Theoretical Computer Science 68 (1989) 57-70
North-Holland

WORST-CASE ANALYSIS OF THE SET-UNION PROBLEM WITH EXTENDED BACKTRACKING

Giorgio GAMBOSI*

Istituto di Analisi dei Sistemi ed Informatica—C.N.R., viale Manzoni 30, 00185 Roma, Italy

Giuseppe F. ITALIANO**

Dipartimento di Informatica e Sistemistica, Universitá di Roma "La Sapienza", via Buonarroti 12, 00185 Roma, Italy

Maurizio TALAMO*

Dipartimento di Informatica ed Applicazioni, Universitá di Salerno, 84100 Salerno, Italy and Istituto di Analisi dei Sistemi ed Informatica—C.N.R., Roma, Italy

Communicated by M. Nivat
Received December 1987
Revised February 1988

Abstract. In this paper, an extension of the well known set union problem is considered, where backtracking over sequences of Union operations is allowed. A data structure is presented which maintains a partition of an n-item set making it possible to perform each *Union* in $O(\lg \lg n)$ time, each *Find* in $O(\lg n)$ time and allows backtracking over the Unions in $O(1)$ time. Moreover, it is shown that the data structure can be slightly modified as to present an $O(k + m \lg n)$ time complexity on a sequence of k Unions and Backtracks and m Finds. The space complexity of both versions of such a data structure is $O(n)$.

1. Introduction

The *set-union* problem, together with its variants, is certainly one of the most extensively studied problems in recent years [1, 3, 4, 9, 11-13, 15, 18, 19, 21, 22, 24]. The original problem is that of maintaining a representation of a partition of a set $S = \{1, 2, \ldots, n\}$ in equivalence classes under the following two operations:

Union(X, Y): return a new partition of S in which classes X, Y are merged into a new equivalence class $X \cup Y$ named X.

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

At the beginning, the partition consists of n singletons $\{1\}, \{2\}, \ldots, \{n\}$. The name of the initial set $\{i\}$ is i.

* Partially supported by Enidata S.p.A. in the framework of the ALPES Esprit Project.
** Partially supported by the MPI Project "Design and analysis of algorithms" and by Selenia S.p.A.
Present affiliation: Department of Computer Science, Columbia University, New York, NY 10027, U.S.A.

The best solution for such problem has been presented in [21]: it requires $O(n)$ space and $O(m\alpha(m+n, n)+n)$ running time, where m is the number of Find operations performed and $\alpha(\ldots,\ldots)$ is a very slowly growing (almost constant) function. This results in an amortized time complexity [20] $O(1)$ for Unions and $O(\alpha(m+n, n))$ for Finds.

This has also been proved in [21] to be a lower bound for the amortized complexity of the problem in the (quite general) class of separable algorithms.

An interesting extension of the problem has been proposed in [13]. In it, a new operation *Deunion* is introduced defined as follows:

Deunion: undo the last Union performed so far, i.e. return to the state immediately preceding the execution of the last Union.

This operation was suggested by motivations arising from the implementation of backtracking in the framework of Prolog environment design [10, 14, 23]. In such a context, a sequence of Unions models a sequence of unifications between terms and performing a Deunion corresponds to backtrack to a preceding state in the deduction process.

An algorithm is proposed in [13] which is efficient with respect to amortized time complexity, while the space complexity of the data structure is left as an open problem. Successively, the problem has been completely characterized in [24], where $\lg n/(\lg \lg n)$ upper and lower bounds on the amortized time complexity are derived. The space complexity of the data structure used to establish the upper bound is $O(n \lg n)$.

In this paper, a generalization of such a problem is considered to the case where a (real) weight w is associated with each Union performed and the Deunion operation is substituted by:

Backtrack: return to the partition immediately preceding the execution of the Union of largest weight performed so far, i.e. undo all Unions performed as long as the Union of largest weight is removed.

Moreover, a slight variant of the Union operation is considered where each set has an associated name equal to its canonical element (let us define an element of a set X as *canonical* if it is associated with the root of the tree representing X). Thus, the definition of the Union operation is the following:

Union(x, y, w): return a new partition of S in which classes of canonical elements x, y are merged in a unique class (with canonical element either x or y). A weight w is associated with the operation.

It is easy to note that the *union-find-backtrack* problem reduces easily to the *union-find-deunion* problem simply by letting all Unions have the same weight. Motivations for such an extension derive by the implementation of search heuristics [17] and fast backtracking in Prolog environments. The impact of the technique presented here, which was first introduced in [7] while in [6, 8] other extensions of set-union are considered.

Note that both in the applications considered above and, generally, in all the applications considered for the set union problem, relevance is only given to the

possibility of efficiently testing whether or not two items are in the same set, without caring about the actual name of such a set. Hence, from a practical point of view, the definition of Unions given is equivalent to the classical one.

The main results of this paper are concerned with the worst case analysis of the Union, Find and Backtrack operations. In particular, a data structure is introduced which supports each Union in $O(\lg \lg n)$ time, each Find in $O(\lg n)$ time and each Backtrack in $O(1)$ time, requiring only $O(n)$ space.

Moreover, a second data structure is given which presents an $O(1)$ amortized time complexity on Unions and Backtracks and an $O(\lg n)$ amortized time complexity on Finds. Also this data structure presents an $O(n)$ space complexity.

As a consequence, our first approach compares favorably with Westbrook and Tarjan's algorithms not only when the single operation worst-case time complexity is taken into account, but also according to the space × time complexity of any sequence of operations. In fact, if we assume to perform a sequence of m Finds, Unions and Backtracks (or the corresponding Deunions), Westbrook and Tarjan's algorithms require $O(nm \lg^2 n/\lg \lg n)$ space × time, while our approach leads to an $O(nm \lg n)$ space × time complexity. This ratio can also be improved by considering the second data structure introduced.

The paper is organized as follows: in Section 2 some simplified approaches to the problem are considered. For the sake of comparison, in Section 3 the results given in [13, 24] are extended to the case of the *union-find-backtrack* problem. In Section 4 a data structure for the *union-find-backtrack* problem is introduced and its worst case time and space complexities are derived. Moreover, a modification of such a data structure which performs better in terms of amortized complexity is presented. Section 5 contains some concluding remarks.

2. Simple approaches

In this section, algorithms are presented for the on-line maintenance of a collection of disjoint sets under an arbitrary sequence of Union, Find and Backtrack operations. Let us start by presenting a rather simplified algorithm: next, more efficient and sophisticated approaches will be introduced.

2.1. A naive algorithm

The above operations can be performed using the classical set union data structure [21], where each set is represented by a rooted tree whose nodes correspond to the elements of the set.

In order to make Backtrack possible, an additional stack is used. When a Union has to be performed, linking by rank or by size can be applied. Furthermore, a pointer to the new link together with the value of the largest weight thus far introduced is pushed on the stack. Notice that the largest weight introduced can be easily found as the maximum value between the weight of the performed Union

and the weight on the top of the stack (i.e. the largest weight before the execution of such Union).

According to this technique, the time required to carry out a Union is O(1). Since the height[1] of all trees created by linking by rank or by size is $O(\lg n)$ [21], Find(x) takes at most $O(\lg n)$ time to return to the root of the tree containing x. In order to backtrack, it is necessary to pop from the stack the pointers to be cancelled until a weight value different from the one on the top of the stack is encountered. Unfortunately, the reconstruction of the old configuration can even require $O(n)$ worst-case time. Hence, the following theorem follows.

Theorem 2.1. *It is possible to perform a Union in* $O(1)$ *time, a Find in* $O(\lg n)$ *time and a Backtrack in* $O(n)$ *time. The space required is* $O(n)$.

Proof. Derives from the considerations given above. □

2.2. A lazy algorithm

The time required for backtracking can be reduced by using a lazy approach in which the links cancelled by Backtracks are not removed immediately.

In such a framework, a link (p, q) is considered *live* as long as it corresponds to a Union which has not been cancelled by backtracking. More formally, if we denote by $L(p, q)$ the Union of largest weight performed at the time link (p, q) was introduced, (p, q) is *live* if and only if $L(p, q)$ has not been removed by Backtracks.

While dealing with a Union which introduces a link from p to q, if its weight exceeds the weight value on the top of the stack, the Union is pushed onto the stack in the form of the pair of names of the (canonical elements of the) united sets and $L(p, q)$ is set to (p, q). Otherwise, nothing is pushed onto the stack and $L(p, q)$ is the top element of the stack. In both cases, we associate with the link (p, q) a pointer to the top element of the stack (i.e. to $L(p, q)$).

Notice that now the stack contains only Unions which were of largest weight when introduced (referred in the sequel as *L-unions*) and that the liveness of any link (p, q) can be tested by checking whether the corresponding L-union $L(p, q)$ is still in the stack.

In order to reuse stack records as soon as they are popped, without waiting for all the links pointing to them to be removed, a stamping technique is introduced: namely, we store in each link (p, q) a mark which properly characterizes the corresponding $L(p, q)$ Union. Stack records also contain the corresponding stamp. According to this technique, a link (p, q) is live if and only if the record pointed to by (p, q) is still on the stack and has the same stamp as (p, q) itself. Such an approach makes it possible to bound the number of stamps used, as proved by the following lemma.

[1] The height of a node x in a tree is the length of the longest path from a leaf to x.

Lemma 2.1. *It is possible to introduce a stamping on the Unions performed using a number of different stamps equal to the maximum number of links in the structure.*

Proof. Let l denote the maximum number of links which can be in the structure at any time. Stamps can be organized as an array of size l named *Stamp* such that entry *Stamp*$[k]$ stores the number of links in the structure whose stamp is equal to k. Furthermore, entries containing 0, corresponding to unused stamps, are linked in a list, referred to as *free list.*

When a link is introduced, two cases are possible. If it gets a currently used stamp k, the entry *Stamp*$[k]$ is incremented by 1. Otherwise, an unused stamp is removed from the free list and its entry is initialized to 1. On the other hand, the deletion of a link involves a decrement of the corresponding *Stamp* entry or an insertion in the free list (if it was the only link using that stamp). All these operations require constant time, hence they do not affect the overall time bounds. Notice that the free list is able to return a stamp each time it is required, since at most l stamps can be used at the same time. □

The different operations can now be implemented as follows:

Union(p, q, w): remove the dead links leaving p and q. If w exceeds the weight of the top element of the stack, then push the Union together with the actual stamp onto the stack. Insert a link between p and q and associate with it a pointer to the top of the stack and the actual stamp.

Find(x): starting from the node corresponding to x, follow the live link leaving the node (if such link exists). The liveness of a link can be tested in O(1) time as described above. Repeat until a node with no outgoing live link is entered: return the element associated with such a node.

Backtrack: remove the top element from the stack of L-unions.

The following lemma guarantees that each Union can be performed in O(1) time:

Lemma 2.2. *At any time there is at most one link (dead or alive) leaving a node in the structure. Hence, the maximum number of links in the structure is* $n-1$.

Proof. The only operations which introduce new links are Unions. Furthermore, a link can only be created between roots, i.e. nodes with no leaving live links. Since before creating a new link from p to q all dead links leaving p and q are removed, the lemma can easily be obtained by induction on the number of Unions performed. □

Backtrack also requires constant time, but, unfortunately, it does not allow a complete reconstruction of the old configuration. In particular, the original ranks (or sizes) of the reinstantiated sets are no longer available. This implies a $O(n)$ worst-case time for Finds, since a technique of balanced linking is no longer applicable.

As a result, we obtain the following theorem:

Theorem 2.2. *It is possible to perform both Union and Backtrack in* $O(1)$ *time, while each Find requires* $O(n)$ *worst-case time. The space required is* $O(n)$.

Proof. Derives by Lemma 2.2 and by the considerations given above. □

3. The Union-Find-Deunion approach

Mannila and Ukkonen extended [14] the classical set union problem by introducing a *Deunion* operation, whose effect is to undo the last Union performed not yet undone. They gave an algorithm for it and claimed that it performs an intermixed sequence of m Finds, k Unions and at most k Deunions on n originally singleton sets in $O((k+m)\lg\lg n)$ time, without characterizing the space complexity of their approach.

Recently, Westbrook and Tarjan [24] showed that Mannila and Ukkonen's claim was faulty. In fact, they proved that any separable pointer-based algorithm for the *union-find-deunion* problem requires $\Omega(m \lg n/\lg\lg n))$ time in performing a sequence of m find, union and deunion operations. In the same paper, Westbrook and Tarjan also gave several algorithms based on the approach of [13] with amortized running time of $O(\lg n/\lg\lg n)$, thus matching the introduced lower bound. The space required by all these algorithms is $O(n \lg n)$ [24].

It is not difficult to see that there is a relationship between Backtracks and Deunions. In fact, the *union-find-backtrack* problem reduces to the *union-find-deunion* problem when all the Unions performed have the same weight. On the other side, each Backtrack operation could be implemented by a suitable number of Deunions.

In the following, we shall see that the same amortized bounds proposed by Westbrook and Tarjan in [24] also hold for the new *union-find-backtrack* problem. Furthermore, we shall analyse the worst-case per operation complexity of their algorithms when applied to the new problem. In such a framework, we derive an $O(1)$ bound for each Union, an $O(\lg n)$ bound for each Find, and an $O(n \lg n)$ bound for each Backtrack.

The ideas underlying the algorithms proposed in [13, 24] are the following ones. A path compaction rule (either path splitting or path halving) is combined with a union rule (either linking by size or linking by rank) [21], thus giving rise to four algorithms which have the same time and space performance.

As usual, we refer to a Union operation not yet undone as *live* and as *dead* otherwise. In order to be prepared for Deunions, a *union stack* is maintained, which contains the tree roots made non roots by the performed live Unions. Furthermore, a *link stack* is maintained at each node. All the links leaving a node are pushed on the link stack of the node in the order of their creation (the last links at the top).

New links are pushed on a link stack either when a Union is performed or in the correspondence of Find operations, as a consequence of path compaction. To each link is associated a Union operation, which is the one whose deletion would invalidate the link. A link is *live* if the associated Union is live, otherwise it is said to be *dead.*

Unions are performed as in the set union algorithms without backtracking [21] with one of the union rules. Moreover, the tree root made non root is pushed on the union stack. Also Finds are performed as in the set union algorithms by means of either a path splitting or a path halving rule, except that each new link (p, q) is pushed onto the link stack of p instead of replacing the old link leaving p. To perform Deunions, the top element of the union stack is popped and the link leaving the corresponding node deleted. Furthermore, all the dead links are popped from the link stacks. This obviously requires some bookkeeping. All the details of the method can be found in [24].

Westbrook and Tarjan called this approach the *eager method,* while they denoted the method followed by Mannila and Ukkonen [13] as the *lazy method,* since Mannila and Ukkonen's algorithm destroys dead links in a lazy fashion. In [24] it is shown that both the eager and the lazy method requires $O(m \lg n/\lg \lg n)$ time in performing a sequence of m Union, Find and Deunion operations. However, the eager method is much more space efficient since it uses $O(n \lg n)$ space in the worst case, while the space usage of the lazy method cannot be bounded by any function of n (the number of elements), but only by a function of m (the total number of operations).

As a complete result, we now show that the space bound given for the eager method is tight, i.e. no better than $O(n \lg n)$ space can be achieved.

Theorem 3.1. *There exist instances of the union-find-deunion problem for which the algorithms proposed in* [24] *require space complexity* $S(n) > cn \lg n$, *where* c *is a suitable constant.*

Proof. We consider the algorithm obtained by combining the linking by size rule and the path splitting technique. The algorithms described in [24] which derives from the other cases can be analyzed very similarly.

Let us first assume that $n = 2^h$ and that the set resulting after the operation $union(a, b)$ is named b. Denote by U_k the sequence of Unions

$$U_k = union((2j-1)2^{k-1}, j2^k) \quad j = 1, \ldots, n/2^k, k = 1, \ldots, h.$$

In a similar manner, define F_k as the sequence

$$F_k = U_k, find(1), find(2), \ldots, find(n) \quad k = 2, \ldots, h.$$

Consider now the following sequence of operations:

$$A = U_1, F_2, F_3, \ldots, F_h.$$

It is easy to derive by induction that the structure resulting from such a sequence corresponds to the transitive closure of a binomial rooted tree [21] and that the number of links in such structure is given by

$$m = \sum_{i=1}^{n} \binom{h}{i} i.$$

The result derives by observing that

$$\sum_{i=1}^{h} \binom{h}{i} i = h \sum_{i=1}^{h} \binom{h-1}{i-1} = h \sum_{i=0}^{h-1} \binom{h-1}{i} = h2^{h-1} = \tfrac{1}{2} n \lg_2 n.$$

The general case derives from a simple extension of the considerations above. □

As far as the *union-find-backtrack* problem is concerned, The Union and Find operations can be performed as described above, while a Backtrack involving r Unions ($1 \leq r \leq n-1$) can be performed by exactly r Deunions. Clearly, some bookkeeping is necessary in order to store the number of Unions which must be undone by each Backtrack. With this technique, the amortized cost of each Union and Find is still $O(\lg n / \lg \lg n)$, while the cost of each Backtrack can be charged to the corresponding set of removed Unions, thus resulting in an $O(1)$ amortized complexity for the Backtrack operation.

The worst-case per operation complexity of the algorithms given in [24], when applied to the union-find-backtrack problem, can be characterized as follows.

Theorem 3.2. *The algorithms proposed in* [24] *make it possible to support each Union in* $O(1)$ *time, each Find in* $O(\lg n)$ *and each Backtrack in time* $O(n \lg n)$. *Their space complexity is* $O(n \lg n)$.

Proof. The worst-case bounds on the Union and Find operations are the same as in the set union problem without backtracking [21]. Consider now the sequence A of Unions and Finds described in the proof of Theorem 3.1 and assume that the first performed Union has the largest weight. After performing the sequence A, perform a Backtrack. Clearly, all the links in the structure become dead and the eager algorithms given in [24] have to destroy them. By Theorem 3.1, this Backtrack operation has to delete $O(n \lg n)$ links, thus with a cost of $O(n \lg n)$. An upper bound on the space complexity was given in [24]. This bound is tight as shown in Theorem 3.1. □

4. The data structure

In this section, we shall see how a modified version of the lazy algorithm described in Section 2 is able to efficiently deal with the *union-find-backtrack* problem. The major drawback of the lazy algorithm is that fast backtracking does not permit the

heights (or the sizes) of the reinstated sets to be restored: in order to deal with this problem, one could think of maintaining information about the ranks (sizes) of a node during the evolution of the structure. This task can be accomplished by associating with each node x a balanced tree [2, 5, 16] $Rank(x)$, such that it is possible to insert a new value for the rank of x or delete an old value for such a rank in time $O(\lg k)$, where k is the number of different values stored.

Each item in $Rank(x)$ (in the sequel referred to as a rank of x) uniquely corresponds to a link l entering x and stores the height of x immediately after the introduction of l. If several links $l_1, l_2, \ldots, l_s$ caused x to have the same height h, then there is only one rank of x containing the value h and corresponding to the eldest live link in $\{l_1, l_2, \ldots, l_s\}$.

The notion of liveness can now be extended to a rank of x. More precisely, a rank of x is said to be *live* if and only if the corresponding link is live, otherwise it is said to be *dead.*

In order to restore the height of a node x while backtracking, the largest live rank of x must be individuated. This information clearly turns out to be useful while implementing linking by rank.

With this additional structure, a $Union(p, q, w)$ requires the following three phases.

(1) *Restoring phase*: Remove the dead links leaving p and q together with their corresponding ranks. Retrieve H_p and H_q, respectively, the actual heights of p and q, as the largest live items in $Rank(p)$ and $Rank(q)$.

(2) *Linking phase*: If w exceeds the weight of the top element of the stack, then push that Union together with the actual stamp on the stack. Link by rank p and q and store in the link a liveness pointer to the top of the stack and the top time stamp.

(3) *Rank updating phase*: If $H_p = H_q$, then break the tie by making p child of q and insert into $Rank(q)$ a rank value $(H_q + 1)$.

On the other hand, Find and Backtrack can be performed as in the lazy algorithm.

The distribution of dead and live ranks into the rank trees obeys the rules given in the following lemmas. As we did for the case of the lazy algorithm, we denote by $L(e)$ the Union with largest weight performed at the time in which the link e was introduced in the structure. Such Unions are referred to as L-unions. Due to the linking phase of the algorithm, the stack can contain only L-unions.

Lemma 4.1 (rank consistency). *If in a rank tree a live rank corresponding to a (live) link e_1 is greater than a rank corresponding to a link e_2, then $L(e_1)$ cannot be below $L(e_2)$ in the stack of L-unions.*

Proof. Since only a Union operation can insert a live item in a rank tree, we shall prove the lemma by induction on the number of Unions performed.

At the beginning, the lemma trivially holds, since there are no live ranks.

Assume now that rank consistency holds before performing a Union which introduces a link from p to q. Let us denote by H_p and H_q the largest live ranks respectively in $Rank(p)$ and $Rank(q)$. If $H_p \neq H_q$, then no new rank is created and

hence rank consistency still holds. If $H_p = H_q = H$, only a new rank $H+1$ is created. This new rank is inserted into $Rank(q)$ and is now the largest live item in this rank tree. This rank value $H+1$ corresponds to the link (p, q) and satisfies the rank consistency property since $l(p, q)$ is at this time on the top of the stack of L-unions. This completes the induction step and proves the Lemma. □

Lemma 4.2 (Rank compressions). *In any rank tree, no dead rank can be smaller than a live rank. That is, in any rank tree there exists a rank value v such that all the ranks less or equal to v are live, while the others are dead (v is clearly the largest live item in the rank tree).*

Proof. We proceed by induction on the number of operations performed.

At the beginning, all the rank trees are empty and thus trivially satisfy the rank compression property.

Assume now that the lemma holds before executing the ith operation and consider the following three cases.

(a) If a Union introducing a link from p to q is performed and no new ranks are created, the lemma still holds. Otherwise, a new rank can be created only if the previous largest live items in $Rank(p)$ and $Rank(q)$ had the same value, say H. In this case, only a new live rank of value $H+1$ is inserted into $Rank(q)$. Notice that if there was a previous rank of q whose value was $H+1$, it cannot be live (otherwise H was not the largest live rank of q). In this case, the new rank can be stored in the same node of the rank tree, thus implicitly deleting the old value. This also assures that each balanced tree can never contain two ranks with the same value. Hence, rank compression still holds for $Rank(p)$ (with largest live item of value H) and for $Rank(q)$ (with the largest live item of value $H+1$).

(b) If a Find is performed, then no rank tree is modified and clearly the lemma still holds.

(c) If a Backtrack is performed, then by Lemma 4.1 only the live ranks of greater value can become dead in a rank tree. Once again, the lemma still holds. □

The following lemma characterizes the worst-case time complexity of maintaining the rank trees.

Lemma 4.3. *It is possible to delete an arbitrary item, to insert an arbitrary item, or to search for the largest live item in a rank tree in* $O(\lg \lg n)$ *worst-case time.*

Proof. Since any node x may have at most $\lg n$ different heights during the execution of the Union, Find and Backtrack operations [21] and no rank tree may contain a pair of ranks with the same value, the size of each rank tree is bounded by $\lg n$. As a consequence, any deletion or insertion of it in the rank trees takes at most $O(\lg \lg n)$ time.

Due to the rank compression property proved in Lemma 4.2, the largest live item in a rank tree separates live ranks from dead ranks and hence may be accessed by traversing at most $O(\lg \lg n)$ nodes in the rank tree. By introducing a bidirectional connection between every rank stored in a balanced tree and the corresponding link, the liveness of a rank can be tested in $O(1)$ time, thus yielding to an $O(1)$ time spent at each node of the rank tree during the search. Hence, an $O(\lg \lg n)$ time for searching the largest live item in a rank tree also follows. □

The worst case complexity of the data structure is analyzed in the following theorem.

Theorem 4.1. *It is possible to perform each Union in* $O(\lg \lg n)$, *each Find in* $O(\lg n)$ *and each Backtrack in* $O(1)$ *time. The space required is* $O(n)$.

Proof. As far as Unions are concerned, a result completely analogous to Lemma 2.2 can be proved. Hence each Union may cause the deletion of at most two dead links. By Lemma 4.3, removing a link (together with the corresponding rank, if any) takes $O(\lg \lg n)$ time. Furthermore, the actual height of the two roots p and q to be linked can be restored in $O(\lg \lg n)$ by means of a search for the largest live item in $Rank(p)$ and $Rank(q)$. The linking phase of a Union requires $O(1)$ time, while $Rank(q)$ can be updated in $O(\lg \lg n)$ time, together with the introduction of a bidirectional connection to the link (p, q).

Find and Backtrack are implemented as in the lazy algorithm. Since linking by rank is now possible, each Find requires at most $O(\lg n)$ time, while each Backtrack may be performed in $O(1)$ time.

The space complexity of the data structure can be characterized as follows. The size of the stack of L-unions cannot exceed $n-1$. By Lemma 2.2 there are at most $n-1$ (dead or live) links at the same time in the structure. Even if a rank tree may contain as many as $\lg n$ ranks, each rank corresponds to a (dead or live) link in the structure. This also implies that the total number of items in the rank trees is bounded by $n-1$. Finally, since there are at most $n-1$ links in the structure, Lemma 2.1 assures that at most $n-1$ different stamps are required. This results in an overall $O(n)$ space complexity. □

Let us now consider the case in which rank trees are substituted by *rank stacks*, i.e. for each node x the ranks associated with x in correspondence to the live Unions performed on x are organized in stack order.

The Union operation $Union(p, q, w)$ is now modified as follows:

(1) *Restoring phase*: In order to retrieve H_p and H_q, respectively, the actual heights of p and q, pop dead ranks from both $Rank(p)$ and $Rank(q)$ until the first live rank is encountered. The correctness of such a procedure is guaranteed by the *rank compression* property of Lemma 4.2. Dead links associated with popped ranks are deleted. H_p and H_q are now stored at the top of $Rank(p)$ and $Rank(q)$ and,

by the rank compression property, all ranks remaining in $Rank(p)$ and $Rank(q)$ are live.

(2) *Linking phase*: If w exceeds the weight of the top element of the stack, then push that Union together with the actual stamp on the stack. Link by rank p and q and store in the link a liveness pointer to the top of the stack and the top time stamp.

(3) *Rank updating phases*: If $H_p = H_q$, then break the tie by making p child of q and push into $Rank(q)$ a rank value $(H_q + 1)$.

Theorem 4.2. *It is possible to design a data structure of space complexity* $S(n) = O(n)$ *which makes it possible to perform a sequence of* k *Unions and Backtracks and of* m *Finds in time* $O(k + m \lg n)$.

Proof. The proof uses the banker's technique introduced in [20]. Let us associate with each item r pushed in rank stacks one *credit* $c(r)$, to be used by Union operations in order to pop such an item from the corresponding rank stack. Thus, a sequence of k unions pops at most k items and, since apart from popping of items each Union takes a constant time, the overall complexity of a sequence of k Unions is $O(k)$.

Since the complexities of the Find and Backtrack operations are not affected by this modification of Unions, it turns out that the time complexity of a sequence of k Unions and Backtracks and m Finds is $O(k + m \lg n)$. Finally, all considerations given in Theorem 4.1 regarding the space complexity remain valid, thus resulting in an $O(n)$ space complexity. □

As a consequence of these bounds, our first approach compares favorably to Westbrook and Tarjan's algorithms not only when the single operation worst case time complexity is taken into account, but also according to the space × time complexity of any sequence of operations. In fact, if we assume a sequence of m Finds, Unions and Backtracks (or the corresponding Deunions), Westbrook and Tarjan's algorithms require $O(nm \lg^2 n / \lg \lg n)$ space × time, while our approach leads to an $O(mn \lg n)$ space × time complexity. This ratio can also be improved by considering the second data structure introduced.

5. Conclusions

In this paper, we have considered an extension of the set union problem, where a more powerful form of backtracking than the one usually considered is allowed. We have proposed data structures which support Finds in $O(\lg n)$ time, Backtracks in $O(1)$ time and Unions in either $O(\lg \lg n)$ time per operation or $O(1)$ amortized time. The space required is $O(n)$ and makes this solution competitive when a sequence of operations is considered from the space × time complexity point of view.

An interesting open problem is whether backtracking increases the lower bound of the single operation worst case time complexity of the set union problem which is known to be $\Omega(\lg n/\lg\lg n)$ [3].

After sending the final version of this paper to the publisher, we learnt that Vestbrook and Tarjan improved to $O(n)$ the space complexity of their data structure.

Acknowledgment

We are indebted to Giorgio Ausiello for many stimulating discussions and to Zvi Galil for his helpful suggestions. We wish also to thank an anonymous referee for pointing out the amortized analysis of Union operations.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, New York, 1972).

[2] G.M. Adelson-Velskii and Y.M. Landis, An algorithm for the organization of the information, *Soviet Math. Dokl.* **3** (1962) 1259-1262.

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

[4] B. Bollobas and I. Simon, On the expected behaviour of disjoint set union problems, in: *Proc. 17th ACM Symp. on Theory of Computing* (1985) 224-231.

[5] R. Bayer and E. McCreight, Organization and maintenance of large ordered indices, *Acta Inform.* **1** (1972) 173-179.

[6] C. Gaibisso, G. Gambosi and M. Talamo, A partially persistent data structure for the set-union problem, *RAIRO—Theoretical Informatics and Applications*, to be published (1989).

[7] G. Gambosi, G.F. Italiano and M. Talamo, Getting back to the past in the Union-Find problem, *5th Symp. on Theoretical Aspects of Computer Science* (1988) 8-17.

[8] G. Gambosi, G.F. Italiano and M. Talamo, The Union-Find problem with dynamic weighted backtracking, *Algorithmica*, submitted.

[9] H.N. Gabow and R.E. Tarjan, A linear time algorithm for a special case of disjoint set union, in: *Proc. 15th ACM Symp. on Theory of Computing* (1983) 246-251.

[10] C.J. Hogger, *Introduction to Logic Programming* (Academic Press, New York, 1984).

[11] J.E. Hopcroft and J.D. Ullman, Set merging algorithms, *SIAM J. Comput.* **2** (1973) 294-303.

[12] M. Loebl and J. Nešetřil, Linearity and unprovability of set union problems, in: *Proc. 20th ACM Symp. on Theory of Computing* (1988). 360-366.

[13] H. Mannila and E. Ukkonen, The set union problem with backtracking, in: *Proc. 13th ICALP* (1986) 236-243.

[14] H. Mannila and E. Ukkonen, On the complexity of unification sequences, in: *Proc. 3rd Conf. on Logic Programming* (1986) 122-133.

[15] K. Mehlhorn, S. Naher and H. Alt, A lower bound for the complexity of the union-split-find problem, in: *Proc. 14th ICALP* (1987) 479-488.

[16] J. Nievergelt and E.M. Reingold, Binary search trees of bounded balance, *SIAM J. Comput.* **2** (1973) 33-43.

[17] J. Pearl, *Heuristics* (Addison-Wesley, Reading, MA, 1984).

[18] R.E. Tarjan, Efficiency of a good but not linear set union algorithms, *J. ACM* **22** (1975) 215-225.

[19] R.E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets, *J. Comput. Sys. Sci.* **18** (1979) 110-127.

[20] R.E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Discr. Meth.* **6** (1985) 306-318.

[21] R.E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. ACM* **31** (1984) 245-281.

[22] J. van Leeuwen and T. van der Weide, Alternative path compression techniques, Tech. Rept. RUU-CS-77-3, Rijksuniversiteit Utrecht, The Netherlands (1977).

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

[24] J. Westbrook and R.E. Tarjan, Amortized analysis of algorithms for set union with backtracking, Tech. Rept. TR-103-87, Dept. of Computer Science, Princeton University (1987).