Fully Persistent Data Structures for Disjoint Set Union Problems

Giuseppe F. Italiano [*] Neil Sarnak [†]

Abstract

We consider the problem of maintaining a partition of n elements of disjoint sets under an arbitrary sequence of union and find operations in a fully persistent fashion. We show how to access (i.e., perform a find operation) and modify (i.e., perform a union operation) each version of the partition in $O(\log n)$ worst-case time and in $O(1)$ amortized space per update. No better bound is possible for pointer based algorithms.

1 Introduction

In this paper we study persistence in disjoint set union data structures. Following Driscoll et al. [8], we define a data structure to be ***ephemeral*** when an update destroys the previous version. A ***partially persistent*** data structure supports access to multiple versions, but only the most recent version can be modified. A data structure is said to be ***fully persistent*** if every version can be both accessed and modified.

The disjoint set union problem consists of maintaining a collection of disjoint sets under an intermixed sequence of the following two operations.

union(A, B) : Combine the sets named A and B into a new set named A;

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

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

The best algorithms to solve this problem are due to Tarjan and van Leeuwen [15, 18]. These algorithms require $O(n)$ space and $O(k\alpha(k + n, n) + n)$ time, where k is the number of find operations performed and α is a very slowly growing function, the functional inverse of Ackermann's function [18]. No better bound is possible for pointer based algorithms [4, 11, 16], and in the cell probe model of computation [9].

While partially persistent data structures for disjoint set union problems have been already proposed in the literature [2, 13], no non-trivial fully persistent data structures for this problem are known. The main difficulty is that the most efficient set union algorithms use linked data

[*]Department of Computer Science, Columbia University, New York, NY 10027 and Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy. Supported in part by an IBM Graduate Fellowship and by the ESPRIT Basic Research Action Program of the European Communities under contract No. 3075 (Project ALCOM). Work done while visiting the IBM T. J. Watson Research Center.

[†]IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598.

structures whose nodes have degrees not necessarily bounded by a constant (see for instance [18]); and unfortunately the techniques proposed by Driscoll et al. [8] to make data structures persistent are not applicable in this case.

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

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

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

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

In the remainder of this paper, we denote by m the total number of Union operations performed. Before stating our bounds, let us review some trivial solutions to the fully persistent disjoint set union problem. If we store each version of the partition explicitly, then each Find can be performed in $O(1)$ time and each Union in $O(n)$ time, at the expense of $O(n)$ storage per update. If we do not store all the versions but just the sequence of Union operations, then each Union can be performed in $O(1)$ time and $O(1)$ space. However, a Find requires now $O(m)$ time. With a little more care, this time can be reduced to $O(n)$.

We show how to perform Union and Find operations in any version in $O(\log n)$ [1] worst-case time and $O(1)$ amortized space for update. No better pointer based algorithm is possible, since in this setting Mannila and Ukkonen [13] showed a lower bound of $\Omega(\log n)$ amortized time for the partially persistent disjoint set union problem.

We then consider a variant of the above problem, called the ***fully persistent disjoint set union problem with deletions***, and in which besides Unions and Finds we allow deletions of an element from a set:

Delete(x,v) : Denote by X the set in version v containing element x. If $|X| = 1$ then do nothing. Otherwise, create a new version of the partition by deleting x from X. In the new version, element x is in a set by itself and the name of set $\{x\}$ is x. The name of set $X' = X - \{x\}$ in the new version is defined as follows. If the name of X was not x, then X' gets the same name as X. Otherwise X' gets as a new name arbitrarily one of its elements.

We show that each Union, Find and Delete operation can be supported in $O(\log n)$ worst-case time and $O(1)$ space per update.

Motivations for studying these problems arise in several application areas. For instance modern high level languages such as Hermes [3] and SETL [6] support sets as basic data types and need fast set manipulation primitives on multiple versions of the same set. Another application is logic programming memory management [12, 19]. In Prolog, for example, variables of clauses correspond to elements of a set, and unifications imply disjoint set union operations. In this case, the availability of multiple versions of disjoint sets allows one to support backtracking and branching techniques, and to trace different program executions at the same time.

[1]All the logarithms are assumed to be to the base 2 unless explicitly specified otherwise.

The remainder of this paper consists of three sections. Section 2 shows how to make set union data structures fully persistent in $O(\log^2 n)$ time and $O(1)$ amortized time per update. We improve the time bound to $O(\log n)$ and show how to deal with Delete operations in section 3. Section 4 contains some concluding remarks.

2 An $O(\log^2 n)$ Fully Persistent Data Structure

In this section we describe techniques to make set union data structures fully persistent.

To perform Union and Find operations, we use the basic set union tree data structures (see for instance [18]), which represent sets making use of rooted trees. Each tree corresponds to a set. Nodes of the tree correspond to elements of the set. The name of the set is stored in the root of the tree, which is referred to as the *canonical element* of the set. Each node of the tree points to its parent, except for the root that points to itself.

We recall that in such an ephemeral data structure, a find(x) can be performed by starting from the node x and by following the pointer to the parent until the tree root (i.e., the canonical element) is reached. The name of the set stored in the canonical element is then returned. Therefore, the time bound of a find operation is proportional to the longest path from a leaf to its root in a set union tree. To keep those paths short, one of the following two *union rules* can be applied while performing a union operation.

union by size : make the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires maintaining the number of descendants for each node throughout all the sequence of operations.

union by rank : make the root of the shallower tree point to the root of the other, arbitrarily breaking a tie. This requires maintaining the height of the subtree rooted at each node, in the following referred to as the *rank* of a node, throughout all the sequences of operations.

Using either *union rule* yields trees of path length no greater than $O(\log n)$. In the following, we describe our data structure when *union by size* is used, but the same results can be easily extended to the case of *union by rank.*

As mentioned in the introduction, we cannot apply the techniques of Driscoll et al. [8] to make the forest of set union trees fully persistent. Indeed, as a consequence of how union operations are carried out, the in-degree of a node in a set union tree is not necessarily bounded by a constant. Our solution is as follows. Throughout the sequence of operations, we maintain a fully persistent balanced search tree $\mathcal{T}$. Version v of $\mathcal{T}$ contains information about version v of the forest of set union trees, as follows. Each version of $\mathcal{T}$ contains n items $1, 2, \ldots, n$ corresponding to the n elements of the disjoint sets. For $1 \leq x \leq n$, item x in version v of $\mathcal{T}$ has associated three fields: *parent*, *size*, and *name*. We will refer to those three fields as $parent(x, v)$, $size(x, v)$ and $name(x, v)$ respectively. Field $parent(x, v)$ stores the parent of element x in version v of the forest of set union trees, if x is a non-root node; otherwise if x is a root, $parent(x, v) = x$. Field $size(x, v)$ is equal to the size of the subtree rooted at x in version v of the forest of set union trees. Finally, $name(x, v)$ is defined only if x is a root in version v of the forest of set union trees, and contains the name of the set; if x is not a root in version v, then $name(x, v)$ contains a special `null` value. Version 0 of $\mathcal{T}$ is initialized as follows. For each element x, $1 \leq x \leq n$, $parent(x, 0) = x$, $size(x, 0) = 1$ and $name(x, 0) = x$. This corresponds to version 0 of the set union data structure and can be done in $O(n)$ time. The operations defined on the fully persistent search tree $\mathcal{T}$ are: *Lookup*(i, v) which returns the three fields associated with element i in version v (namely $parent(i, v)$, $size(i, v)$, and $name(i, v)$), and *Store*(i, f_1, f_2, f_3), which creates a new version of the balanced search tree by storing values f_1, f_2 and f_3 in the three fields of element i.

We now describe how to perform Union and Find operations. To perform a Find(x, v), we use the same algorithm as in the ephemeral data structure; that is we follow the path from x to its root in version v of the forest of set union trees. However, now we have to be able to navigate in version v of the forest of set union trees, disregarding all Union operations with no influence on version v. We do this with the help of the fully persistent balanced search tree $\mathcal{T}$ as follows. We start by computing the parent of x by performing a Lookup(x, v) operation on $\mathcal{T}$. If x is a root (i.e., if the *parent* field returned by the Lookup operation is equal to x) we stop and output the canonical information retrieved. Otherwise, we repeat the same operation for the parent of x, the grandparent of x and so forth until the root is reached. More formally, we can describe the algorithm using the following pseudo-code.

```
Procedure Find($x, v$);
begin
    $\langle parent, size, name \rangle \leftarrow$ Lookup($x, v$);
    $y \leftarrow x$;
    while $y \neq parent$ do begin
        $y \leftarrow parent$;
        $\langle parent, size, name \rangle \leftarrow$ Lookup($y, v$)
    end;
    return($\langle parent, size, name \rangle$)
end;
```

The correctness of the Find operation hinges upon our ability to update the fully persistent balanced search tree $\mathcal{T}$ after a Union(x, y, v) operation. This can be done by first retrieving the size of the sets containing elements x and y in version v and then by merging those two sets by means of a union by size:

```
Procedure Union($x, y, v$);
begin
    $\langle root_x, size_x, name_x \rangle \leftarrow$ Find($x, v$);
    $\langle root_y, size_y, name_y \rangle \leftarrow$ Find($y, v$);
    if $root_x \neq root_y$ then
        if ($size_x \geq size_y$) then begin
                Store($root_x, root_x, size_x + size_y, name_x$);
                Store($root_y, root_x, size_y$, null)
            end
        else begin
                Store($root_y, root_y, size_x + size_y, name_x$)
                Store($root_x, root_y, size_x$, null);
            end
end;
```

Lemma 2.1 *Procedure Find(x, v) correctly returns the canonical element of the set containing x in version v by performing at most $O(\log n)$ Lookup operations on $\mathcal{T}$.*

Proof: Let $\mathcal{F}_v$ be the version v of the forest of set union trees built through Union and Find operations. We remark that $\mathcal{F}_v$ is not maintained explicitly by our algorithm. It can be easily proved by induction on the number of operations that, for each element x and version v, the fields *parent*(x, v), *size*(x, v) and *name*(x, v) correspond respectively in $\mathcal{F}_v$ to the parent of x, to the size of the subtree rooted at x, and to the name of the set containing x if x is a tree root. As a consequence, all the Union operations correctly follow the union by size rule. Thus, in any version of the set union data structure, the path from a leaf to its root is of length at most $\log n$. Therefore, each Find operation performs at most $O(\log n)$ Lookup operations on $\mathcal{T}$. □

Theorem 2.1 *The fully persistent disjoint set union problem can be solved in $O(\log^2 n)$ worst-case time per operation, and in $O(1)$ amortized space per update.*

Proof: Denote by $A(n)$ and $S(n)$ the time required to perform respectively a Lookup and a Store operation on a fully persistent balanced search tree $\mathcal{T}$. A Union operation requires $O(S(n))$ time plus the time required by two Find operations. To bound the time required by a Find, we notice that as a consequence of Lemma 2.1 Find(x, v) examines at most $O(\log n)$ different nodes in the path from x to the root. Therefore, a Find operation can be supported in $O(A(n) \log n)$ time. Choosing the fully persistent balanced search trees of Driscoll et al. [8] gives $A(n) = S(n) = O(\log n)$ in the worst case, giving the $O(\log^2 n)$ worst-case time per operation. With this implementation of $\mathcal{T}$, the space is $O(1)$ amortized per update. □

By using the fully persistent array of Dietz [7] to support Lookup and Store operations, we can achieve a slightly better expected bound:

Corollary 2.1 *The fully persistent disjoint set union problem can be solved in $O(\log n \log \log m)$ expected amortized time per operation, where m is the total number of Union operations performed. The space required is $O(1)$ amortized per update.*

We recall that we do not create a new version of our fully persistent data structure each time a Union(x, y, v) with x and y being already in the same set in version v is executed (i.e., we do not count such a Union as an actual operation). Therefore $m \leq O(n!)$, and $\log \log m \leq O(\log n)$. This implies that the bounds in Corollary 2.1 are never worse than the bounds in Theorem 2.1.

The time bounds can still be improved by making use of a more efficient set union tree data structure proposed by Blum [5] and called a *k-UF* tree. This data structure is able to support each union and find operation in the ephemeral data structure in $O(\frac{\log n}{\log \log n})$ time in the worst case. We need to know the following facts about *k-UF* trees, and we refer the reader to [5] for all the other details of the method.

For any $k \geq 2$, a *k-UF* tree is either a singleton node or a rooted tree T such that the root has at least two children, each internal node has at least k children, and all leaves are at the same level. As a consequence of this definition, the height of a *k-UF* tree with n leaves is not greater than $\lceil \log_k n \rceil$. We refer to the root of a *k-UF* tree as *fat* if it is either a singleton node or it has more than k children, and as *slim* otherwise. A *k-UF* tree is said to be *fat* if its root is fat, otherwise it is referred to as *slim*. Disjoint sets can be represented by *k-UF* trees as follows. Each set is a *k-UF* tree. The elements of the set are stored in the leaves of the tree. There is a canonical element for each set storing the name of the set; once again the canonical element is the tree root. Furthermore, the root also contains the height of the tree and a bit specifying whether it is fat or slim.

A find(x) is performed as described before by starting from the leaf containing x and returning the name stored in the root. This can be accomplished in $O(\log_k n)$ worst-case time. A union(A, B) is performed by first accessing the roots r_A and r_B of the corresponding *k-UF* trees T_A and T_B. Blum assumed that his algorithm obtained r_A and r_B in constant time 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 w be the node on the path from the leftmost leaf of T_A to r_A with the same height as T_B. Clearly, w can be located by following the leftmost path starting from the root r_A in 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 and w is not the root of T_A. Then r_B is made a sibling of w.

Type 2 - Root r_B is fat and w 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 w is equal to r_A and slim. If root r_B is slim, then all the children of r_B are made the leftmost children of w. Otherwise, w is equal to r_A and is slim. In this case, all the children of $w = r_A$ are made the rightmost children of r_B.

Figure 1 shows the three different types of unions in *k-UF* trees. Since each union involves traversing a path in a *k-UF* tree and re-directing at most k pointers (for type 3 unions), it can be supported in $O(\log_k n + k)$ time. Choosing $k = \left\lceil \frac{\log n}{\log\log n} \right\rceil$, gives a bound of $O(\frac{\log n}{\log\log n})$ time for each union and find. We can make Blum's data structure fully persistent with the following bounds.

Theorem 2.2 *The fully persistent disjoint set union problem can be solved either in* $O(\frac{\log^2 n}{\log\log n})$ *worst-case time per operation or in* $O(\frac{\log n \log\log m}{\log\log n})$ *expected amortized time per operation and in* $O(\frac{\log n}{\log\log n})$ *amortized space per update.*

Proof: We follow the same approach as in the proof of Theorem 2.1. Namely we maintain a fully persistent balanced search tree T, such that version v of T stores version v of the forest of *k-UF* trees. Once again, let us denote by $A(n)$ and $S(n)$ respectively the times required to perform a Lookup and a Store operation in a fully persistent balanced search tree of size n. Each time we make a structural change in the forest of *k-UF* trees, the change is recorded with a Store operation in T. Information about version v of the forest of *k-UF* trees can be retrieved by means of Lookup operations on T. However, now there are two differences with the basic set union tree data structure used in the proof of Theorem 2.1.

The first difference is that the size of T is no longer n. Indeed the forest of *k-UF* trees contains n leaves plus the internal nodes. However, the number of internal nodes is bounded above by n due to the properties of *k-UF* trees. As a result, the size of T will be $2n$ instead of n. There is a further complication, since we need to have entries in T also for those internal nodes. But this can be easily handled, since we can assign a unique integer in $[n, 2n]$ to an internal tree node as soon as it is created by simply incrementing a counter.

The second difference is that now a union operation in the ephemeral data structure can change as many as $O(\frac{\log n}{\log\log n})$ pointers, which causes $O(\frac{\log n}{\log\log n})$ Store operations in T. As a result, the time for a Union operation becomes $O(\frac{\log n}{\log\log n} S(n))$.

Due to the properties of *k-UF* trees, the bound for a Find operation becomes $O(\frac{\log n}{\log\log n} A(n))$. Choosing the fully persistent trees of Driscoll et al. [8] for the implementation of T gives the $O(\frac{\log^2 n}{\log\log n})$ time bound. If Store and Lookup operations are supported with a fully persistent array instead, we achieve an $O(\frac{\log n \log\log m}{\log\log n})$ expected amortized bound per operation. In either case the space required becomes $O(\frac{\log n}{\log\log n})$ per update, since there can be that many Store operations required in the worst case by a Union(x, y, v). □

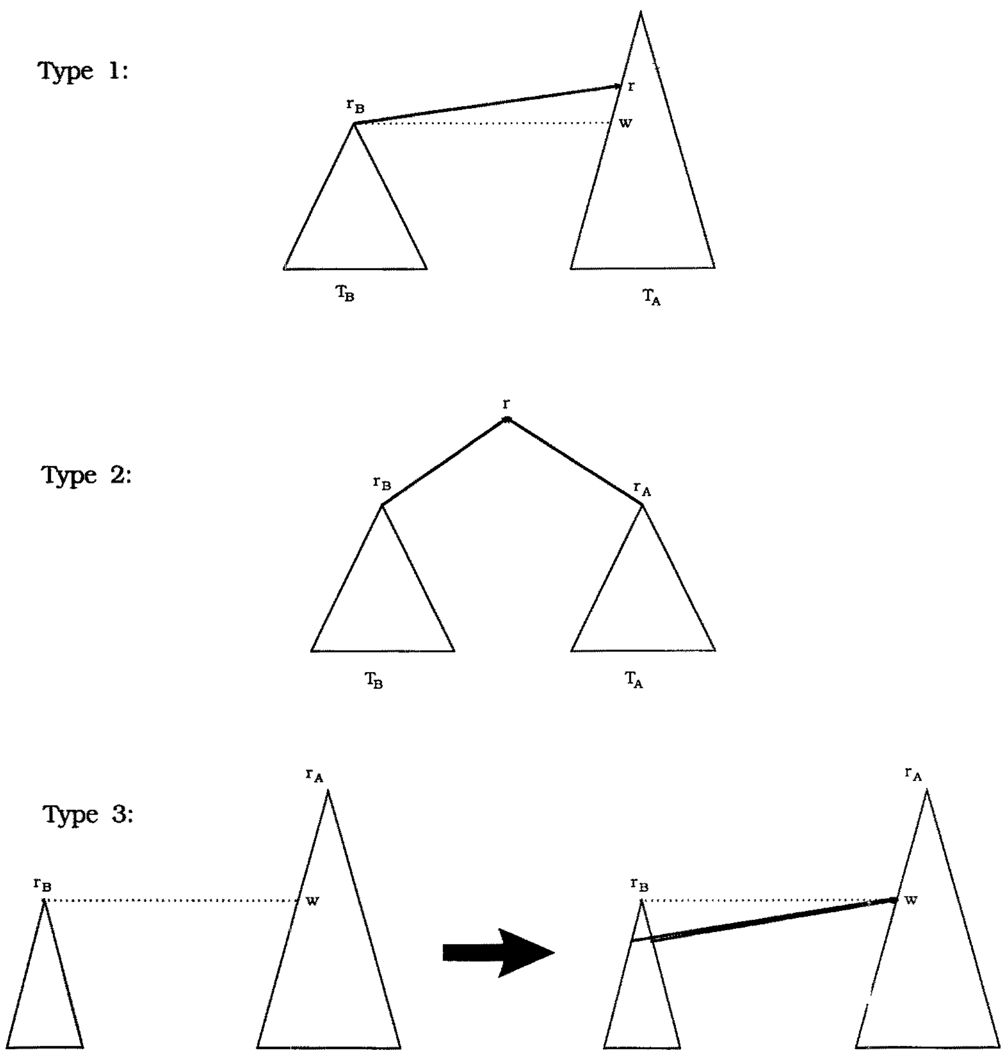

Figure 1: The three types of unions in a *k-UF* tree

3 An Improved Fully Persistent Data Structure

In this section we show how to improve to $O(\log n)$ the time bound for the fully persistent disjoint set union problem. In order to achieve this improvement, we do not use an extra persistent data structure such as the balanced search tree or the array used in the previous section. Rather, we store the disjoint sets in a different ephemeral data structure, and make this data structure persistent.

The ephemeral data structure we use can be described as follows. An (a,b)-tree, for $b \geq 2a$, is a tree with all the leaves at the same level and such that each node but the root has at least a and at most b children. The root has at least 2 and at most b children. In what follows, we denote by $\delta(v)$ the degree of a node v. We represent each disjoint set by an (a,b)-tree, with the elements of the set stored in the leaves. The canonical element storing the name of the set is again the root of the tree. As usual, a find(x) in this ephemeral data structure can be performed by following the path from the leaf corresponding to element x to the tree root, in order to return the name stored there. This requires $O(\log n)$ time. A union(A,B) can be performed by simply concatenating the two (a,b)-trees storing sets A and B, plus some extra bookkeeping to store the name of the new root. A concatenate(T_1,T_2,T) takes two (a,b)-trees T_1 and T_2 and produces a new (a,b)-tree T which has all the leaves of T_1 to the left of the leaves of T_2. The relative order of the leaves previously in T_1 (and T_2) stays unchanged. This can be done in $O(\log n)$ by using a well known algorithm (see for instance [1]).

Huddleston and Mehlhorn [10] showed how to support any sequence of k insertions and deletions in an (a,b)-tree starting from an empty tree in a total of $O(k)$ time. This result can be actually extended to concatenate operations. Before showing this, we need to describe the concatenate algorithm. Let h_1 and h_2 be respectively the heights of T_1 and T_2. Let ρ_1 be the rightmost leaf in T_1 and let σ_2 the leftmost leaf in T_2. With some additional bookkeeping in the root of (a,b)-trees we can access its leftmost and rightmost leaf in constant time. We climb up from ρ_1 and σ_2 towards their roots alternating among them one step at the time. We stop when the first root is entered. Without loss of generality assume that $h_1 \geq h_2$ (otherwise interchange the role of T_1 and T_2 in what follows). Let r_1 and r_2 be respectively the roots of T_1 and T_2, and let v_1 be the node of T_1 in the path from ρ_1 to r_1 at the same height as r_2. T_2 can be implanted in T_1 as follows.

If $\delta(v_1)+\delta(r_2) \leq b$, we combine v_1 and r_2 into one node and stop. Otherwise, we share children among v_1 and r_2 so as to satisfy the degree constraint; then we insert an edge from r_2 to the parent of v_1. We refer to this as a ***sharing step.*** Let x be the parent of v_1. Because of the sharing step, the degree of x is increased by 1. If it is still no greater than b, we stop. Otherwise, $\delta(x) = b+1$. We create a new node x' and move to it the $\left\lceil \frac{b+1}{2} \right\rceil$ rightmost children of x. Then we insert an edge from x' to the parent of x. We call this a ***splitting step.*** We apply the splitting step recursively while going up to the root until either the degree constrained is satisfied or we reach the root r_1. If also the root has to be split, we create a new root r of degree 2. Since each sharing and splitting step can be implemented in constant time, the total cost of concatenate is $O(\min\{h_1,h_2\}+\ell)$, with ℓ being the number of splitting steps applied.

Lemma 3.1 *Each concatenate operation on (a,b)-trees can be performed in $O(1)$ amortized time.*

Proof: To prove the lemma, we show that any sequence of k concatenate operations starting from n singleton (a,b)-trees can be performed in a total of $O(k)$ time. We use the potential technique of Sleator and Tarjan (see for instance [17]). To each node in an (a,b)-tree, we assign a potential function φ defined as follows. Let $A = a-1$. If v is a non-root node then

$$\varphi(v) = \frac{\delta(v)-a}{A}.$$

Otherwise, if r is a root of an (a,b)-tree of height h

$$\varphi(r) = h - 1 + \frac{\max\{0, \delta(r) - a\}}{A}.$$

We define the potential $\varphi(T)$ of an (a,b)-tree T to be $\varphi(T) = \sum_{x \in T} \varphi(x)$. Given a forest of (a,b)-trees, the potential Φ is the sum of the potential of all the trees in the forest. The amortized time of an operation is the actual time plus the change it causes in the potential function. Since the potential is always non-negative and it is initially 0, the total time of a sequence of operations is bounded above by the sum of the amortized times of the operations.

To compute the amortized cost of a concatenate operation, we bound the change in potential due to a sharing step, due to a splitting step, and due to the change of height in the resulting tree.

Consider a sharing step between nodes x and y, and let x borrow $p \leq (b-a)$ children from y. Denote by x' and y' the nodes x and y after the sharing step. Then $\varphi(x') \leq \varphi(x) + \frac{p}{A}$ and $\varphi(y') = \varphi(y) - \frac{p}{A}$. As a consequence, $\varphi(x') + \varphi(y') - \varphi(x) - \varphi(y) \leq 0$. The other possible change in the potential is caused by the insertion of at most two edges leaving x' and y' (if x and y are both roots). This is at most $\frac{2}{A}$. Therefore $\Delta\Phi_{sharing} \leq \frac{2}{A} = O(1)$.

As for a splitting step, assume that a non-root node x is split into x' and x''. Since $\delta(x) = b+1$ and $\delta(x'') = \left\lceil \frac{b+1}{2} \right\rceil$, then $\delta(x') = \left\lfloor \frac{b+1}{2} \right\rfloor$. Therefore $\varphi(x') + \varphi(x'') - \varphi(x) = -\frac{a}{A}$. Furthermore the introduction of an edge from x'' to the parent of x causes an increase of potential of at most $\frac{1}{A}$ in the parent of x. Therefore for a non-root node $\Delta\Phi_{splitting} \leq -1$. If x is a root, the only difference is that now there will be two edges leaving x' and x''. Thus in this case $\Delta\Phi_{splitting} \leq -1 + \frac{1}{A}$.

We are left to bound the change of potential due to change in heights. Before we had two trees of height h_1 and h_2. This contributes $h_1 + h_2 - 2$ to the potential. After a concatenate operation we get a tree of height at most $\max\{h_1, h_2\} + 1$. Consequently, $\Delta\Phi_{height} \leq -(\min\{h_1, h_2\} - 1)$.

We now compute the amortized complexity of a concatenate operation. Let ℓ be the number of splitting steps involved. As mentioned before, the actual time is $\ell + \min\{h_1, h_2\}$. To compute the change in potential, we notice that we can have at most one sharing step; furthermore, among the ℓ splitting steps, at most one can be a root splitting. The change in potential is therefore $\Delta\Phi \leq O(1) + (-\ell + \frac{1}{A}) - (\min\{h_1, h_2\} - 1) \leq O(1) - \ell - \min\{h_1, h_2\}$. This implies the $O(1)$ amortized bound for a concatenate operation. □

Theorem 3.1 *The fully persistent disjoint set union problem can be solved in $O(\log n)$ worst-case time per operation, and in $O(1)$ amortized space per update.*

Proof: Lemma 3.1 implies that each union operation produces $O(1)$ amortized structural change in our ephemeral data structure based on (a,b)-trees. Furthermore, each node of an (a,b)-tree has bounded degree. Therefore, we can apply the technique of Driscoll et al. [8] to make (a,b)-trees fully persistent under any sequence of concatenate operations in $O(\log n)$ worst-case time per operation and with $O(1)$ amortized extra space per update. □

We now turn to the fully persistent disjoint set union problem with deletions, which consists of supporting any sequence of Union, Find and Delete operations. Union and Find operations are performed as before. We support a Delete(x,v) operation as follows. The deletion of an item from the ephemeral (a,b)-tree is carried out using the classical algorithm (see for instance [1, pag. 151]), and we perform this operation in a fully persistent fashion by using the techniques of Driscoll et al. [8]. We need some more care in handling the names of the sets. Denote by X the set containing element x in version v. In the new version created by the Delete(x,v) operation, the name of set $X' = X - \{x\}$ is defined as follows: if the name of X was not x, then X' gets the same name as

X; otherwise X' gets as a new name arbitrarily one of its elements. The new name of X' can be computed as follows. The name of X in version v can be found in by a Find(x, v) operation. If this is different from x, then we stop. Otherwise, we have to find (in version v) an element different from x in the set X. This can be done by going in version v from the root of the (a,b)-tree T_X storing set X to the leftmost and to the rightmost item of T_X.

In order to show that Delete operations can be supported efficiently by using the techniques of Driscoll et al. [8], we have to make sure that each operation in the ephemeral data structure implies an $O(1)$ amortized structural change. We do this by generalizing Lemma 3.1 to a larger repertoire of operations, such as inserting an item into an (a,b)-tree, deleting an item from an (a,b)-tree, and concatenating two (a,b)-trees.

Lemma 3.2 *Any sequence of k concatenate operations, insertions and deletions on (a,b)-trees can be performed in a total of $O(k)$ time.*

Proof: We follow the same potential proof technique used in Lemma 3.1. Huddleston and Mehlhorn [10] showed that k insertions and deletions on an (a,b)-tree require a total of $O(k)$ time. We adapt their proof by defining the potential of a non-root node v to be

$$\varphi(v) = \left(\left|\delta(v) - \left\lceil\frac{b}{2}\right\rceil\right| + a - 1\right).$$

For a root r

$$\varphi(r) = h - 1 + \max\left\{0, \delta(r) - \left\lceil\frac{b}{2}\right\rceil + a - 1\right\}.$$

The potential of the forest of (a,b)-trees is defined as in Lemma 3.1.

Using this potential function, the amortized time of an insertion or of a deletion can be shown to be $O(1)$ by using techniques similar to the ones used in [10]. Furthermore, the same argument used in the proof of Lemma 3.1 shows that the amortized time of a concatenate operation is $O(1)$. □

Lemma 3.2 yields the following theorem.

Theorem 3.2 *There is fully persistent data structure that supports each Union, Find and Delete in $O(\log n)$ worst-case time each, and that requires $O(1)$ amortized space per update.*

Proof: Lemma 3.2 guarantees that the structural change in (a,b)-trees is $O(1)$ amortized per operation. Therefore, we can use the techniques in [8] to make (a,b)-trees fully persistent in $O(\log n)$ worst-case time per insertion, deletion and concatenate operation and $O(1)$ amortized space per update. Furthermore, to compute the new name of the set produced by a Delete operation we need at most two accesses in an (a,b)-tree of version v. Consequently, each Union, Find and Delete operation can be implemented in $O(\log n)$ by means of a constant number of operations on fully persistent (a,b)-trees. □

4 Conclusions

In this paper we have shown how to maintain efficiently multiple versions of a partition undergoing disjoint set union operations. In particular, we have described a fully persistent data structure that is able to support Union, Find and Delete operations in $O(\log n)$ worst-case time each and that requires $O(1)$ amortized extra space per update. This bound is tight since any pointer based algorithm requires $\Omega(\log n)$ amortized time to solve the partially persistent disjoint set union problem, as shown by Mannila and Ukkonen [13]. However, this lower bound does not hold if address arithmetic is allowed. Can we achieve better algorithms for our problems by using the extra power of a Random Access Machine?

From both a theoretical and practical point of view, it would be interesting to perform a more general repertoire of operations in a fully persistent fashion, such as set unions, set differences, set intersections and set equality tests on non-disjoint sets. However, we are not aware of any non-trivial data structure for this problem.

References

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

[2] A. Apostolico, G. Gambosi, G. F. Italiano, M. Talamo, "The set union problem with unlimited backtracking", Tech. Rep. CS-TR-908, Department of Computer Science, Purdue University, 1989.

[3] D. Bacon, V. Nguyen, R. Strom, D. Yellin, "The Hermes language reference manual", IBM Tech. Rep., 1990.

[4] L. Banachowski, "A complement to Tarjan's result about the lower bound on the complexity of the set union problem", *Inform. Processing Lett.* 11 (1980), 59–65.

[5] N. Blum, "On the single operation worst-case time complexity of the disjoint set union problem", *SIAM J. Comput.* 15 (1986), 1021–1024.

[6] R. B. K. Dewar, E. Schonberg, J. T. Schwartz, "Introduction to the use of the Set-Theoretic Programming Language SETL", Courant Institute of Mathematical Sciences, Computer Science Department, New York University, 1981.

[7] P. Dietz, "Fully persistent arrays", *Proc. Workshop on Algorithms and Data Structures* (WADS 1989), *Lecture Notes in Computer Science* vol. 382, Springer-Verlag, Berlin, 1989, 67–74.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan, "Making data structures persistent", *J. Comput. Sys. Sci.* 38 (1989), 86–124.

[9] M. L. Fredman, M. E. Saks, "The cell probe complexity of dynamic data structures", *Proc. 21th Annual ACM Symp. on Theory of Computing*, 1989, 345–354.

[10] S. Huddleston, K. Mehlhorn, "A new data structure for representing sorted lists", *Acta Informatica* 17 (1982), 157–184.

[11] J. A. La Poutré, "Lower bounds for the union-find and the split-find problem on pointer machines", *Proc. 22nd Annual ACM Symposium on Theory of Computing*, 1990, 34–44.

[12] H. Mannila, E. Ukkonen, "On the complexity of unification sequences", *Proc. 3rd International Conference on Logic Programming, Lecture Notes in Computer Science* 225, Springer-Verlag, Berlin, 1986, 122–133.

[13] H. Mannila, E. Ukkonen, "Time parameter and arbitrary deunions in the set union problem", *Proc. 1st Scandinavian Workshop on Algorithm Theory* (SWAT 88), *Lecture Notes in Computer Science* vol. 318, Springer-Verlag, Berlin, 1988, 34–42.

[14] N. Sarnak, R. E. Tarjan, "Planar point location using persistent search trees", *Comm. ACM* 29 (1986), 669–679.

[15] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm", *J. Assoc. Comput. Mach.* 22 (1975), 215–225.

[16] R. E. Tarjan, "A class of algorithms which require non linear time to maintain disjoint sets", *J. Comput. Syst. Sci.* 18 (1979), 110–127.

[17] R. E. Tarjan, "Amortized computational complexity", *SIAM J. Alg. Disc. Meth.* 6 (1985), 306–318.

[18] R. E. Tarjan, J. van Leeuwen, "Worst-case analysis of set union algorithms", *J. Assoc. Comput. Mach.* 31 (1984), 245–281.

[19] D. H. D. Warren, L. M. Pereira, "Prolog – the language and its implementation compared with LISP", *ACM SIGPLAN Notices* 12 (1977), 109–115.