Data Structures and Algorithms for Disjoint Set Union Problems

Zvi Galil
Giuseppe F. Italiano

CUCS-473-89

Data Structures and Algorithms for Disjoint Set Union Problems *

Zvi Galil [1,2]

Giuseppe F. Italiano [1,3,4]

Abstract

This paper surveys algorithmic techniques and data structures that have been proposed to solve the set union problem and its variants. Their discovery required a new set of algorithmic tools that have proven useful in other areas. Special attention is devoted to recent extensions of the original set union problem, and some effort is made to provide a unifying theoretical framework for this growing body of algorithms.

Categories and Subject Descriptors: E.1 [Data Structures]: Trees; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems-*computations on discrete structures*; G.2.1 [Discrete Mathematics]: Combinatorics-*combinatorial algorithms*; G.2.2 [Discrete Mathematics]: Graph Theory-*graph algorithms*.

General Terms: Algorithms, Theory.

Additional Key Words and Phrases: Backtracking, equivalence algorithm, partition, set union, time complexity.

* Work supported in part by NSF Grants DCR-85-11713 and CCR-86-05353.

[1] Department of Computer Science, Columbia University, New York, NY, 10027.

[2] Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

[3] Dipartimento di Informatica e Sistemistica, Università di Roma, Rome, Italy.

[4] Supported in part by an IBM Graduate Fellowship.

1. Introduction

The *set union problem* has been widely studied during the past decades. The problem consists of performing a sequence of operations of the following two kinds on a collection of sets.

union(A,B) : combine the two sets A and B into a new set named A.

find(x) : return the name of the unique set containing the element x.

Initially, the collection consists of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. The name of set $\{i\}$ is i.

The set union problem has applications in a wide range of areas, including among others compiling COMMON and EQUIVALENCE statements in Fortran [Arden et al. 1961; Galler and Fischer 1964], implementing property grammars [Stearns and Lewis 1969; Stearns and Rosenkrantz 1969], computational geometry [Imai and Asano 1984; Mehlhorn 1984c; Mehlhorn and Näher 1986] and several combinatorial problems such as finding minimum spanning trees [Aho et al. 1974; Kerschenbaum and van Slyke 1972], computing least common ancestors in trees [Aho et al. 1973], solving off-line minimum problems [Gabow and Tarjan 1985; Hopcroft and Ullman 1973], finding dominators in graphs [Tarjan 1974] and checking flow graph reducibility [Tarjan 1973].

Very recently many variants of this problem have been introduced, in which the possibility of backtracking over the sequences of unions was taken into account [Apostolico et al. 1988; Gambosi et al. 1988b; 1988c; Mannila and Ukkonen 1986a; 1988; Westbrook and Tarjan 1987]. This was motivated by problems arising in Logic Programming interpreter memory management [Hogger 1984; Mannila and Ukkonen 1986b; 1986c; Warren and Pereira 1977], in incremental execution of logic programs [Mannila and Ukkonen 1988], and in implementation of search heuristics for resolution [Gambosi et al. 1988c; Ibaraki 1978; Pearl 1984].

In this paper we survey the most efficient algorithms designed for these problems. The model of computation considered is the *pointer machine* [Ben-Amram and Galil 1988; Knuth 1968; Kolmogorov 1953; Schönage 1980; Tarjan 1979a]. Its storage consists of an unbounded collection of records connected by pointers. Each record can contain an arbitrary amount of additional information and no arithmetic is allowed to compute the address of a record.

In this model two classes of algorithms were defined, called respectively *separable pointer algorithms* [Blum 1986; Tarjan 1979a] and *nonseparable pointer algorithms* [Mehlhorn et al. 1987].

Separable pointer algorithms run on a pointer machine and satisfy the *separability* assumption as defined in [Tarjan 1979a]. The rules an algorithm must obey to be in such a class are the following [Blum 1986; Tarjan 1979a] :

(i) The operations must be performed on line.

(ii) Each set element is a node of the data structure. There can be also additional nodes.

(iii) (*Separability*). After each operation, the data structure can be partitioned into subgraphs such that each subgraph corresponds exactly to a current set. No edge leads from a subgraph to another.

(iv) To perform find(x), the algorithm obtains the node v containing x and follows paths starting from v until it reaches the node which contains the name of the corresponding set.

(v) During any operation the algorithm may insert or delete any number of edges. The only restriction is that rule (iii) must hold after each operation.

The class of *nonseparable pointer algorithms* [Mehlhorn et al. 1987] does not require the separability assumption. The only requirement is that the number of pointers leaving each record must be bounded by some constant $c > 0$. More formally, rule (iii) above is replaced by the following rule, while the other four rules are left unchanged:

(iii) There exists a constant $c > 0$ such that there are at most c edges leaving a node.

We will see that often these two classes of pointer algorithms admit quite different upper and lower bounds for the same problems.

Another model of computation considered in this paper is the *random access machine*, whose memory consists of an unbounded sequence of registers, each of which is capable of holding an arbitrary integer. The main difference with pointer machines is that in random access machines the use of address arithmetic techniques is permitted. A formal definition of random access machines can be found in [Aho et al. 1974, pp. 5–14].

The remainder of the paper consists of eight sections. In section 2 we survey the most efficient algorithms known for solving the set union problem. Section 3 deals with the set union problem on adjacent intervals, while in section 4 data structures which allow us to undo the last union performed are presented. This result has been recently generalized in two different directions. First, in section 5 we describe techniques for undoing any union in the sequence of operations performed. Second, in section 6 and 7 we show how to undo any number of union operations (not only the last). In section 8, we use some of the presented techniques in order to obtain partially persistent data structures (as defined in [Driscoll et al. 1986]) for the set union problem. Finally, section 9 lists some open problems and concluding remarks.

2. The Set Union Problem

The *set union problem* consists of performing a sequence of union and find operations, starting from a collection of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. The initial name of set $\{i\}$ is i. Due to the definition of the union and find operations, there are two invariants which hold at any time. First, the sets are always disjoint and define a partition of the elements into equivalence classes. Second, the name of each set corresponds to to one of the items contained in the set itself.

A different version of this problem considers the following operation instead of unions.

unite(A,B) : combine the two sets A and B into a new set, whose name is either A or B.

Unite allows the name of the new set to be arbitrarily chosen. This is not a significant restriction in the applications, where one is mostly concerned on testing whether two

elements belong to the same equivalence class, no matter what the name of the class can be. Surprisingly, some extensions of the set union problem have quite different time bounds depending on whether unions or unites are considered. In the following, we will deal with unions unless otherwise specified.

2.1. Amortized complexity

In this section we will describe optimal algorithms for the set union problem [Tarjan 1975; Tarjan and van Leeuwen 1984], when the amortized time complexity is taken into account. We recall that the amortized time is the running time per operation averaged over a worst-case sequence of operations [Tarjan 1985]. For the sake of completeness, we first survey some of the basic algorithms that have been proposed in the literature [Aho et al. 1974; Fischer 1972; Galler and Fischer 1964].

Most of these algorithms represent sets making use of rooted trees, following a technique introduced by Galler and Fischer [1964]. Each tree corresponds to a set. Nodes of the tree correspond to elements of the set and the name of the set is stored in the root of the tree. In the remainder of the paper, we will assume that all the children of a node are linked in a doubly linked list and each node contains a pointer to its parent and a pointer to its leftmost child. This will guarantee that each edge of a tree can be traversed in both directions.

In the *quick-find* algorithm, every element points to the root of the tree. To perform a union(A, B), all the element of one set are made children of the tree root of the other. This leads to an $O(n)$ time complexity for each union and to $O(1)$ for each find.

A more efficient variant attributed to McIlroy and Morris [Aho et al. 1974] and known as *weighted quick-find*, uses the freedom implicit in each union operation according to the following rule.

union by size [Galler and Fischer 1964] : make the children of the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires that the size of each tree is maintained.

Although this rule does not improve the worst-case time complexity of each operation, it improves to $O(\log n)$ [1] the amortized complexity of a union [Aho et al. 1974].

The *quick-union* algorithm is able to support each union in $O(1)$ time and each find in $O(n)$ time [Galler and Fischer 1964]. The height of each tree can now be greater than 1. A union(A, B) is performed by making the tree root of one set child of the tree root of the other. A find(x) is performed by starting from the node x and by following the pointer to the parent until the tree root is reached. The name of the set stored in the tree root is then returned.

Also this time bound can be improved by using the freedom implicit in each union operation, according to one of the following two *union rules*. This gives rise to *weighted quick-union* algorithms.

[1] Throughout this paper all logarithms are assumed to be to the base 2, unless explicitely otherwise specified.

union by size [Galler and Fischer 1964] : make the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires maintaining the number of descendants for each node.

union by rank [Tarjan and van Leeuwen 1984] : make the root of the shallower tree point to the root of the other, arbitrarily breaking a tie. This requires maintaining the height of the subtree rooted at each node, in the following referred to as the *rank* of a node.

If the root of the tree containing A is made child of the root of the tree containing B, the names A and B are swapped between the roots. With either union rule, each union can be performed in $O(1)$ time and each find in $O(\log n)$ time [Galler and Fischer 1964; Tarjan and van Leeuwen 1984]. Finally, a better solution can be obtained if one of the following *compaction rules* is applied to the find path.

path compression [Hopcroft and Ullman 1973] : make every encountered node point to the tree root.

path splitting [van Leeuwen and van der Weide 1977; van der Weide 1980] : make every encountered node (except the last and the next to last) point to its grandparent.

path halving [van Leeuwen and van der Weide 1977; van der Weide 1980] : make every other encountered node (except the last and the next to last) point to its grandparent.

Combining the two choices of a union rule and the three choices of a compaction rule, six possible algorithms are obtained. They all have an $O(\alpha(m+n,n))$ amortized time complexity, where α is a very slowly growing function, a functional inverse of Ackermann's function [Ackermann 1928]. For the proof of the following theorem, the reader is referred to [Tarjan and van Leeuwen 1984].

Theorem 2.1.1. [Tarjan and van Leeuwen 1984] *The algorithms with either linking by size or linking by rank and either compression, splitting and halving run in* $O(n+m\alpha(m+n,n))$ *time on a sequence of at most* n *unions and* m *finds.*

No better amortized bound is possible for any separable pointer algorithm, as the following theorem shows.

Theorem 2.1.2. [Banachowski 1980; Tarjan 1979a; Tarjan and van Leeuwen 1984] *Any separable pointer algorithm requires* $\Omega(n+m\alpha(m+n,n))$ *worst-case time for processing a sequence of* n *unions and* m *finds.*

Proof : See Theorem 2 in [Tarjan and van Leeuwen 1984]. •

2.2. Single operation worst-case time complexity

The algorithms which use any union and any compaction rule have still single-operation worst-case time complexity $O(\log n)$ [Tarjan and van Leeuwen 1984]. Blum

[1986] proposed a data structure for the set union problem which supports each union and find in $O(\log n/\log\log n)$ time in the worst case. He also proved that no better bound is possible for any separable pointer algorithm.

The data structure used to establish the upper bound is called *k-UF tree*. For any $k \geq 2$, a *k-UF tree* is a rooted tree T such that:

(i) the root has at least two children;
(ii) each internal node has at least k children;
(iii) all leaves are at the same level.

As a trivial consequence of this definition, the height of a *k-UF* tree with n leaves is not greater than $\lceil \log_k n \rceil$.

We refer to the root of a *k-UF tree* as *fat* if it has more than k children, and as *slim* otherwise. A *k-UF tree* is said to be *fat* if its root is fat, otherwise it is referred to as *slim*.

Disjoint sets can be represented by *k-UF* trees as follows. The elements of the set are stored in the leaves and the name of the set is stored in the root. Furthermore, the root contains also the height of the tree and a bit specifying whether it is fat or slim.

A find(x) is performed as described in the previous section by starting from the leaf containing x and returning the name stored in the root. This can be accomplished in $O(\log_k n)$ worst-case time.

A union(A, B) is performed by first accessing the roots r_A and r_B of the corresponding *k-UF trees* T_A and T_B. Blum assumed that his algorithm obtained in constant time r_A and r_B before performing a union(A, B). If this is not the case, r_A and r_B can be obtained by means of two finds (i.e., find(A) and find(B)), due to the property that the name of each set corresponds to one of the items contained in the set itself. We now show how to unite the two *k-UF trees* T_A and T_B. Assume without loss of generality that $height(T_B) \leq height(T_A)$. Let v be the node on the path from the leftmost leaf of T_A to r_A with the same height as T_B. Clearly, v can be located by following the leftmost path starting from the root r_A for exactly $height(T_A) - height(T_B)$ steps. When implanting T_B on T_A, only three cases are possible, which give rise to three different types of unions.

Type 1 - Root r_B is fat (i.e., has more than k children) and v is not the root of T_A. Then r_B is made a sibling of v.

Type 2 - Root r_B is fat and v is the root of r_A. A new (slim) root r is created and both r_A and r_B are made children of r.

Type 3 - Root r_B is slim. All the children of r_B are made the rightmost children of v.

Theorem 2.2.1. [Blum 1986] *k-UF trees can support each union and find in* $O(\log n/\log\log n)$ *time in the worst case. Their space complexity is* $O(n)$.

Proof: Each find can be performed in $O(\log_k n)$ time. Each union(A, B) can require at most $O(\log_k n)$ time to locate the nodes r_A, r_B and v as defined above. Both type 1 and type 2 unions can be performed in constant time, while type 3 unions require at most $O(k)$ time, due to the definition of slim root. Choosing $k = \lceil \log n/\log\log n \rceil$, the claimed time bound is obtained. The space complexity derives easily from the fact that a *k-UF tree* with n leaves has at most $2n - 1$ nodes. Henceforth the forest of *k-UF trees* which store the disjoint sets requires at most a total of $O(n)$ space. •

Blum showed also that this bound is tight for the class of separable pointer algorithms. In particular, the reader is referred to [Blum 1986] for the proof of the following theorem.

Theorem 2.2.2. [Blum 1986] *Every separable pointer algorithm for the disjoint set union problem has single-operation worst-case time complexity at least* $\Omega(\log n/\log\log n)$.

2.3. Average case complexity

The expected running time of the basic algorithms described in section 2.1 has been investigated [Bollobás and Simon 1985; Doyle and Rivest 1976; Knuth and Schönage 1978; Yao 1976] under different assumptions on the distribution of the input sequence. In the rest of this section, we will assume that $O(n)$ union and find instructions are being performed. This is not a significant restriction for the asymptotical time complexity as shown for instance in [Hart and Sharir 1986].

Yao [1976] defined two different models of probability based on Random Graphs and proved that in one of these models the weighted quick-union algorithm executes n union and find instructions in $O(n)$ expected time.

On the other hand, Doyle and Rivest [1976] proved that the weighted quick-find algorithm requires between n and $2n$ steps in the average, assuming that each pair of sets is equally likely to be merged by a union operation. However, this assumption does not apply to situations where one is interested in joining sets containing two elements chosen independently with uniform probability.

Later on, this result was extended under a different model based on Random Graphs by Knuth and Schönage [1978]. Subsequently, Bollobás and Simon [1985] proved that the expected running time of the weighted quick-find algorithm is indeed $cn + o(n/\log n)$, where $c = 2.0487\ldots$.

The reader is referred to the original papers [Bollobás and Simon 1985; Doyle and Rivest 1976; Knuth and Schönage 1978; Yao 1976] for all the details concerning the analysis of the expected behavior of these algorithms.

2.4. Special linear cases

The most efficient algorithms for the set union problem are optimal for the class of separable pointer algorithms. As a consequence, in order to get a better bound, one should either consider a special case of set union or take advange of the more powerful capabilities of nonseparable pointer algorithms and random access machines [Aho et al. 1974]. Gabow and Tarjan [1985] used both these ideas to devise one algorithm which runs in linear time on a random access machine for a special case in which the structure of the unions is known in advance. This result is of theoretical interest as well as significant in several applications [Gabow and Tarjan 1985].

The problem can be formalized as follows. We are given a tree T containing n nodes which correspond to the initial n singleton sets. Denoting by $parent(v)$ the parent of the

node v in T, we have to perform a sequence of union and find operations such that each union can be only of the form $union(parent(v), v)$. For such a reason, T will be called the *static union tree* and the problem will be referred to as the *static tree set union*. Also the case in which the union tree can dynamically grow by means of new node insertions (referred to as *incremental tree set union*) can be solved in linear time. In the following, we will briefly sketch the solution of the static tree set union problem, referring the reader to [Gabow and Tarjan 1985] for incremental tree set union.

Gabow and Tarjan's static tree algorithm partitions the nodes of T in suitably chosen small sets, called *microsets*. Each microset contains less than b nodes (where b is such that $b = \Omega(\log\log n)$ and $b = O(\log n/\log\log n)$), and there are at most $O(n/b)$ microsets. To each microset S a node $r \notin S$ is associated, referred to as the *root* of S, such that $S \cup \{r\}$ induces a subtree of T with root r.

The roots of the microsets are maintained as a collection of disjoint sets, called *macrosets*. Macrosets allow to access and manipulate microsets.

The basic ideas underlying the algorithm are the following. First, the *a priori* knowledge about the static union tree allows to precompute the answers to the operations to be performed in microsets by using some table look-up. Second, we apply any one of the six optimal algorithms described in section 2.1 to maintain the macrosets. By combining these two techniques, a linear-time algorithm for this special case of the set union problem can be obtained. The algorithm is quite complicated and all the low-level details as well as the proof of the following theorem can be found in [Gabow and Tarjan 1985].

Theorem 2.4.1. [Gabow and Tarjan 1985] *If the knowledge about the union tree is available in advance, each union and find operation can be supported in $O(1)$ amortized time. The total space required is $O(n)$.*

Very recently Loebl and Nešetřil [1988] presented a linear-time algorithm for another special case of the set union problem. They considered sequences of unions and finds with a constraint on the subsequence of finds. Namely, the finds are listed in a *postorder* fashion, where a postorder is a linear ordering of the leaves induced by a drawing of the tree in the plane. In this framework, they proved that such sequences of union and find operations can be performed in linear time, thus getting $O(1)$ amortized time per operation. However a slightly more general class of input sequences, denoted by *local postorder*, was proved not to be linear (even if its rate of growth is unprovable in the theory of finite sets). A preliminary version of these results was reported in [Loebl and Nešetřil 1988].

3. The Set Union Problem on Intervals

In this section, we shall restrict our attention to the *set union problem on intervals*. This problem can be defined in the following general framework [Mehlhorn et al. 1987]. Perform a sequence of the following three operations on a linear list $\{1, 2, \ldots, n\}$ of items

union(x) : given the marked item x, unmark this item.

find(x) : given the item x, return $y = \min\{z \mid z \geq x \text{ and } z \text{ is marked }\}$.

split(x) : given the unmarked item x, mark this item.

Marked items partition the list into adjacent intervals. A union(x) joins two adjacent intervals, a find(x) returns the right endpoint of the interval containing x and a split(x) splits the interval containing x. Adopting the same terminology used in [Mehlhorn et al. 1987], we will refer to the set union problem on intervals as the *union-split-find problem*. After having tackled this problem, we will consider two particular cases: the *union-find problem* and the *split-find problem*, where only union, find and respectively split and find operations are allowed.

The union-split-find problem and its subproblems have applications in a wide range of areas, including computational geometry [Imai and Asano 1984; Mehlhorn 1984c; Mehlhorn and Näher 1986], shortest paths [Mehlhorn 1984b; Ahuja et al. 1988] and the longest common subsequence problem [Aho et al. 1983; Apostolico and Guerra 1987].

3.1. Union-Split-Find

In this section we will describe optimal separable and nonseparable pointer algorithms for the union-split-find problem. The best separable algorithm for this problem runs in $O(\log n)$ worst-case time for each operation, while nonseparable pointer algorithms require only $O(\log \log n)$ worst-case time for each operation. In both cases, no better bound is possible.

As far as separable pointer algorithms are concerned, the upper bound can be easily obtained by means of balanced trees [Aho et al. 1974; Adelson-Velskii and Landis 1962; Mehlhorn 1984a; Nievergelt and Reingold 1973], while for the proof of the following lower bound the reader is referred to [Mehlhorn et al. 1987].

Theorem 3.1.1. [Mehlhorn et al. 1987] *For any separable pointer algorithm, both the worst-case per operation time complexity of the split-find problem and the amortized time complexity of the union-split-find problem are* $\Omega(\log n)$.

Turning to nonseparable pointer algorithms, the upper bound can be found in [Karlsson 1984; Mehlhorn and Näher 1986; van Emde Boas 1977; van Emde Boas et al. 1977]. In particular, van Emde Boas et al. [1977] introduced a priority queue which supports among other operations *insert*, *delete* and *successor* on a set whose elements belong to a fixed universe $S = \{1, 2, \ldots, n\}$. The time required by each of those operation is $O(\log \log n)$. Originally, the space was $O(n \log \log n)$ but later was improved to $O(n)$ by van Emde Boas [van Emde Boas 1977]. A detailed description of the data structure and its time complexity can be found in [van Emde Boas 1977; van Emde Boas et al. 1977]. The above operations correspond respectively to union, split and find, and therefore the following theorem easily follows.

Theorem 3.1.2. [van Emde Boas 1977] *There exists a data structure supporting each union, find and split in $O(\log\log n)$ worst-case time. The space required is $O(n)$.*

Proof : See Theorem 3 in [van Emde Boas 1977] •

The bound obtained by means of van Emde Boas' priority queue is tight, as the following theorem shows.

Theorem 3.1.3. [Mehlhorn et al. 1987] *For any nonseparable pointer algorithm, both the worst-case per operation time complexity of the split-find problem and the amortized time complexity of the union-split-find problem are $\Omega(\log\log n)$.*

Proof : See Theorem 1 in [Mehlhorn et al. 1987]. •

Notice that this implies that for the union-split-find problem the separability assumption causes an exponential loss of efficiency. It is still open whether the use of nonseparable pointer algorithms can improve the time complexity of the more general set union problem.

3.2. Union-Find

The union-find problem is a restriction of the set union problem described in section 2, when only adjacent intervals are allowed to be joined. Henceforth, both the $O(\alpha(m+n,n))$ amortized bound given in Theorem 2.1.1 and the $O(\log n/\log\log n)$ single-operation worst-case bound given in Theorem 2.2.1 still hold.

However, while Tarjan's proof of the $\Omega(\alpha(m+n,n))$ amortized lower bound works also for the union-find problem, Blum's proof does not seem to be easily adaptable to the new problem. Hence, it remains an open problem whether no better bound than $O(\log n/\log\log n)$ is possible for the single-operation worst-case time complexity of separable pointer algorithms.

It is also open whether less than $O(\log\log n)$ worst-case per operation time can be achieved for nonseparable pointer algorithms. Gabow and Tarjan used the data structure described in section 2.4 to obtain an $O(1)$ amortized time on a random access machine.

3.3. Split-Find

According to Theorems 3.1.1, 3.1.2 and 3.1.3, the two algorithms given for the more general union-split-find problem, are still optimal for the single-operation worst-case time complexity of the split-find problem. As a result, each split and find operation can be supported in $\Theta(\log n)$ and in $\Theta(\log\log n)$ time respectively in the separable and nonseparable pointer machine model.

The amortized complexity of this problem can be reduced to $O(\log^* n)$, where $\log^* n$ is the *iterated logarithm* function [1], as shown by Hopcroft and Ullman [1973]. Their algorithm

[1] $\log^* n = \min\{i \mid \log^{[i]} n \leq 1\}$, where $\log^{[i]} n = \log\log^{[i-1]} n$ for $i > 0$ and $\log^{[0]} n = n$. Roughly speaking, it is the number of times the logarithm must be taken to obtain a number less than one.

is based upon an extension of an idea by Stearns and Rosenkrantz [1969]. The basic data structure is a tree, for which each node at level i, $i \geq 1$, has at most $2^{F(i-1)}$ children, where $F(i) = F(i-1)2^{F(i-1)}$, for $i \geq 1$, and $F(0) = 1$. A node is said to be *complete* either if it is at level 0 or if it is at level $i \geq 1$ and has $2^{F(i-1)}$ children, all of which are complete. The invariant maintained for the data structure is that no node has more than two incomplete children. Moreover, the incomplete children (if any) will be leftmost and rightmost. As in the usual tree data structures for set union, the name of a set is stored in the tree root.

At the beginning, such a tree with n leaves is created. Its height is $O(\log^* n)$ and therefore a find(x) carried out as usual will require $O(\log^* n)$ time to return the name of the set. To perform a split(x), we start at the leaf corresponding to x and traverse the path to the root to partition the tree into two trees. It is possible to show that using this data structure, the amortized cost of a split is $O(\log^* n)$ [Hopcroft and Ullman 1973].

This bound can be further improved to $O(\alpha(m, n))$ as showed by Gabow [1985]. The algorithm used to establish this upper bound relies on a sophisticated partition of the items contained in each set. The underlying data structure is quite complicated and the reader is referred to [Gabow 1985] for the proof of the following theorem.

Theorem 3.3.1. [Gabow 1985] *There exists a data structure supporting a sequence of m find and split operations in $O(m\alpha(m, n))$ worst-case time. The space required is $O(n)$.*

It is still open whether an amortized bound less than $O(\alpha(m, n))$ can be obtained on a pointer machine. Gabow and Tarjan, using the power of a random access machine, were able to achieve $\Theta(1)$ amortized time. This bound is obtained by employing a slight variant of the data structure sketched in section 2.4. The details can be found in [Gabow and Tarjan 1985].

4. The Set Union Problem with Deunions

Mannila and Ukkonen [1986a] defined a generalization of the set union problem, referred to in the following as *set union with deunions*, in which in addition to union and find the following extra operation is allowed.

deunion : undo the most recently performed union operation not yet undone.

Motivations for studying this problem arise in Prolog interpreter memory management without function symbols [Hogger 1984; Mannila and Ukkonen 1986b; 1986c; Warren and Pereira 1977]. Variables of Prolog clauses correspond to the elements of the sets, unifications correspond to unions and backtracking corresponds to deunions [Mannila and Ukkonen 1986b].

Recently, the amortized complexity of set union with deunions was characterized by Westbrook and Tarjan [1987], who derived a $\Theta(\log n / \log\log n)$ upper and lower bound. The upper bound is obtained by extending the path compaction techniques described in

the previous sections in order to deal with deunions. The lower bound holds for separable pointer algorithms. The same upper and lower bounds hold also for the single-operation worst-case time complexity of the problem.

4.1. Amortized complexity

In this section, we will describe $\Theta(\log n/\log\log n)$ amortized algorithms for the set union problem with deunions [Mannila and Ukkonen 1987; Westbrook and Tarjan 1987]. They all use one of the union rules combined with path splitting and path halving. Path compression with any one of the union rules leads to an $O(\log n)$ amortized algorithm, as it can be seen by first performing $n-1$ unions which build a binomial tree [Tarjan and van Leeuwen 1984] of depth $O(\log n)$ and then by repeatedly carrying out a find on the deepest leaf, a deunion and a redo of that union.

In the following, a union operation not yet undone will be referred to as *live*, and as *dead* otherwise. To handle deunions, a *union stack* is maintained, which contains the roots made nonroots by live unions. Furthermore, for each node x a *node stack* $P(x)$ is maintained, which contains the pointers leaving x created either by unions or by finds. During a path compaction caused by a find, the old pointer leaving x is left in $P(x)$ and each newly created pointer (x, y) is pushed onto $P(x)$. The bottommost pointer on these stacks is created by a union and will be referred to as a *union pointer*. The other pointers are created by path compaction and are called *find pointers*. Each of these pointers is associated to a unique union operation, the one whose undoing would invalidate the pointer. The pointer is said to be *live* if the associated union operation is live, and it is said to be *dead* otherwise.

Unions are performed as in the set union problem, except that for each union a new item is pushed onto the union stack, containing the tree root made nonroot and some bookkeeping information about the set name and either size or rank. To perform a deunion, the top element is popped from the union stack and the pointer leaving that node is deleted. The extra information stored in the union stack allows to maintain correctly set names and either sizes or ranks.

There are actually two versions of these algorithms, depending on when dead pointers are destroyed from the data structure. *Eager algorithms* pop pointers from the node stacks as soon as they become dead (i.e., after a deunion operation). On the other hand, *lazy algorithms* destroy dead pointers in a lazy fashion while performing subsequent union and find operations. Combined with the allowed union and compaction rules, this gives a total of eight algorithms. They all have the same time and space complexity, as the following theorem shows.

Theorem 4.1.1. *Either union by size or union by rank in combination with either path splitting or path halving gives both eager and lazy algorithms which run in $O(\log n/\log\log n)$ amortized time for operation. The space required by all these algorithms is $O(n)$.*

Proof : The time bounds for the eager and lazy algorithms follow from Theorem 1 and Theorem 2 in [Westbrook and Tarjan 1987]. The space bound for the eager algorithms

was recently improved from $O(n \log n)$ to $O(n)$ [Mannila and Ukkonen 1987; Westbrook and Tarjan 1987]. Also the space complexity of the lazy algorithms can be shown to be $O(n)$ by following the stamping technique introduced by Gambosi et al. [1988b], which allows to establish that the lazy algorithms require no more space than their eager counterparts. •

This bound is tight as shown in the following theorem, whose proof can be found in [Westbrook and Tarjan 1987].

Theorem 4.1.2. [Westbrook and Tarjan 1987] *Every separable pointer algorithm for the set union problem with deunions requires at least* $\Omega(\log n / \log \log n)$ *amortized time per operation.*

4.2. Single operation worst-case time complexity

In this section we will show that an extension of Blum's data structure described in Section 2.2 can support also deunions. As a result, the augmented data structure will support each union, find and deunion in $O(\log n / \log \log n)$ time in the worst case, with an $O(n)$ space usage.

For any integer $k \geq 2$, a *separator k-tree* (*sk-tree*, in short) is a rooted tree T with nodes uniquely labeled in $[1, n]$ and such that:

(i) T is a *k-UF tree*;

(ii) each pointer of T can be a simple pointer or a *separator*. To each separator an integer in $[1, n]$ is associated.

In the following we will assume $k = \lceil \log n / \log \log n \rceil$. *Sk-trees* are represented as follows. For each node v, the children of v are linearly ordered from left to right in a doubly linked list. Appropriate marks identify separator pointers.

As in the case of *k-UF* trees, an sk-tree T corresponds to a set, the elements of the set being stored in the leaves of T and the tree root containing the name of the set.

A find(x) is performed as in *k-UF* trees, and therefore in time proportional to the height of T.

Also a union(A, B) is performed as in the case of *k-UF* trees. The only difference is that now when performing a type 3 union, which makes all the pointers previously entering a root r_2 enter a node v in another sk-tree T_1, the pointer connecting the leftmost child of r_2 to v is marked a separator, and the label of r_2 (i.e. the old name of the set represented by T_2) is stored in the separator.

Furthermore, because of the linear order on the children of each node, each union can be implicitely described by its *characteristic pointer*, defined as follows. The characteristic pointer of a type 1 union is the only new pointer introduced. The characteristic pointer of a type 2 union is the leftmost pointer introduced (only two new pointers are introduced and both point to the same root). Finally, the characteristic pointer of a type 3 union is the separator associated to that union. The node from which the characteristic pointer is

leaving is called the *characteristic node*. The introduction of characteristic pointers and characteristic nodes enables one to perform deunions. In addition, each time a union is executed some extra information is stored in a *union stack*, as follows. Following each union operation, a pointer to the characteristic node is pushed onto the union stack, along with the type identifier (1, 2 or 3) of that union.

We now describe how deunions can be performed. Type 1 and type 2 unions are easily undone in constant time, by popping the top item in the union stack and accessing the characteristic node. In case of a type 1 union, the pointer to the parent of the characteristic node is made *null*. In case of a type 2 union, both the characteristic node and its sibling have the pointer to the parent set to *null*. To undo a type 3 union, we access the separator pointed to by the top of the stack and disconnect this pointer and all the pointers to its right. All the nodes now detached from the tree are made children of a new root to which the name stored in the separator is assigned. By the definition of type 3 union, this requires $O(k)$ time.

The correctness hinges on the following lemma.

Lemma 4.2.1. *Sk-trees are correctly maintained during any sequence of union, find and deunion operations.*

Proof : Since finds do not modify the data structure and unions are performed as in *k-UF* trees, it is only necessary to prove that any union-deunion pair leaves the structure unchanged.

For type 1 and type 2 unions, this is trivially true.

For what concerns type 3 unions, we first notice that to each separator exactly one name (i.e., the name erased by the corresponding union) is associated. In fact, when a pointer e is made separator, it is moved and attached to the right of some pre-existing pointers. Hence, it is no longer the lefmost pointer entering a node and it cannot result again in the separator associated to any other subsequent union operation.

Let us prove that a type 3 union-deunion pair leaves the structure unchanged. In fact, a type 3 union can only move pointers to the right of its corresponding separator, while maintaining their relative left-to-right order and storing the name erased in the separator. The corrsponding type 3 deunion will redirect all the pointers to the right of the separator to a new node which gets the only name stored in the separator itself (i.e., the old name of the set), while maintaining again the left-to-right order between pointers.

This guarantees that the structure of sk-trees is maintained in any sequence of union. find and deunion operations. •

Using sk-trees each union. find and deunion operation requires $O(\log n/\log\log n)$ time in the worst case. No better bound is possible for any separable pointer algorithm.

Theorem 4.2.1. [Apostolico et al. 1988] *Sk-trees support each union, find and deunion in* $\Theta(\log n/\log\log n)$ *worst-case time. The total space required is* $O(n)$.

Proof : Unions and deunions require $O(k)$ worst-case time. Any find take time proportional to the height of an sk-tree. Since an sk-tree is subject to the same height bound as a *k-UF* tree (namely $O(\log_k n)$), the time bound now follows immediately. As

for the space complexity, sk-trees require the same space as k-UF trees. Since the stack records correspond to unions not yet undone, and there are at most $n-1$ of these, the data structure requires a total of $O(n)$ space.

The lower bound is a trivial consequence of Theorem 2.2.2. •

5. The Set Union Problem with Arbitrary Deunions

Mannila and Ukkonen [1988] introduced another variant of the set union problem, called *set union problem with arbitrary deunions*. This problem consists of maintaining a collection of disjoint sets under an intermixed sequence of the following operations.

union(x,y,A) : combine the sets containing elements x and y into a new set named A.

find(x) : output the name of the set which currently contains element x.

deunion(i) : undo the i-th union so far performed.

After a deunion(i), the name of the sets are as if the i-th union had never occurred.

Motivations for studying this problem arise in the incremental execution of logic programs [Mannila and Ukkonen 1988]. In their paper, Mannila and Ukkonen proved an $\Omega(\log n)$ lower bound for the set union problem with arbitrary deunions. They gave also one algorithm with an $O(\log n)$ single-operation worst-case time complexity. Unfortunately, their algorithm is not correct. In the following, we present an optimal algorithm whose running time is $O(\log n)$ per operation in the worst case. First, let us turn to the lower bound argument.

Theorem 5.1. [Mannila and Ukkonen 1988] *The amortized complexity of the set union problem with arbitrary deunions is* $\Omega(\log n)$ *for separable pointer algorithms and* $\Omega(\log \log n)$ *for nonseparable pointer algorithms.*

Proof : By reduction to the union-find-split problem [Mehlhorn et al. 1987], as characterized in Theorems 3.1.1 and 3.1.3. For the details of the reduction see Theorem 5 in [Mannila and Ukkonen 1988]. •

The upper bound argument used by Mannila and Ukkonen [1988] is based on the linking and cutting trees of Sleator and Tarjan [1983; 1985]. The linking and cutting trees maintain a collection of rooted trees under an arbitrary sequence of operations chosen from a suitable repertoire. For our purposes, it suffices to consider only the following ones.

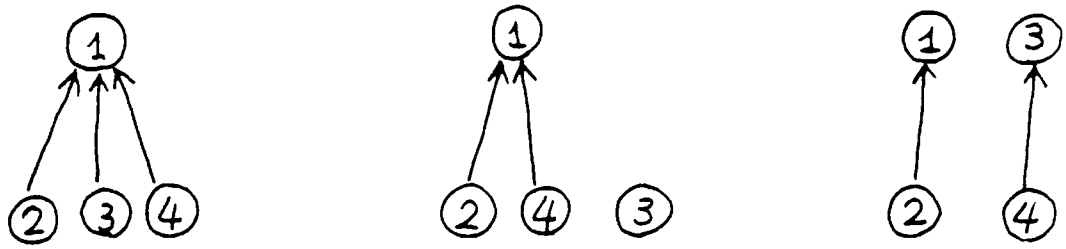
findroot(x) : output the root of the tree containing the node x;

link(x,y) : add an edge from a root x to a node y in a different tree, thus combining the two trees containing x and y into one new tree;

cut(x) : delete the edge from x to its parent, thus giving rise to two new trees and destroying the old one.

In [Mannila and Ukkonen 1988] the usual rooted tree representation of sets with the names stored in the root is used. Furthermore, each find is implemented by means of a

findroot, while a union(x, y, A) is performed by linking the roots of the two trees containing the nodes x and y. A deunion(i) is carried out by simply cutting the edge introduced by the i-th union union. Unfortunately, linking the tree roots does not yield a correct algorithm, as the counterexample shown in Figure 5.1 points out.



a) Tree obtained after performing union(1,2,A), union(2,3,B) and union(3,4,C) on the initial collection of singleton sets.

b) Tree obtained by Mannila and Ukkonen's algorithm after a deunion(2).

c) Tree which should be obtained after a deunion(2).

Figure 5.1

However, the upper bound of the set union problem with arbitrary deunion is still $O(\log n)$ for separable pointer algorithms. This can be established using either the linking and cutting trees of Sleator and Tarjan [1983; 1985] or the topology trees introduced by Frederickson [1985], in such a way that for each tree also a name is maintained. In the following, we will describe how to augment topology trees in order to deal with this problem. A similar argument can be applied also to linking and cutting trees.

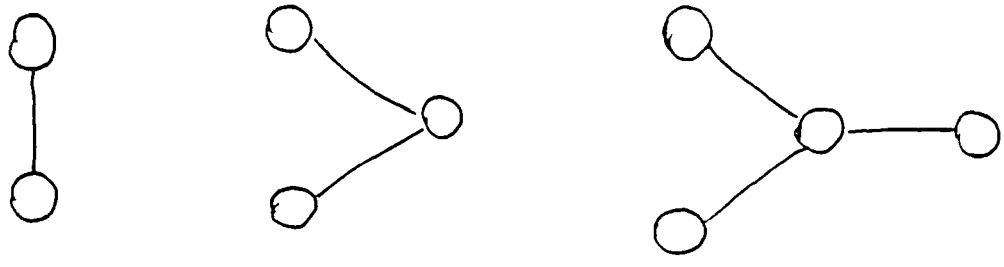
The topology tree is a data structure capable of maintaining a forest of trees under arbitrary insertions and deletions of edges. If the forest contains a total of n nodes, both updates can be carried out in $O(\log n)$ time in the worst case.

In the following, we will deal with trees in which no node has degree greater than three. This is not a significant restriction, since every tree can be transformed into a

new tree whose nodes have degree at most three, by following the well known technique described in [Harary 1969, p. 132]. The new tree will have still $O(n)$ edges and nodes.

The idea underlying Frederickson's data structure is that of defining a *topological partition* of a tree into clusters of nodes subject to the following rules. In what follows, we will refer to the *degree* of a cluster as the number of tree edges with exactly one endpoint in the cluster.

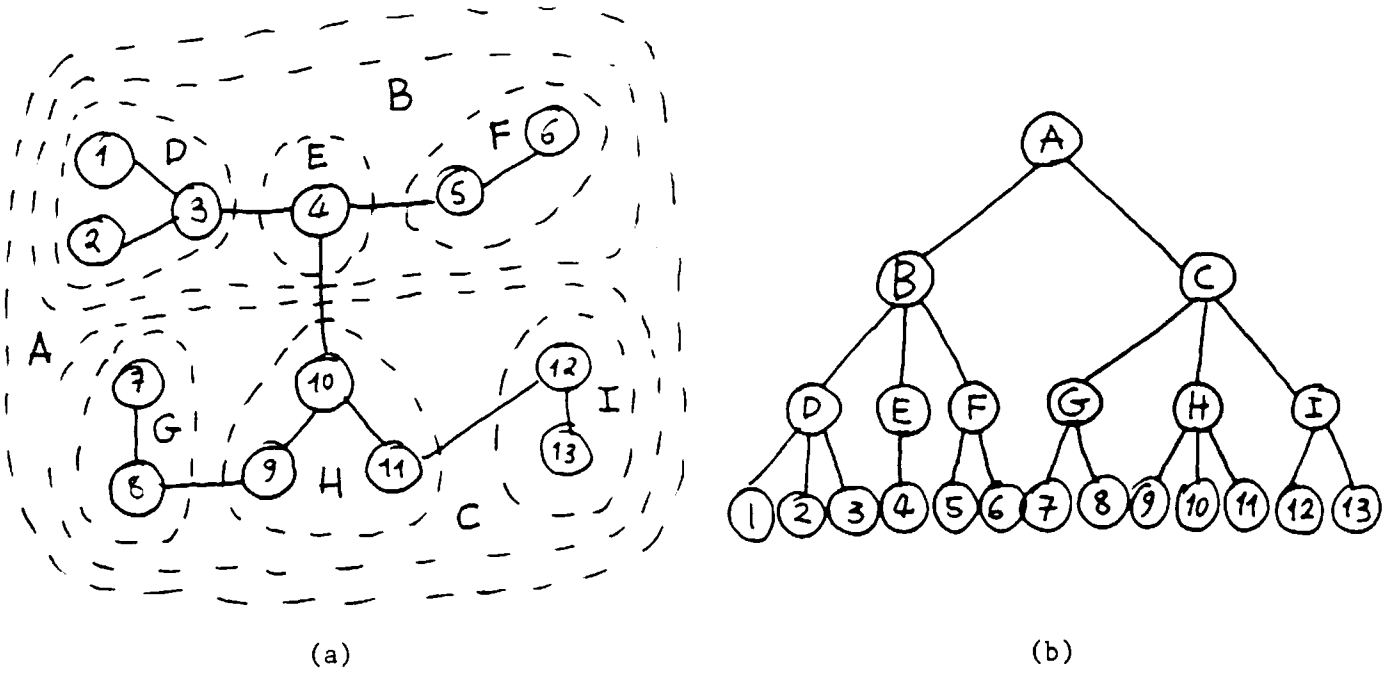
1. For each level i, the clusters at level i forms a partition of the nodes.
2. A cluster at level 0 contains only one node.
3. A cluster at level $i > 0$ is either of degree at most three and is obtained by the union of k clusters ($2 \leq k \leq 4$) of level $i-1$ connected together in one of the ways shown in figure 5.2 or a cluster of level $i-1$ and degree three.



Topologies for clusters

Figure 5.2

A *topology tree* was defined by Frederickson as a tree in which each internal vertex has at most four children, a vertex at level i represents a cluster of level i in the topological partition and a vertex has children corresponding to the clusters whose union is the cluster it represents. Figure 5.3 exhibits a topology tree of a given tree.



(a) (b)

a) Topological partition of a tree

b) The corresponding topology tree

Figure 5.3

Inserting or deleting an edge in the original tree involves operation like splitting a topology tree or merging two topology trees, which implies a costant amount of time to be spent for each vertex along a costant number of paths in the topology tree. Since its height can be at most $O(\log n)$, a topology tree can be updated in $O(\log n)$ worst-case time due to the insertion or the deletion of an edge in the original tree. All the details of the method are spelled out in [Frederickson 1985].

Using topology trees, we are now able to prove the upper bound for the set union with arbitrary deunions. A union(x, y, A) is performed by inserting the edge (x, y) and therefore by merging two topology trees. Some special care has to be taken to maintain the name of the set as explained later. A deunion(i) cuts the edge introduced by the i-th union by splitting a topology tree. A find(x) is carried out as usual.

Since in this case the trees are unrooted and a union(x, y, A) joins directly the two nodes x and y (instead of their roots as in Mannila and Ukkonen's algorithm), a deunion() returns to the correct state as if the i-th union had never occurred. The details and the complexity of the algorithm are characterized by the following theorem.

Theorem 5.2. *There exists a data structure which supports each union, find and*

deunion(i) in $O(\log n)$ time and $O(n)$ space.

Proof : Define an augmented version of topology trees in which for each cluster an additional information, called its *label*, is maintained. The label of a cluster is defined as the most recent name introduced by a union which linked two nodes in the cluster.

In order to test which is the most recent union in a cluster, we can assign a different stamp to each performed union, obtained for instance by incrementing a counter. Unfortunately, in this case the space required by the stamps is not bounded by any function of n (the number of nodes) but only by a function of m (the total number of operations).

To overcome this drawback, we use the space saving stamping technique introduced in [Gambosi et al. 1988b] which allows to recycle unused stamps. In this technique, $n-1$ different stamps are maintained in two lists, a list of *used* stamps and a list of *free* stamps. Each time we perform a union, we get the first item in the free list. This stamp is associated to the operation and inserted at the end of the used list. Each time a deunion(i) is performed, the stamp associated to the union that has to be removed is deleted from the used list and returned to the free list. As a consequence, stamps are ordered in the used list according to the time in which their associated unions were performed. Clearly, at any time at most $n-1$ different stamps are maintained in both lists, and the used list contains exactly one stamp for each union not yet undone. Using the data structure proposed by Dietz and Sleator [1987] to maintain the list of used stamps, we can insert an item, delete an item or compare the relative order of two items of this list in constant time. Hence, given two stamps we can decide in $O(1)$ time which is the one corresponding to the more recent union.

We need to augment topology trees as follows. For each vertex representing a cluster in the topology tree, both its label (i.e., the name of the most recent union performed between nodes in the cluster) and its stamp (i.e., the stamp associated to such a union) are maintained. Due to the definition of label of a cluster, stamps are *heap-ordered* in the sense that for each nonroot vertex v in the topology tree the stamp stored in v is before the stamp stored in parent(v) in the list of used stamps. Updating labels of clusters during either a union or a deunion(i) (i.e., either an insertion or a deletion of an edge in the tree) simply means to restore the heap order for the stamps after either splitting a topology tree or merging two topology trees. In the worst case, this requires that a constant number of paths in the topology tree are examined for each operation. Since Dietz and Sleator's data structure allows us to spend only $O(1)$ time for each vertex examined in the topology tree while restoring the heap order for the stamps, a total of $O(\log n)$ time per update results. Because at most $O(n)$ different stamps are needed, the space is $O(n)$.

The described algorithm runs on a pointer machine, since so do both Dietz and Sleator's algorithm [Dietz and Sleator 1987] and Willard's algorithm [Willard 1982] which is called as a subroutine by the former. •

6. The Set Union Problem with Dynamic Weighted Backtracking

In this section we will consider a further extension of the set union problem with deunions, by assigning weights to each union and by allowing to backtrack either to the

union of maximal weight or to a generic union so far performed. We will refer to this problem as the *set union problem with dynamic weighted backtracking*. The problem consists of supporting the following operations.

union(A, B, w): combine sets A, B into a new set named A and assign weight w to the operation.

find(x): return the name of the set containing element x.

increase_weight(i, Δ): increase by Δ the weight of the i-th union performed, $\Delta > 0$.

decrease_weight(i, Δ): decrease by Δ the weight of the i-th union performed, $\Delta > 0$.

backweight: undo all the unions performed after the one with maximal weight.

backtrack(i): undo all the unions performed after the i-th one.

Motivations for the study of the set union problem with dynamic weighted backtracking arise in the implementation of search heuristics in the framework of Prolog environment design [Hogger 1984; Warren and Pereira 1977]. In such a context, a sequence of union operations models a sequence of unifications between terms [Mannila and Ukkonen 1986b], while the weight associated to a union allows to evaluate the goodness of the state resulting by the unification to which the union is associated. Thus, backtracking corresponds to return to the most promising state examined so far, in the case of a failure of the current path of search. Furthermore, the repertoire of operations is enhanced by allowing to update (both to increase and to decrease) the weight associated to each union already performed. This operation adds more heuristic power to the algorithms for Prolog interpreter memory management, and therefore improves the practical performance of the previous known "blind" uniform-cost algorithms.

The possibility to backtrack to the state just before any union performed so far is maintained, as implemented by the backtrack(i) operation. This makes it possible to implement several hybrid strategies based on best-first search combined with backtracking [Ibaraki 1978; Pearl 1984] in the framework of the control of the resolution process.

6.1. Single-operation worst-case time complexity

Before analysing the time complexity of the set union problem with weighted backtracking, we need to introduce a data structure, called *Backtracking Queue* (in short BQ). To each item x of a BQ a real value $v(x)$, called its *key*, is associated. Furthermore, each time a new item x enters a BQ, an ordinal number $p(x)$ is associated to it, referred to as its *position* into the queue. The operations defined on BQ's are the following.

insert(x, v, Q) : insert item x with key v into the BQ Q. If exactly i items were previously in Q, the position of x is defined as $p(x) = i + 1$.

increase(i, Δ, Q) : if Q contains fewer than i elements, then return Q. Otherwise increase by $\Delta > 0$ (Δ real) the key associated to the item x in position i.

decrease(i, Δ, Q) : if Q contains fewer than i elements, then return Q. Otherwise decrease by $\Delta > 0$ (Δ real) the key associated to the item x in position i.

$back_1(Q)$: return to the state of the BQ Q just before the current item of largest key was inserted into Q.

$back_2(i, Q)$: if Q contains fewer than i elements, then return Q. Otherwise return to the state of the BQ just before the item currently in position i was inserted.

So, BQ's are priority queues for which some sort of getting back to the past is allowed. It is possible to support all the above operations in $O(\log n)$ worst-case time. The details can be found in [Gambosi et al. 1988c].

Theorem 6.1.1. [Gambosi et al. 1988c] *There exists an implementation of BQ's which supports each insert, increase, decrease, $back_1$ and $back_2$ in $O(\log n)$ worst-case time and requires $O(n)$ space.*

Gambosi et al. [1988c] introduced a data structure for the set union problem with weighted backtracking. The data structure supports each operation in $O(\log n)$ worst-case time and requires $O(n)$ space. They showed also that no better bound is possible for any nonseparable pointer algorithm. We will give a sketch of the data structure. The low-level details can be found in [Gambosi et al. 1988c].

During the execution of any sequence of *union*, *find*, *backweight*, *backtrack*, *increase_weight* and *decrease_weight*, all the items are partitioned into a collection of disjoint sets. As usual, we will refer to a *union* as *live* if it has not been undone by backtracking and as *dead* otherwise.

At any time the actual partition is the same that would have been resulted from simply applying the currently live unions to the initial set of singletons, in the exact order in which such unions were performed in the actual sequence of operations. This individuates a virtual sequence of live unions. It is therefore possible to uniquely denote each live *union* by the ordinal number it gets in that virtual sequence of unions. Furthermore, it can be proved that each *union*, as long as it is live, maintains the same ordinal number it was given at the time of its creation.

The ideas underlying the data structure which supports the above operations are the following. As before, we maintain every set as a tree, whose root contains the name of the set. When a *union* is performed, exactly one pointer linking two tree roots is introduced, which is associated to the *union* operation. Hence, also a pointer is said to be *live* or *dead* according to its corresponding *union*: live pointers return a connection which has not yet been cancelled by backtracking.

At the time of its execution, each *union* is associated to its ordinal number in the virtual sequence of live unions. This number is also stored as the label of the pointer corresponding to that *union*. The operations are implemented according to a lazy method, that is pointers invalidated by backtracking are not removed immediately from the structure. As a consequence, both live and dead pointers may be in the data structure at the same time and some stamping techniques [Gambosi et al. 1988b] can be used to discriminate between the two kinds of pointers. However, at any time there is at most one (live or dead) pointer leaving any given node.

In order to be able to perform union by rank, the ranks of the nodes are maintained during the evolution of the structure. Following a technique described also in [Gambosi ..

al. 1988a; 1988b], to each node a balanced tree [Adelson-Velskii and Landis 1962; Aho et al. 1974; Nievergelt and Reingold 1973], referred to as $rank(x)$, is associated. Each item of $rank(x)$ (in the sequel referred to as a *rank* of x) corresponds to a pointer (v, x) entering x. Its key in the balanced tree $rank(x)$ is the ordinal number of the *union* which introduced (v, x), while it also stores the rank of x immediately after the introduction of (v, x). Notice that, as a consequence of this definition, there exists a one to one correspondence between pointers and rank items: in order to allow fast references between ranks and pointers, a bidirectional link is introduced between a rank item and its corresponding pointer. The notion of liveness can now be extended to a rank of any node x: hence, a rank of x is said to be *live* if and only if the corresponding pointer is live, otherwise it is said to be *dead*.

To restore the correct rank of any node x in presence of live and dead ranks, the largest live rank (i.e. the value corresponding to the live union with largest ordinal number) must be individuated in $rank(x)$. This information clearly turns out to be useful while implementing the union by rank rule.

Similarly, balanced trees may be also associated to each node x in order to store the names assigned to sets represented by trees rooted at x during the evolution of the structure: such a balanced tree will be referred to as $name(x)$. As in the case of the *rank* balanced trees, each item in $name(x)$ corresponds to a pointer (v, x) entering x, has the ordinal number of that pointer as a key and stores the name associated to x after the introduction of (v, x). Once again, there are bidirectional links between names (items in $name(x)$) and the corresponding pointers and the notion of liveness can be extended to the names. When the current name of a root r must be individuated, a search for the largest live item in $name(r)$ must be accomplished (i.e., a search for the name corresponding to the live union with largest ordinal number).

The weights of the live unions are maintained in a Backtracking Queue Q. Each item in Q represents a live union together with its current weight, while the position into Q is its ordinal number in the virtual sequence of live unions.

Furthermore, the following invariant (referred to as *pointer consistency*) is maintained: "*If there are k live unions at a given time, then a pointer in the data structure is dead if and only if it is labeled with an ordinal number larger than k*".

Notice that whenever pointer consistency holds, the liveness of a pointer can be verified in constant time by simply comparing its label with the value i_{max} available in the Backtracking Queue.

The different operations can be implemented as follows.

union(A, B, w) - Let us denote by x and y the tree roots of the sets A and B. Remove all pointers leaving x and y together with their associated ranks and names. Locate and remove the pointer (if any) previously introduced with ordinal number $i_{max}+1$. Since this pointer has an ordinal number exceeding i_{max}, due to pointer consistency it is certainly dead and hence can be deleted together with its rank and name. Restore the actual heights h_x and h_y of x and y by means of a search for the largest live items in $rank(x)$ and $rank(y)$. Link by rank x and y, associating to the new pointer the ordinal number $i_{max}+1$. Insert the new rank and the name A of the resulting set either in $rank(x)$ and $name(x)$ or in $rank(y)$ and $name(y)$, according to the pointer introduced. Perform an *insert*$((x, y), w, Q)$ into the Backtracking Queue. Finally, set i_{max} to $i_{max}+1$.

find(x) - starting from node x, follow the live pointer leaving the node. The liveness of a pointer can be tested in costant time as described above. Repeat until a node $r(x)$ with no live outgoing pointer is entered. This node is the tree root of the set containing x. A search for the largest live item in $name(r(x))$ returns the actual name of the set containing x.

increase_weight(i, Δ) - perform an *increase*(i, Δ, Q) on the Backtracking Queue.

decrease_weight(i, Δ) - perform a *decrease*(i, Δ, Q) on the Backtracking Queue.

backweight - perform a $back_1(Q)$ on the Backtracking Queue.

backtrack(i) - perform a $back_2(i, Q)$ on the Backtracking Queue.

The worst-case complexity per operation of the data structure can be characterized by the following theorem. For its proof, the reader is referred to [Gambosi et al. 1988c].

Theorem 6.1.2. [Gambosi et al. 1988c] *It is possible to perform each union, find, increase_weight, decrease_weight, backweight and backtrack in* $O(\log n)$ *time. The space required is* $O(n)$.

6.2. A lower bound for set union with weighted backtracking

The bound in Theorem 6.1.2 is the best possible for any nonseparable pointer algorithm. Recall rules (i)-(v) in the definition of nonseparable algorithms given in section 1. The algorithm of section 6.1 clearly obeys these rules. For such a class of algorithms, the following lower bound holds.

Theorem 6.2.1. *Let* A *be any nonseparable pointer algorithm. Then there exists a sequence of weighted unions, finds, increase_weight, decrease_weight, backweight, backtrack, such that the worst-case per operation time complexity of* A *is* $\Omega(\log n)$.

Proof : By reduction to the partially persistent set union problem (see Section 8). In other terms, we will show that we can implement unweighted unions (referred to as *union'*), finds and finds in the past (both referred to as *find'*) as defined for instance in [Mannila and Ukkonen 1988] by using weighted *unions*, *finds* and *backtrack* as defined for our problem. Since Mannila and Ukkonen proved an $\Omega(\log n)$ bound for the worst-case single operation time complexity of the partially persistent set union problem [Mannila and Ukkonen 1988] for the class of algorithms defined above, the same lower bound will also apply to the worst-case single operation time complexity of the set union problem with weighted backtracking.

Unweighted unions (*unions'*) can be easily implemented by means of weighted *unions* where the weight w is always constant, while finds can be implemented in the same fashion in the two problems. In order to perform a *find'*(x, k) as defined in [Mannila and Ukkonen 1988], that is to return the name of the set which contained item x just after the k-th union was performed, we first perform a *backtrack*(k), that is we undo all the unions performed after the k-th one, and keep track of all the work done while switching from

the old representation to the new one. A $find(x)$ is now able to correctly return the name of the set containing x after the k-th *union*. Using the extra information computed while backtracking we can finally rebuild the old representation as if no backtracking had ever taken place. The time is still bounded by the worst-case time complexity of a *backtrack*. This completes the reduction to the partially persistent set union problem. As a consequence of Theorem 2 in [Mannila and Ukkonen 1988], we can conclude that the lower bound for the worst-case per operation time complexity of the set union problem with weighted backtracking is $\Omega(\log n)$. •

6.3. Amortized analysis

In this section, we will refine the algorithm presented in section 6.1 in order to get better amortized bounds. Such solution relies on the use of an extension of *Fibonacci Heaps* [Fredman and Tarjan 1987], which we call *Backtracking Fibonacci Heaps* (*BF-heaps* for short). A BF-heap maintains a collection of items of a set $S = \{s_1, s_2, \ldots, s_n\}$, where each item s has an associated value $v(s)$, under a sequence of the following operations.

$insert(s, v)$: insert item s with $v(s) = v$ in the heap: assign to s an integer $p(s)$, as defined in section 6.1 and referred to as its position in the queue.

$findmax$: return (a pointer to) the element with maximal value in the heap.

$decrease(s, \Delta)$: decrease by Δ the value $v(s)$.

$increase(s, \Delta)$: increase by Δ the value $v(s)$.

$back_1$: delete all elements inserted in the heap after the insertion of the maximum.

$back_2(i)$: delete all elements inserted in the heap after the insertion of the element in position i.

$meld(h_1, h_2)$: meld heaps h_1 and h_2.

A BF-heap is essentially an F-heap where each item s stores its associated position $p(s)$. Furthermore, an array ACCESS$[1, \ldots, n]$ is introduced which, for each entry $i \leq i_{max}$, stores a pointer to the item s with $p(s) = i$. In general, the structure may store a set of *dead* elements, i.e. elements deleted by $back_1$ and $back_2$ operations but still not removed from the heap. A dead element s can be identified since $p(s) > i_{max}$. While i_{max} represents the number of live elements in the structure, let I_{MAX} be the *total* number of elements stored.

To obtain a better amortized bound we substitute BQ's with BF-heaps in the algorithm sketched in the previous section. It is possible to state the following lemma.

Lemma 6.3.1. *If we begin with no BF-heaps and perform an arbitrary sequence of k operations, with $n \leq k$ insert and $m \leq k$ decrease, the total time complexity of such a sequence is $O(k + (m + n) \log n)$. Hence, each findmax, increase, $back_1$, $back_2$ and meld can be supported in $O(1)$ amortized time, while the amortized time complexity of both decrease and insert is $O(\log n)$*

Proof : $increase(s, \Delta)$ and $meld(h_1, h_2)$ are performed as defined on F-heaps. Let us now sketch the implementation of the remaining operations.

- An $insert(s, v)$ is performed by inserting in the BF-heap the new element with $p(s) = i_{max} + 1$. If entry $i_{max} + 1$ of ACCESS refers to a (dead) element s', such element is deleted from the BF-heap. As for F-heaps, the actual insertion of an element is performed by means of a *meld* operation. Variable i_{max} is increased by 1 and variable I_{MAX} stores the value $max(i_{max}, I_{MAX})$.
- A $decrease(s, \Delta)$ is carried out by first deleting s and by saving $v(s)$ and $p(s)$. Then an $insert(s, v(s) - \Delta)$ in position $p(s)$ is performed.
- A *findmax* consists of deleting from the BF-heap all elements referred to by locations ACCESS[j] with $I_{MAX} \geq j > i_{max}$ (i.e. all elements s with $p(s) > i_{max}$). A number of deletions is performed which is equal to the number of dead nodes in the BF-heap, thus resulting in an $O(\log n)$ cost amortized on the set of *insert* operations which introduced the dead nodes. Then the new maximum has to be found. This causes a search among the $O(\log n)$ root nodes of the underlying F-heap. Last, I_{MAX} is set equal to i_{max}.
- A $back_1$ updates i_{max} to the value of the maximal node in the BF-heap.
- A $back_2(i)$ sets i_{max} to i.

Following the technique of recovering dead nodes and pointers during *findmax* and *insert* operations as described in [Gambosi et al. 1988a], it is possible to verify that such a structure requires $O(n)$ space. Moreover, introducing the new operations does not affect the amortized complexity of operations already defined on F-heaps, since all the new operations either are defined in terms of operations on F-heaps or have an $O(1)$ time complexity. •

The following theorem is easily derived from Lemma 6.3.1 and from Theorem 6.1.2.

Theorem 6.3.1. [Gambosi et al. 1988c] *It is possible to perform each backweight, backtrack and increase_weight in $O(1)$ amortized time and find, decrease_weight and union in $O(\log n)$ amortized time. The space required is $O(n)$.*

A slightly better amortized bound for *find* as the ratio of finds to unions and backtracks tends to increase can be obtained as suggested by Tarjan [1988], by using the data structure described in [Westbrook and Tarjan 1987] in combination with BF-heaps. This results in a $O(\log n / max\{1, \log(\gamma \log n)\})$ amortized bound for finds, where γ is the ratio of the number of finds to the number of unions and backtracks in the sequence.

7. The Set Union Problem with Unrestricted Backtracking

A further generalization of the set union problem with deunions was considered in [Apostolico et al. 1988]. This generalization was called the *set union problem with unrestricted backtracking*, since the limitation that at most one union could be undone per operation was removed.

As before, we denote a union not yet undone by *live*, and by *dead* otherwise. In the set union problem with unrestricted backtracking, deunions are replaced by the following more general operation.

backtrack(i) : Undo the last i live unions performed, for any integer $i \geq 0$.

Note that this problem lies in between set union with deunions and set union with weighted backtracking. In fact, as previously noted, it is more general than the set union problem with deunions, since a deunion can be implemented as *backtrack(1)*. On the other hand, it is a particular case of the set union with weighted backtraking, when only unweighted union, find and backtrack operations are considered. As a consequence, its time complexity should be between $O(\log n / \log\log n)$ and $O(\log n)$. In this section we will show that in fact a $\Theta(\log n)$ bound holds for nonseparable pointer algorithms, thus proving that set union with unrestricted backtracking is as difficult as set union with dynamic weighted backtracking. Surprisingly, the time complexity reduces to $\Theta(\log n / \log\log n)$ for separable pointer algorithms when unites instead of unions are performed (i.e., when the name of the new set can be arbitrarily chosen).

7.1. Amortized complexity

It is not surprising that there is a strict relationship between backtracks and deunions. We already noted that a *backtrack(1)* is simply a deunion operation. Furthermore, a *backtrack(i)* can be implemented by performing exactly i deunions. Hence, a sequence of m_1 unions, m_2 finds and m_3 backtracks can be carried out by simply performing at most m_1 deunions instead of the backtracks. Applying Westbrook and Tarjan's algorithms to the sequence of union, find and deunion operations, a total of $O((m_1 + m_2)\log n / \log\log n)$ worst-case running time will result. As a consequence, the set union problem with unrestricted backtracking can be solved in $O(\log n / \log\log n)$ amortized time per operation. Since backtracks contain deunions as a particular case, this bound is tight for the class of separable pointer algorithms.

Westbrook and Tarjan's algorithms, despite their amortized efficiency, are not very efficient when the worst-case per operation time complexity of the set union problem with unrestricted backracking is taken into account.

Using sk-trees, a backtrack(i) can require $\Omega(i \log n / \log\log n)$ worst-case time. Also note that the worst-case time complexity of backtrack(i) is at least $\Omega(i)$ as long as one insists on deleting pointers as soon as they are invalidated by backtracking (as the *eager* methods described in section 4.1 do), since in this case at least one pointer must be removed for each erased union. This is clearly undesirable, since i can be as large as $n-1$. In order to overcome this difficulty, dead pointers have to be destroyed in a *lazy* fashion. Worst-case per operation efficient algorithms will be shown in the next subsection.

7.2. Single operation worst-case time complexity

The set union problem with unrestricted backtracking can be considered in two versions, depending on whether we consider *union* or *unite* operations. Surprisingly, these

two versions have completely different single-operation worst-case time complexity. In fact, in case of unions (i.e., when the name of the new set is not arbitrarily chosen), a $\Theta(\log n)$ bound holds for nonseparable pointer algorithms. But if we allow the name of the new set to be arbitrarily chosen (i.e., if we perform unite instead of union), then the complexity of the problem reduces to $\Theta(\log n / \log\log n)$ for separable pointer based algorithms.

The following theorem holds for the set union with unrestricted backtracking, when union operations are taken into account.

Theorem 7.2.1. *It is possible to perform each union, find and backtrack(i) in $O(\log n)$ time in the worst case. This bound is tight for nonseparable pointer algorithms.*

Proof : The upper bound is a straightforward consequence of Theorem 6.1.2, since unrestricted backtracking is a particular case of weighted backtracking. Furthermore, the proof of the lower bound given in Theorem 6.2.1 for nonseparable pointer algorithms holds also for the new problem, since it makes use only of union, find and backtrack. •

In the following, we will restrict our attention to the version where *unite* operations are performed (instead of *unions*) and show that the upper bound reduces to $O(\log n / \log\log n)$. No better bound is possible for separable pointer algorithms. The upper bound is based on a data structure which stores a collection of disjoint sets in such a way that the identity of each member of the collection is preserved. We denote this data structure by *reminescent separator k-tree* or in short *rsk-tree*. As usual, we assume $k = \lceil \log n / \log\log n \rceil$.

We give a high-level description of rsk-trees and their properties, together with the implementation of the unite, find and backtrack operations. All the details of the method are contained in [Apostolico et al. 1988].

Rsk-trees are a lazy version of sk-trees. They do not destroy immediately the pointers made void by backtracking. Rather, these pointers are maintained in the structure and removed in a lazy fashion. Clearly, the implementation of unions and finds described in the previous section for the sk-trees must be slightly changed in order to take in account the dead pointers still present in the structure.

An rsk-tree is an sk-tree whose pointers are labeled as *live*, *dead*, or *cheating*, and whose separator pointers are in addition labeled as either *active* or *inactive*. Live pointers represent a connection which has not been cancelled by backtracks, while dead pointers represent no connection at all: although still in the structure, dead pointers only wait to be destroyed. Between live and dead pointers, lie cheating pointers. They are derived from dead type 3 unites. As a consequence, they represent a faulty connection and henceforth do not have to be destroyed but only to be replaced by the right pointers. As for sk-trees, separators are associated to type 3 unites. They are *active* if their associated unite is live *inactive* otherwise.

In general, a pointer may fall in any of these classes, except that a pointer e $(v, parent(v))$ cannot be cheating unless either e is an inactive separator or there is an inactive separator to the left of e within distance k (i.e. among the k siblings to the left of v there is a node u, such that $(u, parent(v))$ is an inactive separator). This restriction

corresponds to the fact that both cheating pointers and inactive separators are due to dead type 3 unites, which move at most k pointers.

We describe how finds and unites must be modified in order to take into account the dead and cheating pointers present in the structure. Let $S_1, S_2, \ldots, S_p$ be the disjoint sets stored in the rsk-tree T. We now show how to execute a find by computing a map from the set of leaves of T to the set of names $S_1, S_2, \ldots, S_p$. Let x be a leaf of T and also a member of the set S_q, $1 \leq q \leq p$. Let Y be the name of S_q. We ascend from x towards the root of T following live pointers until a node is met without outgoing live pointers. We call such a node the *apex of* x and we shall refer to it as $apex(x)$. Only three different cases can occur which correspond to three types of apices:

Live apex - there is no (live, dead or cheating) pointer leaving apex(x), i.e., apex(x) is the root r of T. The label of r is the name Y of S_q.

Dead apex - apex(x) is such that its outgoing pointer is dead. The label of apex(x) is the name Y of S_q.

Cheating apex - apex(x) is such that its outgoing pointer e is cheating. If e is an inactive separator, then the name of S_q can be found in the label of e. Otherwise, there is at least one inactive separator within distance k to the left of e. In this case, the name of S_q can be found in the label of the nearest separator to the left of e.

We now describe how unite operations can be performed. Let A and B be two different classes of the partition of S, such that $A \neq B$. In the collection of rsk-trees that represents this partition, let T_A and T_B be the rsk-trees storing respectively A and B. We recall that, since we allow that any two disjoint sets be stored in the same data structure, T_A and T_B may coincide even if $A \neq B$.

A unite(A, B) must have as the unique effect that the live paths from any element of $A \cup B$ must lead now to the same label (either A or B), corresponding to the fact that such elements are now in the same set (named either A or B). Any live path in either T_A or T_B starting from leaves not in $A \cup B$ must continue to lead to the same label as it did prior to performing unite(A, B).

Roughly speaking, the first step to be performed consists of detaching from T_A and T_B the subtrees which store respectively A and B. Suppose now that we want to detach from T_A the subtree which stores the elements of the set A and let $apex(A)$ denote the apex of all the nodes in the set A. This detachment depends on the type of apex encountered If apex(A) is live, then no detachment is involved at all. If apex(A) is dead, then the detachment can be simply accomplished by removing the pointer leaving apex(A). In the case where apex(A) is cheating, we know that either the pointer leaving apex(A) or some inactive separator to its left within distance k, stores A as its label. In both cases, we reach this inactive separator which stores A and, starting with it, cut all pointers to its right up to and excluding the first inactive separator, if any such separator exists. Furthermore the nodes now detached are made children of a new root labeled with A.

Having detached the two subtrees as described above, we now combine them into a single tree, using the union algorithm described fro sk-trees in section 4.2. The resulting tree is still an rsk-tree in that sense that it may store also sets in the collection other than $A \cup B$, since in the subtrees containing A and B only the roots and possibly the children of the roots were modified.

It remains to show how a backtrack(i) may be performed on rsk-trees. The effect of such operation is to undo the last i unites performed not yet undone, which results in simply changing some pointers from live to either dead or cheating and from cheating to dead. Furthermore, some active separators may become inactive. This requires obviously some bookkeeping in order to associate pointers and separators with the corresponding unites and to easily check the state of pointers and/or separators. It is important to note that while performing a backtrack(i), no pointer is destroyed from the structure. What happens is simply that, as a consequence of this operation, some pointers and/or separators change their state. All the details of the method are contained in [Apostolico et al. 1988]. The time and space complexity of the algorithm are characterized by the following theorem.

Theorem 7.2.2. [Apostolico et al. 1988] *Rsk-trees support each unite and find operation in* $O(\log n/\log\log n)$ *time, each backtrack in* $O(1)$ *time and require* $O(n)$ *space.*

Proof : See Theorem 4.1 in [Apostolico et al. 1988]. •

As a consequence, this algorithm generalizes the bounds obtainable with Westbrook and Tarjan's algorithms [1987], since it achieves the same optimal performance, but in the worst (not only amortized) case.

No better bound is possible for any separable pointer algorithm. Recall rules (i)-(v) as defined in section 1 for separable pointer algorithms. The lazy algorithm described above clearly obeys rules (i), (ii), (iv) and (v). It satisfies also rule (iii), if we regard pointers as disappearing from the model as soon as they become either cheating or dead, as observed by Westbrook and Tarjan [1987]. In fact, the presence of cheating and dead pointers has no effect on the performance of the algorithm in the model, since they give connections which are never followed. For the class of separable pointer algorithms, the following lower bound holds.

Theorem 7.2.3. *For any* n, *any separable pointer algorithm for the set union with unrestricted backtracking has single-operation time complexity at least* $\Omega(\log n/\log\log n)$ *in the worst case.*

Proof : It is a trivial extension of Theorem 2.2.2, which states that it suffices to consider only unite and find operations. •

It is somewhat surprising that the two versions of the set union problem with unrestricted backtracking have such a different time complexity, and that the version with unites can be solved quite more efficiently than the version with unions.

We recall here that after a unite(A, B), the name of the newly created set is either A or B. This is not a significant restriction in the applications, where one is mostly concerned on testing whether two elements belong to the same equivalence class, no matter what the name of the class can be.

The lower bound of $\Omega(\log n)$ is a consequence of Theorem 2 in [Mannila and Ukkonen 1988], which depends heavily on the fact that each union cannot arbitrarily choose a new name. The crucial idea behind the proof of Theorem 2 in [Mannila and Ukkonen 198

is that at some point we may have to to discriminate between $\Theta(n)$ different names of a set containing any given element in order to output a correct answer. But, if a new name can be arbitrarily chosen after performing a union, the inherent complexity of the set union problem with unrestricted backtracking reduces to $\Omega(\log n/\log\log n)$. Hence, the constraint on the choice of a new name is responsible for the gap between $\Omega(\log n/\log\log n)$ and $\Omega(\log n)$.

8. Partially Persistent Data Structures for Set Union

In this section we describe partially persistent [Driscoll et al. 1986; Overmars 1983] data structures for the set union problem. In such a case, union is defined as usual and creates a new version of the data structure. As a consequence, if l unions were performed ($0 \leq l \leq n-1$) there are exactly $l+1$ versions of the data structure numbered from 0 to l. A find operation is now extended as follows:

find_past(x,k) - return the name of the set which contained the element x in the k-th version of the data structure. This operation is defined only for $0 \leq k \leq l$.

A $find\_past(x,l)$ is performed on the last version of the data structure and is therefore equivalent to the classical definition of find previously given. As in the case of the set union problem with unrestricted backtracking, we have two versions of this problem depending on whether union or unite operations are performed. The time complexity of the two versions is quite different, as shown in the following two theorems.

Theorem 8.1. *There exists a data structure which supports each union and find_past in $O(\log n)$ worst-case time with an $O(n)$ space usage. No better bound is possible for nonseparable pointer algorithms.*

Proof: Consider the data structure introduced for dealing with the set union problem with weighted backtracking. Unions can be carried out exactly in the same way, with the only difference that now the weights can be neglegted. In addition l, the total number of unions performed, is maintained. With this information, a find_past(x,k) may be carried out in three steps.

1. Set i_{max} to $i_{max} - k$.
2. Perform a find(x) in the resulting data structure.
3. Restore the correct value of i_{max} by adding k to it.

Note that this is somewhat similar to performing a virtual backtracking whose effect is undone at the end of the find_past operation.

Since a find_past is implemented by means of a find plus some operations which require constant time, the time and space bounds now easily follow from Theorem 7.4. Furthermore, the time bound is tight for the class of nonseparable pointer algorithms as a consequence of Theorem 2 in [Mannila and Ukkonen 1988]. •

The amortized time of a union can be further reduced to $O(1)$ by using the data structures introduced in [Brown and Tarjan 1980; Huddleston and Mehlhorn 1982]. Different data structures can be also used to establish the previous upper bound, as shown for instance in [Gaibisso et al. 1987; Mannila and Ukkonen 1988]. Furthermore, if we perform unites instead of unions, a better algorithm can be found.

Theorem 8.2. *There exists a data structure which supports each unite and find_past in $O(\log n/\log\log n)$ time with an $O(n)$ space usage. No better bound is possible for separable pointer algorithms.*

Proof : For the upper bound, consider rsk-trees and apply the same argument as in Theorem 8.1. The lower bound is a straightforward consequence of Theorem 2.2.2. •

As in the case of the set union problem with unrestricted backtracking, the constraint on the choice of a new name is responsible for the gap between $\Omega(\log n/\log\log n)$ and $\Omega(\log n)$.

9. Conclusions and Open Problems

In this paper we have described the most efficient known algorithms for solving the set union problem and some of its variants. Most of the algorithms we have described are optimal with respect to a certain model of computation (e.g., pointer machines with or without the separability assumption and random access machines). There are still several intriguing open problems in all the models of computation we have considered. In particular, it is still open whether both the amortized and the single-operation worst-case complexity of the following problems can be improved.

1. The set union problem.
2. The set union problem with deunions.
3. The set union problem with arbitrary deunions.
4. The set union problem with unrestricted backtracking.

If possible, these improvements will require either a nonseparable pointer algorithm or the extra power of a random access machine.

Furthermore, there are also no lower bounds for some of the set union problems on intervals. In the pointer machine model with the separability assumption, there is no lower bound for the amortized complexity of union-find and split-find as well for the worst-case complexity of union-find. In the realm of nonseparable pointer algorithms, it remains still open whether both the $O(\log n/\log\log n)$ worst-case bound [Blum 1986] for union-find and the $O(\alpha(m,n))$ amortized bound [Gabow 1985] for split-find can be improved. The two problems require $\Theta(1)$ amortized time on a random access machine as shown by Gabow and Tarjan [1985].

Finally, we showed in section 8 how to access efficiently past versions of set union data structures by studying partial persistence in the set union problem. It seems to be worth of further investigation to study whether these techniques can be extended in order to

both access and modify the past versions of the set union data structures, thus obtaining fully persistent data structures [Driscoll et al. 1986]. This problem is significant for several applications as well as being of theoretical interest [Driscoll et al. 1986; Overmars 1983].

Acknowledgements We would like to thank Alberto Apostolico, Hal Gabow, Lane Hemachandra, Bob Tarjan and Henryk Wozniakowski for many valuable comments and suggestions. We are also grateful to Giorgio Gambosi and Maurizio Talamo for pointing out [Dietz and Sleator 1987].

References

ACKERMAN, W. 1928. Zum Hilbertshen Aufbau der reelen Zahlen. *Math. Ann.* 99, pp. 118–133.

ADELSON-VELSKII, G. M., AND LANDIS, Y. M. 1962. An algorithm for the organization of the information. *Soviet. Math. Dokl.* 3, pp. 1259–1262.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1973. On computing least common ancestors in trees. *Proc. 5th Annual ACM Symposium on Theory of Computing*, pp. 253–265.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, Mass.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1983. *Data structures and algorithms.* Addison-Wesley, Reading, Mass.

AHUJA, R. K., MEHLHORN, K., ORLIN, J. B., AND TARJAN, R. E. 1988. Faster algorithms for the shortest path problem. Manuscript.

APOSTOLICO, A., GAMBOSI, G., ITALIANO, G. F., AND TALAMO, M. 1988. An $O(\log n / \log\log n)$ algorithm for the set union problem with unrestricted backtracking. Manuscript.

APOSTOLICO, A., AND GUERRA, C. 1987. The longest common subsequence problem revisited. *Algorithmica* 2, pp. 315–336.

ARDEN, B. W., GALLER, B. A., AND GRAHAM, R. M. 1961. An algorithm for equivalence declarations. *Comm. ACM* 4, pp. 310–314.

BANACHOWSKI, L. 1980. A complement to Tarjan's result about the lower bound on the complexity of the set union problem. *Inform. Processing Lett.* 11, pp. 59–65.

BEN-AMRAM, A. M., AND GALIL, Z. 1988. On pointers versus addresses. *Proc. 29th Annual Symposium on Foundations of Computer Science*, pp. 532–538.

BLUM, N. 1986. On the single operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput.* 15, pp. 1021–1024.

BOLLOBÁS, B., AND SIMON, I. 1985. On the expected behaviour of disjoint set union problems. *Proc. 17th Annual ACM Symposium on Theory of Computing*, pp. 224–231.

BROWN, M. R., AND TARJAN, R. E. 1980. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput.* 9, pp. 594–614.

DIETZ, P. F., AND SLEATOR, D. D. 1987. Two algorithms for maintaining order in a list. *Proc. 19th Annual ACM Symposium on Theory of Computing*, pp. 365–372.

DOYLE, J., AND RIVEST, R. 1976. Linear expected time of a simple union-find algorithm. *Inform. Processing Lett.* 5, pp. 146–148.

DRISCOLL, J. R., SARNAK, N., SLEATOR, D. D., AND TARJAN, R. E. 1986. Making data structures persistent. *Proc. 18th Annual ACM Symposium on Theory of Computing*, pp. 109–121.

FISCHER, M. J. 1972. Efficiency of equivalence algorithms. In *Complexity of computer computations*, R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, pp. 153–168.

FREDERICKSON, G. N. 1985. Data structures for on-line updating of minimum spanning trees with applications. *SIAM J. Comput.* 14, pp. 781–798.

FREDMAN, M. L., AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.* 34, pp. 596–615.

GABOW, H. N. 1985. A scaling algorithm for weighted matching on general graphs. *Proc. 26th Annual Symposium on Foundations of Computer Science*, pp. 90–100.

GABOW, H. N., AND TARJAN, R. E. 1985. A linear time algorithm for a special case of disjoint set union. *J. Comput. Sys. Sci.* 30, pp. 209–221.

GAIBISSO, C., GAMBOSI, G., AND TALAMO, M. 1987. A partially persistent data structure for the set union problem. Manuscript.

GALLER, B. A., AND FISCHER, M. J. 1964. An improved equivalence algorithm. *Comm. ACM* 7, pp. 301–303.

GAMBOSI, G., ITALIANO, G. F., AND TALAMO, M. 1988a. Getting back to the past in the union find problem. *Proc. 5th Symposium on Theoretical Aspects of Computer Science* (STACS 1988), *Lecture Notes in Computer Science* 294, Springer-Verlag, Berlin, pp. 8-17.

GAMBOSI, G., ITALIANO, G. F., AND TALAMO, M. 1988b. Worst-case analysis of the set union problem with extended backtracking. *Theoret. Comput. Sci.*, to appear.

GAMBOSI, G., ITALIANO, G. F., AND TALAMO, M. 1988c. The set union problem with dynamic weighted backtracking. Manuscript.

HARARY, F. 1969. *Graph theory*. Addison-Wesley, Reading, Mass.

HART, S., AND SHARIR, M. 1986. Non-linearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica* 6, pp. 151–177.

HOGGER, C. J. 1984. *Introduction to logic programming*. Academic Press.

HOPCROFT, J. E., AND ULLMAN, J. D. 1973. Set merging algorithms. *SIAM J. Comput.* 2, pp. 294–303.

HUDDLESTON, S., AND MEHLHORN, K. 1982. A new data structure for representing sorted lists. *Acta Informatica* 17, pp. 157–184.

IBARAKI, T. 1978. M-depth search in branch and bound algorithms. *Int. J. Comput Inform. Sci.* 7, pp. 313–373.

IMAI, T., AND ASANO, T. 1984. Dynamic segment intersection with applications. *Proc 25th Annual Symposium on Foundations of Computer Science*, pp. 393–402.

KARLSSON, R. G. 1984. Algorithms in a restricted universe. Tech. Report CS-84-50. Department of Computer Science, University of Waterloo.

KERSCHENBAUM, A., AND VAN SLYKE, R. 1972. Computing minimum spanning trees efficiently. *Proc. 25th Annual Conf. of the ACM*, pp. 518–527.

KNUTH, D. E. 1968. *The Art of Computer Programming.* Vol. 1: *Fundamental Algorithms.* Addison-Wesley, Reading, Mass.

KNUTH, D. E. 1973. *The Art of Computer Programming.* Vol. 3: *Sorting and Searching.* Addison-Wesley, Reading, Mass.

KNUTH, D. E., AND SCHÖNAGE, A. 1978. The expected linearity of a simple equivalence algorithm. *Theoret. Comput. Sci.* 6, pp. 281–315.

KOLMOGORV, A. N. 1953. On the notion of algorithm. *Uspehi Mat. Nauk.* 8, pp. 175–176.

LOEBL, M., AND NEŠETŘIL, J. 1988. Linearity and unprovability of set union problem strategies. *Proc. 20th Annual ACM Symposium on Theory of Computing*, pp. 360–366.

MANNILA, H., AND UKKONEN, E. 1986a. The set union problem with backtracking. *Proc. 13th International Colloquium on Automata, Languages and Programming* (ICALP 86). *Lecture Notes in Computer Science* 226, Springer-Verlag, Berlin, pp. 236–243.

MANNILA, H., AND UKKONEN, E. 1986b. On the complexity of unification sequences. *Proc. 3rd International Conference on Logic Programming. Lecture Notes in Computer Science* 225, Springer-Verlag, Berlin, pp. 122–133.

MANNILA, H., AND UKKONEN, E. 1986c. Timestamped term representation for implementing Prolog. *Proc. 3rd IEEE Conference on Logic Programming*, pp. 159–167.

MANNILA, H., AND UKKONEN, E. 1987. Space-time optimal algorithms for the set union problem with backtracking. Technical Report C-1987-80, Department of Computer Science, University of Helsinki, Helsinki, Finland.

MANNILA, H., AND UKKONEN, E. 1988. Time parameter and arbitrary deunions in the set union problem. Technical Report A-1988-4, Department of Computer Science, University of Helsinki, Helsinki, Finland.

MEHLHORN, K. 1984a. *Data structures and algorithms.* Vol. 1: *Sorting and searching.* Springer-Verlag, Berlin.

MEHLHORN, K. 1984b. *Data structures and algorithms.* Vol. 2: *Graph algorithms and NP-completeness.* Springer-Verlag, Berlin.

MEHLHORN, K. 1984c. *Data structures and algorithms.* Vol. 3: *Multidimensional searching and computational geometry.* Springer-Verlag, Berlin.

MEHLHORN, K., AND NÄHER, S. 1986. Dynamic Fractional Cascading. Technical Report TR 06/1986, FB10, Universität des Saarlandes, Saarbrücken, West Germany.

MEHLHORN, K., NÄHER, S., AND ALT, H. 1987. A lower bound for the complexity of the union-split-find problem. *Proc. 14th International Colloquium on Automata, Languages*

and Programming (ICALP 87), *Lecture Notes in Computer Science* **267**, Springer-Verlag, Berlin, pp. 479–488.

NIEVERGELT, J., AND REINGOLD, E. M. 1973. Binary search trees of bounded balance. *SIAM J. Comput.* **2**, pp. 33–43.

OVERMARS, M. H. 1983. The design of dynamic data structures. *Lecture Notes in Computer Science* **156**, Springer-Verlag, Berlin.

PEARL, J. 1984. *Heuristics*. Addison-Wesley, Reading, Mass.

SCHÖNAGE, A. 1980. Storage modification machines. *SIAM J. Comput.* **9**, pp. 490–508.

SLEATOR, D. D., AND TARJAN, R. E. 1983. A data structure for dynamic trees. *J. Comput. Sys. Sci.* **26**, pp. 362–391.

SLEATOR, D. D., AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.* **32**, pp. 652–686.

STEARNS, R. E., AND LEWIS, P. M. 1969. Property grammars and table machines. *Information and Control* **14**, pp. 524–549.

STEARNS, R. E., AND ROSENKRANTZ, P. M. 1969. Table machine simulation. *Conf. Rec. IEEE 10th Annual Symp. on Switching and Automata Theory*, pp. 118–128.

TARJAN, R. E. 1973. Testing flow graph reducibility. *Proc. 5th Annual ACM Symp. on Theory of Computing*, pp. 96–107.

TARJAN, R. E. 1974. Finding dominators in directed graphs. *SIAM J. Comput.* **3**, pp. 62–89.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* **22**, pp. 215–225.

TARJAN, R. E. 1979a. A class of algorithms which require non linear time to maintain disjoint sets. *J. Comput. Sys. Sci.* **18**, pp. 110–127.

TARJAN, R. E. 1979b. Application of path compression on balanced trees. *J. Assoc. Comput. Mach.* **26**, pp. 690–715.

TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.* **6**, pp. 306–318.

TARJAN, R. E. 1988. Personal communication.

TARJAN, R. E., AND VAN LEEUWEN, J. 1984. Worst-case analysis of set union algorithms. *J. Assoc. Comput. Mach.* **31**, pp. 245–281.

VAN EMDE BOAS, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Processing Lett.* **6**, pp. 80–82.

VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. 1977. Design and implementation of an efficient priority queue. *Math. Systems Theory* **10**, pp. 99–127.

VAN LEEUWEN, J., AND VAN DER WEIDE, T. 1977. Alternative path compression techniques. Technical Report RUU-CS-77-3, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands.

VAN DER WEIDE, T. 1980. *Data structures: an axiomatic approach and the use of binomial trees in developing and analyzing algorithms.* Mathematisch Centrum, Amsterdam. The Netherlands.

WARREN, D. H. D., AND PEREIRA, L. M. 1977. Prolog – the language and its implementation compared with LISP. *ACM SIGPLAN Notices* 12. pp. 109–115.

WESTBROOK, J., AND TARJAN, R. E. 1987. Amortized analysis of algorithms for set union with backtracking. Technical Report TR-103-87, Department of Computer Science. Princeton University, Princeton.

WILLARD, D. E. 1982. Maintaining dense sequential files in a dynamic environment. *Proc. 14th Annual ACM Symposium on Theory of Computing.* pp. 251–260.

YAO, A. C. 1976. On the average behavior of set merging algorithms. *Proc. 8th Annual ACM Symposium on Theory of Computing*, pp. 192–195.