space complexity of UF(i) is $O(n)$, since the elements at layer $i > 1$ are the leaves of the trees UF(i) consists of and since all nodes in a tree except the root have at least two sons (cf. Subsection 3.3.2). Therefore, we have established the following theorem.

Theorem 3.4.4 *A UF(i) structure and the algorithms that solve the Union-Find problem can be implemented as a pointer/ $\log n$ solution such that the following holds. The total time that is needed for all Union operations in a UF(i) structure for a universe with n elements is $O(n.a(i,n))$ and the time needed for a Find operation is $O(i)$, whereas the initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Corollary 3.4.5 *Let $i \geq 1$. Then there exists a structure and algorithms for the Union-Find problem that can be implemented as a pointer/ $\log n$ solution such that for a universe of n elements all Unions can be performed in $O(n.a(i,n))$ time and each Find can be performed in $O(1)$ time, while the structure uses $O(n)$ space and can be initialised in $O(n)$ time.*

3.5 An Alternative for Path Compression

By applying UF(i) structures for appropriate values of i, we obtain a Union-Find structure that has the same total complexity as the method using path compression, but that has a trade-off in the worst-case complexities for Union and Find operations. This is expressed in the following theorems.

Theorem 3.5.1 *There exists a data structure and algorithms that solve the Union-Find problem with the following properties: the total time needed for all Unions and m Finds is $O(n + m.\alpha(n,n))$, while each Find takes $O(\alpha(n,n))$ time, where n is the total number of elements ($n \geq 2$). Moreover, the data structure and algorithms can be implemented with this performance as a pointer/ $\log n$ solution.*

Proof. First compute an Ackermann net for n. Then $\alpha(n,n)\,(= \alpha_{ack})$ can be obtained from the net. Now use UF($\alpha(n,n)$). □

Theorem 3.5.2 *There exists a data structure and algorithms that solve the Union-Find problem with the following properties: the total time needed for all Unions and m Finds is $O(n + m.\alpha(m,n))$, while the f^{th} Find takes $O(\alpha(f,n))$ time, where n is the total number of elements ($n \geq 2$). Moreover, the data structure and algorithms can be implemented with this performance as a pointer/ $\log n$ solution.*

Proof. We make use of UF(i) structures. All the set names that are present at some time are contained in a list. Hence, if some set name is removed because of a Union, then it must be removed from this list too. (This can be easily implemented by providing additional pointer fields in set names to form a doubly linked list.) Moreover, some additional variables are maintained, which will be introduced henceforth. Initially, make a UF(i) structure with $i = \alpha(n,n)$. This is performed like in the case of Theorem 3.5.1. At any moment, let f be the number of Finds performed thus far. Each time that $\alpha(f,n)$ becomes one smaller than $i\ (= \alpha(f-1,n))$, rebuild the structure UF(i) to a UF($i-1$)-structure. The rebuilding is as follows.

If $i = 1$ then nothing needs to be done, since we have for all future values of f occurring in this situation: $\alpha(f,n) = 1$. Otherwise, we only have to check whether $\alpha(f,n)$ decreases by one after we have increased f. This can be inspected by checking whether $a(i-1,n) \leq 4.\lceil \frac{f}{n} \rceil$ (cf. Lemma 2.3.7). The value $a(i-1,n)$ can be obtained in $O(1)$ time from the Ackermann node $(min\{i-1, \alpha_{ack}+1\}, -1)$ that can be reached by means of pointer $rowi$ pointing to $(min\{i, \alpha_{ack}+1\}, -1)$ and by means of a net pointer (cf. Subsection 3.3.3). Hence, the comparison can be made in $O(1)$ time. (Note that the value $4.\lceil \frac{f}{n} \rceil$ can easily be maintained for increasing f by means of comparisons and additions in $O(1)$ time and $O(1)$ space. In this way it is not necessary to use divisions and to take entier values each time.)

If indeed $a(i-1,n) \leq 4.\lceil \frac{f}{n} \rceil$ then the UF(i) structure is rebuilt to a UF($i-1$) structure in the following way. First adapt pointer *rowi* pointing to $(min\{i, \alpha_{ack}+1\}, -1)$ to point to node $(min\{i-1, \alpha_{ack}+1\}, -1)$ which can be done in $O(1)$ time. Moreover, for each set name s, dispose all status records $status_{s,j}$ for $1 \leq j \leq i-1$ (which are at most n records, cf. Subsection 3.3.3). For each set name s enumerate its elements e.g. in the way of procedure *generate** (cf. Figure 3.9) that, contrary to procedure *generate*, does not remove the intermediate nodes yet.

Figure 3.9: Procedure generate*(s,i) in UF(i) $(i \geq 1)$.

```
(1) function generate*(s,i); return < list of nodes >;
(2) {generate*(s,i) generates the elements in set(s,i)}
(3) if $i = 1 \vee$ weight$(s,i) = 1 \longrightarrow$ generate* := sons(s);
(4) [] $i > 1 \wedge$ weight$(s,i) > 1 \longrightarrow$ generate* := sons(generate*(s,i-1));
(5) fi
```

If $set(s,i)$ contains only one element, then the only thing to do is to make a status record $status_{s,i-1}$ with values $weight(s,i-1) = 1$, $level(s,i-1) = -1$, $clus(s,i-1) = nil$ and with corresponding pointer fields (the pointers into the Ackermann net can be adapted by means of the "old" record $status_{s,i}$).
If $i-1=1$, adapt all father values of the elements to s, perform the original procedure $generate(s,i)$ to get rid of all "old" intermediate nodes between s and $set(s,2)$ and to initialise $sons(s)$ to the list of these elements. Finally, make a status record $status_{s,i-1}$ with $weight(s,i-1) =$ [the number of elements in the set].
Otherwise (i.e., $i-1 > 1$ and $weight(s,i) > 1$), make a new cluster node c, make $father(c) := s$ and adapt all father values of the elements to c. Then adapt $sons(c)$ to the list of these elements and perform the original procedure $generate(s,i)$ to get rid of all "old" intermediate nodes between s and $set(s,i)$. Finally, adapt $sons(s)$ to $\{c\}$ and adapt the status record $status_{s,i-1}$ as follows: $weight(s,i-1) =$ [the number of elements in the set], $level(s,i-1) = -1$, $clus(s,i-1) = c$ and adapt the corresponding pointer fields accordingly (the pointers into the Ackermann net can be adapted by means of the "old" record $status_{s,i}$.) and make a status record $status_{s,i-2}$ similar to the case above.

Finally, for all cases, adapt the pointer from node s that points to the record $status_{s,i}$ such that it points to $status_{s,i-1}$ and dispose record $status_{s,i}$. Trivially, all this can be done in $O(n)$ time.

This rebuilding of the structure to a UF($i-1$) structure is now performed in the following way. Until n next Finds have been passed or a next Union has to be performed, perform $O(1)$ time of the building of UF($i-1$) per Find instruction and if a Union operation occurs before n next Finds have been performed, perform the remainder of the building first during this Union operation and then perform the usual Union operation on this new structure. It is easily seen that during the rebuilding there always remains a tree path between an element and its set name, which is of length at most i. Therefore, during the rebuilding, a Find operation can be performed in $O(i) = O(i-1)$ time (since $i-1 \geq 1$). Moreover, a Union is never executed during a period of rebuilding.

We now show that a rebuilding is completed before a next one has to be started and we consider the time complexities.

Let f_1 and f_2 be two consecutive values of f after which a rebuilding is started. From Lemma 2.3.12 it follows that $f_2 - f_1 \geq 2.n$ and hence that a rebuilding is completed before a next one is started (cf. the conditions for starting a rebuilding). Hence, at each moment the structure that is present is either UF($\alpha(f,n)$) or an "intermediate" structure between UF($\alpha(f,n)+1$) and UF($\alpha(f,n)$) such that the root paths of the elements are of length at most $\alpha(f,n)+1$ ($\leq 2.\alpha(f,n)$). Therefore, the time needed for a Find operation is obviously $O(\alpha(f,n))$ (Note that an "intermediate" structure is not used for Unions, but only for Find operations.)

We show that the time needed for performing all rebuildings and all Union and Find operations together is $O(n + m.\alpha(m,n))$.

Initially, we have the structure UF(i) with $i = \alpha(n,n)$. By Theorem 3.4.4 and Lemma 2.3.7 it follows that all Unions in this structure take $O(n)$ altogether.

Now consider the rebuildings of a UF(i) structure to a UF($i-1$) structure. Suppose this is started because of the $(f+1)^{th}$ Find operation: let f be such that $\alpha(f+1,n) = \alpha(f,n) - 1$. By Lemma 2.3.11 we have if $i := \alpha(f,n)$:

$$8.(f+1) \geq n.a(i-1,n) \quad \text{and} \quad f \geq n \geq 2. \tag{3.5}$$

Now charge all cost for performing the rebuilding and for performing future Unions in UF($i-1$) to the previous $\lceil \frac{1}{2} f \rceil$ Find operations. Then by Theorem 3.4.4 and (3.5) it follows that each of these Finds is charged for $O(1)$ time. By Lemma 2.3.12 it follows that any Find operation can only be charged at most once. Therefore all Union and (re-) building operations take $O(n+m)$ time together.

Finally, consider the cost of all Find operations. We already showed that the f^{th} Find operation takes at most $c.\alpha(f,n)$ time for some appropriate constant c. Hence, the total cost of these operations is bounded by

$$\begin{aligned}
&\sum_{f=1}^{m} c.\alpha(f,n) \\
&= \quad c.\sum_{f=1}^{m} \alpha(m,n) + c.\sum_{f=1}^{m} (\alpha(f,n) - \alpha(m,n)) \\
&= \quad c.m.\alpha(m,n) + c.\sum_{\alpha=\alpha(m,n)+1}^{\alpha(1,n)} (\alpha - \alpha(m,n)).|\{f|\alpha(f,n) = \alpha\}|
\end{aligned}$$

$$\begin{aligned}
&= c.m.\alpha(m,n) + c.\sum_{\alpha=\alpha(m,n)+1}^{\alpha(1,n)} |\{f|\alpha(f,n) \geq \alpha\}| \\
&\leq c.m.\alpha(m,n) + c.\sum_{i=0}^{\alpha(1,n)-\alpha(m,n)-1} (\frac{1}{3})^i.m \\
&\leq 3.c.m.\alpha(m,n)
\end{aligned}$$

where $\alpha(f,n) > \alpha(m,n) \Rightarrow f < m$ and $f \leq m \Rightarrow \alpha(f,n) \geq \alpha(m,n)$ are used (cf. Corollary 2.3.8) and where Lemma 2.3.12 provides the first unequality. This concludes the proof of the theorem. □

3.6 Increasing the Number of Elements

We now consider structures that, aside from the operations Union and Find, allow the operation

- Insert(x): add a new element x to the universe, create the singleton set $\{x\}$ and output the name of this set.

In this way the collection of elements can be augmented. We call the problem that deals with the above three operations the Union-Find-Augment Problem. Note that in order to have the appropriate Ackermann values we have to augment the Ackermann net from time to time (cf. Section 2.4).

Theorem 3.6.1 *The UF(i) structure can be augmented to allow Insert operations, such that it remains a data structure with algorithms that can be implemented as a pointer/* $\log n$ *solution and that solves the Union-Find problem. The total time that is needed for all Union operations in a UF(i)-structure until a moment on which there are n elements is* $O(n.a(i,n))$ *while the time needed for a Find operation is* $O(i)$*, an insertion can be performed in* $O(1)$ *time and the entire structure takes* $O(n)$ *space* ($i \geq 1$, $n \geq 2$)*. The initialisation can be performed in* $O(n_{init})$ *time, where* n_{init} *is the number of elements at the initialisation.*

Proof. It is easily seen that a UF(i) allows element insertions with the above time bounds if the required Ackermann values are available and if there always is a pointer available to the Ackermann node $(min\{i, \alpha_{ack}+1\}, -1)$ (viz., parameter *rowi* in Subsection 2.4.2). Therefore, the only difficulty is to augment the Ackermann net properly from time to time. We do this as follows. Let $n_{init} > 1$ be initial the number of elements. Initially, make an Ackermann net for value $n_{ack} = 2.2^{\lceil \log n_{init} \rceil}$. (This can easily be done by making such a net for value $2.n_{init}$, and then by taking for n_{ack} the value $2^{a(1,n)}$ which would have been stored in the node $(1, a(1,n))$ if it was not replaced by ∞.)

Now each time an element is inserted in a collection with n elements such that $2.n \leq n_{ack} < 2.(n+1)$, the Ackermann net is to be augmented to a net for $2n_{ack}$. However, the augmentation of the Ackermann net can give a new list for row $\alpha(n_{ack}, n_{ack}) + 2$. Hence if $i > \alpha(n_{ack}, n_{ack}) + 1$ then the pointers of the status records of UF(i) that point into the Ackermann net for layers $\alpha(n_{ack}, n_{ack}) + 2$ up to i need to be adapted to point to the new list. This is done by means of a list of all set names that are present at some time. Moreover, the variable *rowi* need to be adapted.

By Lemma 2.4.4 it follows that adaptations of the Ackermann net can be performed in $O(\alpha(n_{ack}, n_{ack})) = O(\alpha(n,n))$ time. Adaptations of the status records can be performed in $O(n)$ time, since obviously there are only $O(n)$ status records in a structure with n nodes (cf. Subsection 3.3.3) and since the relevant pointers have to be redirected to one of only 5 new Ackermann nodes only. Until $\frac{1}{2}n$ next Finds or Inserts have been passed or a next Union has to be performed, perform $O(1)$ time of these Ackermann calculations per Find instruction and if a Union operation occurs before $\frac{1}{2}n$ next Finds or Inserts have been performed, perform the remainder of the calculations first during this Union operation and then perform the usual Union operation . Then it can easily be seen that the adaptations of the Ackermann net are completed before new adaptations need to be performed (since before a new adaptation, the number of nodes must be doubled) and before a next Union is executed, and that the time bounds for the three operations Union, Find and Insert do not change in order. Finally, in this way the Ackermann net always contains all relevant values up to n (the number of nodes that are present), since always $n_{ack} \geq n$.

Remark: note that the adaptations of status records have to be performed as long as $i > \alpha(n_{ack}, n_{ack}) + 1$ only. Of course, this will not too often be the case. On the other hand, this can also be solved by creating an Ackermann net with i rows in the initialisation anyway, thus spending $O(n_{init} + i)$ time for initialisation and by adapting rows j with $1 \leq j \leq i$ only. □

Theorem 3.6.2 *There exists a structure that solves the Union-Find-Augment Problem in total time $O(n + m.\alpha(m,n))$, where n is the total number of elements and m is the number of Finds. Moreover, the f^{th} Find is performed in $O(\alpha(f, n_f))$ time, where n_f is the number of elements at the time of the f^{th} Find. An operation Insert(x) is performed in $O(1)$ time. The structure can be implemented with this performance as a pointer/ $\log n$ solution.*

Proof. We define a structure by using a UF(i) structure in which the operations Union, Find and Insert are performed and by rebuilding (transforming) the UF(i) structure to a UF(i') structure ($i' \neq i$) from time to time. By Theorem 3.6.1 it follows that a UF(i) structure allows Insert operations and that an Insert operation can be performed in $O(1)$ time. Like in Theorem 3.5.2 we maintain a list of actual set names (that obviously allows an insert operation too). In this way we have a structure with the operations Union, Find and Insert, together with additional computations, the so-called general updates. (The rebuilding of UF(i) structures is a part of these general updates.)

The following parameters are maintained with the following meaning at every moment during the entire sequence of operations. (In this description of the parameters, the initialisation of the entire structure is considered to be the first general update.) Let n_{base} denote the number of elements at the start of the last general update. Let f_{base} denote the number of completed Finds performed up to the start of the last general update. Let f_{last} denote the number of completed Finds performed since the start of the last general update. Let n denote the number of elements that are present. Let α_{base} be the value i that corresponds to the present structure UF(i) or that corresponds to the structure UF(i) that is being build at that moment. The parameters are changed as follows. Parameter f_{last} is increased by one at the end of a Find operation and parameter n is increased by one at the end of an Insert operation (note that an element is considered to be present after the insertion operation for that element is completed), whereas all parameters except parameter n are changed by a general update (according to the above description). Moreover, the pointer *rowi* into the Ackermann net (cf. Subsection 2.4.2) always points to node $(\alpha_{base}, -1)$ in the Ackermann net (which is always present). We first describe the strategy and prove a claim, and we show afterwards how the relevant values $\alpha(p, q)$ can be obtained.

Initially, let n and n_{base} be equal to the number n_{init} of elements, and let f_{last} and f_{base} be zero. Build an Ackermann net for $n_{ack} = 2^{\lceil \log 16 n_{base} \rceil}$ (cf. the proof of Theorem 3.6.1) and build a UF(α_{base}) structure with $\alpha_{base} = \alpha(f_{base}, 4n_{base})$.

Afterwards, perform the following strategy (that is related to the strategy presented in Theorem 3.5.2). Each time that at the end of an operation Find or Insert (hence *just after* the regular update of the relevant parameters) the condition

$$(\ \alpha(f_{base} + f_{last}, 4n) < \alpha_{base} \ \wedge\ f_{last} \geq 2 f_{base}\ ) \ \vee\ n = 4n_{base} \tag{3.6}$$

holds, we perform a so called *general update* as follows.

1. Adapt $f_{base} := f_{base} + f_{last}$, $f_{last} := 0$, $n_{base} := n$, $\alpha_{old} := \alpha_{base}$ and $\alpha_{base} := \alpha(f_{base}, 4n)$.
2. (a) Augment the Ackermann net for $n_{ack} = 2^{x_{ack}}$ to an Ackermann net for $n'_{ack} = 2^{x'_{ack}}$ such that $16 n_{base} \leq n'_{ack} < 32 n_{base}$ (if necessary).
   (b) If $\alpha_{base} \neq \alpha_{old}$ rebuild the present structure UF(α_{old}) to a UF(α_{base}) structure and adapt pointer *rowi* at the beginning of this rebuilding.

As stated above we do not consider how to compute value $\alpha(f_{base}, 4n)$. The above augmentation of the Ackermann net is performed once or twice in the way of Theorem 2.4.4 and takes $O(\alpha(n_{ack}, n_{ack})) = O(\alpha(n_{base}, n_{base}))$ time. It will appear that

$\alpha_{old} - 1 \leq \alpha_{base} \leq \alpha_{old} + 2$, which yields that pointer *rowi* can be adapted in $O(1)$ time. The above rebuilding of UF(α_{old}) to UF(α_{base}) is performed in the way of Theorem 3.5.2 and takes $O(n_{base})$ time (by Theorem 3.6.1). (During the augmentation and rebuilding new elements are inserted as new elements in UF(α_{base}).)

The general update is executed as follows.

1. The adaptation of the parameters is performed immediately at the end of the Find or Insert operation in which condition (3.6) becomes true. These adaptations will appear to take $O(1)$ time.
2. The execution of the augmentation of the Ackermann net (a) and the execution of the rebuilding of the structure (b) (henceforth just called augmentation and rebuilding) is performed in the same way as in the case of Theorem 3.5.2, where Insert operations are treated in the same way as Find operations: until $\frac{1}{2}.n_{base}$ next Finds or Inserts have been passed or a next Union has to be performed, perform $O(1)$ time of the augmentation or the rebuilding per Find or Insert instruction and if a Union operation occurs before $\frac{1}{2}.n_{base}$ next Finds and Inserts have been performed, perform the remainder of the rebuilding first during this Union operation and then perform the usual Union operation on this new structure.

The above extra $O(1)$ time that is spent in Find or Insert operations does not increase the worst-case time order of a these operations. Therefore we will ignore this extra time for these two operations henceforth. Note that the execution of a general update is distributed over at most $\frac{1}{2}.n_{base}$ operations. Moreover, note that if condition (3.6) becomes true then either an augmentation or a rebuilding needs to be performed anyway. Finally, it is easily seen that always

$$\alpha_{base} = \alpha(f_{base}, 4n_{base}) \wedge n_{base} \leq n \leq 4n_{base}. \tag{3.7}$$

Claim 3.6.3 *If the strategy described above is followed, then at every moment*

$$\alpha_{base} - 1 \leq \alpha(f_{base} + f_{last}, 4n) \leq \alpha_{base} + 2.$$

Moreover, there are at least $\frac{8}{3}n_{base}$ Find operations or at least $3n_{base}$ Insert operations after the start of a general update with n_{base} elements before a next one is started. Therefore a general update is finished before a next one can be started.

Proof. Just after the execution of part 1 of a general update, the inequality stated in the claim obviously holds (cf. (3.7)).

If at some moment $\alpha(f_{base} + f_{last}, 4n) < \alpha(f_{base}, 4n_{base})$ and hence by (3.7) $\alpha(f_{base} + f_{last}, 4n_{base}) < \alpha(f_{base}, 4n_{base})$ while no general update is started, it follows by the update condition (3.6) that $f_{last} < 2f_{base}$ and hence $f_{last} + f_{base} < 3f_{base}$. On the other hand, if a general update is going to be started then either $\alpha(f_{base} + f_{last}, 4n) = \alpha(f_{base}, 4n_{base}) - 1$ or $f_{last} = 2f_{base}$ because f_{last} is increased one at a time and because a is rebuilding started as soon as condition (3.6) is true. Concluding, we have $f_{last} + f_{base} \leq 3f_{base}$ or $\alpha(f_{base} + f_{last}, 4n) = \alpha(f_{base}, 4n_{base}) - 1$. Now Lemma 2.3.12 gives in case of $f_{last} + f_{base} \leq 3f_{base}$ that $\alpha(f_{base} + f_{last}, 4n_{base}) \geq \alpha(f_{base}, 4n_{base}) - 1$ and hence by (3.7) $\alpha(f_{base} + f_{last}, 4.n) \geq \alpha(f_{base}, 4.n_{base}) - 1$.

On the other hand, since $n \leq 4n_{base}$ we have $\alpha(f_{base} + f_{last}, 4n) \leq \alpha_{base} + 2$ which is seen as follows. If $a(i,n) \leq x \wedge i \geq 1 \wedge x \geq 4$ then $a(i, 4n) \leq x + 2$ and hence by Lemma 2.3.10 and Lemma 2.3.6 $a(i+2, 4n) < max\{\frac{1}{9}(x+2), 4\} \leq max\{\frac{1}{6}x, 4\}$. Now let $x = 4.\lceil\frac{f_{base}+f_{last}}{4n_{base}}\rceil$. Since by (3.7)

$$\lceil\frac{f_{base} + f_{last}}{4n}\rceil \geq \lceil\frac{f_{base} + f_{last}}{16n_{base}}\rceil \geq \frac{1}{4}.\lceil\frac{f_{base} + f_{last}}{4n_{base}}\rceil$$

it follows by the above observations that

$$a(i,n) \leq 4\lceil\frac{f_{base} + f_{last}}{4n_{base}}\rceil \Rightarrow a(i+2, 4n) \leq 4\lceil\frac{f_{base} + f_{last}}{4n}\rceil$$

and hence $\alpha(f_{base} + f_{last}, 4n) \leq \alpha_{base} + 2$.

Consider the condition (3.6) again :

$$(\ \alpha(f_{base} + f_{last}, 4n) < \alpha_{base} \wedge f_{last} \geq 2f_{base}\ )\ \vee\ n = 4n_{base}.$$

By Corollary 2.3.8 and Lemma 2.3.11 it follows that $(f_{base} + f_{last} > 4n \geq 4n_{base}\ \wedge\ f_{last} \geq 2f_{base})\ \vee\ n = 4n_{base}$ and hence $f_{last} \geq \frac{8}{3}n_{base} \vee n = 4n_{base}$. Hence, at least $\frac{8}{3}.n_{base}$ Find or at least $3.n_{base}$ Insert operations must be performed after a general update with n_{base} elements before a next one is started. Since a general update takes $\frac{1}{2}n_{base}$ operations at the most, these are finished before the next one is started. □

We discuss how to compute the relevant values $\alpha(p,q)$. The value $\alpha(f_{base} + f_{last}, 4.n)$ used in a general update can be obtained in a way similar to that of Theorem 3.5.2 as follows.

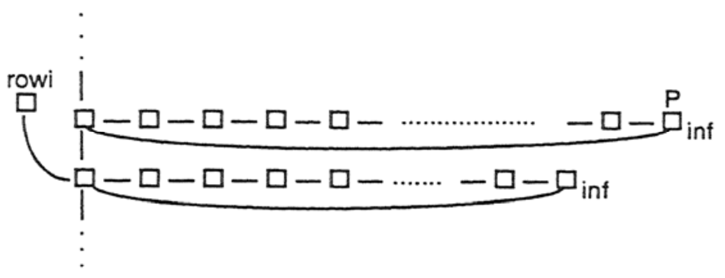
First we consider how to compute condition (3.6) only. Note that by Claim 3.6.3 the value $\alpha(f_{base} + f_{last}, 4n)$ only needs to be available if at least $2n_{base}$ Find operations or $2n_{base}$ Insert operations have been performed since the last general update, i.e., $f_{last} \geq 2n_{base}$ or $n - n_{base} \geq 2n_{base}$. Therefore we augment condition (3.6) to

$$(\ \alpha(f_{base} + f_{last}, 4n) < \alpha_{base} \wedge f_{last} \geq 2f_{base} \wedge f_{last} + n \geq 3n_{base}\ )\ \vee\ n = 4n_{base}. \quad (3.8)$$

At the time that $f_{last} + n \geq 2n_{base}$ holds, the last augmentation of the Ackermann net for n_{ack} with $16n_{base} \leq n_{ack} \leq 32n_{base}$ is completed, and hence the net fits for value $4n$ ($\leq 16n_{base}$). Moreover, the condition only uses whether $\alpha(f_{base} + f_{last}, 4n) <$

α_{base}. Therefore, it suffices to compare the value $a(\alpha_{base} - 1, 4n)$ with the fraction $4.\lceil \frac{f_{last}+f_{base}}{4n} \rceil$ (cf. Lemma 2.3.7), and only if $\alpha_{base} > 1$.
The value $a(\alpha_{base} - 1, 4n)$ can be obtained as follows. Pointer *rowi* points to node $(\alpha_{base} - 1, -1)$. Therefore node $P = (\alpha_{base} - 1, a(\alpha_{base} - 1, n_{ack}))$ are available in $O(1)$ time, together with value $a(\alpha_{base} - 1, n_{ack})$ (cf. Subsection 2.4.2). Now traverse the list for row $\alpha_{base} - 1$ backwards starting from P until we have an Ackermann node which has a predecessor with value smaller then $4n$. If the number of nodes passed in this way is x, then apparently $a(\alpha_{base} - 1, 4n) = a(\alpha_{base} - 1, n_{ack}) - x$ (cf. Figure 3.10).

Figure 3.10:



Since we have that $4n \leq 16n_{base} \leq n_{ack} \leq 32n_{base} \leq 32n$ it follows that $x = O(1)$ and hence that the above manipulations take $O(1)$ time. Hence, this comparison can be performed in $O(1)$ time. Finally note that since $a(i, n) \leq n$ $(i, n \geq 1)$ and since $4.\lceil \frac{f_{base}+f_{last}}{4n} \rceil$ is only used to compare with $a(i, n)$, we only need the value $min\{4.\lceil \frac{f_{base}+f_{last}}{4n} \rceil, 4n\}$, which can be maintained by means of additions, subtractions and comparisons only in $O(1)$ time and $O(1)$ space for increasing n and $f_{base} + f_{last}$.

On the other hand, if $\alpha(f_{base} + f_{last}, 4n)$ has to be computed in the case that $n = 4n_{base}$, this can be performed similarly for rows α_{base} up to $max\{\alpha_{base}+2, \alpha_{ack}\}$ only, since $\alpha(f_{base} + f_{last}, 4n) \leq \alpha_{base} + 2$ (cf. Claim 3.6.3) and since by $4n \leq n_{ack}$ we have $\alpha(f_{base} + f_{last}, 4n) \leq \alpha_{ack} = \alpha(n_{ack}, n_{ack})$. (Like above, the Ackermann net fits for value $4n$.) Therefore, this can be performed in $O(1)$ time too.

Finally, concerning the initialisation of the entire system, value $\alpha(0, 4n_{init})$ can be obtained during the initial construction of the Ackermann net in a similar way.

We show that this strategy yields the time bounds stated above.

By Claim 3.6.3, Equation (3.7) and the observation that at any time an augmentation and a rebuilding can be performed in $O(n_{base})$ time, it follows that all augmentations and rebuildings together that are performed up to some moment take at most $O(n + m)$ time where n is the number of elements at that moment and m is the total number of Finds that have been performed up to that moment.

By Theorem 3.6.1 a Find operation in UF(i) can be performed in O(i) time. Furthermore, note that like in the proof of Theorem 3.5.2 during the rebuilding of a UF(i) structure to a UF(i') structure a Find operation can be performed in $O(i')$ time, since the root path of each element is at most of length $i' + 1$ (since by Claim 3.6.3 $i-1 \leq i' \leq i+2$) Let c' be a constant in accordance with this observations, i.e., each Find takes at most $c'.\alpha_{base}$ time. Moreover, let c'' be a constant in accordance Theorem 3.6.1, i.e., all Unions performed in UF(α_{base}) need at most $c''.n.a(\alpha_{base}, n)$ time together if the final number of elements is n (for all $n \geq 2$). Let $c = max\{c', c''\}$.

By Claim 3.6.3 it follows that a Find operation takes at most

$$c.\alpha_{base} \leq 2.c.\alpha(f_{last} + f_{base}, 4n) \leq 4.c.\alpha(f_{last} + f_{base}, n)$$

time. This yields the time bound for a single Find operation (i.e., the $(f_{last} + f_{base})^{th}$ Find) as stated in the theorem.

We now consider the cost of Finds an Union together. Let the parameters f_{base}, f_{last} and n_{base} be defined as before. Then $C(f_{base}, f_{last}, n_{base})$ denotes the cost of all $f_{base} + f_{last}$ Find operations together with the cost of all Union operations performed until the start of the last (re-)building. (Note that the cost of augmentations and rebuildings are *not* included.) We show that at any time

$$C(f_{base}, f_{last}, n_{base}) \leq 28.c.\alpha_{base}.f_{base} + c.\alpha_{base}.f_{last} + 6.c.n_{base}. \qquad (3.9)$$

Obviously, (3.9) holds initially. Note that this cost parameter can be changed by a Find operation or a general update only. Therefore we consider four cases in which this cost parameter can be changed, viz. the Find operation and three cases of the general update, and show that (3.9) is preserved in these cases. Below, the unprimed parameters denote the parameters right before *these* four cases of the operations while the primed parameters denote the parameters right after the adaptations of the parameters.

1. The execution of a Find operation.

   By the choice of c it follows that a Find operation always takes at most $c.\alpha_{base}$ time. Hence, (3.9) remains valid in case of the execution of a Find operation.

2. Starting a general update when $\alpha(f_{base} + f_{last}, 4n) = \alpha_{base}$, i.e., no rebuilding needs to be performed. Then the adaptations of the parameters in the general update obviously do not violate (3.9).

3. Starting a general update when $\alpha(f_{base} + f_{last}, 4n) < \alpha_{base}$, i.e., a rebuilding is started. Then $\alpha(f_{base} + f_{last}, 4n) = \alpha_{base} - 1$ (by Claim 3.6.3). According to (3.6) we distinguish two cases:

- $f_{last} \geq 2 f_{base}$
- $n = 4 n_{base}$.
  In this case, we have by $\alpha(f_{base} + f_{last}, 4.n) < \alpha_{base} = \alpha(f_{base}, 4n_{base})$, by Lemma 2.3.6 and by Lemma 2.3.7

$$\begin{aligned} 4.\lceil \frac{f_{base}}{4.n_{base}} \rceil &< a(\alpha(f_{base} + f_{last}, 4.n), 4n_{base}) \\ &\leq a(\alpha(f_{base} + f_{last}, 4.n), 4.n) \\ &\leq 4.\lceil \frac{f_{last} + f_{base}}{4.n} \rceil. \end{aligned}$$

  Because $n = 4n_{base}$ this implies $f_{last} > 3 f_{base}$.

Hence, in both cases we have $f_{last} \geq 2 f_{base}$.
Since $\alpha(f_{base} + f_{last}, 4n) < \alpha_{base} = \alpha(f_{base}, 4n_{base}) \leq \alpha(f_{base}, 4n) \leq \alpha(0, 4n)$ we find by using Lemma 2.3.11

$$4.n.a(\alpha(f_{base} + f_{last}, 4n), 4n) \leq 8.(f_{base} + f_{last}). \tag{3.10}$$

and hence by Lemma 2.3.6

$$4.n.a(\alpha_{base}, 4n) \leq 8.(f_{base} + f_{last}). \tag{3.11}$$

The total cost for all Unions performed after the start of the last (re-)building of UF(α_{base}) is at most $c.n.a(\alpha_{base}, n)$ time and hence by (3.11) at most $8.c.(f_{base} + f_{last})$ time. Therefore,

$$\begin{aligned} &C(f'_{base}, f'_{last}, n'_{base}) \\ &\leq \; C(f_{base}, f_{last}, n_{base}) + 8.c.(f_{base} + f_{last}) \\ &\leq \; 28.c.\alpha_{base}.f_{base} + c.\alpha_{base}.f_{last} + 6.c.n_{base} + 8.c.(f_{base} + f_{last}) \\ &\leq \; 28.c.(\alpha_{base} - 1).f_{base} + 26.c.f_{last} + c.\alpha_{base}.f_{last} + 6.c.n_{base} \\ &\leq \; 28.c.(\alpha_{base} - 1).(f_{base} + f_{last}) + 6.c.n_{base} \\ &= \; 28.c.\alpha'_{base}.f'_{base} + 6.c.n'_{base} \end{aligned}$$

where the third inequality follows with $f_{base} \leq \frac{1}{2} f_{last}$ and the fourth inequality follows with $\alpha_{base} - 1 = \alpha' \geq 1$.

4. Starting a general update when $\alpha(f_{base} + f_{last}, 4n) > \alpha_{base}$ and (hence) $n = 4n_{base}$, i.e., a rebuilding is started.

   Combining Lemma 2.3.11 and Lemma 2.3.7 gives

$$a(\alpha(f_{base}, n), n) \leq 4 \vee n.a(\alpha(f_{base}, n), n) \leq 8 f_{base}. \tag{3.12}$$

The total cost for all Unions performed after the start of the last (re-) building of UF(α_{base}) is at most $c.n.a(\alpha_{base}, n)$ time and hence by (3.12) at most $8.c.f_{base} + 16.c.n_{base}$ time. Therefore,

$$\begin{aligned} & C(f'_{base}, f'_{last}, n'_{base}) \\ \leq\ & C(f_{base}, f_{last}, n_{base}) + 8.c.f_{base} + 16.c.n_{base} \\ \leq\ & 28.c.\alpha_{base}.f_{base} + c.\alpha_{base}.f_{last} + 6.c.n_{base} + 8.c.f_{base} + 16.c.n_{base} \\ \leq\ & 28.c.(\alpha_{base}+1).(f_{base}+f_{last}) + 22.c.n_{base} \\ \leq\ & 28.c.\alpha'_{base}.f'_{base} + 6.c.n'_{base}. \end{aligned}$$

By the above result and by the choice of c, at any moment the total cost of *all* Unions and Finds is bounded by

$$C(f_{base}, f_{last}, n_{base}) + c.n.a(\alpha_{base}, n)$$

and hence by

$$28.c.\alpha_{base}.(f_{base} + f_{last}) + 6.c.n_{base} + c.n.a(\alpha_{base}, n)$$

which is

$$O(\alpha_{base}.(f_{base} + f_{last}) + n)$$

because of (3.12) and $n_{base} \leq n \leq 4n_{base}$. Since an Insert operation takes $O(1)$ time and since the time needed for all augmentations and rebuildings is $O(n + m)$, the total cost at any moment is (by using $n_{base} \leq n \leq 4n_{base}$, Claim 3.6.3 and $\alpha(f_{base} + f_{last}, 4n) \leq 2 + \alpha(f_{base} + f_{last}, n) \leq 3\alpha(f_{base} + f_{last}, n) = 3\alpha(m, n)$)

$$O(n + m.\alpha(m, n))$$

where n is the number of elements at that moment and m is the number of Finds performed up to that moment. This concludes the proof. □

3.7 Concluding Remarks

In this chapter we have presented a collection of Union-Find structures, including structures that a have time complexity equal to the algorithms using path compression, but that have a small worst-case time complexity for the Finds instead of for the Unions.

Note that $\alpha(m,n) \leq 3$ for $n \leq 2^{2^{\cdot^{\cdot^{2}}}} \Big\}$ 65536 two's. Therefore in practice there is no need to perform transformations of structures like those occurring in Section 3.6: structure UF(3) suits for all practical situations. The time bound for the Unions in UF(3) is $c.n.a(3,n) \leq 4.c.n$ for such n, where c is a relatively small constant (cf.

Section 3.4 for its definition). Moreover, a Find can be performed in $\leq 3.c_{point}$ time, where c_{point} is the time needed to perform a pointer comparison and to access a node by means of a pointer (which is small).
Of course, the same can be said for UF(2): all Unions on n elements take $\leq c.n.a(2,n) \leq 4.c.n$ time for $n \leq 2^{16} = 65536$ and take $\leq 5.c.n$ time for *very* large practical values $n \leq 2^{2^{\cdot^{\cdot^{2}}}} \Big\}^{5 \text{ two's}} = 2^{65536}$ (which is *slightly* more than the time needed in UF(3)), whereas a Find operation takes $\leq 2c_{point}$ time.
Moreover, note that in all practical situations for UF(2) and UF(3) only the nontrivial Ackermann values 16 and 65536 need to available (being $A(2,3)$ and $A(2,4) = A(3,3)$) respectively), so there is no need to compute Ackermann values (neither in the initialisation nor in case new elements are inserted like in Section 3.6).

Therefore, we conjecture that UF(2) and UF(3) are fast and simple structures for all practical situations, with a constant time Find query.

On the other hand, note that all arithmetic occurring in the algorithms can be performed by using additions, subtractions and comparisons only. Furthermore, in case arrays are used for representing elements, an Ackermann matrix can be used instead of an Ackermann net. (In this case the array that contains these values needs to be of size $O(\log n. \log n)$ only.)

In case a set must be enumerated, note that this can easily be done in linear time (i.e., linear to the set size) if the set name or an element of that set are given. Viz., by performing a call $generate^*(s,i)$ (cf. Fig. 3.9) in the structure $UF(i)$ that is currently used, where if an element $x \in set(s,i)$ is given instead of set name s itself, then this call is preceded by $s := FIND(x)$.

Finally, we mention some direct applications for special cases of the Union-Find problem. Firstly, if the number of Finds m is known in advance, then *each* Find can be executed in $O(\alpha(m,n))$ time by taking structure UF($\alpha(m,n)$), where $\alpha(m,n)$ can be computed similar to the way described in the proof of Theorem 3.6.2. Secondly, the Union-Find algorithm for the special case of the Union-Find problem on a Random Access Machine that is presented in [13] (i.e., where the structure of the sequence of Union applications is known in advance), can be altered to an algorithm with the same overall time bound $O(n + m)$ such that each Find operation takes $O(1)$ time in the worst case. This can be done by applying UF(2) instead of an algorithm with path compaction.

Chapter 4

A Fast and Optimal Algorithm for the Generalized Split-Find Problem on Pointer Machines

4.1 Introduction

Let U be a universe of n elements (listed in a fixed order). Suppose U is partitioned into a collection of ordered sets whose contents are given in order, and suppose we want to be able to perform the following operations:

- Split(x): split the set that contains element x into two new sets, viz, one set containing all elements smaller then x and one set containing all elements that are greater than or equal to x,
- Find(x): return the name of the set in which element x is contained.

The occurring set names must satisfy the condition that at every moment, the names of the existing sets are distinct. This problem of efficiently maintaining the sets and supporting the operations is widely known as the Split-Find problem.

In [16], an algorithm was presented for the Split-Find problem that runs in $O((n + m).\log^* n)$ time for m operations. Later, in [13] an algorithm was given for the Split-Find problem that runs on a RAM in linear time. This algorithm relies on the RAM feature of address calculation and makes use of precomputed tables that e.g. encode structural information of subuniverses of size $O(\log\log n)$ in integers of at most $\log n$ bits. In [17], this solution was extended to deal with insertions. In [11], a solution for the Split-Find problem was presented that runs on a pointer machine in $O(n + m.\alpha(m, n))$ time. In Chapter 5, we will prove that this time complexity is optimal on pointer machines.

In this chapter, we consider the Generalized Split-Find problem on pointer machines. We consider the operation Split(x, y) that is given by

- Split(x, y): split the set that contains both x and y into two parts, viz. one set containing all elements z with $x \leq z < y$, and one set containing all remaining elements.

Here, y is allowed to have the value nil, which is interpreted as ∞ in the above inequality. Moreover, just before such a Split, a new element may be inserted as the predecessor of element x, and similar for y. (Note that if we always take $y = nil$, then we obtain the original Split-Find problem with only one split value.)

We present a collection of Generalized Split-Find structures that take $O(1)$ time per Find in the worst case and $O(n.a(i, n))$ time for all Split operations together for any fixed but freely chosen value of i. By means of these structures we construct a structure and algorithms that have a total time complexity of $O(n + m.\alpha(m, n))$ for m Finds and $n - 1$ Splits. All structures can be implemented in this time bound on a pointer machine. The generalized operations are important in e.g. problems in which named cyclic lists exist, where from time to time such a list is split into two parts according to two "split nodes", and each part forms a new named list again. Such an application can be found in the maintenance of 3-edge-connected components or 3-vertex-connected components of graphs (see e.g. Chapters 6, 8 and 9). In the last part of this chapter, we generalize the problem to operations for splitting at any fixed number of elements and adapt our solutions for this case. In such operations for a fixed number k, a set is partitioned into intervals according to the (at most k) splitting elements: then, if the intervals are numbered consecutively, one resulting set consists of the concatenation of all even numbered intervals whereas the other resulting set consists of all odd numbered intervals. Note that this indeed is a generalization of the above Split for two elements. The Generalized Split-Find structures we present are closely related to the structures in [11], but, nevertheless, it seems that the structures in [11] cannot straightforwardly be adapted to allow Generalized Splits.

In Section 4.2, we describe the Generalized Split-Find problem on a pointer machine. In Section 4.3, the Generalized Split-Find structures GSF(i) are presented by means of a recurrence, where a GSF(i) structure has time complexity $O(n.a(i, n))$ for all Splits and for n elements, while an individual Find can be performed in $O(i)$ worst-case time. These time bounds are proved in Section 4.4. In Section 4.5, a structure is given based on transformations of GSF(i) structures into GSF$(i - 1)$ structures: the resulting Generalized Split-Find structure has a time complexity of $O(n+m.\alpha(m, n))$ for all Splits on n elements and m Finds. In Section 4.6, we consider the multiple Split for any fixed number of split elements as defined above and discuss simple operations to detect the order of elements in a set.

4.2 The Generalized Split-Find Problem

We formulate the Generalized Split-Find problem (for splits with two split elements) in terms of nodes as follows. Let U be a linearly ordered collection of nodes, called elements. Suppose U is partitioned into a collection of ordered sets and suppose to each set a (new) unique node is related, called *set name*. We want to be able to perform the following operations:

- Split(x, y) (where either $y = nil$, or y is in the same set as x and $x < y$): given (pointers to) elements x and y (where y may be an element or nil), split the set that contains x into two new sets, viz., one set containing all elements z that satisfy $x \leq z < y$, and one set containing all other elements (where nil is taken to be ∞ and where the original set in which x was contained is destroyed), and relate set names to the two new sets

- Find(x): given (a pointer to) element x, return (a pointer to) the name of the set in which element x is contained.

The occurring set names must satisfy the condition that, at every moment, the names of the existing sets are distinct.

In particular, we will consider Extension Splits, that are given as follows.

- Split$((x', x), (y', y))$: given x and y as above, and given x' and y' that are (pointers to) new nodes or are nil, where y' satisfies $y = nil \Rightarrow y' = nil$, insert x' and y' as the predecessors of x and y respectively in the set in which x and y are contained (if $x' \neq nil$ and $y' \neq nil$, respectively), and perform Split(x, y).

Hence, a call Split(x, y) corresponds to a call Split$((nil, x), (nil, y))$. In the sequel, we will refer to the first Splits as regular Splits if we want to make a distinction. However, we will often omit the words "Extension" and "regular" in cases where it is clear which operations are considered.

The Split operation can be used in graphs for "splitting named cycles into two subcycles", where from time to time the name of a cycle must be obtained. The "splitting cycles into two subcycles" is as follows: split the cycle into two new parts, defined by two splitting nodes, and make a new cycle of each part. If the spit nodes are x and y, then this corresponds to Split$(min\{x, y\}, max\{x, y\})$ if the cycles are represented by ordered lists. (The minimum and maximum can easily be obtained, see Section 4.6.) This operation will be used in algorithms for maintaining the 3-edge-connected components of graphs. Moreover, the Extension Split can be applied in cases where (a representative of) each split node must be present in each resulting cycle: in each resulting cycle, a new representative of the split node that is not contained in that resulting cycle is inserted. Hence, the Extension Split operation can be considered to represent a "Split" in which firstly the nodes x and y themselves can be "split" (each can be split into two nodes) and then the regular Split is performed on the set of nodes. This operation can be used in algorithms for maintaining 3-vertex-connected components of graphs. We will call the above operations for cycles so-called *Circular Splits*.

4.3 The Generalized Split-Find Structure GSF(i)

In this section, we present a collection of structures GSF(i) ($i \geq 1$) that allow Extension Split and Find operations as described in Section 4.2. We will refer to Extension Splits by Splits. Let $i \geq 1$. A GSF(i) structure is a collection of rooted trees. The collection of trees is changed by Split operations. For each set name s, let the ordered set corresponding to name s be denoted by $set(s,i)$. Each set name is the root of a tree. The leaves of the tree with root s are the elements in $set(s,i)$ and have an equal distance $\leq i$ to the root. The nodes of the tree without the root can be split into layers of nodes that have equal distance to the root. The layer that contains the elements of $set(s,i)$ is called layer i, the other occurring layers are numbered consecutively in a decreasing order starting from layer i. For each layer there is a linear list of nodes of which the layer consists, called the linear layer list. For a node x, the linear layer list in which it is contained is denoted by $linlist(x)$. The order of the elements of $set(s,i)$ equals the order of the elements in the linear layer list of layer i.

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

Trees and linear layer lists are represented as follows. For each node x, the field $father(x)$ contains a pointer to its father if x is not a root and it contains the value nil otherwise. List $sons(x)$ contains the collection of sons of x, i.e, it contains pointers to its sons, and each son contains a pointer to the record in which it occurs. Moreover, $nrsons(x)$ is the number of sons of x. We will henceforth assume that for a node x, every change in the value of $father(x)$ or the removal of node x updates $sons(c)$ and $nrsons(c)$ of the old (and the new) father c of x properly. (This can be performed in $O(1)$ time, trivially.) Furthermore, each node x contains the fields $left(x)$ and $right(x)$ that contains pointers to the predecessor node and the successor node in the linear layer list of x respectively (and nil if such a node does not exist).

Structure GSF(i) allows the operations SPLIT($(x', x), (y', y), i$) and FIND(x) that satisfy the specification given in Section 4.2, where the parameter i in the former refers to the structure GSF(i) in which it is used. Function FIND(x) is given in Figure 3.1. (FIND(x) chases father pointers until a root is reached, which is the name of the set in which x is contained. It takes $O(i)$ time.)

The structures GSF(i) are defined inductively for $i > 1$, starting from a base structure GSF(1). First we outline the structure GSF(1) in Subsection 4.3.1, and then we describe GSF(i) for $i > 1$ in Subsection 4.3.2.

In the sequel, we denote by "GSF(i)-elements" the elements that are involved in the Generalized-Split-Find problem to be solved by the GSF(i) structure. Moreover, sometimes we will refer by "GSF(i) structure" to the algorithms too.

4.3.1 The Split-Find Structure GSF(1).

Structure GSF(1) is the structure that is derived from the straightforward set-merging algorithm. Recall that the set corresponding to set name s is denoted by $set(s,1)$ and that the nodes in $set(s,1)$ are in layer 1. According to the above constraints, for every element x, $father(x)$ contains a pointer to the name of the set in which it is contained and for every set name s, $sons(s) = set(s,1)$.

If GSF(1) is used to solve the Split-Find problem, then the initialisation for some (sub-)collection of elements into sets is straightforward (for any initial collection of sets, but usually one set being the universe), where an initial set is assumed to be given as the ordered linear list *linlist* of the elements. Procedure *build* that performs the initialisation is given in Figure 4.1 (where the last three parameters of the procedure are constants: in the next subsection we extend the procedure to $build(z, l, clussize, i)$ for GSF(i) with $i > 1$).

The Split operations can now be performed by the procedure SPLIT$((x', x), (y', y), 1)$ that is given in Figure 4.2. The auxiliary procedures *insert*, *splitlist* and *countparts* are given in Figures 4.3, 4.4 and 4.5. For brevity, we write statements like $left(p) := q$ and $x_{left} := left(x)$ instead of **if** $p \neq nil \longrightarrow left(p) := q$ **fi** and **if** $x \neq nil \longrightarrow x_{left} := left(x)$ $[\!]$ $x = nil \longrightarrow x_{left} := nil$ **fi** .

The Split operation is based on changing the father pointers of the smallest of the two resulting sets that are the result of the Split.

Figure 4.1: Procedure *build* in GSF(1).

```
(1) procedure build($z, \infty, \infty, 1$);
(2) create a set name $s$;
(3) for all $w \in linlist(z) \longrightarrow$ father($w$) := $s$ rof;
(4) left($s$) := right($s$) := $nil$
```

Figure 4.2: The Split procedure in GSF(1).

```
**procedure** SPLIT($(x',x),(y',y),1$);
$\{(Find(x) = Find(y) \wedge x < y) \vee y = nil\}$
{pre: left($x$) $\neq nil \vee y \neq nil$}
insert($x',x$) ; insert($y',y$);
$s := father(x)$;
splitlist($x, y, x_{conv}, x_{left}, y_{left}$):
countparts($x, x_{conv}, \infty, minweight, x_{min}, x_{max}$);
build($x_{min}, \infty, \infty, 1$) ;
```

Figure 4.3: Procedure *insert* in GSF(i) ($i \geq 1$).

```
**procedure** insert($x',x$);
{pre: $x = nil \Rightarrow x' = nil$}
**if** $x' \neq nil \longrightarrow$ left($x'$) := left($x$); right(left($x'$)) := $x'$;
                  right($x'$) := $x$; left($x$) := $x'$; father($x'$) := father($x$)
[] $x' = nil \longrightarrow$ skip
**fi**;
```

Figure 4.4: Procedure *splitlist* in GSF(i) ($i \geq 1$).

```
**procedure** splitlist($x, y$, **output** : $x_{conv}, x_{left}, y_{left}$);
$x_{left}$ := left($x$); $y_{left}$ := left($y$);
right($x_{left}$) := $y$; left($y$) := $x_{left}$;
right($y_{left}$) := $nil$; left($x$) := $nil$;
**if** $y \neq nil \longrightarrow x_{conv} := y$
[] $y = nil \longrightarrow x_{conv} := x_{left}$
**fi**;
```

Figure 4.5: Procedure *countparts*.

```
(1)  procedure countparts($x, x_{conv}, maxcount$, output : $minweight, x_{min}, x_{max}$);
(2)  {pre: $maxcount \geq 1$ }
(3)  start counting the number of nodes in the lists that contain $x$ and $x_{conv}$
(4)  as follows: "simultaneously" traverse both lists and alternatively
(5)  encounter a new element of each list;
(6)  stop as soon as either:
(7)  (*)a list has been counted completely; then
(8)      $minweight$:= the number of nodes counted in this list;
(9)      $x_{min}$:= an element of this list; $x_{max}$ := an element of the other list;
(10) (*)the lists have not been traversed completely yet, but
(11)     in both lists $maxcount$ nodes are counted;
(12)     then $minweight := maxcount + 1$ and $x_{min} := x$ and $x_{max} := x_{conv}$;
```

After having inserted the new elements x' and y' the procedure works as follows. If the ordered set that contains x must be split at x (and y), then first the linear list of elements (viz., $linlist(x)$) is split into two lists according to the specifications of the Split operation (cf. Section 4.2). That is, the two lists contain the elements of the ordered sets that result of splitting the set in the proper order. This is performed by procedure $splitlist(x, y, x_{conv}, x_{left}, y_{left})$ (cf. Figure 4.4). Moreover the procedure outputs a value x_{conv} that is an element of the resulting "converse" list, i.e, the resulting list that does not contain x, and it outputs the old predecessors x_{left} and y_{left} of x and y respectively. It is easily seen that this "converse" list is not empty, since either y or the predecessor of x exist (i.e., are not nil) (cf. line 3 of procedure SPLIT). (These old predecessors of x and y are not needed in GSF(1): these will be used in subsequent algorithms for $GSF(i)$ with $i > 1$.) After the splitting of the list into two new lists, the father names of the nodes in one of the lists have to be changed. To do this, first the smallest of the two lists is determined. This is done by means of a call of procedure $countparts(x, x_{conv}, maxcount, minweight, x_{min}, x_{max})$ (cf. Figure 4.5) that outputs the values $minweight$, x_{min} and x_{max} as follows. If both lists (i.e., the list containing x and that containing x_{conv}) have more then $maxcount$ elements, then $minweight$ has the value $maxcount + 1$, $x_{min} = x$ and $x_{max} = x_{conv}$. Otherwise, the (or: a) smallest list is determined, $minweight$ contains its size and x_{min} is an element of it, while x_{max} is an element of the other list. Henceforth, we denote a linear list resulting from a splitting in procedure *splitlist*, by means of a prime. Hence, we have $|linlist'(x_{min})| = minweight$. (In $GSF(1)$, we do not use the parameters *maxcount* and x_{max} of procedure *countparts*, but we will need them in $GSF(i)$ with $i > 1$.) It is easily seen that *countparts* satisfies the above specifications, and that it can be executed in time at most $c_{cp}.min\{|linlist'(x)|, |linlist'(x_{conv})|, maxcount\}$ and hence at most $c_{cp}.min\{|linlist'(x_{min})|, maxcount\}$ time (by using that $x \neq nil \neq x_{min}$) (where c_{cp} is some appropriate constant)

After having determined the smallest list, all elements of that list are put beneath a new set node s' by means of procedure *build* in line 8 (cf. Figure 4.1), which concludes the procedure.

4.3.2 The Split-Find Structure GSF(i) for $i > 1$.

Let $i > 1$. Structure GSF(i) is a structure that satisfies the following conditions. Recall that the ordered set corresponding to set name s is denoted by $set(s, i)$ and that nodes in $set(s, i)$ are in layer i.

Two cases are distinguished.

- If $set(s, i)$ contains more than one element, then $set(s, i)$ is partitioned into consecutive intervals (so-called *clusters*) of elements. For each cluster C there is a unique so-called cluster node c (not being an element in $set(s, i)$); all nodes in cluster C have node c as their father and $sons(c) = C$. The cluster nodes are (linearly) ordered by the order of the corresponding intervals of $set(s, i)$ (where for any two intervals A and B in the partition, we write $A \leq B$ if $x \leq y$ holds for all $x \in A$, $y \in B$). We denote the ordered set of these cluster nodes by $clusset(s, i)$.

  For each cluster node $c \in clusset(s, i)$, there is a parameter $highindex(c)$, that is an integer ≥ 1. For set name s, there exists a value $highindex(s, i)$ such that for all $c \in clusset(s, i)$

  $$highindex(c) = highindex(s, i) \tag{4.1}$$

  and such that

  $$16.A(i, highindex(s, i) + 1) \geq |set(s, i)|. \tag{4.2}$$

  Note that $highindex(s, i)$ needs not to be the lowest number that satisfies this inequality, and that the above restriction on $highindex$ is equivalent to

  $$highindex(s, i) + 1 > a(i, \lceil \frac{|set(s, i)| + 1}{16} \rceil). \tag{4.3}$$

  (The value $highindex(s, i)$ is fixed.)

  A cluster node $c \in clusset(s, i)$ satisfies

  $$|sons(c)| \leq 16.A(i, highindex(c)) \tag{4.4}$$

The subtree between s and $clusset(s, i)$ is a tree of a GSF($i-1$)-structure: the ordered nodes of $clusset(s, i)$ are the elements of the ordered set named s in a GSF($i-1$) structure. Thus:

$$set(s, i-1) = clusset(s, i).$$

- If $set(s, i)$ consists of precisely one element e, then either we have $father(e) = s$ and $sons(s) = \{e\}$, or we have the previous situation (i.e., there exists a cluster node).

If GSF(i) is used to solve the Split-Find problem, then the initialisation for some (sub-)collection of ordered sets is as follows (where the ordered sets are supposed to be given as a linear list *linlist* in which the elements appear in increasing order b.m.o. *left* and *right* pointers): for each ordered set, some element x is taken, and procedure $build(x, a(i, n_x), 16A(i, a(i, n_x)), i)$ is executed (which is given in Figure 4.6), where n_x is the number of nodes in the set containing x. Note that since $16A(i, a(i, n_x)) \geq n_x$, we have that *clussize* in this procedure is larger then the number of elements in $linlist(z)$ in the procedure: hence, if $n_x > 1$ then the tree that is built consists of the elements with exactly one cluster node above it (cf. line 5-6 and 8-9), while the cluster node has the set name as its father.

The splitting of a set can now be performed by the algorithm SPLIT$((x', x), (y', y), i)$ given in Figure 4.7. We do not consider the problem of how to obtain the Ackermann values and how to store the values *highindex* yet. The procedures that are called within procedure SPLIT are given in the figures 4.7, 4.4, 4.5, 4.6, 4.8 and 4.9.

Procedure SPLIT$((x', x), (y', y), i)$ operates in the following way. The procedure is based on splitting $linlist(x)$ into two lists and on counting the sizes of the lists just as for GSF(1), as far as these sizes do not exceed some size *maxcount*. Then, if the smallest resulting list allows cluster nodes with a lower value of *highindex*, then the father pointers of this list are changed to new cluster nodes, otherwise a recursive call is performed.

We describe the procedure in more detail. The algorithms work as follows after having inserted nodes x' and y'. First the linear list $linlist(x)$ is split into two lists according to the specifications of the Split operation. This is performed by procedure $splitlist(x, y, x_{conv}, x_{left}, y_{left})$ (cf. Figure 4.4), that outputs the values x_{conv}, x_{left} and y_{left} as described in Subsection 4.3.1. Henceforth, we denote these two resulting lists by $linlist'(x)$ and $linlist'(x_{conv})$. After the splitting of the list, the set name for the elements in one of the new lists has to be changed. To do this, the smallest of the two lists is determined if it has at most $16.A(i, h)$ elements. This is done by means of the call $countparts(x, x_{conv}, 16.A(i, h), minweight, x_{min}, x_{max})$ (cf. Figure 4.7) that outputs the values *minweight*, x_{min} and x_{max} as described in

Figure 4.6: Procedure *build* in GSF(i) ($i > 1$).

```
procedure build($z, h, clussize, i$);
{pre: $i > 1 \wedge clussize \geq 2$}
if $|linlist(z)| = 1$ ⟶create a set name $s$; father($z$) := $s$
[] $|linlist(z)| > 1$
  ⟶make an ordered list $D$ of intervals of $linlist(z)$ that all
    have size $clussize$ except for the last interval that may be smaller;
    do $D \neq \emptyset$
        ⟶ take the front interval $I$ in $D$ and remove it from $D$;
            create a cluster node $c$; highindex($c$) := $h$;
            for all $w \in I$ ⟶ father($w$) := $c$ rof;
            enqueue($c, clusqueue$)
    od;
    make a linear list of the nodes in $clusqueue$ by
    adapting the left and right pointers properly;
    $anext := a(i-1, |clusqueue|)$;
    build($c, anext, 16A(i-1, anext), i-1$)
fi;
```

Figure 4.7: The split procedure in GSF(i) ($i > 1$).

```
procedure SPLIT($(x', x), (y', y), i$);
{pre: $(\text{Find}(x) = \text{Find}(y) \wedge x < y) \vee y = nil$}
{pre: $\text{left}(x) \neq nil \wedge y \neq nil$}
insert($x', x$); insert($y', y$);
$C_x$ := father($x$); $C_y$ := father($y$); $h$ := highindex($C_x$);
splitlist($x, y, x_{conv}, x_{left}, y_{left}$);
countparts($x, x_{conv}, 16.A(i,h), minweight, x_{min}, x_{max}$);
if $minweight \leq 16.A(i,h)$
  ⟶$h_{new} := max\{h-1, 1\}$;
    modify($x, y, x_{min}, i$)
    build($x_{min}, h_{new}, 16A(i, h_{new}), i$)
[] $minweight = 16.A(i,h) + 1$
  ⟶splitcluster($x_{left}, x, C'_x$);
    splitcluster($y_{left}, y, C'_y$);
    SPLIT($(C'_x, C_x), (C'_y, C_y), i-1$);
fi;
```

Figure 4.8: Procedure *modify* in GSF(i) ($i > 1$).

```
procedure modify($x, y, x_{min}, i$);
{ pre: $i > 1 \wedge u \neq nil$ }
$C_x :=$ father($x$); $C_y :=$ father($y$);
$FN := \{\text{father}(z) | z \in linlist(x_{min})\}$ ;
for all $w \in linlist(x_{min}) \longrightarrow$father($w$) $:= nil$ for;
$FND := \{C \in FN | \text{nrsons}(C) = 0\}$ ;
if $|FND| > 0$
   $\longrightarrow C_{min} :=$ an element of $FND$;
        obtain $C'_x$ and $C'_y$ being the cluster nodes that define the splitting of
        $linlist(C_x)$ that has $FND$ as one of the resulting lists (with,
        hence, $C'_x = C_x$ or $C'_x =$ right($C_x$), and similar for $C_y$);
        splitlist($C'_x, C'_y, \cdot, \cdot, \cdot$);
        if $i - 1 > 1 \longrightarrow$modify($C'_x, C'_y, C_{min}, i - 1$) fi;
        dispose all nodes $C \in FND$
fi;
```

Figure 4.9: Procedure *splitcluster* in GSF(i) ($i > 1$).

```
procedure splitcluster($z_{left}, z$, output : $C'_z$);
if $z_{left} \neq nil \wedge$ father($z_{left}$) = father($z$)
                 $\longrightarrow$ create a cluster node $C'_z$
                     $w := z_{left}$; $C_z :=$ father($z$);
                     do $w \neq nil \wedge$ father($w$) $= C_z$
                          $\longrightarrow$ father($w$) $:= C'_z$; $w :=$ left($w$)
                     od;
                     highindex($C'_z$) := highindex($C_z$)
[] $z_{left} = nil \vee$ father($z_{left}$) $\neq$ father($z$)
                 $\longrightarrow C'_z := nil$
fi;
```

Section 4.3.1. Afterwards, two cases are distinguished, according to the critical size of $16.A(i,h)$ for the smallest of the two lists:

- The smallest list has at most $16.A(i,h)$ elements. Then the first alternative of the if-statement in line 8-11 is executed. Then all elements in the smallest list (that contains the element x_{min}) are deleted from the "old" tree, an entirely new tree is built for these elements, and the old tree is adapted according to the GSF(i) specifications. This is performed by procedures *modify* and *build* (given in figures 4.8 and 4.6).

  Procedure *modify* (given in Figure 4.8) that is called in line 10 of procedure SPLIT works as follows. First, it deletes the elements of $linlist'(x_{min})$ from the tree by setting their father pointers to *nil* (line 5); then it puts the cluster nodes that do not have any sons left in a list FND (lines 4 and 6) and it determines the proper split nodes for an artificial split, such that FND is one of the parts resulting from the split (line 9-11, this can be done in $O(1)$ time). In line 12, it splits the list of cluster nodes accordingly, and in line 13, the cluster nodes in FND are deleted from the tree by a recursive call. These cluster nodes are deleted in line 14. Therefore, in line 14 the tree with the remaining cluster nodes as leaves is a GSF($i-1$) tree (by an inductive argument; note that if $i-1=1$, then the recursive call need not to be applied). Therefore, line 15 yields that the remaining tree is a GSF(i) tree.

  Procedure *build* obviously creates a new tree that satisfies the conditions of GSF(i). In particular, it is easily seen that the equations (4.2), (4.1) and (4.4) are satisfied indeed.

- Both new lists contain more then $16.A(i,h)$ nodes. We now have the following situation.

  Observation 4.3.1 *In this case we have $father(x) \neq father(y)$ if $y \neq nil$.*

  This is seen as follows. If $y \neq nil$, then at the beginning of procedure SPLIT, $father(y)$ has at most $16A(i,h)$ sons. Moreover, in line 4 new elements are inserted as the direct predecessor of x and y only. Since there are at least $16A(i,h)+1$ elements $z \in linlist'(x)$ that therefore all satisfy $x \leq z < y$, it follows that x and y cannot have the same father.

  In this case the second alternative of the if-statement in line 12-15 is performed. The nodes in $linlist'(x_{conv})$ that have $father(x)$ as their father are "put" beneath a new cluster node C'_x. The same is done for $linlist'(x)$, $father(y)$ and C'_y. This is done in lines 13 and 14 by means of procedure *splitcluster* (given in Figure 4.9). Afterwards, we have a new splitting problem on $clusset(s, i-1)$ (where s is the name of the set that is being split), viz. the splitting of this set at the values C_x and C_y after having inserted the new nodes C'_x

and C'_y (as far as these are not nil). This is performed by the recursive call SPLIT$((C'_x, C_x), (C'_y, C_y), i-1)$. Note that the conditions "$Find(C_x) = Find(C_y) \wedge C_x < C_y) \vee C_y = nil$" and "$left(C_x) = nil \Rightarrow C_y \neq nil$" are satisfied at the moment of calling SPLIT$((C'_x, C_x), (C'_y, C_y), i-1)$, since w.r.t. the first condition, Observation 4.3.1 yields that if $y \neq nil$ then $C_x \neq C_y$ and hence $C_x < C_y$, and since w.r.t. the second condition, we have that if $C_y = nil$ then $y = nil$, and since at least $16A(i,h)+1$ elements are smaller than x, C_x must contain a predecessor. Finally, it is easily seen that after SPLIT$((C'_x, C_x), (C'_y, C_y), i-1)$ has been performed, that the trees have been split according to the values x and y.

4.3.3 Representations

We descibe how to represent and how to obtain the information that is used in the previous subsection. Again we describe the representation inductively. Consider a GSF(i) structure for some $i \geq 1$. Suppose that the maximal number of elements that is in any initial set is n_0. Moreover, suppose that an Ackermann net for some n_{ack} with $n_{ack} \geq n_0$ is present.

Each node has, besides the fields described above, a pointer field *ack* and a field *highindex*. If the value $highindex(x)$ is defined for node x, then its field *highindex* contains that value $highindex(x)$, otherwise the field is not defined. If node x is in layer i of the GSF(i) structure, then field *ack* of x contains a pointer into the Ackermann net that points to node $(min\{i, \alpha_{ack}+1\}, -1)$.

During an execution of the procedures, the value $A'(i,l)$ is substituted for $A(i,l)$. This does not affect the algorithms, which is seen as follows.

Observation 4.3.2 *At any time, $highindex(c) \leq a(i, n_0)$ holds for any cluster node in GSF(i).*

This observation holds indeed, since initially all *highindex* values are at most $a(i, n_0)$, and since new *highindex* values are not larger than the old ones.

Like in Chapter 3, the value $A(i,h)$ can be substituted by $A'(i,h)$ in the procedures (where in the initial call $build(z, a(i,n_0), A(i, a(i,n_0)), i)$, the value $A'(i, a(i,n_0))$ can be used instead of the A-value, for similar reasons). We assume that when a GSF(i) structure is initialised, all elements contain a pointer to node $(min\{i, \alpha_{ack}+1\}, -1)$ of the Ackermann net.

Note that at the moment that $build(z, h, 16A(i,h), i)$ or $splitcluster(z_{left}, z, C'_z)$ is called, then z is an element (in layer i) that (therefore) contains a pointer to node $(min\{i, \alpha_{ack}+1\}, -1)$ of the net. Hence, a pointer to node $(min\{i, \alpha_{ack}+1\}, -1)$ of the net can be obtained in $O(1)$ time. (In fact in procedure *splitcluster* this pointer can be obtained in $O(1)$ time from C_z too.) Therefore during the creation of a node in these procedures, the initialisation of the *ack* field takes $O(1)$ time only, which we will henceforth consider to be part of the $O(1)$ time of creating such a node.

Now, since there are at most n_0 GSF(i)-elements in any set in GSF(i) (cf. Section 4.4), there are at most n_0 cluster nodes in any tree of GSF(i) and hence an Ackermann net for n_{ack} suffices for GSF($i-1$) too.

Ackermann values are used in procedures SPLIT and *build* only. Note that now all Ackermann values $A(i,k)$ that are required in the procedure SPLIT, can be obtained from node $(min\{i, \alpha_{ack}+1\}, -1)$ of the Ackermann net in $O(k)$ time. Moreover, the value $a(i-1, |clusqueue|)$ for the recursive call in procedure *build* can be obtained in $O(a(i-1,|clusqueue|))$ time. Finally, the value $a(i-1, n_x)$ needed in the call *build* for the initialisation can be obtained from node x and its pointer to node $(min\{i,\alpha_{ack}+1\},-1)$ of the Ackermann net in $O(a(i-1,n_x))$ time. (Remark that in a node c with $highindex(c) = k$, a pointer to Ackermann node $(min\{i,\alpha_{ack}+1\},k)$ can be stored too. This decreases the $O(k)$ time for the first of the above cases to $O(1)$ time.)

4.4 Complexity of GSF(i).

In the sequel we will denote the procedures $build(z,l,clussize,i)$, $modify(x,y,x_{min},i)$ and SPLIT$((x',x),(y',y),i)$ by $build_i$, $modify_i$, and SPLIT$_i$, respectively, to make only parameter i explicit.

We consider the cost of a sequence of procedure executions in GSF(i). Note that a FIND operation does not affect the GSF(i) structure and that an execution takes $O(i)$ time. Therefore we do not need to consider the complexity of this operation any further.

Consider a call $modify_i$. There are two cases in which $modify_i$ can be called, viz., within procedure SPLIT$_i$ or within procedure call $modify_{i+1}$ if the GSF(i) structure is part of a GSF($i+1$) structure. The call of $modify_i$ performed in SPLIT$_i$ is denoted as a dependent call (w.r.t. GSF(i)), whereas the other calls are denoted as independent.

Similarly, a dependent call of procedure $build_i$ (w.r.t. GSF(i)) is a call performed in SPLIT$_i$, and an independent call is a call performed otherwise.

Observation 4.4.1 *Procedure $build(z,l,clussize,i)$ (given in Figure 4.1 and 4.6) can be executed in at most $d.|linlist(z)|$ time, where d is some constant.*

Proof. For $i=1$, the assertion obviously holds for some constant d_1. Now suppose procedure $build(z,l,clussize,i-1)$ works in at most $d.|linlist(z)|$ time for some constant d. Then it is easily seen that the execution of the procedure takes at most $d'.|linlist(z)|$ time apart from the recursive call. For, note that the inverse Ackermann value $a(i-1, |clusqueue|)$ that is computed in line 15 of the procedure, can be obtained in $O(|clusqueue|) = O(|linlist'(z)|)$ time from the Ackermann net by using the pointer to node $(min\{i, \alpha_{ack}+1\}, -1)$ of the net; a similar observation holds for the Ackermann value $A(i-1, anext)$ computed in line 16. Consider the case in which the recursive call is performed. Then $clussize \geq 2$ and hence it follows that the number of cluster nodes that is created is at most $\frac{3}{4}.|linlist(z)|$. Hence, the recursive call takes at most $\frac{3}{4}.d.|linlist(z)|$ time. By taking $d = max\{4d', d_1\}$ the assertion follows. □

Observation 4.4.2 *Procedure $modify(x, y, x_{min}, i)$ (given in Figure 4.8) can be executed in at most $d'.(|linlist(x_{min})| + dispnod)$ time, where dispnod is the number of nodes disposed by the procedure and where d' is some constant.*

Proof. This easily follows by induction. □

We charge the cost d' for a node that is disposed to its creation: this does not affect the order of the time complexity of executions of the procedures together with the initialisation (b.m.o. *build*), and this yields that we have a net cost of $d'.|linlist(x_{min})|$ for a dependent call $modify_i$ and a net cost of 0 for an independent call.

Observation 4.4.3 *A call of the procedure $modify(x, y, x_{min}, i)$ has a net cost of $d'.|linlist(x_{min})|$ if it is dependent, and a net cost 0 if it is independent.*

We consider a sequence of independent calls of $build_i$, $SPLIT_i$ and $modify_i$ operations in which precisely one independent $build_i$ operation occurs, viz., as the first operation. By Observation 4.4.3, it suffices to consider independent calls of $build_i$ and $SPLIT_i$ only. We call these operations the independent operations on GSF(i).

Note that all the above operations only affect the (tree of the) set that has to be split and that other sets (trees) are left unchanged. Therefore we only need to consider the case in which initially there is precisely one set.

Let n_0 ($n_0 > 1$) be the initial number of elements that is present, and let n be the total number of elements that ever exist (recall that $SPLIT$ may insert two new elements). We assume that an Ackermann net for $n_{ack} \geq n_0$ is present, and that each element contains a pointer to node $(min\{i, \alpha_{ack}+1\}, -1)$ of the net, in its field *ack*. We assume that for the initialisation b.m.o. an independent call $build_i$ a linear list *linlist* is given, containing the elements in the proper order. By Observation 4.4.1, procedure $build_i$ can be executed in $O(n_0)$ time, if n_0 is the number of elements in the list. We show that a sequence of Splits in GSF(i) take $O(n.a(i, n_0))$ time altogether.

4.4.1 Complexity of GSF(1)

We prove that GSF(1) requires $O(n.\log n_0)$ time for all independent operations together. Obviously, the number of splits that is performed is at most $n-1$.

Consider a single execution of procedure $SPLIT((x',x),(y',y),1)$. Since the split is nontrivial, we must have $|linlist'(x_{min})| \geq 1$ afterwards. Moreover, note that despite the possible insertion of two new elements in the ordered set, the resulting sets have size not exceeding the size of the original set. (For, a node x' (y') that is inserted, will not be part of the resulting set in which x (y') is contained.) Obviously, the execution of procedure *splitlist* takes $O(1)$ time, and the execution of *countparts* takes $c_{cp}.|linlist'(x_{min})|$ time (cf. Subsection 4.3.1). Since the building of a new tree for $linlist'(x_{min})$ in line 8 can be performed in $d.|linlist'(x_{min})|$ time (cf. Observation 4.4.1), it follows that the entire execution of procedure SPLIT takes at most $c_{SPLIT}.|linlist'(x_{min})|$ time (for suitable c_{SPLIT}). We distinguish two cases.

- $|linlist'(x_{min})| \leq 2$. Then procedure SPLIT takes at most $2c_{SPLIT}$ time. Obviously, all executions of procedure SPLIT that satisfy $|linlist'(x_{min})| \leq 2$ take at most $2.c_{SPLIT}.n$ time together.

- $|linlist'(x_{min})| \geq 3$. Now charge the cost of such a split to the nodes in $linlist'(x_{min})$ by charging at most c_{SPLIT} time to each node. Then (if $linlist(x)$ is the ordered set before the execution of the considered SPLIT operation) the following holds:

  $$|linlist(x)| + 2 \geq 2.|linlist'(x_{min})|$$

  since $linlist'(x_{min})$ is the smallest of the two lists resulting from the split and since before the splitting of the list at most two new nodes are inserted in it. Therefore, every node that is in $linlist'(x_{min})$ has become element of a new set that is at most $\frac{3}{4}$ times the size of the old set in which it was contained before the split. Since initially there are sets that are of size at most n_0 and since the independent operations do not yield larger sets, this can happen at most $4.\lceil\log n_0\rceil$ times for a node. Hence, each node is charged to for at most $4.c_{SPLIT}.\lceil\log n_0\rceil$ time. Hence, all such executions of procedure SPLIT for a collection of n nodes take at most $4.c_{SPLIT}.n.\lceil\log n_0\rceil$ time ($= 4.c_{SPLIT}.n.a(1,n_0)$ time).

By the above case analysis, it follows that all Splits take at most $c_0.n.\log n_0$ time together. Moreover, it is easily seen that at any time the GSF(1) structure requires at most $2n$ nodes.

4.4.2 Complexity of GSF(i) for $i > 1$

We consider the complexity of all Splits in GSF(i) with $i > 1$. We perform the analysis by means of induction to i.

Suppose GSF($i-1$) takes at most $c.m.a(i-1, m_0)$ time for all Splits starting from a set with initial size $m_0 > 1$ and with a total collection of m elements. We consider the cost of all Splits for a total collection of n elements by means of GSF(i), where n_0 is the size of the initial set of elements.

Consider procedure SPLIT($(x', x), (y', y), i$). We divide this procedure into several parts: some procedure calls within SPLIT($(x', x), (y', y), i$) and the remainder. For a procedure call we consider the net cost of the call, i.e., the cost that is not charged to another procedure.

1. The (dependent) calls $modify_i$ and $build_i$ in SPLIT$_i$ (line 10-11)
2. The call *splitcluster* in SPLIT$_i$ (line 13-14)
3. The call *countparts* in SPLIT$_i$ (line 7)
4. The recursive call SPLIT$_{i-1}$ in SPLIT$_i$ (line 15).
5. The rest of procedure SPLIT$_i$

We compute the cost of each part for all executions of SPLIT$_i$ together. However, we first consider procedure SPLIT$_i$ more carefully.

Observation 4.4.4 *Let $1 \leq h_0 \leq a(i, n_0)$. Then at most $\frac{n}{16A(i,h_0)}$ calls of SPLIT$_i$ have $h = h_0$ and $minweight \geq 16A(i,h) + 1$ in line 7 and line 12.*

Proof. Let h_0 be as above. Consider a call SPLIT($(x', x), (y', y), i$) that yields $h = h_0$ and $minweight > 16A(i, h)$ at line 7 and 12. Then apparently $|linlist'(x)| \geq 16A(i, h) + 1$ and $|linlist'(x_{conv})| \geq 16A(i, h) + 1$.

The sets resulting from a Split operation are not larger than the original set on which the operation is performed: for, a node x' (or y') that is inserted, is inserted as the predecessor of x (y) and hence it will not be part of the resulting set that contains x (y). Obviously procedure $modify_i$ does not extend a set either. Since the total collection of elements is n, since the only operations that change set sizes are $SPLIT$ and $modify$, and since both the sets resulting form such a call have size at least $16A(i, h) + 1$ this implies that there can be at most $\frac{n}{16A(i,h)}$ such calls. □

1. The Calls $modify_i$ and $build_i$ in SPLIT$_i$

Consider the two consecutive calls of procedures $modify_i$ and $build_i$ in procedure $SPLIT_i$. By observations 4.4.1 and 4.4.3, it follows that the time for the execution of the calls is bounded by $d.|linlist'(x_{min})| = d.$*(the number of processed elements)* for some constant d. We distinguish two cases.

- $h = 1$: then there are at most 32 elements that are processed and hence the cost of the calls $modify_i$ and $build_i$ in this case is $O(1)$. We will therefore charge this $O(1)$ cost to procedure SPLIT$_i$ itself (hence, it will be accounted for in part 5).
- $h > 1$: we charge the cost of the calls $modify_i$ and $build_i$ to the processed elements. Note that the processed elements will have new fathers that have a lower *highindex* value than the old fathers. Moreover, an element never has a new father with a higher *highindex* value. Therefore, the number of times that an element can be charged to is bounded by the number of different *highindex* values, which is at most $a(i, n_0)$ by using Observation 4.3.2. Hence, the total cost of all the calls of $modify_i$ and $build_i$ for $h > 1$ is at most $c_1.n.a(i, n_0)$ for some constant c_1.

Hence, the total net cost of all calls $modify_i$ and $build_i$ is at most $c_1.n.a(i, n_0)$.

2. The Call *splitcluster* in SPLIT$_i$

Consider a value of h, $1 \leq h \leq a(i, n_0)$. Procedure *splitcluster* is called in procedure SPLIT$_i$ only if $minweight = 16A(i, h) + 1$. Hence, by Observation 4.4.4, procedure *splitcluster* is called at most $\frac{2n}{16A(i,h)}$ times. The calls $splitcluster(x_{left}, x, C'_x)$ and $splitcluster(y_{left}, y, C'_y)$ can obviously be executed in at most $c'_2.A(i, h)$ time together, since C_x and C_y have at most $16A(i, h) + 2$ sons. (Or even $16A(i, h) + 1$ sons by using Observation 4.3.1.)

This yields that the cost of all these calls of procedure *splitcluster* for fixed value h is at most $c_2.n$ for some constant c_2.

Since $1 \leq h \leq a(i, n_0)$, the total cost of all procedure calls *splitcluster* is at most $c_2.n.a(i, n_0)$.

3. The Call *countparts* in SPLIT$_i$

In subsection 4.3.1, it was seen that the call

$$countparts(x, x_{conv}, 16A(i, h), minweight, x_{min})$$

takes at most $c_{cp}.min\{|linlist'(x_{min})|, 16A(i,h)\}$ time. We consider two situations for fixed value of h, $1 \leq h \leq a(i,n)$.

- $minweight \leq 16A(i,h)$. Then the cost of the procedure call is $c_{cp}.|linlist'(x_{min})|$. We charge this cost to the execution of procedure $build_i$, that is executed in this case too: this increases the cost of that procedure with an additional factor only.
- $minweight = 16A(i,h)+1$. Then the cost of the procedure call is $c_{cp}.16A(i,h)$. We charge this cost to the executions of procedure *splitcluster*, that is executed in this case too: this increases the cost of that procedure with an additional factor only.

Therefore the net cost of procedure *countparts* is 0.

4. The Recursive Call SPLIT$_{i-1}$ in SPLIT$_i$

The recursive calls SPLIT$_{i-1}$ are performed on cluster nodes. Therefore, we first consider cluster nodes and the conditions for the execution of a recursive call SPLIT$_{i-1}$. Note that $SPLIT_{i-1}$ is called only if $minweight \geq 16A(i,h) + 1$ and hence there are at least two cluster nodes in the tree that is operated on.

Observation 4.4.5 *The operations on cluster nodes are:*

1. *the creation of a singleton set of exactly one new cluster node by procedure $build_i$ (called in line 11 of $SPLIT_i$), where the cluster node has the set name as its father: such a node is called a trivial cluster node*
2. *the creation of a complete set of at least two new cluster nodes by procedure $build_i$ (called in line 11 of $SPLIT_i$): these nodes are called initial cluster nodes*
3. *the Splitting of a set of at least two cluster nodes by $SPLIT_{i-1}$ (called in line 15 of $SPLIT_i$)*
4. *the creation of a new cluster node by procedure splitcluster (called in line 13-14 of $SPLIT_i$): such a node is called an incremental cluster node*
5. *the disposal of a cluster node by procedure $modify_i$*

By Observation 4.4.5, it follows that for $SPLIT_{i-1}$ we only have to consider non-trivial cluster nodes.

It is easily seen that for any cluster node c the value $highindex(c)$ is fixed. We call a cluster node c with $highindex(c) = h$ an h-cluster node. Similarly, we say that a recursive call SPLIT$((C'_x, C_x), (C'_y, C_y), i-1)$ is an h-call or an h-Split if $h = highindex(C_x)$ $(= highindex(C_y))$. We compute the cost of all h-calls for fixed value h.

Let h be a fixed number satisfying $1 \leq h \leq a(i,n)$. We consider the cost of all recursive h-calls SPLIT$((C'_x, C_x), (C'_y, C_y), i-1)$.

Consider the operations on cluster nodes starting from some *build*$_i$ operation that creates a set S of at least 2 h-cluster nodes. Then the size k_0 of such an initial set S is at most $16A(i, h+1)$. For such a set S of k_0 cluster nodes, the cost of all $SPLIT_{i-1}$ operations on these nodes in GSF$(i-1)$ is at most $c.k.a(i-1, k_0) \leq$ $c.k.\ a(i-1, 16A(i, h+1))$, where k is the total number of cluster nodes that are in set S or that are created during the considered operations.

Hence, the total cost of all calls $SPLIT_{i-1}$ in GSF$(i-1)$ on all these h-cluster nodes is at most

$$c.(\textit{total number of nontrivial } h\textit{-cluster nodes}).\ a(i-1, 16A(i, h+1)).$$

Note that an element can have only one initial h-cluster node as its father. For, if the father of an element changes, then the new father is either an incremental h-cluster node, a trivial h-cluster node, or a $h-1$-cluster node. Moreover, note that if $n > 1$ then there are at most $2.\frac{n}{16A(i,h)}$ initial h-cluster nodes, since initial h-custer nodes are created in a set of a least two cluster nodes where each such cluster node has at least $16A(i,h)$ sons, except for possibly the last one (cf. line 5-7 and 8-9 of procedure *build*). By Observation 4.4.4 it follows that there are at most $\frac{n}{16A(i,h)}$ calls of SPLIT$_i$ in which the condition in line 12 is true and hence there are at most $\frac{2n}{16A(i,h)}$ incremental nodes. Hence, there are at most $4n/(16A(i,h))$ nontrivial h-cluster nodes. Therefore, the total cost for all h-Splits is at most

$$\begin{aligned}
& c.\frac{n}{4A(i,h)}.\ a(i-1, 16A(i, h+1)) \\
\leq\ & \frac{1}{4}c.\frac{n}{A(i,h)}.\ (a(i-1, A(i-1, A(i,h))) + 4) \\
=\ & \frac{1}{4}c.n + \frac{1}{4}c.n.\frac{4}{A(i,h)} \\
\leq\ & \frac{3}{4}.c.n
\end{aligned}$$

by using $i > 1$, Equation (2.1), Lemma 2.3.4 and $h \geq 1$ respectively.

Since there are $a(i, n_0)$ applicable values h of $highindex$ to be considered, this yields that the total time complexity of all operations $SPLIT_{i-1}$ in GSF$(i-1)$ is at most $\frac{3}{4}c.n.a(i.n_0)$.

5. The Rest of Procedure SPLIT$_i$

The execution of all statements together except those considered in the previous parts require at most $c'_4.(1+h)$ time per call of $SPLIT_i$, where h is the *highindex* value during the call. (For, the computation of a value $A(i,h)$ takes $O(h)$ time.) Since there are at most n Splits and since $highindex(c) \leq a(i,n_0)$, this takes at most $c_5.n.a(i,n_0)$ time altogether.

The Total Complexity.

Combining the parts yields that the total time is at most

$$(c_1 + c_2 + \frac{3}{4}c + c_5).n.a(i,n_0)$$

which is at most $c.n.a(i,n_0)$ if $c = max\{c_0, 4.(c_1+c_2+c_5)\}$. Hence, GSF($i$) takes at most $c.n.a(i,n_0)$ time for all Splits together.

We now consider the space complexity of GSF(i) structures for $i > 1$. Suppose that any GSF($i-1$) structure for m elements has at most $4m$ nodes. Now consider a GSF(i) structure for n elements. We prove that the space complexity is at most $4n$. Consider the total number of cluster nodes that ever exist. In part 4, we have already seen that the number of nontrivial h-cluster nodes for some h is at most $\frac{n}{4A(i,h)}$. Therefore the total number of nontrivial cluster nodes is at most

$$\sum_{h=1}^{a(i,n)} \frac{n}{4A(i,h)} \leq \frac{1}{4}n$$

Since nontrivial cluster nodes are the elements of a GSF($i-1$) structure this yields that there are at most $\frac{1}{4}.4.n$ nodes in the GSF($i-1$) structures for these cluster nodes. At any time the number of trivial cluster nodes is at most n. Since each trivial cluster node has only the set name as its father, the trivial cluster nodes give rise to an extra amount of n nodes except for the set names. Finally, elements may have not have a cluster node as their father, but the set name instead. Hence, all elements together with their set names are at most $2n$ nodes. Hence, we have at most $4n$ nodes in the structure.

Since GSF(1) has at most $2n$ nodes, it follows by induction that GSF(i) has at most $4n$ nodes for any i.

4.4.3 Total Complexity of GSF(i) for $i \geq 1$

Note that an Extension Split on a set S may have two sets S_1 and S_2 as result with $|S| = |S_1|$, viz., if S_2 is $\{x\}$ or $\{x, y'\}$ and if $x' \neq nil$ (or with x and y reversed and $y \neq nil$). In that case, the only changes on the GSF(i) tree for set S is the replacement of x by x'. This takes $O(1)$ time. If we separately consider these Splits, together with Splits on sets of size at most 2, then we are left with splits on sets of size at least 3 that yield two sets of smaller size. We call the latter splits *essential splits*. By induction, it is easily verified, that there are at most $2n_0 - 5$ essential Splits. Hence, if we consider essential splits in GSF(i) only, then we have at most $2n_0 - 5$ splits and at most $3n_0$ elements in GSF(i). (All other splits take $O(1)$ time and can be performed "outside the scope" of GSF(i).)

By means of induction we have established the following result.

Lemma 4.4.6 *The total time that is needed for all regular Splits in GSF(i) on a universe of n_0 elements is $O(n_0.a(i, n_0))$ ($i \geq 1$, $n_0 \geq 2$). The total time that is needed for all Extension Splits in GSF(i) is $O(n + n_0.a(i, n_0))$, where n is the resulting number of elements.*

We use an Ackermann net for $n_{ack} = n_0$. By Lemma 2.4.3, the Ackermann net can be computed in $O(\log n_0)$ time and takes $O(\log n_0)$ space (together with a pointer to node $(min\{i, \alpha_{ack} + 1\}, -1)$ of the net). (Note that when GSF(i) is used, in this way all elements can be provided with a pointer to node $(min\{i, \alpha_{ack} + 1\}, -1)$ in $O(n_0)$ time. This pointer initialisation preceeds the initialisation described in Subsection 4.3.2 (viz., the execution of procedure *build*).)

We have established the following theorem.

Theorem 4.4.7 *For every $i > 0$, the GSF(i) structure is a data structure with algorithms that can be implemented as a pointer/$\log n$ solution and that solves the Generalized Split-Find problem. The total time that is needed for all regular Split operations in a GSF(i) structure for a universe with n_0 elements is $O(n_0.a(i, n_0))$ and the time needed for a Find operation is $O(i)$, whereas the initialisation can be performed in $O(n_0)$ time and the entire structure takes $O(n_0)$ space ($i \geq 1$, $n_0 \geq 2$). If Extension Splits are executed, then the total time that is needed for all Extension Splits is $O(n+n_0.a(i, n_0))$ and the structure takes $O(n)$ space, where n is the resulting number of elements.*

4.5 Structures That Are Optimal on Pointer Machines

By applying GSF(i) structures for appropriate values of i, we obtain a structure that is optimal for pointer machines. This is expressed in the following theorems.

Theorem 4.5.1 *There exists a data structure and algorithms that solve the Generalized Split-Find problem with the following properties: the total time needed for all regular Splits and m Finds is $O(n_0+m.\alpha(n_0,n_0))$, while each Find takes $O(\alpha(n_0,n_0))$ time, and where n_0 is the initial number of elements ($n_0 \geq 2$). The data structure uses $O(n_0)$ space and can be initialised in $O(n_0)$ time. The data structure and algorithms can be implemented as a pointer/ $\log n$ solution. If Extension Splits are executed, then the total time needed for all Splits and m Finds is $O(n+m.\alpha(n_0,n_0))$ and the space complexity is $O(n)$, where n is the resulting number of elements.*

Proof. Like Theorem 3.5.1. □

Theorem 4.5.2 *There exists a data structure and algorithms that solve the Generalized Split-Find problem with the following properties: the total time needed for all regular Splits and m Finds is $O(n_0+m.\alpha(m,n_0))$, while the f^{th} Find takes $O(\alpha(f,n_0))$ time, and where n_0 is the initial number of elements ($n_0 \geq 2$). The data structure uses $O(n_0)$ space and can be initialised in $O(n_0)$ time. The data structure and algorithms can be implemented as a pointer/ $\log n$ solution. If Extension Splits are executed, then the the total time needed for all Splits and m Finds is $O(n+m.\alpha(m,n_0))$ and the space complexity is $O(n)$, where n is the resulting number of elements.*

Proof. We make use of GSF(i) structures. All the set names are contained in a list. (This implies that when a new set name is created, it is inserted in the list.) The transformation of structures is performed similar as for Theorem 3.5.2, where we have the following remarks. If Extension Splits are considered, then only the set names for sets of size ≥ 3 are in the list, the transformations are only applied on the elements of sets of size at least 3, and n_0 (and not n) is considered in the transformation condition. The procedure $build_{i-1}$ is used to initialise a new structure GSF($i-1$): it is performed for each set that is present, with the same arguments as in case of the initialisation, where a slight adaptation is performed: the creation of a set name in line 3 is omitted, but the actual set name s of the set that is considered is taken. □

4.6 Extensions

4.6.1 The Multiple Split

In this section we consider the multiple Split operation. Let k be some fixed even number, $k \geq 2$. Then the k-multiple Split operation Split$_k(t)$ is defined as follows: $t = (e_i)_{1\leq i\leq k}$ is a sequence of l ($l \leq k$) distinct elements concatenated with $k-l$ values ∞, where all elements are in the same set S, and where the elements e_i occur in order in the sequence. The set S has to be split into two subsets S_1 and S_2, defined as follows:

$$S_1 = \{x \in S | \underset{i}{\exists} [0 \leq 2i \leq k-1 \wedge e_{2i+1} \leq x < e_{2i+2}]\}$$

and S_2 is the remainder, where e_{k+1} is taken to be ∞ .

Obviously, this is a generalisation of the Split operation we have considered before. The structures and algorithms we have presented can be adapted to be used for this generalization. This is performed as follows. We denote the new structures by $\text{GSF}_k(i)$.

As for $k = 2$, we consider an extended Split operation $\text{SPLIT}_k(t', t)$ where t' is the sequence of new elements that are to be inserted and where t may contain one of these new elements too: for each existing element e_i in $t = (e_i)_i$ the corresponding e'_i of $t' = (e'_i)_i$ is either a new element that has to be inserted as the predecessor of e_i, or it contains the value nil; for each e_i with $e_i = \infty$ we have $e'_i = nil$; finally, for a new element e_i, we have $e'_i = nil$ and $e'_{i+1} = e_i$.

Now the adaptations in the procedures for $\text{GSF}_k(1)$ are as follows. Firstly, procedure *splitlist* is adapted in the obvious way to obtain the two new lists of elements. This changes the time complexity by an additional term k. The remainder of the procedures for GSF(1) is not changed.

The total complexity analysis does not change apart from the constants: obviously set sizes never increase because of a Split. We consider splits that yield a smallest set of size at most $2k$ and splits yielding sets of size more than $2k$: in the latter situation the smallest of the resulting lists has size that is at most $\frac{3}{4}$ of the size of the original set. In this way we obtain the same complexity (with a larger constant, of course).

The adaptations in the procedures for $\text{GSF}_k(i)$ with $i > 1$ are as follows (apart from some obvious adaptations). In procedure SPLIT the two calls of *splitcluster* are replaced by other calls in the following way: the k elements are divided in consecutive parts such that each part has the same father. Then, for each part all sons of the father are split into two subparts, where one subpart has a newly created father f' and the other subpart has the original father f. The subpart that contains the last split element has the old father f as its father, the remainder has f' as its father.

The recursive call is now performed on these fathers, where the new fathers are the new cluster nodes. Moreover, if an old father f had an even number of sons that were split elements, then the new father f' occurs as a predecessor of f in the split sequence t of SPLIT_{i-1} too. Finally, the constants 16 that appear in equations 4.2, 4.3 and 4.4 and in the procedures are replaced by the constant $16k^2$.

Now the complexity analysis is as follows. Firstly, the factor 16 in Observation 4.4.4 is replaced by $16k^2$. Obviously, the analysis of procedures *build* and *modify* does not change. On the other hand procedure *splitcluster* is called at most k times per SPLIT-call. However, by Observation 4.4.4 this still yields the same order of complexity. Moreover, the charging of the cost of procedure *countparts* can be done similarly. The cost of the recursive call is similar, apart from replacing 16 by $16k^2$ in the analysis and apart from the fact that there are at most k incremental nodes created per SPLIT$_i$ operation. Since all occurrences of constant 16 are replaced by $16k^2$ this yields that the total number of incremental nodes is at most $\frac{n}{16k.A(i,h)}$. Then the total cost can be bounded by the same result. Finally, the cost of the rest of the procedure obvioulsy is $O(k+h)$. This yields a time bound of $c_k.n.a(i,n_0)$ for GSF$_k(i)$, where c_k is a constant that is dependent of k. Hence, the theorems stated in the previous sections also hold for the k-Split problem for some fixed k.

4.6.2 Additional Operations

For the Split operation, the conditions that if $y \neq nil$, then x and y must be in the same set and $x < y$, must be satisfied to yield a (nontrivial) split. Therefore, we have the operation ORDER(x,y) that outputs true if $x \leq y$ and nil otherwise. (Note that by means of this operation we can easily obtain the operation SPLIT(min(x,y),max(x,y)).) Moreover, we also want to be able to perform the operations PREV(x) and NEXT(x) for each element x that return its predecessor and its successor in its set if these exist, and nil otherwise: these operations enable us to test beforehand whether a split operation is useful or not. (For, if x is the first element of its set and $y = nil$, then a split operation does not change anything.)

The operations ORDER, PREV and NEXT can easily be implemented as follows. During the initialisation of a set, the elements in a set are number consecutively starting from one. Moreover, in case Extension Splits are performed, a new node x' that is inserted as the predecessor of x gets the same number as x. (This suffices, since after the Split, these elements will be in different sets). Obviously, in this way the ORDER between two elements in a set can easily be obtained in $O(1)$ time at any moment during the Split-Find problem. On the other hand, the operations PREV and NEXT can easily be performed by means of the pointers $left$ and $right$ in each element, that point to the predecessor and the successor element in that set.

Suppose we want that at any time the largest element of a set is the output of a Find operation on an element of that set. Then this can be maintained as follows for GSF(i). Below we will denote by the name of the set the root of the tree in GSF(i), like before. At any time, each node in the tree has a pointer to its its rightmost son (i.e., according to the order of the sons). When a Split is performed, these pointers can easily be maintained within the same order of complexity. Now, during a Find the largest element of a set can be obtained from the set name in $O(i)$ time by following these pointers starting from the set name. Therefore, all previous time bounds for the GSF(i) structures remain valid.

4.7 Increasing the Number of Elements

For application in Chapter 8, we now consider structures that, besides the operations Split and Find, allow the creation of a new set in the universe. In this way the collection of elements can be augmented.

Theorem 4.7.1 *The GSF(i) structure allows creations of new sets. The total time that is needed for all Split operations in a GSF(i)-structure until a moment on which there are* n *elements is* $O(n.a(i,n))$*, while the time needed for a Find operation is* $O(i)$*, and the entire structure takes* $O(n)$ *space* ($i \geq 1$, $n \geq 2$)*. The initialisation can be performed in* $O(n_{init})$ *time, where* n_{init} *is the initial number of elements.*

Proof. The creation of a new set S in a structure GSF(i) requires a call of procedure *build* on the elements of S, together with the augmentation of the Ackermann net that is used. We can do this in a way similar to the proof of Theorem 3.6.1. □

Theorem 4.7.2 *There exists a structure that solves the Generalized Split-Find Problem and that allows creations of new sets, such that the following holds. The total time is* $O(n + m.\alpha(m,n))$*, where* n *is the total number of elements and* m *is the number of Finds. Moreover, the* f^{th} *Find is performed in* $O(\alpha(f,n_f))$ *time, where* n_f *is the number of elements at the time of the* f^{th} *Find. The structure can be implemented as a pointer/*$\log n$ *solution.*

Proof. The strategy is similar to that in the proof of Theorem 3.6.2. □

4.8 Concluding Remarks

In this chapter, we have presented a collection of structures for the Generalized Split-Find problem, that can be implemented on pointer machines, including structures that have time complexity that is optimal for pointer machines. All arithmetic occurring in the algorithms can be performed by using additions, subtractions and comparisons only. Like in Chapter 3, in practice there is no need to perform transformations of structures like those occurring in Section 4.5: structures $SF(2)$ and $SF(3)$ are suited for all practical situations.

Chapter 5

Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines

5.1 Introduction

As we mentioned in the previous chapters, there are several Union-Find algorithms that run on pointer machines in $O(n + m.\alpha(m, n))$ time for $n - 1$ Unions and m Finds, and that use a form of path compaction [31, 33]. In Chapter 3, we presented a new algorithm without path compaction that runs on a pointer machine and that has a worst-case time bound of $O(\alpha(f, n))$ for the f^{th} Find, within the bound of $O(n + m.\alpha(m, n))$ time for $n - 1$ Unions and m Finds on universes of n elements as a whole. In this chapter, we consider the problem of obtaining lower bounds for the Union-Find problem on pointer machines. We also consider lower bounds for the Split-Find problem on pointer machines.

In 1979, Tarjan [32] proved a lower bound on the time complexity of Union-Find programs on a pointer machine that satisfy the *separation condition* (defined in detail below): such programs of $n-1$ Unions and m Finds take at least $\Omega(m.\alpha(m, n))$ time, if $m \geq n$. In [3, 33] the bound was extended to $\Omega(n + m.\alpha(m, n))$ time for all n and m. The proof of the bound relies heavily on the *separation condition* (cf. [32]), which is the following property:

> At any time during the computation, the contents of the memory of the pointer machine can be partitioned into collections of records such that each collection corresponds to a currently existing set, and no record in one collection contains a pointer to a record in another collection.

As shown in [27], the separation condition can imply a loss of efficiency (see also

Table 5.1). Hence, the lower bound of [32] is not general enough for pointer machines. (Moreover, not all known Union-Find algorithms that run on a pointer machine satisfy the separation condition: the algorithm in Chapter 3 does not satisfy it because a list of all records with set names is used. However, since the list is not used for Finds, the model in [32] can be liberalized such that the algorithm implies a modified algorithm with the same time bound that does satisfy the condition.)

In this chapter, we prove a $\Omega(n+m.\alpha(m,n))$ lower bound for the Union-Find problem on a general pointer machine, without the separation condition. A consequence of the lower bound is that the Union-Find algorithms given in [31, 33] and in Chapter 3 are optimal for pointer machines.

In [11] an algorithm for the Split-Find problem is presented that runs in $O(n + m.\alpha(m,n))$ time on a pointer machine, and in the previous chapter we presented an algorithm for the generalized Split-Find problem with the same complexity. Until now, no lower bound was known for the Split-Find problem on a pointer machine.

We prove a $\Omega(n + m.\alpha(m,n))$ lower bound for the Split-Find problem on general pointer machines too. A consequence of the lower bound is that the above Split-Find algorithms are optimal for pointer machines.

Our proofs use inductive structures that are related to the inductive structures used in the previous chapters. The lower bounds are proved for *all* possible sequences of Unions (or Splits, respectively) that are in some class of "balanced" sequences of Unions (or Splits) and that may be known in advance: each such sequence can be intermixed with appropriate Finds to yield the lower bound. Some consequences are that the special cases of the Union-Find problem that can be solved in linear time on a RAM (cf. [13]) (viz., where the structure of the (arbitrary) Union sequence is known in advance) do not have a linear solution on a pointer machine, and that although the Split-Find problem can be solved in linear time on a RAM (cf. [13]), this is not possible on a pointer machine.

Table 5.1: Complexity of Set Manipulation on Pointer Machines

Problem[1]	General model		Separation condition	
UNION-FIND				
worst case/instruction	$O(\log\log n)$	[7][2]	$\Theta(\frac{\log n}{\log\log n})$	[4]
amortized	$\Theta(n + m.\alpha(m,n))$	**new**	$\Theta(n + m.\alpha(m,n))$	[32]
SPLIT-FIND				
worst case/instruction	$\Theta(\log\log n)$	[27]	$\Theta(\log n)$	[27]
amortized	$\Theta(n + m.\alpha(m,n))$	**new**	$\Theta(n + m.\alpha(m,n))$	**new**

[1] n is the number of elements and m is the number of Finds

[2] for special cases of the Union-Find problem

Recently, in [10] a lower bound was proved for the Union-Find problem on the Cell Probe Machine with word size $\log n$, where n is the size of the universe. Our result does not use any restrictions on the word size, but is only based on properties of addressing by means of pointers instead. Some previous lower bounds for the Union-Find and the Split-Find problem on pointer machines were given for the worst-case time of the Union-Find problem on a pointer machine with the separation condition [4] and the worst-case time of the Split-Find problem [27]. Table 5.1 gives an overview of the existing and new results for lower bounds on pointer machines. (The upper bound for the Split-Find problem on pointer machines with the separation condition is given in [24].)

As remarked by Tarjan in [32], for each individual Union-Find problem on n elements there exists a dedicated pointer machine that solves the problem in linear time. (E.g., take a pointer machine with at most n pointers per node and link each element to a central node and link the central node to each set name.) Therefore, it is not possible to have a non-trivial general lower bound for all pointer machines with a varying number of pointers per node. (Note that this observation holds for all related problems too, including worst-case problems.) Tarjan conjectured that for individual pointer machines the α-bound should hold. In this chapter, we prove that this bound holds indeed. Moreover, we show that there is a uniform constant d that holds for *all* pointer machines, such that a lower bound of $d.(n + m.\alpha(m, n))$ steps holds for all m and *asymptotically* for n. This implies that there is no "asymptotic speed up" for the Union-Find problem if we increase the maximal number of pointers per node in a pointer machine. Note that this is the strongest result that is possible. The same observations can be made w.r.t. the Split-Find problem.

This chapter is organized as follows. In Section 5.2 the model of pointer machines is considered. In Section 5.3 we define some notions w.r.t. Unions and we introduce machines for which we prove lower bounds in Section 5.4. In Section 5.5 the actual lower bound for the Union-Find problem is proved. In Section 5.6 the lower bound for the Split-Find problem is proved.

5.2 Pointer Machine Model

The computational model we use is a liberal version of the pointer machine as described in [32] (see also [20, 21, 30]). A *pointer machine* consists of a collection of nodes. A pointer is the specification of some node (namely, of the node pointed to). Each node contains c fields that each may contain one pointer or the value *nil* ($c \geq 1$). (Note that in this chapter we make difference between a pointer and the value *nil*.) The instructions that a pointer machine can execute are of the following types:

- the creation of a new node with *nil* in all its fields,

- a change of the contents of a field of a node.

We call a pointer machine with c fields per node a c-pointer machine. A *program* is a sequence of instructions to be executed by a pointer machine. (The instructions given above are more *liberal* than those in [32] since we do not restrict the way of addressing yet. The special way of addressing will be condensed in the definition of the cost of the operation Find. Furthermore, we do not consider an output instruction explicitly.)

A pointer machine can be regarded as a dynamic directed graph when a pointer to node y in a field of some node x is represented by an edge (x, y). A *path* from node x to node y is a sequence of nodes such that each node contains a pointer to its successor in the sequence and the first and last node of the sequence are x and y, respectively. The length of a path is the number of nodes in it, not counting its first node. The *distance* from x to y is the minimum length of any path from x to y.

The *Union-Find problem* on a pointer machine can be formulated as follows (also cf. [32] or [27, 31, 33]). Let U be a collection of nodes, called elements. Suppose U is partitioned into a collection of sets, and suppose to each set a (possibly new) unique node is related, called "set name". This partition is called the initial partition. (For the regular Union-Find problem the sets in the partition are singleton sets; however, for convenience in our analysis, we allow other partitions too.) The problem is to carry out a sequence of the following operations:

- Union(A,B): join the sets A and B (destroying the old sets A and B) and relate a set name to the resulting set

- Find(x): return the name of the current set in which element x is contained.

The occurring set names must satisfy the condition that, at every moment, the names of the existing sets are distinct. (Note that the name of the resulting set is not prescribed by the Union operation.) Moreover, the operations are carried out *semi on-line*, i.e., each operation must be completed before the next operation is known, while the subsequence of Unions may be known in advance.
An *execution* of a sequence of Union and Find operations on a pointer machine consists of a (so-called initial) contents of the pointer machine together with a sequence of programs that carries out the Union-Find problem according to the following rules:

1. initially, before the first operation is carried out, the contents of the pointer machine, called the initial contents, reflects the initial partition of the universe: i.e., for each element there exists a path to the (unique) name of the set in which it is contained.

2. each Union is carried out by executing a Union program, which halts having modified the contents of the pointer machine to reflect the Union (where some node is indicated as the name of the resulting set) and, hence, to reflect the new partition of the universe.
3. each Find is carried out by executing a Find program, which halts having identified the name of the set containing the considered element while the pointer machine still reflects the (unchanged) partition of the universe.
4. for each Union or Find operation in the sequence, the corresponding program is not executed until the program of its predecessor operation has halted.

The *cost* of an execution of a sequence of Union and Find operations is the cost of he Union and Find operations, which are defined as follows:

- the cost of a Union is the number of *pointer addings*, i.e., changes in fields that change the contents of a field (whatever the contents was) into some pointer (hence, not *nil*).
- the cost of Find(x) is the length of the shortest path from x to its set name at the start of the Find together with the number of pointer addings performed during the Find.

Then the *number of (pointer machine) steps* performed during the execution of a Union-Find problem certainly is at least the cost of that execution as we defined it, with a minimum of one step per operation. (We will use the notion of steps only in some final theorems.) Note that in our complexity measure (viz, cost and number of steps) we do not account for any change of the contents of a field to *nil*.

5.3 Turn Sequences and GU(i,c,p) Machines

In this section and in the next, we only consider the Union operation and a related operation. Consider a universe V. Let US be a sequence of Unions on V starting from partition P and resulting in partition P'. We represent each Union by the pair (A, B) of the two sets A and B that are joined by it. Henceforth we use the sequence $((A_k, B_k))_k$ so obtained to denote the Union sequence US. US is called *complete* if P consists of singleton sets and $P' = \{V\}$.

Suppose universe V has 2^x elements (for some integer x). Let P be a partition of V into sets of size 2^a (for some integer a). A *Union Turn* or *0-Turn* T with initial partition P is a collection of pairs (A, B) of sets $A, B \in P$ such that each set in partition P occurs exactly once as a component in the collection of pairs. (The Union Turn actually denotes the joining of the sets in the pairs.) Partition

$P' = \{A \cup B | (A, B) \in T\}$ is called the result partition of T (consisting of sets of size 2^{a+1}). A *0-Turn sequence* $TS = (T_i)_i$ is a sequence of 0-Turns T_i such that the result partition of any 0-Turn is the initial partition of the next 0-Turn (if any) in the sequence.

Now consider some subuniverse $U \subseteq V$ and some α, $0 \leq \alpha < \frac{1}{2}$, with $|U| \geq (1 - \alpha).|V|$. Consider a 0-Turn T on V. Then the restriction of T to U is given by

$$T|_U = \{(A \cap U, B \cap U) | (A, B) \in T\}.$$

We call $T|_U$ an *α-Turn* or just a *Turn*. The initial partition of $T|_U$ consists of all non-empty sets occurring in the Turn and the result partition is the collection $\{A \cup B | (A, B) \in T|_U \wedge A \cup B \neq \emptyset\}$. We say that the sets in such a partition of U have *α-size* 2^a if the sets in the corresponding partition of V have size 2^a. (Note that the actual universe $V \supseteq U$ does not need to be known explicitly: a follows directly and uniquely from the partition of U, since, by $0 \leq \alpha < \frac{1}{2}$, the partition consists of sets of size $\leq 2^a$ of which at least one must have size $> 2^{a-1}$.) Now consider a 0-Turn sequence $TS = (T_i)_i$ on V. Then the sequence $TS|_U := (T_i|_U)_i$ is called an *α-Turn Sequence* on universe U. The *initial partition* of the sequence is the initial partition of its first Turn and the *result partition* of the sequence is the result partition of its last Turn. Note that both the universe U, the initial partition and the final partition are completely determined by the α-Turn sequence. A 0-Turn sequence is called *complete* if the initial partition consists of singleton sets and the result partition consists of one set.

The *operation* α-Turn T is given by: for each pair $(A, B) \in T$ $(A \neq \emptyset \vee B \neq \emptyset)$, join the sets A and B (destroying the old sets A and B if both A and B are nonempty) and relate some set name to the resulting set $A \cup B$. (Note that if e.g. $A \neq \emptyset = B$ then set A remains unchanged, but it may get a new name.) The names of the resulting sets have to be distinct.

We now consider the actual *executions* of sequences as described above. An *execution* of a Union sequence US is defined as an execution of a sequence of Union and Find operations (as defined in Section 5.2) consisting of the Union sequence US only, where the non-occurrence of the Find operations may be known in advance (and, hence, because of the semi on-line condition, the entire Union sequence may be known in advance). An *execution* of an α-Turn Sequence on a pointer machine consists of a (so-called initial) contents of the pointer machine together with a sequence of executions of α-Turn operations according to the following rules:

1. initially, before the first operation is carried out, the contents of the pointer machine (called the initial contents) reflects the initial partition of the universe: i.e., to each nonempty set some (unique) set name is related and for each element there exists a path to the name of the set in which it is contained.

2. each α-Turn is carried out by executing a program, which halts having modified the contents of the pointer machine to reflect the α-Turn and, hence, to reflect the new partition of the universe.

3. for each operation in the sequence, the corresponding program is not executed until the program of its predecessor operation has halted.

The above executions are called *UF(i, c)-executions* if the executions are performed on a c-pointer machine and if initially (i.e., when the pointer machine reflects the initial partition) and at the end of each operation (i.e., when the pointer machine reflects the partition resulting from the operation) each element has distance at most i to its set name.

Let TS be a 0-Turn sequence. Then a Union sequence obtained from TS by replacing each Turn by a sequence of its pairs is called an *implementation* of TS. A Union sequence is called *balanced* if it is an implementation of a complete 0-Turn sequence. A Union sequence on a universe U of n elements is called *sub-balanced* if it is a complete Union sequence on U that consists of a balanced Union sequence on some subuniverse $V \subseteq U$ with $|V| > \frac{1}{2}n$ that is intermixed with additional Unions. Obviously, for any universe there exists a sub-balanced Union sequence on it.

Lemma 5.3.1 *Let TS be a complete 0-Turn sequence. Let US be a Union sequence that is an implementation of TS. Let E be a UF(i, c)-execution of US. Then there exists a UF(i, c)-execution of TS with cost that is at most the cost of E.*

Proof. The UF(i, c)-execution E is a valid execution of TS if all instructions in E for the Unions corresponding to one Turn are executed consecutively as one program. □

Definition 5.3.2 *Let $i \geq 1$ and $1 \leq c \leq p$. A GU(i, c, p) machine G (Generic Union machine) is a pointer machine that is used for the execution of an α-Turn sequence and for which the following constraints and modifications hold:*

1. *at any moment the collection of nodes in G is partitioned into $i + 1$ disjoint sets, called layers. The layers are numbered from 0 to i. Every node remains in the same layer.*

2. *at any moment set names are in layer 0 and elements are in layer i.*

3. *nodes in layer i have p fields and all other nodes have c fields.*

4. *a field of a node in layer j ($0 \leq j \leq i$) contains either the value* nil *or a pointer to a node in layer $j - 1$ (if $j \geq 1$).*

Lemma 5.3.3 *Let TS be a 0-Turn sequence on a universe U of n elements (n is a power of two). Let E be a UF(i, c)-execution of TS and let C be the cost of E. Then there exists an execution EE of TS on a GU($i, c+1, c+1$)-machine GG such that initially in GG, when GG reflects the initial partition of TS, there are at most $2.(c+1)^i.n$ fields that contain a pointer, and such that EE has cost that is at most $2.i.(c+1)^{i-1}.C$ if $i \geq 2$ and at most C if $i = 1$.*

Proof. Let G be a c-pointer machine G on which execution E is performed. Let the c fields of a node be numbered from 1 to c. We first derive an execution EE' on a GU($i, c+1, c+1$) machine GG' from E. Every node x in G has for each j ($0 \leq j \leq i$) a (fixed) representative node x_j in layer j of GG' and each node in GG' is a representative of one node in G. Let the fields of a node in GG' be numbered from 0 to c. Then execution EE' is obtained from E by maintaining the following relations:

- for each node x in G the representative x_j in GG' with $1 \leq j \leq i$ contains a pointer to the representative x_{j-1} in its 0^{th} field;
- if in G node x contains a pointer to node y in its a^{th} field ($1 \leq a \leq c$), then in GG' node x_j ($1 \leq j \leq i$) contains a pointer to y_{j-1} in its a^{th} field;
- all other fields in GG' contain nil.

The elements in GG' are the representatives e_i of the elements e in G (i.e., these nodes e and e_i are identified with each other). The set names in GG' are the representatives x_0 of nodes x that occur as set names in GG'.

We describe how to obtain an execution EE on GG. Each node x' in GG' has at most one representative node x in GG and conversely, each node in GG is the representative of precisely one node in GG'. Moreover, node x in GG is in the same layer as its original x' in GG'. Then execution EE is obtained from EE' by the following rules:

- the initial contents of GG consists of those nodes x for which node x' in the initial contents of GG' is reachable from some element in GG' (i.e., there exists a path in GG' from some element to x').
- at the end of each operation GG contains all nodes x that either existed in GG at the start of that operation or of which the (possibly just created) original x' in GG' is reachable from some element in GG' at the end of that operation in EE'.
- initially and at the end of each operation the contents of the fields satisfy: if in GG' the a^{th} field of node x' contains a pointer to node y', then in GG the a^{th} field of node x (if present) contains a pointer to node y (if present) and it contains nil otherwise.

Note that a node in GG' can only become reachable from some element in GG' if some pointer adding occurs in a field in G. Each field in G corresponds to at most i fields outside layer 0 in GG. Each pointer added in such a field points to a node in a layer j with $0 \leq j < i$. Moreover, a node in layer j: $0 < j < i$ of GG' has at most $\sum_{k=j}^{1}(c+1)^{j-k} \leq 2.(c+1)^{j-1} - 1 \leq 2(c+1)^{i-2} - 1$ nodes outside layer 0 of GG' that are reachable from it. Therefore it follows that any pointer adding in G can certainly be performed within a factor $i.(1+(c+1).(2.(c+1)^{i-2}-1)) \leq 2.i.(c+1)^{i-1}$ of cost if $i \geq 2$. (For, any node that becomes reachable has $c+1$ fields that may contain a pointer (at a cost of 1 per pointer) except for the nodes in layer 0 that contain *nil* in their fields only (having cost 0).) For $i = 1$ we obtain factor 1, since a layer j with $0 < j < 1$ does not exist.

Finally it is easily seen that initially in GG there are at most $2.(c+1)^{i}.n$ fields that do not contain *nil* (and even $(c+1).n$ for $i = 1$). □

5.4 Lower Bounds on GU(i,c,p) Machines

In this section we will prove lower bounds for GU(i, c, p) machines.

Lemma 5.4.1 *Let G be a GU(1, c, p) machine. Let TS be an α-Turn sequence for some α, $0 \leq \alpha \leq \frac{1}{4}$, and let n be the number of elements. Suppose the initial partition consists of sets of α-size 2^{q_0} and the result partition consists of sets of α-size 2^{q_1}. Suppose $q_1 - q_0 \geq 4p$. Let E be an execution of TS on G. Then at least $\frac{1}{12}.n.(q_1 - q_0)$ pointer addings occur in E.*

Proof. Let U be the universe of elements of TS. By the definition of α-Turn sequence, there exists a universe $V \supseteq U$ and a 0-Turn sequence TSO on V such that $TS = TSO|_U$ and $n \geq (1-\alpha).|V|$. Let integer v be given by $|V| = 2^v$. Hence, $n \geq (1-\alpha).2^v$.

We define a so-called matching sequence of an execution of TS or TSO as follows. Let TT denote TS or TSO. Let EE be an execution of TT on G. Firstly, for a Turn T in TT a matching sequence for T w.r.t. EE is a sequence that contains all the pairs (e, s) of elements e and set names s such that s is the set name for e at the end of the program for T in EE. A matching sequence of EE is a sequence of pairs obtained by replacing each Turn in TT by a matching sequence for that Turn w.r.t. EE.

Consider an execution EE of TSO on G. Let M be a matching sequence of EE. Then it obviously consists of $(q_1 - q_0).2^v$ pairs. For some node s that occurs as a set name in M, consider the last time that s is the name of a set in M. Let A be this set and let T_A be the 0-Turn that yields set A. Suppose A has 2^a elements. Then the matching sequence for T_A occurring in M contains 2^a different pairs with s as set name. For all 0-Turns preceding T_A at most one set per 0-Turn has s as its set name. Therefore, at most $1+2+2^2+\ldots+2^{a-1}=2^a-1$ pairs in M contain s as set name before the matching sequence for T_A occurs in M. (These pairs may contain elements of A.) Therefore at least half of the pairs in M that contain s as set name are distinct. Hence the number of distinct pairs of elements and set names in M is at least $\frac{1}{2}.(q_1-q_0).2^v$.

Hence, any matching sequence of an execution of TSO contains at least

$$\frac{1}{2}.(q_1-q_0).2^v \tag{5.1}$$

different pairs.

Consider the execution E of TS. Let M be a matching sequence for E. Note that E can be augmented to be an execution of TSO by performing at most $(q_1-q_0).(2^v-n)$ pointer addings in the 2^v-n elements of $V\backslash U$ during the Turns. Then M appropriately intermixed with $(q_1-q_0).(2^v-n)$ pairs for the elements in $V\backslash U$ is a matching sequence of the resulting execution of TSO. By (5.1) this gives that there must be at least $\frac{1}{2}.(q_1-q_0).(2n-2^v)$ different pairs in M.

Note that each pair (x,s) in matching sequence M corresponds to a pointer to set name s in some field of node x. Since initially every element in G has at most p pointers, it follows that the total amount of pointer addings in E is at least

$$\begin{aligned}
&\frac{1}{2}(2n-2^v)(q_1-q_0)-n.p\\
\geq\;& \frac{1}{2}(2n-\frac{1}{1-\alpha}.n)(q_1-q_0)-n.p\\
\geq\;& \frac{1}{2}(2n-(\frac{4}{3}\alpha+1).n)(q_1-q_0)-n.p\\
=\;& (\frac{1}{2}-\frac{2}{3}\alpha).n.(q_1-q_0)-n.p\\
\geq\;& (\frac{1}{4}-\frac{2}{3}\alpha).n.(q_1-q_0)\\
\geq\;& \frac{1}{12}.n.(q_1-q_0).
\end{aligned}$$

by using $n\geq(1-\alpha).2^v$, $0\leq\alpha\leq\frac{1}{4}$ and $4p\leq q_1-q_0$. □

Corollary 5.4.2 *Let G be a $GU(1,c,p)$ machine. Let TS be a complete 0-Turn sequence on n elements (n is a power of two). Suppose $4p\leq a(1,n)$. Let E be an execution of TS on G. Then at least $\frac{1}{12}.n.a(1,n)$ pointer addings occur in E.*

We introduce some notions. An execution of an α-Turn sequence TS on a GU(i,c,p) machine is called *conservative* if the program for each Turn is minimal w.r.t. changes of contents of fields: i.e., the omission of one field change in the program for the

Turn would yield that at the end of the Turn there would be no path from some element to its (new) set name. As a consequence, changes of the contents of fields from a pointer to *nil* do not occur in a conservative execution: all field changes are pointer addings.

Obviously, for each execution E of an α-Turn sequence TS on a GU(i,c,p)-machine G there exists a conservative execution E' of TS on G with cost not exceeding the cost of E and that starts with the same initial contents of G. This is seen as follows:

- the initial contents for E' equals that for E,
- all creations of nodes performed by E are also performed by E',
- the program for a Turn in E' may (only) change the contents of a field into the contents that the field has at the end of the program for that Turn in E,
- the program for a Turn in E' is minimal w.r.t. changes of contents of fields: i.e., the omission of one field change in the program would yield that at the end of the Turn some element would not have a path to its (new) set name in G.

Obviously, E' is conservative and the cost of E' does not exceed the cost of E. Therefore it suffices to consider conservative executions only.

We need the following claim.

Claim 5.4.3 *Let G be a GU(i,c,p) machine. Let TS be an α-Turn sequence. Suppose the initial partition of TS consists of sets of α-size 2^a. Let E be a conservative execution of TS on G. Suppose that in the initial contents of G for E at most F fields contain the same pointer. Then at the moment in E that G reflects a partition that consists of sets of α-size 2^b, at most $F+2.c^{i-1}2^b$ fields contain the same pointer.*

Proof. The bound trivially holds for $a=b$. Moreover, no fields contain pointers to nodes in layer i. Hence we only need to consider pointers to nodes outside layer i. Suppose the bound holds for some b with $b \geq a$. Then initially (if $b=a$) or after the execution of the Turn that yields sets of α-size 2^b (if $b>a$) at most $F+c^{i-1}2^{b+1}$ fields contain the same pointer in G. Consider G at the end of the execution of the Turn yielding sets of α-size 2^{b+1}. Colour all fields with new pointers arisen from this Turn red. For any node x outside layer i there are at most c^{i-1} set names that are reachable from node x, say that the collection of these set names is $S(x)$. Moreover, since the Turn sequence is executed conservatively, for every red field with a pointer to x there exists some element e for which all paths from e to its (unique) set name in $S(x)$ use that red field. (Consequently, for distinct red fields with a pointer to x such elements are distinct.) Since the sets arising from the Turn have size at most 2^{b+1}, there are at most $2^{b+1} \cdot c^{i-1}$ red fields with a pointer to x. Hence, at most $F+c^{i-1}(2^{b+1}+2^{b+1}) \leq F+c^{i-1}2^{b+2}$ fields contain a pointer to x. □

Lemma 5.4.4 *Let G be a $GU(i,c,p)$ machine for some $i > 1$. Let TS be an α-Turn sequence TS for some α, $0 \leq \alpha \leq 2^{-(i+1)}$, and let n be the number of elements. Suppose the initial partition of TS consists of sets of α-size $A(i,q_0)$ and the resulting partition of TS consists of sets of α-size $A(i,q_1)$. Let E be an execution of TS on G, where in the initial contents of G at most $A(i,q_0+1)$ fields contain the same pointer. Suppose $c^{i-1} \cdot p \leq A(i,q_0)$, $q_0 \geq 4$ and $q_1 - q_0 \geq 4$. Then at least $12^{-i}.n.(q_1 - q_0)$ pointer addings occur in E.*

Proof. We prove the lemma by induction. Let $i \geq 2$. Suppose that if $i - 1 \geq 2$ then the lemma holds for $i - 1$. We prove that the lemma holds for i.

W.l.o.g. E is conservative. Let U be the universe of the elements in TS. There exists a universe $V \supseteq U$ with $|V| = 2^v$ and a 0-Turn sequence TSO on V such that $TS = TSO|_U$ and $n \geq (1-\alpha)2^v$. We split TS into consecutive subsequences TS^{pre}, TS^{post} and TS_k $(0 \leq k \leq \lfloor\frac{q_1-q_0-1}{3}\rfloor - 1)$ such that $TS = TS^{pre}, (TS_k)_k, TS^{post}$ and for each k, the initial partition of TS_k consists of sets of α-size $A(i, q_0 + 3k + 1)$ and the result partition consists of sets of α-size $A(i, q_0 + 3k + 4)$. (The subsequence TS^{post} may be empty.) Let TSO be split into subsequences TSO^{pre}, TSO^{post} and TSO_k such that $TSO_k|_U = TS_k$. (Obviously TS_k is an α-Turn sequence that is the restriction of TSO_k to U.)

Consider execution E. Let C_k be the contents of G at the start of the execution of TS_k. Then C_k represents the partition in sets of α-size $A(i, q_0 + 3k + 1)$. Since E is conservative, it follows by Claim 5.4.3 that in C_k the number of fields that contain the same pointer is at most

$$\begin{aligned} & A(i,q_0+1) + 2 \cdot c^{i-1} \cdot A(i,q_0+3k+1) && (5.2)\\ \leq\ & A(i,q_0+3k+1) \cdot (1 + 2A(i,q_0)) && (5.3)\\ \leq\ & (A(i,q_0+3k+1))^2 && (5.4)\\ \leq\ & A(i,q_0+3k+2) && (5.5)\end{aligned}$$

since initially in G at most $A(i, q_0 + 1)$ fields contain the same pointer and since $c^{i-1} \leq c^{i-1}p \leq A(i,q_0)$, $i \geq 2$ and $q_0 \geq 4$.

By Claim 5.4.5 (given below) it follows that at least $\frac{1}{2.12^{i-1}}n$ pointer addings occur in E for the execution of TS_k. Hence, by $q_1 - q_0 \geq 4$ at least

$$\lfloor\frac{q_1 - q_0 - 1}{3}\rfloor.(\frac{1}{2.12^{i-1}}.n) \geq \frac{1}{2.12^{i-1}}.\frac{q_1 - q_0}{6}.n \geq \frac{1}{12^i}.(q_1 - q_0).n$$

pointer addings occur during execution E of TS.

We are left with the task to prove Claim 5.4.5.

Claim 5.4.5 *Let $0 \leq k \leq \lfloor\frac{q_1-q_0-1}{3}\rfloor - 1$. Let A be an execution of TS_k on G. Suppose that initially in G (when the partition in sets of α-size $A(i, q_0 + 3k + 1)$ is reflected) at most $A(i, q_0 + 3k + 1))^2 \leq A(i, q_0 + 3k + 2)$ fields contain the same pointer. Then A contains at least $\frac{1}{2.12^{i-1}}n$ pointer addings.*

Proof. W.l.o.g. A is conservative. (Note that every change in a field of an element is a pointer adding now.) Suppose A contains less than $\frac{1}{2.12^{i-1}}n$ pointer addings. Let U' be the collection of elements of which the contents of the fields are not changed. Then U' satisfies

$$n' := |U'| \geq (1 - \frac{1}{2.12^{i-1}})n \geq (1 - \frac{1}{2.12^{i-1}}).(1 - \frac{1}{2^{i+1}}).2^v = (1 - \alpha').2^v \qquad (5.6)$$

for some α' with $0 \leq \alpha' \leq 2^{-i}$. Let TS'_k be given by $TS'_k = TSO_k|_{U'}$. Then TS'_k is an α'-Turn sequence on universe U', its initial partition consists of sets of α'-size $A(i, q_0 + 3k + 1)$ and its result partition consists of sets of α'-size $A(i, q_0 + 3k + 4)$.

We construct an execution A' of TS'_k on a GU$(i-1, c, cp)$ machine G' by means of execution A of TS_k as follows.

For each node y at layer $i-1$ or i of G, we denote by $p_k(y)$ the contents of the k^{th} field of y. (Note that $1 \leq k \leq c$ for layer $i-1$ and $1 \leq k \leq p$ for layer i.) For each node y at layer $i-1$ of G', we denote by $p'_k(y)$ the contents of the k^{th} field of y $(1 \leq k \leq cp)$. Then execution A' is obtained from A by maintaining the following relations:

- the contents of G' is identical to the contents of G with respect to layers 0 to $i-2$: i.e., the collection of nodes in these layers are identical and the fields of these nodes contain pointers to the same nodes (if any),
- layer $i-1$ of G' consists of the elements of U' only; these elements have cp pointer fields,
- for an element $e \in U'$ in G', the contents of its fields $p'_h(e)$ in G' $(1 \leq h \leq cp)$ are given by

  $$p'_{(l-1)c+k}(e) = p_k(p_l(e)) \qquad (1 \leq l \leq p,\ 1 \leq k \leq c)$$

  which is *nil* if $p_l(e) = nil$ (and which is the contents of the k^{th} field of the node pointed at by $p_l(e)$ otherwise).

It is easily seen that initially and at the end of each Turn there is a path from an element $e \in U'$ to its set name s in G iff there is a path from e to s in G'. Therefore A' is an execution on G' of the α'-Turn sequence TS'_k on U'.

By the condition given in the claim we have that initially in G (when G reflect the initial partition in sets of α-size $A(i, q_0 + 3k + 1)$) at most $(A(i, q_0 + 3k + 1))^2 \leq A(i, q_0 + 3k + 2)$ fields contain the same pointer. Since the contents of the fields of the elements in U' are not changed by A in G, this gives that execution A' on G' contains at most

$$A(i, q_0 + 3k + 2) \cdot P \qquad (5.7)$$

pointer addings if P is the number of pointer addings performed in A. Moreover, it follows that initially at most

$$(A(i, q_0 + 3k + 1))^4 \tag{5.8}$$

fields in G' contain the same pointer.

We show that the number of pointer addings in A' is at least $\frac{1}{2}.12^{-(i-1)}.n.A(i, q_0 + 3k + 2)$.

Let x and y be given by $x = A(i, q_0 + 3k)$ and $y = A(i, q_0 + 3k + 3)$. Hence, by (2.1) and $i \geq 2$

$$A(i-1, x) = A(i, q_0 + 3k + 1) \tag{5.9}$$
$$A(i-1, y) = A(i, q_0 + 3k + 4). \tag{5.10}$$

Note that by $q_0 \geq 4$ and $i \geq 2$ we have

$$x = A(i, q_0 + 3k) \geq q_0 \geq 4 \tag{5.11}$$
$$y - x = A(i, q_0 + 3k + 3) - A(i, q_0 + 3k) \geq A(i, q_0 + 3k + 2) \geq 4. \tag{5.12}$$

Now we have that A' is an execution of the α'-Turn sequence TS'_k on the GU$(i-1, c, cp)$ machine G' with $0 \leq \alpha' \leq 2^{-i}$, and that the initial partition of TS'_k consists of sets of α'-size $A(i-1, x)$ and the result partition consists of sets of α'-size $A(i-1, y)$. We show that for $i - 1 = 1$ and $i - 1 \geq 2$ the additional constraints for using Lemma 5.4.1 or the induction hypothesis on A' are satisfied.

- $i - 1 \geq 2$. Then by (5.8) we have that initially in G' at most

  $$(A(i, q_0 + 3k + 1))^4 = (A(i-1, x))^4 \leq A(i-1, x+1) \tag{5.13}$$

  fields contain the same pointer by using (5.9) and $A(i-1, x+1) = A(i-2, A(i-1, x)) \geq A(1, A(i-1, x)) = 2^{A(i-1,x)} \geq (A(i-1, x))^4$ where the last inequality follows with $A(i-1, x) \geq x \geq A(i, q_0) \geq A(3, 4) \geq 100$ (by (5.11) and $i - 1 \geq 2$).

  Note that since $1 \leq c^{i-1} \cdot p \leq A(i, q_0) \leq A(i-1, x)$ holds (viz., by the conditions of Lemma 5.4.4) we have

  $$1 \leq c^{i-2} \cdot (cp) \leq A(i-1, x) \tag{5.14}$$

  and that by (5.12) and (5.11) we have

  $$y - x \geq 4 \wedge x \geq 4. \tag{5.15}$$

  By (5.13), (5.14) and (5.15), the induction hypothesis for $i - 1$ yields that there occur at least

  $$12^{-(i-1)}.n'.(y - x) \geq \frac{1}{2.12^{i-1}}.n.A(i, q_0 + 3k + 2)$$

  pointer addings in A' by using (5.6) and (5.12).

- $i - 1 = 1$. Unequality (5.12) and the conditions in Lemma 5.4.4 give $y - x \geq A(i, q_0 + 3k + 2) \geq 4.A(i, q_0) \geq 4 \cdot cp$. Hence

$$y - x \geq 4cp \tag{5.16}$$

  By (5.16) and Lemma 5.4.1 it follows that there occur at least

$$\frac{1}{12}.n'.(y - x) \geq \frac{1}{12}.\frac{1}{2}.n.A(i, q_0 + 3k + 2)$$

  pointer addings in A' by using (5.6) and (5.12).

By the above case analysis it follows that at least $\frac{1}{2}.12^{-(i-1)}.n.A(i, q_0 + 3k + 2)$ pointer addings occur in A'. By (5.7) it follows that there are at least $\frac{1}{2}.12^{-(i-1)}.n$ pointer addings in A. Contradiction with the assumption that there are less than $\frac{1}{2}.12^{-(i-1)}.n$ pointer addings. This proves Claim 5.4.5. □

This concludes the proof of Lemma 5.4.4. □

Lemma 5.4.4 yields the following result.

Corollary 5.4.6 *Let G be a $GU(i, c, c)$ machine for some $i > 1$. Let TS be a complete 0-Turn sequence and let n be the number of elements. Suppose $c^i \leq A(i, \lfloor\frac{1}{2}.a(i, n)\rfloor - 1)$ and $a(i, n) \geq 10$. Let E be an execution of TS on G, where initially in G at most c^{i-1} fields contain the same pointer. Then at least $\frac{1}{2}.12^{-i}.n.a(i, n)$ pointer addings occur in E.*

Proof. W.l.o.g. E is conservative. Let $q_0 = \lfloor\frac{1}{2}.a(i, n)\rfloor - 1$ and $q_1 = a(i, n) - 1$. Then at the moment that that G reflects the partition with sets of α-size $A(i, q_0) = 2^b$ (for some b), it follows by Claim 5.4.3 that in G at most

$$c^{i-1} + 2.c^{i-1}.2^b \leq A(i, q_0).(1 + 2.A(i, q_0)) \leq A(i, q_0 + 1)$$

fields contain the same pointer (by using $i > 1$ and $q_0 \geq 4$). By Lemma 5.4.4 it follows that at least $\frac{1}{2}.12^{-i}.n.a(i, n)$ pointer addings occur in the part of execution E that corresponds to the subsequence of TS with the initial partition consisting of sets of size $A(i, q_0)$ and resulting partition consisting of sets of size $A(i, q_1)$. Thus the cost of E is at least $\frac{1}{2}.12^{-i}.n.a(i, n)$. □

5.5 A General Lower Bound for the Union-Find Problem

Lemma 5.5.1 *Let $i \geq 1$, $c \geq 1$. Let E be a UF(i, c)-execution of a complete 0-Turn sequence TS on n elements (n is a power of two).*

Then E costs at least $\frac{1}{4.12^i.i.(c+1)^{i-1}}.n.a(i,n) - (c+1).n$ pointer addings if $i \geq 2$, $a(i,n) \geq 10$ and $A(i, \lfloor \frac{1}{2}a(i,n) \rfloor - 1) \geq (c+1)^i$, and it costs at least $\frac{1}{12}.n.a(1,n)$ pointer addings if $i = 1$ and $a(i,n) \geq 4(c+1)$.

Proof. Let C be the cost of E. From Lemma 5.3.3 it follows that there exists an execution E' of TS on a $GU(i, c+1, c+1)$ machine G with cost at most $2.i.(c+1)^{i-1}.C$ if $i > 1$ and with cost at most C if $i = 1$, while initially at most $2.(c+1)^i.n$ fields in G contain a pointer.

- For $i = 1$ Corollary 5.4.2 gives that the cost of execution E' is at least $\frac{1}{12}.n.a(1,n)$. Hence, $C \geq \frac{1}{12}.n.a(1,n)$.
- For $i > 1$ we change execution E' into execution E'' as follows. Consider the initial contents of G for E'. Colour a *minimal* collection of fields red such that for each element in G there is a path to its set name using pointers in red fields only: i.e., if some red field would not be red, then there would be some element that would not have a path to its set name via red fields only any more. Colour all other fields that contain a pointer blue. Now the (new) initial contents of G for E'' equals that for E' except that all fields that are not red contain *nil*. Furthermore, execution E'' consists of first adding all pointers in blue fields (at the beginning of the execution of the first operation) followed by E'.

  Hence, the cost of E'' is at most $2.i.(c+1)^{i-1}.C + 2.(c+1)^i.n$. Moreover, initially in G at most $(c+1)^{i-1}$ fields contain the same pointer, since every set consists of one element and since the number of red fields is minimal (also cf. the proof of Claim 5.4.3). By Corollary 5.4.6 it follows that the cost of E'' is at least $\frac{1}{2}.12^{-i}.n.a(i,n)$, which establishes the result for C.

This concludes the proof of Lemma 5.5.1. □

Lemma 5.5.2 *Let $i \geq 1$, $c \geq 1$ and $n \geq 1$. Let n and c satisfy $\alpha(n,n) > \alpha(c,c) + 1$. Let US be a sub-balanced Union sequence on a universe of n elements. Then every UF(i, c)-execution of US with $1 \leq i \leq \alpha(n,n) - 3$ costs at least $n.a(i+1,n)$ pointer addings.*

Proof. Consider a UF(i,c)-execution E of US. Let E have cost C. Since US is sub-balanced, US consists of a balanced Union sequence US' on a subuniverse of $2^v > \frac{1}{2}n$ elements that is intermixed with additional Unions. We modify execution E into execution E' for US' as follows. For each Union Un in US' let $Pre(Un)$ be the longest subsequence of US that ends with Un and that does not contain Unions of US' except for Un. Then a program for a Union Un in US' consists of the sequence of instructions in E for the Unions in $Pre(Un)$. In this way we obtain execution E' that obviously is a UF(i,c)-execution of US' with cost at most C.

Let TS' be the 0-Turn sequence of which US' is an implementation. Then by Lemma 5.3.1 there exists a UF(i, c)-execution E'' of TS' with cost at most C. We show that the cost of E'' is at least $n.a(i+1, n)$. First we show that E'' satisfies the conditions of Lemma 5.5.1. Note that by Lemma 2.3.13 we have

$$a(i,n) \geq 8.12^i.i.(c+1)^{i-1}.(\,2a(i+1,n)+c+1). \tag{5.17}$$

- For $i = 1$ we have by (5.17) $a(1,n) \geq 8.12.(\,2a(2,n)+c+1) \geq 4(c+1)+1$ and hence by $n' > \frac{1}{2}n$ we have $a(1,n') \geq a(1,n) - 1 \geq 4(c+1)$.
- If $i \geq 2$ then we have (by $n' > \frac{1}{2}n$ and by (5.17))

  $$\begin{aligned} & A(i, \lfloor \frac{1}{2}a(i,n')\rfloor - 1) \geq A(i, \lfloor \frac{1}{2}a(i,n)\rfloor - 2) \\ \geq \;& A(i, 4.12^i.i.(c+1)^{i-1}.(\,2a(i+1,n)+c+1)-2) \geq A(i,(c+1)^i) \geq (c+1)^i \end{aligned}$$

  and $a(i,n') \geq \frac{1}{2}a(i,n) \geq 4.12^i.i.(c+1)^{i-1}.(\,2a(i+1,n)+c+1) \geq 10$.

Hence by Lemma 5.5.1 the cost of E'' is at least

$$\frac{1}{12}.n'.a(1,n')$$

if $i = 1$ and at least

$$\frac{1}{4.12^i.i.(c+1)^{i-1}}.n'.a(i,n') - (c+1).n'$$

if $i > 1$. By using $a(i,n') \geq \frac{1}{2}a(i,n)$, $n' > \frac{1}{2}n$ and (5.17) this is at least $n.a(i+1,n)$. This concludes the proof of Lemma 5.5.2. □

Theorem 5.5.3 *There exists a constant $d > 0$ such that:*

> *For any c-pointer machine, for any integer f and for any sub-balanced Union sequence on a universe of n elements there exists a Union-Find problem consisting of the Union sequence intermixed with f Find operations whose execution by the c-pointer machine has a cost that is at least $d.f.\alpha(f,n)$ if $\alpha(n,n) > \alpha(c,c)+1$.*

Proof. Let n and c satisfy the constraints given above. Consider some sub-balanced Union sequence US on n elements. Let

$$i = max\{j | [f \leq \frac{n.a(j,n)}{j} \wedge 1 \leq j \leq \alpha(n,n)-2] \vee j = 1\}. \tag{5.18}$$

We construct a Union-Find problem that contains US as the subsequence of Unions and that costs at least $f.i$ and we show that $i \geq \frac{1}{3}.\alpha(f,n)$. We distinguish two cases.

- $i = 1$. Then at any moment after the first Union, at least one element cannot equal its set name and hence any f Finds performed on such elements cost at least f together.
- $i > 1$. We construct a Union-Find problem semi on-line, starting from the (known) sequence US of Union operations and intermix it with Finds. If at some moment when some partition is reflected (i.e., initially or at the end of some operation) there is an element that has distance $\geq i$ to its set name, and if less than f Finds have been performed thus far, then perform a Find on that element. Otherwise perform the next Union or stop if a next Union does not exist. Let E be the execution of the Unions and Finds obtained in this way.

  We distinguish 2 cases.

  - At least f Finds have been performed. Then obviously these Finds have cost at least $\geq f.i$.
  - Less than f Finds have been performed. We change E into an execution E' of Union sequence US as follows. The initial contents of the pointer machine for execution E' is the contents for E at the beginning of the first Union. All Finds occurring before the first Union are ignored w.r.t. E'. (These Finds are condensed in the new initial contents of the pointer machine.) Furthermore, each execution of a Find (not occurring before the first Union) is appended to the execution of its previous Union. Then obviously the number of pointer addings in E' is at most that in E. Because less than f Finds have been performed, it follows that initially and at the end of the (thus extended) execution of each Union all elements have distance $< i$ to their set names. Therefore, E' is a UF$(i-1, c)$-execution of US with $1 \leq i-1 \leq \alpha(n,n)-3$. By Lemma 5.5.2 it follows that at least $n.a(i,n)$ pointer addings occur in E'. Hence the cost of E is at least $n.a(i,n)$.

  Hence in both cases the cost is at least $min\{f.i, n.a(i,n)\}$. By $i > 1$ and (5.18) we have $f \cdot i \leq n.a(i,n)$. Hence the cost of E is at least $f.i$.

We show that $i \geq \frac{1}{3}.\alpha(f,n)$. We distinguish three cases.

- $1 \leq i < \alpha(n,n) - 2$. Then by (5.18) $\frac{n.a(i+1,n)}{i+1} < f$. Then we certainly have by Lemma 2.3.16 $n.a(i+2,n) < f$. By Lemma 2.3.7 it follows that $i+2 \geq \alpha(f,n)$ and hence by $i \geq 1$ it follows that: $i \geq \frac{1}{3}.(i+2) \geq \frac{1}{3}.\alpha(f,n)$.
- $i = \alpha(n,n) - 2$ (and hence $\alpha(n,n) \geq 3$). From $\alpha(f,n) \leq \alpha(n,n)$ it follows that $i = \alpha(n,n) - 2 \geq \frac{1}{3}.\alpha(n,n) \geq \frac{1}{3}.\alpha(f,n)$.
- $i = 1 > \alpha(n,n)-2$. Hence $\alpha(n,n) \leq 2$. From $\alpha(f,n) \leq \alpha(n,n)$ it follows that $i = 1 \geq \frac{1}{2}\alpha(n,n) \geq \frac{1}{2}\alpha(f,n)$.

Combining the above cases gives that $i \geq \frac{1}{3}.\alpha(f,n)$.

By combining the above results it follows that the cost is at least $\frac{1}{3}.f.\alpha(f,n)$. □

Theorem 5.5.3 implies that even if all Unions are known in advance, the worst case time bound is still $\Omega(f.\alpha(f,n))$ for all sub-balanced Union sequences on a pointer machine that are intermixed with f appropriate Finds. Hence the linear bound proved in [13] for Union-Find problems in which the structure of the (arbitrary) Union sequence is known in advance and that is implemented on a RAM, does not extend to a pointer machine.

Finally, since each operation takes at least one step on a pointer machine, we obtain the following theorem.

Theorem 5.5.4 *There exists a constant $d > 0$ such that:*

> *For any c-pointer machine and for any n and f with $\alpha(n,n) > \alpha(c,c)+1$ there is a Union-Find problem on n elements with a sequence of $n-1$ Union and f Find operations whose execution by the c-pointer machine requires at least $d.(n+f.\alpha(f,n))$ steps.*

Corollary 5.5.5 *For any pointer machine there exists a constant $d > 0$ such that for any $n > 1$ and $f \geq 0$ there is a Union-Find problem on n elements with a sequence of $n-1$ Union and f Find operations whose execution by the pointer machine requires at least $d.(n+f.\alpha(f,n))$ steps.*

5.6 A General Lower Bound for the Split-Find Problem

We first describe the Split-Find problem on a pointer machine. Let U be a linearly ordered collection of nodes, called elements. Suppose U is partitioned into a collection of sets and suppose a (possibly new) unique node is related to each set, called set name. (For the regular Split-Find problem the partition consists of one set only.) We want to be able to perform the following operations:

- Split(A, B): split the set $A \cup B$ with $A < B$ (i.e., $x < y$ for all $x \in A$, $y \in B$) and $A \neq \emptyset \neq B$ into the two new sets A and B (destroying the old set $A \cup B$) and relate set names to the resulting sets.
- Find(x): return the name of the current set in which element x is contained.

The occurring set names must satisfy the condition that, at every moment, the names of the existing sets are distinct. (Note that the name of the resulting set is not prescribed by the Split operation.) Moreover, the operations are carried out *semi on-line*, i.e., each operation must be completed before the next operation is known, while the subsequence of Splits may be known in advance. The definition and rules for pointer machine executions that solve the Split-Find problem are similar to those for the Union-Find problem as given in Section 5.2.

We use the results of Section 5.4 to obtain a lower bound for the Split-Find problem. Like in Section 5.3 we consider a Split sequence as a sequence of pairs $((A_k, B_k))_k$, where a pair (A_k, B_k) represents the operation Split(A_k, B_k). We define a sub-balanced Split sequence as a reversed sub-balanced Union sequence. Then, obviously, there exists a sub-balanced Split sequence on every universe. (Note that we can also define Split Turns and sub-balanced Split sequences independent of Union Turns and Union sequences.) A UF(i, c)-execution of a Split sequence is defined similar as for a Union sequence. We prove the equivalent of Lemma 5.5.2.

Lemma 5.6.1 *Let $i \geq 1$, $c \geq 1$ and $n \geq 1$. Let n and c satisfy $\alpha(n,n) > \alpha(c,c)+1$. Let S be a sub-balanced Split sequence on a universe of n elements. Then any UF(i,c)-execution of S with $1 \leq i \leq \alpha(n,n) - 3$ costs at least $n.a(i+1,n)$ pointer addings.*

Proof. Consider a UF(i, c)-execution E of Split sequence S and let C be the number of pointer addings in it. Let G be the pointer machine on which E is executed. Modify the execution such that no changes in fields from a pointer to *nil* are performed and such that no creation of nodes occur in E (which can easily be obtained by assuming that nodes that are created during E exist in the initial contents of G already, where they contain *nil* in their fields at that moment). Obviously the thus modified execution E is still an execution of S and contains exactly C changes of field contents.

Let S^{-1} be the reverse sequence of S. Then S^{-1} is a sub-balanced Union sequence on universe U. We construct an execution E' of S^{-1} by means of execution E of S as follows. The initial contents of G for E' equals the final contents of G after execution E (i.e., the contents of G at the moment that the program of the last operation in S halts). Then E' is obtained by maintaining the following relations with as few pointer addings as possible. At the end of an execution of a Union in S^{-1}, pointer machine G has the same contents as at the beginning of the corresponding Split in S. Then apparently, a change of the contents of a field by E' during the execution of a Union occurs only if there is a change of the contents of that field by E during the corresponding Split in S. Hence, E' contains at most C changes of field contents.

Since at the beginning or at the end of every Union in S^{-1} the contents of the pointer machine is identical to that at the end or at the beginning of the corresponding Split in S, it follows that E' is a UF(i, c)-execution of Union sequence S^{-1}. Lemma 5.5.2 yields that $C \geq n.a(i+1,n)$. □

Completely similar to the proof of Theorem 5.5.3 we can prove the following theorem.

Theorem 5.6.2 *There exists a constant $d > 0$ such that:*

> *For any c-pointer machine, for any integer f and for any sub-balanced Split sequence on a universe of n elements there exists a Split-Find problem consisting of the Split sequence intermixed with f Find operations whose execution by the c-pointer machine has a cost that is at least $d.f.\alpha(f,n)$ if $\alpha(n,n) > \alpha(c,c) + 1$.*

Theorem 5.6.2 implies that even if all Splits are known in advance, the worst-case time bound on a pointer machine is still $\Omega(f.\alpha(f,n))$ for all sub-balanced Split sequences that are intermixed with appropriate Finds. Hence the linear bound proved in [13] for Split-Find problems on a RAM does not extend to a pointer machine, even if the sequence of Splits is known in advance.

Like for the Union-Find problem we obtain the following further results.

Theorem 5.6.3 *There exists a constant $d > 0$ such that:*

> *For any c-pointer machine and for any n and f with $\alpha(n,n) > \alpha(c,c) + 1$ there is a Split-Find problem on n elements with a sequence of $n - 1$ Split and f Find operations whose execution by the c-pointer machine requires at least $d.(n + f.\alpha(f,n))$ steps.*

Corollary 5.6.4 *For any pointer machine there exists a constant $d > 0$ such that for any $n > 1$ and $f \geq 0$ there is a Split-Find problem on n elements with a sequence of $n-1$ Split and f Find operations whose execution by the pointer machine requires at least $d.(n + f.\alpha(f,n))$ steps.*

Finally, we make some remarks about the separation condition for the Split-Find problem. In case the separation condition holds, the lower bound of Theorem 5.6.3 becomes valid for a uniform d for all n independent of c. (However, in this case we need to include all changes of contents of fields in our ultimate complexity measure, i.e., including changes to *nil*.) This matches the result in [32] for the Union-Find problem with the separation condition. We will not present the proof here.

5.7 Concluding Remarks

We remark that even if during a Union or a Split the new set name is not assigned to the resulting set immediately, but is assigned to it at some later time before or during the first Find that is performed on an element of that set, Theorems 5.5.4 and 5.6.3 still hold. We omit the proofs.

Chapter 6

Maintenance of the 2- and 3-Edge-Connected Components of Graphs: Basic Solutions

6.1 Introduction

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

We present a data structure and algorithms for maintaining the 2- and 3-edge-connectivity relation of a graph. The algorithm starts from an empty graph of n nodes in which edges are inserted one by one, and at any time and for any two nodes x and y, the query that asks whether x and y are 2- or 3-edge-connected can be answered in $O(1)$ time. The insertion of e edges takes $O(n \log n + e)$ time altogether. By using additional data structuring techniques that we will present in Chapter 7, the time bounds for maintaining the 2- and 3-edge-connected components can be improved, as will be demonstrated in Chapter 8.

This chapter is organized as follows. In Section 6.2, we introduce some terminology and state properties dealing with connectivity. In Section 6.3, we consider the dynamic 2-edge-connectivity problem, and in Section 6.4, we consider the dynamic 3-edge-connectivity problem. In the latter case, we first consider maintaining the 3-edge-connected components in 2-edge-connected graphs and then extend this to general graphs.

6.2 Preliminaries

6.2.1 Graphs and Terminology

Definition 6.2.1 *Two nodes x and y are k-edge-connected iff there exist k edge-disjoint paths between x and y.*

It is easily seen that we may require that the paths referred to in the definition are simple paths, without affecting the definition. Furthermore, it is easily seen that if two nodes are k-edge-connected, then they are k'-edge-connected for any k' with $1 \leq k' \leq k$. Note that for $k = 2$, two nodes are 2-edge-connected iff they lie on a common elementary cycle. The following lemma due to Menger (cf. [26]) characterizes pairs of k-edge-connected vertices.

Lemma 6.2.2 [**Menger**] *Nodes x and y are k-edge-connected ($k \geq 1$) iff after the removal of any set of at most $k - 1$ edges x and y are (still) connected.*

If the removal of a set of edges separates the vertices x and y (i.e., disconnects x and y), then this set is called a *cut edge set* for x and y.

Lemma 6.2.3 *k-edge-connectivity is an equivalence relation on the set of nodes of a graph.*

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

Definition 6.2.4 *Let $G =< V, E >$ be a graph. Let $C \subseteq V$ be an equivalence class w.r.t. 2-edge-connectivity. Then $< C, \{(e, x, y) \in E | x, y \in C\} >$ is a 2-edge-connected component of G (induced by C).*

The following lemma is based on the observation that for two nodes that are k-edge-connected ($k \geq 2$), there exist k edge-disjoint simple paths between them, and hence, all the nodes on these paths are 2-edge-connected.

Lemma 6.2.5 *Let $G =< V, E >$ be a graph. Let H be a 2-edge-connected component of G. Then H is a 2-edge-connected graph. Moreover, nodes $x, y \in H$ are k-edge-connected in H iff they are k-edge-connected in G ($k \geq 1$).*

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

For an equivalence class C of nodes w.r.t. 3-edge-connectivity, we can define the notion of a 3-edge-connected component (induced by that class) such that Lemma 6.2.5 holds for 3-edge-connectivity too. We will not give the formal definition here, but only state that it contains the edges of the graph that have both end nodes in C, together with for each pair of nodes x and y in C, a number of new edges between them that equals the maximal number of nontrivial edge-disjoint paths between x and y that intersect with C at x and y only, and that intersect with $V\backslash C$.

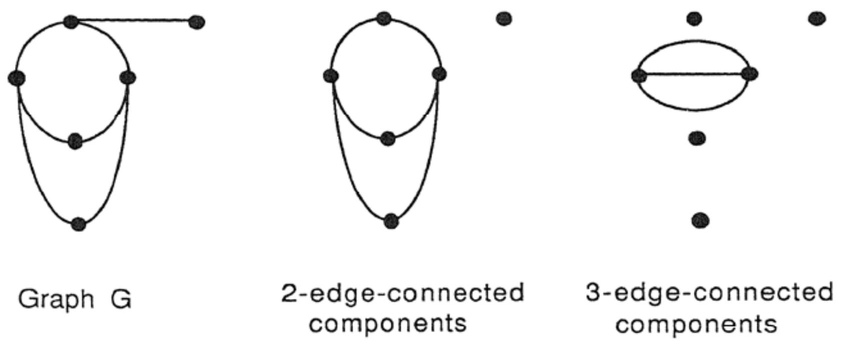
Henceforth, we will usually call an equivalence class for 2-edge-connectivity a *2ec-class*, and an equivalence class for 3-edge-connectivity a *3ec-class*.

By means of Lemma 6.2.5 the following corollary easily follows.

Corollary 6.2.6 *Let G be a graph. Let C_2 be a 2ec-class and let C_3 be a 3ec-class of G. Then either $C_2 \cap C_3 = \emptyset$ or $C_3 \subseteq C_2$. Let H be the 2-edge-connected component of G induced by C_2. If $C_3 \subseteq C_2$ then C_3 is a 3ec-class of H.*

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

Figure 6.1: The 2- and 3-edge-connected components of graph G.



In this chapter, we will represent the 2ec-classes and the 3ec-classes of a graph by means of a "super" graph. To this end, we introduce the notion of a *class node*.

Definition 6.2.7 *Let $G = < V, E >$ be a graph. Let V be partitioned into classes and let some new node be related to each class, where each such node is called the class node of the class which it represents. Let $cc(x)$ be the class node of the class containing node x ($x \in V$). Then the induced node set $cc(V)$, the induced edge set $cc(E')$ of a set of edges $E' \subseteq E$ and the induced graph $cc(G)$ are given by*

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

Lemma 6.2.8 *Let $G = < V, E >$ be a graph and let k be a positive integer. Let V be partitioned into classes and let some new node be related to each class. Suppose that any two nodes x and y that are in the same class are k-edge-connected. Let $cc(x)$ be the class node of the class containing node x ($x \in V$). Then the following holds.*

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

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

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

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

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

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

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

6.2.2 Problem Description

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

insert$((e,x,y))$**:** insert the edge (e,x,y).

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

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

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

In addition, the set 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 somewhere in the graph either the connectivity relation changes or for the 2ec-problem the 2-edge-connectivity relation changes, or for the 3ec-problem the 2-edge-connectivity or 3-edge-connectivity relation changes. An insertion is called *nonessential* otherwise.

6.2.3 Representation and Algorithms

By Lemma 6.2.3, k-edge-connectivity is an equivalence relation. In our algorithms we need operations on equivalence classes like joining classes and determining in which class an element is contained. This problem is abstractly dealt with in the Union-Find problem. In this chapter, we use a simple algorithm (UF(1), cf. Subsection 3.3.1 or [1]) taking $O(n \log n)$ time for all Unions together and $O(1)$ time per Find. This is good enough, since the additional computations already take $O(n \log n)$ time.

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

For maintaining the 3-edge-connected components, we also need a structure for the *Circular Split-Find problem* (cf. Chapter 4). In this chapter, we use a simple algorithm (GSF(1), cf. Subsection 4.3.1) taking $O(n \log n)$ time for all Circular Splits together and $O(1)$ time per Find, again since additional computations already take $O(n \log n)$ time.

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

Two-Edge-Connectivity

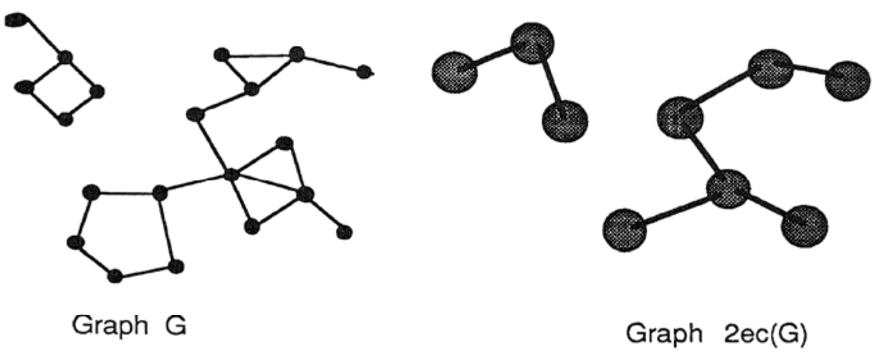
L Graph Observations

' $=< V, E >$ be a graph. The set V can be partitioned into 2ec-classes. Let
2ec-class C be represented by a new node c, called the *class node* of C. Let
) be the class node of the 2ec-class in which the node x is contained. We define
aph $2ec(G)$ as follows (according to Definition 6.2.7):

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

:, $2ec(G)$ is the graph that is obtained if we contract each 2-edge-connected
onent into one (representing) class node. Since $2ec(V)$ represents the set of
lence classes of G (for 2-edge-connectivity), it follows by Lemma 6.2.8 (sub 3)
$ec(G)$ is a forest (cf. Figure 6.2). We maintain the 2-edge-connectivity relation
edge insertions by means of the graph $2ec(G)$.

Figure 6.2: Graph G and the corresponding graph $2ec(G)$.



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

$(x) \neq c(y)$. Then by Lemma 6.2.8 (sub 3) $2ec(x)$ and $2ec(y)$ are not connected
n $2ec(G)$. Hence, $(e, 2ec(x), 2ec(y))$ connects two trees in $2ec(G)$, which now
ecome joined into one tree.

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

$2ec(x)$ and $2ec(y)$ in a tree of $2ec(G)$ and a cycle arises. Hence, all class nodes on the tree path from $2ec(x)$ to $2ec(y)$ become 2-edge-connected in $2ec(G)$. By Lemma 6.2.8 (sub 3) all nodes in V that are contained in the corresponding classes become 2-edge-connected too. The update can now be performed in the following way.

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

3. $2ec(x) = 2ec(y) \wedge c(x) = c(y)$. Then the edge (e, x, y) connects two nodes that are 2-edge-connected in G, and, hence, insertion of this node will not affect the 2-edge-connectivity relation (cf. Lemma 6.2.8, sub 3).

6.3.2 The Algorithms

We will now describe the different steps in more detail.

In our algorithms we represent each of the collections of connected classes and 2ec-classes of a graph G by a Union-Find structure (cf. Subsection 6.2.3), where the name of each class is the class node of that class (i.e., a Find on an element of a class outputs the class node related to that class). Therefore we (may) denote $Find_c(x)$ or $Find_{2ec}(x)$ by $c(x)$ or $2ec(x)$ too (cf. Subsection 6.2.3). We represent the forest $2ec(G)$ by means of rooted trees. We denote a rooted forest by $2ec(G)^R$ without making the roots explicit in our description.

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

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

Now, suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G = < V, E >$. Then after inserting edge (e, x, y) in the proper adjacency lists, procedure $insert_2$ given in Figure 6.4 updates the structure as follows. (The sub-procedures of $insert_2$ are given in Figure 6.5 and Figure 6.6.) We distinguish the three previous cases.

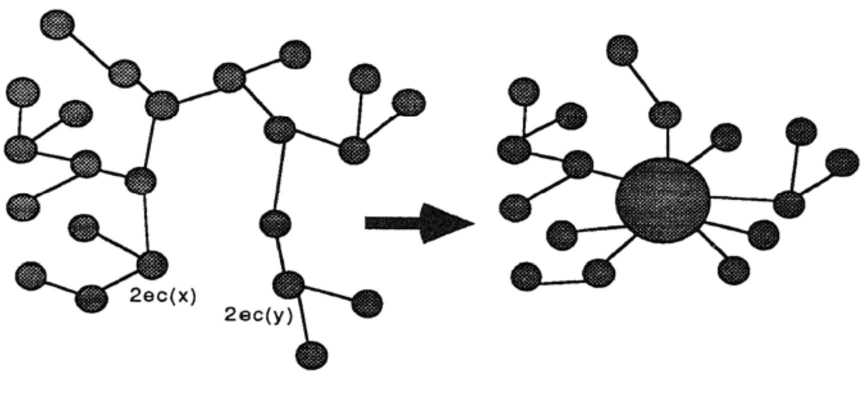
1. $c(x) \neq c(y)$ (line 2-8). Then $(e, 2ec(x), 2ec(y))$ connects two different trees in $2ec(G)$ that now are joined to one tree. Since the trees are represented as rooted trees, this means that one of the two trees has to be redirected w.r.t. the father relation of classes. We take the tree with the smallest size, i.e., the

tree that has the least number of nodes that are contained in the classes in that tree. (This can be determined by means of a parameter in the Union-Find structure for connected components.) W.l.o.g., this is the tree containing $2ec(y)$. It suffices to "reverse" the father pointers for the nodes on the root path of the class node $2ec(y)$ (i.e., the path from node $2ec(y)$ to the root of its tree). This is performed by procedure *ReverseRootPath* that is given in Figure 6.5.

2. $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$ (line 9-12). All classes on the tree path in $2ec(G)$ between $2ec(x)$ to $2ec(y)$ become 2-edge-connected and must be joined. This is done as follows. First of all, the tree path P between $2ec(x)$ and $2ec(y)$ is obtained by means of procedure call $TreePath_2$ that outputs tree path P together with a pointer *fath* to the father edge of the nearest common ancestor *top* of $2ec(x)$ and $2ec(y)$ in $2ec(G)^R$ (this pointer is *nil* if this edge does not exist). These are obtained by stepwise traversing the root paths from $2ec(x)$ and $2ec(y)$ in an alternating way (cf. Figure 6.6) until a node *top* has been visited by both traversals. This class node *top* is the nearest common ancestor of $2ec(x)$ and $2ec(y)$. Then the path between $2ec(x)$ and $2ec(y)$ consists of the two parts of these root paths up to and including this "first mutual class node".

   The joining of the classes on P is done by means of Union_{2ec} operations. Note that the father edge of the resulting class is the father edge of the (old) class *top*. (Cf. Figure 6.3.)

Figure 6.3: Joining the classes of the tree path from $2ec(x)$ to $2ec(y)$.



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

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

Figure 6.4: Procedure $insert_2(e, x, y)$.

```
(1)  procedure $insert_2(e, x, y)$;
(2)  if $c(x) \neq c(y)$
(3)     $\longrightarrow$ if $size(c(x)) \geq size(c(y))$
(4)           $\longrightarrow$ $ReverseRootPath(2ec(y))$; $father(2ec(y)) := (e, x, y)$
(5)        $[\!]$ $size(c(x)) < size(c(y))$
(6)           $\longrightarrow$ $ReverseRootPath(2ec(x))$; $father(2ec(x)) := (e, x, y)$
(7)        fi;
(8)        $Union_c(c(x), c(y))$
(9)  $[\!]$ $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$
(10)    $\longrightarrow$ $Treepath(2ec(x), 2ec(y), P, fath)$;
(11)       for all $C \in P \backslash \{2ec(x)\}$ $\longrightarrow$ $Union_{2ec}(C, 2ec(x))$ rof;
(12)       $father(2ec(x)) := fath$
(13) $[\!]$ $c(x) = c(y) \wedge 2ec(x) = 2ec(y)$
(14)    $\longrightarrow$ skip
(15) fi
```

Figure 6.5: Procedure $ReverseRootPath(C)$.

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

6.3.3 Time Bounds

We consider the time complexity of the algorithm. The insert operations can be performed in $O(n \log n + e)$ time for e edge insertions together (where n is the number

Figure 6.6: Procedure $TreePath_2(C, D,$ **output** $P, fath)$.

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

(description)

    • stepwise traverse the root paths from $C$ and $D$ alternatingly, i.e., by performing
      steps of the traversals of these root paths in an alternating way. During these
      traversals, mark the class nodes encountered and stop the traversals if one of
      the two path traversals reaches a class node $top$ that has been marked by the
      other traversal;

    • path $P$ between $C$ and $D$ consists of the two parts of these root paths up to
      and including $top$;

    • $fath := father(top)$;

    • remove the marks
```

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

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

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

A query *2ec-comp*(x) is simply done by performing a $Find_{2ec}(x)$, which takes $O(1)$ time.

Theorem 6.3.1 *There exist a data structure and algorithms that solve the 2ec-problem, which can be implemented as a pointer/* $\log n$ *solution, such that the following holds. Starting from an empty graph* $G = < V, \emptyset >$*, the insertion of* e *edges takes* $O(n \log n + e)$ *time altogether, if* n *is the number of nodes in* G*. Any 2ec-comp*(x) *query can be answered in* $O(1)$ *time. The data structure can be initialised in* $O(n)$ *time and uses* $O(n)$ *space.*

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

6.3.4 Algorithms for Initially Connected Graphs

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

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

Since at any time, every 2ec-class induces a subtree of $ST(G)$, and since the tree $ST(G)$ can be constructed in advance (i.e., the tree is not built on-line), we can use the Union-Find algorithms of [13] to maintain these classes: this algorithm runs in $O(n+m)$ time for m Finds and n nodes for this special case of the Union-Find problem. (It runs on a RAM but not on a pointer machine with this time complexity.) Moreover, as remarked in Subsection 3.7, a Find can be performed in $O(1)$ worst-case time.

We give the algorithms in case the initial graph is a tree. Consider graph G that initially is a tree (without additional edges). The initialisation of the data structure we use is as follows: implement the tree as a rooted tree and initialise the Union-Find structure of [13] accordingly. For each set name, the *father* pointer is set to the father edge of the (only) node in the set. 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. (Therefore, we may actually omit these additional *father* pointers for set names, since in the rooted tree the father relation is already implemented. In that case we may omit the setting of these *father* pointers in the procedure below, too.)

Now, procedure $insert_2$ as given in Figure 6.4 can be used for the insertion of an edge (e, x, y) again, with the following marginal changes. Firstly, the test $c(x) = c(y)$ and the case for $c(x) \neq c(y)$ (line 2-8) can be omitted, since the graph (always) is connected. Secondly, procedure $TreePath_2(2ec(x), 2ec(y), P, fath)$, given in Figure 6.6, must return the nodes on P in the right order from $2ec(x)$ to $2ec(y)$, and subsequently the Unions in line 11 of procedure $insert_2$ must be performed in the order of P (to meet the conditions in [13]), hence, starting from $2ec(x)$. (Note that this is not really a change, since we only specify the first node of the returned path P, and since we have not specified the order of the *for all* loop in line 11 of $insert_2$ before. Of course, we can also adapt line 11 such that P may start at either of the end nodes $2ec(x)$ and $2ec(y)$ instead.)

We consider the time complexity of the method as described for trees. As before, the tree path computations take $O(n)$ time altogether. However, now all Unions and m Finds only take $O(n + m)$ time. Finally, each insertion takes two Finds and $O(1)$ time, apart from the cost of path computations and Unions.

In case the initial graph is connected but not a tree, then we do the following. First obtain a spanning tree of the graph, and initialise the structure for this tree. Then insert the edges of the graph that are not in the tree by means of the operation $insert_2$. Then the actual insertions can be performed. Obviously, this initialisation can be done in $O(e_0)$ time, if e_0 is the number of edges in the initial graph. (Note that $e_0 \geq n$.)

Hence, we have the following theorem.

Theorem 6.3.2 *There exists a structure and algorithms that solve the 2ec-problem for graphs G that are initially connected, and that can be implemented on a RAM, such that the following holds. Starting from a connected graph G, m insert operations take $O(n + m)$ time, if n is the number of nodes in G. Any 2ec-comp(x) query and any nonessential insertion can be performed in $O(1)$ time. The initialisation can be performed in $O(e_0)$ time and the entire structure takes $O(n)$ space, where e_0 is the number of edges in the initial graph.*

6.4 Three-Edge-Connectivity

We now extend the results to the maintenance of the 3-edge-connectivity relation in a graph. We first introduce some notions and prove some properties for them.

In Subsection 6.4.1, we consider maintaining the 3-edge-connectivity relation in 2-edge-connected graphs, and, in Subsection 6.4.2, we consider the problem for general graphs.

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

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

Hence, $3ec(G)$ is the graph that is obtained if we contract each 3-edge-connected component into one representing (class) node (see Figure 6.7 if G is 2-edge-connected). By Lemma 6.2.8 (sub 3) it follows that $3ec(G)$ does not contain pairs of distinct class nodes that are 3-edge-connected in $3ec(G)$.

6.4.1 Two-Edge-Connected Graphs

Throughout this subsection, we suppose that the graph G is 2-edge-connected. By Lemma 6.2.8 (sub 3) for 2-edge-connectivity, every two distinct class nodes must lie on a common elementary cycle in $3ec(G)$. On the other hand, simple cycles cannot intersect in more than one class node, since $3ec(G)$ does not contain pairs of distinct class nodes that are 3-edge-connected. (The proof is as follows. Suppose that two different simple cycles S_1 and S_2 intersect in at least two different nodes. Take a maximal part P of one cycle, w.l.o.g. cycle S_1, that consists of at least three nodes, and that has no nodes in common with S_2 except for both its end nodes, say x and y. Then P together with the two paths between x and y obtained by "splitting" S_2 at x and y, yield 3 edge-disjoint paths between x and y, which gives a contradiction.) Therefore, it follows that each edge in $3ec(G)$ is on exactly one simple cycle in $3ec(G)$.

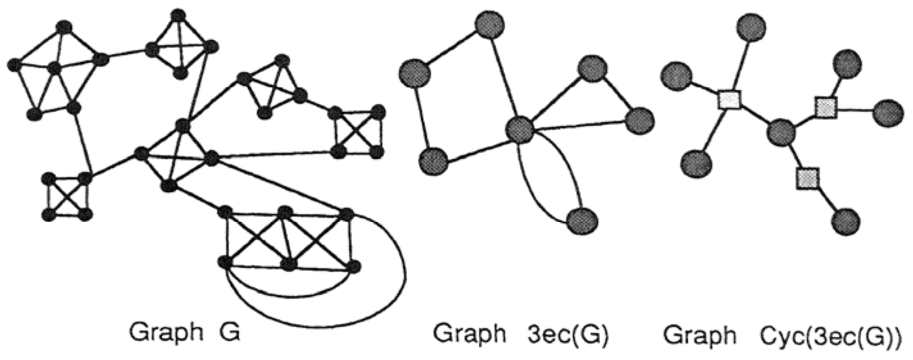
Let $Cyc(3ec(G))$ be the graph that is constructed from $3ec(G)$ as follows. Each non-trivial simple cycle (i.e., consisting of at least two distinct class nodes) is represented by a distinct node, called a *cycle node*. Let $cn(3ec(G))$ be the set of cycle nodes. For a cycle node s let $cycle(s)$ be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

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

Hence, $Cyc(3ec(G))$ consists of the class nodes and cycle nodes of $3ec(G)$, where a class node c is adjacent to a cycle node s in $Cyc(3ec(G))$ iff c lies on cycle s in $3ec(G)$ (i.e., c is "incident" with cycle s). Therefore, graph $Cyc(3ec(G))$ shows the incidence relation for class nodes and cycles. The structure of $Cyc(3ec(G))$ is illustrated in Figure 6.7, where the cycle nodes are drawn as boxes.

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

Figure 6.7: A 2-edge-connected graph G and the related graphs $3ec(G)$ and $Cyc(3ec(G))$.



Lemma 6.4.1 *Let G be a 2-edge-connected graph. Let $c, d \in 3ec(G)$. Let P be a path between c and d in $Cyc(3ec(G))$. Then there are 2 edge disjoint paths in $3ec(G)$ between c and d that only consist of edges from the cycles represented by the cycle nodes on P.*

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

Lemma 6.4.2 *Let G be a 2-edge-connected graph. Then $Cyc(3ec(G))$ is a tree.*

Proof. Let c and d be two class nodes in $Cyc(3ec(G))$. By Lemma 6.2.8 (sub 3), graph $3ec(G)$ is connected. Hence, there is a simple path P in $3ec(G)$ between class nodes c and d. We can construct a path in $Cyc(3ec(G))$ between c and d by the observation that each edge (e, f, g) on P is in some simple cycle s and hence that there are edges between f and s and between g and s in $Cyc(3ec(G))$. Hence, all class nodes are connected in $Cyc(3ec(G))$. On the other hand, each cycle node is adjacent to at least one class node. Hence, $Cyc(3ec(G))$ is connected.

On the other hand, suppose there is a nontrivial simple cycle in $Cyc(3ec(G))$. Then it consists of at least distinct 2 class nodes c and d and at least 2 cycle nodes. Lemma 6.4.1 yields that there are at least 4 edge disjoint paths between c and d in $3ec(G)$, since an edge in $3ec(G)$ is contained in precisely one simple cycle. Hence, by Definition 6.2.1 c and d are 3-edge-connected in $3ec(G)$. This contradicts Lemma 6.2.8 (sub 3). □

Graph observations

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

Suppose a new edge (e, x, y) is inserted in the 2-edge-connected graph $G =< V, E >$ $((e, x, y) \notin E)$. Because G is 2-edge-connected, we have two cases. If $3ec(x) = 3ec(y)$ then the edge connects two nodes that are 3-edge-connected in G, and, hence (by Lemma 6.2.8, sub 3), insertion of this edge does not affect the $3ec$-relation and the graphs $3ec(G)$ and $Cyc(3ec(G))$ remain unchanged. So, we are left to consider the other case: $3ec(x) \neq 3ec(y) \wedge 2ec(x) = 2ec(y)$. Then edge $(e, 3ec(x), 3ec(y))$ arises as an inserted edge in $3ec(G)$ and connects two class nodes $3ec(x)$ and $3ec(y)$ in $3ec(G)$.

Lemma 6.4.3 *Let G be a 2-edge-connected graph. Suppose edge $(e, 3ec(x), 3ec(y))$ is inserted in the graph $3ec(G)$. Then all the class nodes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ become 3-edge-connected in $3ec(G)$, while the other pairs of distinct class nodes in $3ec(G)$ stay only 2-edge-connected.*

Proof. Let P be the tree path in $Cyc(3ec(G))$ between the class nodes $3ec(x)$ and $3ec(y)$. Let c and d be any two class nodes on P. Now split P into 3 disjoint parts: part P_1 from $3ec(x)$ to c, part P_2 from c to d and part P_3 from d to $3ec(y)$. By Lemma 6.4.1, there exist 2 edge-disjoint paths Q_1 and Q_2 in $3ec(G)$ between c and d that only consist of edges from cycles represented by cycle nodes on P_2. Similarly, it follows from Lemma 6.4.1 that there exists a path R_1 from c to $3ec(x)$ that only uses edges from the cycles represented by cycle nodes on P_1, and a path R_2 from $3ec(y)$ to d only using edges from cycles represented by cycle nodes on P_3. Let Q_3 be the path $R_1, (e, 3ec(x), 3ec(y)), R_2$ from c to d. Then it follows that Q_3 has no edges in common with Q_1 and Q_2. Hence, by Lemma 6.2.1 c and d are 3-edge-connected.

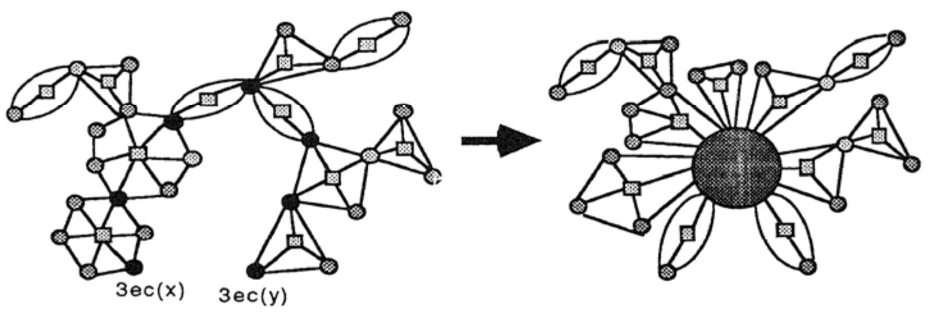
On the other hand, let c and d be 2 distinct class nodes in $3ec(G)$ such that c is not on P. Consider a cycle node r that is adjacent to class node c in $Cyc(3ec(G))$ such that r separates c from $3ec(x)$ and $3ec(y)$. (This node exists because r is not on P.) The deletion of the two edges in $3ec(G)$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

By Lemma 6.4.3 all class nodes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ become 3-edge-connected in $3ec(G)$ and, hence, by Lemma 6.2.8 (sub 3) all the corresponding classes form a new class. The update can be performed in the following way:

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

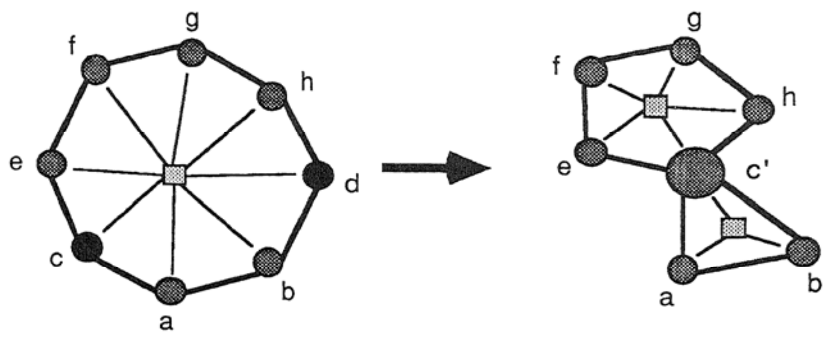
The update is illustrated in Figure 6.8. The cycle tree changes as follows. Consider

Figure 6.8: Adapting the tree path between $3ec(x)$ and $3ec(y)$.



the simple cycle s and the class nodes c and d ($c \neq d$) such that s, c and d are on P and $c, d \in cycle(s)$. Then classes c and d are joined into the new class c'. The original simple cycle s splits into two "smaller" simple cycles, each one consisting of the class node c' for the new class and of the class nodes of one of the two parts of the cycle *between* c and d, in the same cyclic order (cf. Figure 6.9). One or both of these two new cycles may be a trivial cycle: i.e., consisting of class node c' only (which is the case if one of the parts mentioned above of the cycle is empty).

Figure 6.9: Splitting cycles.



The algorithms

In our algorithms we represent the collection of 3ec-classes of a graph G by a Union-Find structure (cf. Subsection 6.2.3), where the name of each class is the class node of that class (i.e., a Find on an element of a class returns the class node related to that class). Therefore we (may) denote $Find_{3ec}(x)$ by $3ec(x)$ too (cf. Subsection 6.2.3).

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

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

To implement the relation between a class representative $repr(c, s)$ and the corresponding edge between c and s in $Cyc(3ec(G))$, we use a Circular Split-Find and a Union Find structure, from which the end nodes c and s of that edge can be obtained. (Hence, in contrast to the representation of ordinary edges in the graph G, a class representative $repr(c, s)$, which represents an edge between c and s in $Cyc(3ec(G))$, does not have direct pointers to the end nodes c and s of that edge.) These structures are used in the following way. A class representative $repr(c, s)$ is an element of the so-called *cycle list* for cycle node s and of the so-called *representative set* for class node c, which are given as follows. The *cycle list for a cycle node* s contains the class representatives $repr(c, s)$ of all class nodes c in cycle s in $3ec(G)$ in the order in which these class nodes occur in cycle s. The collection of cycle lists is implemented as a Circular Split-Find structure (cf. Subsection 6.2.3), where the name of a cycle list for cycle node s is s itself. (Hence, a Find on an element of that list returns node s.) We denote a Circular Split or a Find in this structure by $Split_{cyc}$ or $Find_{cyc}$ respectively. The *representative set for a class node* c is the set that contains the representatives $repr(c, s)$ for all cycle nodes s for which $repr(c, s)$ exists. The collection of representative sets is implemented as a Union-Find structure, where the name of the representative set for class node c is c itself. (Hence, a Find on an element of that set returns the node c.) In the algorithms we perform a Union on two representative sets (of class representatives) for two class nodes c and d iff the corresponding classes c and d (of ordinary nodes) in the graph are joined. Therefore we will not make these joinings explicit in our algorithms. We denote a Find in this structure by $Find_{class}$. Hence, the operations $Find_{class}$ and $Find_{cyc}$ on a class representative yield the end nodes of the edge that is related to it.

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

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

Now, edge insertions can be handled as follows. Suppose edge (e, x, y) is inserted in graph $G =< V, E >$ with $(e, x, y) \notin E$. Then after inserting this edge in the incidence lists, procedure $insert_3$ (given in Figure 6.10) performs the updates as follows. We distinguish the two cases. If $3ec(x) = 3ec(y) \wedge 2ec(x) = 2ec(y)$ (line 2), then nothing needs to be done. Otherwise, we have $3ec(x) \neq 3ec(y) \wedge 2ec(x) = 2ec(y)$ (line 3-7). All class nodes on the tree path in $Cyc(3ec(G))$ between $3ec(x)$ and $3ec(y)$ become 3-edge-connected in $3ec(G)$. The procedure first determines this tree path (line 4) and then adapts the cycle tree accordingly by first splitting all cycles on P (line 5) and then joining all classes on P (line 6). This is done as follows.

1. *The computation of the tree path (line 4).* In line 4, the tree path P between $3ec(x)$ and $3ec(y)$ is obtained by traversing the root paths of $3ec(x)$ and $3ec(y)$ alternatively like in Section 6.2. This is performed by the call of procedure $TreePath_3$ which is given in Figure 6.11. This procedure returns the tree path P. Moreover, it detects whether the nearest common ancestor *top* of $3ec(x)$ and $3ec(y)$ in $Cyc(3ec(G))^R$ is a cycle node (if this is the case, $topcyc = true$ is returned) and it returns the class representative $father(top)$ in the parameter

toprepr.

2. *The splitting of cycles* (line 5) is performed by procedure *AdjustCycles*, which is given in Figure 6.12. The strategy is as follows: let c, s and d be three consecutive nodes on P,where s is a cycle node. Note that, since $Cyc(3ec(G))^R$ is a rooted tree, either c or s contains a pointer to $repr(c, s)$ and that the same holds for d, s and $repr(d, s)$. Therefore, these records can be obtained by using these pointers. The cycle list for cycle node s is split into two parts: the part from $repr(c, s)$ up to but excluding $repr(d, s)$ and the part from $repr(d, s)$ up to but excluding $repr(c, s)$. This is done by a Circular Split operation at those two elements. (Each part forms a new cyclic list, for which a new cycle node is generated.) If one (or both) of these two lists appears to correspond to a trivial cycle (i.e., it contains only one element), then this list is deleted. Note that if s' and s'' are the cycle nodes resulting from the Circular Split, then the class representatives denoted by $repr(c, s')$ and $repr(d, s'')$ (after the Circular Split) actually are the class representatives formerly denoted by $repr(c, s)$ and $repr(d, s)$. Each resulting cycle node s' or s'' (which can be obtained by the Find$_{cyc}$ operation) gets a father pointer to $repr(c, s')$ or $repr(d, s'')$ respectively.

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

4. Finally, the $father(c')$ value for the newly formed class c' is assigned by procedure *AdjustFathers* (line 7). Procedure *AdjustFathers* is given in Figure 6.13. The father values are updated according to the following observations.

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

   - *top* is a class node that is the root. Then the new class node c' will be the root of the new tree.
   - *top* is a class node that is not the root. Then the father of *top* in $Cyc(3ec(G))$ is a cycle node s for which no Circular Split is performed on its cycle list. Then s must be the father of the new class c'.
   - *top* is a cycle node (that is not the root). Then let a be the class node that is the father of *top* in $Cyc(3ec(G))^R$. Note that the father of the new class c' must be the cycle node s that contains both a and c': this cycle node s may be different from *top* since a Circular Split just generates two new cyclic lists with two names (being the resulting cycle nodes).

Hence, the father of c' is s and the father of s is a. Note that s can be obtained by $Find_{cyc}(toprepr)$. Then $repr(c', s)$ can be obtained by $father(s)$, since all involved cycle nodes have their father pointers to the representative of class c'. (See Figure 6.14 that shows the results of the four successive parts as distinguished above for this case (top is a cycle node), where a class representative together with the father pointer to that class representative is indicated as a directed edge as follows: e.g., if the father pointer of node t points to $repr(d, t)$, then this is indicated as the directed edge from t to d.)

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

Figure 6.10: Procedure $insert_3((e, x, y))$.

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

Complexity

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

Concluding the above observations we obtain the following property.

Figure 6.11: Procedure $TreePath_3(c, d,$ **output** $P, toprepr, topcyc)$.

---

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

(description)

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

Figure 6.12: Procedure $AdjustCycles(P)$.

---

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

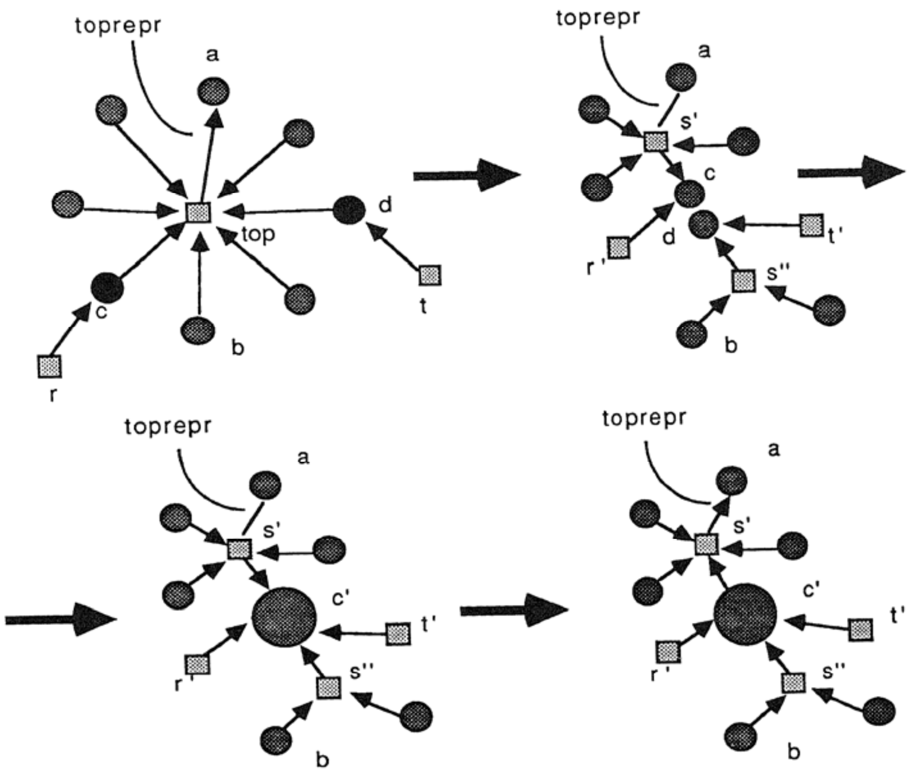
- obtain the class representatives $repr(c, s)$ and $repr(d, s)$ by means of the fields $father(c)$, $father(d)$ and $father(s)$.
- split the cycle list for cycle node s into two parts by a Circular Split $Split_{cyc}(repr(c, s), repr(d, s))$: the part from $repr(c, s)$ up to but excluding $repr(d, s)$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

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

Figure 6.13: Procedure *AdjustFathers*$(3ec(x), toprepr, topcyc)$.

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

Figure 6.14: Changes in the father relation.



Property 6.4.4 *A call of procedure $insert_3$ in a 2-edge-connected graph takes $O(1 + d)$ steps plus the time needed to join 3ec-classes and to perform Circular Splits, where d is the value by which the number of classes decreases.*

Observe that there exist $O(n)$ different classes during all insertions. Moreover, initially for the given 2-edge-connected graph, there are at most $2(n-1)$ class representatives, since each class representative corresponds to an edge in $Cyc(3ec(G))$, the tree $Cyc(3ec(G))$ contains at most n class nodes, leaves of the tree are class nodes and edges connect cycle nodes and class nodes only. Hence, we obtain the following lemma, which e.g. can be used in Chapter 8.

Lemma 6.4.5 *Given a 2-edge-connected graph G of n nodes with a cycle tree, there exists a data structure for the 3ec-problem (that also maintains a cycle tree), such that the following holds. The total time for m insertions and queries is $O(m + n)$ 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.*

6.4.2 General Graphs

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

Note that for detecting the 3ec-classes, it suffices to detect the 3ec-classes inside the 2-edge-connected components (cf. Lemma 6.2.6). Therefore, our algorithms for general graphs maintain the 2ec-classes by using the previous solutions for 2-edge-connectivity (Section 6.3) and maintain the 3ec-classes by using the previous solutions for 3-edge-connectivity within 2-edge-connected components (Subsection 6.4.1).

The representation of a graph consists of the representations and the data structures of both Section 6.3 and Subsection 6.4.1 (for 2-edge-connectivity and 3-edge-connectivity respectively). Hence, there is a cycle tree (of 3ec-class nodes) for each 2-edge-connected component.

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

Suppose edge (e, x, y) is inserted in graph G, yielding graph G'. Then the updates are performed by procedure $INSERT$ (given in Figure 6.16), that is based on procedure $insert_2$ (cf. Figure 6.4). The procedure works as follows. Three cases are considered (cf. Figure 6.16).

If $c(x) \neq c(y)$ (line 2-3), then the 2ec-classes do not change. Therefore the computations performed in $insert_2$ for this case (i.e. line 2-8 of Figure 6.4) suffice here.

Otherwise, if $2ec(x) = 2ec(y)$ (line 23-24) then the edge is inserted inside a 2-edge-connected component. Therefore procedure $insert_3$ (Figure 6.10) is performed, that deals with 3ec-classes within a 2-edge-connected component.

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

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

We consider the updates more precisely.

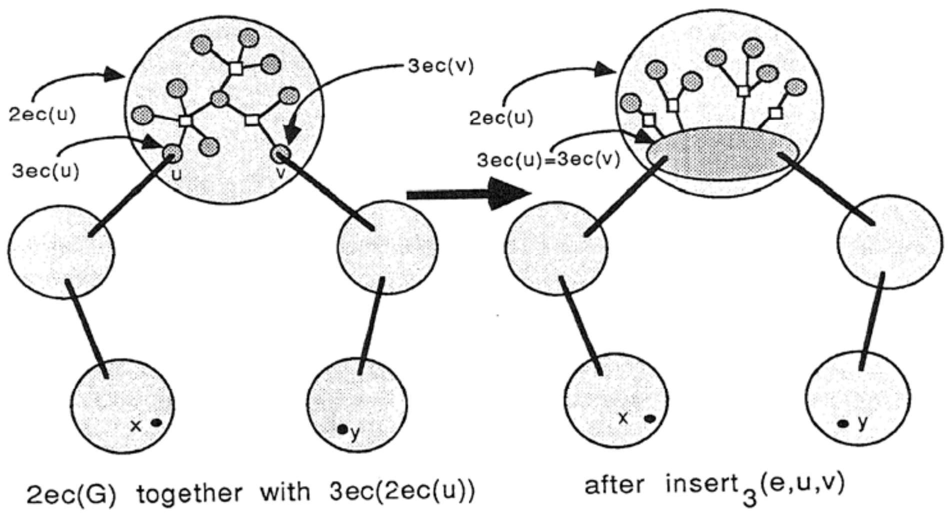
Consider the changes of the 3ec-classes that occur in 2ec-classes on P_2. Consider a particular 2ec-class C on P_2 in $2ec(G)$. Let u and v be the two nodes in C that are end nodes of interconnection edges on CS_2. Then there is a new path between u and v in G' that does not intersect with C except for u and v, where such a path did not exist in G before. Hence, considered within C only, this corresponds to inserting a temporary edge between the nodes u and v (cf. Figure 6.15). Therefore, we can first insert a temporary edge between u and v to update the 3ec-classes (and hence the cycle tree) inside C (causing u and v to be in the same 3ec-class) and then perform all remaining updates w.r.t. the insertion of (e, x, y).

Now suppose all these "local" insertions are performed for the 2ec-classes on P_2. Then the two edges in CS_2 that are incident with one 2ec-class C on P_2 have their end nodes in the same (updated) 3ec-class in C. Call such a 3ec-class the interconnection 3ec-class. Then all these interconnection 3ec-classes form a new cycle r. Hence, all the updated cycle trees for the 2ec-classes on P_2 (that result from the local insertions of temporary edges) must be linked to the new cycle node r. All these cycle trees now form one new tree together.

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

First, the tree path P_2 in $2ec(G)$ is computed together with the corresponding sequence CS_2 that also contains the interconnection edges and the edge (e, x, y). Note

Figure 6.15: Tree path versus temporary edges.



that these interconnection edges can easily be obtained from the father fields of all class nodes that are on P_2. (In fact this sequence can be obtained in $TreePath_2$ instead of P_2.) Then for each pair of nodes u and v that are in a 2ec-class on P_2 and that are end nodes of two consecutive edges in CS_2 (where u and v may be equal), procedure $insert_3((e', u, v))$ is executed to adapt the 3ec-classes inside class $2ec(u)$ by means of a temporary edge (e', u, v) that only exists during this execution (cf. Figure 6.15). Moreover, the cyclic list CS_3 of the interconnection 3ec-classes is extended with the (updated) class node $3ec(u)$ $(= 3ec(v))$. Finally, if the 2ec-class $2ec(u)$ is not the largest class c_0 "on" P_2 (i.e., it does not contain the largest number of nodes), then class node $3ec(u)$ is made to be the new root of the (updated) cycle tree in which it is contained (inside the 2ec-class $2ec(u)$) by reversing the root path of node $3ec(u)$. This is done by procedure $ReverseRootPath_3$, which works similar to procedure $ReverseRootPath_2$ with obvious adaptations.

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

The Union-Find and the Circular Split-Find structures that we use here are the basic structures that take $O(n. \log n)$ time altogether for all the Unions/Splits on n elements and that take $O(1)$ time for each Find (cf. Subsection 6.2.3).

Complexity

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

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

Figure 6.16: Procedure $INSERT((e, x, y))$.

```
**procedure** $INSERT((e, x, y))$;
**if** $c(x) \neq c(y)$
    $\longrightarrow insert_2((e, x, y))$
[] $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$
    $\longrightarrow Treepath_2(2ec(x), 2ec(y), P_2, fath)$;
        let $c_0$ be the class node on $P_2$ with the largest number of nodes in its class;
        construct the cyclic sequence $CS_2$ from $P_2$ by inserting the
        interconnection edges between consecutive class nodes on $P_2$ and by
        inserting edge $(e, x, y)$ between class nodes $2ec(x)$ and $2ec(y)$;
        $CS_3 := \emptyset$ ;
        **for all** end nodes $u, v$ of consecutive edges in $CS_2$ with $2ec(u) = 2ec(v)$
                $\longrightarrow insert_3((e', u, v))$, for a temporary edge $(e', u, v)$;
                    insert $3ec(u)$ in $CS_3$;
                    **if** $2ec(u) \neq c_0 \longrightarrow ReverseRootPath_3(3ec(u))$
                    [] $2ec(u) = c_0 \longrightarrow cc_0 := 3ec(u)$
                    **fi**
        **rof**;
        **for all** $C \in P \setminus \{2ec(x)\} \longrightarrow Union_{2ec}(C, 2ec(x))$ **rof**;
        $father(2ec(x)) := fath$;
        make a new cycle $r$ of the class nodes in $CS_3$ in the same cyclic order
        in which they appear in $CS_3$, where $cc_0$ is the father of the new cycle
        node $r$ and the father of the class nodes in $CS_3 \setminus \{cc_0\}$ is $r$
[] $c(x) = c(y) \wedge 2ec(x) = 2ec(y)$
    $\longrightarrow insert_3((e, x, y))$
**fi**
```

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

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

By Property 6.4.4 the execution of $insert_3$ takes $O(1+d)$ time apart from the time needed to join 3ec-classes and to perform Circular Splits, where the number of 3ec-classes decreases with d. Firstly note that the number of calls of procedure $insert_3$ in line 12 of procedure $INSERT$ is $O(|CS_2|)$. Therefore we can charge $O(1)$ time per call to line 5 without increasing the order of time complexity too. A similar remark can be made for the call of procedure $insert_3$ in line 24: $O(1)$ time can be charged to procedure call $INSERT$. Hence, we only need to consider the remaining part $O(d)$ of the cost $O(1+d)$ of a call of $insert_3$. Since initially there are n 3ec-classes and since the number of classes only decreases and never increases, it follows that the remaining time $O(d)$ spent by procedure $insert_3$ (where d is the decrease in the number of 3ec-classes) adds up to $O(n)$ for all calls together. The Union-Find structure and the Circular Split-Find structure take $O(n.\log n)$ time for all Unions and Splits, since there are $O(n)$ elements occurring in these structures.

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

We have thus proved the following theorem.

Theorem 6.4.6 *There exist a data structure and algorithms that solve the 3ec-problem, which can be implemented as a pointer/* $\log n$ *solution, such that the following holds. Starting from an empty graph* $G = < V, \emptyset >$*, the insertion of* e *edges take* $O(n \log n + e)$ *time altogether, where* n *is the number of nodes in* G*. The queries 2ec-comp(x) and 3ec-comp(x) can be answered in* $O(1)$ *time. The data structure can be initialised in* $O(n)$ *time and uses* $O(n)$ *space.*

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

6.5 Concluding Remarks

In this chapter, we have presented algorithms for maintaining the 2- and 3-edge-connected components in a graph under the insertion of edges and vertices. The insertion of e edges costs $O(n.\log n + e)$ time in total, while at any moment connectivity queries can be answered in time $O(1)$. The time bound for the insertions together with the cost for queries can be improved to $O(n + m.\alpha(m, n))$, where m is the total number of queries and edge insertions and n is the number of nodes, using a number of sophisticated data structuring techniques. These results will be presented in the next chapters, since the additional data structures are rather complicated. Moreover, the same time bounds will be achieved for 2- and 3-vertex-connectivity.

Chapter 7

Fractionally Rooted Trees

7.1 Introduction

A major problem in obtaining better time bounds for the connectivity problems is obtaining paths in trees if, in addition, trees may be linked from time to time. In the previous chapter, we used ordinary rooted trees for obtaining tree paths between nodes. However, if we use rooted trees, where the father relation is implemented accordingly, then, each time when two trees are linked, the father relation must be adapted in one of the trees. If this is done in the way of a simple Union-Find structure ($UF(1)$, cf. Subsection 3.3.1 or [1]), viz., by processing the smallest tree, then this takes $O(n. \log n)$ time for n nodes.

Another problem is that a tree may be changed itself ("internal change"): cf. the cases of inserting an edge in the trees that are used for the 2ec-problem or the 3ec-problem. However, in the previous chapter, we have seen that for a spanning tree ST of graph G, each 2ec-class of nodes induces a subtree of ST. Moreover, two different 2ec-classes can have at most one node in common. Therefore, 2ec-classes partition ST into subtrees such that two different subtrees have at most one node in common, but they do not have edges in common. Therefore, we can express 2ec-classes in terms of trees with edge classes, where trees are linked from time to time, and where the only "internal change" consists of joining classes of edges. This partially motivates the description that we will give below.

In this chapter, we present a data structure, called the *fractionally rooted tree.* It allows the linking of trees, while tree paths can be obtained efficiently at any moment. Moreover, it allows classes of edges that induce subtrees to be treated as single units, i.e., for a tree path that runs through some subtrees that are induced by classes of edges, it is not needed that every node or edge on the tree path is returned, but one representative for each subtree suffices. The data structure can be maintained in $O(n + m.\alpha(m, n))$ time for n nodes and m operations, where the allowed operations are as follows. First, there is a query of the form: given two nodes of a tree, return two tree edges that are in the same class of edges and such that each node is incident with at least one edge. Second, one can join all edge classes that occur on the tree path between two nodes into a new edge class (destroying these old edge classes). Third, one can link two trees. (Actually, the operation for joining edge classes is split into two different operations that we will both need in Chapter 8, but this will be considered later.)

In Chapter 8, fractionally rooted trees will be used to obtain optimal solutions for both the 2ec-problem and the 3ec-problem, while in Chapter 9, we will also use the data structure to obtain an optimal solution for the problem of maintaining the 2-vertex-connected components.

This chapter is organised as follows. In Section 7.2, we describe the problem, i.e., we describe the trees and operations for which the data structure can be used, and we give observations and ideas that we will use in our data structure. In Section 7.3, we describe a data structure called a division tree, that forms an essential part of the fractionally rooted tree. In Section 7.4, we present the fractionally rooted trees. Finally, in Section 7.5, we determine the time complexity of the operations.

7.2 Problem Description and Observations

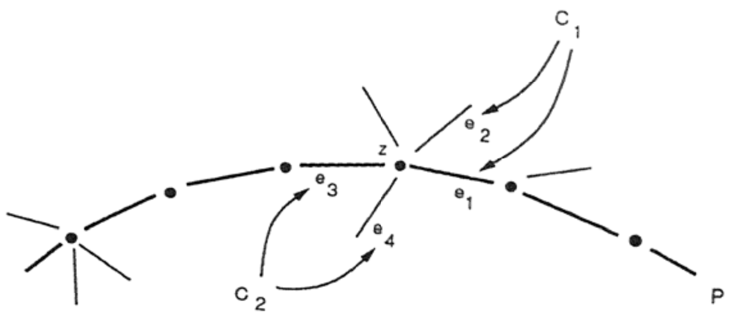
7.2.1 Problem Description

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

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

Let x and y be two nodes in the same tree of F. Let P be the tree path between x and y. We call a node x on tree path P an *internal node* of P if it is incident with two edges of P that are in the same edge class. We call a node of P a *boundary node* otherwise. Hence, a boundary node is either one of the end nodes x or y of P, or it is a node for which its two incident edges on P are in different classes. A *boundary edge set* for a boundary node z on P is a set of (0, 1 or 2) edges that contains for each edge e of P that is incident with z, exactly one edge e' which is incident with z and which is in the same edge class as e. (See Figure 7.1, 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 boundary edge sets for z on P.) A *boundary list* for the two nodes x and y is a list consisting of the boundary nodes of P, where each boundary node has a sublist that contains

Figure 7.1: Boundary edge sets



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

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

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

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

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

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

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

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

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

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

7.2.2 Observations and Ideas

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

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

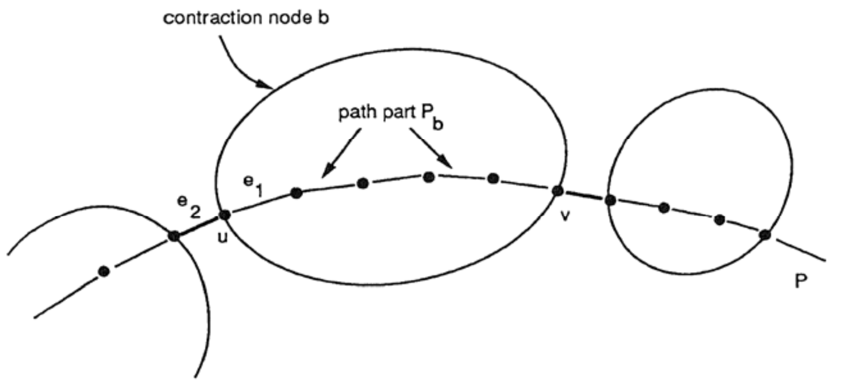
A boundary list B between two nodes x and y in T can now be obtained as follows.

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

that is contained in b. For, either an end node of P is contained in b or the edges of P' (in T') that are incident with b are in the different classes. In the latter case, the originals of these two edges (on P) are in different edge classes, and hence there is at least one node of P contained in b of which the two incident edges of P are in different classes. Therefore, each boundary node of P is contained in a boundary node of P' and each boundary node of P' contains a boundary node of P.

Now, suppose that b is a boundary node of P'. Consider the part P_b of P inside contraction node b. We consider the relation between boundary nodes in P_b and P (see Figure 7.2). Trivially, a boundary node of P that is contained in P_b is a boundary node of P_b too. Now, let z be a boundary node of P_b. Let the end nodes of P_b be u and v. If $z \notin \{u, v\}$, then z is a boundary node of P too, and a boundary edge set for z on P_b is a boundary edge set for z on P. If $z \in \{u, v\}$ and $u \neq v$, then z is an end node of P_b and hence a boundary edge set for z on P_b contains only one edge, say edge e_1. Let e_2 be the original of the edge in a boundary edge set for b on P' that is incident with z, if e_2 exists (i.e., if $z \notin \{x, y\}$). If e_2 exists, and if e_1 and e_2 are in the same edge class, then z is an internal node of P. Otherwise, z is a boundary node; then a boundary edge set for z on P_b extended with e_2 (if e_2 exists) is a boundary edge set of z for P. Finally, if $z = u = v$ then P_b consists of node z only. Hence, a boundary edge set for z $(= u = v)$ on P_b consists of z with an empty sublist. In that case the original(s) of the edge(s) in a boundary edge set for b on P' form a boundary edge set for z on P. (Since otherwise, b would not be a boundary node of P'.)

Figure 7.2: Considering a part P_b of P.



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

We will present fractionally rooted trees and algorithms for maintaining them that are based on the above observations.

7.3 Division Trees

7.3.1 Description of the Data Structure and the Operations

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

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

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

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

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

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

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

We describe some further aspects of the division trees.

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

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

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

The extended subtrees in a division tree have the following additional implementation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- For each edge class C in $extree(b)$ for some $b \in CN(T)$, there are the following pointers:
  - pointer $max(C)$ to the m-marked edge of C.
  - pointer $ext(C)$ to an external edge in it (if there exists any).
  - pointer $edge(C)$ to the c-marked edge in it (if it exists).

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

7.3.2 Implementation of the Operations

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

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

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

basic-boundary(x, y): Note that x and y are in the same elementary subtree. If $x = y$ then return the boundary list BL consisting of node x with an empty sublist. Otherwise, $x \neq y$ and the following is done. The boundary list BL is obtained as follows. First the boundary nodes (together with boundary edge sets) for the root paths of x and y are partially computed: viz., two boundary lists $s(x)$ and $s(y)$ are computed as follows. The two lists $s(x)$ and $s(y)$ start with x and y with empty sublists respectively. Then the two lists $s(x)$ and $s(y)$ are stepwisely computed in an alternating way until a node top has been visited by both computations. A computation step for sequence $s(x)$ (or $s(y)$) is as follows: obtain the father edge (e, z, z') of the last node z in the sequence, (if any, otherwise skip the rest of the step), insert the edge in the sublist of z, obtain the edge $max(class(e)) = (e', u, v)$ and obtain the father node of e' (being u or v); then insert the father node at the end of the list and insert edge (e', u, v) in its sublist. (The stop condition can be checked by marking all nodes that are visited: it becomes true if a node is visited that is already marked. After the traversals the nodes are unmarked. Cf. Subsection 6.3.2.) (Note that now $s(x)$ and $s(y)$ are boundary lists for their end nodes.) (It follows that each edge in the sublist of a node is either its father edge or it is an m-marked edge.)

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

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

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

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

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

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

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

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

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

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

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

7.4 The Fractional Structure

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

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

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

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

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

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

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

Note that from the above description the following follows.

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

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

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

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

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

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

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

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

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

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

The structures FRT(i) are defined inductively in a way similar to Chapter 3. 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.

7.4.1 The Structure FRT(1)

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

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

For each tree name s we have a parameter $weight(s,1)$ that contains the number of nodes in $tree(s,1)$: $weight(s,1) = |tree(s,1)|$ (Note that we count the nodes of $tree(s,1)$, cf. Notation 2.1.1.)

We give the algorithms for the operations.

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

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

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

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

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

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

If FRT(1) is used directly on F_0 (i.e., $F_1 = F_0$), then the initialisation for some (sub-)collection of nodes in singleton trees is as follows. Relate a tree name s to each singleton tree. For each node x with name s for the singleton tree consisting of x, the following initialisation is performed: $contr(x) = s$, $nodes(s) = \{x\}$, $weight(s,1) = 1$. (Note that the insertion of a singleton set consisting of a newly created node can easily be performed in this way too.) If we want to initialise the structure for some a forest F_0 not necessarily consisting of singleton trees, where there is a list of the names of the existing edge classes and for each name there is a sublist with the edges in the corresponding class, then this can be performed as follows. First the forest is initialised as a forest of division trees, where each division tree contains exactly one elementary subtree, viz. the tree itself. This is done in the way described in Section 7.3. Hence, for each tree there is exactly one (new) contraction node. Then the contraction node s is taken to be the tree name in FRT(1) and $weight(s, 1) =$ [the number of nodes in the tree].

7.4.2 The Structure FRT(i) for i>1

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

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

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

$$2.A(i, lowindex(s, i)) \leq weight(s, i). \tag{7.1}$$

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

Two cases are distinguished.

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

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

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

  $$\mid nodes(b) \mid \geq 2.A(i, lowindex(s, i)). \tag{7.2}$$

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

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

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

Note that every edge class in $extree(b)$ for some $b \in CN(tree(s, i))$ now contains at most one c-marked edge, which is seen as follows. Let (e, x, y) be a c-marked edge in $extree(b)$, where $contr(x) = b$ and $contr(y) = c$. Let (e, x, y) be contained in class C of $extree(b)$. Then either edge (e, c, d) is the father edge of contraction node c or the edge side $(e, c, d)_c$ is m-marked. By applying the observations of Section 7.3 to $tree(s, i-1)$, there is not another edge in the local edge class of $(e, c, d)_c$ in $tree(s, i)$ that is incident with c and that has one of these two properties. Hence, there is not another c-marked edge in class C.

We give the algorithms for the operations (intermixed with comments). Note that, by (7.1), $lowindex(s, i) \geq 0$ implies that $tree(s, i)$ consists of at least 2 nodes and hence there exists a contraction node c at layer $i - 1$ (hence, $c \neq s$).

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

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

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

- $lowindex(s,i) = lowindex(t,i) \wedge newweight < 2.A(i, ls+1)$. Then *basic-external-insert*$((e,x,y))$ is executed. (Hence, edge (e,x,y) is inserted as an external edge between x and y.)

  Let $c = contr(x)$ and $d = contr(y)$. (Then $c \neq s$ and $d \neq t$ since $0 \leq newweight < 2.A(i, ls+1)$ implies $ls \geq 0$. Hence, c and d are nodes on layer $i-1$.) A new edge (e,c,d) is created. Then $orig(e,c,d) := (e,x,y)$ and $origo_0(e,c,d) := origo_0(e,x,y)$. Subsequently a recursive call $link((e,c,d), s, t, i-1)$ is performed. (This is to link the contractions $tree(s, i-1)$ and $tree(t, i-1)$; then one of the above cases occurs on a layer j with $j < i$.) Then all the affected edges in layer $i-1$ are obtained (i.e., the edges processed by a call *basic-integrate* or *basic-internal-insert* on layer $i-1$, which hence may change the father relations and m-marks of these edges). (Note that these edges can easily be obtained by having the recursive call $link(i-1)$ returning a list of all these edges, where hence the same must be done by calls *basic-integrate* and *basic-internal-insert*.)

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

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

Perform $candidates(x,y,i)$ yielding edges e_x and e_y. If $class_0(origo_0(e_x)) = class_0(origo_0(e_y))$ (i.e., e_x and e_y are in the same global edge class and hence $x \sim y$), then the nodes x and y are put in BL with the edges e_x and e_y in their sublists.

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

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

Otherwise we have $c \neq d$ and the following is done (corresponding to the observations of Subsection 7.2.2). A recursive call $boundary(c,d,i-1)$ is performed that outputs a boundary list BB for c and d, consisting of nodes and edges of the contraction tree in layer $i-1$.

For each node f in BB a list $bl(f)$ is computed as follows (according to the observations in Subsection 7.2.2). First the original(s) in layer i of the edges in the sublist of f are obtained. Let these edge(s) be the edge (e_1, z_1, u) and (if $zz \notin \{c, d\}$) the edge (e_2, z_2, v), where z_1 and z_2 are the nodes in which these edges are incident with $tree(f)$. If $f = c$ or $f = d$ then let $z_2 = x$ or $z_2 = y$ respectively. Then in $subtree(f)$ a boundary list $bl(f)$ for z_1 and z_2 is computed by a call $basic\text{-}boundary(z_1, z_2)$. The sublists of the nodes z_1 and z_2 in $bl(f)$ are extended with edge (e_1, z_1, u) and (if $f \notin \{c, d\}$) edge (e_2, z_2, v) respectively. Finally, a node $z \in \{z_1, z_2\}$ for which the sublist of z in sequence $bl(f)$ consists of two edges that are in a same edge class, is deleted from the sequence (together with its sublist).

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

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

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

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

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

We are left with the problem of how to obtain and store the values *weight*, *lowindex* and the Ackermann values. All these values depend on both the tree name and the layer number. The values $lowindex(s,j)$ and $weight(s,j)$ for all relevant j are stored in a list of records: each records contains these values for some layer j. The tree name s contains a pointer to the begin and the end of the list of records. (The end of the list is the record for layer i if for the FRT(i) structure we have $F_i = F_0$, i.e., FRT(i) is used in some environment not being a part of a FRT($i+1$) structure.) For further details and for the problem of how to obtain Ackermann values we refer to Chapter 3. The approach is similar, where the pointers *contr* in the structures FRT(i) correspond to the pointers *father* in the structures UF(i).

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

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

If FRT(i) is used directly on F_0 (i.e., $F_i = F_0$), then the initialisation for some (sub-)collection of nodes in singleton trees is as follows. Relate a tree name s to each singleton tree. For each node x with name s for the singleton tree consisting of x, the following initialisation is performed: $contr(x) = s$, $nodes(s) = \{x\}$, $weight(s,i) = 1$, $lowindex(s,i) = -1$. (Note that the insertion of a singleton set consisting of a newly created node can easily be performed in this way too.) If we want to initialise the structure for some a forest F_0 not necessarily consisting of singleton trees, where there is a list of the names of the existing edge classes and for each name there is a sublist with the edges in the corresponding class, then this can be performed as follows. First the forest is initialised as a forest of division trees, where each division tree contains exactly one elementary subtree, viz. the tree itself. This is done in the way described in Section 7.3. Hence for each tree there is exactly one (new) contraction node. For a singleton tree, the contraction node is taken to be the tree name s in FRT(i) and then $weight(s,i) := 1$, $lowindex(s,i) := -1$. For a tree T that is not a singleton tree, let c be its (new) contraction node c created by the initialisation as division tree. Relate a tree name s to tree T. Then make $nodes(s) = \{c\}$, $contr(c) = s$, $weight(s,i-1) = 1$ $weight(s,i) =$ [the number of nodes in the tree] and $lowindex(s,i) = lowindex(s,i-1) = -1$.

7.5 Complexity of FRT(i)

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

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

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

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

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

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

We obtain the following result.

Lemma 7.5.1 *Let a FRT(i) structure for a forest with n nodes be given. The structure and the algorithms can be implemented as a pointer/* $\log n$ *solution such that the following holds. An essential sequence in FRT(i) (cf. Section 7.2) needs a total of* $O(n_e.a(i,n_e))$ *steps (*$i \geq 1$*,* $n_e \geq 2$*), where* n_e *is the number of nodes that are not contained in singleton trees after the execution of the sequence.*

Note in the lemma that $n_e \leq n$. The proof of the lemma is given in Subsection 7.6.

7.6 Proof of Lemma 7.5.1.

Lemma 7.5.1 is proved by induction in a way similar to the proof in Chapter 3. We consider the *net cost* of the basic operations, i.e., the cost of the operations except for the cost of Union operations and creations of new singleton sets in Union-Find structures.

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

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

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

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

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

We now consider the complexity of the structures FRT(i). Like in Chapter 3 we do not need to consider the complexity of storing and obtaining the information for each layer that exists for a tree name, since this can easily be charged to other operations by increasing their cost with $O(1)$ time per operation.

We show that an essential sequence in FRT(i) (of procedure *link*(i) *pathrep*(i) and *joinclasses*(i)) takes $O(n.a(i,n))$ steps on n nodes. Moreover, we show that the number of times that an edge becomes affected in FRT(i) (by procedure *basic-internal-insert* or *basic-integrate*, cf. Section 7.3) is at most $a(i,n)$.

We prove this by calculating the *net cost* of the procedures *link*(i), (essential) *boundary*(i) and *joinclasses*(i), the cost of unions and creations of singleton sets in layer i and the cost of essential recursive calls: for each call of the procedures *link*(i), (essential) *boundary*(i) and *joinclasses*(i) in layer i we do not account for steps performed in an essential recursive call or steps regarding Unions or creations of new singleton sets. Here, an *essential recursive call* is any recursive call of these procedures with the restriction that recursive *boundary* calls are essential.

7.6.1 FRT(1)

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

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

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

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

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

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

Unions: There are at most $2n$ edge sides in layer 1. By Lemma 3.4.3, the time for the joinings and insertion of edges in layer 1 is at most $c_U.n.a(1, n)$ for some constant c_U.

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

7.6.2 FRT(i) for i>1

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

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

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

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

Then for each node f in BB a call *boundary* is performed in *tree*(f) that returns $bl(f)$, which takes $O(|bl(f)|)$ steps. Note that then at most 2 nodes may be removed from $bl(f)$ in the subsequent computations, but still $bl(f)$ contains at least one node: since f is a boundary node in BB, there is at least one boundary node left in $bl(f)$ (cf. Subsection 7.2.2). Hence, the net cost of the entire computation of $bl(f)$ is at most $c_3.|bl(f)|$ steps for some constant c_3.

The remaining operations take at most c_4 steps.

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

procedure joinclasses(i): The procedure takes a net number of steps that is linear to the number of classes that will be joined, apart from the steps for the recursive call. Therefore, each step is charged to a current class that is joined (i.e., that is joined with another class). Hence, the total net amount of steps is at most $c_j.n$ for some constant c_j.

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

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

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

1. *The removal of parts of structures:* The removal of parts of structures can be performed in $O(1)$ time per item that must be removed. Therefore, we charge the cost of the removal of an item to its creation. This increases the cost of some operations by constant time only.
2. *The calls of procedure basic-internal-link and basic-integrate:* The execution of the calls of *basic-internal-insert* and *basic-integrate* take at most c_5.*(the number of processed nodes)* steps. Therefore, we charge the cost of the above statements to the processed nodes. Note that in both cases the processed nodes will be contained in a new set that has a higher *lowindex* value than the old set in which they were contained, and that a node will never be contained in a set with a lower *lowindex* value. Therefore the number of times that a node can be charged to is bounded by the number of different *lowindex* values. Since there are at most n (>1) elements in a set, there are by the definition of *lowindex* (cf. (7.1)) at most $a(i,\lceil\frac{n+1}{2}\rceil)+2 \le 3.a(i,n)$ different values. Therefore, the total cost of the considered parts of the procedure is at most $c_6.n.a(i,n)$ steps for some constant c_6.

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

3. *The recursive call Link($i-1$)* We consider the cost of a recursive call $link(i-1)$ in the recursive call part.

   The cost for changing c-marks of edges (not being the inserted edge) in procedure $link(i)$ (and for the related computations) is linear to the number of times that contraction edges are affected in the recursive call $link(i-1)$. Later, in the part considering the recursive calls, we will show that this is at most $\frac{1}{2}.n.a(i,n)$. (This is stated in Observation 7.6.4.) Hence, this takes altogether $c_7.n.a(i,n)$ steps for some constant c_7.

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

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

Unions: There are at most $2n$ edge sides in layer i. By Lemma 3.4.3, the time for the joinings and insertions of edges in layer i is at most $c_U.n.a(i,n)$ for some constant c_U.

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

Observation 7.6.1 *The operations on contraction trees (in layer i) by procedure Link($(e,x,y),i$) are:*

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

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

Similar to the proof of Claim 3.4.2, we can prove the following claim.

Claim 7.6.2 *A recursive call Link($(e,c,d),s,t,i-1$) inside Link($(e,x,y),s,t,i$), with $c = contr(x)$ and $d = contr(y)$, is performed only if*

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

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

Similarly, we say that any recursive call $Link((e,c,d),s,t,i-1)$ is an l-call if $l = lowindex(s,i) = lowindex(t,i)$. A recursive call $boundary(c,d,i-1)$ or $joinclasses(JJ,i-1)$ is an l-call if $l = lowindex(s,i)$, where s is the name of the tree on which the operation is applied. Obviously an l-call operates on l-contraction nodes only, and l-contraction nodes are only operated on by l-calls. We compute the cost of all l-calls for fixed value l.

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

By Claim 7.6.2 and since $|nodes(b)| \geq 2$ for each contraction node b, it follows in case of an l-call Link$(s,t,i-1)$ that we have $l > 1$ and that the size of the set $CN(tree(s,i-1)) \cup CN(tree(t,i-1))$ is $< A(i,l+1)$. Therefore the maximal size of any tree of l-contraction nodes that results from such an l-call is $< A(i,l+1)$. By Observation 7.6.1 and since in an initialisation at most one contraction node per tree is created, it follows that the maximal size of any occurring tree of l-contraction nodes is $\leq max\{A(i,l+1),1\}$.

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

Now let l be fixed number with $1 \leq l \leq a(i,n)$. Now partition the total collection of all l-contraction nodes involved in l-calls into collections that correspond to the maximal sets that ever exist (which is possible because of Observation 7.6.1). Then the size of such a maximal collection is at most $A(i,l+1)$. We have the following observation (that will be proved further on).

Observation 7.6.3 *The sequence of essential recursive l-calls on the nodes of a maximal set in FRT$(i-1)$ is an essential sequence.*

For each such maximal collection of k contraction nodes, the cost of all essential l-calls on these nodes in FRT$(i-1)$ is at most $c.k.a(i-1,k) \leq c.k.\ a(i-1, A(i,l+1))$. Hence, the total cost of all essential l-calls in FRT$(i-1)$ on l-cluster nodes is at most $c.(number\ of\ l\text{-}cluster\ nodes).\ a(i-1,A(i,l+1))$. Since for each l-contraction node b we have $|nodes(b)| \geq 2.A(i,l)$ (cf. (7.2)), and since as long as a node is contained in tree structures with *lowindex* value

l it has the same contraction node in which it is contained, there are at most $n/(2.A(i,l))$ l-contraction nodes. Therefore, the total number of steps for all essential l-calls is at most

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

by using $i > 1$, equation (2.1) and Lemma 2.3.4 respectively.

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

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

Observation 7.6.4 *The number of times that contraction edges are affected in the recursive calls* $link(i-1)$ *is* $\frac{1}{2}.n.a(i,n)$ *altogether.*

We are left to prove Observation 7.6.3.

Proof of Observation 7.6.3. We are left to prove Observation 7.6.3. Suppose some essential operation $boundary(i-1)$ is executed inside operation $boundary(i)$, returning boundary list BB. For each node f in BB with edges e_1 and e_2 in its sublist, there are edges e'_1 and e'_2 in $bl(f)$ such that the originals of e'_1 and e_1 are in the same edge set in F_0 and similarly for e_2 and e'_2. Since the operations in FRT(i) yield a feasible sequence in FRT(i), the call $boundary(i)$ is followed by a call $joinclasses(i)$ that joins the classes of e'_1 and e'_2 inside $extree(f)$. Since these two classes each have at least one external edge in FRT(i), viz., $orig(e_1)$ and $orig(e_2)$, there is a recursive call $joinclasses(i-1)$ with a joining list that contains node f together with two edges in its sublists that are in the same edge sets as e_1 and e_2 respectively. This proves that the sequence of essential recursive l-calls on the nodes of a maximal set in FRT$(i-1)$ is a feasible sequence. This concludes the proof of Observation 7.6.3. □

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

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

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

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

7.6.3 FRT(i) for i≥ 1

From subsections 7.6.1 and 7.6.2, it follows that an essential sequence in FRT(i) on n nodes takes at most $c.n.a(i,n)$ steps. By the observation, that all nodes that still are in singleton trees after executing the sequence are not involved in the algorithms, Lemma 7.5.1 follows.

7.7 FRT Structures

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

Firstly, we now consider the operations as described in Section 7.2. We express these operations in the operations described in Section 7.4. Note that the operations *boundary* and *joinclasses* match in both sections if the appropriate i is used. The operation $link((e,x,y))$ corresponds to the operation

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

in the FRT(i) structure. Hence, the time needed for a *link* operation is now extended with two *treename* operations, being two steps. Hence , this does not increase the order of time complexity of this operation. The operations $equal\text{-}class\text{-}edges(x,y)$ can be performed by a call $candidates(x,y,i)$ returning two edges e_x and e_y and by performing the Find calls $class_0(e_x)$ and $class_0(e_y)$ (in UF_0). Hence, the time needed for such a call is the time for *candidates* and two Find operations in UF_0, which is $O(1)$ steps. Therefore, we can consider the operations as described in Section 7.2 with the same order of complexity. Thus, Lemma 7.5.1 remains valid for these operations (in order of magnitude).

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

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

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

By Lemma 2.3.4, an Ackermann net for n can be computed in $O(\log n)$ time and takes $O(\log n)$ space. Moreover, it is readily verified that the initialisation of FRT(i) can be performed in $O(n)$ time. Finally, by induction to i it easily follows that the total space complexity of FRT(i) is $O(n)$, since layer $i-1$ has at most $\frac{1}{2}.n$ contraction nodes since for each contraction node b we have $|nodes(b)| \geq 2$.

By Lemma 7.5.1, by the above observations and since UF(i) takes $O(n_e.a(i, n_e))$ time for n_e elements, we obtain the following result.

Theorem 7.7.1 *Let a FRT(i) structure for a forest with n nodes be given. The structure and the algorithms can be implemented as a pointer/ $\log n$ solution such that the following holds. An essential sequence (of the operations link, boundary and joinclasses) in FRT(i) needs a total of $O(n_e.i.a(i, n_e))$ time. ($i \geq 1$, $n_e \geq 2$), where n_e is the number of nodes that are not contained in singleton trees after the execution of the sequence. (Of course, $n_e \leq n$.) Each equal-class-edges operation takes $O(i)$ time. Each nonessential call boundary takes $O(i)$ time. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

By using the same solution as in Theorem 3.6.1 for the augmentation of the Ackermann net that is used, the above lemma can be extended with the insertion of new (isolated) nodes in the structure with the same complexity bounds, where the insertion of a new node takes $O(1)$ time.

We define an α-FRT structure (for n nodes) as follows. Initially, a FRT($\alpha(n, n)$) structure is used. From time to time, a transformation is performed, replacing a FRT(i) structure by a FRT($i-1$) structure, viz., each time that $\alpha(q, n)$ decreases by one, where at any moment q is the number of queries *equal-class-edges* and *boundary* performed until then. This is performed similar to the way in the proof of Theorem 3.5.2, where hence now the queries *equal-class-edges* and *boundary* play the role of the Find operations, and where *link* and *joinclasses* play the role of the Union operations. The building of the new structure FRT($i-1$) is done like in Theorem 3.5.2, but instead of building parts of FRT($i-1$) during *equal-class-edges* and *boundary* operations, and using parts of both FRT(i) and FRT($i-1$), we do the following. We have all pointers in the forest F_0 in duplicate, say version 1 and version 2, and we either use version 1 or version 2 of all the pointers. When for FRT(i) version 1 is used, then FRT($i-1$) is builded with version 2 and starting from the moment that FRT($i-1$) is completed the version 2 pointers are used (instead of version 1 pointers).

Then we obtain the following result.

Theorem 7.7.2 *Let an α-FRT structure for an "empty" forest with n nodes be given. The structure and the algorithms can be implemented as a pointer/$\log n$ solution such that the following holds. A matching sequence M of operations link, boundary, joinclasses and equal-class-edges in α-FRT needs a total of $O((n_e + m).\alpha(m,n))$ time, where m is the number of operations equal-class-edges and boundary that is performed, and where n_e is the number of nodes that are contained in non-singleton trees at the end (and, hence, the essential subsequence of M consists of $\theta(n_e)$ operations). The q^{th} call of the operations equal-class-edges and boundary takes $O(\alpha(q,n))$ time if it is a call of equal-class-edges or a nonessential call of boundary. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space.*

The proof is similar to the proof of Theorem 3.5.2. However, for the initial case, i.e., $i = \alpha(n,n)$, an essential sequence takes $O(n_e.i.a(i,n_e)) = O(n_e.i)$ time (by Theorem 7.7.1 and Lemma 2.3.7). For rebuilding a FRT(i) structure to a FRT$(i-1)$ structure, we now charge to each of the last $\lceil \frac{1}{2} f \rceil$ operations *equal-class-edges* and *boundary* for $O(i)$ time, based on Equation (3.5) (note that now $f \geq n$). This $O(i)$ is then included in the cost of these operations, hence augmenting their cost by a constant factor only. Thus, if $m \leq n$, we have $i = \alpha(n,n)$, and the total cost is $O((n_e + f).\alpha(m,n))$. Otherwise, charge the $O(n_e.i)$ cost for FRT$(\alpha(n,n))$ to the first n operations *equal-class-edges* and *boundary*, hence augmenting their costs by a constant factor again. Then, the cost of these operations *equal-class-edges* and *boundary* is computed like in Theorem 3.5.2, yielding the required result.

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

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

By using the same techniques as in Theorem 3.6.2, the above theorem can be extended with the insertion of new (isolated) nodes in the structure with the corresponding complexity bound $O(n + (n_e + m).\alpha(m,n))$ (where m, n, n_e, and q denote the current number at the time of consideration). The strategy is again to start with a structure FRT$(\alpha'(n,n))$ (where $\alpha'(m,n)$ is defined below, satisfying $\alpha'(m,n) = \theta(\alpha(m,n))$), and to replace FRT$(i)$ by FRT(i') (for some $i' \neq i$) in case $\alpha'(q,n)$ decreases or increases (with additional constraints), where at any moment q is the number of queries *equal-class-edges* and *boundary* performed until then, and n is the number of nodes actually present in the structure at that moment, while the insertion of a new node in the structure is deferred until that node is operated on (viz., by operation *link*: it then becomes part of a non-singleton tree). (The deferring of insertions in the actual data structure guarantees, that at any time, $n_{pres} = n_e$, where n_e is as before, and where n_{pres} is the number of nodes present in the thus adapted structure.) All this is performed similar to the method in the proof of Theorem 3.6.2, where now the queries *equal-class-edges* and *boundary* play the role of the Find operations, where *link* and *joinclasses* play the role of the Union operations, and where the building of the new structure FRT(i') is done like in Theorem 3.6.2 with the previous adaptations. We want to remark that if at any time $m = O(n)$ (i.e., at any time the number of operations performed until then is at most linear in the number of nodes present at that time), then the above transformation techniques can be simplified by replacing m by n in the conditions; then only $\alpha(n, n)$ is used and maintained, and only rebuildings from FRT(i) to FRT($i + 1$) are performed, viz., if $\alpha(n, n)$ increases. (This situation occurs in the 2ec-and the 3ec-problem.)

We describe further changes for the above situation w.r.t. the proof of Theorem 3.6.2. Firstly, instead of the inverse Ackermann function $\alpha(m, n)$, a variant is taken, viz.,

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

We have $\alpha'(m, n) = \theta(\alpha(m, n))$. The checking of the transformation condition can be done in a way similar as in the proof of Theorem 3.6.2, and the only necessary arithmetic operations still are addition, subtraction and comparison. Then the complexity part of the proof of Theorem 3.6.2 is changed as follows. Lemmas 2.3.10, 2.3.11, and 2.3.12 are adapted to deal with $i.a(i, n)$ instead of with $a(i, n)$. The cost function (Eq. (3.9)) in the proof of Theorem 3.6.2 is slightly adapted (viz., its constants are changed, and n_{base} is replaced by $n_{base}.\alpha_b$). Since at any moment, the number n_{pres} of nodes actually present in the structure satisfies $n_{pres} = n_e$, and since an insertion takes $O(1)$ time, the resulting time complexity becomes $O(n + (n_e + m).\alpha'(m, n)) = O(n + (n_e + m).\alpha(m, n))$.

7.8 Concluding Remarks

Like in Chapter 3, there is no real need to perform transformations of FRT structures like those occurring in Section 7.7 (Theorem 7.7.2): structure FRT(2) is suited for all practical situations. An essential sequence on n nodes in FRT(2) takes $\leq c.2.n.a(2, n) \leq 8.n$ time for $n \leq 65536$ and $\leq 10.c.n$ time for *very* large practical values $n \leq 2^{65536}$, where c is not too large a constant (cf. Section 7.6 for its definition). The time bound for an essential sequence in FRT(3) is $c.3.n.a(3, n) \leq 12.c.n$ for n with $\alpha(n, n) \leq 3$. Again, in all practical situations for FRT(2) (and FRT(3)), only the nontrivial Ackermann values 16 and 65536 need to be available, so there is no need to compute any further Ackermann values.

Therefore, we conjecture that FRT(2) is a fast structure (i.e., practically linear time) for all practical situations, with constant time *equal-class-edges* queries.

Chapter 8

Maintenance of the 2- and 3-Edge-Connected Components of Graphs: Optimal Solutions

8.1 Introduction

In Chapter 6, a data structure with algorithms was presented for maintaining the 2- and 3-edge-connectivity relation for a graph. The algorithm starts from an empty graph of n nodes in which edges are inserted one by one and where at any time for any two nodes the query that asks whether these nodes are 2- or 3-edge-connected can be answered in $O(1)$ time. The insertion of e edges takes $O(n \log n + e)$ time altogether. We show that by means of fractionally rooted trees the above time bound can be improved for maintaining the 2- and 3-edge-connected components of a general graph, i.e., starting from an empty graph of n nodes. The solution has a total running time of $O(n + m.\alpha(m, n))$, where m is the number of edge insertions and queries. In the next chapter, we also describe solutions for maintaining the 2-vertex-connected and 3-vertex-connected components with the same time bounds. Recently, Westbrook and Tarjan [34] independently obtained the same time bounds for 2-edge/vertex-connectivity. The methods though are quite different. Very recently, Galil and Italiano [14] independently obtained results with these time bounds for a special case of the problem of maintaining 3-edge-connected components of graphs, viz., in which the initial graph is connected.

This chapter is organized as follows. Section 8.2 contains some preliminaries. In Section 8.3, the maintenance of 2-edge-connected components is considered. Finally, Section 8.4 considers the maintenance of 3-edge-connected components.

8.2 Preliminaries

In our algorithms we need a structure for the Union-Find problem. We will use the Union-Find structures presented in Chapter 3. We call the structure with time complexity $O(n + m.\alpha(m, n))$ an *α-UF structure*. Like in Chapter 6, Union-Find structures are used to maintain the equivalence classes for connectivity, 2-edge-connectivity and 3-edge-connectivity. These structures are denoted by UF_c, UF_{2ec}, and UF_{3ec} respectively, where the corresponding Finds on elements x are denoted by $c(x)$, $2ec(x)$ and $3ec(x)$ respectively.

For maintaining 3-edge-connectivity, we also need a structure for the Circular Split-Find problem. In Chapter 4, fast solutions for the Circular Split-Find problem are given that take $O(n + m.\alpha(m, n))$ time for all Circular Splits and m Finds on n elements. We call such a structure an *α-GSF structure*. The Circular Split-Find structure is used in Chapter 6. It will not be used explicitly in this chapter, but we only choose appropriate Circular Split-Find structures when we apply the results of Chapter 6.

8.3 Two-Edge-Connectivity

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

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

We follow a strategy based on observations for $2ec(G)$ in Subsection 6.3.1. We give the further observations that lead to our algorithm.

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

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

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

As observed in Subsection 6.3.4, a 2ec-class (of nodes) induces some subtree in $SF(G)$. Hence, in particular a non-singleton 2ec-class (i.e., with at least 2 nodes) induces some subtree in $SF(G)$. The set of the edges in that subtree is a real edge class. Therefore, if each subtree in $SF(G)$ that is induced by a real edge class is contracted to some node, then we obtain the forest $2ec(G)$ (up to edge names and node names). (Note that the edges in forest $2ec(G)$ correspond to the edges in $SF(G)$ that are in quasi edge classes.)

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

We consider the insertion of edge (e, x, y). We distinguish the two relevant cases of Subsection 6.3.1.

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

Now suppose x and y are in the same tree T of $SF(G)$ (and hence classes $2ec(x)$ and $2ec(y)$ are in the same tree of $2ec(G)$). Let P be the tree path in T between x and y. We use the terminology of Section 7.2. By the definition of edge classes, a boundary node of P is either one of the end nodes x or y, or it is a node for which its two neighbours on P are not both in the same 2ec-class as itself. The two neighbours of an internal node z on P are inside class $2ec(z)$ too. Therefore, if we compute the boundary nodes of P only, then we obtain one or two nodes of each 2ec-class (of nodes) that needs to be joined because of inserting (e, x, y).

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

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

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

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

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

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

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

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

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

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

Lemma 8.3.1 *In a 2EC structure for a graph with n nodes, the time needed for a sequence of essential insertions consists of the time for an essential sequence on n nodes in FRT_{2ec}, the time for $O(n)$ Unions in UF_c and UF_{2ec}, the time for at most $O(n)$ nonessential calls boundary in FRT_{2ec}, the time for at most $O(n)$ Finds in UF_{2ec} and UF_c, together with an additional amount of $O(n)$ time. Each nonessential insertion takes $O(1)$ time together with $\theta(1)$ Finds in UF_c and UF_{2ec}.*

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

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

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

Theorem 8.3.2 *There exists a data structure and algorithms that solve the 2ec-problem and that can be implemented as a pointer/ $\log n$ solution such that the following holds. Starting from an empty graph with n nodes, the total time that is needed for all essential insertions is $O(n.i.a(i,n))$, whereas a query and a nonessential insertion can be performed in $O(i)$ time. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. By Theorem 7.7.1 (for FRT(i)) and Theorem 3.4.4 (for the complexity of UF(i)), it follows that the initialisation can be performed in $O(n)$ time.

Each nonessential call of *boundary* takes $O(i)$ time. Each Find operation in UF(i) takes $O(i)$ time too. Hence, a query can be performed in $O(i)$ time. By Lemma 8.3.1, Theorem 7.7.1 and by Theorem 3.4.4, the lemma follows. □

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

Now take FRT($\alpha(n,n)$) as FRT_{2ec} for a graph with n nodes, where $\alpha(n,n)$ is obtained as in Chapter 3, and take for UF the α-UF structure. Then we obtain the following.

Theorem 8.3.3 *There exists a data structure and algorithms that solve the 2ec-problem and that can be implemented as a pointer/ $\log n$ solution such that the following holds. The total time that is needed starting from an empty graph with n nodes is $O(m.\alpha(m,n))$ (where m is the number of edge insertions and queries), whereas the f^{th} operation is performed in $O(\alpha(f,n))$ time if that operation is query or a nonessential insertion. The initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space.*

Proof. Each query and nonessential insertion corresponds to $\theta(1)$ Finds in the UF structures. Moreover, all essential insertions take at most $O(n)$ Finds. Hence, by Theorem 3.5.2, the f^{th} operation is performed in $O(\alpha(f',n)) = O(\alpha(f,n))$ time (where $f' = \theta(f) + O(n)$ by Lemma 8.3.1) if that operation is query or a nonessential insertion. The remaining statements follow by Theorem 7.7.1 (with $n_e \leq min\{2m,n\}$, where $n_e \leq \{2m,n\}$ is implied by Lemma 8.3.1, since the part of the graph that is operated on contains at most $min\{2m,n\}$ nodes), Lemma 2.3.7 (w.r.t FRT($\alpha(n,n)$)), and by Theorem 3.5.2 (where if $m \leq n$, then Lemma 3.4.3 is applied on the initial UF(i) structure of the α-UF structure instead, viz., where $i = \alpha(n,n)$). □

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

8.4 Three-Edge-Connectivity

We will now extend the results to the maintenance of 3-edge-connected components in a graph, with a time complexity of $O(n + m.\alpha(m, n))$ for n nodes and m queries and insertions.

8.4.1 Observations

We recall the observations of Subsection 6.4.2. For detecting the 3ec-classes it suffices to detect the 3ec-classes inside the 2-edge-connected components. Therefore, our algorithms for general graphs maintain the 2ec-classes (as in Section 8.3), and they maintain the 3ec-classes by using solutions for 3-edge-connectivity within 2-edge-connected components.

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

Suppose edge (e, x, y) is inserted in graph G yielding graph G'. Then the following changes occur. We distinguish three cases (cf. Subsection 6.4.2).

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

Otherwise, if $2ec(x) = 2ec(y)$ then (e, x, y) is inserted inside a 2-edge-connected component and the changes as described in Subsection 6.4.1 occur.

Otherwise we have $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$. Then consider $2ec(G)$. Let P_2 be the tree path between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ (consisting of the class nodes only) (cf. Subsection 6.3.1) and let CS_2 be the cyclic list obtained from P_2 by inserting the interconnection edges between consecutive class nodes of P_2 and by inserting the edge (e, x, y) between class nodes $2ec(x)$ and $2ec(y)$. Then the major changes are the following:

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

3. a new cycle s of 3ec-classes arises; the new cycle node s links the (updated) cycle trees that correspond to the 2ec-classes on CS_2

We consider the changes more precisely.

1. This part is identical to Subsection 6.3.1.
2. We consider the changes of the 3ec-classes that occur in 2ec-classes on P_2. Consider a particular 2ec-class C on P_2 in $2ec(G)$. Let u and v be the two nodes in C that are end nodes of interconnection edges on CS_2. Then there is a new path between u and v in G' that does not intersect with C except for u and v, where such a path did not exist in G before. Hence, considered within C only, this corresponds to inserting a temporary edge between the nodes u and v (cf. Figure 6.15), since the 3ec-classes are completely determined by the 2-edge-connected components in which they are contained (and hence by the nodes in C together with edges of G that have both their end nodes in C. Cf. Corollary 6.2.6). The update of the 3ec-classes (and hence the cycle tree) can be performed in C by the insertion of a temporary edge in the 2-edge-connected component C.
3. Now suppose all these "local" insertions are performed in the 2ec-classes on P_2. Then the two edges in CS_2 that are incident with one 2ec-class C on P_2 have their end nodes in the same (updated) 3ec-class in C. Call such a 3ec-class the *interconnection 3ec-class* in C. Then all these interconnection 3ec-classes form a new cycle s. Then the updated cycle tree T_C in each 2-edge-connected component C on P_2 is linked to the new cycle node s by an edge between cycle node s and the class node of the interconnection 3ec-class in C. All these cycle trees are linked to s and hence now form one new tree together.

8.4.2 Algorithms

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

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

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

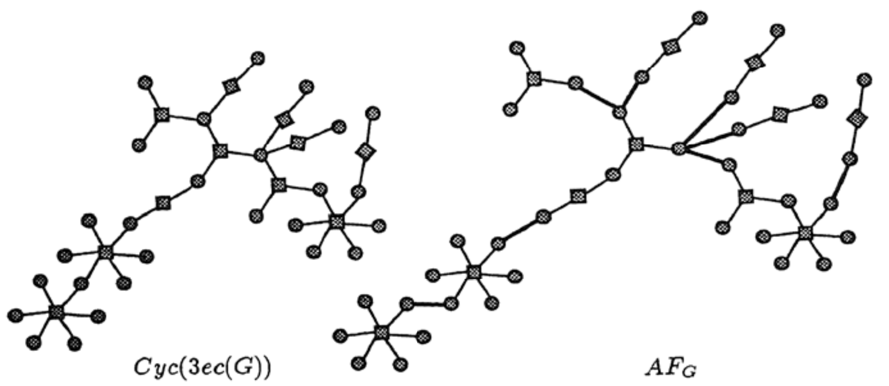
Each (extended) 3ec-class C of G is partitioned into subclasses of nodes. To each subclass a (new) distinct node is related as its name, called the *subclass node*. We call these subclass nodes the subclass nodes for C. The subclass node of the subclass to which a node x belongs is denoted by $sub(x)$.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

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

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

Stated informally, AF_G can be obtained by replacing each class node in $Cyc(3ec(G))$ by some tree of subclass nodes and connectors. See Figure 8.1.

Figure 8.1: Augmented cycle forest



We consider the insertion of an edge (e, x, y) in a 2-edge-connected graph in terms of an augmented cycle forest AF_G for G. Let $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. All class nodes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ become 3-edge-connected in $3ec(G)$ and the corresponding classes form one new class. Note that these classes are the classes that have at least one subclass on the tree path P in AF_G between $sub(x)$ and $sub(y)$. Hence, we can update the structure according to the following observations (also cf. Subsection 6.4.1).

- Two successive subclass nodes on P (without a cycle node in between) correspond to the same class. Hence, it suffices to obtain all the subclass nodes on P that are adjacent to a cycle node on P.

- All the classes of which a subclass node is "on" P must be joined into one new class C'.

- The augmented cycle tree AF_G must be adapted to be an augmented cycle tree for the resulting graph. Hence, all subclass nodes for C' must form a tree and no cycle node may occur in between. In particular, this can be done by joining for each cycle node s on P the two subclass nodes that are its neighbours on P and to split cycle s in AF_G accordingly.

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

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

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

- Each 3ec-class C is partitioned into subclasses obtained by intersecting C with the basic clusters. To each subclass, a unique node is related as the subclass node. The subclass nodes of G are the subclass nodes in AF_G. Then AF_G is an augmented cycle forest for G.

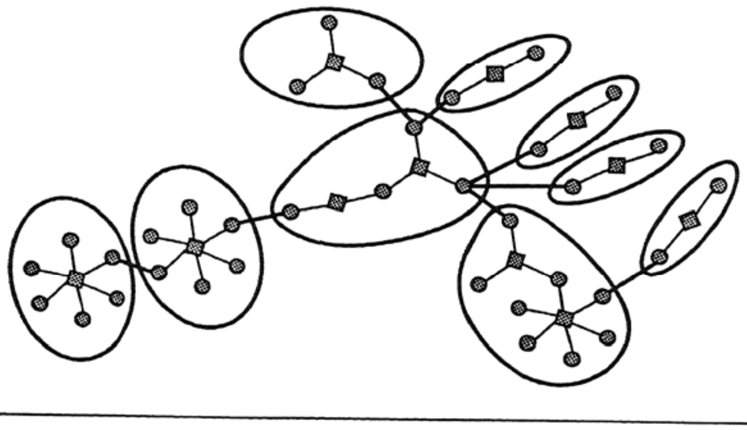
- Each subclass node is considered to be contained in the basic cluster that contains its subclass. Then for a basic-cluster b, the subclass nodes that are contained in b together with appropriate cycle nodes of AF_G induce a subtree of AF_G, denoted by $tree(b)$.
- The edges of AF_G of which the end nodes are in different basic-clusters are connectors.
- There is a connector with end nodes in the basic-clusters b_1 and b_2 iff there is bc-edge between b_1 and b_2.

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

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

We thus have a structure of clusters with bc-edges in between, where the original connector of such a bc-edge "connects" the occurrences in AF_G of some 3ec-class of G inside the two corresponding basic-clusters (viz., the 3ec-classes determined by the end nodes of the connector). See Figure 8.2 for the example of Figure 8.1.

Figure 8.2: A forest $bc(G)$



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

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

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

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

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

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

  A boundary node b of P is either one of the end nodes c or d, or it is a node for which its two incident bc-edges e_1 and e_2 on P are not both in the same bc-edge class. In the latter case this means that the two connectors that are their originals are connectors for different 3ec-classes. Moreover, these connectors are on P'. Hence, at least two different subclasses on P' and at least one cycle node on P' are in cluster b.

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

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

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

We distinguish between the different layers of *representation.*

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

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

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

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

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

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

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

Recall that for a basic-cluster b, the part of AF_G inside basic-cluster b, viz. $tree(B)$, is a cycle tree on the nodes of basic-cluster b. Then $tree(b)$ is implemented as a cycle tree independent of the rest of AF_G or G. It is implemented and maintained as the cycle tree in the former solution of Subsection 6.4.1 (cf. Lemma 6.4.5) (for maintaining 3-edge-connectivity inside a 2-edge-connected graph). We refer to this solution as the *local structure.* The Union-Find and the Circular Split-Find structures used in the local structure are denoted by UF_{loc} and GSF_{loc}. The Find operation in UF_{loc} for a node x (returning the name of its subclass) is denoted by $sub(x)$. The insertion operation in a local structure is denoted by $insertloc_3$.

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

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

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

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

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

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

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

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

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

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

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

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

3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. Firstly, the 2ec-classes that will be joined into one new class are determined. This is done as follows. A boundary list BL for x and y is computed in $2EC$ (this is the first part of the call $insert_2((e, x, y))$). Subsequently the names of the 2ec-classes are obtained, where to each such name k a sublist is related that is the concatenation of all sublists for $z \in BL$ with $2ec(z) = k$. These names are stored in a temporary list TL. Then all edges in the sublists are removed from the sublists that have both their end nodes in the same 2ec-class. (Hence, we are left with a list TL where the sublist of each 2ec-class name k contains the interconnection edges that are incident with 2ec-class k, and with another class of which the name is in TL. Such a sublist consists of exactly two edges except for the sublists of $2ec(x)$ and $2ec(y)$.) Then a list L is constructed from TL consisting of the names and their sublists in TL such that the (1 or 2) neighbours in L of each 2ec-class c in L are the 2ec-classes in which the end node(s) of the edges in its sublists are contained (apart from 2ec-class c). (Note that this can be done by obtaining each for 2ec-class C the other 2ec-classes in which the end nodes of the edges in its sublist are contained, and by setting pointers from C to these 2ec-class names.) (Now L contains the class nodes of the tree path P between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ in the proper order, where the sublist for each class node c contains the interconnection edges between class c and its neighbour(s) on P. Hence, the sublist of 2ec-class name c consists of the interconnection edges that are incident with class c and with the (one or two) neighbours classes in L.)

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

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

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

Cluster node b is linked with the involved trees in $bc(G)$ (corresponding to the 2-edge-connected components that are involved) by means of new bc-edges as follows. For each auxiliary node z_C (in L) together with connector (e', z_C, z'_C), let $b' = clus(z'_C)$. Then a new bc-connector (e', b, b') is created (with the appropriate pointer between (e', z_C, z'_C) and (e', b, b')), and the tree in $bc(G)$ containing b' is linked with b by means of call $link((e', b, b'))$. The edge (e', z_C, z'_C) is related to $sub(z_C)$ as its associated edge. If $sub(z'_C)$ does not have an associated edge yet, then (e', z_C, z'_C) is related to $sub(z'_C)$ as its associated edge. Otherwise, the following is done. Let (e'', z'', z''') be the associated edge for $sub(z'_C)$. Then the operation $joinclasses(J)$ is performed, where J consists of the node b' with the bc-edges (e', b, b') and $(e'', clus(z'), clus(z''))$ in its sublist (to reflect that these two edges are in the same bc-edge class).

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

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

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

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

Lemma 8.4.1 *In a 3EC structure for a graph with n nodes, a nonessential insertion takes $O(1)$ time together with the time for $\theta(1)$ Find operations in a UF structure. The time needed for a sequence of essential insertions in 3EC is at most linear to the time for an essential sequence on $O(n)$ nodes in FRT_{3ec} and an essential sequence on $O(n)$ nodes in FRT_{2ec}, the time for $O(n)$ Unions in the UF structures and $O(n)$ Circular Splits in the GSF_{loc} structure, the time for $O(n)$ nonessential calls boundary in FRT_{2ec} and FRT_{3ec}, the time for all $O(n)$ Finds in the UF structures and the GSF_{loc} structure, together with an additional amount of $O(n)$ time.*

Proof. We define a step be an ordinary computational step or a Find operation in any UF or GSF_{loc} structure. We consider a collection of essential $insert_3$ operations in the considered graph, including the (essential) $insert_3$ operations called in the execution of operation $insert_3$ itself. Therefore, we do not consider the cost of an essential call $insert_3$ inside procedure $insert_3$: we already consider it in the above collection. (We may think of such an $insert_3$ call to occur just before the call $insert_3$ in which it was invoked.)

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

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

All calls $insert_{2ec}$ in the calls $insert_{3ec}$ are essential. Therefore, by Lemma 8.3.1 the claim regarding the operations present in $2EC$ is true.

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

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

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

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

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

   Since there are at most $3n-1$ nodes present, and since these nodes are partitioned into disjoint clusters in which the local structures are applied, at most $O(n)$ essential calls $insertloc_3$ may occur. By Lemma 6.4.5 this takes time linear to the time for Unions and Splits in the structures UF_{loc} and GSF_{loc} respectively, which does not contribute to the net cost, together with $O(n)$ Finds in these structures.

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

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

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

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

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

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

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

Theorem 8.4.2 *A 3EC structure with the algorithms solves the 3ec-problem and can be implemented as a pointer/* $\log n$ *solution such that the following holds. The total time that is needed for all essential insertions starting from an empty graph of* n *nodes is* $O(n.i.a(i,n))$, *whereas the queries and nonessential insertions can be performed in* $O(i)$ *time. The initialisation can be performed in* $O(n)$ *time and the entire structure takes* $O(n)$ *space* ($i \geq 1$, $n \geq 2$).

Proof. It is easily seen that the initialisation can be done in $O(n)$ time. By Lemma 8.4.1, Theorem 7.7.1 and Theorem 3.4.4 (for UF(i) and GSF(i)) the theorem follows. □

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

Theorem 8.4.3 *There exists a structure and algorithms that solve the 3ec-problem and that can be implemented as a pointer/* $\log n$ *solution such that the following holds. The total time that is needed starting from an empty graph with* n *nodes is* $O(m.\alpha(m,n))$ *(where* m *is the number of edge insertions and queries), whereas the* f^{th} *operation is performed in* $O(\alpha(f,n))$ *time if that operation is a query or a nonessential insertion. The initialisation can be performed in* $O(n)$ *time and the entire structure takes* $O(n)$ *space.*

Proof. Like the proof of Theorem 8.3.3. □

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 7.7). Then n, m and f denote the current number at the moment of consideration. (Note that at any time, at most $O(n)$ operations *boundary* are performed on one of the two FRT structures, which simplifies the insertions of new nodes in the FRT structure. Cf. Section 7.7. A similar remark holds for the structures GSF and UF_{loc}.)

8.5 Concluding Remarks

We have presented solutions for the problem of maintaining the 2-edge-connected and the 3-edge-connected components of graphs under insertion of edges and verticves. The solutions take $O(n + m.\alpha(m, n))$ time, starting from the graph $< \emptyset, \emptyset >$, and are optimal on Pointer Machines and Cell Probe Machines. For 2-edge-connectivity and 2-vertex-connectivity, the optimality of solutions that run in $O(n + m.\alpha(m, n))$ time is proved in [34] (where for our results we use that the insertion of a node takes $\Omega(1)$ time). (Note that the complexity of the algorithms in [34] is $O(m'.\alpha(m', n))$, where $m' = m + n$, since we consider m to be the number of queries and edge insertions and n to be the final number of nodes, whereas m' in [34] includes both.) (Actually, the above proofs are for Pointer Machines with the Separation Condition, but by using the results of Chapter 5, the bounds follow for general Pointer Machines.) We give the proof for the 3ec-problem. Like in [34] we use reductions to the Union-Find problem. Consider the Union-Find problem for some collection of elements. For each element x there is a triple of nodes x_1, x_2 and x_3 with edges (x_1, x_2), (x_2, x_3) and (x_3, x_1). Then a query Find(x) is performed by a query *3ec-comp*(x_1) in the graph. Moreover, the joining of two sets is as follows: for each set a triple of nodes for some element in that set is taken, say x_1, x_2 and x_3, and y_1, y_2, and y_3, and then the edges (x_1, y_1), (x_2, y_2) and (x_3, y_3) are inserted in the graph. This yields that every set corresponds to a 3ec-class in the graph. By the lower bounds for the Union-Find problem on both Pointer Machines and Cell Probe Machines (cf. Chapter 5 and [10]), the lower bound of $\Omega(n + m.\alpha(m, n))$ follows for the 3-edge-connectivity problem, if we use that the insertion of a node takes $\Omega(1)$ time.

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

Chapter 9

Maintenance of the 2- and 3-Vertex-Connected Components of Graphs: Optimal Solutions

9.1 Introduction

In the previous chapter, optimal solution were presented for maintaining the 2- and 3-edge-connectivity relation for a graph. We showed that, by means of fractionally rooted trees, the above time bounds can be improved for maintaining the 2- and 3-edge-connected components of a general graph, i.e., starting from an empty graph of n nodes. The solution has a total running time of $O(n + m.\alpha(m, n))$, where m is the number of edge insertions and queries. In this chapter, we continue with the 2- and 3-vertex-connectivity relation, where k-vertex-connectivity is defined as follows: two nodes are k-vertex-connected iff there exist k different vertex-disjoint paths between them. We present an optimal solution for maintaining the 2-vertex-connected components of a graph with the same time bound, i.e., for the 2-vertex-connectivity querying. The solution makes use of fractionally rooted trees and has a total running time of $O(n + m.\alpha(m, n))$, where m is the number of edge insertions and queries and n is the number of nodes. Moreover, we briefly describe an optimal solution for maintaining the 3-vertex-connected components of a graph with the same time bound as well. We will present the detailed solution in a future report and only give a sketch here.

This chapter is organized as follows. Section 9.2 contains some preliminaries. In Section 9.3, the maintenance of 2-vertex-connected components is considered, and in Section 9.4, the maintenance of 3-vertex-connected components is considered.

9.2 Preliminaries

9.2.1 Graphs and Terminology

Definition 9.2.1 *Two nodes x and y are k-vertex-connected iff there exist k different vertex-disjoint paths between x and y.*

It is easily seen that if two nodes are k-vertex-connected, then they are k'-vertex-connected for any k' with $1 \leq k' \leq k$ respectively. We state a lemma of Menger [26].

Lemma 9.2.2 [Menger] *Two non-adjacent nodes x and y are k-vertex-connected ($k \geq 1$), if after the removal of any set of at most $k-1$ vertices, x and y are (still) connected. Two adjacent nodes x and y with l edges that have x and y as end nodes are k-vertex-connected ($k \geq 1$), if after the removal of any set of at most $m \leq min\{k-1, l\}$ edges and $k-m-1$ vertices between x and y, x and y are (still) connected.*

If the removal of a set of vertices separates the vertices x and y (as described in the cases above), then that set is called a *cut set* for x and y.

In particular we have for $k=2$: two nodes are 2-vertex-connected iff they lie on a common simple cycle.

We call a set S of at least 2 nodes a *2vc-class* if the nodes are 2-vertex-connected and if there does not exist a node not in S that is 2-vertex-connected with the nodes of S (i.e., the class is maximal). Furthermore we define a *quasi class* to be any set of two nodes that are the end nodes of a cut edge. We call a set S of at least 2 nodes a *3vc-class* if the nodes are 3-vertex-connected and if there does not exist a node not in S that is 3-vertex-connected with the nodes of S (i.e., the class is maximal).

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.) Similarly, we can define 3-vertex-connected components of graphs (where now new edges are added, like for 3-edge-connectivity).

9.2.2 Problem Description

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

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

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

Is3vc(x, y): output whether x and y are two nodes in the graph that are 3-vertex-connected, and output the name of the 3-vertex-connected component (3vc-class) in which they both are contained (if any).

We call a problem the *2vc-problem* if the operations *insert* and *Is2vc* are considered, and we call it the *3vc-problem* if the operations *insert*, *Is2vc* and *Is3vc* are considered.

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 somewhere in the graph either the connectivity relation changes or, for the 2vc-problem, if the 2-vertex-connectivity relation changes, or, for the 3vc-problem, if the 2-vertex-connectivity or 3-vertex-connectivity relation changes. An insertion is called *nonessential* otherwise.

9.3 Two-Vertex-Connectivity

9.3.1 Graph Observations

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

Lemma 9.3.1 *Graph $2vc(G)$ is a forest, where each tree in $2vc(G)$ corresponds to a connected component in G, i.e., it consists of class nodes together with the nodes of a connected component in G.*

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

Lemma 9.3.2 *If edge (e, x, y) is inserted in graph G, then all the classes of which the class node is on the tree path in $2vc(G)$ between x and y form one new 2vc-class together, while the other 2vc-classes and quasi classes remain unchanged.*

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

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

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

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

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

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

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

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

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

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

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

We consider the insertion of an edge in a graph in terms of edge classes by means of Lemma 9.3.2. Suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G =< V, E >$. We distinguish three cases.

1. $c(x) \neq c(y)$. Then x and y are not connected in $SF(G)$. Hence, (e, x, y) connects two trees in $SF(G)$ that have to be joined into one tree.

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

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

9.3.2 Algorithms

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

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

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

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

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

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

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

A $2VC(i)$ structure is the above structure where $FRT_{2vc} = FRT(i)$ and where $UF_c = UF(i)$. Then we obtain the following result in a way similar to Subsection 8.3.

Theorem 9.3.3 *There exists a data structure and algorithms that solve the 2vc-problem and that can be implemented as a pointer/* $\log n$ *solution such that the following holds. The total time that is needed for all essential insertions starting from an empty graph of* n *nodes is* $O(n.i.a(i, n))$, *whereas a query and a nonessential insertion can be performed in* $O(i)$ *time. The initialisation can be performed in* $O(n)$ *time and the entire structure takes* $O(n)$ *space* $(i \geq 1,\ n \geq 2)$..

Now take α-FRT as FRT_{2vc} for a graph with n nodes, and take α-UF for UF_c. Then we obtain the following result in a way similar to Subsection 8.3, where now Theorem 7.7.2 is used instead of Theorem 7.7.1.

Theorem 9.3.4 *There exists a data structure and algorithms that solve the 2vc-problem and that can be implemented as a pointer/* $\log n$ *solution such that the following holds. The total time that is needed starting from an empty graph with* n *nodes is* $O(m.\alpha(m, n))$ *(where* m *is the number of edge insertions and queries), whereas the* f^{th} *operation can be performed in* $O(\alpha(f, n))$ *time if it is a query or a nonessential insertion. The initialisation can be performed in* $O(n)$ *time and the entire structure takes* $O(n)$ *space.*

The above theorem can be augmented to allow insertion of new nodes in the graph with a time complexity of $O(n + m.\alpha(m, n))$ (cf. Section 7.7). Then n, m and f in the theorem denote the current number at the moment of consideration.

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

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

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

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

9.4 Three-Vertex-Connectivity

In this section, we will briefly describe optimal solutions for the 3vc-problem, i.e., for maintaining the 3-vertex-connected components of general graphs, where edges may be inserted and where queries that ask whether two nodes are 3-vertex-connected are performed from time to time. The solutions have a time complexity of $O(n + m.\alpha(m,n))$ for m insertions and queries on graphs of n nodes. We will present the solution in a future report, and we will only give a rough sketch here. (We omit many details and special cases.)

First, we consider a 2-vertex-connected graph G containing at least three nodes. Henceforth, a 3vc-class of nodes in G is a maximal set of at least three nodes that are 3-vertex-connected, and we call a set of two nodes that is a maximal set of nodes that are 3-vertex-connected a 3vc-pair. G can be subdivided into so-called 3vc-classes, cycles, and bars in a way as follows. A pair of nodes $\{u, v\}$ that is a cut pair (i.e., a cut set) of G, and such that u and v are 3-vertex-connected, is called a 3vc-bar of G. Let $\{u, v\}$ be a 3vc-bar of G. The deletion of u and v from G yields a number of connected components $H_1, .. , H_k$. For each $H_i =< V_i, E_i >$, let H'_i be the subgraph of G induced by the node set $V_i \cup \{u, v\}$, where the edges between u and v are deleted (if any), and where three new edges between u and v are inserted. Then H'_i is 2-vertex-connected, and each 3vc-class of G is contained in exactly one H'_i. Moreover, H'_i and H'_j have exactly the two nodes u and v in common. Finally, each other 3vc-bar of G is contained in exactly one H'_i, and, moreover, it is a 3vc-bar of H'_i. This process can be continued on the resulting graphs, until we have obtained graphs that do not contain 3vc-bars. A final graph in this process can be a 3vc-class of nodes of G, or it can be a simple cycle if all multiple edges between two nodes are replaced by one edge. To denote the latter case, we just refer to the graph as a (simple) cycle. We call each pair of consecutive nodes on such a cycle a cycle bar. The result graphs are independent of the order of splitting.

In this way, we define the *cycle tree* of G corresponding to this splitting as follows. The nodes of the cycle tree are cycle nodes, class nodes, and bars, where each cycle node corresponds to one resulting cycle, each class node corresponds to one 3vc-class, and each bar is a pair of nodes of G that form a 3vc-bar or a cycle bar. The tree is defined recursively as follows. If all the nodes of G are 3-vertex-connected, and, hence, form one 3vc-class of nodes, then the tree is the class node for that 3vc-class. Otherwise, if G is a simple cycle, then the tree consists of the cycle node for that cycle, together with all bars corresponding to cycle bars of the cycle, where the cycle node has a link to each such bar. These links are ordered according to the order in which the bars occur in the cycle. Otherwise, there is a 3vc-bar $[u, v]$ for G. The deletion of u and v from G yields a number of graphs $H'_1, .. , H'_k$ as defined above. The tree for G is constructed as follows. Create the bar $[u, v]$. For each H'_i, obtain its cycle tree. If it has a bar $[u, v]$, then identify this with the above 3vc-bar $[u, v]$. Otherwise, make a link between $[u, v]$ and the unique 3vc-class in H'_i that contains u and v. This yields a tree that is independent of the order of splitting. We denote the tree by $Cyc(G)$. The collection of bars, classes, and cycles in $Cyc(G)$ in which a node x of G is contained is a subtree of $Cyc(G)$, called the subtree induced by x.

Suppose edge (x, y) is inserted in G, where x and y are not 3-vertex-connected. Let $P(x, y)$ be the tree path between the two subtrees induced by x and y in $Cyc(G)$. Then all 3vc-classes on $P(x, y)$, all nodes in bars on $P(x, y)$, and nodes x and y together form one new 3vc-class K resulting from the insertion. Moreover, each cycle occurring on $P(x, y)$ is split into two new cycles.

The implementation of a cycle tree has the following aspects. Each node x has a representative x_s for each cycle s in which it is contained. Each node x has one so-called *main representative* x_K and several so-called *additional representatives* $x_{K,i}$ for each 3vc-class K in which it is contained. The 3vc-classes are implemented as Union-Find structures on these representatives. The name of the set for the 3vc-class is the corresponding class node. Moreover, the cycles are implemented as Circular Split-Find structures on these representatives (in the order determined by the cycle). The name of the list for the cycle is the corresponding cycle node. The bars do not occur as such in the implementation. The representatives of nodes for cycles are also used for representing the 3vc-pairs.

The tree $Cyc(G)$ may be rooted in some class node or cycle node r. This is implemented as follows. For each node x in G, there is a pointer $max(x)$ to its representative in a 3vc-class or cycle that is closest to the root. Moreover, if $max(x)$ is a representative for a cycle s, then $max(x)$ has two pointers to its "neighbours" in s. A class node or a cycle node has two pointers to the representatives of the two nodes in its father bar in $Cyc(G)$, viz., to the representatives for that 3vc-class or cycle. (This implements the bar-cycle/class relation.)

A query that asks whether two nodes x and y are 3-vertex-connected can be performed by means of $max(x)$ and $max(y)$ and the classes or cycles containing these representatives, together with the nodes in the father bars of these classes or cycles. Edge insertions can be processed according to the above observations, by computing $P(x, y)$ in the rooted cycle tree, and by performing Unions and Splits.

The above structure of rooted cycle trees can be used to efficiently process insertions of edges in a 2-vertex-connected graph, while queries are performed from time to time. In [6], an optimal solution for such graphs, i.e., for graphs that are initially 2-vertex-connected, is presented (by means of SPQR trees). However, in a general graph, connected components and 2-vertex-connected components may be joined arbitrarily, and therefore cycle trees for 2-vertex-connected components must be linked from time to time. If we use a straightforward "redirection" technique to maintain the father pointers while trees are linked from time to time, then this takes $O(n. \log n)$ time for the redirections already.

Therefore, to represent the tree $Cyc(H)$ for some 2-vertex-connected component H of a general graph G, where joinings of 2-vertex-connected components must be performed efficiently (and hence the update of the corresponding cycle trees as well), we use cluster partitions of graphs and we define the notion of a *rooted cluster tree* for every 2-vertex-connected component H. Firstly, we still assume G to be a

2-vertex-connected graph to ease our description of the structure. We "change" the cycle tree $Cyc(G)$ by on the one hand augmenting the collection of nodes of G and on the other hand partitioning the thus obtained 3vc-classes into subclasses. We do this as follows.

For graph G and cycle tree $Cyc(G)$, an augmented cycle tree with a subclass augmentation consists of the following. Each (possibly extended) 3vc-class is partitioned into different (but not necessarily disjoint) subclasses consisting of at least three nodes each, where no subclass is a subset of another subclass. To each such subclass a new unique node is related as its name, called a *subclass node.* A 3vc-class is called *simple* if it is "partitioned" into one subclass only and it is called *multiple* otherwise. The subclass augmentation relates to each class node c in $Cyc(G)$ its collection of subclass nodes for c together with in case of a multiple class, a node, being the name of the class. An *augmented cycle tree* AF_G is a tree of which the nodes are the cycle nodes and the bars of $Cyc(G)$, together with the above subclass nodes and class names, and that satisfies

- for each multiple 3vc-class K of G, the subclass nodes and class name for K induce a subtree of AF_G, of which all the edges have this class name as end node,
- $Cyc(G)$ is obtained (up to edge names) from AF_G if for each 3vc-class K the subclass nodes and the class name for K are contracted to the class name of K, and
- a bar adjacent to a subclass node c in AF_G is contained in subclass c.

Our next augmentation of the structures is as follows. For the 2-vertex-connected graph $G =< V, E >$ we have a cluster partition $CP(G)$ and a collection of m-nodes, together with a related augmented cycle tree AF_G for G that satisfy the following. The vertex set of G is partitioned into (not necessarily disjoint) sets of at least three nodes, called *clusters*, such that every two clusters have at most two nodes in common and such that other constraints hold, such as: every cut pair for a pair of nodes in C is contained in C too. The collection of *m-nodes* ("multiple nodes") consists of the nodes that occur in at least two different clusters. To each cluster C, a set $Aux(C)$ of pairs of m-nodes in it is associated, where an m-node occurs in at most two pairs of $Aux(C)$. The following constraints are satisfied.

1. The subclasses of a 3vc-class K in AF_G are the intersections of K with the clusters that consist of at least three nodes. For a subclass contained in a cluster C we say that its subclass node is contained in cluster C.
2. Each cluster C corresponds to a subtree of AF_G, denoted by $tree(C)$. The subtree $tree(C)$ is a cycle tree on the nodes of C that represents the 3-vertex-connectivity relation on the nodes of C.

3. For every cycle node s in AF_G, there is one cluster C that contains cycle s, i.e., such that $tree(C)$ contains cycle node s.

4. For each two nodes x and y in AF_G there is at most one cluster C such that $tree(C)$ contains a bar $[x, y]$ that is incident with two edges of $tree(C)$ (that, hence, are edges between this bar and two nodes that are cycle nodes or subclass nodes in $tree(C)$). Bar $[x, y]$ of such a cluster is called an internal bar.

5. For every pair of nodes $\{x, y\}$ in G, there is at most one cluster C such that $\{x, y\} \in C \backslash Aux(C)$. For an internal bar $[x, y]$ of cluster C, $\{x, y\} \notin Aux(C)$.

In addition, an arithmetic constraint holds. Regarding the implementation of a cluster partition, one of the aspects is that each cluster is represented by a new node, called a cluster node, and for each m-node that occurs in a cluster, there is a representative for the node in that cluster. Similarly, for each multiple 3vc-class K, there are representatives for its subclasses in clusters. These form a set in a Union-Find structure with K as set name.

We next define rooted cluster trees on cluster partitions. A rooted cluster tree $RCT(G)$ for G on $CP(G)$ and an augmented cycle tree AF_G of G rooted in some node r are as follows. AF_G satisfies that class names are adjacent to subclass nodes only. $RCT(G)$ satisfies the following. Its nodes are the cluster nodes of $CP(G)$ together with additional class names for the multiple classes in AF_G. The edges form a rooted in-tree with some root R that is a cluster (each node has exactly one outgoing link except for the root). R is the (unique) cluster containing root r of AF_G. The edges can be distinguished as *class links* and as *cut links*. *A class link* is an edge between a class name K and a cluster that contains a subclass of K. It is called a class link for K. *A cut link* is an edge between two clusters. It has a reference to a pair of 3-vertex-connected m-nodes x and y that form a bar $[x, y]$ in AF_G and that are contained in both clusters. It is called a cut link for bar $[x, y]$. The links can be thought of as follows. The outgoing link of a cluster C is a class link to a class name K, if a subclass of K occurs in C, and if the subclass has K as its father in AF_G. Otherwise, the outgoing link is a cut link for a bar $[x, y]$, such that x and y occur in C and $\{x, y\}$ is a cut pair for any node in C (except for x and y) and (a node in) the root of AF_G. If a cluster C is the target of a cut link for a bar $[x, y]$, then it contains either a subclass node or cycle node that is the father of $[x, y]$ in AF_G, or a subclass of the class corresponding to the subclass that is the father of $[x, y]$ in AF_G

For each node x, $clus(x)$ is a maximal cluster containing x, i.e., $clus(x)$ contains x and all clusters on the root path of $clus(x)$ do not contain x. (In particular, $clus(x)$ is a reference to that cluster C together with a reference to the representative x_C.) Moreover, each cluster C contains a reference $Assoc(C)$ to (the representatives of)

$O(1)$ m-nodes in it (i.e., to their representatives for C). The latter is such that for each node x in C, we have $clus(x) = C$ or $x \in Assoc(C)$.

The idea of processing insertions is as follows. A path in the structure that is related to the path $P'(x,y)$ in AF_G can be obtained as follows. First a path P in $RCT(G)$ is obtained, which is the path between the two subtrees of clusters and class names in $RCT(G)$ that "contain" x or y, respectively. Then, in each cluster C on the path, a path in $tree(C)$ is obtained, specified by the links on P that are incident with C. In this way, an equivalent of $P'(x,y)$ in AF_G can be computed, viz., where each nonempty subsequence of $P'(x,y)$ of at most three nodes in AF_G related to the same 3vc-class may be replaced by another feasible nonempty subsequence for that class. The update is done by performing local updates inside clusters (updating the local tree $tree(C)$ for a cluster C), and by global updates on the cluster tree (joining 3vc-classes).

A query that asks whether two nodes x and y are 3-vertex-connected is performed by inspecting $clus(x)$ and $clus(y)$. If $clus(x)$ contains y, then a local query is performed in $clus(x)$, and similarly with x and y reversed. Otherwise, the father and grandfather of these clusters are used for the queries, which may result in the following intermediate output: a node x, a cluster C_x that contains x, and a subclass k_x in C_x (and similarly for y), such that if x and y are 3-vertex-connected, then k_x and k_y are subclasses of some class that contains both x and y. Then a local query inside cluster C_x is performed, computing whether x is contained in k_x, and similarly for y. For the local queries in a cluster C, the (information of) the local cycle tree $tree(C)$ is used.

To obtain the optimal time bound of $O(n + m.\alpha(m,n))$ for m operations on n nodes, we have the following approach to the implementation on a RAM. We develop a generalisation of the concept of microsets as presented in [13], that allow joining of microsets and queries within the α-bound. We call it *dynamic microsets*. For small 2-vertex-connected components (of size $O(\log\log n)$), a cluster tree consists of one cluster only, and the cluster is a dynamic microset. The dynamic microset for a cluster C encodes the structure of $tree(C)$ into one machine word. Now, if two or more small 2-vertex-connected components are joined, this is done by joining the dynamic microsets, as long as the resulting component is still small. A larger 2-vertex-connected component is represented by a cluster tree with clusters of larger size, where the cluster tree is a cluster tree for a coherent part of this component only, and by parts of the global cycle tree for this component. (E.g., for 2-vertex-connected components of $O(\log n)$ nodes, there are altogether $O(\frac{\log n}{\log\log n})$ clusters of size $O(\log\log n)$, and, similarly, for the remaining components of size $O(n)$, there are altogether $O(\frac{n}{\log n})$ clusters of size $O(\log n)$). For each such cluster C, the local cycle tree $tree(C)$ is implemented as a rooted cycle tree. Then, if two or more 2-vertex-connected components of similar size are joined, this is done by creating a new cluster for the newly arisen cycle (like the new cycle that arises in a similar case for 3-edge-connectivity if 2-edge-connected components are joined), and the rooted cluster trees are linked to it (after some adaptations for the above parts of the global cycle trees). These cluster trees are redirected w.r.t. the father relation, except for the largest one. (We omit many cases and details in this description.) The complexity for these linkings for e.g. the largest components is $O(l.\log l)$ steps, if there are l "large" clusters of size $O(\log n)$ altogether (where a step includes a Find). Since $l = O(\frac{n}{\log n})$, this yields $O(n)$ steps in total only, and a similar analysis holds for the other components.

To obtain the optimal time bound with a pointer/ $\log n$ solution, we use "contraction cluster partitions" of cluster partitions, in a way related to the division trees in Chapter 7. (This solution is more involved than the above RAM solution, once the dynamic microset structure is available.) The idea is to use a so-called contraction partition, where each cluster in it is again partitioned into a "local cluster partition", which, moreover, is implemented as a rooted cluster tree. Since, in general, it is not possible to contract a 2-vertex-connected component into an existing contraction cluster of another 2-vertex-connected component if these components are joined (like this is possible for linking trees in Chapter 7), these contraction partitions are more involved than the division trees. We use these contraction partitions to build a layered structure corresponding to Ackermann values. Then a similar approach for answering queries is possible, where now a "local" query in a cluster C in some layer j of the structure corresponds to a recurrent call in the cluster partition of cluster C in layer $j+1$, if it exists, and to a "normal" local call on $tree(C)$ otherwise.

In this way, we obtain an optimal solution for the 3vc-problem that has a time complexity of $O(n + m.\alpha(m,n))$ for m insertions and queries on graphs of n nodes.

9.5 Concluding Remarks

We have presented an optimal solution for the problem of maintaining the 2-vertex-connected components of graphs under insertions of edges and vertices. The solution takes $O(n + m.\alpha(m,n))$ time, starting from the graph $< \emptyset, \emptyset >$. Like for the structures 2EC(2) and 3EC(2) in the previous chapter, we conjecture that 2VC(2) is a fast and relatively simple structure for all practical situations, with constant-time queries and constant-time nonessential insertions. Finally, we have briefly described an optimal solution for the problem of maintaining the 3-vertex-connected components of graphs under insertions of edges and vertices. The detailed solution will be presented in a future report.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publ. Comp., Reading, Massachusetts, 1974.

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

[3] L. Banachowsky, A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem, Inf. Process. Lett. 11 (1980) 59-65.

[4] N. Blum, On the Single-Operation Worst-Case Time Complexity of the Disjoint Set Union Problem, SIAM J. Comput. 15 (1986) 1021-1024.

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

[6] G. Di Battista and R. Tamassia, On-Line Graph Algorithms with SPQR-Trees, Proc. 17^{th} Int. Colloq. on Automata, Languages, and Programming (ICALP) 1990, 598-611.

[7] P. v. Emde Boas, R. Kaas and E. Zijlstra, Design and Implementation of an Efficient Priority Queue, Math. Systems Theory 10 (1977) 99-127.

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

[9] G.N. Frederickson, Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications, SIAM J. Computing 14 (1985) 781-798.

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

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

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

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

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

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

[16] J.E. Hopcroft and J.D. Ullman, Set-Merging Algorithms, SIAM J. Comput. 2 (1973) 294-303.

[17] H. Imai and T. Asano, Dynamic Segment Intersection Search with Applications, Proc. 25th Ann. Symp. on Found. of Comp. Sci. (FOCS) 1984, 393-402.

[18] G.F. Italiano, Amortized Efficiency of a Path Retrieval Data Structure, Theoretical Computer Science 48 (1986) 273-281.

[19] G.F. Italiano, Finding Paths and Deleting Edges in Directed Acyclic Graphs, Inf. Process. Lett. 28 (1988) 5-11.

[20] D.E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley Publ. Comp., Reading, Massachusetts, 1968.

[21] A.N. Kolmogorov, On the Notion of Algorithms, Uspekhi Mat. Nauk. 8 (1953) 175-176.

[22] M.J. Lao, A New Data Structure for the Union-Find Problem, Inf. Process. Lett. 9 (1979) 39-45.

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

[24] J.A. La Poutré, A Fast and Optimal Algorithm for an Extension of the Split-Find Problem on Pointer Machines, Tech. Rep. RUU-CS-89-20, Utrecht University, 1989 (revised, final version to appear 1991).

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

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

[27] K. Mehlhorn, S. Näher and H. Alt, A Lower Bound for the Complexity of the Union-Split-Find Problem, SIAM J. Comput. 17 (1988) 1093-1102.

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

[29] H. Rohnert, A Dynamization of the All Pairs Least Cost Path Problem, In: K. Mehlhorn (Ed.), 2nd Annual Symposium on Theoretical Aspects of Computer Science 1985, Lecture Notes in Computer Science Vol. 182, Springer-Verlag, Berlin, pp. 279-286.

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

[31] R.E. Tarjan, Efficiency of a Good but Not Linear Set Union Algorithm, J. ACM 22 (1975) 215-225.

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

[33] R.E. Tarjan and J. van Leeuwen, Worst-Case Analysis of Set Union Algorithms, J. ACM 31 (1984) 245-281.

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

Bibliography

The contents of Chapter 3 was published as: J.A. La Poutré, New Techniques for the Union-Find Problem, Tech. Rep. RUU-CS-89-19, Utrecht University, 1989. It also appeared in: Proceedings of the 1^{st} Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 54-63.

The contents of Chapter 4 was published as: J.A. La Poutré, A Fast and Optimal Algorithm for an Extension of the Split-Find Problem on Pointer Machines, RUU-CS-89-20, Utrecht University, 1989, (revised, final version to appear 1991).

The contents of Chapter 5 was published as: J.A. La Poutré, Lower Bounds for the Union-Find Problem and the Split-Find Problem on Pointer Machines, RUU-CS-89-21, Utrecht University, 1989. It also appeared in: Proceedings of the 22^{nd} Annual ACM Symposium on Theory of Computing (STOC), 1990, pp. 34-44. The paper received the "1990 STOC Best Student Paper Award".

The contents of Chapter 6 (except for Subsection 6.3.4) was published as: J.A. La Poutré, J. van Leeuwen and M.H. Overmars, Maintenance of 2- and 3-Connected Components of Graphs, Part I: 2- and 3-Edge-Connected Components, Tech. Rep. RUU-CS-90-26, Utrecht University, 1990. The paper was presented at the French-Israeli Conference on Combinatorics and Algorithms, November 1988, Jerusalem, Israel.

The contents of Chapter 7, Chapter 8, Chapter 9 (except for Section 9.4), and Subsection 6.3.4 was published as: J.A. La Poutré, Maintenance of 2- and 3-Connected Components of Graphs, Part II: 2- and 3-Edge-Connected Components and 2-Vertex-Connected Components, Tech. Rep. RUU-CS-90-27, Utrecht University, 1990.

The contents of Section 9.4 will be published as: J.A. La Poutré, Maintenance of 2- and 3-Connected Components of Graphs, Part III: 3-Vertex-Connected Components, Tech. Rep. Utrecht University, 1991 (to appear).

Acknowledgements

I am grateful to my supervisors Jan van Leeuwen and Mark Overmars. Especially, I owe to them for introducing me into this interesting area of research and for their encouragement.

I also thank Erwin Bakker and Hans Bodlaender for proofreading parts of this thesis.

I thank my family and my friends for their interest in my work and for their support. Especially, many thanks are due to Ad Seebregts, who was always there for me when I needed it.

Samenvatting

Een graafalgoritme heet dynamisch ofwel on-line als het bepaalde informatie gerelateerd aan een graaf onderhoudt, terwijl de graaf regelmatig veranderd wordt. Zo'n verandering is bijvoorbeeld het toevoegen of verwijderen van een knoop of een "edge" (kant). Een dynamisch graafalgoritme zal gebruik maken van een geschikte dynamische datastructuur als datarepresentatie voor de graaf, en informatie over de oude graaf gebruiken om de gewenste informatie voor de nieuwe graaf te berekenen. Het ligt in de verwachting dat op deze manier een dynamisch algoritme een nieuwe oplossing niet steeds vanaf het begin hoeft te berekenen, dus met alleen de nieuwe graaf als input, en dat zo veel sneller een oplossing kan worden verkregen dan met een algoritme dat eenvoudigweg herberekent. In dit proefschrift worden enkele zeer efficiente, dynamische graafalgoritmen ontwikkeld, met inbegrip van de vereiste fundamentele datastructureringstechnieken.

Een probleem dat belangrijk is voor het verkrijgen van zeer efficiente datastructuren voor verscheidene graafproblemen, is het Union-Find probleem. Een alom bekend resultaat van Tarjan [31] is dat $n-1$ Union operaties en m Find operaties op een domein van n elementen kunnen worden uitgevoerd in $O(n+m.\alpha(m,n))$ tijd, waarbij $\alpha(m,n)$ de inverse Ackermann functie is. In hoofdstuk 3 ontwikkelen we een nieuwe benadering tot het probleem en bewijzen we dat de tijd voor de k-de Find operatie kan worden beperkt tot $O(\alpha(k,n))$, terwijl de totale complexiteit van de Unions en de Finds begrensd blijft tot $O(n+m.\alpha(m,n))$ tijd. Deze technieken blijken verwant te zijn met de technieken in [11] die gebruikt worden voor het Split-Find probleem. Omdat in alle praktische gevallen geldt dat $\alpha(k,n) = O(1)$, garanderen de nieuwe algoritmen dat Finds in essentie $O(1)$ tijd zijn, binnen de optimale grens voor het Union-Find probleem als geheel. De algoritmen kunnen worden uitgevoerd op een pointermachine en gebruiken geen padcompressie.

Het duale probleem is het Split-Find probleem. In [11] presenteerde Gabow een algoritme voor dit probleem met een tijdscomplexiteit van $O(n+m.\alpha(m,n))$ voor $n-1$ Split operaties en m Find operaties op een verzameling van n elementen; deze oplossing kan worden uitgevoerd op een pointermachine. In hoofdstuk 4 beschouwen we een generalisatie van het Split-Find probleem die toegepast kan worden in problemen zoals het bijhouden van de 3-edge-samenhangende en 3-vertex-samenhangende componenten van grafen. We presenteren oplossingen die dezelfde tijdscomplexiteit

hebben en die op een pointermachine kunnen worden uitgevoerd.

In 1979 bewees Tarjan [32] de bekende ondergrens voor de tijdscomplexiteit van het Union-Find probleem op pointermachines die aan de zogenaamde separatie conditie voldoen: voor alle n en alle $m \geq n$ bestaat er een rij van $n - 1$ Union en m Find operaties die tenminste $\Omega(n + m.\alpha(m, n))$ tijd kost op een pointermachine die aan de separatie conditie voldoet. In [3, 33] werd deze grens uitgebreid naar $\Omega(n + m.\alpha(m, n))$ voor alle n en m. In hoofdstuk 5 bewijzen we dat deze ondergrens geldt voor een algemene pointermachine (zonder de separatie conditie) en lossen hiermee een belangrijk vermoeden van Tarjan op, dat sedert 1979 open was. We bewijzen dat deze ondergrens ook geldt voor het Split-Find probleem.

In hoofdstuk 6 presenteren we een datastructuur om de 2- en 3-edge-samenhangende componenten van een graaf bij te houden tijdens het toevoegen van edges aan de graaf. Het toevoegen van e edges kost $O(n. \log n + e)$ tijd, startend vanuit de "lege" graaf met n knopen, dat wil zeggen, een graaf zonder edges. Hierbij kunnen tevens knopen worden toegevoegd (in dezelfde tijdsgrenzen, waarbij n dan het uiteindelijke aantal knopen is). Daarnaast kan met de datastructuur op elk moment het volgende type query in $O(1)$ tijd beantwoord worden: zijn twee gegeven knopen 2- of 3-edge-samenhangend? Ter verkrijging van betere tijdsgrenzen ontwikkelen we in hoofdstuk 7 een nieuwe datastructuur, "fractionally rooted tree" genaamd. Hiermee verkrijgen we in hoofdstuk 8 en hoofdstuk 9 optimale oplossingen voor de problemen van het bijhouden van de 2-edge-samenhangende, 3-edge-samenhangende en 2-vertex-samenhangende componenten van grafen. De oplossingen hebben een tijdscomplexiteit van $O(n + m.\alpha(m, n))$ voor m toevoegingen van edges en queries, startend vanuit een lege graaf met n knopen. Hierbij kunnen tevens knopen worden toegevoegd (in dezelfde tijdsgrenzen, waarbij n dan het uiteindelijke aantal knopen is). In hoofdstuk 9 beschrijven we tevens beknopt hoe de 3-vertex-samenhangende componenten van grafen in dezelfde optimale tijd kunnen worden bijgehouden.

Curriculum Vitae

Han La Poutré werd geboren op 15 juni 1962 te Breda. Hij deed Atheneum B aan het Mencia de Mendoza lyceum te Breda (1974 - 1980). Daarna studeerde hij Wiskunde aan de Technische Universiteit Eindhoven (1980 - 1986). Hij behaalde het kandidaatsexamen in juli 1984 (met lof) en het ingenieursexamen in september 1986 (met lof), met informatica als afstudeerrichting en discrete wiskunde als bijvak. Vanaf 1986 was hij werkzaam als wetenschappelijk assistent bij de Vakgroep Informatica van de Rijksuniversiteit te Utrecht.