1989

The Set Union Problem With Unlimited Backtracking

Alberto Apostolico

Giuseppe F. Italiano

Giorgio Gambosi

Maurizic Talamo

Report Number:
89-908

Apostolico, Alberto; Italiano, Giuseppe F.; Gambosi, Giorgio; and Talamo, Maurizic, "The Set Union Problem With Unlimited Backtracking" (1989). *Department of Computer Science Technical Reports*. Paper 774.
<https://docs.lib.purdue.edu/cstech/774>

**THE SET UNION PROBLEM WITH
UNLIMITED BACKTRACKING**

Alberto Apostolico
Giuseppe F. Italiano
Giorgio Gambosi
Maurizio Talamo

CSD-TR-908
September 1989
(Revised September 1990)

The Set Union Problem with Unlimited Backtracking

*Alberto Apostolico*¹

Department of Computer Science, Purdue University, West Lafayette, IN 47907
and

Dipartimento di Matematica Pura e Applicata, Università de L'Aquila, L'Aquila, Italy

*Giuseppe F. Italiano*²

Department of Computer Science, Columbia University, New York, NY 10027
and

Dipartimento di Informatica e Sistemistica, Università di Roma, Roma, Italy

Giorgio Gambosi

Istituto di Analisi dei Sistemi ed Informatica del CNR, Roma, Italy

Maurizio Talamo

Dipartimento di Matematica Pura e Applicata, Università de L'Aquila, L'Aquila, Italy
and

Istituto di Analisi dei Sistemi ed Informatica del CNR, Roma, Italy

Purdue University CS TR 908

(September, 1989)

(Revised September 1990)

Abstract: An extension of the disjoint set union problem is considered, where the extra primitive *backtrack(i)* can undo the last i unions not yet undone. Let n be the total number of elements in all the sets. A data structure is presented that supports each union and find in $O(\log n / \log \log n)$ worst-case time and each *backtrack(i)* in $O(1)$ worst-case time, irrespective of i . The total space required by the data structure is $O(n)$. A byproduct of this construction is a partially persistent data structure for the standard set union problem, capable of supporting union, find and find-in-the-past operations, each in $O(\log n / \log \log n)$ worst-case time. All these upper bounds are tight for the class of separable pointer algorithms as well as in the cell probe model of computation.

Key words: disjoint set union, deunion, unlimited backtrack, design and analysis of algorithms.

AMS subject classification: 68C25

---

¹ Supported in part by the French and Italian Ministries of Education, by NFS Grant CCR-89-00305, by NIH Library of Medicine Grant R01 LM05118, by AFOSR Grant 90-0107 and by the British Research Council Grant SERC-E76797.

² Supported in part by an IBM Graduate Fellowship, by NSF Grants CCR-86-05353 and CCR-88-14977, and by the ESPRIT II Basic Research Actions Program of the European Communities under Contract no. 3075 (Project ALCOM).

1. Introduction

The disjoint *set union problem* has been studied extensively during the past two decades [1, 2, 4, 9, 18, 19, 21]. The problem consists of maintaining an efficient internal representation for a dynamic partition of an n -elements set S which undergoes a sequence of operations of the following kinds:

$\text{union}(A, B, C)$: combine the two subsets of S named, respectively, A and B into a new set named C .

$\text{find}(x)$: return the name of the unique subset of S that currently contains the element x .

Initially, the partition of S consists of the n singleton sets $\{1\}, \{2\}, \dots, \{n\}$, and the name of set $\{i\}$ is i . Various conventions can be made about the way in which the name C is chosen in a union, and they give rise to a small number of variations of the problem. Typically, the name of every set at any time is maintained to coincide with the name (an integer in $[1, n]$) of one of the elements of that set. Also, the name C in a union is usually one of the names of the two input sets. Along these lines, C can be rigidly identified with A , or it can be left unspecified and result in either A or B at runtime, depending on the details of the implementation of a union. All such classes of restrictions do not affect the substance of the set union problem, but they allow to withdraw the third argument C from the format of a union. Throughout this paper, we reason in terms of the primitive $\text{union}(A, B)$, which combines the two subsets named A and B into a new set named either A or B .

The most efficient algorithms for the set union problem were devised by Tarjan [18, 21]. Such algorithms run in $O(n + m\alpha(m + n, n))$ time on a sequence consisting of at most $n - 1$ unions and m finds. Here α is a functional inverse of the Ackermann's function. No better performance is possible for the class of *separable pointer algorithms* [19, 21], i.e., in the *pointer machine* [17, 19] model of computation. The storage of a pointer machine consists of an unbounded collection of records connected by pointers. Each record can contain an arbitrary amount of additional information, but no arithmetic is allowed to compute the address of a record. Separable pointer algorithms must obey the following rules [2, 19] :

- (i) The operations must be performed on line.
- (ii) Each set element is a node of the data structure. There can be also additional nodes.
- (iii) (*Separability*). After each operation, the data structure can be partitioned into subgraphs such that each subgraph corresponds exactly to a current set. No edge leads from a subgraph to another.
- (iv) To perform $\text{find}(x)$, the algorithm obtains the node v containing x and follows paths starting from v until it reaches the node which contains the name of the corresponding set.
- (v) During any operation the algorithm may insert or delete any number of edges. The only restriction is that rule (iii) must hold after each operation.

Very recently, Fredman and Saks [4, 5] showed that even in the powerful *cell probe* model of computation, which encompasses the power of a Random Access Machine, no better performance than $O(n + m\alpha(m + n, n))$ is possible for a sequence of n unions and m finds.

Despite the low amortized [20] bounds, Blum [2] showed that the worst-case bound per oper-

ation for the set union problem is $O(\log n / \log \log n)$ ¹. Also this upper bound is known to be tight for the class of separable pointer algorithms [2] and in the cell probe model of computation [5].

In recent years, some variants of the set union problem were considered, where individual unions or sequence of unions can be backtracked upon [6, 7, 12, 14, 23]. Such extensions are motivated by problems arising in the memory management by Prolog interpreters [8, 12, 13, 22], in the incremental execution of logic programs [14], and in the implementation of search heuristics for resolution [10, 16]. Along these lines, Mannila and Ukkonen [12] proposed the *set union problem with backtracking*, where a third operation *deunion* is introduced that undoes the last union not yet undone.

Westbrook and Tarjan [23] proved that any separable pointer algorithm for the set union problem with backtracking requires $\Omega(m \log n / \log \log n)$ time in performing a sequence of m find, union and deunion operations. They gave also several algorithms with $O(\log n / \log \log n)$ amortized running time, thus matching this lower bound. The overall space required by these algorithms is $O(n)$ [23].

An extension of the set union problem with backtracking, was considered in [6, 7]. In this extension, a real number is assigned to each union as the *weight* of that union, and it is possible to backtrack either to the union of maximal weight or to a generic union performed in the past. This extension has both a *static* [6] and a *dynamic* [7] version, depending on whether or not the weights can be dynamically updated. Both versions can be solved in $O(\log n)$ worst-case time per operation and in $O(n)$ overall space [7]. Also this upper bound is tight for the class known as *non-separable pointer algorithms* [15].

In this paper, we consider a generalization of set union with backtracking where, in addition to the usual union and find operations, a primitive *backtrack(i)* is introduced which undoes the last i unions not yet undone. We call this problem the *set union problem with unlimited backtracking*. An efficient solution to this problem is desirable in several applications, notably, in the implementation of search heuristics for Prolog interpreters [8, 10, 16, 22]. In that framework, sequences of unions correspond to unifications between terms [13], and a multiple deunion would enable one to quickly recover from an unsuccessful search by returning to one of the most promising states among those examined so far.

Since *backtrack(1)* is simply a deunion operation, the algorithms in [23] can be easily adapted to handle also unlimited backtracks, within the same amortized time and space performance. If, however, *backtrack(i)* is regarded as a single operation, then such an implementation of it requires $\Omega(n)$ time in the worst-case.

Our implementation of the set union problem with unlimited backtracking takes worst-case time $O(\log n / \log \log n)$ for each union or find operation and constant time for each *backtrack(i)*, irrespective of i . We use $O(n)$ overall space. Clearly, the $\Omega(\log n / \log \log n)$ per-operation lower bounds of [2] and [5] still holds for our problem, so that our bound is tight both in the separable-pointer and cell-probe models of computation.

A byproduct of our construction is a partially persistent [3] data structure that supports each union, find and *find-in-the-past* operation in $O(\log n / \log \log n)$ worst-case time, with $O(n)$ space

---

¹ Throughout this paper all logarithms are taken to the base 2, unless explicitly noted otherwise.

usage. This is faster than the bound achieved in [14], but the specifications of a union used in [14] are slightly different.

2. Union-Find with Deunion

As mentioned in [23], the data structure proposed by Blum [2] could be easily adapted to support also deunions in time $O(\log n / \log \log n)$ per operation and overall space $O(n)$. In this Section, we carry out the details of this extension which will serve as a basis for the subsequent developments. We start by recalling the tree structure used in [2].

For any $k \geq 2$, a k -UF tree is a rooted tree T such that:

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

Clearly, the height of a k -UF tree with n leaves is bounded by $\lceil \log_k n \rceil$. We say that a node of a k -UF tree is *slim* if it has less than k children, *fat* otherwise. A consequence of the definitions above is that only the root of a k -UF tree can be slim. Disjoint sets are represented by k -UF trees as follows. The elements of the set are stored in the leaves and the name of the set is stored in the root. Furthermore, the root contains also the height of the tree and a bit specifying whether it is fat or slim.

A $\text{find}(x)$ is performed by first climbing up the tree from the leaf containing x , and then returning the name stored in the root. This takes $O(\log_k n)$ time.

To perform $\text{union}(A, B)$ for two non-singleton sets A and B , we need access to the roots r_A and r_B of the corresponding k -UF trees T_A and T_B . Blum assumed that his algorithm obtained r_A and r_B in constant time, prior to performing a $\text{union}(A, B)$. If this is not possible, r_A and r_B can be obtained by means of two finds (i.e., $\text{find}(A)$ and $\text{find}(B)$), due to the property that the name of a set is one of the elements of that set. Once r_A and r_B are available, the two k -UF trees T_A and T_B are combined as follows.

Assume without loss of generality that $\text{height}(T_B) \leq \text{height}(T_A)$. Let v be the node on the path from the rightmost leaf of T_A to r_A such that the subtree of T_A rooted at v has the same height as T_B . Node v is found by starting at r_A and then following the leftmost downward branch of each node for exactly $\text{height}(T_A) - \text{height}(T_B)$ steps. Having reached node v , the manipulations to be performed depend on the type of union, according to the following.

Type 1 - Root r_B is fat (i.e., has no less than k children) and v is not the root of T_A . Then r_B is made a sibling of v .

Type 2 - Roots r_A and r_B are fat and v is 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 - Type 3 covers all remaining possibilities. Specifically, if root r_B is slim, then the children of r_B are made the rightmost children of v . If root r_B is fat, then, since we are not in type 1 or 2, we have that $v = r_A$ and v is slim. In this case, all the children of r_A are made rightmost children of r_B .

Note that new arcs are created only as part of a type 1 or 2 union. Type 3 unions involve instead what we call *re-directing* existing arcs. We make the assumption that the node representing

a singleton set is a fat node. From now on, we fix $k = \lceil \log n / \log \log n \rceil$. This choice of k is motivated by the following theorem by Blum [2].

Theorem 1. [2] *UF trees support either union or find in $O(\log n / \log \log n)$ time and $O(n)$ space.*

Blum [2] proved also that this bound is tight for the class of separable pointer algorithms. Very recently, this result was extended to the cell probe model of computation by Fredman and Saks [5].

Theorem 2. [2, 5] *Any separable pointer algorithm for the disjoint set union problem has single operation worst-case time complexity $\Omega(\log n / \log \log n)$. The same lower bound holds in the cell probe model of computation.*

A UF tree can be easily adapted to support deunions. We list the few upgrades needed. The resulting structure will be called a *DUF tree*. For each node v , the children of v are also linearly ordered from left to right in a doubly linked list. Two DUF trees T_A and T_B are combined in much the same way as UF trees, except that type 3 unions are now expanded as follows. Assume root r_B is slim. All the children of r_B are made the rightmost children of v . The arc connecting the leftmost child of r_B to v is marked a *separator*, and the label of r_B (i.e. the old name of the set represented by T_B) is recorded in that arc. Similar manipulations are performed when r_A is slim.

Because of the linear order on the children of each node, each union can be implicitly described by its *characteristic arc*, defined as follows. The characteristic arc of a type 1 union is $(r_B, \text{parent}(v))$. The characteristic arc of a type 2 union is (r_A, r) . Finally, the characteristic arc of a type 3 union is the separator associated with that union. With the help of a stack P , characteristic arcs enable to perform quick deunions. Following each union, a pointer to its characteristic arc is pushed onto P , along with the type identifier (1, 2 or 3) of that union. Type 1 and type 2 unions are then easily undone in constant time, following the pointer to the characteristic arc. To undo a type 3 union, we access the separator pointed to by the top of the stack and disconnect this arc and all the arcs to its right. All the nodes so detached from the tree are made children of a new root to which the name stored in the separator is assigned. By the definition of type 3 union, this requires $O(k)$ time. Note that $O(n)$ nodes and arcs can be in the structure at any time. The stack-records correspond to unions not yet undone, and there can be at most $n - 1$ such unions. Therefore, the total space required is $O(n)$. In conclusion, the following theorem holds.

Theorem 3. *DUF trees support each union, find and deunion in $O(\log n / \log \log n)$ time and $O(n)$ space.*

3. Upgrading DUF trees

In the set union problem with unlimited backtracking, deunions are replaced by backtracks: for any integer $i \geq 0$, $\text{backtrack}(i)$ undoes the last i valid unions performed. $\text{Backtrack}(i)$ is performed on DUF trees in $O(i \log n / \log \log n)$ time, simply by carrying out i deunions as described in the previous Section. This is clearly undesirable, since i can be $\Theta(n)$. On the other hand, as long as

we insist on deleting arcs the moment that they are invalidated by backtracking (i.e., in the *eager* mode [23]), then the cost of $\text{backtrack}(i)$ is $\Omega(i)$, since at least one arc must be removed for each erased union. To side-step this lower bound, the removal of arcs invalidated by backtracking must be deferred to some possible future operation. This mode of operation is called *lazy*. In a strict sense, the lazy approach infringes the separability condition stated in the introduction. However, the substance of that condition would still be met if one maintains that an arc is never traversed once it is invalidated (cf., e.g., [23]). Our approach guarantees this fact and thus does not depart substantially from the separability assumption.

In what follows, we present a data structure suitable for storing a collection of disjoint sets in such a way that the identity of each set in the collection is preserved. We call this data structure k -*BUF* tree or, with the implicit assumption that $k = \lceil \log n / \log \log n \rceil$, simply *BUF* tree. We will show that *BUF* trees support union and find in $O(\log n / \log \log n)$ time and $\text{backtrack}(i)$ in constant time, independent of i .

We begin by describing the main features of *BUF* trees, and by highlighting the associated implementation of the union, find and backtrack operations. *BUF* trees retain the basic structure of *DUF* trees, but differ from them primarily because of some implicit attributes defined on the arcs. With *BUF* trees, a union is still performed according to one of three different patterns of management, like with *DUF* trees. In particular, we will have that type 1 and type 2 unions create new arcs while type 3 unions only re-direct already existing arcs. With *BUF* trees, however, a union must perform some additional manipulations on arcs, besides those pertaining to the mere aggregation of the two input subsets. In the following, we say that an arc e is *handled* by a certain union only if e is either created or re-directed by that union during the aggregation stage of that union. The main difference with *DUF* trees is that now, due to the lazy approach, we allow arcs and separators to possibly survive in the data structure also after the union which introduced them has been invalidated by backtracking. At any given time, we call a union *valid* if it has not yet been undone by backtracks, *void* otherwise. We further partition void unions as follows. A void union is *persisting* if the arcs handled by that union have not yet been physically removed from the data structure, and is *dissolved* otherwise. This classification of unions induces a corresponding taxonomy on arcs and separators, as follows. In a *BUF* tree, an ordinary arc can be *live*, *dead*, or *cheating*, and a separator arc can be, in addition, either *active* or *inactive*. Informally, live arcs represent connections not yet invalidated by backtracks; this happens when the last union which handled them is still valid. Dead arcs represent instead connections that, although still in the structure, only await to be destroyed; this happens when the first union which created them is a void persisting union. Between live and dead arcs, lie cheating arcs. They occur when the first union which created them is valid but the last union which 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 *DUF* trees, separators are associated with type 3 unions. At any given time, a separator is *active* if its associated union is valid, *inactive* otherwise. A node of a *BUF* tree is *live* if there is at least one live arc entering it, and is *persisting* otherwise. In analogy with the nodes of *DUF* trees, the live nodes of *BUF* trees can be *slim* or *fat*, but this is decided based only on the number of live arcs entering each node. Specifically, a node is slim if the number of live arcs entering it is less than k , and fat if the number of live arcs

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 which 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:

- (i) The first union performed gets ordinal number 1.
- (ii) When a union is made void by backtracking, it relinquishes its ordinal number.
- (iii) A new union gets ordinal number equal to one 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 . This implementation of backtrack does not affect any arc in the forest, but its effect is implicitly recorded in the change of status of some arcs and separators. Part or all of these arcs might be removed or re-directed later, while performing subsequent union operations. In any event, we need to ensure the consistency of the forest of trees under this newly introduced operation. 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 below correctly returns the name of the set currently containing x . We refer to the consistency of the forest as *Invariant 0*. The complete specification of this invariant requires some additional notions.

First, each arc e in a *BUF* tree T has two unions associated with it, as follows. 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 physically undone (i.e., either a valid or a persisting union) which handled e . We will maintain that $ord(first\_union(e)) \leq ord(last\_union(e))$ for every arc e . In a consistent *BUF* tree, an arc e is *dead* if and only if $first\_union(e)$ is void (i.e., e has to be destroyed since it gives a connection made void by some intervening backtrack). Similarly, arc 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 has to be replaced but 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 has also associated the type 3 union which 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 $S_1, S_2, \dots, S_p$ be the disjoint sets stored in T . We specify the mapping from the set of leaves of T to the set of names of $S_1, S_2, \dots, S_p$. 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 towards the root of T following live arcs until a node without outgoing live arcs 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 arc leaving $\text{apex}(x)$, i.e., $\text{apex}(x)$ is the root r of T . We will maintain that the name Y of S_q is stored in r .

Dead apex - The arc leaving $\text{apex}(x)$ is dead. We will maintain that the name of S_q is stored in $\text{apex}(x)$.

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

The above description explains 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 formalize this condition as Invariant 0.

Invariant 0 (Find consistency). Prior to the execution of each operation of a sequence σ of operations, and for every element x of S , the following holds. If $\text{apex}(x)$ is either dead or live, then the name of the set containing x is stored in $\text{apex}(x)$. If $\text{apex}(x)$ is cheating, then the name of the set containing x is stored in the rightmost inactive separator to the left of $\text{apex}(x)$, and such a separator falls within $k - 1$ arcs to the left of $\text{apex}(x)$. •

The following fact is an immediate consequence of Invariant 0.

Fact 1. *BUF trees support each find operation in time $O((k + h)t)$, where t is the time needed to test the status of an arc and h is the maximum length of an ascending path from a leaf x to its apex in the tree.*

In the following sections, we show 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.

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 $\text{union}(A, B)$ consists of finding in T_1 and T_2 the roots of the smallest subtrees which 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 for *DUF* trees in Section 2. 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 Invariant 0 through each union, find and backtrack. This is discussed in the next sections.

4. Dominance trees and the procedure “restore”

As said, we follow the lazy approach of undoing unions made void by backtracks not immediately, but rather during the execution of subsequent unions. Within the claimed time bounds, however,

a single union cannot undo *all* the currently persisting unions. On the other hand, this is also not strictly necessary. What is necessary for a union is to undo all the persisting unions that undermine its own consistent execution, along with the validity of Invariant 0 on the resulting forest of *BUF* trees. It turns out that such a reduced task can be performed within the claimed time and space bounds, at the expense of some additional bookkeeping.

Our technique consists of maintaining the edges in every *BUF* tree T grouped into clusters, a *cluster* being defined as a maximal set E of consecutive sibling arcs with the property that *last_union* is the same for all the elements of E . We will maintain that the size of any cluster is at most $k - 1$ at any given time. At any point in the computation, a cluster is *persisting* if the last field common to its arcs exceeds the current value of i_{max} , and *live* otherwise. This Section describes the structure of such clusters, and then details the operation of a procedure *restore*, that will carry out a recurrent subtask of our *BUF*-tree implementation of a union. In informal terms, the task of *restore* is that of removing all dead arcs from the input cluster, and then partitioning the remaining arcs in a certain number of smaller, yet live clusters. We will see that any union involves at most a constant number of calls to *restore*, and that the cost of each such call is $O(\log n / \log \log n)$ time.

Before describing the structural properties of clusters, we need to make some additional assumptions on the structure of *BUF* trees. To each arc e , two integers $first(e)$ and $last(e)$ are assigned. They represent, respectively, the ordinal number given to $first\_union(e)$ and to $last\_union(e)$. Beside $first(s)$ and $last(s)$, each separator s contains the following additional information. An attribute $separate(s)$ is the ordinal number given to $separate\_union(s)$. Furthermore, $label(s)$ is the name destroyed by $separate\_union(s)$ and $number(s)$ is the total number of arcs moved during the execution of $separate\_union(s)$. These latter arcs will be maintained to fall immediately to the right of s . Since $separate\_union(s)$ is a type 3 union, then $number(s) < k$. By definition of cluster, all the edges in a cluster E have the same value of *last* field. We refer to $last(E)$ as the value shared by the last fields of all the arcs in E .

For each node v , $fat(v)$ is the ordinal number of the last union which made v a fat node, provided that the effects of that union have not been physically removed from the data structure (i.e., that union is not a *dissolved* union). If no such union exists, then $fat(v)$ is undefined. According to this convention, a slim node which was once fat may have a defined fat number. In addition to Invariant 0, we will maintain the invariants given below.

Invariant 1 (The i_{max} invariant). At any time, the following properties hold. For every arc e in a *BUF* tree, arc e is dead if and only if $i_{max} < first(e)$, is cheating if and only if $first(e) \leq i_{max} < last(e)$, and is live if and only if $i_{max} \geq last(e)$. If, in addition, e is a separator, then e is inactive if and only if $i_{max} < separate(e)$ and e is active if and only if $i_{max} \geq separate(e)$. For every node v in the tree such that $fat(v)$ is defined, $fat(v) \leq i_{max}$ if and only if v is fat. •

Maintaining Invariant 1 enables us to test the status of an arc in constant time. One more important consequence of this invariant is that either all arcs in a cluster are live or none is. Let now e and f be two arcs in a cluster E . We write $e < f$ if e precedes f in the left-to-right order, and we denote by $|f - e|$ the number of consecutive arcs between e and f (including both). If s is a separator in E , then we say that s *dominates* f if and only if $s < f$ and $|f - s| \leq number(s)$. We maintain also the following invariant.

Invariant 2 (The nesting invariant). Let E be a cluster. If $|E| = 1$, then the only element of E is not a separator. Assume now $|E| > 1$. Then, if the leftmost arc of E is a separator, say s , then $\text{separate}(s) = \text{last}(s)$ and $\text{number}(s) = |E|$. If the leftmost arc of E is not a separator, then $|E| = 2$, E is the leftmost one among its sibling clusters, and E contains no other separators. In general, if s' and s'' are any two separators in E and s' dominates s'' , then s' dominates also any arc e dominated by s'' . •

The nested structure of a cluster E delimited by a left separator is detailedly described with the aid of a rooted, ordered tree called the *dominance tree* $D(E)$ of E . The leaves of $D(E)$ in preorder correspond bijectively to the arcs of E (including separators) from left to right; the internal vertices of $D(E)$ correspond bijectively to the separators. Thus, given a simple arc e in E , there is only one leaf ℓ in $D(E)$ corresponding to e , while there is a leaf ℓ and also an internal vertex v in $D(E)$ in correspondence with each separator of $D(E)$. If s is such a separator, then ℓ is the leftmost leaf in the subtree of $D(E)$ rooted at v .

The main feature of $D(E)$ is the following. Let ℓ be the leaf of $D(E)$ that corresponds to arc $e \in E$. Then, the internal vertices on the path from ℓ to the root of $D(E)$ correspond to the separators that dominate e , in the same succession as such separators are met in E starting from e and scanning E from right to left (see Fig. 1). In the following, we will not distinguish between an arc or separator of E and its corresponding vertex in $D(E)$, whenever our meaning is made clear by the context.

Besides representing the nestings of separators, dominance trees encapsulate some monotony properties that form the object of our next invariant. Specifically, each vertex v in a dominance tree $D(E)$ gets assigned an integer $r(v)$ ($1 \leq r(v) \leq n$), with the following meaning. If v is a leaf, then $r(v) = \text{first}(v)$. If v is an internal vertex (hence, it maps a separator), then $r(v) = \text{separate}(v)$. We now consider all the arcs entering a node in a *BUF* tree as partitioned into clusters, and we assign similar numbers, denoted by R , to such clusters. Specifically, if either E is the leftmost cluster or $|E| = 1$, then $R(E) = \text{first}(e)$, where e is the leftmost arc in E . Otherwise, $R(E)$ is $\text{separate}(s)$, where s is the separator that coincides with the leftmost arc in E (cf. Invariant 2). The numbers assigned in this way to the vertices of dominance trees and clusters of arcs entering a node will satisfy the monotony condition given below.

Invariant 3 (The monotony invariant). Let the dominance tree $D(E)$ of cluster E be defined. Hence $|E| > 1$. Then the two leftmost children of each internal vertex of $D(E)$ are always two leaves. Moreover, every internal node s (which must correspond to a separator), has $\text{number}(s)$ equal to the number of leaves of the subtree of $D(E)$ rooted at s . Furthermore, if v and v' are sibling vertices of $D(E)$ with $v < v'$, then one of the following two cases applies:

- (i) if v and v' are the two leftmost vertices among their siblings (and thus leaves by the preceding part of this invariant) with $v < v'$, then $r(v) = r(v')$;
- (ii) otherwise $r(v) < r(v')$.

We also have that, if p is the parent of v , then $r(p) > r(v)$.

The individual clusters entering a slim node of a *BUF* tree obey the following rules. The leftmost cluster of arcs entering a slim node contains two arcs and no separators. Furthermore,

if E and E' are clusters of arcs entering the same slim node and E is on the left of E' , then $R(E) < R(E')$. •

As one of the consequences of Invariant 3, we get that if f is an arc dominated by a separator s , then $\text{first}(f) < \text{separate}(s)$. Our last two invariants are as follows.

Invariant 4 (Slim compression). The live arcs entering any slim node are leftmost among their siblings, and have non-decreasing last fields, from left to right. For fat nodes, this property holds for all the arcs that were directed to that node while the node was slim, including the arcs that made the node fat. •

The slim invariant enables us to decide in $O(k)$ time whether a node is slim or fat, simply by examining the at most k leftmost arcs entering that node.

Invariant 5 (Numbering). For any integer i , $1 \leq i \leq n-1$, there are either at most two sibling arcs with first field equal to i or at most one arc with separate field equal to i . Moreover, there are at most $k-1$ sibling arcs with last field equal to i , and such arcs are in a cluster. Let E be this cluster. If E contains only one arc e , then $\text{first}(e) = \text{last}(e)$. If $|E| > 1$, then the first two arcs of E have the same first field, the second arc of E is not a separator, and the remaining arcs possibly existing in E have first fields different from that of the first two arcs. Moreover, if the leftmost arc of E is not a separator, then $|E| = 2$ and the first fields of its two arcs are equal to their last fields. Otherwise, each arc in E has last field strictly greater than first field, and the leftmost arc has separator field equal to i . Finally, given i we can access in constant time the arcs with first field equal to i or with separator field equal to i . •

The numbering invariant guarantees that the size of each cluster is at most $k-1$, and that no two distinct clusters can have arcs with identical last fields. The last part of the invariant implies that a singleton cluster, or a cluster not delimited by a left separator cannot contain cheating arcs. Thus, such types of clusters contain either live or dead arcs.

We are now ready to describe how a persisting cluster of m arcs is detached from its host *BUF* tree in $O(m)$ time, maintaining Invariants 0-5 on the resulting dismembered structure. This is accomplished by the procedure *restore*, which takes as input some arc e and an integer value $i_{\max}$. The specific tasks of *restore* are:

- 1) to identify the cluster E containing e ,
- 2) to delete the dead arcs possibly existing in E and,
- 3) to re-direct the cheating arcs possibly existing in E towards newly introduced roots, in such a way that, letting F be the forest of trees into which T has been dismembered: (3.1) F represents, via Invariant 0, precisely the same collection of subsets of S formerly represented by T , and (3.2) all non-dead arcs of E become live arcs in F .

To analyze what is involved in a $\text{restore}(e, i_{\max})$, let E be the cluster containing e . If e is already live, then, by the $i_{\max}$ invariant, so are all the other arcs in E , so that *restore* does not need to do anything. Henceforth, we assume $\text{last}(e) > i_{\max}$, i.e., e is either cheating or dead.

Then, by the definition of cluster, there cannot be any live arc in E . To deal with the most general case, assume that $D(E)$ is defined (i.e., E has a left separator), and let ℓ be a leaf of $D(E)$. With reference to the *BUF* tree T containing E , let v be the node from which arc ℓ originates, and T' the subtree of T rooted at v . Assume that ℓ is a dead arc of E . By Invariant 0, any leaf of T' connected to v by a path consisting solely of live arcs belongs currently to the set whose name is stored in v . Thus, *restore* can accomplish its task just by deleting ℓ . Assume now that ℓ is a cheating arc, and let $as(\ell)$ and $is(\ell)$ be, respectively, the highest active and lowest inactive separator on the path from ℓ to the root of $D(E)$. Observe that Invariants 2 and 5 guarantee that $is(\ell)$ is always defined in the case being considered. In the following, the expression "to the left of" is used to mean "to the left of and including".

Lemma 1. *In E , $is(\ell)$ is the rightmost inactive separator to the left of ℓ .*

Proof. The assertion follows trivially from the definition of $D(E)$ if ℓ itself is an inactive separator in E . Thus, we concentrate on the case where ℓ is not an inactive separator. Assume for a contradiction that the rightmost inactive separator to the left of ℓ in E is some s' such that $s' \neq is(\ell)$ and $s' \neq \ell$ (see Fig. 2). By Invariant 2 and our choice of $is(\ell)$, separator s' cannot be on the path from ℓ to the root of $D(E)$ and thus does not dominate ℓ . Since s' falls in E between $is(\ell)$ and ℓ , and $is(\ell)$ dominates ℓ , then $is(\ell)$ dominates s' , whence s' must lie in the subtree of $D(E)$ rooted at $is(\ell)$. Since s' is to the left of ℓ in E and $s' \neq \ell$, then in such a subtree of $D(E)$ we have that s' or an ancestor of s' is a left sibling of either ℓ or an ancestor of ℓ . Let then v stand for s' or the ancestor of s' , and let v' stand for ℓ or the ancestor of ℓ , according to the case. By Invariant 3 and the i_{max} invariant, s' an inactive separator implies that v is an inactive separator. By the same invariants, if v' is not ℓ then v' is an active separator. If v' is an active separator, then, always by Invariant 3, $r(v) \leq r(v')$, whence v' active forces v to be active too, a contradiction. If v' coincides with ℓ , then the fact that ℓ is a cheating arc (i.e., $first(\ell) \leq i_{max}$), along with the conditions $r(v') = first(\ell)$, $r(v) = separate(v)$ and $r(v) \leq r(v')$ (cf. Invariant 3) leads again to contradict that v is inactive. •

Lemma 2. *If $as(\ell)$ is defined, then $as(\ell)$ is a direct son of $is(\ell)$. Moreover, the subtree of $D(E)$ rooted at $as(\ell)$ does not contain any inactive separator.*

Proof. If $as(\ell)$ and $is(\ell)$ are both defined, then from the fact that $as(\ell)$ is active and $is(\ell)$ is inactive, we get that $separate(as(\ell)) < separate(is(\ell))$. By Invariant 3, $is(\ell)$ is then an ancestor of $as(\ell)$. That $as(\ell)$ is a son of $is(\ell)$ follows then straightforwardly from their respective definitions. The second part of the claim is an easy consequence of Invariant 3. •

Once $D(E)$ is given, it is easier to specify the operation of *restore* so as to carry out tasks 1-3 consistently with Invariant 0. For this, let $E = \{e_1, e_2, \dots, e_h\}$, $h < k$, be the cluster handled by *restore* and let x_i , $1 \leq i \leq h$, be the node of the *BUF* tree T from which arc e_i originates. As already observed, if one of the e_i 's is live, then all the e_i 's are live, and *restore* can terminate without affecting the structure of T . Assume therefore that E contains only cheating and dead arcs. The only leaves of T for which something must change are those whose previous apex was

one of the x_i 's. If x_i was a dead apex, then *restore* will make x_i a live apex by simply deleting e_i . In this way, the name of the set of leaves having apex in x_i remains the same. If x_i was cheating, then *restore* will move the arc e_i to a new root with name $\text{label}(\text{is}(e_i))$, and reset the last field of e_i to $\text{separate}(\text{as}(e_i))$. By lemma 1, $\text{is}(e_i)$ is the rightmost inactive separator to the left of e_i , so that also in this case all the nodes with apex x_i preserve their name. Each one of the above cases can be handled trivially in $O(k)$ time, but *restore* must update *all* the arcs of E within this bound. The main handle for this is given by the nested structure of $D(E)$. To clarify this point, we describe a computation on $D(E)$ that we call *dismember* (see Fig. 3).

The goal of *dismember* is threefold. First, it will disconnect from its father every internal node of $D(E)$ that corresponds to an inactive separator. Thus, in the forest of trees produced by *dismember*, no internal vertex other than a root can be an inactive separator. Second, *dismember* will delete every leaf corresponding to a dead arc. Finally, *dismember* will reset the last field of every surviving leaf ℓ to the separate field of the highest active separator on the path from ℓ to the root of $D(E)$, if such a separator exists, and to $\text{first}(\ell)$ otherwise. Thus, there will be only live leaves in the output forest. Observe that these goals are unambiguous and mutually consistent, in force of lemma 1 and lemma 2. The computation can be scheduled according to the preorder visit of the vertices of $D(E)$. It starts thus at the root of $D(E)$ and proceeds with the help of a stack P , which is used to store the inactive separators encountered in the visit. An inactive separator s is pushed onto P the first time it is visited, and popped from P immediately after all nodes in the subtree rooted at s have been handled. When a separator is popped from P , it is also disconnected from its father in $D(E)$, and it relinquishes its attributes as a separator in E . Assume that separator s was just pushed onto P . The computation considers all the children of s from left to right. If the child being considered is a leaf, then its last field is immediately updated. If it is an active separator, then *dismember* visits the subtree of $D(E)$ rooted at such a separator updating all leaves in that subtree. Finally, if the child being considered is an inactive separator, then it is pushed onto P and the computation proceeds recursively on the children of such a separator.

It is clear that *dismember* takes time $O(|E|)$. Assume that, whenever *dismember* deleted a leaf of $D(E)$, it also removed the corresponding arc of E . This accomplishes subtask 2 of *restore*. The following few extra manipulations on the forest at the outset of *dismember* suffice to accomplish subtask 3. First, for each tree D in that forest, the root x of a new *BUF* tree is created. Next, the arcs of E that are mapped into the leaves of D are considered in their left-to-right order, and each arc is re-directed to x in succession, along with its applicable attribute fields (i.e., first and last, and, for separators, also label, separate and number). The only exception to this rule is represented by the leftmost arc, which corresponds to the leftmost leaf of D and also to the root of D . This arc surrenders its separator attributes, thus relinquishing its status as a separator, but its label field is stored into node x . The remaining separators are active, and they retain their attributes. Observe, incidentally, that the number field of each such separator is still consistent, in force of the second part of lemma 2 (no pruning of $D(E)$ took place below an active separator). At this point, lemma 1 and Invariant 0 yield that subtask 3.1 of *restore* is accomplished provided only that every surviving arc of E is live. Recall that the only field changed by *dismember* is the last field of cheating leaves and their associated separators. Specifically, the last field of a leaf ℓ is set equal to $\text{separate}(\text{as}(\ell))$

if $as(\ell)$ is defined, and to $first(\ell)$ otherwise. Invariant 3 guarantees then that every leaf of D has become live in this way, which accomplishes subtask 3.2.

We now consider subtask 1, and also dispose of the cases where $D(E)$ is not defined. Clearly, $restore(e, i_{max})$ can check the status of e in constant time, by the i_{max} invariant. For live e , the procedure does nothing more. Thus, we concentrate on the cases where e is either dead or cheating. By Invariant 5 and definition of cluster, the cluster E containing e is formed by at most $k - 1$ arcs. Thus, E can be identified trivially in $O(k)$ time, by checking the last fields of the arcs in an interval containing e and of size at most $k + 1$. If E is not delimited by a left separator, then (cf. Invariant 5 and the comment following it) we have $|E| = 2$, and the arcs in E are dead. The procedure deletes these 2 arcs and terminates, in constant time.

As it is easily checked, there is no need to maintain dominance trees explicitly. The traversal of $D(E)$ performed by *dismember* can be simulated by scanning E from left to right with an auxiliary stack. The stack is used as before to store the separators encountered in the scanning, a separator being kept in the stack until all the arcs within its dominion have been updated. Although $D(E)$ is not given explicitly, the procedure can use some easy bookkeeping on the number fields of the separators in order to detect the condition that the dominion of a separator has been exhausted. In conclusion, we can record the following theorem.

Theorem 4. *There is an implementation of $restore(e, i_{max})$ that takes time $O(k)$.*

We now show that *restore* preserves our invariants.

Theorem 5. *The procedure *restore* maintains Invariants 0-5.*

Proof. Let $E = \{e_1, e_2, \dots, e_h\}$, $h < k$, be the cluster handled by *restore* and let x_i , $1 \leq i \leq h$ be the node of the *BUF* tree T from which arc e_i originates. Since *restore* does nothing if one (hence, every) e_i is live, we assume henceforth that E contains only cheating and dead arcs.

That Invariant 0 is maintained by *restore* follows straightforwardly from the discussion preceding theorem 4. That discussion also shows that *restore* preserves the part of Invariant 1 that involves i_{max} . Consider now the part of Invariant 1 that involves the fat field. Since all newly introduced nodes are slim by construction, then these nodes do not have a defined fat field. The only other node of T whose fat field could be possibly affected by *restore* is the node v which the arcs in the cluster E were entering prior to *restore*. Since the arcs of E are not live, however, they did not contribute in any way to the fatness of v (only live arcs do). Since the procedure does not change the value of i_{max} , then v will remain slim or fat after *restore*, consistently as before.

To discuss the next invariants, consider the forest of trees produced by *dismember*. We have already seen that every nontrivial tree in such a forest represents a collection of formerly cheating arcs of E that were changed into live arcs. We show now that *restore* has actually done more than just resuscitate those arcs. Specifically, we claim that every nontrivial tree in the forest produced by *restore* represents a collection of live *clusters* that obey, with their associated dominance trees, every applicable property in Invariants 2-5.

For this, let D be one of the trees produced by *dismember*, and s the inactive separator of E that corresponds to the root of D . Consider the children of s in $D(E)$, from left to right. The

first observation is that, if s became the root of nontrivial tree D , then s has at least two children, and the two leftmost children of s in D are precisely the two leftmost children of s in $D(E)$. In fact, let ℓ and ℓ' be the two leftmost children of s in $D(E)$. Then, Invariant 3 guarantees that $\text{first}(\ell) = \text{first}(\ell')$. If ℓ and ℓ' are both dead, then they are deleted. However, no sibling of ℓ in $D(E)$ could be a live leaf or an active separator in this case, because of the monotonicity of the r -values prescribed by Invariant 3 for ℓ and its siblings. Hence s could not be the root of a tree in the forest built by *dismember*. Assume now that ℓ and ℓ' are cheating. Then, $as(\ell)$ and $as(\ell')$ are not defined and $is(\ell) = is(\ell') = s$. In this case, s will be the root of a tree, within which ℓ and ℓ' will still be the two leftmost children of the root. Thus, s has at least two children in D , and such children are leaves of D . These two leaves form the leftmost cluster in the new *BUF* tree created by *restore*. By the horizontal monotonicity of Invariant 3, the size of this cluster is 2. By the operation of *restore*, neither arc in the cluster is a separator. This cluster complies with every applicable part of Invariants 2-5.

The other children of the root s of D are either leaves or active separators that did not fall within the dominion of any other active separators of E . Let s' be one such child of s , and consider the two possible cases below.

Case 1: s' is a leaf. Then *dismember* set $\text{last}(s') = \text{first}(s')$. Recall that, in $D(E)$, $\text{first}(s') = \tau(s')$. If s'' is the immediate right sibling of s' , then $\tau(s'') > \tau(s')$ by Invariant 3. Hence s' becomes a singleton cluster in D , with R number equal to the old r -number of s' .

Case 2: s' is an active separator of E . Recall that *dismember* assigns to s' and all of its descendants a last field equal to $\text{separate}(s')$, and leaves $\text{number}(s')$ untouched. The subset of E that is represented by the leaves of D forms a cluster delimited by a left separator and with a consistent separator nesting. The R -number of such a cluster is the old r -number of s' . Clearly, the subtree of D rooted at s' is the consistent dominance tree of such a cluster.

In view of lemma 1 and lemma 2, the analysis above shows that Invariant 2 is preserved by *restore*. Since no number field or r -number is altered, then the part of Invariant 3 that concerns these fields and numbers is preserved. By the operation of *restore*, the leaves in the subtrees rooted at the children of s will be directed towards the same root of a newly created *BUF* tree. Our analysis of cases 1 and 2 above displays that the monotonicity of the r -numbers on the children of s before *dismember* guarantees the monotonicity of the R -values of clusters entering this root. With regard to the node of the *BUF* tree that the arcs of E entered before *restore*, clearly the R values of the former siblings of E was not been affected, whence their relative order is preserved. Thus, Invariants 2 and 3 are thoroughly maintained.

We now turn to Invariant 4. Since $|E| < k$, and all the re-directed arcs will enter new nodes, no fat node is introduced by *restore*. The novel slim nodes vacuously comply with Invariant 4, since all arcs entering them are live. The non-decreasing ordering of the last fields of such arcs is secured by Invariant 3. In fact, the new last fields are former r -numbers (i.e., either separate or first fields depending upon the type of node - an internal active separator or a leaf - encountered by *dismember*), and these r -numbers obeyed Invariant 3. Consider now the *BUF*-tree node v which arcs in E entered before *restore*. The only situation under which such arcs are disconnected from v is when they are not live. But in such a situation the arcs of E did not contribute in any way to the fatness or slimness of v . Thus, *restore* preserves also Invariant 4.

Finally, we deal with Invariant 5. Recall that *restore* does not introduce new values for either first or separate fields. Furthermore, all the re-directed arcs which get same last field are siblings because of the implementation of *restore* and do not exceed $k - 1$, since $|E| < k$ by hypothesis. We have seen that a new cluster E' created by *restore* contains only one arc ℓ only if ℓ was a leaf in the input dominance tree $D(E)$ and $as(\ell)$ was not defined. In this case, $last(\ell)$ was updated by *restore* to $first(\ell)$, consistently with Invariant 5. A new cluster E' , $|E'| > 1$, without a left separator is created by *restore* only when the two leftmost leaves of $D(E)$ are encountered, and such leaves are assigned identical last and first fields. Otherwise, if $|E'| > 1$, E' was obtained as a subtree of $D(E)$ rooted at some active separator s . In this case, having established already Invariant 3, it follows that the two leftmost arcs in E' have the same first field, and that the second arc in E' is not a separator. We have also seen that, in this case, all the arcs in E' get $separate(s)$ as their new last field, and that such a new last field is greater than all their first fields, also by Invariant 3. By definition of dominance tree, the leftmost arc in E' is the separator s . Thus, for every leaf ℓ in E' , we get that $last(\ell) = separate(s)$ as prescribed by Invariant 5. In conclusion, also Invariant 5 is preserved. •

Before continuing with our discussion, it is instructive to revisit the outline of a *BUF*-tree union given at the end of Section 3. In that outline, we said that a necessary preliminary stage of a $union(A, B)$ consists of locating and detaching the roots of the two subtrees that contain A and B . But our description of *restore* implies that the procedure locates and detaches in general also other trees, that are not needed in the union. This is necessary in order to maintain a consistent record of the past history encoded in the nested structure of clusters. Detaching only the subtrees of the *BUF* trees that are needed to perform the current union besets the consistency of the clusters that account for those subtrees at that moment. In particular, an edge e could be subtracted from the dominion of some separators without those separators becoming aware of this fact. This would infringe the consistency of the number fields that are affected by the loss of e , thus undermining the consistency of future detachments.

5. Union-Find with unlimited backtracking

In this Section, we show that *BUF* trees support any union or find in $O(\log n / \log \log n)$ worst-case time, and $backtrack(i)$ in constant time, irrespective of i .

We study unions first. In terms of *BUF* trees, $union(A, B)$ transforms the current input forest F of *BUF* trees into a new forest F' that meets the following specifications. First, F' represents, via Invariant 0, the same partition of S as F , except for the fact that A and B are consolidated into a single set. Second, invariants 1-5 still hold on F' . Before proving this, we describe how to support $union(A, B)$.

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 . The management of 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 Invariant 5. To restore this invariant, the procedure must remove from the forest F possibly existing arcs either with first field

or separate field equal to i_{max} . By Invariant 5, there were originally either at most two sibling arcs e' and e'' with first field equal to i_{max} or at most one arc e''' with separate field equal to i_{max} , and such arcs can be accessed in constant time. The procedure deletes these arcs by means of either a $restore(e', i_{max})$ or a $restore(e''', i_{max})$, depending on the case. As a result, the forest F is transformed into an equivalent forest F'' no arc of which is labeled i_{max} . By theorem 4 and theorem 5, F'' still satisfies invariants 0-5, and F'' was 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, at a cost $O(k + h)$ (cf. Invariant 0) in the worst case, where h is the maximum possible length for a path originating at a leaf in a BUF tree and containing only live arcs. Clearly, invariants 0-5 are not affected by this stage.

Next, $union(A, B)$ transforms F'' into an equivalent forest F''' , with the property that $apex(A)$ and $apex(B)$ are live in F''' . Letting e_A and e_B be the arcs respectively leaving $apex(A)$ and $apex(B)$, this involves the two calls $restore(e_A, i_{max})$ and $restore(e_B, i_{max})$. Thus, F''' is produced in $O(k)$ time, and it meets invariants 0-5 because of theorem 4 and theorem 5.

Let now 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 $height(T_B) \leq height(T_A)$. Observe that $height(T_A)$ cannot exceed h , since 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 re-tracking the find that produced r_A for $height(T_B)$ steps. The procedure now selects one of the following three modes of operations, in analogy with a DUF -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 arc entering $parent(v)$. At this point, it is set $first((r_B, parent(v))) = last((r_B, parent(v))) = i_{max}$.

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 arcs will be the rightmost arcs entering v . If v is slim, these arcs will be the rightmost live arcs entering v . The arc s connecting the leftmost child of r_B to v is marked 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 arcs moved. Finally, for every re-directed arc e , $last(e)$ is set to i_{max} .

To complete the management of $union(A, B)$, $fat(v)$ is set to i_{max} if appropriate (cf. type 1 and 3), and a pointer indexed by i_{max} is directed towards the arc(s) (cf. type 1 or 2) or separator

(type 3) introduced by the procedure. The fatness of a node can be tested in $O(k)$ time by a walk starting at its leftmost child (cf. invariant 4).

We now prove the following lemma.

Lemma 3. *Let h be the maximum possible length for a path originating at a leaf in a BUF tree and containing only live arcs. The above implementation of $\text{union}(A, B)$ requires $O(k + h)$ time and preserves invariants 0-5.*

Proof. The time bound is immediate from the above discussion. Therefore, we are left to show that a union preserves all the invariants. Since the three initial possible calls to *restore* preserve the invariant, it is enough to show that the invariants are preserved after any of the three types of manipulations described above. Consider first type 1 and type 2 unions. They introduce new clusters with one and two arcs, respectively.

To see that find consistency is maintained, observe that only nodes in A and B may have their apex changed. If this change occurs, the new apex will be live, and it will consistently store the name of $A \cup B$.

Consider now the $i_{\max}$ invariant. The only arcs for which something has changed are the newly introduced arcs: they are at most two, have last value equal to $i_{\max}$, and are live. Also the last part of invariant 1 still holds. For this, consider first a type 1 union, which introduces only the arc $(\tau_B, \text{parent}(v))$. The field $\text{fat}(\text{parent}(v))$ is unaffected if either $\text{parent}(v)$ was fat or $\text{parent}(v)$ was slim and did not become fat through the union. If, on the other hand, $\text{union}(A, B)$ made $\text{parent}(v)$ fat, then we have seen that it also sets $\text{fat}(\text{parent}(v)) = i_{\max}$, thus preserving the last part of invariant 1. In case of a type 2 union, the new node τ is introduced, and τ is reached by the two arcs $((\tau_A, \tau)$ and $(\tau_B, \tau))$. Node τ is thus slim, and $\text{fat}(\tau)$ is, consistently, undefined. This completes the analysis of Invariant 1 for type 1 or 2 unions.

Consider now the impact on Invariant 2 of type 1 and type 2 unions. A new cluster E with $|E| > 1$ can be created only by a type 2 union. In this case, E does not contain any separator, and, having no siblings, it is vacuously the leftmost cluster. If E has only one arc, then E has been created by a type 1 union. Again E contains no separators.

Invariants 3 and 4 are trivially maintained by a type 2 union, as well as by a type 1 union the new arc introduced by which enters a fat $\text{parent}(v)$. Consider now a type 1 union the new arc introduced by which reaches a slim $\text{parent}(v)$. In this case, the new arc $e = (\tau_B, \text{parent}(v))$ is inserted immediately after the rightmost live arc entering $\text{parent}(v)$, and we have $\text{first}(e) = \text{last}(e) = i_{\max}$. Thus, $\text{first}(e)$ will be larger than the first and separate fields of all of its left siblings, which consist only of live arcs or active separators in force of the slim compression and numbering invariants. Similarly, $\text{last}(e)$ will be larger than the last field of every left sibling of e .

It is easily checked that a type 1 or type 2 union also maintains Invariant 5.

We turn now to type 3 unions. Let $e_i = (x_i, v)$, $1 \leq i \leq h < k$, be the arcs re-directed by $\text{union}(A, B)$ as they appear in the forest F' . Clearly, Invariant 0 is still valid in F' . In fact, the only nodes of F which had their apex changed are the nodes the old apex of which was one of the x_i 's. The procedure provided for these nodes to have a new and consistent live apex.

The arcs $e_1, e_2, \dots, e_h$ are live and have last field equal to $i_{\max}$. Moreover, $\text{separate}(e_1) = i_{\max}$ and e_1 is an active separator. The fat field of node v is correctly updated following a type 1 or

type 3 union. Thus, Invariant 1 is preserved.

A type 3 union introduces one new cluster $E = \{e_1, e_2, \dots, e_h\}$, by ordered aggregation of the clusters of edges entering a slim node of F''' . In the new cluster E , the leftmost edge e_1 is made a separator. Furthermore, the last field of all the arcs in E will be set to i_{max} , and therefore $\text{last}(E) = \text{separate}(e_1)$ as required by invariant 2. Since Invariant 2 was valid in F''' for each one of the individual clusters contributing to E , then $\text{number}(e_1) = |E|$. Thus, E satisfies Invariant 2. Reasoning along the same lines, it is easy to check that E will satisfy also the invariants 3 and 4.

The first part of Invariant 5 is preserved by the first call to *restore*, while the rest of this invariant follows from the validity of Invariant 3 at the inception of the union. •

We now focus on the *BUF*-tree implementation of backtracks.

Lemma 4. *For any values of i_{max} and i , $\text{backtrack}(i)$ can be performed on a forest of *BUF* trees in constant time, preserving invariants 0-5.*

Proof. As said, $\text{backtrack}(i)$ is performed by setting $i_{max} = \max(i_{max} - i, 0)$, i.e., in constant time for any value of i . Hence, we only need to prove that $\text{backtrack}(i)$ maintains invariants 0-5. Since the effect of a backtrack is null unless the value of i_{max} is altered, we can safely assume $i_{max} - i \geq 0$. Then, we may regard a $\text{backtrack}(i)$ as a sequence of i consecutive $\text{backtrack}(1)$, and we only need to prove that, if invariants 0-5 were valid before performing a $\text{backtrack}(1)$, they are still valid immediately afterwards. To fix the ideas, let i_{max}^{old} and $i_{max}^{new} = i_{max}^{old} - 1$ be the values of i_{max} immediately prior to and after $\text{backtrack}(1)$, respectively.

We distinguish two cases, depending on the type of union undone. Let u be this union and let A and B the two sets unified by u .

If u is a type 1 or 2 union, then u introduced at most two arcs, and such arc(s) are made now dead by the backtrack. Assume for generality that two arcs, say $e_1 = (x_1, v)$ and $e_2 = (x_2, v)$, were introduced by u . Clearly, $\text{first}(e_1) = \text{first}(e_2) = i_{max}^{old} > i_{max}^{old} - 1 = i_{max}^{new}$, hence these arcs become consistently dead. If e_1 and e_2 were cheating, then their death did not affect the fatness of v . If they were live, then v may have become, from fat, slim. But this implies that u was the last surviving union which made v fat, whence after backtrack $\text{fat}(v) = i_{max}^{old}$ exceeds $i_{max} = i_{max}^{new}$. Since no arc other than e_1 and e_2 is affected by this backtrack, this guarantees the validity of invariant 1.

Invariant 0 is also preserved. In fact, the only leaves the apex of which was possibly changed are those ending up with apex at either x_1 or x_2 . The union operation u which the backtrack is voiding, however, did not delete the old names A and/or B stored in these nodes. The leaves in the subtrees rooted at x_1 and/or x_2 are thus given back the old name A and/or B . This consistently reflects that u was made void.

The slim compression invariant is propagated by the validity, prior to $\text{backtrack}(1)$, of the Invariant 4 itself and of the part of Invariant 3 that concerns the R -numbers of clusters entering slim nodes. No part of Invariant 2, 3 or 5 is affected by a backtrack operation, so that these invariants are maintained too.

Assume now that u is of type 3, and let $e_1 = (x_1, v), e_2 = (x_2, v), \dots, e_h = (x_h, v)$, with $h \leq k$, be the arcs issued by u . By hypothesis, the last field of these arcs was equal to i_{max}^{old} prior

to backtrack, and therefore is strictly larger than $i_{\max}^{\text{new}} = i_{\max}^{\text{old}} - 1$. Hence, these arcs become consistently cheating (recall that, by Invariant 5, the first field of each arc in a cluster is strictly smaller than the last field of that arc). Since these arcs were live, then v may become, from fat, slim. This means that u was the last surviving union which made v fat, and therefore $\text{fat}(v) = i_{\max}^{\text{old}}$ is now greater than $i_{\max}^{\text{new}}$. This settles Invariant 1.

Clearly, only the leaves that, prior to operation u , had apex at one of the x_i 's are affected by the backtrack. By the structure of a type 3 union, however, the name of each such leaf was stored in $\text{label}(e_1)$ as part of the execution of u . Having assumed Invariant 3 valid prior to the backtrack, we are guaranteed that, afterwards, e_1 is the rightmost inactive separator to the left of each e_i , $1 \leq i \leq h$. Thus, Invariant 0 is preserved. Slim compression descends from the validity of Invariants 2, 5, and Invariant 4 itself prior to $\text{backtrack}(1)$, while Invariants 2, 3 and 5 are all maintained vacuously. •

In order to prove our claimed time bounds, we show now that, at any time in a *BUF* tree forest, the length of a path consisting of live arcs cannot exceed $O(\log_k n)$. This clearly establishes our bound for finds, and it combines with lemma 3 to yield an identical bound for the union. Our desired property shall follow from the following lemma.

Lemma 5. *At any time and for every arc $e = (x, v)$ in a BUF-tree forest, if x is not a leaf then $\text{fat}(x)$ is defined and, moreover, $\text{fat}(x) < \text{first}(e)$.*

Proof. We proceed by induction on the number of operations performed. Initially, there are n singleton trees and the claim holds vacuously, since there are no arcs and no fat nodes in the structure.

Assuming now that the claim holds before the i -th operation, $i \geq 1$, we prove that it holds also afterwards. The proof is straightforward in the case of finds and backtracks, since these operations do not alter any of the parameters in the claim. Thus, we concentrate on unions.

Let then the i -th operation be $\text{union}(A, B)$ where A and B are two arbitrary sets in the current partition of S .

We first show that the procedure *restore* preserves the property of the claim. To see this, let $E = e_1, e_2, \dots, e_h$, $1 \leq h < k$, be the cluster of arcs managed by a *restore*. As we know (cf. Invariant 1 and the definition of cluster), either all arcs in E are live or none is. Since *restore* does nothing on live arcs, we concentrate on the case where E contains a mixture of dead and cheating arcs. We need to show that the claim holds after *restore* for the arcs in E only, since every other arc or node was not affected.

Let x_i ($1 \leq i \leq h$) be the node from which the arc e_i originated immediately prior to *restore*. By inductive hypothesis, $\text{fat}(x_i) < \text{first}(e_i)$ ($i = 1, 2, \dots, h$). We now distinguish two cases for each arc e_i in E , depending on whether e_i is dead or cheating. If arc e_i is dead, then *restore* simply deletes e_i , leaving a *BUF* tree rooted at x_i . The nodes of such a *BUF* tree still satisfy the invariant, by the inductive hypothesis. If, on the other hand, arc e_i is cheating, then by Invariant 1 $\text{first}(e_i) \leq i_{\max}$. By assumption, $\text{fat}(x_i) < \text{first}(e_i)$, so that $\text{fat}(x_i) < i_{\max}$. As a consequence, the union who made x_i fat is still valid, and therefore x_i is still fat. The procedure *restore* re-directs

e_i to a new node, as explained in its description, which does not modify $fat(x_i)$ and $first(e_i)$. Since $fat(x_i)$ and $first(e_i)$ remain unchanged for each re-directed cheating arc e_i , the claim holds after *restore* for every such arc. The nodes introduced by *restore* are slim nodes, and thus do not have a defined fat field. Clearly, no arc leaves such newly created nodes. In conclusion, *restore* maintains our claim.

Recall that *restore* is called for three times at the beginning of a union, the second and third time in order to produce the two trees T_A and T_B . We need now to show that the unification of T_A and T_B preserves the claim. Let r_A and r_B be the respective roots of T_A and T_B , and let i_{max} be the ordinal number of the present union. As usual, we distinguish three types of unions.

If a type 1 union is performed, then r_B is fat and therefore by Invariant 1 $fat(r_B) < i_{max}$. A new arc e leaving from r_B is introduced, and $first(e)$ is set to i_{max} . As a consequence, $fat(r_B) < first(e)$. Since this is the only change in the data structure, the claim is maintained.

If a type 2 union is performed, then r_A and r_B are both fat. Therefore, $fat(r_A) < i_{max}$ and $fat(r_B) < i_{max}$. The only change in the data structure is that a new node r and two new arcs (r_A, r) and (r_B, r) are introduced. Since $first(r_A, r) = first(r_B, r)$ is set to i_{max} , then $fat(r_A) < first(r_A, r)$ and $fat(r_B) < first(r_B, r)$. Hence, the claim is maintained.

If a type 3 union is performed, then either r_A is slim or r_B is slim. Assume to fix the ideas that r_B is slim. Then, at most $k - 1$ nodes (i.e., all the children x of r_B for which the arc (x, r_B) is live) are given a new parent v , but neither the first field of the re-directed arcs (x, r_B) nor the fat field of the previous children of r_B is affected. As a consequence, we have only to check that the node v still fulfills the claim. If v was fat, then $fat(v) < i_{max}$, and, by inductive hypothesis, the arc leaving v (if any) had first field greater than $fat(v)$. If v was slim and there is no arc leaving v , then the claim will trivially still hold for v . Assume now that v is slim and there is an arc e leaving v . Then we claim that e must be dead. In fact, assume by contradiction that e is either live or cheating. This implies that, because of Invariant 1, $first(e) < i_{max}$, and because of the inductive assumption, $fat(v) < first(e) < i_{max}$, which contradicts the hypothesis of v being slim. Therefore, the arc e leaving v must be dead. By Invariant 1, this is equivalent to saying that $first(e) > i_{max}$. If the new children of v do not make it fat, then $fat(v)$ remains unchanged and the claim still trivially holds for v by propagation of the inductive hypothesis. On the other hand, if due to the type 3 union being performed v becomes fat, then $fat(v)$ changes to i_{max} . But since e is dead, then $first(e) > i_{max}$ and therefore $fat(v) < first(e)$. As a consequence, v will fulfill the claim in this case, too.

This complete the induction step of the union operation and establishes the lemma. •

Remark: The crucial implication of lemma 5 is that live arcs can only originate from either leaves or fat nodes. Therefore, in any path composed only of live arcs, only the node at the top can be slim. Since, by definition, all arcs traversed by $find(x)$ except the last one are live, it follows that a $find(x)$ encounters only fat nodes on the path from x to $apex(x)$, with the only possible exception of $apex(x)$ itself. Let then $x = v_0, v_1, v_2, \dots, v_{h-1}, v_h = apex(x)$ be the ordered sequence of nodes visited by a generic $find(x)$ while climbing a path of length h up to $apex(x)$. Then, v_{h-1} has at least k live edges entering it, i.e., at least k fat children. Iterated application of lemma 5 to these

children, their own children, and so on, yields that there are at least k^{h-1} leaves connected to v_{h-1} by means of paths consisting only of live edges. Since $k^{h-1} \leq n$, it follows that h is $O(\log_k n)$.

Theorem 6. *BUF trees support each union and find in $O(\log n / \log \log n)$ time, and backtrack(i) in $O(1)$ time irrespective of i . The overall space required is $O(n)$.*

Proof. The time bounds follow from lemma 3, lemma 4 and lemma 5. The space complexity of the data structure is dictated by the maximum number of arcs that may be present in it at any given time. New arcs are introduced only by unions, and each union can introduce at most two arcs. However, we have seen that when a union getting ordinal number i is performed, the arcs possibly created by a past union with the same ordinal number are removed from the data structure. This guarantees that, at any time, at most $2(n-1)$ arcs may exist in the data structure. If persisting nodes are removed as soon as there are no edges entering them, then the total space required by the data structure is $O(n)$. •

6. Conclusion

We have introduced a data structure for the efficient management of set union with unlimited backtracking. Our approach stays within the guidelines of separable pointer algorithms, if one only relaxes the separability condition to an extent that is deemed acceptable [23]. Our per-operation worst-case bounds are tight both for this model as well as for the more powerful cell probe model of computation.

BUF trees represent also a partially persistent [3] data structure to be used in the following variant of the set union problem. In this variant, the set union is defined as usual, but a find operation is formatted as $find(x, k)$, where x is the name of an element of S and k is a nonnegative integer, not exceeding the ordinal number of the last union so far performed. The task of $find(x, k)$ is to return the name of the subset which contained the element x at the time when only the first k unions had been performed. To perform a $find(x, k)$ on a BUF tree, it is sufficient to temporarily set i_{max} to $i_{max} - k$, and then proceed as per an ordinary $find(x)$.

Acknowledgments

We are grateful to Moti Yung for his helpful comments in reading an earlier draft of this paper.

References

- [1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, "*The design and analysis of computer algorithms*", Addison-Wesley, 1974.
- [2] N. Blum, "On the single operation worst-case time complexity of the disjoint set union problem", *SIAM J. Comput.* 15 (1986), 1021–1024.
- [3] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan, "Making data structures persistent", *J. Comput. Sys. Sci.* 38 (1989), 86–124.
- [4] M. L. Fredman, "On the cell probe complexity of the set union problem", Tech. Rep. TM-ARH-013-570, Bell Communications Research, January 31, 1989.
- [5] M. L. Fredman, M. Saks, "The cell probe complexity of dynamic data structures", *Proc. 21st ACM Symp. on Theory of Computing*, 1989, 345–354.
- [6] G. Gambosi, G. F. Italiano, M. Talamo, "Getting back to the past in the union find problem", *Proc. 5th Symp. on Theoretical Aspects of Computer Science*, 1988, 8–17.
- [7] G. Gambosi, G. F. Italiano, M. Talamo, "The set union problem with dynamic weighted backtracking", *Proc. 26th Annual Allerton Conf. on Communication, Control and Computing*, 1988.
- [8] C. J. Hogger, "*Introduction to logic programming*", Academic Press, 1984.
- [9] J. E. Hopcroft, J. D. Ullman, "Set merging algorithms", *SIAM J. Comput.* 2, (1973), 294–303.
- [10] T. Ibaraki, "M-depth search in branch and bound algorithms", *Int. J. Comp. Inform. Sc.*, 7, (1978), 313–373.
- [11] D. E. Knuth, "*The Art of Computer Programming, Vol. 1, Fundamental Algorithms*", Addison-Wesley, Reading, MA, 1968.
- [12] H. Mannila, E. Ukkonen, "The set union problem with backtracking", *Proc. 13th Int. Colloquium on Automata, Languages and Programming*, 1986, 236–243.
- [13] H. Mannila, E. Ukkonen, "On the complexity of unification sequences", *Proc. 3rd Conf. on Logic Programming*, 1986, 122–133.
- [14] H. Mannila, E. Ukkonen, "Time parameter and arbitrary deunions in the set union problem", *Proc. 1st Scandinavian Workshop on Algorithm Theory*, 1988, 34–42.
- [15] K. Mehlhorn, S. Naher, H. Alt, "A lower bound for the complexity of the union-split-find problem", *SIAM J. Comput.* 17 (1988), 1093–1102.
- [16] J. Pearl, "*Heuristics*", Addison-Wesley, Reading, MA, 1984.
- [17] A. Schönhage, "Storage modification machines", *SIAM J. Comput.* 9 (1980), 490–508.

- [18] R. E. Tarjan, "Efficiency of a good but not linear set union algorithms", *J. Assoc. Comput. Mach.* 22 (1975), 215–225.
- [19] R. E. Tarjan, "A class of algorithms which require non linear time to maintain disjoint sets", *J. Comp. Syst. Sci.* 18 (1979), 110–127.
- [20] R. E. Tarjan, "Amortized computational complexity", *SIAM J. Alg. Disc. Meth.* 6 (1985), 306–318.
- [21] R. E. Tarjan, J. van Leeuwen, "Worst-case analysis of set union algorithms", *J. Assoc. Comp. Mach.* 31 (1984), 245–281.
- [22] D. H. D. Warren, L. M. Pereira, "Prolog – the language and its implementation compared with LISP", *ACM SIGPLAN Notices* 12 (1977), 109–115.
- [23] J. Westbrook, R. E. Tarjan, "Amortized analysis of algorithms for set union with backtracking", *SIAM J. Comput.* 18 (1989), 1–11.

(a)

(b)

Figure 1. BUF trees and Dominance trees.

a) The BUF tree produced, for $k > 10$, by applying to an initial singleton partition the ordered sequence of unions: $\text{union}(A,B)$, $\text{union}(H,I)$, $\text{union}(L,M)$, $\text{union}(C,D)$, $\text{union}(E,F)$, $\text{union}(F,G)$, $\text{union}(F,I)$, $\text{union}(D,G)$, $\text{union}(A,E)$, $\text{union}(B,L)$. The first two numbers at the bottom right of each arc represent, respectively, the *first* and *last* field for that arc. Separators are also labeled with a third number, representing their *separate* field. This sequence produces 3 clusters and the 4 separators (C,X) , (E,X) , (H,X) and (L,X) .

b) Dominance trees associated with the clusters of Figure 1(a). The number on the left of node v represents $r(v)$.

Figure 2 : Illustrating Lemma 1

Figure 3

The effect of a dismember with $i_{max} = 5$ on the second cluster of Fig. 1 (b).