BACKTRACKING

An *equivalence relation* on a finite set S is a binary relation that is reflexive symmetrically and transitively. That is, for s, t, and u in S, we have that sRs, if sRt, then tRs; and if sRt and tRu, then sRu. Set S is partitioned by R into *equivalence classes,* where each class contains all and only the elements that obey R pairwise.

Many computational problems involve representing, modifying, and tracking the evolution of equivalence classes of an equivalence relation that varies with time, starting from a given initial configuration. Such an initial configuration is often the one where equivalence coincides with equality; i.e., each element of S forms a *singleton* class all by itself. The study of these equivalence maintenance programs was motivated originally by the problem of processing some declarations of the FORTRAN language, such as EQUIVALENCE and COMMON. The EQUIVALENCE(A,B,C(3)) declaration, for instance, indicates that the variables A, B, and C(3) (the third element of array C) are to share the same location in memory. In general, this poses no special problem except for the fact that individual arrays have to occupy consecutive locations in memory, a circumstance that may create havoc as a result of careless declarations. For instance, a declaration such as EQUIVALENCE((A(1),B(1)), (A(2),B(3)) violates this condition on array B and is thus unacceptable.

For further illustration, consider the problem of finding a minimum spanning tree in a connected weighted graph $G = (V, E, W)$ having vertices in V, edges in E, and edge weights in W. A minimum spanning tree for G is a subgraph $T = (V, E')$ connecting all vertices of G by precisely $(|V| - 1)$ edges, in such a way that the edges in E' do not form any cycles, and the sum of the weights in W' is minimal with respect to all possible selections for edges in E'. One proven method to compute T (1,2) is as follows. First, sort the edges in E in order of increasing weight and put each vertex into a separate, singleton *connected component.* Consider now the edges of E in succession, the lightest first. In correspondence with edge (v, w), do the following:

> Find the component currently containing v, and let this be A.
>
> Find the component currently containing w, let it be B.
>
> If now A and B are different components, then combine them into a single new component and add edge (v, w) to T.

The examples considered pose instances of the set union problem, the general formulation of which is to maintain a collection of disjoint sets under an intermixed sequence of the following two kinds of operations:

> *union(A, B).* Combine the two sets A and B into a new set named A.
>
> *find(x).* Return the name of the set containing element x.

The operations are presented on line, namely, each operation must be processed before the next one is known. Initially, the collection consists of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$, and the name of set $\{i\}$ is i, $1 \le i \le n$. Figure 1 illustrates an example of set union operations.

The set union problem has many applications in a very wide range of areas besides those already mentioned of COMMON and EQUIVALENCE statements in FORTRAN compilers (3,4) and minimum spanning trees (1,2). A list, by no means exhaustive, would include implementing property grammars (5,6), computational geometry problems (7–9), testing equivalence of finite state machines (1,10), string algorithms (11,12), logic programming and theorem proving (13–16), and several combinatorial problems such as solving dynamic edge- and vertex-connectivity problems (17), computing least common ancestors in trees (18), solving off-line minimum problems (19,20), finding dominators in graphs (21), and checking flow graph reducibility (22).

The focus of the present discussion is represented by the several variants of set union, where the possibility of backtracking over the sequences of unions is taken into account (23–27). These variants are motivated especially by problems

{1} {2} {3} {4} {5} {6} {7}

(a)

{1, 3} {5, 2} {4} {6} {7}

(b)

{4, 1, 3, 7} {5, 2} {6}

(c)

{4, 1, 3, 7, 5, 2} {6}

(d)

Figure 1. Examples of set union operations: (a) the initial collection of disjoint sets; (b) the disjoint sets of (a) after performing "union(1, 3)" and "union(5, 2)"; (c) the disjoint sets of (b) after performing "union(1, 7)" followed by "union(4, 1)"; (d) the disjoint sets of (c) after performing "union(4, 5)."

arising in logic programming interpreter memory management (28–31), in incremental execution of logic programs (26), and in implementation of search heuristics for resolution (32,33). Special cases of backtracking through a special primitive "split" are found also in connection with some of the geometric and string matching problems cited earlier. For the sake of self-containment, our exposition must start with a brief account of the classical set union problem. We undertake this task soon after the outline of computational models, which is given next.

MODELS OF COMPUTATION

Different models of computation have been developed for analyzing data structures. One model of computation is the *random-access machine,* in which memory consists of an unbounded sequence of registers, each capable of holding one integer. In this model, the address of a memory register is provided directly, or it may be obtained as the result of some arithmetic operations. It is usually assumed that the size of a register is $O(\log n)$ bits in terms of the size n of the input. (In accordance with standard notation, $f = O[g(n)]$ is used to indicate the existence of a constant c and of a positive integer n_0 such that $f(n) \leq g(n)$ for $n \geq n_0$. Also, "log" denotes the logarithm to the base 2.) A more detailed description of random-access machines can be found in (1). Another model of computation, known as the *cell probe model of computation,* was introduced by Yao (34). In the cell probe, the cost of a computation is measured by the total number of memory accesses to a random-access memory with cell size of $\lceil \log n \rceil$ bits. All other computations are assumed to be performed for free and thus are not accounted for. The cell probe model is more general than the random-access machine, which makes it sometimes more convenient in attempts at establishing lower bounds. A third model of computation is the *pointer machine* (35–39). Its storage consists of an unbounded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information, but no calculations are allowed to compute the address of a register. Thus, the only way to access a register is by following pointers. This is the main difference between random-access machines and pointer machines. Throughout the discussion, we use the terms random-access algorithms, cell-probe algorithms, and pointer-based algorithms to refer to algorithms tailored to random-access machines, the cell probe model, and pointer machines, respectively.

Among pointer-based algorithms, two different classes are defined specifically for set union problems: separable pointer algorithms (39) and nonseparable pointer algorithms (40).

Separable pointer algorithms run on a pointer machine and satisfy the separability assumption, introduced in (39) and recalled later in this article. A separable pointer algorithm makes use of a linked data structure, i.e., 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 in the graph. The algorithm solves the set union problem according to the following rules (39,41).

1. The operations must be performed on line, i.e., each operation must be executed before the next one is known.
2. Each element of each set is a node of the data structure. There can be additional (working) nodes.
3. (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.
4. To perform "find(x)," the algorithm obtains the node v corresponding to element x and follows the paths, starting from v until reaching the node that contains the name of the corresponding set.
5. During any operation, the algorithm may insert or delete any number of edges. The only restriction is that rule 3 must hold after each operation.

The class of nonseparable pointer algorithms (40) does not require the separability assumption. The only requirement is that the number of edges leaving each node must be bounded by some constant $c > 0$. Formally, all rules except rule 3 are left unchanged, while rule 3 is reformulated as follows:

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

As we shall see in the course of our discussion, separable and nonseparable pointer-based algorithms often admit quite different upper and lower bounds for the same problems.

THE SET UNION PROBLEM

As stated earlier, the set union problem consists of performing a sequence of "union" and "find" operations, starting from a collection of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. Since there are at most n items to be united, the number of unions in any sequence of operations is bounded above by $(n - 1)$. Throughout, the following invariant conditions are preserved; first, the sets are always disjoint and define a partition of $\{1, 2, \ldots, n\}$; second, each set is named after a representative chosen among its own elements. Thus, for example, the initial name of set $\{i\}$ is i. It is easily seen that the maintenance of these invariants does not pose implementation problems. In fact, sets are typically implemented as rooted trees, following a representation introduced by Galler and Fischer (4). A separate tree is assigned to each disjoint set, with each node of that tree corresponding to a distinct element in the corresponding set. The element stored at the root of the tree also serves as the name of the set. Each node has a pointer to its parent. In the following, we use $p(x)$ to refer to the parent of node x.

A notable variant of the problem results from the following modification of "union":

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

The only difference between "union" and "unite" is that "unite" allows the name of the new set to be chosen arbitrarily (e.g., at run time by the algorithm). In most applications, this does not pose a restriction, since one is only interested in testing whether two elements belong to the same set, not to how names are given. However, some extensions of the set union problem have quite different time bounds depending on whether "unions" or "unites" are considered.

Throughout our discussion, we will deal with "unions" unless specified otherwise.

The best classical algorithms for the set union problem (42,43) sought to optimize their amortized time complexity, i.e., the running time per operation as averaged over a worst-case sequence (see 44) for a thorough treatment). Before describing them, it is instructive to review some of the basic approaches to the problem (1,4,45). These are the quick-find, the weighted quick-find, the quick-union, and the weighted quick-union, algorithms. As the names suggest, the quick-find algorithm performs "find" operations quickly, while the quick-union algorithm performs "union" operations quickly. Their weighted counterparts speed up these computations by introducing some weighting rules during "unions" operations.

The quick-find algorithm is performed 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 on 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. Because a set can have as many as n elements, this gives a time complexity proportional to n in the worst case for each union. To perform a "find(x)," return the name stored in the parent of x. Because all trees are maintained at height 1, the parent of x is a tree root. Consequently, a "find" requires $O(1)$ time.

The more efficient variant known as weighted quick-find, and attributed to McIlroy and Morris [see (1)], makes better use of the degrees of freedom inherent in the implementation of "union": the latter is now executed taking weights into consideration, as follows.

union by size. Make the children of the root of the smaller tree point to the root of the larger one, arbitrarily breaking a tie.

This rule adds the (easy) requirement that notion of the size of each tree be maintained throughout in any sequence of operations. Following the rule does not lead to an improved worst-case time complexity for individual operations. However, it yields an $O(\log n)$ amortized bound for a "union" (see, e.g., 1).

Each set is also represented by a tree in the quick-union algorithm (4). However, there are two main differences with respect to the data structure used by the quick-find algorithm. First, the height of a tree can now be greater than 1. Second, the representative of each set is stored only at the root of the corresponding tree, whence the notion of a special node is forfeited. A "union(A, B)" is performed by making the root of the tree representing set B a child of the tree root of set A. A "find(x)" is performed starting from the node x 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. As a result, the quick-union algorithm supports "union" in $O(1)$ time and "find" in $O(n)$ time.

This time bound can also be improved by exploiting the freedom in our tree implementations to choose which one of the two sets gets to name the new representative. More specifically, two weighted quick-union algorithms follow immediately from adoption of one of the following rules.

union by size. Make the root of the smaller tree point to the root of the larger one, breaking ties arbitrarily.

union by rank (43). Make the root of the shorter tree point to the root of the taller one, breaking ties arbitrarily.

These rules introduce some extra bookkeeping. In fact, the first rule requires maintaining, for each node in the forest, the number of its descendants, referred to as the *size* of that node. The second requires maintaining, for each node, its *rank,* defined as the height of the subtree rooted at that node. After a "union (A, B)," the name of the new tree root is set to A. It can be easily proved [see, e.g., (43)] that the height of the trees achieved with either the "union by size" or the "union by rank" rule is never more than $\log n$. Thus, with either rule, each "union" completes 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 nodes encountered on the path traversed in the course of each "find" (see Figure 2).

path compression (20). Make every encountered node a child of the root of the tree.

path splitting (46,47). Make every encountered node (except the last and the next to last) point to its grandparent.

path halving (46,47). 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. As shown in (43), they all have an $O[\alpha(m + n, n)]$

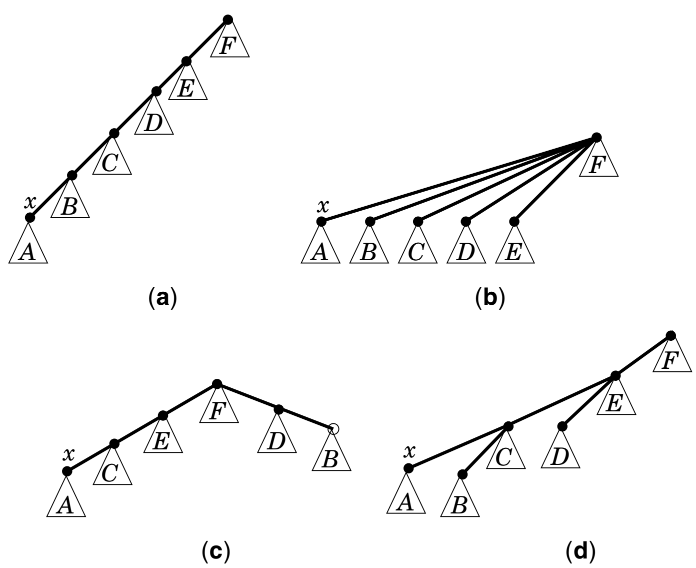

Figure 2. An illustration of path compaction techniques: (a) the tree before performing a "find(x)" operation; (b) path compression; (c) path splitting; (d) path halving.

amortized time complexity, where α is a very slowly growing function, a functional inverse of Ackermann's function (48).

Theorem 1. (43) A sequence of at most $(n - 1)$ "unions" and m "finds" takes $O[n + m\alpha(m + n, n)]$ time by any fixed combination of union by size or rank with path compression, splitting, or halving.

No better amortized bound is possible for separable and nonseparable pointer algorithms or in the cell probe model of computation. Formally, with $g = \Omega(f)$ used to signify that $f = O(g)$, we record the following theorem.

Theorem 2. (43,49,50) Any pointer-based or cell probe algorithm requires $\Omega[n + m\alpha(m + n, n)]$ worst-case time for processing a sequence of $(n - 1)$ "unions" and m "finds."

The bound of Theorem 2 does not rule out that a better bound is possible for a special case of set union. In fact, Gabow and Tarjan (19) proposed a random-access algorithm that runs in linear time in the special case where the structure of the "union" operations is known in advance. Interestingly, Tarjan's lower bound for separable pointer algorithms applies also to this special case, and thus the power of a random-access machine seems crucial in achieving a linear-time algorithm. This result is of theoretical interest as well as significant to many applications, such as scheduling problems, the off-line minimum problem, finding maximum matching on graphs, very large scale integration (VLSI) channel routing, finding nearest common ancestors in trees, and flow graph reducibility (19).

One more special case of the set union problem where amortized linear time suffices was studied by Loebl and Nešetřil (51), and it involves a restriction on the subsequence of "finds."

THE WORST-CASE TIME COMPLEXITY OF A SINGLE OPERATION

The algorithms that use any union and any compaction rule still have single-operation worst-case time complexity proportional to log n (43), since such may be the height of some of the trees created by any of the union rules. Set union algorithms where some form of backtracking is possible are analyzed in terms of single-operation performance, rather than amortization. The complexity achievable by a single "union" or "find" in a sequence of such operations is also a topic of intrinsic interest, and we discuss it in some detail in this section.

Blum (41) 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, and showed that this $\log n/\log\log n$ is the actual lower bound for separable pointer-based algorithms.

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

1. The root has at least two children.
2. Each internal node has at least k children.
3. All the leaves are at the same level.

As a consequence of this definition, the height of a k-UF tree with n leaves is at most $\lceil \log_k n \rceil$. We refer to the root of a k-UF tree as *fat* if it has more than k children, and as *slim* otherwise. In addition, 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, along the lines already described in the previous section, by starting from the leaf containing x and returning the name stored in the root. This requires time at most proportional to $\log_k n$.

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 combining T_A and T_B, 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 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, i.e., either "root r_B is slim" or "$v = r_A$ is slim." If root r_B is slim, then all the children of r_B are made the rightmost children of v, and r_B is deleted. Otherwise, all the children of the slim node $v = r_A$ are made the rightmost children of r_B, and r_A is deleted.

Note that type 1 and type 2 unions *create* new pointers, while type 3 unions only *redirect* already existing pointers.

Theorem 3. (41) k-UF trees support each "union" and "find" in $O(\log n/\log\log n)$ time. Their space complexity is $O(n)$.

Proof: Each "find" can be performed in $O(\log_k n)$ time. Each "union(A, B)" takes $O(\log_k n)$ time to locate the nodes r_A, r_B, and v defined earlier. Both type 1 and type 2 unions can be performed in constant time, while type 3 unions require $O(k)$ 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. Thus, the forest of k-UF trees requires $O(n)$ space in total to store all the disjoint sets.

Blum showed also that this bound is tight for the class of separable pointer algorithms. Fredman and Saks (49) showed that the same lower bound holds in the cell probe model of computation.

Theorem 4. (41,49) Every separable pointer or cell probe algorithm for the disjoint set union problem has single-operation worst-case time complexity $\Omega(\log n/\log\log n)$.

THE SET UNION PROBLEM WITH DEUNIONS

In this section, we undertake discussion of those variants of the set union problem where it is possible to undo one or more of the unions performed in the past. This feature comes in several forms, and is generally referred to as *backtracking.* One of its main applications is found in logic programming interpreter memory management without function symbols (29), because a popular logic programming language, Prolog, uses unification and backtracking as crucial operations (31). We illustrate this with the help of the following example, and refer the interested reader to (52) for further details.

Consider a database consisting of the following four assertions:

```
likes(alice,running)
likes(alice,snorkeling)
likes(bob,snorkeling)
likes(bob,alice)
```

which stand to represent the facts that Alice likes running, that Alice and Bob like snorkeling, and that Bob likes Alice. The question "Is there anything that Bob and Alice both like?" is phrased in Prolog as follows:

```
?- likes(alice, X), likes(bob,X).
```

Prolog reacts to this question by attempting to unify the first term of the query with some assertion in the database. The first matching fact found in our case is `likes(alice,running)`. As a result, the terms `likes(alice, running)` and `likes(alice,X)` are unified, and Prolog instantiates `X` to `running` everywhere `X` appears in the query. The database is then searched for the second term in the query, which is now `likes(bob,running)` because of the previous substitution. However, this term fails to unify with any other term in the database.

Then Prolog backtracks, i.e., it "undoes" the last unification performed: it undoes the unification of `likes(alice, running)` with `likes(alice,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(alice,snorkeling)`, and therefore the variable `X` is instantiated to `snorkeling` everywhere `X` appears. As before, Prolog now tries to unify the second term, searching this time for `likes(bob,snorkeling)`. This can be unified with the third assertion in the database, whence Prolog notifies the user by answering:

```
X=snorkeling.
```

In summary, the execution of a Prolog program without function symbols can be regarded as a sequence of unifications and deunifications. This class of problems was modeled by Mannila and Ukkonen (25) as a variant of the set union problem, which they called set union with deunions, characterized by the fact that the following operation is added to the standard ones of "union" and "find."

deunion. Undo the union performed most recently and not yet undone.

The set union problem with deunions can be solved by a modification of Blum's data structure described earlier. To facilitate deunions, we maintain a union stack that stores some auxiliary information related to bookkeeping of unions. "Finds" are performed as described earlier. Unions require some additoinal work to maintain the union stack. We now sketch which information is stored in the union stack. For the sake of simplicity, we do not take into account names of the sets, so that ours will be a description of "unite" rather than "union." However, names are easily maintained in some extra fields stored in the union stack. Initially, the union stack is empty. When a type 1 union is performed, we proceed as described earlier, and then push onto the union stack a record containing a reference to the old root r_B. Similarly, when a type 2 union is performed, we push onto the union stack a record containing a reference to r_A and a reference to r_B. Finally, when a type 3 union is performed, we push onto the union stack a reference to the leftmost child of either r_B or r_A, depending on the two cases. The pointer leaving this leftmost child is called a *separator,* as it separates the newly moved pointers from the rest of the pointers entering the same node.

Deunions basically use the top stack record to invalidate the last union performed. Indeed, we pop the top record from the union stack, and check whether the union to be undone is of type 1, 2, or 3. For type 1 unions, we use the reference to r_B to delete the pointer leaving this node, thus restoring it as a root. For type 2 unions, we follow the references to r_A and r_B and delete the pointers leaving these nodes and their parent. For type 3 unions, we follow the reference to the node, and move this node together with all its right siblings as a child of a new root. Note that this corresponds to redirecting the associated separator together with all the pointers to its right.

It can be easily shown that this augmented version of Blum's data structure supports each "union," "find," and "deunion" in $O(\log n/\log\log n)$ time and space $O(n)$. This was proved to be a lower bound for separable pointer algorithms by Westbrook and Tarjan:

Theorem 5. (27) Every separable pointer algorithm for the set union problem with deunions requires $\Omega(\log n/\log\log n)$ amortized time per operation.

The union stack bookkeeping just described can be applied to all of the union rules and path compaction techniques described earlier, thereby accommodating deunions in those contexts. However, path compression with any one of the union rules leads to an amortized algorithm only bounded by $O(\log n)$, as can be seen by first building a binomial tree [refer, e.g., to (43)] of depth $O(\log n)$ with $(n - 1)$ unions, and then by carrying out repeatedly a sequence consisting of a "find" on the deepest leaf, a "deunion," and a redo of that union. Westbrook and Tarjan (27) showed that using either one of the union rules combined with path splitting or path halving would result in $O(\log n/\log \log n)$ amortized algorithms for the set union problem with deunions. We now describe their algorithms.

Let a union operation not yet undone be referred to as *live,* the union being *dead* otherwise. Again, deunions make use of a union stack, in which those roots that lost their status as a consequence of some live unions are maintained. In addition, we maintain for each node x a node stack $P(x)$, which contains the pointers originating from x as the result of either "unions" or "finds." During the path compaction accompanying a "find," the pointer from x now being disrupted is kept in $P(x)$, and the newly created pointer is pushed on top of it. Clearly, the pointer at the bottom of any of these stacks is always created by a union, and is thus called a *union pointer.* The other pointers are created by the path compaction performed during subsequent "finds" and are called *find pointers.* Each of these pointers is associated with a unique union operation, namely, the one undoing that would invalidate the pointer. A pointer is said to be live if its associated union operation is live, 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, which contains the old tree root and some auxiliary 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 whether or not 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 only while performing subsequent "union" and "find" operations. Combined with the applicable union and compaction rules, this gives a total of eight algorithms. They all have the same time and space complexity, as the following theorem claims.

Theorem 6. (27) An eager or lazy algorithm based on any fixed combination of union by size or rank with either path splitting or path halving runs in amortized time $O(\log n/\log \log n)$ per operation and overall linear space.

SPLIT AND THE SET UNION PROBLEM ON INTERVALS

In some applications, the individual sets constituting our partition may be subjected to disaggregations that do not necessarily correspond to undoing some previous union. In other words, these applications encompass our notion of backtracking, but do reduce to backtracking. In particular, the role of deunion is now taken by a new primitive split. One notable instance of these problems is represented by the set union problem on intervals, which consists of maintaining a partition of a list $\{1, 2, \ldots, n\}$ into adjacent, consecutive intervals, each interval a sublist of the form $\{i, i + 1, \ldots, i + d\}$. Union is now defined only on adjacent intervals. Formally, letting S_i $(1 \leq i \leq k)$ be the ordered list of intervals in the partition, the problem consists of performing a sequence of operations, each chosen arbitrarily from the following repertoire.

union(S_1, S_2, S). Combine the adjacent sets S_1 and S_2 into the new set $S = S_1 \cup S_2$.

find(x). Return the name of the set containing x

split(S, S_1, S_2, x). Cleave S in correspondence with element x so as to produce the two sets $S_1 = \{a \in S | a < x\}$ and $S_2 = \{a \in S | a \geq x\}$.

This interval union-split-find problem (40) and its restrictions find applications in a wide range of areas, including problems in computational geometry such as dynamic segment intersection (7–9), shortest-paths problems (53,54), and the longest common subsequence problem (11,12). The latter arises in many applications, including sequence comparison in molecular biology and the widely used diff file comparison program (11), and we shall discuss it briefly. The problem can be defined as follows. Let x be a string of symbols over some alphabet. A subsequence of x is any string w obtained by removing one or more, not necessarily consecutive, symbols from x. The longest common subsequence problem for input strings x and y consists of finding a string w that is a subsequence of both x and y and that has maximum possible length.

The problem can be formulated in terms of union-split-find (11), and then solved according to a paradigm due to Hunt and Szymanski (12). For simplicity, we describe only how to find the length of a longest common subsequence, and leave the computation of the subsequence itself for an exercise. Let, then, $x = x_1, x_2, \ldots, x_m$ and $y = y_1, y_2, \ldots, y_n$ be the two input strings, and assume without loss of generality $m < n$. For each symbol a in the input alphabet, compute OCCURRENCES(a) $= \{i | y_i = a\}$, i.e., the ordered list of positions in y occupied by an a. The algorithm then performs m successive main stages, each stage being associated with a symbol of x, as follows: Stage j $(1 \leq j \leq m)$ consists of computing in succession the length of a longest subsequence between prefix $x_1, x_2, \ldots, x_j$ of x and the consecutive prefixes $y_1, y_2, \ldots, y_i$ of y. For $k = 1, 2, l_j$, let A_k be the interval of positions of y that yield a longest common subsequence with $x_1, x_2, \ldots, x_j$ of length k. Observe that the sets A_k partition $\{1, 2, \ldots, n\}$ into adjacent intervals, where each A_k contains consecutive integers, and the entries of A_{k+1} are larger than those in A_k, for any k. Assume that we had already computed the sets A_k relative to some position $(j - 1)$ of the string x. Now we show how to update those intervals so that they apply to 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 $x_1, x_2, \ldots, x_j$ 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 indices $s \geq r$ belong to A_{k+1} when x_j is considered. The pseudocode in Fig. 3 describes this algorithm. The reader is referred to (11,12) for details of the method and to (55,56) for upgrades and additional references.

The time complexity of this algorithm is proportional to the number p of pairs of matching symbols that can be formed between x and y, multiplied by the cost of each individual primitive set operation performed. We summarize next what is known about such a cost.

There are optimal separable and nonseparable pointer algorithms for the interval union-split-find problem. The best separable algorithm for this problem runs in $O(\log n)$ time for each operation, while non-separable pointer algorithms re-

```
begin
    initialize A_0 = {0, 1, . . ., n};
    for i := 1 to n do
          A_i := ∅;
    for j := 1 to n do
          for r ∈ OCCURRENCES(x_j) do begin
                k := 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
```

Figure 3. Finding the longest common subsequence.

quire only $O(\log \log n)$ time for each operation. In both cases, no better bound is possible.

For separable pointer algorithms, the upper bound descends from balanced tree implementation (1,15), while the lower bound was proved by Mehlhorn et al. (40).

Theorem 7. (40) 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 non-separable pointer algorithms, the upper bound can be found in (32,9,57,60). In particular, van Emde Boas et al. (60) introduced a priority queue which supports among other operations *insert, delete* and *successor* on a set with elements belonging to a fixed universe $S = \{1, 2, \ldots, n\}$. The time required by each of those operations is $O(\log \log n)$. Originally, the space was $O(n \log \log n)$ but later it was improved to $O(n)$. It is easy to show [see also (40)] that the above operations correspond respectively to union, split, and find, and therefore the following theorem holds.

Theorem 8. (57) Each "union," "find," and "split" can be implemented in $O(\log \log n)$ worst-case time. The space required is $O(n)$.

We observe that the algorithm based on van Emde Boas' priority queue is inherently nonseparable. Mehlhorn et al. (40) proved that this is indeed the best possible bound that can be achieved by a nonseparable pointer algorithm:

Theorem 9. (40) 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 Theorems 7 and 8 imply that for the interval union-split-find problem, the separability assumption causes an exponential loss of efficiency.

As mentioned, special cases of union-split-find also have been considered: the interval union-find problem and the interval split-find problem, respectively allowing union-find and split-find operations only. Most corresponding bounds can be derived from our discussion and are left for an exercise. The interested reader may also refer to (20,58,59), among other references, for details.

THE SET UNION PROBLEM WITH UNLIMITED BACKTRACKING

Other variants of the set union problem with deunions have been considered, including set union with arbitrary deunions (26,60), set union with dynamic weighted backtracking (24), and set union with unlimited backtracking (23). Here we will discuss only set union with unlimited backtracking and refer the interested reader to the literature for the other problems.

As before, we classify a union as live if not yet undone, and dead otherwise. In the set union problem with unlimited backtracking, deunions are replaced by the following, more general, operation with the parameter a nonnegative integer i:

backtrack(i). Undo the last i live unions performed.

The name of this problem derives from the fact that the limitation that at most one union could be undone per operation is removed. Note that this problem is more general than the set union problem with deunions, because a deunion can be simply implemented as backtrack(1). Furthermore, the effect of a backtrack(i) may be achieved 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 either Westbrook and Tarjan's algorithms or Blum's modified algorithm 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. Because deunions are a special case of backtracks, this bound is tight for the class of separable pointer algorithms in force in Theorem 5.

However, 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. Indeed, 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 earlier), because in this case at least one pointer must be removed for each erased union. This is clearly undesirable, because i can be as large as $(n - 1)$. To avoid this lower bound, the only possibility is to defer the removal of pointers invalidated by backtracking to some possible future operation, in a lazy fashion. In a strict sense, this lazy approach infringes on the separability condition stated earlier. However, the substance of that condition would still be met if one maintains that a pointer is never followed once it is invalidated [see, e.g., (27)].

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

Theorem 10. (60) 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.

Apostolico et al. (23) showed that, when "unites" instead of "unions" are performed (i.e., when the name of the new set can be arbitrarily chosen by the algorithm), a better bound for separable pointer algorithms can be achieved. In the following, we present the data structure by Apostolico et al. (23). This data structure is called *k-BUF* tree or, with the implicit assumption that $k = \lceil \log n/\log \log n \rceil$, simply BUF tree. BUF trees support "union" and "find" in $O(\log n/\log \log n)$ time and backtrack(i) in constant time, independent of i.

We now describe the main features of BUF trees, and will highlight the implementation of union, find, and backtrack operatons. BUF trees retain the basic structure of the k-UF trees described in earlier sections, but differ from them primarily because of some implicit attributes defined on the pointers. With BUF trees, three are still three different types of unions, as with k-UF trees. In particular, we will have that type 1 and type 2 unions create new pointers, while type 3 unions only redirect already existing pointers. With BUF trees, however, a union must perform some additional operations on pointers. In the following, we say that a pointer e is handled by a certain union only if e is either created or redirected by that union during the aggregation stage of that union. It was stated earlier that a separator is the leftmost pointer redirected by a type 3 union. The main difference with k-UF trees is that now, due to the lazy approach, we allow pointers and separators to possibly survive in the data structure also, after the union that introduced therm has been invalidated by backtracking. At any given time, we call a union *valid* if it has not yet been undone by backtracks, and *void* otherwise. We further partition void unions as follows: A void union is *persisting* if the pointers handled by that union have not yet been actually removed from the data structure, and is *dissolved* otherwise. This classification of unions imposes a corresponding taxonomy on pointers and separators, as follows: In a BUF tree, an ordinary pointer can be live, dead, or cheating, and a separator pointer can be, in addition, either active or inactive. Informally, live pointers represent connections not yet invalidated by backtracks; this happens when the last union that handled them is still valid. Dead pointers represent, instead, connections that, although still in the structure, are only waiting to be destroyed; this happens when the first union that created them is a void persisting union. Between live and dead pointers lie cheating pointers. They occur when the first union that created them is valid, but the last union that handled them is a persisting type 3 union. Therefore, they represent faulty connections that do not have to be destroyed, but only replaced by the corresponding correct connections. As in k-UF trees, separators are associated with type 3 unions. At any given time, a separator is active if its associated union is valid, and inactive otherwise. A node of a BUF tree is live if there is at least one live pointer entering it, and is persisting otherwise. In analogy with the nodes of k-UF trees, the live nodes of BUF trees can be slim or fat, but this is decided based only on the number of live pointers entering each node. Specifically, a node is slim if the number of live pointers entering it is less than k, and fat if the number of live pointers entering it is at least k.

Assume that we perform an intermixed sequence σ of union, find, and backtrack operations starting from the initial partition of S into n singletons. The partition of S that results from σ is the same as that produced by applying to S, in the same order as in σ, only those unions that are valid (i.e., not undone by backtracks) at the completion of σ. The subsequence of σ consisting only of unions that are still valid by the end of σ (i.e., by neglecting the unions made void by backtracking) is called the virtual sequence of unions. The following rules ensure that at any time each currently valid union u is assigned a unique integer ord(u), representing the ordinal number of u in the current virtual sequence of unions:

1. The first union performed gets ordinal number 1.
2. When a union is made void by backtracking, it relinquishes its ordinal number.
3. A new union gets ordinal number equal to 1 plus the ordinal number of the last valid union performed.

At some point of the execution of σ, let $i_{\max}$ be the ordinal number of the last valid union performed so far. "Backtrack (i)" consists of removing the effect of the last i valid unions, that is, the effect of the last i unions in the current virtual sequence of unions. We perform "backtrack(i)" simply by setting $i_{\max} = \max\{i_{\max} - i, 0\}$, i.e., in constant time irrespective of i. Note that this implementation of backtrack does not affect any pointer in the forest, but its effect is implicitly recorded in the change of status of some pointers and separators. Part or all of these pointers might be removed or redirected later, while performing subsequent union operations.

To perform a "find(x)" correctly, we need to ensure the consistency of the forest of BUF trees. By the forest being consistent, we mean that each tree in the forest stores a collection of sets in the current partition in such a way that, for any x, a "find(x)" executed as specified in the following correctly returns the name of the set currently containing x. We refer to the consistency of the forest as find consistency, which we will maintain as invariant throughout the sequence of operations. The complete specification of this invariant requires some additional notions.

First, each pointer e in a BUF tree T has two unions associated with it. The first union, denoted first_union(e), is the union that created e. The second union, last_union(e), is the last union not yet actually undone (i.e., either a valid or a persisting union) that handled e. We will maintain that ord(first_union(e)) $\leq$ ord(last_union(e)) for every pointer e. In a consistent BUF tree, a pointer e is dead if and only if "first_union(e)" is void (i.e., e has to be destroyed because it gives a connection made void by some intervening backtrack). Similarly, pointer e is cheating if and only if "first_union(e)" is valid and "last_union(e)" is void (i.e., e gives a faulty connection, and hence it has to be replaced, but it is not completely destroyed). Finally, e is live (i.e., it gives a connection not yet affected by backtracking) if and only if "last_union(e)" is still valid. In addition to "first_union" and to "last_union," each separator s also has associated the type 3 union that made it a separator. In the following, such a union will be referred to as separate_union(s). A separator s is active if and only if "separate_union(s)" is valid, inactive otherwise.

To complete our description of a consistent BUF tree T, let $S_1, S_2, \ldots, S_p$ be the disjoint sets stored in T. We specify the mapping from the set of leaves of T to the set of names of $S_1, S_2, \ldots, S_p$. 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. Ascend from x toward the root of T following live pointers until a node without an outgoing live pointer is met. Call this node *apex(x)*. In a consistent BUF tree, an apex falls always in one of the following three classes.

Live apex. There is no pointer leaving "apex(x)," i.e., "apex (x)" is the root r of T. We will maintain that the name Y of S_q is stored in r.

Dead apex. The pointer leaving "apex(x)" is dead. We will maintain that the name of S_q is stored in "apex(x)."

Cheating apex. The pointer e leaving "apex(x)" is cheating. In this case, we will maintain that at least one inactive separator falls within $(k - 1)$ pointers to the left of e, and the name of S_q is stored in the rightmost such separator.

These descriptions explain how a "find" is performed on a BUF tree. Throughout the sequence of union, find, and backtrack operations we need to maintain the forest of BUF trees in such a way that any arbitrary "find" would give a consistent answer. We now formalize this invariant:

(Find consistency). Prior to the execution of each operation, and for every element x of S, the following holds. IF "apex(x)" is either dead or live, then the name of the set containing x is stored in "apex(x)." If "apex(x)" is cheating, then the name of the set containing x is stored in the rightmost inactive separator to the left of "apex (x)," and such a separator falls within $(k - 1)$ pointers to the left of "apex(x)."

An immediate consequence of "find consistency" is that BUF trees support each "find" operation in time $O[(k + h)t]$, where t is the time needed to test the status of a pointer and h is the maximum length of a path from a leaf x to its apex in the tree. In (23), Apostolico et al. showed that it is possible to implement BUF trees in such a way that t is $O(1)$ and h is $O(\log_k n)$. This immediately yields the claimed $O(\log n/\log\log n)$ time bound for each "find."

Two additional invariants are maintained throughout the sequence of operations:

(Slim compression). The live pointers entering any slim node are leftmost among their siblings, and have nondecreasing last fields, from left to right. For fat nodes, this property holds for all the pointers that were directed to that node while the node was slim, including the pointers that made the node fat.

(Numbering). For any integer i, $1 \le i \le (n - 1)$, there are either at most two sibling pointers with first field equal to i or at most one pointer with separate field equal to i. Moreover, there are at most $(k - 1)$ sibling pointers with last field equal to i.

We now examine what is involved in performing union operations. Let A and B be two different subsets of the partition of S, such that $A \neq B$. In the collection of BUF trees that represents this partition, let T_1 and T_2 be the trees storing, respectively, A and B. We remark that two disjoint sets can happen to be stored in the same tree, so that T_1 and T_2 may coincide even if $A \neq B$. The first task of "union(A, B)" consists of finding in T_1 and T_2 the roots of the smallest subtrees that store, respectively, A and B. These roots are located by performing two "finds." The associated subtrees have to be detached from their host trees and then combined into a single tree. Once the two subtrees have been located and detached, their unification requires a treatment quite similar to that of the union procedure described earlier for k-UF trees. The most delicate part of the process, however, is in the first stage. The correctness of the two initial "finds" depends on our ability to preserve "find consistency" through each "union," "find," and "backtrack."

We now describe how to perform "unions." In terms of BUF trees, a "union(A, B)" transforms the current input forest F of BUF trees into a new forest F' that meets the following specifications. First, F' represents, via "find consistency," the same partition of S as F, except for the fact that A and B are now joined in a single set. Second, "find consistency" and "numbering" must still hold on F'.

To deal with the most general case, we assume that A and B are stored in two subtrees of some BUF tree(s) in F. Dealing with simpler cases is similar and will be omitted. Recall that "union(A, B)" must increment $i_{\max}$ by 1, the updated value of $i_{\max}$ being assigned to this union as its ordinal number. This increment of $i_{\max}$ may infringe the "numbering" invariant. To restore this invariant, we remove from the forest F possibly existing pointers either with first field or separate field equal to $i_{\max}$. By the same invariant, there were originally either at most two sibling pointers e' and e'' with first field equal to $i_{\max}$, or at most one pointer e''' with separate field equal to $i_{\max}$, and such pointers can be accessed in constant time. We delete these pointers, and transform the forest F into an equivalent forest F'', no pointer of which is labeled $i_{\max}$. In (23), it is shown that the new forest F'' still satisfies the three invariants and can be produced in $O(k)$ time.

The next task consists of locating in F'', from input A and B, both "apex(A)" and "apex(B)." This stage is accomplished by performing two "finds," which, by "find consistency," require $O(k + h)$ worst-case time, where h is the maximum possible length for a path originating at a leaf in a BUF tree and containing only live pointers. Clearly, the three invariants are not affected by this stage. Next, we transform F'' into an equivalent forest F''', with the property that "apex(A)" and "apex(B)" are live in F'''. This is done by "cleaning" "apex(A)" and "apex(B)": this phase is quite sophisticated, and we refer the interested reader to (23) for the full details of the method. We only mention here that F''' can be produced in $O(k)$ time, and it again meets the three invariants.

Now let T_A and T_B be the BUF (sub)trees of F''' storing, respectively, A and B, and let r_A and r_B be their respective roots. The final task of "union(A, B)" is that of combining T_A and T_B into a single (sub)tree, thus producing the final forest F'. Assume without loss of generality that height(T_B) $\le$ height (T_A). Observe that height(T_A) cannot exceed h, because there is a live path from leaf A to r_A. Our BUF tree union locates a live node v in T_A having the same height as r_B. This takes $O(h)$ steps, e.g., by retracking the "find" that produced r_A for height(T_B) steps. We select one of the following three modes of operation, in analogy with a k-UF tree union.

Type 1. r_B is fat and $v \neq r_A$. Root r_B is made a sibling of v, according to the following rule. If parent(v) is fat, r_B is made the rightmost child of parent(v). If parent(v) is slim, r_B is attached to the right of the rightmost live pointer entering parent(v). At this point, it is set first{[r_B, parent(v)]} = last{[r_B, parent(v)]} = $i_{\max}$. Finally, fat[parent(v)] is set to $i_{\max}$ if appropriate.

Type 2. r_B and $v = r_A$ are both fat nodes. A new node r is created, and the name of r is copied from the name of either r_A or r_B. Next, both r_A and r_B are made children of r, thereby relinquishing their respective names. Finally, first[(r_A, r)], first[(r_B, r)], last[(r_A, r)], and last[(r_B, r)] are all set to $i_{\max}$.

Type 3. This type covers all remaining possibilities, i.e., either root r_B is slim or root $v = r_A$ is slim. We only describe how the case of a slim r_B is handled, the other case being symmetric. Proceeding from left to right, every live child x of r_B is made a child of v, with the following policy. If v is fat, the newcomer pointers will be the rightmost pointers entering v. If v is slim, these pointers will be the rightmost live pointers entering v. The pointer s connecting the leftmost child of r_B to v is marked as a separator with separate(s) = $i_{\max}$. Moreover, the old name of r_B is stored into "label(s)," and "number(s)" is set to the total number of pointers moved. For every redirected pointer e, "last(e)" is set to $i_{\max}$. Finally, "fat(v)" is set to $i_{\max}$ if appropriate.

Finally, a reference indexed by $i_{\max}$ is directed toward the pointer(s) (cf. type 1 or 2) or separator (type 3) introduced by the union. By "slim compression," the fatness of a node can be tested in $O(k)$ time by a walk starting at its leftmost child. This completes our description of union operations. Using BUF trees, Apostolico et al. were able to prove the following theorem.

Theorem 11. (23) BUF trees support each unite and find operation in $O(\log n/\log \log n)$ time and each backtrack in $O(1)$ time, and require $O(n)$ space.

No better bound is possible for any separable pointer algorithm or in the cell probe model of computation, as can be shown by a trivial extension of Theorem 4.

BIBLIOGRAPHY

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Reading, MA: Addison-Wesley, 1974.
2. A. Kerschenbaum and R. van Slyke, Computing minimum spanning trees efficiently, *Proc. 25th Annu. Conf. ACM,* 1972, pp. 518–527.
3. B. W. Arden, B. A. Galler, and R. M. Graham, An algorithm for equivalence declarations, *Commun. Assoc. Comput. Mach.,* **4**: 310–314, 1961.
4. B. A. Galler and M. Fischer, An improved equivalence algorithm, *Commun. Assoc. Comput. Mach.,* **7**: 301–303, 1964.
5. R. E. Stearns and P. M. Lewis, Property grammars and table machines, *Inf. Contr.,* **14**: 524–549, 1969.
6. R. E. Stearns and P. M. Rosenkrantz, Table machine simulation. *Conf. Rec. IEEE 10th Annu. Symp. Switching Automata Theory,* 1969, pp. 118–128.
7. T. Imai and T. Asano, Dynamic segment intersection with applications, *J. Algorithms,* **8**: 1–18, 1987.
8. K. Mehlhorn, *Data Structures and Algorithms,* vol. 3, *Multidimensional Searching and Computational Geometry,* Berlin: Springer-Verlag, 1984.
9. K. Mehlhorn and S. Näher, Dynamic fractional cascading, *Algorithmica,* **5**: 215–241, 1990.
10. J. E. Hopcroft and R. M. Karp, An algorithm for testing the equivalence of finite automata, Tech. Rep. TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, NY, 1971.
11. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms,* Reading, MA: Addison-Wesley, 1983.
12. J. W. Hunt and T. G. Szymanski, A fast algorithm for computing longest common subsequences, *Commun. Assoc. Comput. Mach.,* **20**: 350–353, 1977.
13. H. Aït-Kaci, An algebraic semantics approach to the effective resolution of type equations, *Theoret. Comput. Sci.,* **45**: 293–351, 1986.
14. H. Aït-Kaci and R. Nasr, LOGIN: A logic programming language with built-in inheritance, *J. Logic Program.,* **3**: 185–215, 1986.
15. G. Huet, Resolutions d'equations dans les langages d'ordre 1, 2, . . . ω (in French). Ph.D. dissertation, Univ. de Paris VII, France, 1976.
16. J. S. Vitter and R. A. Simons, New classes for parallel complexity: A study of unification and other complete problems for P, *IEEE Trans. Comput.* **C-35**: 1989.
17. J. Westbrook and R. E. Tarjan, Maintaining bridge-connected and biconnected components on-line, *Algorithmica,* **7**: 433–464, 1992.
18. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, On computing least common ancestors in trees. *Proc. 5th Annu. ACM Symp. Theory Comput.,* 1973, pp. 253–265.
19. H. N. Gabow and R. E. Tarjan, A linear time algorithm for a special case of disjoint set union, *J. Comput. Syst. Sci.,* **30**: 209–221, 1985.
20. J. E. Hopcroft and J. D. Ullman, Set merging algorithms, *SIAM J. Comput.,* **2**: 294–303, 1973.
21. R. E. Tarjan, Finding dominators in directed graphs. *SIAM J. Comput.,* **3**: 62–89, 1974.
22. R. E. Tarjan, Testing flow graph reducibility. *Proc. 5th Annu. ACM Symp. Theory Comput.,* 1973, pp. 96–107.
23. A. Apostolico et al., The set union problem with unlimited backtracking, *SIAM J. Comput.,* **23**: 50–70, 1994.
24. G. Gambosi, G. F. Italiano, and M. Talamo, Worst-case analysis of the set union problem with extended backtracking, *Theoret. Comput. Sci.,* **68**: 57–70, 1989.
25. H. Mannila and E. Ukkonen, The set union problem with backtracking. *Proc. 13th Int. Colloquium on Automata, Languages and Programming* (ICALP 86), 1986; *Lecture Notes in Computer Science,* vol. 226, Berlin: Springer-Verlag, pp. 236–243.
26. H. Mannila and E. Ukkonen, Time parameter and arbitrary deunions in the set union problem. *Proc. 1st Scandinavian Workshop on Algorithm Theory* (SWAT 88), 1988; *Lecture Notes in Computer Science,* vol. 318, Berlin: Springer-Verlag, pp. 34–42.
27. J. Westbrook and R. E. Tarjan, Amortized analysis of algorithms for set union with backtracking, *SIAM J. Comput.,* **18**: 1–11, 1989.
28. C. J. Hogger, *Introduction to Logic Programming,* New York: Academic Press, 1984.
29. H. Mannila and E. Ukkonen, On the complexity of unification sequences. *Proc. 3rd Int. Conf. Logic Programming,* 1986; *Lecture Notes in Computer Science,* vol. 225, Berlin: Springer-Verlag, pp. 122–133.
30. H. Mannila and E. Ukkonen, Timestamped term representation for implementing Prolog. *Proc. 3rd IEEE Conf. Logic Programming,* 1986, pp. 159–167.

31. D. H. D. Warren and L. M. Pereira, Prolog—The language and its implementation compared with LISP, *ACM SIGPLAN Notices,* **12**: 109–115, 1977.

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

33. J. Pearl, *Heuristics,* Reading, MA: Addison-Wesley, 1984.

34. A. C. Yao, Should tables be sorted? *J. Assoc. Comput. Mach.,* **28**: 615–628, 1981.

35. A. M. Ben-Amram and Z. Galil, "On pointers versus addresses, *J. Assoc. Comput. Mach.,* **39**: 617–648, 1992.

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

37. A. N. Kolmogorov, On the notion of algorithm, *Uspehi Mat. Nauk.,* **8**: 175–176, 1953.

38. A. Schönage, Storage modification machines, *SIAM J. Comput.,* **9**: 490–508, 1980.

39. R. E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets, *J. Comput. Syst. Sci.,* **18**: 110–127, 1979.

40. K. Mehlhorn, S. Näher, and H. Alt, A lower bound for the complexity of the union-split-find problem, *SIAM J. Comput.,* **17**: 1093–1102, 1990.

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

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

43. R. E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.,* **31**: 245–281, 1984.

44. R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Meth.,* **6**: 306–318, 1985.

45. M. J. Fischer, Efficiency of equivalence algorithms. In R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations,* New York: Plenum Press, pp. 153–168.

46. J. van Leeuwen and T. van der Weide, Alternative path compression techniques, Tech. Rep. RUU-CS-77-3, Dept. of Computer Science, University of Utrecht, The Netherlands, 1977.

47. T. van der Weide, *Data Structures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms,* Amsterdam, The Netherlands: Mathematisch Centrum, 1980.

48. W. Ackermann, Zum Hilbertshen Aufbau der reelen Zahlen (in German), *Math. Ann.,* **99**: 118–133, 1928.

49. M. L. Fredman and M. E. Saks, The cell probe complexity of dynamic data structures. *Proc. 21st Annu. ACM Symp. Theory Comput.,* 1989, pp. 345–354.

50. J. A. La Poutré, Lower bounds for the union-find and the split-find problem on pointer machines. *Proc. 22nd Annu. ACM Symp. Theory Comput.,* 1990, pp. 34–44.

51. M. Loebl and J. Nešetřil, Linearity and unprovability of set union problem strategies. *Proc. 20th Annu. ACM Symp. Theory Comput.,* 1988, pp. 360–366.

52. W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* Berlin: Springer-Verlag, 1981.

53. R. K. Ahuja et al., Faster algorithms for the shortest path problem, *J. Assoc. Comput. Mach.,* **37**: 213–223, 1990.

54. K. Mehlhorn, *Data Structures and Algorithms,* vol. 2, *Graph Algorithms and NP-Completeness,* Berlin: Springer-Verlag, 1984.

55. A. Apostolico and C. Guerra, The longest common subsequence problem revisited, *Algorithmica,* **2**: 315–336, 1987.

56. D. Eppstein et al., Sparse dynamic programming I: Linear cost functions, *J. Assoc. Comput. Mach.,* **39** (3): 519–545, 1992.

57. P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inf. Process. Lett.,* **6**: 80–82, 1977.

58. N. Blum and H. Rochow, A lower bound on the single-operation worst-case time complexity of the union-find problem on intervals, *Inf. Process. Lett.,* **51**: 57–60, 1994.

59. H. N. Gabow, A scaling algorithm for weighted matching on general graphs. *Proc. 26th Annu. Symp. Foundations Comput. Sci.,* 1985, pp. 90–100.

60. Z. Galil and G. F. Italiano, A note on set union with arbitrary deunions, *Inf. Process. Lett.,* **37**: 331–335, 1991.

Alberto Apostolico
Purdue University
Giuseppe F. Italiano
Università "Ca' Foscari" di Venezia