INFORMATIQUE THÉORIQUE ET APPLICATIONS

C. GAIBISSO
G. GAMBOSI
M. TALAMO

A partially persistent data structure for the set-union problem

Informatique théorique et applications, tome 24, n° 2 (1990), p. 189-202

<http://www.numdam.org/item?id=ITA_1990__24_2_189_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 24, n° 2, 1990, p. 189 à 202)

A PARTIALLY PERSISTENT DATA STRUCTURE FOR THE SET-UNION PROBLEM (*)

by C. GAIBISSO (1), G. GAMBOSI (1) (2) and M. TALAMO (1) (2)

Communicated by G. AUSIELLO

Abstract. – *We consider an extension of the well known Set-Union problem, where a search in the history of the partition is possible. A partially persistent data structure is presented which maintains a partition of an n-item set with no overhead on the worst case complexity of the ephemeral structure,* i.e. *it performs each* Union *in* $O(1)$ *time, each* Find *in* $O(\lg n)$ *time and each search in the past in* $O(\lg n)$ *time. The space complexity for such a structure is* $O(n)$.

Résumé. – *On considère une extension du problème bien connu de l'Union des ensembles, où il est possible d'effectuer une recherche de l'histoire de la partition. L'article introduit une structure de données partiellement persistante qui maintient une partition d'un ensemble de n-éléments sans aucun coût additionnel, même dans le cas de la complexité de la structure éphémère.*

1. INTRODUCTION

The Set-Union problem and its variants have been extensively studied in recent years [1, 2, 3, 6, 8, 9, 10, 11, 12, 14, 15].

The original problem was [9] that of maintaining a representation of a partition of a set $S=\{1, 2, \ldots, n\}$ under the following two operations:

Union (X, Y, Z): return a new partition of S in which subsets X and Y are merged into one subset $Z=X \cup Y$;

Find (x): given an item $x \in S$, return the name of the (unique) subset containing x.

Initially, each element is assumed to be a singleton.

(*) Received in February 1988, revised in December 1988.

(1) Istituto di Analisi dei Sistemi ed Informatica del C.N.R., Viale Manzoni 30, 00185 Roma, Italy.

(2) Partially supported by ENIDATĂ S.p.A. in the framework of the Alpes-Esprit Project.

A first naive solution, proposed by Galler and Fischer [7], requires $O(1)$ time per *Union* and $O(n)$ time per *Find* in the worst case.

This bound has been remarkably improved by the use of balanced techniques of linking (link by rank, link by size) [7, 14], which made it possible to derive solutions requiring $O(1)$ time per *Union* and $O(\lg n)$ time per *Find* in the worst case.

The best solution with this type of approach is due to Blum [2] and requires $O(\lg n/\lg \lg n)$ single operation worst-case time complexity.

After the introduction of the path compression technique [1], the problem has been extensively studied from the point of view of worst-case time on sequences of *Union* and *Find* operations, *i.e.* of the amortized complexity of operations [13].

In this direction Fischer [5] derived an upper bound of $O(p \lg \lg n)$, where p is the number of operations (both *Unions* and *Finds*) performed; Hopcroft and Ullman [9] improved such bound to $O(p \lg^* n)$, where $\lg^* n$ is the iterated logarithm, defined by $\lg^{(0)} n = n$, $\lg^{(i)} n = \lg^{(i-1)} \lg n$, for $i \geqq 1$, and $\lg^* n = \min \{ i \mid \lg^{(i)} \leqq 1 \}$.

The best solution has been given by Tarjan in [14]. It requires $\Omega(m \alpha(m+n, n)+n)$ running time, where m is the number of *Find* operations performed and α is a very slowly growing (almost constant) function.

It is worth noting that all these time bounds refer to data structures with $O(n)$ space complexity.

Several variants of this problem have been approached, introducing further operations.

Mannila and Ukkonen in [10] introduced a new operation *Deunion* which undo the last *Union* performed, *i.e.* returns to the partition just before the execution of such Union.

In [10] one algorithm has been proposed for maintaining a partition under sequences of *Union*, *Find* and *Deunion* operations and its amortized time complexity has been analyzed. The space complexity of such approach, which was left as an open problem by the authors, has been successively proved to be non linear [8].

After that, the problem has been completely characterized in [16], where $\lg n/\lg \lg n$ upper and lower bounds on the amortized time complexity are derived.

A further generalization of the problem to the case where a (real) weight w is associated to each *Union* operation performed has been proposed in [8]. In such a paper the authors condidered the substitution of the *Deunion*

operation with a *Backtrack* one, whose effect is to return to the situation immediately before the execution of the *Union* of maximum weight thus far performed, *i.e.* to undo all the *Union* operations performed as long as the *Union* of maximum weight has been removed. They also show how to perform each *Union* in $O(\lg \lg n)$, each *Find* in $O(\lg n)$ and each *Backtrack* in $O(1)$ worst-case times with a space complexity $O(n)$.

In this paper we consider an extension of the classical Set-Union problem, introducing a new kind of *Find*, referred as *PFind*, defined as follows:

PFind(x, k): given an item $x \in S$, return the name of the (unique) subset containing x after the k-th *Union* operation was performed.

The *PFind* operation is indeed a search in the history of the partition. Thus our aim is to obtain a partially persistent [4] version of the Set-Union problem, introducing a data structure which supports accesses to all versions in its history, while only the newest version can be modified.

It is worh noting that such an operation includes the usual *Find* as a particular case.

Motivations for the study of the Union-PFind problem may arise for example from the implementation of search heuristics in the framework of logic programming environment design.

The main results of the paper are concerned with the worst-case per operation analysis of the *Union* and *PFind* operations.

We show how to perform each *Union* in $O(1)$, and each *PFind* in $O(\lg n)$ worst-case times, using a partially persistent data structure which requires $O(n)$ space complexity. It is worth noting that no overhead is introduced on the worst-case time and space complexity of the ephemeral structure, even if it does not satisfy the condition introduced in [4] for an efficient [$O(1)$ overhead] transformation from ephemeral to partially persistent data structures.

The remainder of this paper is organized as follows: in section 2 we introduce a data structure for the *Union-PFind* problem, in section 3 we analyse its worst-case time and space complexity. Section 4 contains some concluding remarks.

2. THE DATA STRUCTURE

Recalling the concepts introduced in the last section, the problem we consider is that of mantaining a representation of a partition of a set

$S=\{1, 2, \ldots, n\}$ under the following two operations:

Union (X, Y, Z): return a new partition of S in which subsets X and Y are merged into one subset $Z=X \cup Y$;

PFind(x, k): return the name of the (unique) subset containing $x \in S$ just after the *k-th Union* operation $(1 \leq k \leq n-1)$ was performed. If $n' < n-1$ Unions have been performed thus far and $k > n'$, PFind(x, k) reduces to PFind (x, n').

In order to support the *PFind* operation the data structure used to support the classical Set-Union problem has been modified in the following way:

– if a link has been introduced by the *i-th Union* operation performed, such a link is marked with the integer i. The mark associated to a link l will be referred as *Mark*(l);

– each node n in the data structure has an associated binary search tree *Tree*(n), such that each node in *Tree*(n) corresponds to a link l entering n and stores the name associated to such a node after the *Union* introducing l was performed. Initially to each node n is associated a BST with only one node $(0, n)$.

In the sequel *Root*(X) will refer to the root of the tree representing subset X in the data structure.

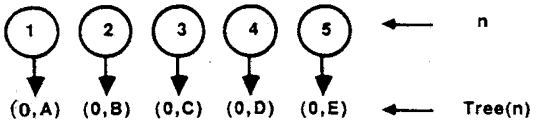
The different operations can now be implemented as follows:

Union (X, Y, Z): a new link, (*Root* (X), *Root* (Y)) or (*Root* (Y), *Root* (X)), is introduced according to the [15] linking "by size" or linking "by rank" strategies, depending on the size or the rank of the involved trees. Furthermore if such an operation is the *i-th Union* performed, a new node (i, Z) is added to *Tree*(*Root*(Y)).

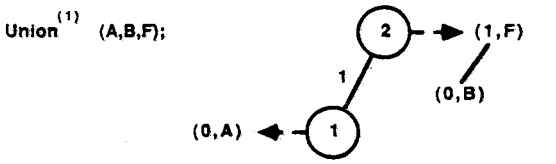
PFind (x, k): starting from node x, traverse the path from this node to *Root*(T) until a node n is reached such that either $n = Root(T)$ or for the (unique) link l outgoing from n the condition $Mark(l) > K$ holds. Let $D=\{x \mid x = Mark(l), l \text{ enters } n, x \leq k\}$: return the name stored by the node in *Tree*(n) corresponding to the link $L' = max\, D$.

The following example illustrates the concepts just introduced, as far as the implementation of the *Union* and *PFind* operations is concerned.

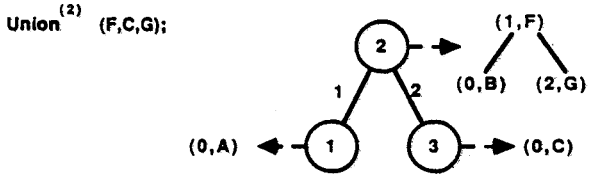
Figure 2.1 *a* shows the data structure representing the initial partition of a set $S=\{1, 2, 3, 4, 5\}$, while figures 2.1 *b*, 2.1 *c*, 2.1 *d* and 2.1 *e* how such a structure evolves when the specified *Union* operation is performed [the *i-th Union* operation performed is referred as *Union*$^{(i)}$].



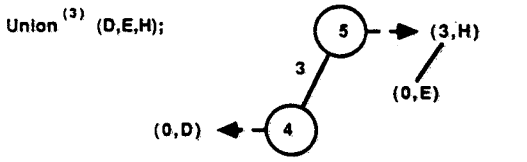
(a)



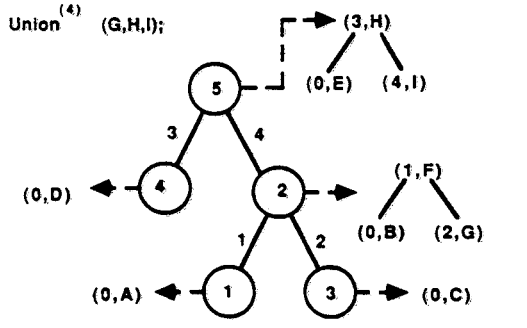
(b)



(c)



(d)



(e)

Figure 2.1.*a* **Figure 2.1.*b*** **Fig. 2.1.*c***
Figure 2.1.*d* **Figure 2.1.*e***

Once *Union* [4] has been performed we have:

PFind(3, 2) = G, since starting from node "3" in figure 2.1*e* the link leaving suh a node is marked 2, and in *Tree* (2) the mark 2 is associated to *G*;

PFind(1, 3) = *G*, since starting from node "1" and moving to the root of the tree, the link leaving node "2" is marked 4, and in *Tree* (2) the node corresponding to the maximum mark less than or equal to 3 is (2, *G*);

PFind (4, 4) = *I*, since starting from node "4" the root of the tree is reached, and the node corresponding to the maximum mark less than or equal to 4 in *Tree* (5) is (4, *I*).

3. COMPLEXITY ANALYSIS

As far as the worst-case time and space complexity are concerned, the following theorem can be stated:

THEOREM 1: *With the previously introduced data structure it is possible to perform:*

(*a*) *the Union operation in* $O(1)$ *worst case time;*

(*b*) *the PFind operation in* $O(\log n)$ *worst case time;*

(*c*) *the amount of space to store the data structure is* $S(n) = O(n)$.

Proof: Let us first prove the following lemma:

LEMMA 1: *If I is a set (eventually infinite) whose elements are ordered by some linear order "<", then* $\forall$ *set* $S \equiv \{a_1, a_2, \ldots, a_n\}$, *such that* $a_i \in I$, $i = 1, 2, \ldots, n$, *it is possible to construct a binary search tree* $T_n (|T_n| = n)$ *for* S, *inserting elements from* S_n *in increasing order and in such a way that the insertion of each element can be performed in constant time. Futhermore Height* $(T_n) = \lceil \lg n \rceil + 1$ *and the space required for such a construction is* $O(n)$.

Proof: Let us consider the following algorithm:

let us suppose, without loss of generality, that $a_i < a_j$ iff $i < j$: to each element to insert a_i, the algorithm associates a node n_i in the structure which is a quadruple

$$\langle INF(n_i); \quad RIGHT(n_i); \quad LEFT(n_i); \quad HEIGHT(n_i) \rangle$$

where:

- $INF(n_i)$ is the information associated to the node, *i.e.* a_i;
- $RIGHT(n_i)$ is a pointer to a node n_j such that $i<j$;
- $LEFT(n_i)$ field is a pointer to a node n_k such that $k<i$;
- $HEIGHT(n_i)$ is defined as follows: if n_i is inserted as a leaf then $HEIGHT(n_i)=1$, otherwise if n_i is inserted as parent of a node n_j then $HEIGHT(n_i)=HEIGHT(n_j)+1$.

As described in the following, the algorithm only inserts nodes of outdegree 0 or 1.

In the sequel a nodes n will be referred as an "incomplete node" if $HEIGHT(n)>1$ and at least one among $RIGHT(n)$ and $LEFT(n)$ is *nil*, and as an "out of level node" if in the structure $HEIGHT(Father(n))-HEIGHT(n)>1$.

The algorithm builds the structure according to a left-to-right and bottom-up strategy. That is, for each node n in the structure, if $HEIGHT(n)=k>1$, then n has been introduced in the structure after the $2^{k-1}-1$ nodes belonging to the left subtree of n and before any node in its right subtree.

The algorithm acts as follows: for each new node p to insert

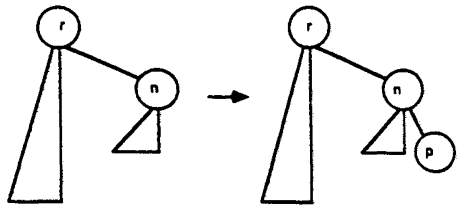
(*a*) if an incomplete node n is present in the structure, then $HEIGHT(p)$ is set to 1 and p is inserted as right as right son of n (fig. 3.1 *a*). It will be proved that at most one incomplete node can exist during the construction and that, in such a case, it appears on the rightmost path of the structure;

(*b*) if no incomplete node is present in the structure but there is at least one out of level node, let n be the last out of level node inserted: then p is inserted as right son of $Father(n)$ and n becomes its left son (*fig.* 3.1 *b*). As will be proved later, all out of level nodes, if exist, appear on the rightmost path of the structure. Furthermore $HEIGHT(p)$ is set to $HEIGHT(n)+1$;

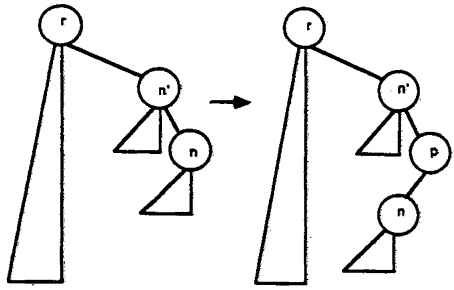
(*c*) if no incomplete or out of level node is present and r is the root of the structure, then p is inserted as the new root of the structure and r becomes its left son (fig. 3.1 *c*).

In the following a more formalized version of the algorithm is presented using a Pascal-like notation.

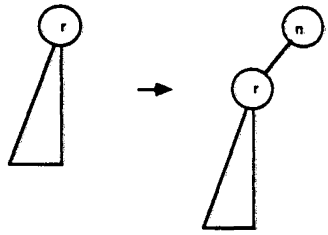
In such a program each node of the built structure is represented as a record of type "Node", while the information related to out of level nodes is contained in a stack implemented as a list of records of type "StackEntry". Two variables "Root" and "Incomplete" of type "NodePointer" stores respectively the reference to the root of the built structure, and the reference



(*a*)



(*b*)



(*c*)

Figure 3.1.*a* **Figure 3.1.*b*** **Figure 3.1.*c***

to the incomplete node, if exists, while a variable "S" of type "Stack of StackEntry" stores the reference to the top of the stack.

The kernel of the program is the procedure "Insert" which is invoked each time a new element has to be inserted in the structure.

```
main ( )
Type
Stack = Stack of StackEntry;
NodePointer = ^ Node;
Node = record
        Inf : Integer;
        Left : NodePointer;
        Right : NodePointer;
        Height : Integer
        end;
StackEntry = record
             Son : NodePointer;
             Parent : NodePointer
             end;
Var
Root, Incomplete : NodePointer;
S : Stack;
Begin
Root := nil;
Incomplete := nil;
Init(S);
For ("Each Inf to insert") Do Insert(Inf)
End;
Procedure Insert(Inf : Integer);
  Var
  AuxNode : NodePointer;
  AuxStack : StackEntry;
  Begin
  New(AuxNode);
  AuxNode ↑. Inf := Inf;
  AuxNode ↑. Right := nil;
  AuxNode ↑. Left := nil;
  (*Empty Structure*)
  If (Root = nil)Then
    Begin
    Root := AuxNode;
    AuxNode ↑. Height := 1
    End;
  Else
  (*The Structure Contains an Incomplete Node*)
  If (Incomplete ≠ nil) Then
    Begin
    AuxNode ↑. Height = 1;
    Incomplete ↑. Right := AuxNode; (* (1) *)
    If ((Incomplete ↑. Height − AuxNode ↑. Height) > 1)
      Then Push (S, AuxNode, Incomplete);
```

```
   Incomplete := nil;
   End;
Else
(*The Structure Contains No Incomplete Nodes but at Least One Out of Level Node*)
If (Not(Empty(S)) Then
   Begin
   AuxStack := Pop(S);
   AuxNode ↑. Left := AuxStack ↑. Son;
   AuxStack ↑. Parent ↑. Right := Auxnode; (* (4) *)
   AuxNode ↑. Height := (AuxNode ↑. Left ↑. Height) + 1;
   Incomplete := AuxNode; (* (2) *)
   If (AuxStack ↑. Parent ↑. Height − AuxNode ↑. Height) > 1
      Then Push (S, AuxNode, AuxStack ↑. Parent)
   End;
Else
(* The Structure Contains neither Incomplete nor Out of Level Nodes *)
   begin
   AuxNode ↑. Height := Root ↑. Height + 1;
   AuxNode ↑. Left := Root;
   Root := AuxNode; (* (5) *)
   Incomplete := Root (* (3) *)
   End
End;
```

In the sequel the BST obtained after the insertion of the j-th element will be referred as T_j.

To prove the correctness and the performances of the algorithm the following lemmas will be proved $\forall n$, $\forall$ set $S \equiv \{a_1, a_2, \ldots, a_n\}$, and T_i, $i = 1, 2, \ldots, n$.

In each proof the presence of an incomplete node, the presence of no incomplete nodes but of at least one out of level node, and the presence of neither incomplete nor out of level nodes, will be referred respectively as case (i), (ii) and (iii).

LEMMA 2: *$\forall j$, $1 \leqq j \leqq n$, T_j contains at most one incomplete node n, and for such a node, if it exists, $RIGHT(n) = nil$ and $LEFT(n) \neq nil$.*

Proof: the proof is by induction.

For $j = 1$ the thesis is true, since T_1 contains no incomplete node.

Assume the induction hypothesis true for T_{j-1}.

If the j-th element has to be introduced:

(i) let n' be an incomplete node in T_{j-1}: then, by the inductive hypothesis, n' is the only incomplete node and only $RIGHT(n')$ is equal to nil; n' will be completed by the algorithm [point (1)], and hence T_j contains no incomplete node;

(ii) n_j will become the only incomplete node [point (2) of the algorithm] of T_j and the algorithm leaves only $RIGHT\ (n_j)$ equal to *nil*;

(iii) as point (ii) [point (3) of the algorithm]. ■

LEMMA 3: $\forall j,\ 1 \leqq j \leqq n$, *all out of level nodes and the incomplete node appear in the rightmost path of* T_j.

Proof: The proof is by induction.

For $j=1$ the thesis is true, since T_1 contains neither incomplete nor out of level nodes.

Assume the induction hypothesis true by T_{j-1}.

If the j-th element has to be introduced:

(i) let n' be the only incomplete node in T_{j-1}: by the inductive hypothesis, n' appaears on the rightmost path of such a structure and by lemma 2. $RIGHT(n')=nil$. The algorithm sets $RIGHT(n')$ [point (1)] making it point to n, hence if n results to be an out of level node it still appears on the rightmost path of T_j;

(ii) let n' be the parent of the last inserted out of level node in the structure: by the inductive hypothesis n' appears on the rightmost path of T_{j-1}. Once inserted n becomes the only incomplete node of T_j, but since it has been inserted as right son of n' [point (4)], it still appears on its rightmost path;

(iii) once inserted n becomes the only incomplete node of T_j, but since it has been inserted as the new root of such a structure [point (5) of the algorithm], it still appears on its rightmost path. ■

It is now possible to prove the correctness of the algorithm proving that:

$$\forall j,\quad 1 \leqq j \leqq n,\ T_j \textit{ is a BST.}$$

Proof: the proof is by induction.

For $j=1$ the thesis is obviously true.

Assume the induction hypothesis true for T_{j-1}.

If the j-th element has to be introduced:

(i) let n be the only incomplete node in T_{j-1}: by lemma 2. $RIGHT(n)=nil$ and by lemma 3. n appears on the rightmost path of such a structure. Since a_j follows a_i, $i=1, 2, \ldots, j-1$, and n_j will become the right son of n, T_j will still be a BST;

(ii) let n be the parent of the last inserted out of level node in T_{j-1}: by lemma 3 it appears on the rightmost path of such a structure. Since a_j follows

a_i, $i=1, 2, \ldots, j-1$, and n_j will become the right son n, T_j will still be a BST;

(iii) in such a case the thesis is trivially true. ■

As far as the algorithm performances are concerned it is easy to derive that the time required to insert an element is constant and the space required to manage insertions if $O(n)$, due to the fact that each node in the BST, each element in the stack and the information related to an incomplete node requires a costant arount of memory to store it, at most one incomplete node appears in the structure, and no element is duplicated in the stack.

Let us now prove that $\lfloor \lg n \rfloor + 1$ is an upper bound for *Height* (T_S):

Proof: the proof is by induction.

For $j=1$ the thesis is obviously true;

Assume the induction hypothesis true for T_{j-1}.

If the j-th element has to be introduced:

(i) in this case $Height(T_j) = Height(T_{j-1})$; but for the inductive hypothesis

$$Height(T_{j-1}) \leqq \lfloor \lg(j-1) \rfloor + 1,$$

hence

$$Height(T_j) \leqq \lfloor \lg(j-1) \rfloor + 1 \leqq \lfloor \lg(j) \rfloor + 1;$$

(ii) as case (i);

(iii) if neither incomplete not out of level nodes appear in the structure then $\exists k \in N$, such that $j=2^k$ and *Height* $(T_{j-1})=k$. Now *Height* $(T_j) = Height$ $(T_{j-1})+1=k+1=\lfloor \lg j \rfloor + 1$. ■

It is now possible to prove theorem 1.

Proof: (*a*) by Lemma 1 the insertions of a new element in the BST associated to any node in the data structure takes $O(1)$, thus each *Union* operation can be performed in constant time;

(*b*) each *PFind*(x, k) implies two searches in the structure:

the first, starting from node x to locate either a node such that for its leaving link l, $mark(l) > k$ or the root of the tree containing x. This search takes obviously $O(\log n)$ time.

The other one, in the BST associated to such a node, to access to the name of the subset containing "x" after the *k-th Union* operation was performed; by lemma 1 also this second search takes $O(\log n)$ time.

Hence each *PFind* operation takes $O(\log n)$ time;

(*c*) since by lemma 1 to manage insertions in each BST T associated to each node in the data structure it is required an amount of space which is $O(m)$, where m is the number of elements in T, and since the sum of the elements in such BSTs is obviously $O(n)$, $S(n)=O(n)$. ■

4. CONCLUSIONS AND FURTHER WORK

In this paper we have considered an extension of the Set-Union problem, where a search in the history of the partition is possible.

We have proposed a partially persistent data structure which supports each *Union* in $O(1)$ and each *PFind* in $O(\lg n)$ worst case times, while the space required is $O(n)$, making this structure competitive with the ephemeral ones proposed for the classical problem.

In this direction the following topics seems worth of further study:

– an amortized complexity analysis, perhaps introducing an alternative path compression technique to the ones proposed in [1];

– effectiveness of using self adjusting data structures;

– a further extension to the problem allowing updates to all the versions in the history of the partition (fully persistent data structure [4]).

REFERENCES

1. A. V. Aho, J. E. Hopcroft and 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*, Proc. 2nd Symp. on Theoretical Aspects of Computer Science, 1985.
3. B. Bollobas and I. Simon, *On the Expected Behavior of Disjoint Set Union Algorithms*, Proc. 17th ACM Symp. on Theory of Computing, 1985.
4. J. R. Driscoll, N. Sarnak, D. D. Sleator and R. E. Tarjan, *Making Data Structures Persistent*, Proc. 18th Symp. on Theory of Computing STOC, 1986.
5. M. J. Fischer, *Efficiency of Equivalence Algorithms, in Complexity of Computations*, R. E. Miller and J. W. Thatcher Eds., Plenum Press, New York, 1972.
6. H. N. Gabow and R. E. Tarjan, *A Linear Time Algorithm for a Special case of Disjoint Set Union*, Proc. 15th A.C.M. Symp. on Theory of Computing 1983.
7. B. A. Galler and M. J. Fischer, *An Improved Equivalence Algorithm*, Comm. ACM 7, 1964.
8. G. Gambosi, G. F. Italiano and M. Talamo, *Worst-Case Analysis of the Set Union Problem with Backtracking*, to appear on "Theoretical Computer Science", 1989.

9. J. E. HOPCROFT and J. D. ULLMAN, *Set Merging Algorithms*, S.I.A.M. J. Comput., 2, 1973.
10. H. MANNILA and E. UKKONEN, *The Set Union Problem with Backtracking*, Proc. 13th I.C.A.L.P., 1986.
11. R. E. TARJAN, *Efficiency of a Good but not Linear Disjoint Set Union Algorithm*, J. A.C.M., 22, 1975.
12. R. E. TARJAN, *A Class of Algorithms which Require Linear Time to Mantain Disjoint Sets*, J. Computer and System Sciences, 18, 1979.
13. R. E. TARJAN, *Amortized Computational Complexity*, S.I.A.M. J. Alg. Discr. Meth., 6, 1985.
14. R. E. TARJAN and J. VAN LEEUWEN, *Worst-Case Analysis of Set Union Algorithms*, J. A.C.M. 31, 1984.
15. J. VAN LEEUWEN and T. VAN DER WEIDE, *Alternative Path Compression Techniques*, Techn. Rep. RUU-CS-77-3, Rijksuniversiteit Utrecht, The Netherlands.
16. J. WESTBROOK and R. E. TARJAN, *Amortized Analysis of Algorithms for Set-Union with Backtracking*, Tech. Rep. TR-103-87, Dept. of Computer Science, Princeton University, 1987.