New Techniques for the Union-Find Problem

J.A. La Poutré

RUU-CS-89-19
August 1989

Utrecht University

Department of Computer Science

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

New Techniques for the Union-Find Problem

J.A. La Poutré

Technical Report RUU-CS-89-19
August 1989

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

An extended abstract of this paper will appear in the Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 90), January 1990, San Francisco (U.S.A.).

New Techniques for the Union-Find Problem*

J.A. La Poutré

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

Abstract

A well-known result of Tarjan (cf. [10]) states that a program of up to n 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 this paper 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 FIND's remains bounded by $O(n + m.\alpha(m, n))$. The technique is part of a family of algorithms that can achieve various trade-offs in cost for the individual instructions. The new algorithm is important in all set-manipulation problems that require frequent FIND's. Because $\alpha(m, n)$ is $O(1)$ in all practical cases, the new algorithms guarantees that FIND's are essentially $O(1)$ worst case, within the optimal bound for the UNION-FIND problem as a whole. The algorithms run on a pointer machine and do not use any form of path compression.

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 set name 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

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

Union-Find programs is widely known as the disjoint set union problem or just the "Union-Find problem".
Several data structures and algorithms for the Union-Find problem have been developed. In [4] a set union algorithm was presented that takes $O((n+m).\log^* n)$ time for all Unions on n elements and for m Finds, where $\log^*$ denotes the "iterated log-function". In 1975 (cf. [10]) Tarjan considered the well-known set union algorithm that uses path compression. He proved that the worst-case time bound for this algorithm is $O(m.\alpha(m,n))$ for $n-1$ Unions and $m \geq n$ Finds, where function α is defined as follows. Firstly, the Ackermann function $A(i,x)$ is defined for $i,x \geq 0$ by

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

and the functional inverse of the Ackermann function is defined for $m,n \geq 1$ by

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

The algorithm can be run on a *pointer machine* (i.e., a machine model in which no arithmetic on memory addresses is possible) and moreover it 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. [11, 9] for a precise description). In [11] a lower bound was proved on the time complexity of any Union-Find program that can be run on a pointer machine and that satisfies the separation condition: a program of $n-1$ Unions and m Finds takes at least $\Omega(m.\alpha(m,n))$ time, if $m \geq n$. In [2] and [12] the lower bound was extended to $\Omega(n+m.\alpha(m,n))$ time for all n and m. Until now all known algorithms that can be run on a pointer machine satisfy the separation condition. Finally, in [5] the algorithm presented in [4] 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 paper 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 that is given by

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

(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 function.)

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. [10, 12]) 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 and results stated in this paper have the following applications. On the one hand, 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. [8]). Furthermore, because $\alpha(m,n)$ is $O(1)$ in all practical cases, the new algorithms 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 can be used to design an efficient (generalized) Split-Find structure that runs on a pointer machine (cf. [6]) (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 e.g. a structure to maintain 2- and 3-(edge-)connected components (cf. [8]). 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. Furthermore, the techniques can be used as a part of a proof for a general lower bound for the complexity of Union-Find algorithms on a pointer machine (cf. [7]).

As in [9, 10, 11, 12] 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 would appear to be different). However, our procedures are such that the 1-1 correspondences can easily be maintained when necessary.

The paper is organised as follows. In Section 2 the Ackermann function and pointer machines are considered. In Section 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 4. In Section 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 6 we consider the problem of insertions of elements.

2 Preliminaries

2.1 The Union-Find problem and pointer machines

We formulate the Union-Find problem in terms of nodes as follows (also cf. [9, 10, 11, 12]). 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 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.

As usual, a pointer is a specification of some node (e.g., its unique name or some other identifier) and a node may contain additional information in some field (e.g. a string, symbol or pointer).

The above description differs slightly from the description given in [10, 12] 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.

We describe a pointer machine informally. For a detailed description we refer to [10]. A pointer machine is a machine of which the memory exists of (equal) records. A record in memory is only accessible by means of a pointer to that record, i.e., a pointer can be seen as the internal address of the record in memory. Fields of a record may contain either data values or pointer values. However, no arithmetic on pointers is possible: the only operations on pointers is assignment and comparison. Therefore, a record can not be obtained by calculation of its address but only by means of following a sequence of pointers.

Now in the Union-Find problem on a pointer machine each node is *identified* with some unique record in the memory of that 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 [9, 10, 11, 12]): if these elements in the environment are records themselves, the 1-1 correspondence can be implemented by means of bidirectional pointers. 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 1 can be achieved.

Finally, in the algorithms we use lists (or equivalently: sets) that can be manipulated in the following way. We make use of lists of (pointers to) nodes that allow the following operations: two lists can be unioned or a node can be inserted in a list, both in $O(1)$ time, and a list can be enumerated or removed in linear time. Obviously, an implementation of a list as doubly linked linear list of nodes with additional pointers to its two end nodes will do (where the union can be performed just by concatenating the linear lists). We do not make these operations explicit in our algorithms, but only state the mathematical operations like "$\cup$" for the union of two lists and "$\{x\}$" for the list consisting of the element x only.

2.2 The Ackermann function

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

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

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

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

In fact, for every i, $A(i+1,x)$ is the result of x recurrent applications of function $A(i,.)$ (cf. $A(1,x)$, $A(2,x)$ and $A(3,x)$), as stated below.

Lemma 2.1 *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 2.2 *$A(i',x') \geq A(i,x)$ 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. [10].) □

Definition 2.3 *1. The row inverse a of the Ackermann function is defined by*

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

for $i,n \geq 0$.

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

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

for $m \geq 0$, $n \geq 1$. Here we take $\lceil 0 \rceil = 1$ in contrast to its usual definition.

Note that $\alpha(0,n) = \alpha(n,n)$. The above two definitions differ slightly from those appearing in [10, 11, 12]. However, it is easily shown that the differences are bounded by some additive constants (except for the functions $a(0,n)$ and $a(1,n)$). We state some lemmas.

Lemma 2.4

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

Proof. By (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), (1) and $i \geq 1$ we find

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

□

The following lemma shows the relation between two successive row inverse functions.

Lemma 2.5 *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 2.4 it follows by induction that $a(i,n) = a(i, a^{(j)}(i-1,n)) + j$ if $a^{(j)}(i-1,n) \geq 1$. Since $a(i,1) = 0$, this yields the required result. □

Thus we have for $n \geq 1$:

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

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

Lemma 2.6 *$a(i,n) \leq a(i',n')$ for $i \geq i'$, $n' \geq n$.*

Proof. By Lemma 2.2. □

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

Lemma 2.7 *$\alpha(m,n) = min\{i \geq 1 | a(i,n) \leq 4.\lceil m/n \rceil\}$.*

Corollary 2.8 *$\alpha(m',n') \leq \alpha(m,n)$ for $m' \geq m$ and $n' \leq n$.*

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

For simplicity, we extend the Ackermann function as follows:

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

Notation 2.9 *The set of all integers greater or equal to -1, is denoted by $\mathbb{N}_{-1}$. The number of elements of a set S is denoted by $|S|$.*

We state some more lemmas that we will need in the sequel.

Lemma 2.10 *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 2.4 gives

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

and therefore by (2)

$$a(i-1,n) > A(i, a(i,n) - 2). \tag{4}$$

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 2.2) in (4) yields

$$a(i-1,n) > 3.a(i,n) \tag{5}$$

□

Lemma 2.11 *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 2.7

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

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

Lemma 2.12 *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 2.7

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

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

$$\frac{f}{n} \in \mathbb{N} \;\wedge\; 4.\frac{f}{n} < a(\alpha(f+1,n),n) \leq 4.(\frac{f}{n}+1). \tag{6}$$

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 2.7 we must have $a(\alpha_1,n) \geq 5$. Applying Lemma 2.10 yields

$$a(\alpha_1-1,n) > 3.a(\alpha_1,n). \tag{7}$$

Combining (7) and (6) 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 (6) $\frac{f_1}{n}$ and $\frac{f_2}{n}$ are integers, this gives $f_2 \geq 3.f_1$. Lemma 2.11 provides the second inequality $f_1+1 > n$ since $\alpha(f_1+1,n) < \alpha(f_1,n) \leq \alpha(0,n)$. □

2.3 Obtaining Ackermann values

First 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 a integer, $n > 1$. Suppose we only need Ackermann values $A(i,x)$ to compare with numbers n' for $0 < n' < n$, i.e. evaluating a predicate $n' < A(i,x)$, and only for $i \geq 1$ and $-1 \leq x \leq a(i,n)$. (In our 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 $A(i,x)$ with $x \leq a(i,n)$.) If all Ackermann values that are at least n are replaced by the value ∞, then this does not influence the above comparisons. Note that by (2) $A(i,a(i,n)) \geq n$ and $A(i,a(i,n)-1) < n$. By Equation (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 2.2 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 $\alpha(n,n)+1$ are identical.

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

$$1 \leq i \leq \alpha(n,n)+1 \ \wedge \ -1 \leq x \leq a(i,n).$$

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 and we obtain a lemma.

Definition 2.13 *Let n_{ack} be an integer, $n_{ack} > 1$. An Ackermann table for n_{ack} with row number α_{ack} is a table A' with number $\alpha_{ack} = \alpha(n_{ack}, n_{ack})$ that contains values $A'(i,x)$ for*

$$1 \leq i \leq \alpha(n_{ack}, n_{ack}) + 1 \ \wedge\ -1 \leq x \leq a(i, n_{ack}) \tag{8}$$

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

Lemma 2.14 *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 Subsection we consider how to compute Ackermann tables for some n_{ack} and how to store them: firstly by a matrix array and afterwards by a pointer structure.

2.3.1 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 1, where n is taken instead of n_{ack}. We consider the algorithm briefly.

Note that in line 2 $i+1$ is used instead of just 1. This is for convenience in the sequel. In line 2 it is used that $A(1,x) = 2^x$ and that $2^{x+1} = 2^x + 2^x$. Furthermore, if row $i+1$ is computed, then ain has the value $a(i,n)$ (i.e., the "last" element of row i and hence $A'(i, ain) = \infty$). Moreover, the definition of the Ackermann function is used straightforward and finally 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. 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 $A'(i+1,x) + A'(i+1,x)$.

2.3.2 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 collection of nodes (records) that represent the entries $A'(i,x)$. Apart from n_{ack}, a positive integer i_0 is given too.

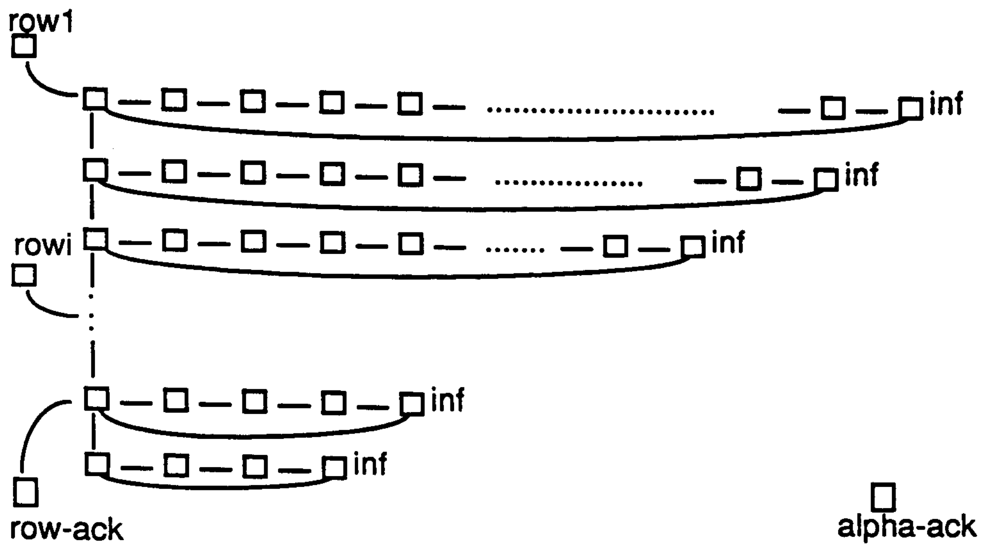
Figure 1: The (matrix) computation of the Ackermann table A'.

```
(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) $\alpha_{ack} := i-1$
```

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 first 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 $A'(i,x)$.) Moreover, the Ackermann net contains the parameter α_{ack}, the pointers row_{ack} and row_1 that point to the first nodes of rows α_{ack} and 1 respectively (i.e., 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. We call such a node net an Ackermann net for n_{ack}.

Now 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 w.r.t. the algorithm given in Figure 1. Firstly, together with 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 1 (viz., line 1 and 5) a node is created an it is appended to the linear "vertical" list that corresponds to column -1. Moreover, such a node is taken to be the first node of the "horizontal"

Figure 2: An Ackermann net.



list corresponding to row $i+1$. For each other value $A'(i+1,x)$ $(x \geq 0)$ that is computed in the algorithm of Figure 1, a node is created containing that value and it 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. Note that function A is monotone (cf. Lemma 2.2). Since during the computing of row $i+1$ the computed values $A'(i+1,x)$ are computed for increasing 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 (viz. "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 2.10 and Lemma 2.7 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 2.15 *An Ackermann net for n_{ack} can be computed in $O(\log n_{ack})$ time.*

Finally we consider augmenting 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 2.16 *Let an Ackermann net be given for $n_{ack} = 2^{x_{ack}}$. Then the net can be augmented to a net for $2.n_{ack} = 2^{x_{ack}+1}$ in $O(\alpha(n_{ack}, n_{ack}))$ time.*

Proof. Suppose we have an Ackermann net for $n_{ack} = 2^{x_{ack}}$. Now adapt 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 rest of the rows are augmented as follows: if row i ($i \leq \alpha_{ack}$) is not adapted, 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 b.m.o. 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. (1)) and append a node with value ∞ to it. Then adapt the values and pointers accordingly in node $(i + 1, -1)$. Otherwise, nothing need 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} with one and adapt the pointer row_{ack} accordingly. After this procedure is finished, adapt $n_{ack} := 2.n_{ack}$ and $x_{ack} := x_{ack} + 1$. Moreover, the pointer $rowi$ can easily be adapted 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. $\square$

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 Subsection 2.1. 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 Subsection 2.1. Function UNION(s, t, i) will be given in the sequel. Function FIND(x) is given in Figure 3. 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: 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]). The complex version will be given in Subsection 3.2, but first we outline the structure of UF(1) in Subsection 3.1.

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.

3.1 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 node s we have a parameter $weight(s, 1)$ that contains the size of $set(s, 1)$: $weight(s, 1) = \mid set(s, 1) \mid$.

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 4.

Figure 4: 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 $s$
```

The algorithm is based on changing the father pointers of the smallest of the two sets that are involved in the Union. We state here that 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. Subsection 2.1).

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 $lowindex(s, i) \in \mathsf{N}_{-1}$ that satisfies

$$2.A(i, lowindex(s, i)) \leq weight(s, i). \tag{9}$$

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

$$lowindex(s, i) < a(i, \lceil \frac{weight(s, i) + 1}{2} \rceil). \tag{10}$$

(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 (not being 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 $\mid sons(c) \mid \geq 2$)

$$\mid sons(c) \mid \geq 2.A(i, lowindex(s,i)). \tag{11}$$

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:

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

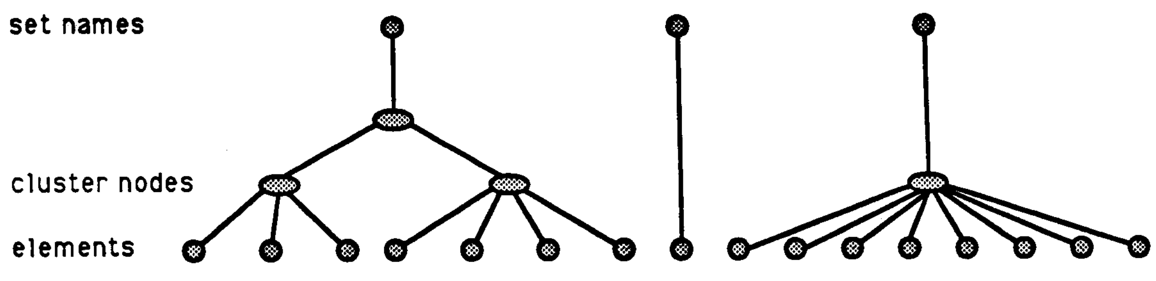
Finally, $clus(s,i)$ contains a pointer to an arbitrary cluster node in $clusset(s,i)$.

- 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 5). 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 5: 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 intitialisation 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 6. 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 7. 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 recurrent call is performed (lie 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 resulting unioned set.) 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 8). 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 "large enough" (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 9), 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 "large enough" (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. 10).

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).

Figure 6: 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
```

We want to state here that the value *lowindex*(s,i) can also be defined as the largest value that satisfies (9). In that case the corresponding UNION procedure is obtained from the procedure give in Figure 6 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 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).

Figure 7: 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 ∨ weight(s,i) = 1 ⟶ generate := sons(s);
(5) [] i > 1 ∧ weight(s,i) > 1 ⟶ clusset := generate(s,i − 1);
(6)                               generate := sons(clusset);
(7)                               dispose all nodes of clusset
(8) fi
```

Figure 8:

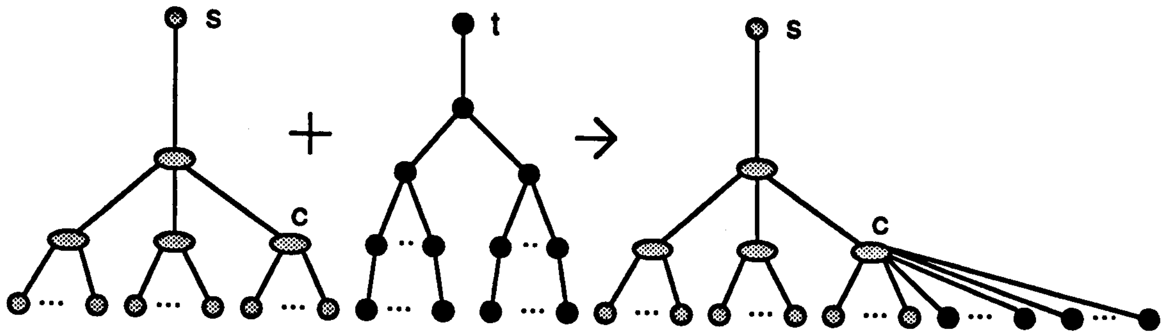


Figure 9:

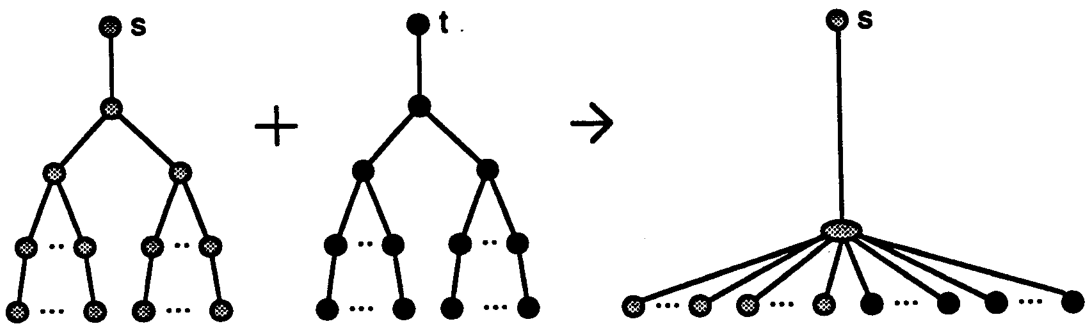
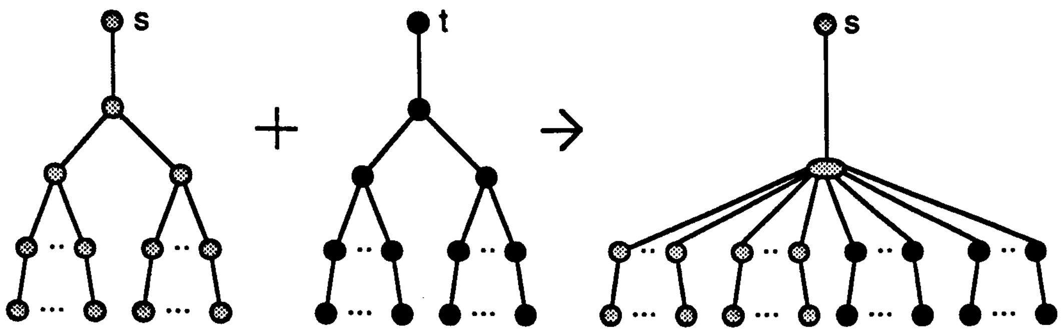


Figure 10:



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 6 (cf. lines 3, 4, 12 and 14) and by the definition of *lowindex* (cf. Section 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 2.14 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 6 can be performed.

The statement "remove (set) node t" occurring in lines 10 and 21 have 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 3.2 can easily be augmented to initialize the pointers described above without increasing the total time order.

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 4.

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.

Corresponding to [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, and 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 old 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.

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 $c_1'.\mid set(s,i) \mid$, since the number of cluster nodes for layer i is at most half the number of elements at layer i (cf. Subsection 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. (9) and (10)) 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.

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 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 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 6 that the recursive call is performed only if

$$\begin{aligned} &lowindex(s,i) = lowindex(t,i) \wedge \\ &weight(s,i) + weight(t,i) < 2.A(i, lowindex(s,i)+1). \end{aligned} \tag{12}$$

Since $-1 \leq l \leq 1$ implies that $2.max\{2.A(i,l), 1\} \geq 2.A(i, l+1)$, it follows by (12) and (9) 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 4.2 and by Subsection 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 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 $c.(\textit{number of } l\textit{-cluster nodes}).\ a(i-1,A(i,l+1))$. Since each l-cluster node has at least $2.A(i,l)$ elements as its sons (cf. (11)), 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 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 (1) and Lemma 2.4 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 $\frac{1}{2}c.n.a(i.n)$.

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.

4.4 The rest of the procedure

The execution of all statements together except those considered in subsections 4.1, 4.2 and 4.3, require 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.

4.5 The total complexity of Unions

Combining the results of subsections 4.1 to 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 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 2.15 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 3.1 (for $i = 1$) and Subsection 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 3.2). Therefore, we have established the following theorem.

Theorem 4.4 *A UF(i) structure and the algorithms that solve the Union-Find problem can be implemented on a pointer machine 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 4.5 *Let $i \geq 1$. Then there exists a structure and algorithms for the Union-Find problem that can be implemented on a pointer machine 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.*

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 exchanges in the worst-case complexities for Union and Find operations. This is expressed in the following theorems.

Theorem 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 on a pointer machine.*

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

Theorem 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 on a pointer machine.*

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 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 need to be done since then 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 2.7). 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). 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). For each set name s enumerate its elements e.g. in the way of procedure *generate** (cf. Figure 11) that, contrary to procedure *generate*, does not remove the intermediate nodes yet.

Figure 11: 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 ∨ weight(s,i) = 1 ⟶ generate := sons(s);
(4) [] i > 1 ∧ weight(s,i) > 1 ⟶ 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 2.12 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 4.4 and Lemma 2.7 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 2.11 we have if $i := \alpha(f,n)$:

$$8.(f+1) \geq n.a(i-1,n) \ \wedge \ f \geq n \geq 2. \tag{13}$$

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 4.4 and (13) it follows that each of these Finds is charged for $O(1)$ time. By Lemma 2.12 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

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

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

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 2.8) and where Lemma 2.12 provides the first unequality. This concludes the proof of the theorem. □

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. Subsection 2.3).

Theorem 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 run on a pointer machine 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 2.3.2). 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 2.16 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) 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 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 on a pointer machine.*

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 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 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 2.3.2) always points to node $(\alpha_{base}, -1)$ in the Ackermann net (which is always present). We first describe the strategy and prove Claim 6.3 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 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 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

$$(\, \alpha(f_{base} + f_{last}, 4n) < \alpha_{base} \wedge f_{last} \geq 2 f_{base} \,) \vee n = 4 n_{base} \tag{14}$$

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 2.16 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 5.2 and takes $O(n_{base})$ time (by Theorem 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 (14) 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 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 (14) becomes true then either an augmentation or a rebuilding needs to be performed anyway. Finally, it is easily seen that always

$$\alpha_{base} = \alpha(f_{base}, 4n_{base}) \wedge n_{base} \leq n \leq 4n_{base}. \tag{15}$$

Claim 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. (15)).

If at some moment $\alpha(f_{base} + f_{last}, 4n) < \alpha(f_{base}, 4n_{base})$ and hence by (15) $\alpha(f_{base} + f_{last}, 4n_{base}) < \alpha(f_{base}, 4n_{base})$ while no general update is started, it follows by the update condition (14) 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 (14) 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 2.12 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 (15) $\alpha(f_{base} + f_{last}, 4.n) \geq \alpha(f_{base}, 4.n_{base}) - 1$.

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 2.10 and Lemma 2.6 $a(i+2, 4n) < max\{\frac{1}{9}(x+2), 4\} \leq max\{\frac{1}{6}x, 4\}$. Now let $x = 4.\lceil\frac{f_{base}+f_{last}}{4n_{base}}\rceil$. Since by (15)

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

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 (14) again :

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

By Corollary 2.8 and Lemma 2.11 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 5.2 as follows.

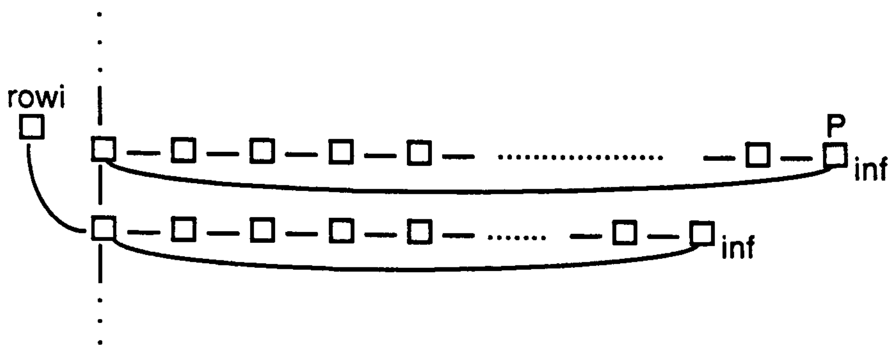
First we consider how to compute condition (14) only. Note that by Claim 6.3 the value $\alpha(f_{base} + f_{last}, 4n)$ only needs to be available if at least $2n_{base}$ Find operations or $2n_{base}$ Insert operations have been performed since the last general update, i.e., $f_{last} \geq 2n_{base}$ or $n - n_{base} \geq 2n_{base}$. Therefore we augment condition (14) to

$$(\,\alpha(f_{base} + f_{last}, 4n) < \alpha_{base} \wedge f_{last} \geq 2f_{base} \wedge f_{last} + n \geq 3n_{base}\,) \vee n = 4n_{base}. \quad (16)$$

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 2.7), 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 2.3.2). 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 12).

Figure 12:



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 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 6.3, Equation (15) 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

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 (20) at most $8.c.f_{base}+$ $16.c.n_{base}$ time. Therefore,

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

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 (20) and $n_{base} \le n \le 4n_{base}$. 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 $n_{base} \le n \le 4n_{base}$, Claim 6.3 and $\alpha(f_{base} + f_{last}, 4n) \le 2 + \alpha(f_{base} + f_{last}, n) \le 3\alpha(f_{base} + f_{last}, n) = 3\alpha(m,n)$)

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

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. □

7 Concluding Remarks

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

However, note that $\alpha(m,n) \le 3$ for $n \le 2^{2^{\cdot^{\cdot^{2}}}}\Big\}$ 65536 two's. Therefore in practice there is no need to perform transformations of structures like those occurring in Section 6: structure UF(3) suits for all practical situations. The time bound for the Unions in UF(3) is $c.n.a(3,n) \le 4.c.n$ for such n, where c is not too large a constant

(cf. Section 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 $\leq c.n.a(2,n) \leq 4.n$ time for $n \leq 2^{16} = 65536$ and take $\leq 5.c.n$ time for *very* large practical values $n \leq \left. 2^{2^{\cdot^{\cdot^{2}}}} \right\} \text{5 two's} = 2^{65536}$ (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 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.)

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 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 [3] (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.

8 Acknowledgements

I want to thank Jan van Leeuwen and Mark Overmars for very useful comments.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publ. Comp., Reading, Massachusets, 1974.

[2] 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.

[3] H.N. Gabow and R.E. Tarjan, A Linear-Time Algorithm for a Special Case of Disjoint Set Union, J. of Computer and System Sciences, no. 30, 1985, pp. 209-221.

[4] J.E. Hopcroft and J.D. Ullman, Set-Merging Algorithms, SIAM J. Comput. 2 (1973), 294-303.

[5] M.J. Lao, A New Data Structure for the Union-Find Problem, Inf. Process. Lett. 9 (1979), 39-45.

[6] J.A. La Poutré, A Fast and Optimal Algorithm for the Split-Find Problem on Pointer Machines, in preparation.

[7] J.A. La Poutré, Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines, in preparation.

[8] J.A. La Poutré, J. van Leeuwen and M.H. Overmars, Efficiently Maintaining 2- and 3- (Edge-)Connected Components of Graphs, in preparation.

[9] K. Mehlhorn, S. Näher and H. Alt, A Lower Bound for the Complexity of the Union-Split-Find Problem, SIAM J. Computing, vol. 17, no. 6 (1988), pp. 1093-1102.

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

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

[12] R.E. Tarjan and J. van Leeuwen, Worst-Case Analysis of Set Union Algorithms, J. ACM 31, No. 2, April 1984, pp. 245-281.