SIAM J. COMPUT.
Vol. 18, No. 1, pp. 1-11, February 1989

© 1989 Society for Industrial and Applied Mathematics
001

AMORTIZED ANALYSIS OF ALGORITHMS FOR SET UNION WITH BACKTRACKING*

JEFFERY WESTBROOK† AND ROBERT E. TARJAN‡

Abstract. Mannila and Ukkonen [*Lecture Notes in Computer Science* 225, Springer-Verlag, New York, 1986, pp. 236-243] have studied a variant of the classical disjoint set union (equivalence) problem in which an extra operation, called de-union, can undo the most recently performed union operation not yet undone. They proposed a way to modify standard set union algorithms to handle de-union operations. In this paper several algorithms are analyzed based on their approach. The most efficient such algorithms have an amortized running time of $O(\log n/\log\log n)$ per operation, where n is the total number of elements in all the sets. These algorithms use $O(n \log n)$ space, but the space usage can be reduced to $O(n)$ by a simple change. The authors prove that any separable pointer-based algorithm for the problem requires $\Omega(\log n/\log\log n)$ time per operation, thus showing that our upper bound on amortized time is tight.

Key words. amortization, set union, data structures, algorithms, backtracking, logic programming

AMS(MOS) subject classifications. 68P05, 68Q25, 68R10

1. Introduction.

The classical disjoint set union problem is that of maintaining a collection of disjoint sets whose union is $U=\{1, 2, \cdots, n\}$ subject to a sequence of m intermixed operations of the following two kinds:

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

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

The initial collection consists of n singleton sets, $\{1\}, \{2\}, \cdots, \{n\}$. The name of initial set $\{i\}$ is i. For simplicity in stating bounds we assume $m=\Omega(n)$. This assumption does not significantly affect any of the results, and it holds in most applications.

Several fast algorithms for this problem are known [10], [13]. They all combine a rooted tree set representation with some form of path compaction. The fastest such algorithms run in $O(\alpha(m, n))$ amortized time[1] per operation, where α is a functional inverse of Ackermann's function [10], [13]. No better bound is possible for any pointer-based algorithm that uses a separable set representation [11]. For the special case of the problem in which the subsequence of union operations is known in advance, the use of address arithmetic techniques leads to an algorithm with an amortized time bound of $O(1)$ per operation [2].

Mannila and Ukkonen [7] studied a generalization of the set union problem called *set union with backtracking*, in which the following third kind of operation is allowed:

de-union: Undo the most recently performed union operation that has not yet been undone.

This problem arises in Prolog interpreter memory management [6]. Mannila and Ukkonen showed how to extend path-compaction techniques to handle backtracking. They posed the question of determining the inherent complexity of the problem, and

* Received by the editors June 29, 1987; accepted for publication (in revised form) April 4, 1988. This research was partially supported by National Science Foundation grant DCR-8605962 and Office of Naval Research contract N00014-87-K-0467.

† Computer Science Department, Princeton University, Princeton, New Jersey 08544.

‡ Computer Science Department, Princeton University, New Jersey 08544 and AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[1] The *amortized time* is the time of an operation averaged over a worst-case sequence of operations. See Tarjan's survey paper [12].

they claimed an $O(\log \log n)$ amortized time bound per operation for one algorithm based on their approach. Unfortunately, their upper bound argument is faulty.

In this paper we derive upper and lower bounds on the amortized efficiency of algorithms for set union with backtracking. We show that several algorithms based on the approach of Mannila and Ukkonen run in $O(\log n/\log \log n)$ amortized time per operation. These algorithms use $O(n \log n)$ space, but the space can be reduced to $O(n)$ by a simple change. We also show that any pointer-based algorithm that uses a separable set representation requires $\Omega(\log n/\log \log n)$ amortized time per operation. All the algorithms we analyze are subject to this lower bound. Improving the upper bound of $O(\log n/\log \log n)$, if it is possible, will require the use of either a non-separable pointer-based data structure or of address arithmetic techniques.

The remainder of this paper consists of four sections. In § 2 we review six algorithms for set union without backtracking and discuss how to extend them to handle backtracking. In § 3 we derive upper bounds for the amortized running times of these algorithms. In § 4 we derive a lower bound on amortized time for all separable pointer-based algorithms for the problem. Section 5 contains concluding remarks and open problems.

2. Algorithms for set union with backtracking.

The known efficient algorithms for set union without backtracking [10], [13] use a collection of disjoint rooted trees to represent the sets. The elements in each set are the nodes of a tree, whose root contains the set name. Each element contains a pointer to its parent. Associated with each set name is a pointer to the root of the tree representing the set. Each initial (singleton) set is represented by a one-node tree.

To perform union(A, B), we make the tree root containing B point to the root containing A, or alternatively make the root containing A point to the root containing B and swap the names A and B between their respective elements. (This not only moves the name A to the right place but also makes undoing the union easy, as we shall see below.) The choice between these two alternatives is governed by a *union rule.* To perform find(x), we follow the path of pointers from element x to the root of the tree containing x and return the set name stored there. In addition, we apply a *compaction rule,* which modifies pointers along the path from x to the root so that they point to nodes farther along the path.

We shall consider the following possibilities for the union and compaction rules:

Union Rules:

Union by weight: Store with each tree root the number of elements in its tree. When doing a union, make the root of the smaller tree point to the root of the larger, breaking a tie arbitrarily.

Union by rank: Store with each tree root a nonnegative integer called its *rank.* The rank of each initial tree root is zero. When doing a union, make the root of smaller rank point to the root of larger rank. In the case of a tie, make either root point to the other, and increase the rank of the root of the new tree by one.

Compaction Rules (see Fig. 1):

Compression: After a find, make every element along the find path point to the tree root.

Splitting: After a find, make every element along the find path point to its grandparent, if it has one.

Halving: After a find, make every other element along the find path (the first, third, etc.) point to its grandparent, if it has one.

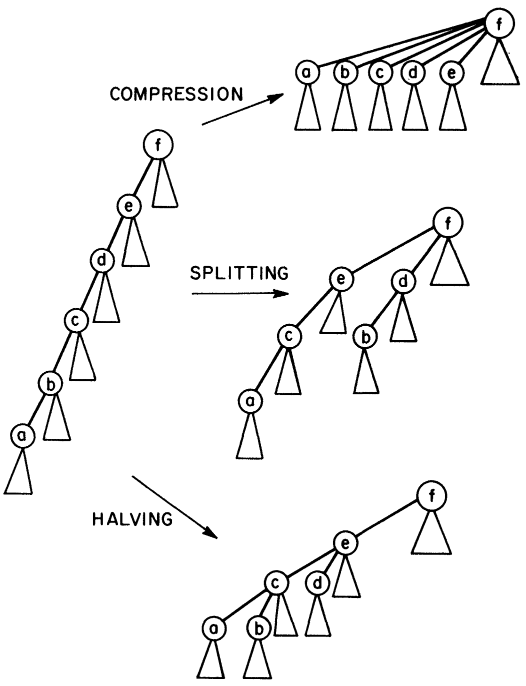

FIG. 1. *Path compression, path splitting, and path halving. The element found is "a."*

The two choices of a union rule and three choices of a compaction rule give six possible set union algorithms. Each of these has an amortized running time of $O(\alpha(m, n))$ per operation [13].

We shall describe two ways to extend these and similar algorithms to handle de-union operations. The first method is the one proposed by Mannila and Ukkonen; the second is a slight variant.

We call a union operation that has been done but not yet undone *live.* We denote a pointer from a node x to a node y by (x, y). Suppose that we perform finds without doing any compaction. Then performing de-unions is easy: to undo a set union we merely make null the pointer added to the data structure by the union. To facilitate this, we maintain a *union stack,* which contains the tree roots made nonroots by live unions. To perform a de-union, we pop the top element on the union stack and make the corresponding parent pointer null.

This method works with either of the two union rules. Some bookkeeping is needed to maintain set names and sizes or ranks. Each entry on the union stack must contain not only an element but also a bit that indicates whether the corresponding union operation swapped set names. If union by rank is used, each such entry must contain a second bit that indicates whether the union operation incremented the rank of the new tree root. The time to maintain set names and sizes or ranks is $O(1)$ per union or de-union; thus each union or de-union takes $O(1)$ time, worst-case. Either union rule guarantees a maximum tree depth of $O(\log n)$ [13]; thus the worst-case time per find is $O(\log n)$. The space needed by the data structure is $O(n)$.

Mannila and Ukkonen's goal was to reduce the time per find, possibly at the cost of increasing the time per union or de-union and increasing the space. They developed the following method for allowing compaction in the presence of de-unions. Let us call the forest maintained by the noncompacting algorithm described above the *reference* *forest.* In the compacting method, each element x has an associated *pointer stack* $P(x)$, which contains the outgoing pointers that have been created during the course of the algorithm but have not yet been destroyed. The bottommost pointer on this stack is one created by a union. Such a pointer is called a *union pointer.* The other pointers on the stack are ones created by compaction. They are called *find pointers.* Each pointer (x, y) of either type is such that y is a proper ancestor of x in the reference forest.

Each pointer has an *associated union operation*, which is the one whose undoing would invalidate the pointer. To be more precise, for a pointer (x, y) the associated union operation is the one that created the pointer (z, y) such that z is a child of y and an ancestor of x in the reference forest. As a special case of this definition, if (x, y) is a union pointer, then $z = x$ and the associated union operation is the one that created (x, y). A pointer is *live* if its associated union is live.

Unions are performed as in the noncompacting method. Compactions are performed as in the set union algorithm without backtracking, except that each new pointer (x, y) is pushed onto $P(x)$ instead of replacing the old pointer, leaving x. When following a find path from an element x, the algorithm pops dead pointers from the top of $P(x)$ until $P(x)$ is empty or a live pointer is on top. In the former case, x is the root of its tree; in the latter case, the live pointer is followed.

This algorithm requires a way to determine whether a pointer is live or dead. For this purpose the algorithm assigns each union operation a distinct number as it is performed. Each entry on the union stack contains the number of the corresponding union. Each pointer on a pointer stack contains the number of the associated union and a pointer to the position on the union stack where the entry for this union was made. This information can be computed in $O(1)$ time for any pointer (x, y) when it is created. If (x, y) is a union pointer, the information is computed as part of the union. If (x, y) is a find pointer, then the last pointer on the find path from x to y when (x, y) was created has the same associated union as (x, y) and has stored with it the needed information. To test whether a pointer is live or dead, it is merely necessary to access the union stack entry whose position is recorded with the pointer and test first, if the entry is still on the stack, and second, whether its union number is the same as that stored with the pointer. If so, the pointer is live; if not, dead.

The implementation of de-union must be changed slightly, to preserve the invariant that in every pointer stack all the dead pointers are on top. To perform a de-union, the algorithm pops the top entry on the union stack. Let x be the element in this entry. The algorithm pops $P(x)$ until it contains only one pointer, which is the union pointer created by the union that is to be undone. The algorithm restores the set names and sizes or ranks as necessary, and pops the last pointer from $P(x)$. Because of the compaction, the state of the data structure after a de-union will not in general be the same as its state before the corresponding union.

We call this method the *lazy method* since it destroys dead pointers in a lazy fashion. Either of the union rules and any of the compaction rules can be used with the method. The total running time is proportional to m plus the total number of pointers created. (With any of the compaction rules, a compaction of a find path containing $k \geqq 2$ pointers results in the creation of $\Omega(k)$ pointers, $k-1$ in the case of compression or splitting and $\lfloor k/2 \rfloor$ in the case of halving.)

An alternative to the lazy method is the *eager method*, which pops pointers from pointer stacks as soon as they become dead. To make this popping possible, each union stack entry must contain a list of the pointers whose associated union is the one corresponding to the entry. When a union stack entry is popped, all the pointers on its list are popped from their respective pointer stacks as well. Each such pointer will be on top of its stack when it is to be popped. To represent such a pointer, say (x, y), in a union stack entry, it suffices to store x. With this method, numbering the union operations is unnecessary, as is popping pointer stacks during finds.

The time required by the eager method for any sequence of operations is only a constant factor greater than that required by the lazy method, since both methods create the same pointers but the eager method destroys them earlier. With either union rule, the eager method uses $O(n \log n)$ space in the worst case, since the maximum tree depth is $O(\log n)$ and all pointers on any pointer stack point to distinct elements. (From bottom to top, the pointers on $P(x)$ point to shallower and shallower ancestors of x.)

The lazy method also has an $O(n \log n)$ space bound [3]. For any node x, consider the top pointer on $P(x)$, which is to a node, say y. Even if the pointer from x to y is currently dead, it must once have been live, and all pointers currently on $P(x)$ point to distinct nodes on the tree path from x to y as it existed when the pointer from x to y was live. Thus there can be only $O(\log n)$ such pointers. The total number of pointers therefore is $O(n \log n)$. The total number of numbers needed to distinguish relevant union operations is also $O(n \log n)$, which implies that the total space needed is $O(n \log n)$, as claimed.

The choice between the lazy and eager methods is not clear-cut. As we shall see at the end of § 3, a small change in the compaction rules reduces the space needed by either method to $O(n)$.

3. Upper bounds on amortized time. The analysis to follow applies to both the lazy method and the eager method. If we ignore the choice between lazy and eager pointer deletion, there are six versions of the algorithm, depending on the choice of a union rule and a compaction rule.

As a first step on the analysis, we note that compression with either union rule is no better in the amortized sense than doing no compaction at all, i.e., the amortized time per operation is $\Omega(\log n)$. The following class of examples shows this. For any k, form a tree of 2^k elements by doing unions on pairs of elements, then on pairs of pairs, and so on. This produces a tree called a *binomial tree* B_k, whose depth is k. (See Fig. 2.) Repeat the following three operations any number of times: do a find on the deepest element in B_k, undo the most recent union, and redo the union. Each find creates $k-1$ pointers, which are all immediately made dead by the subsequent de-union. Thus the amortized time per operation is $\Omega(k) = \Omega(\log n)$.

Both splitting and halving perform better; each has an $O(\log n/\log\log n)$ amortized bound per operation, in combination with either union rule. To prove this, we need a definition. For an element x, let size(x) be the number of descendants of x (including itself) in the reference forest. The *logarithmic size* of x, lgs(x), is $\lfloor \lg \text{size}(x) \rfloor$.[2]

We need the following lemma concerning logarithmic sizes when union by weight is used.

LEMMA 1 [10]. *Suppose union by weight is used. If node v is the parent of node w in the reference forest, then* $\text{lgs}(w) < \text{lgs}(v)$. *Any node has logarithmic size between* 0 *and* $\lg n$ *(inclusive).*

Proof. When a node v becomes the parent of another node w, $\text{size}(w) \leqq 2\,\text{size}(v)$ by the union by weight rule. Later unions can only increase size(v) and cannot increase size(w) (unless the union linking v and w is undone). The lemma follows. □

[2] For any x, $\lg x = \log_2 x$.

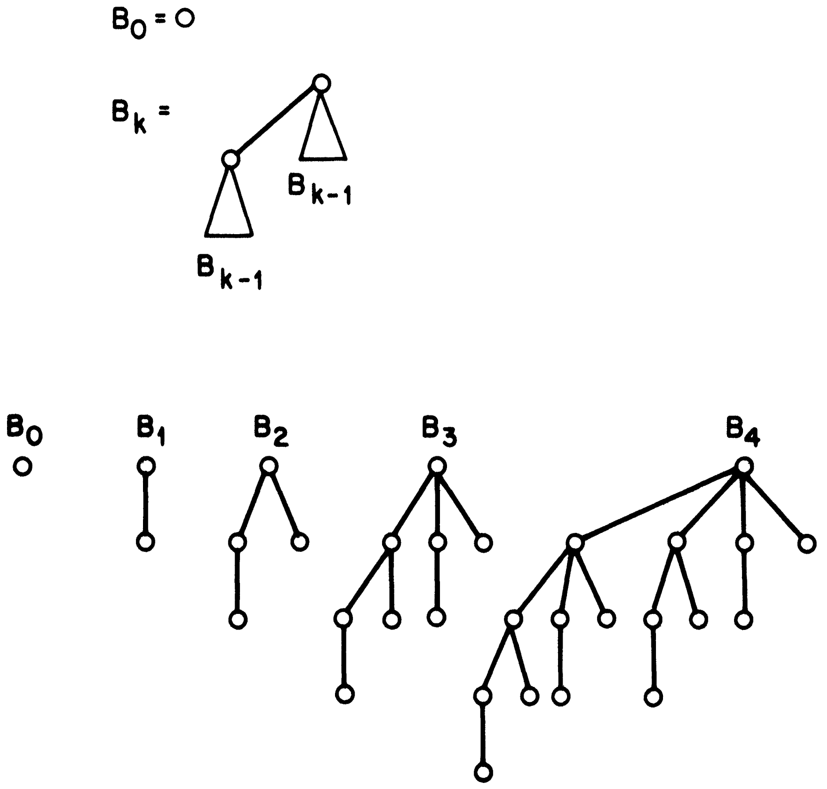

FIG. 2. *Binomial trees.*

THEOREM 1. *Union by weight in combination with either splitting or halving gives an algorithm for set union with backtracking running in $O(\log n/\log\log n)$ amortized time per operation.*

Proof. We shall charge the pointer creations during the algorithm to unions and finds in such a way that each operation is charged for $O(\log n/\log\log n)$ pointer creations. For an arbitrary positive constant $c<1$, we call a pointer (x, y) *short* if $\operatorname{lgs}(y)-\operatorname{lgs}(x)\leqq c \lg\lg n$ and *long* otherwise. (The logarithmic sizes in this definition are measured at the time (x, y) is created.) We charge the creation of a pointer (x, y) as follows:

(i) If y is a tree root, charge the operation (union or find) that created (x, y).
(ii) If y is not a tree root and (x, y) is long, charge the find that created (x, y).
(iii) If y is not a tree root and (x, y) is short, charge the union that most recently made y a nonroot.

A find with splitting creates two new paths of pointers, and a find with halving creates one new path of pointers. Thus $O(1)$ pointers are charged to each operation by (i). The number of long pointers along any path can be estimated as follows. For any long pointer (x, y), $\operatorname{lgs}(y)-\operatorname{lgs}(x)>c\lg\lg n$. Logarithmic sizes strictly increase along any path and are between 0 and $\lg n$ by Lemma 1. Thus if there are k long pointers on a path, $\lg n\geqq kc\lg\lg n$, which implies $k\leqq \lg n/(c\lg\lg n)$. Thus a find with either splitting or halving can create only $O(\log n/\log\log n)$ long pointers, which means that $O(\log n/\log\log n)$ pointers are charged to each find by (ii).

It remains for us to bound the number of pointers charged by (iii). Consider a union operation that makes an element x a child of another element y. Let I be the time interval during which pointers are charged by (iii) to this union. During I, the sizes, and hence the logarithmic sizes, of all descendants of x remain constant. Interval I ends with the undoing of the union.

For each descendant w of x, at most one pointer (w, x) can be charged by (iii) to the union, since the creation of another such pointer charged by (iii) cannot occur at least until x again becomes a root and then becomes a nonroot, which can only happen after the end of I. Thus the number of pointers charged by (iii) to the union is at most one per descendant w of x such that $\mathrm{lgs}(x)-\mathrm{lgs}(w)\leqq c \lg\lg n$.

Since logarithmic sizes strictly increase along tree paths, any two elements u and v with $\mathrm{lgs}(u)=\mathrm{lgs}(v)$ must be unrelated, i.e., their sets of descendants are disjoint. This means that the number of descendants w of x with $\mathrm{lgs}(w)=i$ is at most $\mathrm{size}(x)/2^i \leqq 2^{\mathrm{lgs}(x)+1-i}$, and the number of descendants w of x with $\mathrm{lgs}(x)-\mathrm{lgs}(w)\leqq c\lg\lg n$ is at most

$$\sum_{i=\mathrm{lgs}(x)-\lfloor c \lg\lg n\rfloor}^{\mathrm{lgs}(x)} 2^{\mathrm{lgs}(x)+1-i} \leqq 2^{\lfloor c\lg\lg n\rfloor+2} = O((\log n)^c) = O(\log n/\log\log n),$$

since $c<1$. Thus there are $O(\log n/\log\log n)$ pointers charged to the union by (iii). □

The same result holds if union by rank is used instead of union by weight, but in this case the proof becomes a little more complicated because logarithmic sizes need not strictly increase along tree paths. We deal with this by slightly changing the definition of short and long pointers. We need the following lemma.

LEMMA 2 [13]. *Suppose union by rank is used. If node v is the parent of node w in the reference forest, then* $0\leqq \mathrm{lgs}(w)\leqq \mathrm{lgs}(v)\leqq \lg n$ *and* $0\leqq \mathrm{rank}(w)<\mathrm{rank}(v)\leqq \lg n$.

Proof. The first group of inequalities is immediate. The definition of union by rank implies $\mathrm{rank}(w)<\mathrm{rank}(v)$. A proof by induction on the rank of v shows that $\mathrm{size}(v)\geqq 2^{\mathrm{rank}(v)}$, which implies that $\mathrm{rank}(v)\leqq \lg n$. □

THEOREM 2. *Union by rank in combination with either splitting or halving gives an algorithm for set union with backtracking running in $O(\log n/\log\log n)$ amortized time per operation.*

Proof. We define a pointer (x, y) to be *short* if $\max\{\mathrm{lgs}(y)-\mathrm{lgs}(x), \mathrm{rank}(y)-\mathrm{rank}(x)\}\leqq c\lg\lg n$ and *long* otherwise, where $c<1$ is a positive constant. We charge the creation of pointers to unions and finds exactly as in the proof of Theorem 1 (rules (i), (ii), and (iii)). The number of pointers charged by rule (i) is $O(1)$ per union or find, exactly as in the proof of Theorem 1. A long pointer (x, y) satisfies at least one of the inequalities $\mathrm{lgs}(x)-\mathrm{lgs}(x)>c\lg\lg n$ and $\mathrm{rank}(y)-\mathrm{rank}(x)>c\lg\lg n$. Along any tree path only $O(\log n/\log\log n)$ long pointers can satisfy the former inequality and only $O(\log n/\log\log n)$ long pointers can satisfy the latter, by Lemma 2. It follows that only $O(\log n/\log\log n)$ pointers can be charged per find by rule (ii).

To count short pointers, we have one additional definition. For a nonroot element x, let $p(x)$ be the parent of x in the reference forest. A nonroot x is *good* if $\mathrm{lgs}(x)<\mathrm{lgs}(p(x))$ and *bad* otherwise, i.e., if $\mathrm{lgs}(x)=\mathrm{lgs}(p(x))$. The definition of lgs implies that any element can have at most one bad child. The bad elements thus form paths called *bad paths* of length $O(\log n)$; all elements on a bad path have the same logarithmic size. We call the element of largest rank on a bad path the *head* of the path. The head of a bad path is a bad element whose parent is either a good element or a tree root.

Consider a union operation that makes an element x a child of an element y. We count short pointers charged to this union as follows:

(1) *Short pointers leading from good elements.* If v and w are good elements such that $\mathrm{lgs}(v)=\mathrm{lgs}(w)$, then v and w are unrelated in the reference forest, i.e., they have

disjoint sets of descendants. The analysis that yielded the count of short pointers in the proof of Theorem 1 applies to the good elements here to yield a bound of $O((\log n)^c) = O(\log n/\log\log n)$ short pointers leading from good elements that are charged to the union by (iii).

(2) *Short pointers leading from bad elements.* Consider the number of bad paths from which short pointers can lead to x. The head of such a path is an element w such that $p(w)$ is either good or a tree root, and $\text{lgs}(x) - \text{lgs}(w) \leqq c \lg\lg n$. Heads of different bad paths have different parents. The analysis that counts short pointers in the proof of Theorem 1 yields an $O((\log n)^c)$ bound on the number of bad paths from which short pointers can lead to x. Along such a bad path, rank strictly increases, and the definition of shortness implies that only the $c \lg\lg n$ elements of largest rank along the path can have short pointers leading to x. The total number of short pointers leading from bad nodes that are charged to the union by (iii) is thus $O(c \log\log n\ (\log n)^c) = O(\log n/\log\log n)$. $\square$

We conclude this section by discussing how to reduce the space bound for both the lazy method and the eager method to $O(n)$. This is accomplished by making the following simple changes in the compaction rules. If union by size is used, the compaction of a find path is begun at the first node along the path whose size is at least $\lg n$. If union by rank is used, the compaction of a find path is begun at the first node whose rank is at least $\lg\lg n$. With this modification, only $O(n/\log n)$ nodes have find pointers leaving them, and the total number of pointers in the data structure at any time is $O(n)$. The analysis in Theorems 1 and 2 remains valid, except that there is an additional time per find of $O(\log\log n)$ to account for the initial, noncompacted part of each find path.

4. A general lower bound on amortized time.

We shall prove that the bound in Theorems 1 and 2 is best possible for a large class of algorithms for set union with backtracking. Our computational model is the *pointer machine* [4], [5], [9], [11] with an added assumption about the data structure called *separability*. Related results follow. Tarjan [11] derived an amortized bound in this model for the set union problem without backtracking. Blum [1] derived a worst-case-per-operation lower bound for the same problem. Mehlhorn, Näher, and Alt [8] derived an amortized lower bound for a related problem. Their result does not require separability.

The algorithms to which our lower bound applies are called *separable pointer algorithms.* Such an algorithm uses a linked data structure that can be regarded as a directed graph, with each pointer represented by an edge. The algorithm solves the set union with backtracking problem according to the following rules:

(i) The operations are presented on-line, i.e., each operation must be completed before the next one is known.

(ii) Each set element is a node of the data structure. There can be any number of additional nodes.

(iii) (Separability.) After any operation, the data structure can be partitioned into node-disjoint subgraphs, one corresponding to each currently existing set and containing all the elements in the set. The name of the set occurs in exactly one node in the subgraph. *No edge leads from one subgraph to another.*

(iv) The cost of an operation find(x) is the length (number of edges) of the shortest path from x to the node that holds the name of the set containing x. This length is measured at the beginning of the find, i.e., before the algorithm changes the structure as specified in (v).

(v) During any find, union, or de-union operation, the algorithm can add edges to the data structure at a cost of one per edge, delete edges at a cost of zero, and move, add, or delete set names at a cost of zero. The only restriction is that separability must hold after each operation.

The eager method of § 2 obeys rules (i)–(v). This is also true of the lazy method, if we regard pointers as disappearing from the model data structure as soon as they become dead. This does not affect the performance of the algorithm in the model, since once a pointer becomes dead it is never followed.

THEOREM 3. *For any n, any $m = \Omega(n)$, and any separable pointer algorithm, there is a sequence of m find, union, and de-union operations whose cost is $\Omega(m \log n/\log \log n)$.*

Proof. We shall prove the theorem for n of the form 2^{2^k} for some $k \geqq 1$ and for $m \geqq 4n$. The result follows for all n and $m = \Omega(n)$ by padding the expensive problem instances constructed below with extra singleton sets on which no operations take place and with extra finds.

In estimating the cost of a sequence of operations, we shall charge the cost of adding an edge to the data structure to the deletion of the edge. Since this postpones the cost, it cannot increase the total cost of a sequence.

We construct an expensive sequence as follows. The first $n-1$ operations are unions that build a set of size n by combining singletons in pairs, pairs in pairs, and so on. The remaining operations occur in groups, each group containing between 1 and $2n-2$ operations. Each group begins and ends with all the elements in one set. We obtain a group of operations by applying the appropriate one of the following two cases (if both apply, either may be selected). Let $b = \lfloor \lg n/(2 \lg \lg n) \rfloor$.

(1) If some element in the (only) set is at distance at least b away from the set name, do a find on this element.

(2) If some sequence of ℓ de-unions will force the deletion of ℓb edges from the data structure (to maintain separability), do these de-unions. Then do the corresponding unions in the reverse order, restoring the initial set of size n.

We claim that if there is only one set, formed by repeated pairing, then case (1) or case (2) must apply. If this is true, we can obtain an expensive sequence of operations by generating successive groups of operations until more than $m-2n+2$ operations have occurred, and then padding the sequence with enough additional finds to make a total of m operations. The cost of such a sequence is at least $(m-3n+3)b = \Omega(m \log n/\log \log n)$.

It remains to prove the claim. Suppose case (2) does not apply. We shall show that case (1) does. Let $f = (\lg n)^2$. For $0 \leqq i \leqq \lg n/\lg f$ we define a partition P_i of the nodes of the data structure as follows:

$$P_i = \{X \mid X \text{ is the collection of nodes in the subgraph corresponding to one of the sets that would be formed by doing } f^i - 1 \text{ de-unions}\}.$$

Observe that $|P_i| = f^i$. Also $f^{\lg n/\lg f} = n$, so P_i is defined for $i \leqq \lg n/\lg f$. In particular P_b is defined, since $b = \lfloor \lg n/(2 \lg \lg n) \rfloor = \lfloor \lg n/\lg f \rfloor$.

For $0 \leqq i \leqq \lg n/\lg f$, we define the collection D_i of *deep sets* in P_i as follows:

$$D_i = \{X \in P_i \mid \text{all elements in } X \text{ are at distance at least } i \text{ from the name of the single set}\}.$$

Let $d_i = |D_i|$. We shall show that $d_b > 0$, which implies the existence of an element at distance at least b away from the name of the single set; hence case (1) applies.

Let ℓ_i be the number of edges leading from one set in P_i to another. We have $\ell_i \leqq bf^i$, since otherwise performance of $f^i - 1$ de-unions would force the deletion of bf^i edges from the data structure, and case (2) would apply.

Now we derive a recursive bound on d_i. We have $d_1 = f - 1$, since only one of the f sets in P_1 can contain the only set name. We claim that $d_{i+1} \geqq fd_i - \ell_i$. To verify the claim, let us consider D_i. Since $n = 2^{2^k}$ and the union structure of the only set forms a binomial tree, each set X in D_i consists of f sets in P_{i+1}, all of whose elements are at distance at least i from the name of the only set. For an element $x \in X$ to be at distance exactly i from the set name, some edge must lead from x to a set in P_i other than X; otherwise X would not be in D_i. There are ℓ_i such edges. Each such edge can eliminate one set in P_{i+1} from being in D_{i+1}. But this leaves $fd_i - \ell_i$ sets in D_{i+1}, namely the fd_i sets into which the sets in D_i divide, minus at most ℓ_i eliminated by edges between different sets in P_i. That is, $d_{i+1} \geqq fd_i - \ell_i$, as claimed.

Applying the bound $\ell_i \leqq bf^i$ gives $d_{i+1} \geqq fd_i - bf^i$. Using $d_1 = f - 1$, a proof by induction shows that $d_i \geqq f^{i-1}(f-(i-1)b-1)$.

We wish to show that $d_b > 0$. This is true provided that $(f-(b-1)b-1) > 0$. But $f = (\lg n)^2$ and $b = \lfloor \lg n/(2 \lg \lg n) \rfloor$, giving $(f-(b-1)b-1) = (f-b^2+b-1) \geqq \frac{3}{4}(\lg n)^2 > 0$, since we are assuming $n \geqq 4$, which implies $b^2 \leqq (\lg n)^2/4$ and $b \geqq 1$. Thus $d_b > 0$, which implies that some element is at distance at least b from the set name, i.e., case (1) applies. □

5. Remarks. Our bound of $\Theta(\log n/\log \log n)$ on the amortized time per operation in the set union problem with backtracking is the same as Blum's worst-case bound per operation in the set union problem without backtracking [1]. Perhaps this is not a coincidence. Our lower bound proof resembles his. Furthermore the data structure he uses to establish his upper bound can easily be extended to handle de-union operations; the worst-case bound per operation remains $O(\log n/\log \log n)$ and the space needed is $O(n)$.

The compaction methods have the advantage over Blum's method that as the ratio of finds to unions and de-unions increases, the amortized time per find decreases. The precise result is that if the ratio of finds to unions and de-unions in the operation sequence is γ and the amortized time per union and de-union is defined to be $\Theta(1)$, then the amortized time per find is $\Theta(\log n/(\max\{1, \log(\gamma \log n)\}))$. This bound is valid for any value of γ, and it holds for splitting or halving with either union rule, and it is the best bound possible for any separable pointer algorithm. This can be proved using straightforward extensions of the arguments in §§ 3 and 4. The space bound can be made $O(n)$ by an extension of the idea proposed at the end of § 3. If the de-union operations occur in bursts, the time per operation decreases further, but we have not attempted to analyze this situation.

Perhaps the most interesting open problem is whether the lower bound in § 4 can be extended to nonseparable pointer algorithms. (In place of separability, we require that the out-degree of every node in the data structure be constant.) We conjecture that the bound in Theorem 3 holds for such algorithms. The techniques of Mehlhorn, Näher, and Alt [8] suggest an approach to this question, which might yield at least an $\Omega(\log \log n)$ bound if not an $\Omega(\log n/\log \log n)$ bound on the amortized time.

REFERENCES

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

[2] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209-221.

[3] G. GAMBIOSI, G. F. ITALIANO, AND M. TALAMO, *Getting back to the past in the union-find problem*, in 5th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 294, Springer-Verlag, Berlin, 1988, pp. 8-17.

[4] D. E. KNUTH, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[5] A. N. KOLMOGOROV, *On the notion of algorithm*, Uspekhi Mat. Nauk, 8 (1953), pp. 175-176.

[6] H. MANNILA AND E. UKKONEN, *On the complexity of unification sequences*, in 3rd International Conference on Logic Programming, July 14–18, 1986, Lecture Notes in Computer Science 225, Springer-Verlag, New York, 1986, pp. 122–133.

[7] ———, *The set union problem with backtracking*, in Proc. 13th International Colloquium on Automata, Languages, and Programming (ICALP 86), Rennes, France, July 15–19, 1986, Lecture Notes in Computer Science 226, Springer-Verlag, New York, 1986, pp. 236–243.

[8] K. MEHLHORN, S. NÄHER, AND H. ALT, *A lower bound for the complexity of the union-split-find problem*, in Proc. 14th International Colloquium on Automata, Languages, and Programming (ICALP 87), Karlsruhe, Federal Republic of Germany, July 13–17, 1987, Lecture Notes in Computer Science 267, Springer-Verlag, New York, 1987, pp. 479–488.

[9] A. SCHÖNHAGE, *Storage modification machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

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

[11] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[12] ———, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 306–318.

[13] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.