Data Structures and Algorithms for Disjoint Set Union Problems

ZVI GALIL

Department of Computer Science, Columbia University, New York, NY 10027 and Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel

GIUSEPPE F. ITALIANO

Department of Computer Science, Columbia University, New York, NY 10027 and Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Rome, Italy

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

Categories and Subject Descriptors: E.1 [**Data**]: 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: equivalence algorithm, partition, set union, time complexity

INTRODUCTION

The *set union problem* has been widely studied during the past decades. The problem consists of maintaining a collection of disjoint sets under the operation of union. More precisely, the problem is to perform a sequence of operations of the following two kinds on disjoint 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.

The operations are presented on line; namely, each operation must be performed before the next one is known. Initially, the collection consists of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. The name of set $\{i\}$ is i. The input to the set union problem is therefore the initial collection of disjoint sets and the sequence of operations to be processed on line; the output of the problem consists of the answers to find operations. Figure 1 shows examples of union and find operations.

The set union problem has many applications in a wide range of areas. In the following we mention some of them, but the list is by no means exhaustive. Main applications of the set union problem include handling COMMON and EQUIVALENCE statements in FORTRAN compilers [Arden et al. 1961; Galler and Fischer 1964], implementing property grammars [Stearns and Lewis

CONTENTS

$\{1\}$ $\{2\}$ $\{3\}$ $\{4\}$ $\{5\}$ $\{6\}$

(a) The initial collection of disjoint sets Find(i) returns i, $1 \le i \le 6$

$\{1,3\}$ $\{2,5\}$ $\{4\}$ $\{6\}$

(b) The disjoint sets of (a) after performing union(1, 3) and union(5, 2) Find(1) and find(3) return 1, find(2) and find(5) return 5, find(4) returns 4, and find(6) returns 6

$\{1,3,4\}$ $\{2,5\}$ $\{6\}$

(c) The disjoint sets of (b) after performing union(4, 1) Find(1), find(3), and find(4) return 4, find(2) and find(5) return 5, and find(6) returns 6

$\{1,3,4,2,5\}$ $\{6\}$

(d) The disjoint sets of (c) after performing union(4, 5) Find(1), find(2), find(3), find(4), and find(5) return 4, and find(6) returns 6

Figure 1. Examples of union and find operations.

1969; Stearns and Rosenkrantz 1969], computational geometry problems [Imai and Asano 1987; Mehlhorn 1984c; Mehlhorn and Näher 1990], testing equivalence of finite state machines [Aho et al. 1974; Hopcroft and Karp 1971], computing the longest common subsequence of two sequences [Aho et al. 1983; Hunt and Szymanski 1977], performing unification in logic programming and theorem proving [Huet 1976; Vitter and Simons 1986], and several combinatorial problems such as finding minimum spanning trees [Aho et al. 1974; Kerschenbaum and van Slyke 1972], solving dynamic edge- and vertex-connectivity problems [Westbrook 1989 and Tarjan 1989b], 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].

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. 1989; Gambosi et al. 1988, 1989, 1991; Mannila and Ukkonen 1986a, 1988; Westbrook and Tarjan 1989a]. 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 [Ibaraki 1978; Pearl 1984].

In this paper we survey the most efficient algorithms known for set union problems and some of their variants. We present the algorithms and discuss some of their applications.

Preliminaries

Although the problems discussed in this paper have wide practical applications, we consider the analysis of the algorithms for their solution from the perspective of theoretical computer science. We concentrate on the ways in which resource usage (computing time, memory space) grows with increasing problem size. Therefore, we compare the merit of different algorithms based on formulas for their resource usage rather than on experimental information. Further, we ignore, for the most part, the constant factors in these formulas giving only order of magnitude evaluations. That is, we concentrate on asymptotic growth rates rather than on exact mathematical expressions.

We use the notation $O(f(n))$ (capital oh notation) to describe the set of functions that grow asymptotically *no faster* than $f(n)$. The notation $\Omega(f(n))$ describes the set of functions that grow asymptotically *at least* as fast as $f(n)$. Functions that grow asymptotically *at the same rate* as $f(n)$ are expressed by $\Theta(f(n))$. Formally the definitions for O, Ω, and Θ are as follows [Knuth 1976]:

$g(n) \in O(f(n))$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$

$g(n) \in \Omega(f(n)$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) \geq cf(n)$ for all $n \geq n_0$

$g(n) \in \Theta(f(n)$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) = cf(n)$ for all $n \geq n_0$

Sometimes it is useful to say that one function has strictly smaller asymptotic growth than another. The lowercase oh notation is defined as follows:

$$g(n) \in o(f(n)) \quad \text{if and only if} \quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0.$$

That is, $o(f(n))$ is the set of functions with order strictly smaller than $f(n)$.

We consider the analysis of algorithms from two different points of view. In one we analyze the time complexity of a sequence of operations; in the other we analyze the time complexity of a single operation. When we consider a sequence of operations, we estimate either the *worst case* of the algorithm (i.e., we consider those input instances of that size on which the algorithm takes the most time) or the *average case* of the algorithm (i.e., we average the execution times on instances of a certain size). When we consider a single operation, we estimate the *amortized complexity* [Tarjan 1985] (namely, we average the running time per operation over a worst-case sequence of operations) or the worst-case complexity. In case of the single operation complexity (either amortized or worst case), we sometimes give different bounds for the different operations (in our case unions and finds). Note that unless we give different bounds for the different operations, the worst-case complexity of a sequence of operations determines the amortized complexity of single operations and vice versa.

Models of Computation

Different models of computation have been developed for analyzing algorithms that solve set union problems. The main 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 unbonded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information, and no arithmetic is allowed to compute the address of a register. In this model two classes of algorithms were defined: separable pointer algorithms [Tarjan 1979a] and nonseparable pointer algorithms [Mehlhorn et al. 1988].

Separable pointer algorithms run on a pointer machine and satisfy the *separability* assumption as defined in Tarjan [1979a] (see below). A separable pointer algorithm makes use of a linked data structure, namely a collection of records and pointers that can be thought of as a directed graph: Each record is represented by a node, and each pointer is represented by an edge of the graph. The algorithm solves the set union problem according to the following rules [Blum 1986; Tarjan 1979a]:

(i) The operations must be performed on line; that is, each operation must be executed before the next one is known.

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

(iii) (*Separability*). After each operation, the data structure can be partitioned into disjoint subgraphs such that each subgraph corresponds to exactly one current set. The name of the set occurs in exactly one node in the subgraph. No edge leads from one subgraph to another.

(iv) To perform find(x), the algorithm obtains the node v corresponding to element x and follows paths starting from v until it reaches the node that 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. 1988] does not require the separability assumption. The only requirement is that the number of edges leaving each node must be bounded by some constant $c > 0$. More formally, rule iii 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.

Often these two classes of pointer algorithms admit quite different upper and lower bounds for the same problems.

A second 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. To make the various lower and upper bounds meaningful, it is usually assumed that the size of a register is bounded by $O(\log n)$[1] bits. A formal definition of random access machines can be found in Aho et al. [1974, pp. 5–14].

A third model of computation, known as the *cell probe model of computation*, was introduced by Yao [1981]. In this model, the cost of computation is measured by the total number of memory accesses to a random access memory with $\lceil \log n \rceil$ bits cell size. All other computations are considered to be free.

Organization of the Paper

The remainder of the paper consists of six sections. Section 1 surveys the most efficient algorithms known for solving the set union problem. Section 2 deals with the set union problem on adjacent intervals. Section 3 presents data structures that allow us to undo the last union performed. This result has been recently generalized in several directions, which are dealt with in Section 4. Section 5 shows some of the techniques used to obtain persistent data structures, as defined in Driscoll et al. [1989], for the set union problem. Finally, Section 6 lists some open problems and presents concluding remarks.

1. 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. The number of unions in any sequence of operations is bounded above by $n - 1$. Due to the definition of the union and find operations, there are two invariants that hold at any time. First, the sets are always disjoint and define a partition of the set $\{1, 2, \ldots, n\}$. Second, the name of each set corresponds to one of the items contained in the set itself. The set union

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

problem requires developing data structures and algorithms that are efficient for both union and find operations since the applications use both operations.

A different version of this problem considers the following operation instead of union:

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 many applications, where one is mostly concerned with testing whether two elements belong to the same set, no matter what the name of the set can be. 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.

1.1 Worst-Case Complexity

In this section, we describe algorithms for the set union problem [Tarjan 1975; Tarjan and van Leeuwen 1984], giving the optimal worst-case time complexity for a sequence of operations (and thus the amortized time complexity per operation). For the sake of completeness, we first survey some basic algorithms that have been proposed in the literature [Aho et al. 1974; Fischer 1972; Galler and Fischer 1964]. These are the *quick-find*, the *weighted quick-find*, the *quick-union*, and the *weighted quick-union* algorithms.

The *quick-find* algorithms allow one to perform find operations quickly; the *quick-union* algorithms allow one to perform union operations quickly. Their weighted counterparts speed these computations up by introducing some *weighting rules*.

Most of these algorithms represent sets as 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 corresponding set. The name of the set is stored in the root of the tree. Each tree node has a pointer to its parent. We refer to $p(x)$ as the parent of node x.

The *quick-find* algorithm can be described as follows. Each set is represented by a tree of height 1. Elements of the set are the leaves of the tree. The root of the tree is a special node that contains the name of the set. Initially, singleton set $\{i\}$, $1 \le i \le n$, is represented by a tree of height 1 composed of one leaf and one root. To perform a union(A, B), all the leaves of the tree corresponding to B are made children of the root of the tree corresponding to A. The old root of B is deleted. This maintains the invariant that each tree is of height 1 and can be performed in $O(|B|)$ time, where $|B|$ denotes the total number of elements in set B. Since a set can have as many as $O(n)$ elements, this gives an $O(n)$ time complexity in the worst case for each union. To perform a find(x), return the name stored in the parent of x. Since all trees are maintained of height 1, the parent of x is a tree root. Consequently a find requires $O(1)$ time. The same algorithm can be described by using an array instead of trees [Aho et al. 1974].

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

Union by size: 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 throughout any sequence of operations.

Although this rule does not improve the worst-case time complexity of each operation it improves to $O(\log n)$ the amortized bound of unions [Aho et al. 1974], as the following lemma shows.

Lemma 1.1.1

The weighted quick-find algorithm is able to perform each find operation in $O(1)$

time. The total time required to perform $n - 1$ union operations is $O(n \log n)$.

Proof. Each find operation is implemented as in the *quick-find* algorithm and therefore requires $O(1)$ time. Denote by N_i the total number of times a node i is moved from a set to another because of a union operation. The total time required to perform $n - 1$ union operations is

$$O\left(\sum_{i=1}^{n} N_i\right).$$

If the *union by size* rule is used, each time node i is moved from a set S_1 to another set S_2 because of a union (S_2, S_1), the size of the new set S_2 after this operation is at least twice the size of the set S_1 in which i was before. As a consequence, $N_i \le \log n$ for $i = 1, 2, \ldots, n$. Therefore, the total time needed to perform $n - 1$ unions is

$$O\left(\sum_{i=1}^{n} N_i\right) \le O(n \log n). \qquad \square$$

The *quick-union* algorithm [Galler and Fischer 1964] can be described as follows. Again, each set is represented by a tree. There are, however, two main differences with the data structure used by the *quick-find* algorithm. The first is that the height of a tree can be greater than 1. The second is that each node of each tree corresponds to an element of a set, and therefore there is no need for special nodes. Once again, the root of each tree contains the name of the corresponding set. A union (A, B) is performed by making the tree root of set B a child of the tree root of set A. A find(x) is performed by starting from the node x and 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. As a result, the *quick-union* algorithm is able to support each union in $O(1)$ time and each find in $O(n)$ time.

The 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 two *weighted quick-union* algorithms.

union by size: 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, in the following referred to as the *size* of a node, throughout all the sequence of operations.

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, throughout all the sequences of operations.

After a union(A, B), the name of the new tree root is set to A.

Lemma 1.1.2

If either union rule is used, any tree of height h must contain at least 2^h elements.

Proof. We prove only the case where the *union by size* rule is used. The case of *union by rank* can be proved in a similar fashion. We proceed by induction on the number of union operations. Before the first union, the lemma is clearly true since each set is of height 0 and contains at least $2^0 = 1$ element. Assume that the lemma is true before a union(A, B). Assume without loss of generality that $|A| \ge |B|$ so the root of B will be made a child of the root of A. Denote by $h(A)$ and $h(B)$ the heights of A and B before the union operation and by $h(A \cup B)$ the height of the combined tree $A \cup B$. Clearly, $h(A \cup B) = \max\{h(A), h(B) + 1\}$. We have

$$\begin{aligned} |A \cup B| &\ge \max\{|A|, 2|B|\} \\ &\ge \max\{2^{h(A)}, 2^{h(B)+1}\} \\ &= 2^{\max\{h(A), h(B)+1\}} = 2^{h(A \cup B)}. \end{aligned}$$

$\square$

As a consequence of the above lemma, the height of the trees achieved with either the *union by size* or the *union by*

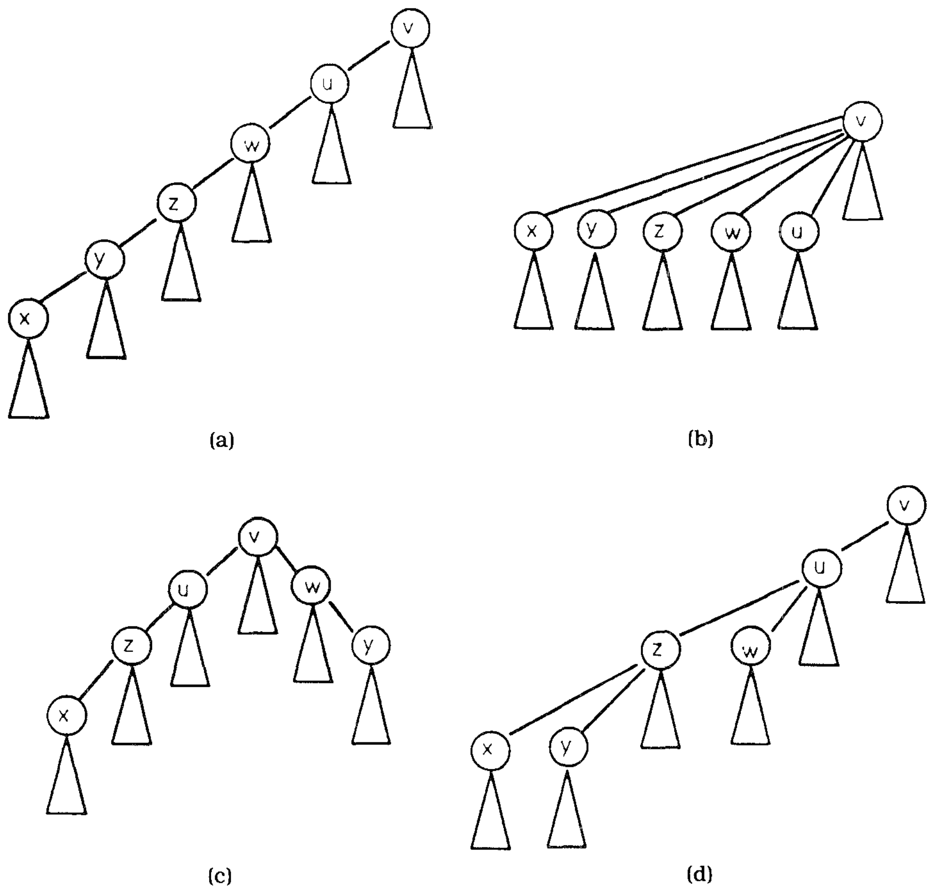

Figure 2. Path compaction techniques. (a) The tree before performing a find(x) operation; (b) path compression; (c) path splitting; (d) path halving.

rank rule is never more than $\lceil \log \rceil$ n. Henceforth, with either rule each union can be performed in $O(1)$ time and each find in $O(\log n)$ time.

A better amortized bound can be obtained if one of the following *compaction rules* is applied to the find path (see Figure 2).

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

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

path halving [van der Weide 1980; van Leeuwen and van der Weide 1977]: 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 [1928] function.

Theorem 1.1.1

[Tarjan and van Leeuwen 1984] The algorithms with either union rule combined with either compaction rule run in

$O(n + m\alpha(m + n, n))$ time on a sequence of at most $n - 1$ unions and m finds.

No better amortized bound is possible for separable and nonseparable pointer algorithms or in the cell probe model of computation.

Theorem 1.1.2

[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 $n - 1$ unions and m finds.

Theorem 1.1.3

[Fredman 1989; Fredman and Saks 1989] The set union problem requires $\Omega(n + m\alpha(m + n, n))$ worst-case time in the cell probe model of computation.

Theorem 1.1.4

[La Poutré 1990b] Any nonseparable pointer algorithm requires $\Omega(n + m\alpha(m + n, n))$ worst-case time for processing a sequence of $n - 1$ unions and m finds.

Although the six algorithms given above are all asymptotically optimal, there are certain differences of practical importance among the various union and compaction rules. First, *union by rank* seems preferable to *union by size* since it requires less storage. Indeed, it needs only $O(\log\log n)$ bits per node to store a rank in the range $[0, \lceil \log n \rceil]$, whereas *union by size* needs $O(\log n)$ bits per node to store a size in the range $[1, n]$. Second, path halving seems to be preferable among the three compaction rules. Path compression requires two passes over the find path, whereas path halving and path splitting can be implemented in only one pass. Furthermore, path halving has the advantage over path splitting in that it requires nearly as many as half pointer updates along the find path. Indeed, path halving needs to update every other node instead of every node in the find path.

An algorithm that performs find operations more efficiently has been recently proposed by La Poutré [1990a]. Following a technique first introduced by Gabow [1985], he showed how to support the ith find operation in $O(\alpha(i, n))$ time in the worst case, while still solving the set union problem in a total of $O(n + m\alpha(m + n, n))$ worst-case time.

We conclude this section by mentioning that the trade-off between union and find operations has been recently studied by Ben-Amran and Galil [1990]. They gave an algorithm that for any integer $k > 0$; supports unions in $O(a(k, n))$ amortized time and finds in $O(k)$ amortized time, where $a(k, n)$ is a row inverse of Ackermann's function [Tarjan 1983].

1.2 Single Operation Worst-Case Time Complexity

The algorithms that use any union and any compaction rule still have single-operation worst-case time complexity $O(\log n)$ [Tarjan and van Leeuwen 1984] since the trees created by any of the union rules can have a height as large as $O(\log n)$. Blum [1986] proposed a data structure for the set union problem that supports each union and find in $O(\log n / \log\log n)$ time in the worst case.

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 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 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 also contains 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 and T_A, only three cases are possible, which gives rise to three different types of unions.

Type 1: Root r_B is fat (i.e., it 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 fat and equal to r_A (the root of T_A). A new (slim) root r is created, and both r_A and r_B are made children of r.

Type 3: This deals with the remaining cases, that is, either root r_B is slim or v is equal to r_A and slim. If root r_B is slim, all the children of r_B are made the rightmost children of v. Otherwise, v is equal to r_A and is slim. In this case, all the children of $v = r_A$ are made the rightmost children of r_B.

Theorem 1.2.1

[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, whereas type 3 unions require at most $O(k)$ time due to the definition of a slim root. Choosing $k = \lceil \log n / \log\log n \rceil$ yields the claimed time bound. The space complexity is derived easily from the fact that a *k-UF tree* with ℓ leaves has at most $2\ell - 1$ nodes. Henceforth the forest of *k-UF trees* that store the disjoint sets requires the most a total of $O(n)$ space. □

Blum also showed that this bound is tight for the class of separable pointer algorithms.

Theorem 1.2.2

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

Recently, Fredman and Saks [1989] showed that the same lower bound holds in the cell probe model of computation.

Theorem 1.2.3

[Fredman and Saks 1989] Any algorithm for the set union problem requires $\Omega(\log n / \log\log n)$ single-operation worst-time case in the cell probe model of computation.

1.3 Average Case Complexity

The average running time of the quick union, quick-find, and weighted quick-find algorithms described in Section 1.1 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 assume $O(n)$ union and find instructions are being performed. This is not a significant restriction for the asymptotic time complexity as shown, for instance, by Hart and Sharir [1986]. We denote by $E[t_n^{(QF)}]$, $E[t_n^{(WQF)}]$, and $E[t_n^{(QU)}]$ the average running time of quick-find, weighted quick-find, and quick-union algorithms, respectively, which perform $O(n)$ set union operations on n elements. Three models of random input sequences have been considered in the literature: the *random graph model*, the *random spanning tree model*, and the *random components model.*

1.3.1 Random Graph Model

The random graph model was proposed by Yao [1976] based upon the random graph model by Erdös and Rényi [1960]. Each of the $\binom{n}{2}$ undirected edges between two vertices of an n-vertices graph is chosen independently according to a Poisson process. When an edge (x, y) is chosen, a *union*(x, y) is executed if x and y are in different sets.

Theorem 1.3.1

[Bollobás and Simon 1985; Knuth and Schönage 1978] The average running time of quick-find and weighted quick-find algorithms in the random graph model is

$$E\left[t_n^{(QF)}\right] = \frac{n^2}{8} + o\left(n(\log n)^2\right);$$

$$E\left[t_n^{(WQF)}\right] = cn + o\left(\frac{n}{\log n}\right),$$

where $c = 2.0847\ldots$.

1.3.2 Random Spanning Tree Model

Each sequence of union operations corresponds to a "union tree" in which the edge (x, y) means that the set containing x is merged into the set containing y. In the random spanning tree model, all possible union trees are equally likely; there are n^{n-2} possible unoriented trees and $(n\text{-}1)!$ ways of choosing edges in each tree.

Theorem 1.3.2

[Knuth and Schönage 1978; Yao 1976] The average running time of quick-find and weighted quick-find algorithms in the random spanning tree model is

$$E\left[t_n^{(QF)}\right] = \sqrt{\frac{\pi}{8}}\, n^{3/2} + O(n \log n);$$

$$E\left[t_n^{(WQF)}\right] = \frac{1}{\pi} n \log n + O(n).$$

1.3.3 Random Components Model

In the simplest model, it is assumed that at any given time each pair of sets is equally likely to be merged by a union operation. This is also the least realistic model. Indeed, this assumption does not apply when one is interested in joining sets containing two elements chosen independently with uniform probability.

Theorem 1.3.3

[Doyle and Rivest 1976] The average running time of the quick-union algorithm in the random components model is

$$n \le E\left[t_n^{(QU)}\right] \le 2n.$$

The reader is referred to the original papers [Bollobás and Simon 1985; Doyle and Rivest 1976; Knuth and Schönage 1978; Yao 1976] for details concerning the analysis of the expected behavior of these algorithms. Besides the significance of the three random input models chosen, these results underline that *weighted quick-find* algorithms are much faster than *quick-find* algorithms not only in the worst case but also in the average.

1.4 Special Linear Cases

The six algorithms using either union rule and either compaction rule as described in Section 1.1 run in $O(n + m\alpha(m, n))$ time on a sequence of at most $n - 1$ union and m find operations. No better amortized bound is possible for separable and nonseparable pointer algorithms or in the cell probe model of computation. As a consequence, to get a better bound, we must consider a special case of set union. Gabow and Tarjan [1985] used this idea to devise one random access machine algorithm that runs in linear time for a special case in which the structure of the unions is known in advance. Interestingly, Tarjan's lower bound for separable pointer algorithms also applies to this restricted problem. This result is of theoretical interest and is significant in many applications, such as scheduling problems, the off-line min problem, finding maximum matching on graphs, VLSI channel routing, finding nearest common ancestors in trees, and flow graph reducibility [Gabow and Tarjan 1985].

The problem can be formalized as follows. We are given a tree T containing n nodes that 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 is 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. We first briefly sketch the solution of the static tree set union problem.

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 facilitate access to and manipulation of microsets.

The basic ideas underlying the algorithm are the following. First, a priori knowledge about the static union tree can be used to precompute the answers to the operations performed in microsets using a table look up. Second, any one of the six optimal algorithms described in Section 1.1 can be used 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.

Theorem 1.4.1

[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).

The same algorithm given for the static tree set union can be extended to the incremental tree set union problem. For this problem, the union tree is not known in advance but is allowed to grow only one node at the time during the sequence of union and find operations performed. This problem has application in several algorithms for finding maximum matching in general graphs. The incremental tree set union algorithm is similar to the algorithm for static tree set union. The only difference is in the construction of microsets, which now might change over time because of new node insertions in the union tree T. The basic idea is to start with only one microset, the root of T. When a new node w is inserted to T, say as a child of v, w is put in the same microset as v. If the size of this microset does not exceed b, nothing need be done. Otherwise, the microset is split into two new microsets. It can be proved that this split can be performed in $O(b)$ time and the total number of such splits is at most $O(n/b)$. Therefore, the incremental tree set union problem can also be solved in linear time.

Theorem 1.4.2

[Gabow and Tarjan 1985] The algorithm for incremental tree set union runs in a total of O(m + n) time and requires O(n) preprocessing time and space.

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. A slightly more general class of input sequences, denoted by *local postorder*, was proved not to be linear, but its rate of growth is not provable in the theory of finite sets. A preliminary version of these results was reported in Loebl and Nešetřil [1988].

1.5 Applications of Set Union

In this section, we list a few applications of the set union problem coming from areas such as combinatorial optimization, graph algorithms, and theorem proving. We describe how to take advantage of set union algorithms for finding minimum spanning trees [Aho et al. 1974; Kerschenbaum and van Slyke 1972], for maintaining on-line connected components of undirected graphs [Even and Shiloach 1981], and for performing unification of first-order terms [Aït-Kaci 1986; Aït-Kaci and Nasr 1986; Huet 1976; Vitter and Simons 1986].

A *spanning tree* $S = (X, T)$ of a graph $G = (V, E)$ is a tree that has the same vertex set G (i.e., $X = V$) and that contains only edges of G (i.e., $T \subseteq E$). If each edge (i, j) of G is assigned a cost $c(i, j)$, then the cost of spanning tree is defined as

$$c(S) = \sum_{(i,j) \in T} c(i,j).$$

A *minimum spanning tree* of G is a spanning tree of minimum cost. This problem arises naturally in many applications, including communication networks and distributed computation. Aho et al. [1974] give the following algorithm to compute the minimum spanning tree of a graph, which was first developed by Kruskal [1956]:

```
begin
  $S \leftarrow \emptyset$;
  for each vertex $v \in V$ do
      initialize a singleton set $\{v\}$;
  while there is more than one set left
    do begin
      choose the cheapest edge $(u, v) \in E$;
      delete $(u, v)$ from $E$;
      $A \leftarrow FIND(u)$;
      $B \leftarrow FIND(v)$;
      if $A \neq B$ then begin
          $UNION(A, B)$;
          add $(u, v)$ to $S$
      end;
  end;
end
```

The algorithm starts with a spanning forest of n singleton trees and examines all the edges of the graph in order of increasing costs. When an edge (u, v) is examined, the algorithm checks whether (u, v) connects two trees of the spanning forest. This is done by checking whether vertices u and v are in different trees of the spanning forest. If so, the trees are combined and (u, v) is inserted into the spanning tree. As shown in the above pseudocode, this algorithm can be efficiently implemented if the trees of the spanning forest are represented as disjoint sets subject to union and find operations.

There are numerous spanning tree problems that also benefit from ideas behind the set union algorithms. Among them are testing whether a given spanning tree is minimum [Tarjan 1979b], performing sensitivity analysis on minimum spanning trees [Tarjan 1982], and finding edge replacements in minimum spanning trees [Tarjan 1979b].

Another application of set union algorithms is the on-line maintenance of the connected components of an undirected graph. Namely, we are given an undirected graph in which edges can be inserted one at a time and for which connectivity questions, such as "Are vertices u and v in the same connected component?" must be answered at any time in an on-line fashion. As noted by Even and Shiloach [1981], this problem can be solved by maintaining the connected components of the graph as disjoint sets subject to union and find operations. Checking whether two vertices u and v are in the same connected component can be performed by testing whether find(u) = find(v). The insertion of a new edge (x, y) can be accomplished by first checking whether x and y are already in the same connected component by means of two find operations. If so, the representation of disjoint sets need not be changed. Otherwise, x and y are in two different connected components and therefore in two different sets, say A and B. As a result of the insertion of edge (x, y), A and B are combined through a union operation. By using the set union algorithms with any union rule and path compaction, the total time required to maintain an n-vertices graph under m edge insertions and k connectivity question is $O(n + k\alpha(k, n))$.

A third application of set union algorithms is the unification problem. Informally, the unification problem can be stated as follows: "Given two different descriptions, can an object fitting both descriptions be found?" More formally, a description is an expression in logic composed of function symbols, variables, and constants. Finding value of variables that make two expressions equal is an important process in logic and deduction called unification. The reader is referred to the excellent survey of Knight [1989] for a more precise definition of the unification problem and its significance in different interdisciplinary areas.

Many algorithms proposed to solve the unification problem exploit the efficiency of set union algorithms [Aït-Kaci 1986; Huet 1976; Vitter and Simons 1986]. The main idea behind those algorithms is to use a graph representation for terms to be unified, where function symbols, variables, and constants are the vertices of the graph. Initially, each node is in a singleton class. Throughout the execution of the algorithm, equivalence classes of vertices are maintained. If two vertices u and v belong to the same equivalence class, they have been unified by the algorithm. As the unification proceeds, different equivalence classes are combined together. If at the end of this algorithm a class contains different function symbols, the two original terms were not unifiable. For more details see Huet [1976] and Vitter and Simons [1986]. We note that these algorithms resemble another algorithm (again based upon set union data structures) required to solve equivalence of finite state machines [Hopcroft and Karp 1971] (see Aho et al. [1974, pp. 143–145]).

Another interesting application of set union algorithms is the depth determination problem. In this problem, one is interested in maintaining information about the depth of vertices in a forest of trees subject to update operations, such as linking two trees by inserting a new edge. Details can be found in Aho et al. [1974, pp. 141–142].

2. SET UNION PROBLEM ON INTERVALS

In this section, we describe efficient solutions to the *set union problem on intervals*, which can be defined as follows. Informally, we would like to maintain a partition of a list $\{1, 2, \ldots, n\}$ in adjacent intervals. A union operation joins two adjacent intervals, a find returns the name of the interval containing x, and a split divides the interval containing x (at x itself). More formally, at any time we maintain a collection of disjoint sets A_i with the following properties. The A_i's, $1 \leq i \leq k$, are disjoint sets whose members are ordered by the relation $\leq$ and such that

$$\bigcup_{i=1}^{k} A_i = \{1, 2, \ldots, n\}.$$

Furthermore, every item in A_i is less than or equal to all the items in A_{i+1}, for $i = 1, 2, \ldots, n - 1$. In other words, the intervals A_i partition the interval $[1, n]$. Set A_i is said to be adjacent to sets A_{i-1} and A_{i+1}. The set union problem on intervals consists of performing a sequence of the following operations:

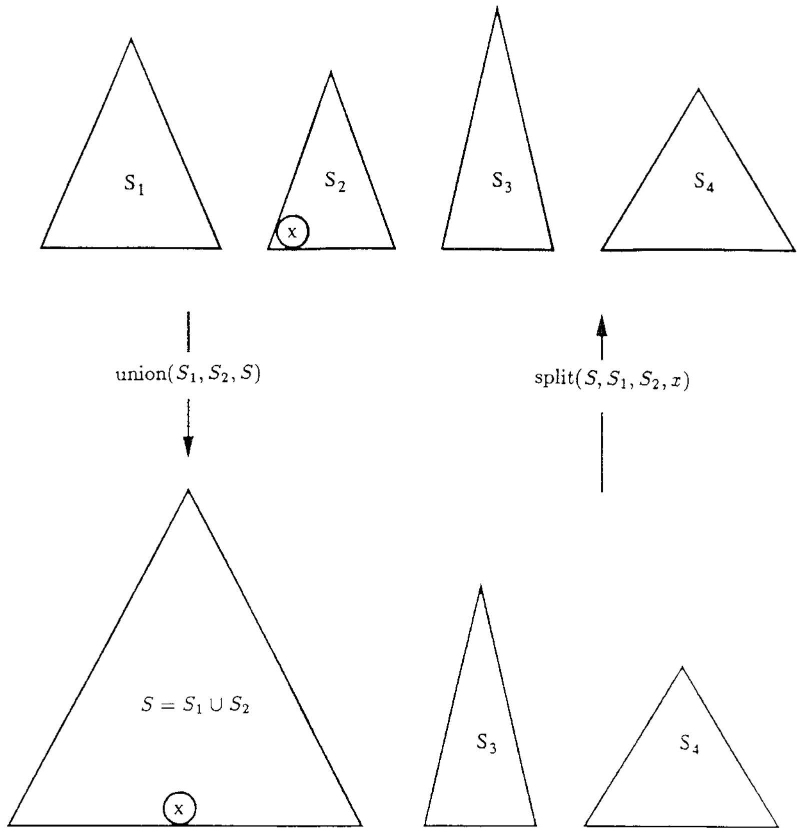

Figure 3. Union and split operations

union(S_1, S_2, S): Given the adjacent sets S_1 and S_2, combine them into a new set $S = S_1 \cup S_2$.

find(x): Given the item x, return the name of the set containing x.

split(S, S_1, S_2, x): Partition S into two sets $S_1 = \{a \in S \mid a < x\}$ and $S_2 = \{a \in S \mid a \geq x\}$. Figure 3 shows examples of union and split operations.

Adopting the same terminology used in Mehlhorn et al. [1988], we will refer to the set union problem on intervals as the *interval union-split-find problem*. After discussing this problem, we consider two particular cases: the *interval union-find problem* and the *interval split-find problem*, where only union-find and split-find operations are allowed, respectively.

The interval union-split-find problem and its subproblems have applications in a wide range of areas, including problems in computational geometry such as dynamic segment intersection [Imai and Asano 1987; Mehlhorn 1984c; Mehlhorn and Näher 1990], shortest paths problems [Ahuja et al. 1990; Mehlhorn 1984b], and the longest common subsequence problem [Aho et al. 1983; Hunt and Szymanski 1977].

2.1 Interval Union-Split-Find

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

The upper bound for separable pointer algorithms can be easily obtained by means of balanced trees [Aho et al. 1974; Adelson-Velskii and Landis 1962; Mehlhorn 1984a; Nievergelt and Reingold 1973], whereas the following lower bound holds.

Theorem 2.1.1

[Mehlhorn et al. 1988] For any separable pointer algorithm, both the worst-case per operation time complexity of the interval split-find problem and the amortized time complexity of the interval 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 [1990], van Emde Boas [1977], and van Emde Boas et al. [1977]. In particular, van Emde Boas et al. [1977] introduced a priority queue that supports among other operations *insert*, *delete*, and *successor* on a set with elements belonging to a fixed universe $S = \{1, 2, \dots, n\}$. 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 $O(n)$. It can be shown [Mehlhorn et al. 1988] that the above operations correspond to union, split, and find, respectively, and therefore the following theorem holds.

Theorem 2.1.2

[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 $O(n)$.

No better bound can be achieved for nonseparable pointer algorithms.

Theorem 2.1.3

[Mehlhorn et al. 1988] For any nonseparable pointer algorithm, both the worst-case per operation time complexity of the interval split-find problem and the amortized time complexity of the interval union-split-find problem are $\Omega(\log\log n)$.

Notice that this implies that for the interval union-split-find problem the separability assumption causes an exponential loss of efficiency.

2.2 Interval Union-Find

The interval union-find problem is a restriction of the set union problem described in Section 1, where only adjacent intervals are allowed to be joined. Henceforth, both the $O(\alpha(m + n, n)$ amortized bound given in Theorem 1.1.1 and the $O(\log n/\log\log n)$ single-operation worst-case bound given in Theorem 1.2.1 still hold.

Whereas Tarjan's proof of the $\Omega(\alpha(m + n, n))$ amortized lower bound also holds for the interval union-find problem, Blum's proof does not seem to be easily adaptable to the new problem. Hence, it remains an open problem whether a 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 [1985] used the data structure described in Section 1.4 to obtain an $O(1)$ amortized time on a random access machine.

2.3 Interval Split-Find

According to Theorems 2.1.1, 2.1.2 and 2.1.3, the two algorithms given for the more general interval union-split find problem are still optimal for the single-operation worst-case time complexity of the interval 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,[2] as shown by Hopcroft and Ullman [1973]. Their algorithm is based upon an extension of an idea by Stearns and

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

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.

Initially, such a tree with n leaves is created. Its height is $O(\log^* n)$, and therefore a find(x) 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 shown by Gabow [1985]. The algorithm used to establish this upper bound relies on a sophisticated partition of the items contained in each set.

Theorem 2.3.1

[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 $O(n)$.

Very recently, La Poutré [1990b] proved that this bound is tight for (both separable and nonseparable) pointer algorithms.

Theorem 2.3.2

[La Poutré 1990b] Any pointer algorithm requires $\Omega(n + m\alpha(m, n))$ time to perform $n - 1$ split and m find operations.

Using the power of a random access machine, Gabow and Tarjan [1985] were able to achieve $\Theta(1)$ amortized time for the interval split-find problem. This bound is obtained by using a slight variant of the data structure sketched in Section 1.4.

2.4 Applications of Set Union on Intervals

The problem of determining the longest common subsequence of two input sequences is an application of set union on intervals. This problem has many applications, including sequence comparison in molecular biology and the widely used *diff* file comparison program [Aho et al. 1983]. The problem can be defined as follows. A subsequence of a sequence x is obtained by removing elements from x. The elements removed need not necessarily be contiguous. The *longest common subsequence* of two sequences x and y is a longest sequence that is a subsequence of both x and y.

Hunt and Szymanski [1977] devised a solution to this problem based upon set union on intervals. Denote by $x = x_1, x_2, \ldots, x_m$ and $y = y_1, y_2, \ldots, y_n$ the two sequences whose longest common subsequence we would like to compute. Without loss of generality, assume $m < n$ (otherwise exchange the role of the two sequences in what follows). For each symbol a of the alphabet over which the two sequences are defined, compute $OCCURRENCES(a) = \{i \mid y_i = a\}$. This gives us all the positions of the sequence y at which symbol a appears.

For the sake of simplicity, we only mention how to find the length of the longest common subsequence of x and y. The longest common subsequence itself can be found with some additional bookkeeping. The algorithm considers each symbol x_j, $1 \leq j \leq m$, and computes, for $0 \leq i \leq n$, the length of the longest common subsequence of the two prefixes $x_1, x_2, \ldots, x_j$ and $y_1, y_2, \ldots, y_i$. To accomplish this task efficiently, for a fixed j we define sets A_k of indexes as follows. A_k consists of all the integers i such that the longest common subsequence of $x_1, x_2, \ldots, x_j$ and $y_1, y_2, \ldots, y_i$ has length k. Notice that the sets A_k partition $\{1, 2, \ldots, n\}$ into adjacent intervals since each A_k contains consecutive integers and items in A_{k+1} are larger than those in A_k for any k. Assume we have already computed the sets A_k up to position $j - 1$ of the string x. We show how to update them to apply the position j. For each r in $OCCURRENCES(x_j)$, we consider whether we can add the match between x_j and y_r to the longest common subsequence of whether we can add the match between x_j and y_r to the longest common subsequence of $x_1, x_2, \ldots, x_j$ and $y_1, y_2, \ldots, y_r$. The crucial point is that if both $r - 1$ and r are in A_k, then all the indexes $s \geq r$ belong to A_{k+1} when x_j is considered. The following pseudocode describes this algorithm. The reader is referred to Hunt and Szymanski [1977] for details.

```
begin
  initialize $A = \{0, 1, \ldots, n\}$;
  for $i \leftarrow 1$ to $n$ do
      $A_i \leftarrow \emptyset$;
  for $j \leftarrow 1$ to $n$ do
      for $r \in OCCURRENCES(x_j)$ do begin
          $k \leftarrow FIND(r)$;
          if $k = FIND(r - 1)$ then begin
              $SPLIT(A_k, A_k, A'_k, r)$;
              $UNION(A'_k, A_{k+1}, A'_k)$
          end;
      end;
  return($FIND(n)$)
end
```

We conclude this section by mentioning that Hunt and Szymanski's algorithm has been recently improved by Apostolico and Guerra [1987] and by Eppstein et al. [1990]. Both these algorithms use similar paradigms but more sophisticated data structures.

3. SET UNION PROBLEM WITH DEUNIONS

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

denunion: Undo the most recently performed union operation not yet undone.

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

3.1 Algorithms for Set Union with Deunions

Recently, the amortized complexity of set union with deunions was characterized by Westbrook and Tarjan [1989a], 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 also hold for the single-operation worst-case time complexity of the problem.

We now describe $\Theta(\log n / \log\log n)$ amortized algorithms for the set union problem with deunions [Mannila and Ukkonen 1987; Westbrook and Tarjan 1989a]. 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, since it can be seen by first performing $n - 1$ unions that build a binomial tree (as defined, for instance, in 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 otherwise as *dead*. 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 the path compaction performed during the find operations and are called *find pointers*. Each of these pointers is associated with 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 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 is used to maintain set names and either sizes or ranks.

There are actually two versions of these algorithms, depending on when dead pointers are removed 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* remove 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 3.1.1

Either union by size or union by rank in combination with either path splitting or path halving gives both eager and lazy algorithms that 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 [1989a]. The space bound for the eager algorithms is $O(n)$. The space complexity of the lazy algorithms can be shown to be $O(n)$ by following the stamping technique introduced by Gambosi et al. [1989b], which establishes that the lazy algorithms require no more space than their eager counterparts. □

This bound is tight for separable pointer algorithms.

Theorem 3.1.2

[Westbrook and Tarjan 1989a] Every separable pointer algorithm for the set union problem with deunions requires at least $\Omega(\log n / \log \log n)$ amortized time per operation.

As for the single-operation worst-case time complexity of set union with deunions, an extension of Blum's data structure described in Section 1.2 can also support 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.

3.2 Applications of Set Unions with Deunions

The main application of set union with deunions arise in logic programming interpreter memory management without function symbols [Mannila and Ukkonen 1986b]. Indeed, the most popular logic programming language, Prolog, uses unification and backtracking as crucial operations [Warren and Pereira 1977]. We now consider the following example, borrowed from Clocksin and Mellish [1981] to show how unification and backtracking work in Prolog.

Consider a database consisting of the following assertions

likes(mary, food)
likes(mary, wine)
likes(john, wine)
likes(john, mary)

whose meaning is representing the facts that Mary likes food, Mary and John like wine, and John likes Mary. The question, "Is there anything John and Mary both like?" is phrased in Prolog as follows:

?- likes(mary, X), likes(john, X).

Prolog answers this question by first attempting to unify the first term of the query with some assertion in the database. The first matching fact found is **likes(mary, food)**. As a result, the terms **likes(mary, food)** and **likes(mary, X)** are unified and Prolog instantiates **X** to **food** everywhere **X** appears in the query. Now the database is searched for the second term in the query, which is **likes(john, food)** because of the above substitution. But these term fails to unify with any other term in the database.

Then Prolog backtracks, that is, it "undoes" the last unification performed: It undoes the unification of **likes(mary, food)** with **likes(mary, X)**. As a result, the variable **X** becomes noninstantiated again. Then Prolog tries to reunify the first term of the query with another term in the database. The next matching fact is **likes(mary, wine)**, and therefore the variable **X** is instantiated to wine everywhere **X** appears. As before, Prolog now tries to unify the second term, searching this time for **likes(john, wine)**. This can be unified with the third assertion in the database and Prolog notifies the user with the answer

X = wine.

Consequently, the execution of a Prolog program without function symbols can be seen as a sequence of unifications and deunifications. Therefore, data structures for the set union problems with deunions can be effectively used for this problem. More details on this application can be found in Mannila and Ukkonen [1986b].

4. EXTENSIONS OF THE SET UNION PROBLEM WITH DEUNIONS

Recently, other variants of the set union problem with deunions have been considered, such as set union with arbitrary deunions [Mannila and Ukkonen 1988], set union with dynamic weighted backtracking [Gambosi et al. 1991c], and set union with unlimited backtracking [Apostolico et al. 1989]. In the following sections, we will discuss these problems and give the best algorithms known for their solutions.

4.1 Set Union with Arbitrary Deunions

Mannila and Ukkonen [1988] introduced a 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 that currently contains element x.

deunion(i): Undo the ith union so far performed.

After a deunion(i), the name of the sets are as if the ith union has never occurred.

Motivations for studying this problem arise in the incremental execution of logic programs [Mannila and Ukkonen 1988]. The following lower bound can be obtained by reduction to the interval union-find-split problem [Mehlhorn et al. 1988], as characterized in Theorems 2.1.1 and 2.1.3.

Theorem 4.1.1

[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.

There is actually an $O(\log n)$ time algorithm matching this lower bound, as the following theorem states.

Theorem 4.1.2

[Galil and Italiano 1991] There exists a data structure that supports each union,

find, and deunion(i) in O(log n) time and O(n) space.

4.2 Set Union with Dynamic Weighted Backtracking

Gambosi et al. [1991] considered a further extension of the set union problem with deunions by assigning weights to each union and by allowing backtracking either to the union of maximal weight or to a generic union so far performed. They 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 ith union performed, $\Delta > 0$.

decrease_weight(i, Δ): decrease by Δ the weight of the ith 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 ith 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], whereas the weight associated with a union is used to evaluate the goodness of the state resulting from the unification to which the union is associated. Thus, backtracking corresponds to returning 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 the weight associated with each union already performed to be updated (both increased and decreased). This operation adds more heuristic power to the algorithms for logic programming interpreter memory management and therefore improves the practical performance of the previous known "blind" uniform-cost algorithms. Backtracking to the state just before any union is performed is 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].

Theorem 4.2.1

[Gambosi et al. 1991] 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). This bound is the best possible for any nonseparable pointer algorithm.

Better amortized bounds can be achieved. Indeed, it is possible to perform each *backweight*, *backtrack*, and *increase_weight* in $O(1)$ amortized time and *decrease_weight* and *union* in $O(\log n)$ amortized time. Finds can be supported in $O(\log n/\max\{1, \log(\gamma \log n)\})$ amortized time, where γ is the ratio of the number of finds to the number of unions and backtracks in the sequence. This can be obtained by using the data structure described in Westbrook and Tarjan [1989a] in combination with Fibonacci heaps [Fredman and Tarjan 1987]. The space required is $O(n)$.

4.3 Set Union with Unlimited Backtracking

A further generalization of the set union problem with deunions was considered in Apostolico et al. [1989]. This generalization is called the *set union problem with unlimited 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 otherwise by *dead*. In the set union problem with unlimited 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 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 backtracking, where 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 $O(\log n)$. Apostolico et al. [1989] showed that the time complexity of this problem is $\Theta(\log n/\log\log n)$ for separable pointer algorithms when unites instead of unions are performed (i.e., when the name of the new set can be arbitrarily chosen).

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 unlimited 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.

Using either Westbrook and Tarjan's algorithms or Blum's augmented data structure, each backtrack(i) can require $\Omega(i \log n/\log\log n)$ in the worst case. 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 in the *eager* methods described in Section 3.1). This is so, 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 $n - 1$. To overcome this difficulty, dead pointers must be removed in a *lazy* fashion.

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

Theorem 4.3.1

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 4.2.1 since unlimited backtracking is a particular case of weighted backtracking. Furthermore, the proof of the lower bound for set union with weighted backtracking can be adapted for the new problem. □

If *unite* operations (instead of *unions*) are performed, the upper bound reduces to $O(\log n/\log\log n)$. No better bound is possible for separable pointer algorithms.

Theorem 4.3.2

[Apostolico et al. 1989] There exists a data structure that supports each unite and find operation in O(log n / log log n) time, each backtrack in O(1) time, and requires O(n) space.

No better bound is possible for any separable pointer algorithm or in the cell probe model of the computation.

Theorem 4.3.3

For any n, any separable pointer algorithm for the set union with unlimited backtracking has single-operation time complexity at least Ω(log n / log log n) in the worst case. The same lower bound holds also in the cell probe model of computation.

Proof. It is a trivial extension of Theorems 1.2.2 and 1.2.3, which state that it suffices to consider only unite and find operations. □

It is somewhat surprising that the two versions of the set union problem with unlimited backtracking have such a different time complexity and that the version with unites can be solved 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 significant restriction in many applications, where one is mostly concerned with testing whether two elements belong to the same equivalence class, no matter what the name of the class. 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 a union cannot arbitrarily choose a new name. The crucial idea behind the proof of Theorem 2 in Mannila and Ukkonen [1988] is that at some point we may have to choose among $\Theta(n)$ 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 unlimited 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 $\Omega(\log n)$.

5. PERSISTENT DATA STRUCTURES FOR SET UNION

In this section we describe persistent data structures for the set union problem [Driscoll et al. 1989; Overmars 1983]. Following Driscoll et al. [1989], we define a data structure to be *ephemeral* when an update destroys the previous version. A *partially persistent* data structure supports access to multiple versions, but only the most recent version can be modified. A data structure is said to be *fully persistent* if every version can be both accessed and modified.

The *fully persistent disjoint set union problem* can be defined as follows. Throughout any sequence of operations, multiple versions of a set union data structure are maintained (i.e., multiple versions of the partition are maintained). Union operations are updates, whereas find operations are accesses. If the jth union operation applies to version v, $v \leq j$, the result of the update is a new version j. The operations on the fully persistent data structure can be defined as follows (we use uppercase initials to distinguish them from the corresponding operations on the ephemeral data structure).

Union(x, y, v): Denote by X and Y the two sets in version v containing, respectively, x and y. If $X = Y$, do nothing. Otherwise, create a new version in which X and Y are combined into a new set. The new set gets the same name as X.

Find(x, v): Return the name of the set containing element x in version v.

Initially the partition consists of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$, and the name of set $\{i\}$ is i. This is version 0 of the set union data structure. The restricted case in which union operations are allowed to modify only the most recent version defines the *partially persistent disjoint set union problem*.

We first consider the partially persistent disjoint set union problem. As in the case of the set union problem with unlimited 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 5.1

[Mannila and Ukkonen 1988] There exists a data structure that solves the partially persistent disjoint set union problem in $O(\log n)$ worst-case time per operation with an $O(n)$ space usage. No better bound is possible for pointer based algorithms.

The amortized time of a union can be further reduced to $O(1)$ by using the data structures introduced in Brown and

Tarjan [1980] and 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. [1990] and Mannila and Ukkonen [1988]. Furthermore, if we perform unites instead of unions, a better algorithm can be found.

Theorem 5.2

[Apostolico et al. 1989] If unites instead of unions have to performed, the partially persistent disjoint set union problem can be solved in $O(\log n/\log\log n)$ time per operation with an $O(n)$ space usage. No better bound is possible for separable pointer algorithms and in the cell probe model of computation.

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

Turning to the fully persistent disjoint set union problem, Italiano and Sarnak [1991] gave an efficient data structure for this problem that achieves the following bounds.

Theorem 5.3

[Italiano and Sarnak 1991] The fully persistent disjoint set union problem can be solved in $O(\log n)$ worst-case time per operation and in $O(1)$ amortized space per update. No better pointer based algorithm is possible.

6. CONCLUSIONS AND OPEN PROBLEMS

In this paper we described the most efficient known algorithms for solving the set union problem and some of this problem variants. Most of the algorithms we described are optimal with respect to a certain model of computation (e.g., pointer machines with or without the separ-ability assumption, random access machines). There are still several intriguing problems in all the models of computation we have considered. Fredman and Saks' lower bound analysis [Fredman 1989; Fredman and Saks 1989] settled both the amortized time complexity and the single operation worst-case time complexity of the set union problem in the cell probe model of computation. La Poutré [1990b] settled the amortized time complexity of the set union problem for nonseparable pointer algorithms. It is still open, however, as to whether in these models the amortized and the single operation worst-case complexity of the set union problems with deunions or backtracking can be improved. Furthermore, there are 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 nor is there one for the worst-case complexity of interval union-find. In the realm of nonseparable pointer algorithms, it remains open whether the $O(\log n/\log\log n)$ worst-case bound [Blum 1986] for interval union-find is tight. This problem requires $\Theta(1)$ amortized time on a random access machine as shown by Gabow and Tarjan [1985].

ACKNOWLEDGMENTS

We would like to thank Alberto Apostolico, Giorgio Ausiello, Hal Gabow, Lane Hemachandra, Heikki Mannila, Kurt Mehlhorn, Neil Sarnak, Bob Tarjan, Esko Ukkonen, Jan van Leeuwen, and Henryk Wozniakowski who read earlier versions of this paper and supplied many helpful comments. We are also grateful to Salvatore March and to the anonymous referees for their valuable suggestions.

This research was supported in part by NSF Grant CCR-88-14977, by the ESPRIT II Basic Research Actions Program of the European Communities under contract 3075 (Project ALCOM), and by the Italian MURST Project, Algoritmi e Strutture di Calcolo. The second author was supported in part by an IBM Graduate Fellowship.

REFERENCES

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

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

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1973. On computing least common ancestors in trees. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, 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. 1990. Faster algorithms for the shortest path problem. *J. ACM 37*, 213-223.

AÏT-KACI, H. 1986. An algebraic semantics approach to the effective resolution of type equations. *Theor. Comut. Sci. 45*, 293-351.

AÏT-KACI, H., AND NASR, R. 1986 LOGIN: A logic programming language with built-in inheritance. *J. Logic Program. 3.*

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

APOSTOLICO, A., GAMBOSI, G., ITALIANO, G. F., AND TALAMO, M 1989. The set union problem with unlimited backtracking. Tech. Rep. CS-TR-908, Dept. of Computer Science, Purdue University.

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

BANACHOWSKI, L. 1980. A complement of Tarjan's result about the lower bound on the complexity of the set union problem. *Inf. Process. Lett. 11*, 59-65.

BEN-AMRAM, A. M., AND GALIL, Z. 1988. On pointers versus addresses. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, pp 532-538. To appear in *J. ACM.*

BEN-AMRAM, A. M., AND GALIL, Z. 1990. On a tradeoff in data structures. Manuscript.

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

BOLLOBÁS, B., AND SIMON, I. 1985. On the expected behaviour of disjoint set union problems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing* ACM Press, New York, NY, 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*, 594-614.

CLOCKSIN, W. F., AND MELLISH, C. S. 1981. *Programming in Prolog.* Springer-Verlag, Berlin.

DOYLE, J., AND RIVEST, R. 1976. Linear expected time of a simple union-find algorithm. *Inf. Process. Lett. 5*, 146-148.

DRISCOLL, J. R., SARNAK, N., SLEATOR, D. D., AND TARJAN, R. E. 1989. Making data structures persistent. *J Comput. Syst. Sci. 38*, 86-124.

EPPSTEIN, D., GALIL, Z., GIANCARLO, R. AND ITALIANO, G F. 1990. Sparse dynamic programming. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms.* Society for Applied and Industrial Mathematics, Philadelphia, PA, pp. 513-522. To appear in *J. ACM.*

ERDÖS, P. AND RÉNYI, A. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci. bf 5*, 17-61.

EVEN, S. AND SHILOACH, Y. 1981. An on-line edge deletion problem. *J. ACM 28*, 1-4.

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.

FREDMAN, M. L. 1989. On the cell probe complexity of the set union problem. Tech. Memorandum TM-ARH-013-570, Bell Communications Research.

FREDMAN, M. L. AND SAKS, M. E. 1989. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing.* ACM Press, New York, NY, pp. 345-354.

FREDMAN, M. L., AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J ACM 34*, 596-615.

GABOW, H. N. 1985. A scaling algorithm for weighted matching on general graphs In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, 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. Syst. Sci. 30*, 209-221.

GAIBISSO, C., GAMBOSI, G., AND TALAMO, M. 1987 A partially persistent data structure for the set union problem. RAIRO Theoretical Informatics and Applications. *24*, pp. 189-202

GALIL, Z., ITALIANO, G. F. 1989. A note on set union with arbitrary deunions. *Inf Process. Lett. 37*, pp. 331-335.

GALLER, B. A., AND FISCHER, M. J. 1954. An improved equivalence algorithm *Commun. ACM 7*, 301-303.

GAMBOSI, G., ITALIANO, G. F., AND TALAMO, M. 1988. Getting back to the past in the union find problem. In *Proceedings of the 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. 1989. Worst-case analysis of the set union problem with extended backtracking. *Theor. Comput. Sci. 68* (1989), 57-70.

GAMBOSI, G., ITALIANO, G F., AND TALAMO, M. 1991. The set union problem with dynamic weighted backtracking. *BIT.* To be published

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

HOGGER, C. J. 1984. *Introduction to Logic Programming.* Academic Press.

HOPCROFT, J. E., AND KARP, R M. 1971. An algorithm for testing the equivalence of finite automata. Tech. Rep. TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

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

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

HUET, G. 1976. Resolutions d'equations dans les langages d'ordre 1, 2, . . . ω. Ph.D. dissertation, Univ. de Paris VII, France.

HUNT, J. W., AND SZYMANSKI, T. G. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM 20*, 350-353.

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

IMAI, T., AND ASANO, T. 1987. Dynamic segment intersection with applications. *J. Algorithms 8*, 1-18.

ITALIANO, G. F., AND SARNAK, N. 1991. Fully persistent data structures for disjoint set union problems. In *Proceedings Workshop on Algorithms and Data Structure* (Ottawa, Canada). To appear.

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

KERSCHENBAUM, A., AND VAN SLYKE, R. 1972. Computing minimum spanning trees efficiently. In *Proceedings of the 25th Annual Conference of the ACM*. ACM Press, New York, NY, pp. 518-527.

KNIGHT, K. 1989. Unification: a multidisciplinary survey *ACM Comput. Surv. 21*, 93-124.

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

KNUTH, D. E. 1976. Big omicron and big omega and big theta. *SIGACT News 8*, 18-24.

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

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

KRUSKAL, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc. 7*, 48-50.

LA POUTRÉ, J. A. 1990a. New techniques for the union-find problem. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 54-63.

LA POUTRÉ, J. A. 1990b. Lower bounds for the union-find and the split-find problem on pointer machines. In *Proceedings of the 22th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, pp. 34-44.

LOEBL, M., AND NEŠETŘIL, J. 1988. Linearity and unprovability of set union problem strategies. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, pp. 360-366.

MANNILA, H., AND UKKONEN, E. 1986a. The set union problem with backtracking. In *Proceedings of the 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. In *Proceedings of the 3rd International Conference on Logic Programming. Lecture Notes in Computer Science 225*. Springer-Verlag, Berlin, pp. 122-133.

MANNILA, H., AND UKKONEN, E. 1986c. Time-stamped term representation for implementing Prolog. In *Proceedings of the 3rd IEEE Conference on Logic Programming*. The MIT Press, Cambridge, MA, pp. 159-167.

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

MANNILA, H., AND UKKONEN, E. 1988. Time parameter and arbitrary deunions in the set union problem. *In Proceedings of the 1st Scandinavian Workshop on Algorithm Theory* (SWAT 88). *Lecture Notes in Computer Science 318*. Springer-Verlag, Berlin, pp. 34-42.

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. 1990. Dynamic fractional cascading. *Algorithmica 5*, 215-241.

MEHLHORN, K., NÄHER, S., AND ALT, H. 1988. A lower bound for the complexity of the union-split-find problem. *SIAM J. Comput. 17*, 1093-1102.

NIEVERGELT, J., AND REINGOLD, E. M. 1973. Binary search trees of bounded balance. *SIAM J. Comput. 2*, 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*. 490-508.

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

STEARNS, R. E., AND ROSENKRANTZ, P. M. 1969. Table machine simulation. *Conference Records IEEE 10th Annual Symposium on Switching* *and Automata Theory*. IEEE Computer Society, pp. 118–128.

TARJAN, R. E. 1973. Testing flow graph reproducibility In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, pp. 96–107.

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

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

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

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

TARJAN, R. E. 1982. Sensitivity analysis of minimum spanning trees and shortest path trees. *Inf. Process. Lett. 14*, 30–33

TARJAN, R. E. 1983. *Data structures and network algorithms*. SIAM, Philadelphia, Penn.

TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods 6*, 306–318.

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

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.

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

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

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

VITTER, J S., AND SIMONS, R. A. 1986. New classes for parallel complexity: A study of unification and other complete problems for P *IEEE Trans. Comput. C-35*.

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

WESTBROOK, J. 1989. Algorithms and data structures for dynamic graphs problems Ph.D. dissertation. Also Tech. Rep. CS-TR-229-89, Dept. of Computer Science, Princeton University, Princeton, N.J

WESTBROOK, J., AND TARJAN, R. E. 1989a. Amortized analysis of algorithms for set union with backtracking. *SIAM J Comput. 18*, 1–11.

WESTBROOK, J., AND TARJAN, R. E. 1989b. Maintaining bridge-connected and biconnected components on-line. Tech. Rep. CS-TR-228-89, Dept. of Computer Science, Princeton University, Princeton, N.J.

YAO, A C. 1976. On the average behavior of set merging algorithms. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, pp 192–195.

YAO, A. C. 1981. Should tables be sorted? *J. ACM 28*, 615–628.

Received November 1988; final revision accepted September 1990