JOURNAL OF COMPUTER AND SYSTEM SCIENCES 30, 209–221 (1985)

A Linear-Time Algorithm for a Special Case of Disjoint Set Union

HAROLD N. GABOW*

University of Colorado at Boulder, Boulder, Colorado

AND

ROBERT ENDRE TARJAN

AT & T Bell Laboratories, Murray Hill, New Jersey 07974

Received July 12, 1983; revised December 10, 1984

This paper presents a linear-time algorithm for the special case of the disjoint set union problem in which the structure of the unions (defined by a "union tree") is known in advance. The algorithm executes an intermixed sequence of m union and find operations on n elements in $O(m+n)$ time and $O(n)$ space. This is a slight but theoretically significant improvement over the fastest known algorithm for the general problem, which runs in $O(m\alpha(m+n, n)+n)$ time and $O(n)$ space, where α is a functional inverse of Ackermann's function. Used as a subroutine, the algorithm gives similar improvements in the efficiency of algorithms for solving several other problems, including two-processor scheduling, matching on convex graphs, finding nearest common ancestors off-line, testing a flow graph for reducibility, and finding two disjoint directed spanning trees. The algorithm obtains its efficiency by combining the fast algorithm for the general problem with table look-up on small sets, and requires a random access machine for its implementation. The algorithm extends to the case in which single-node additions to the union tree are allowed. The extended algorithm is useful in finding maximum cardinality matchings in nonbipartite graphs.

1. INTRODUCTION

The disjoint set union problem occurs frequently in the design of combinatorial algorithms [1, pp. 124–145; 14]. We shall formulate this problem as follows. We wish to carry out an intermixed sequence of three kinds of operations, which access and modify a collection of disjoint sets:

makeset(x): Create a new singleton set $\{x\}$ whose name is x. This operation is only allowed if x is in no existing set.

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

unite(x, y): Create a new set that is the union of the sets containing x and y.

* Research partially supported by the National Science Foundation, Grant MCS78-18909.

0022-0000/85 $3.00

571/30/2-5

The name of the new set is the name of the old set containing x. This operation assumes that x and y are initially in different sets, and destroys the old sets containing x and y.

The operations must be carried out on-line; that is, each one must be completed before the next one is known. We shall use n to denote the total number of elements (that is, the number of *makeset* operations) and m to denote the total number of unions and finds.

This problem has many applications and has been widely investigated. (See [4, 17, 27, 30].) The fastest known algorithm runs in $O(m\alpha(m+n, n)+n)$ time and $O(n)$ space, where α is a functional inverse of Ackermann's function [27, 31]. There are in fact a number of such fast algorithms, all minor variants of each other [31]. We shall call these algorithms *α-algorithms*. The α-algorithms run on a pointer machine [30] and, as one would expect, perform quite well in practice.

Nevertheless it is an interesting theoretical problem to determine whether there is a linear-time algorithm for disjoint set union. Under certain technical restrictions $\Omega(m\alpha(m+n, n)+n)$ is a lower bound on the worst-case running time of any set union algorithm on a pointer machine [30]. Thus to obtain a linear-time algorithm we must either confine our attention to a special case of set union or take advantage of the more powerful capabilities of random-access machines [1, pp. 12–19]. The result of this paper combines both of these ideas. We give an algorithm that runs in linear-time on a random-access machine for the special case of set union in which the structure of the unions, as defined by a "union tree," is known in advance. This case occurs in many applications, for each of which our result gives an improved algorithm. Although the reults may appear to be of only theoretical interest, experiments with an implementation of a restricted case of our algorithm indicate that in practice it is competitive with α-algorithms [12].

We solve the following problem, which we call *static tree set union*. We are given a (rooted) tree T of n nodes. Initially every node v of the tree is in a singleton set $\{v\}$ named v. We denote the parent of node v in the tree by $p(v)$; if v is the root of the tree, $p(v)$ has the special value **null**. We wish to perform on-line an intermixed sequence of *find* and *link* operations on the sets, where *find* is defined as before and *link* (v) is equivalent to *unite*$(p(v), v)$; we allow a *link* operation only on a node other than the root of the tree. Note that each set existing during the process induces a subtree of T; the name of the set is the root of the corresponding subtree.

This version of set union differs from the general problem in that the "union tree" T is known in advance. We can use our knowledge of T to precompute the answers to *find* operations on small sets. The resulting algorithm combines table look-up on small sets with an α-algorithm run on a universe of size $o(n)$. The algorithm needs $O(m+n)$ time and $O(n)$ space on a random-access machine with unit cost measure and $O(\log n)$[1] word length [1, pp. 12–19].

[1] Throughout this paper we use base-two logarithms.

We develop our algorithm in Section 2 of the paper. In Section 3 we extend the algorithm to the case in which the union tree can grow by single-node additions (*incremental tree set union*). The extended algorithm also runs in $O(m+n)$ time and $O(n)$ space. In Section 4 we list ten applications. A preliminary version of this paper appeared as [9].

2. Static Tree Set Union

To solve the static tree set union problem, we partition the nodes of T into *microsets*. This partition has nothing to do with the sets defined by the link operations; it is computed in a preprocessing step and remains fixed as the links and finds are executed. The microsets have three properties:

(a) Every microset contains fewer than b nodes, where b is a parameter to be chosen later;

(b) There are $O(n/b)$ microsets;

(c) If S is a microset, there is a node $r \notin S$ such that $p(v) \in S \cup \{r\}$ for every node $v \in S$. Node r is called the *root* of microset S. The set $S \cup \{r\}$ induces a subree of T with root r; thus S induces a collection of subtrees of T, all with a common parent. As a special case we allow r to be **null**; in this case S induces a subtree of T whose root is the same as the root of T.

We shall describe the set union algorithm in a top-down fashion, concurrently describing the data structures it uses. We number the microsets consecutively from one. Within each microset, we number the vertices consecutively from one. With each vertex v, we store *micro*(v), the number of the microset containing v, and *number*(v), the number of v within its microset. Thus the pair *micro*(v), *number*(v) uniquely identifies v. For each microset i we build a table *node*$(i, *)$ such that *node*(i, j) is the node in microset i with number j. (Note that *node* is *not* a 2-dimensional array, since the range of values of j depends on the value of i; rather, it is a collection of 1-dimensional arrays.) All the node tables together require a total of n words of memory since there is one entry per node.

To represent the collection of sets defined by the *link* operations, we mark the nodes that are set names. To store the marks, we use a table *mark*$(i, *)$ for each microset i, such that *mark*$(i, j) = 0$ if *node*(i, j) is marked, (i.e., it is a set name) and *mark*$(i, j) = 1$ otherwise. We allow the index j to have the range $1 \leqslant j < b$ for every value of i; if j is not the number of a node in microset i, *mark*$(i, j) = 0$. For any value of i, *mark*$(i, *)$ is a vector of $b-1$ bits. By choosing $b \leqslant w$, where w is the word length of the random-access machine, we can fit each mark table into a single computer word. We can also treat each mark table as an integer (whose binary representation is the sequence of bits in the table) and perform arithmetic on this integer in $O(1)$ time.

Our implementation of the *link* operation is such that its only effect is to alter the mark tables. Initially *mark*$(i, j) = 0$ for all microsets i and all values of j in the range

$1 \leqslant j < b$. (Initializing the mark table for a given microset i requires $O(1)$ time: we set $mark(i, *) = 0$.) We define *link* as follows:

```
1. procedure link(v);
2.     mark(micro(v), number(v)) := 1
3. end link;
```

Executing *link* takes $O(1)$ time.

The operation *find*(v) must return the nearest marked ancestor of v; that is, the nearest ancestor $node(i, j)$ of v such that $mark(i, j) = 0$. (We regard a node as an ancestor of itself.) To carry out *find*(v) we use a combination of two methods. To give access within microsets, we use the following procedure (whose implementation we describe later):

microfind(v): Return the nearest marked ancestor of v that is in the same microset as v. If there is no such node (the nearest marked ancestor of v is in another microset), return the root of the microset containing v.

To give access across microset boundaries, we maintain a collection of disjoint sets, called *macrosets*, whose elements are the roots of the microsets (excluding **null**). We manipulate the macrosets by means of the operations *makemacroset*, *macrofind*, and *macrounite*, implemented using any α-algorithm. We initialize the macrosets by executing *makemacroset*(v) for every microset root v, thus making each such root into a singleton macroset. This takes $O(1)$ time per root [31] for a total of $O(n/b)$ time.

We define *find* as follows. (Our program notation is essentially Dijkstra's guarded command language [3] augmented with procedures; we use a vertical bar "|" in place of Dijkstra's box "□".) (Roughly speaking, "**if** ... → ... | ..." corresponds to **if** ... **then** ... **else** ..."; "**do** ... → ..." corresponds to **while** ... **do** ...."

```
1.  function find(v);
2.    local x;
3.    x = v;
4.    if micro(x) ≠ micro(microfind(x)) →
5.      x := macrofind(microfind(x));
6.    do micro(x) ≠ micro(microfind(x)) →
7.        macrounite(macrofind(x), x);
8.        x := macrofind(x)
9.      od
10.   fi;
11.   return microfind(x)
12. end find;
```

LEMMA 1. *The find algorithm is correct.*

Proof. For any node x, if $micro(x) \neq micro(microfind(x))$, then $microfind(x)$ is the root of the microset containing x. It follows by induction that after step 5, the

node denoted by variable x in the program is always a microset root, and the macroset operations are executed only on microset roots. For any value of x, *microfind*(x) is an ancestor of x, and the only possible marked node on the tree path joining x and *microfind*(x) is *microfind*(x). Another induction shows that after any step, for any microset root y, *macrofind* (y) is the nearest ancestor y' of y such that y' is a microset root and the operation *macrounite*(*microfind*(y'), y') has not been perfomed. Furthermore the only possible marked node on the tree path joining y and *macrofind*(y) is *macrofind*(y). A third induction shows that, for the nodes denoted by variables x and v in the program, x is always an ancestor of v, and the only possible marked node on the tree path joining v and x is x. The correctness of the algorithm is immediate; termination is guaranteed by the fact that each successive value of x is a proper ancestor of the previous value. ▮

LEMMA 2. *If b is $\Omega(\log\log n)$ and each execution of* microfind *requires $O(1)$ time, then the total time for m intermixed* link *and* find *operations is $O(m+n)$.*

Proof. The *link* operations require a total of $O(n)$ time. The proof of Lemma 1 implies that just before step 7 in *find*, x, and *microfind*(x) are in different macrosets. Thus the total number of executions of step 7, summed over all the finds, is $O(n/b)$. It follows that the total time for all the finds is $O(m+n/b)$ plus the time for $O(n/b)$ *macrounite* and $m+O(n/b)$ intermixed *macrofind* operations. The time for the macroset instructions is $O((m+O(n/b))\,\alpha(m+O(n/b),\ O(n/b))+O(n/b))$, which is $O(m+n)$ if b is $\Omega(\log\log n)$ [27]. ▮

Remark. The proof of Lemma 2 does not require that b be $\Omega(\log\log n)$; much smaller values of b suffice [27].

As we have mentioned, initialization of the macrosets requires $O(n/b)$ time. We must still describe how to initialize the microsets and their data structures and how to carry out *microfind*. Let us first consider the latter problem. We need a compact way to represent the forest (in T) induced by a microset. With each microset i we store its root, denoted by *root*(i), and its parent table *parent*(i, *), defined by *parent*(i, j) $= k > 0$ if the parent of *node*(i, j) is *node*(i, k), *parent*(i, j) $= 0$ if the parent of *node*(i, j) is not in microset i or if *node*(i, j) has no parent (i.e., *node*(i, j) is the root of the tree T). For each i we allow all values of j in the range $1 \leqslant j < b$; if j is not the number of a node in microset i, *parent*(i, j) $= 0$.

Given the table *parent*(i, *) and the size of microset i, the structure of microset i is uniquely determined. (A given parent table can correspond to several microsets of different sizes but this is not important for our purposes.) A parent table requires $(b-1)\lceil \log b \rceil$ bits of storage; thus if we choose b such that $(b-1)\lceil \log b \rceil < w$ (recall that w is the word length of the random-access machine), we can fit each parent table into a single computer word. We can also treat a parent table as an integer, on which we can do arithmetic in $O(1)$ time. Note that not every integer in the range $[0 \cdots 2^{(b-1)\lceil \log b \rceil} - 1]$[2] corresponds to a possible parent table. Such an

[2] We use the notation $[j \cdots k]$ to denote the set of integers i such that $j \leqslant i \leqslant k$.

integer q corresponds to a possible parent table if and only if it has two properties when interpreted as a function from $[1 \cdots b-1]$ to $[0 \cdots 2^{\lceil \log b \rceil}-1]$: $q(j) \leqslant b-1$ for all j (i.e., q defines a directed graph on at most $b-1$ vertices), and $q^{(i)}(j) \neq j$ for all j and all $i \geqslant 1$ (i.e., the graph is acyclic), where $q^{(0)}(j)=j$ and $q^{(i)}(j)=q(q^{(i-1)}(j))$ for $i \geqslant 1$. (That is, $q^{(i)}$ is the function defined by applying q, i times.) Note that if $q^{(i)}(j)=0$ for some j, $q^{(i')}(j)$ is undefined for $i' > i$; we regard an undefined value as different from any integer. We interpret q as such a function by padding its binary representation on the left with zeros if necessary to obtain $(b-1)\lceil \log b \rceil$ bits and interpreting each group of $\lceil \log b \rceil$ consecutive bits as a function value.

To facilitate *microfind* operations we construct a 3-dimensional table *answer* (q, s, j). The indices q, s, and j range over $[0 \cdots 2^{(b-1)\lceil \log b \rceil}-1]$, $[0 \cdots 2^{b-1}-1]$, and $[1 \cdots b-1]$, respectively. (We intend q to be a parent table, s to be a mark table, and j to be a node number.) We define *answer*(q, s, j) as follows. We interpret q to be a function from $[1 \cdots b-1]$ to $[0 \cdots 2^{\lceil \log b \rceil}-1]$ and s to be a function from $[1 \cdots b-1]$ to $\{0, 1\}$. Then *answer*$(q, s, j)=k>0$ if q corresponds to a possible parent table (that is, q has the two properties given above) and there is an integer $i \geqslant 0$ such that $q^{(i)}(j)=k$, $s(k)=0$, and $s(q^{(i')}(j))=1$ for $0 \leqslant i' < i$; *answer* $(q, s, j)=0$ if q does not correspond to a possible parent table or if it does but $s(q^{(i)}(j))=1$ for all $i \geqslant 0$ such that $q^{(i)}(j) > 0$.

Given the answer table, we can define *microfind* as follows:

function *microfind*(v);
  local i, j, k;
  $i := micro(v)$; $j := number(v)$; $k := answer(parent(i, *), mark(i, *), j)$;
  return if $k=0 \rightarrow root(i) \mid k>0 \rightarrow node(i, k)$
end *microfind*;

Executing *microfind* takes $O(1)$ time, as required in the hypothesis of Lemma 2.

To construct the answer table, we iterate over the possible values of q and s. For each pair q, s, we can compute *answer*(q, s, j) for all j in the range $[1 \cdots b-1]$ in $O(b)$ time, as follows. We interpret q as a function, check its range, construct the graph whose vertices are the integers in $[1 \cdots b-1]$ and whose edges are the pairs $(i, q(i))$ such that $i, q(i) \in [1 \cdots b-1]$, and check the graph for cycles. If the range of q is contained in $[0 \cdots b-1]$ and the graph is acyclic, the graph must be a forest (with edges directed toward the tree roots), and we can compute *answer*(q, s, j) for all j by interpreting s as a function and traversing the forest in preorder. If not, we set *answer*$(q, s, j)=0$ for all j.

If we choose b so that $b2^{(b-1)(\lceil \log b \rceil + 1)} = O(n)$, we can construct the entire answer table in $O(n)$ time. Note that this construction is part of the initialization and only occurs once.

The last part of the algorithm to be filled in is the initialization of the microsets and their associated data structures. We divide the tree T into microsets by traversing it in postorder. For each node v, we maintain a count $d(v)$ of its remaining descendants (including itself) not yet placed in a microset. When placing a node in

a microset, we delete it from the tree. To decide when to form microsets, we apply the following steps to each node v in postorder (we assume that the children of each node are ordered arbitrarily).

Step 1. Let $d(v) = 1$ and let w be the first child of v (or **null** if there is no such child).

Step 2. While $d(v) < (b+1)/2$ and $w \neq$ **null**, replace $d(v)$ by $d(v) + d(w)$ and w by the next child of v after w (or **null** if there is no such child).

Step 3. If $d(v) < (b+1)/2$, process the next vertex after v in postorder. Otherwise form a new microset consisting of all descendants of the remaining children of v up to but not including w. Assign this microset the next available number, say i. Define the root of the microset to be v. Number the vertices in the microset consecutively from one, defining $micro(u)$ and $number(u)$ for each such node u. Build $node(i, *)$, $mark(i, *)$, and $parent(i, *)$. Delete all vertices in the microset from the tree. Let $d(v) = 1$. Go to Step 2.

After the tree root is processed, we form one last microset consisting of all the remaining vertices (including at least the tree root); the root of this microset is **null**.

For the procedure to be correct, we must have $b \geq 2$. Then in Step 2 it is always the case that $d(w) < (b+1)/2$, which means that in Step 3 $d(v) < b+1$, and every microset formed contains fewer than b nodes. (The last microset contains fewer than $(b+1)/2$ nodes.) Thus the microsets have property (a). (See the beginning of this section for a definition of properties (a), (b), and (c)). Every microset except the last contains at least $(b-1)/2$ nodes; thus the total number of microsets is at most $2n/(b-1)+1$, and the microsets have property (b). Property (c) is obvious by construction. Constructing a microset takes time proportional to the number of nodes it contains; thus the total time to construct the microsets is $O(n)$.

This completes our description of the algorithm. Let us summarize the constraints on b. We need $b \geq 2$ for the microset construction, $b = \Omega(\log \log n)$ for the time bound of Lemma 2 to apply, $b2^{(b-1)\lceil \log b \rceil + 1} = O(n)$ to construct the answer table in $O(n)$ time, and $(b-1)\lceil \log b \rceil \leq w$, where w is the word length, to fit each parent table and mark table into a single word of storage. Assuming $w = O(\log n)$, any choice of b such that $b = \Omega(\log \log n)$ and $b = O(\log n / \log \log n)$ will do. (As noted after the proof of Lemma 2, even much smaller values of b suffice.) Thus we obtain the following theorem:

THEOREM 1. *With an appropriate choice of b, the algorithm for static tree set union runs in $O(m+n)$ time with $O(n)$ preprocessing and uses $O(n)$ space.*

We have chosen the encoding scheme for the structure of a microset (its parent table) so that the method of this section will extend directly to incremental tree set union. If, however, static tree set union is the problem of interest, the encoding scheme can be improved somewhat. A parent table needs $O(b \log b)$ bits to represent a forest of $b-1$ nodes, but we can reduce the space to $O(b)$ with the

following representation: We number the vertices in a microset in preorder (with respect to the underlying forest) and use a table *forest*$(i, *)$, where *forest*(i, j) is the number of children of *node*(i, j). The forest is uniquely determined by this table. We encode *forest*$(i, *)$ by a bit vector formed by writing *forest*$(i, 1)$, *forest*$(i, 2)$,... in unary, separated by zeros; this vector contains at most $2b-3$ bits. The details of using this representation can be found in the preliminary version of this paper [9]; with it, we are able to choose a larger value of b; namely, $b = \lfloor \frac{1}{3} \log(n/\log n) \rfloor$.

A special case that deserves mention is when the union tree T is a path. This case has many applications (see Sect. 4) and allows the microset representation to be considerably simplified. We choose each microset to be a path of $b-1$ nodes (padding out one of the microsets with dummy nodes). This eliminates the need for either a *parent* or *forest* table, and makes the answer table 2-dimensional rather than 3-dimensional. Microset initialization is also simplified since there is no need for a depth-first search of T.

In practice in this case some computers allow the answer table to be eliminated entirely. When a microset is a path the answer table serves to locate the first zero bit beyond a given bit position in the mark table. On some computers this can be done in a few machine instructions, such as by a floating point normalize or a variable-length shift (if we exchange the roles of zero and one in the mark table).

3. Incremental Tree Set Union

We can extend the algorithm of Section 2 to the case in which the tree T is allowed to grow a node at a time. We define the *incremental tree set union* problem as follows. Initially T consists of a single node, the root. In addition to *find* and *link* operations, we allow operations of the following kind:

grow(v, w): Add w to T by making v its parent. This operation is only allowed if v is a node in T and w is a new node not in T.

Note that the number of *grow* operations is $n-1$.

Our algorithm for incremental tree set union is identical to the algorithm of Section 2 except in the construction of the microsets, which change over time. In addition to the data structures listed in Section 2, we maintain for each microset i the number of nodes it contains, denoted by *size*(i), and a list of its nodes. Initially there is only one microset, consisting of the root of T. We perform *grow*(v, w) as follows:

Step 1. Make w a child of v in T.

Step 2. Let $i = micro(v)$. Add w to microset i. Add 1 to *size*(i). If *size*$(i) = b$, go to Step 3. Otherwise, define *micro*(w) to be i, *number*(w) to be *size*(i), *node*$(i,$ *size*$(i))$ to be w, and *parent*$(i,$ *size*$(i))$ to be *number*(v). Stop.

Step 3. Split microset i into new microsets by constructing the tree induced by the *root*(i) and the nodes in microset i and using a postorder traversal like that

described in Section 2, with the following changes: Replace the test "$d(v) < (b+1)/2$" in Steps 2 and 3 by the test "$d(v) \leqslant (b+2)/4$". Let the last constructed microset have number i instead of giving it a new number. Omit *root*(i) from this last microset; this node remains the root of microset i. Initialize the mark table for every new microset by copying the appropriate part of the mark table for the old microset. Execute *makemacroset*(v) for every new microset root. (This does not include *root*(i).)

To verify the correctness of the method, note that every microset constructed by *grow* in Step 3 contains at most $b/2$ nodes. Step 3 only can occur after at least $b/2$ additions to an old microset. Thus Step 3 only occurs $O(n/b)$ times. Each new microset constructed in Step 3 except the last contains more than $(b-2)/4$ or at least $(b-1)/4$ nodes; thus at most $4b/(b-1)+1 = O(1)$ new microsets are constructed by one execution of Step 3. It follows that the total number of microsets ever constructed is $O(n/b)$, and the microsets have properties (a)–(c) of Section 2. The correctness of the entire method follows. (Note that the creation of new singleton macrosets does not affect the correctness of the find algorithm.) Each execution of *grow* requires $O(1)$ time if Step 3 is not executed and $O(b)$ time if it is executed, so the total time of all the *grow* operations is $O(n)$. We conclude:

THEOREM 2. *With an appropriate choice of b, the algorithm for incremental tree set union runs in $O(m+n)$ time with $O(n)$ preprocessing (to construct the* answer *table) and uses $O(n)$ space.*

Remark. Choosing b requires knowing n in advance. We can get around not knowing n in advance by choosing a new value of b and reinitializing the answer table and the macrosets each time n doubles. The total time for reinitializing is $O(1+2+4+\cdots n) = O(n)$ and Theorem 2 still holds.

4. APPLICATIONS

We conclude by listing ten applications of the algorithms in Sections 2 and 3. (Our list is intended to be illustrative, not inclusive.) For each problem in the list except one, we obtain a linear-time algorithm (improving the previously best almost-linear-time algorithm).

The first five examples use the algorithm of Section 2 applied to the special case in which the elements of the universe are the numbers from 1 to n and the unite operations implied by the links are of the form *unite*($i-1, i$) where $i \geqslant 2$. (Thus the union tree T is a path of n nodes.)

(1) *Two-processor scheduling.* The input consists of a collection of unit-time tasks with a partial order. the object is to schedule the tasks on two processors to minimize the last completion time. The algorithm of Gabow [8] runs in $O(m+n)$ time, improved from $O(m+n\alpha(n, n))$, when implemented using static tree set union.

Here n is the number of tasks and m is the number of explicit constraints defining the partial order.

(2) *Multiprocessor scheduling algorithms.* There are two related applications. The first is to computing a processing schedule from a priority list. The input is a collection of n unit-time tasks with a partial order of m constraints, a priority list giving a total order of the tasks, and a number of processors $p \leqslant n$. The object is to schedule the tasks so that the next task to begin is the first available task in the priority list. The algorithm of Sethi [25] runs in $O(m+n)$ time, improved from $O(m+n\alpha(n, n))$, using static tree set union.

The second application is optimum scheduling on an interval dag. The input is a collection of n unit-time tasks, an m-constraint partial order that is an interval dag, and a number of processors $p \leqslant n$. The object is to schedule the tasks to minimize the last completion time. Papadimitriou and Yannakakis give an $O(m+n)$-time algorithm to find a priority list defining an optimum schedule [7, 23]. Combining this with the above algorithm for priority list scheduling gives an $O(m+n)$-time scheduling algorithm, improved from $O(m+n\alpha(n, n))$.

(3) *The off-line min problem* [1, pp. 139–141]. The object is to maintain a set of integers in the range $[1 \cdots n]$ under two operations: *insert*(i), which adds element i to the set, and *extract min*, which deletes and returns the minimum element. If each integer is inserted only once and the entire sequence of operations is given off-line, static tree set union applies to solve this problem in $O(n)$ time, improved from $O(n\alpha(n, n))$.

(4) *Matching on convex graphs and scheduling with release times and deadlines.* These two problems are closely related. In the first, the object is to find a maximum matching on a convex, n-vertex bipartite graph. The algorithm of Lipski and Preparata [19] runs in $O(n)$ time, improved from $O(n\alpha(n, n))$, using static tree set union.

In the second problem, the input is a collection of unit-time tasks, each with an integer release time and deadline, and a number of processors $p \leqslant n$. The object is to schedule each task between its release time and deadline. Frederickson [5] gives an algorithm that involves the off-line min problem (3). Using our algorithm for off-line min, the running time of Frederickson's method is improved from $O(n\alpha(n, n))$ to $O(n)$. (The space needed is $O(d+n)$, where d is the largest deadline.)

(5) *VLSI channel routing.* The input to this problem is a set of 2-terminal nets; the desired output is a wire layout on a channel of least possible width. The algorithm of Preparata and Lipski [24] runs in $O(n)$ time, improved from $O(n\alpha(n, n))$.

The next three applications use the general case of static tree set union.

(6) *Nearest common ancestors.* Aho, Hopcroft, and Ullman [2, 29] give an $O(n+m\alpha(m+n, n))$-time, $O(n)$-space algorithm to compute the nearest common ancestors of m pairs of nodes in an n-node tree off-line. The algorithm of Section 2 improves this method to $O(n+m)$ time. Harel and Tarjan [10, 11] have also given

a linear-time algorithm for this problem. Their algorithm is more complicated than the one given here but extends to solve the "half-line" problem, in which the tree is fixed but the nearest common ancestor requests arrive on-line, in $O(m+n)$ time. (Our concept of microsets can be used to simplify their algorithm.)

(7) *Flow graph reducibility.* Static tree set union improves the method of Tarjan [26] for testing flow graph reducibility of an n-vertex, m-edge graph from $O((m\alpha(m, n))$ to $O(m)$ time. (In flow graphs $n = O(m)$.)

(8) *Two directed spanning trees.* Given a flow graph the object is to find two directed spanning trees with as few common edges as possible. Static tree set union improves the algorithm of Tarjan [28] for this problem from $O(m\alpha(m, n))$ to $O(m)$ time.

Our next-to-last application uses incremental tree set union.

(9) *Maximum cardinality matching on nonbipartite graphs.* The algorithm of Gabow [6] runs in $O(nm)$ time, improved from $O(nm\alpha(m, n))$, using incremental tree set union. Here n is the number of vertices and m the number of edges in the graph; we assume $n = O(m)$. A more efficient algorithm discovered by Micali and Vazirani [22] runs in $O(\sqrt{nm})$ time. Their algorithm uses disjoint set union; Micali and Vazirani state without proof that the "special structure of blossoms" implies a linear time bound if an appropriate α-algorithm is used [22, p. 21]. However the proof is complicated (over fifty pages long [21]). Using incremental tree set union gives the $O(\sqrt{n}\, m)$ time bound directly. Both matching algorithms use $O(n+m)$ space.

Our final example is a data manipulation problem that is a time-reversed version of disjoint set union.

(10) *The set-splitting problem.* Given an initial set consisting of the integers $\{1, 2, \ldots, n\}$, we wish to carry out an intermixed sequence of operations of the following two types:

find(i): Return the name of the set containing integer i.

split(i): Split the set containing integer i into two sets, one containing all integers less than i, the other all integers greater than or equal to i.

In their paper on disjoint set union [13] Hopcroft and Ullman describe an $O((m+n)\log^* n)$-time algorithm for this problem, where m is the number of operations and $\log^* n$ is the "iterated logarithm," the number of times the logarithm must be taken to obtain a number less than one. Using a variant of the static tree set union algorithm, we can solve this problem in $O(m+n)$ time. We shall sketch the idea.

We combine two methods: the table lookup scheme used for microsets in Sections 2 and 3, and a "relabel the smaller half" method. We can solve the set-splitting problem in $O(1)$ time per find plus $O(n \log n)$ time for all the splits by storing with each integer the name of the set containing it and, when splitting a set, renaming the half containing fewer elements. (See, e.g., [1, pp. 124–129].) To obtain an $O(m+n)$ time bound for set splitting, we use three levels of sets: *microsets*, *mez-*

zosets, and *macrosets*. Initially we partition the set $\{1, 2, ..., n\}$ into intervals called mezzosets, of size $\lfloor \log n \rfloor$. We partition each mezzoset into intervals called microsets, of size $\lfloor \log \log n \rfloor$.

The current partition of the integers into sets induces a partition of the unsplit mezzosets, another partition of the unsplit microsets, and a third partition of the split microsets. We use the table lookup method on the split microsets and the "relabel the smaller half" method on the unsplit microsets and also on the unsplit mezzosets. To find integer i, we check whether its microset or mezzoset has split and find it using the appropriate method. To split a set at i, we split at most two microsets and two mezzosets, and update the three levels of the data structure appropriately. When the details of this method are worked out, the running time is seen to be $O(m+n)$. (For a detailed description and analysis of a similar method for a different problem, see [11].) Using the representation of microsets via *forest* tables outlined at the end of Section 2, we can avoid using mezzosets, thus obtaining an $O(m+n)$ time bound using only a 2-level data structure (see [9].)

In conclusion we note that there are important applications of disjoint set union that our algorithm does not handle, such as checking the equivalence of two deterministic finite automata [1, pp. 143–145]. The path compression technique has additional applications [29] to which our table look-up approach does not seem to apply. Nevertheless the special case of disjoint set union that we have been able to handle is significant both in theory and in practice.

Since the appearance of the preliminary version of this paper [9], further applications and extensions have been discovered by other authors. Imai and Asano [15] have used our set union method to quickly carry out breadth-first and depth-first searches of intersection graphs of vertical and horizontal line segments in the plane. Their methods extend to the solution of various problems for this class of graphs. A similar algorithm for breadth-first search was independently discovered by Lipski [20]. Imai and Asano [16] also extended our set-splitting algorithm (application (10)) to allow incremental insertion of new items. They use this algorithm as a subroutine in dynamic segment intersection search. Harel [32] has extended our ideas to give an $O(m)$ time bound for computing dominators in an n-vertex, m-edge graph, improved from $O(m\alpha(m, n))$ [18]. (Here we assume $n = O(m)$.)

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Annalysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
2. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, On finding lowest common ancestors in trees, *SIAM J. Comput.* **5** (1976), 115–132.
3. E. W. DIJKSTRA, "A Discipline of Programming," Prentice–Hall, Englewood Cliffs N. J., 1976.
4. J. DOYLE AND R. L. RIVEST, Linear expected time of a simple union-find algorithm, *Inform. Process. Lett.* **5** (1976), 146–148.
5. G. N. FREDERICKSON, Scheduling unit-time tasks with integer release times and deadlines, *Inform. Process. Lett.* **16** (1983), 171–173.

6. H. N. GABOW, An efficient implementation of Edmonds' algorithm for maximum matching on graphs, *J. Assoc. Comput. Mach.* **23** (1976), 221–234.
7. H. N. GABOW, A linear-time recognition algorithm for interval dags, *Inform. Process. Lett.* **12** (1981), 20–22.
8. H. N. GABOW, An almost-linear algorithm for two-processor scheduling, *J. Assoc. Comput. Mach.* **29** (1982), 766–780.
9. H. N. GABOW AND R. E. TARJAN, A linear-time algorithm for a special case of disjoint set union *in* "Proc. 15th Annual ACM Sympos. on Theory of Computing," 1983, pp. 246–251.
10. D. HAREL, A linear time algorithm for the least common ancestors problem, *in* "Proc. 21st Annual Sympos. on Found. of Comput. Sci.," 1980, pp. 308–319.
11. D. HAREL AND R. E. TARJAN, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13** (1984), 338–355.
12. B. HAVENS, "Experiments on an Asymptotically Optimum, Special Purpose Set Merging Algorithm," M. S. thesis, Department of Computer Science, University of Colorado, Boulder, Clo., 1983.
13. J. E. HOPCROFT AND J. D. ULLMAN, Set merging algorithms, *SIAM J. Comput.* **2** (1973), 294–303.
14. E. HOROWITZ AND S. SAHNI, "Fundamentals of Computer Algorithms," Computer Science, Potomac, Md., 1978.
15. H. IMAI AND T. ASANO, Efficient algorithms for geometric graph search problems, *J. Algorithms*, to appear.
16. H. IMAI AND T. ASANO, Dynamic segment intersection search with applications, *in* "Proc. 25th Annual IEEE Sympos. on Found. of Comput. Sci.," 1984, pp. 393–402.
17. D. E. KNUTH AND A. SCHÖNHAGE, The expected linearity of a simple equivalence algorithm, *Theoret. Comput. Sci.* **6** (1978), 281–315.
18. T. LENGAUER AND R. E. TARJAN, A fast algorithm for finding dominators in a flowgraph, *ACM Trans. Programming Lang. Systems* **1** (1979), 121–141.
19. W. LIPSKI, JR. AND F. . PREPARATA, Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems, *Acta Inform.* **15** (1981), 329–346.
20. W. LIPSKI, JR., An $O(n \log n)$ Manhattan path algorithm, *Inform. Process. Lett.* **19** (1984), 99–102.
21. S. MICALI, Private communication, May 1982.
22. S. MICALI AND V. V. VAZIRANI, An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs, *in* "Proc. 21st Annual IEEE Sympos. on Found. of Comput. Sci., 1980, pp. 17–27.
23. C. H. PAPADIMITRIOU AND M. YANNAKAKIS, Scheduling interval-ordered tasks, *SIAM J. Comput.* **8** (1979), pp. 405–409.
24. F. P. PREPARATA AND W. LIPSKI, JR., Three layers are enough, *in* "Proc. 23rd Annual IEEE Sympos. on Found. of Comput. Sci., 1982, pp. 350–357.
25. R. SETHI, Scheduling graphs on two processors, *SIAM J. Comput.* **5** (1976), 73–82.
26. R. E. TARJAN, Testing flow graph reducibility, *J. Comput. System Sci.* **8** (1974), 355–365.
27. R. E. TARJAN, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22** (1975), 215–225.
28. R. E. TARJAN, Edge-disjoint spanning trees and depth-first search, *Acta Inform.* **6** (1976), 171–185.
29. R. E. TARJAN, Applications of path compression on balanced trees, *J. Assoc. Comput. Mach.* **26**, No. 4 (1979), 690–715.
30. R. E. TARJAN, A class of algorithms which require non-linear time to maintain disjoint sets, *J. Comput. System Sci.* **18** (1979), 110–127.
31. R. E. TARJAN AND J. VAN LEEUWEN, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.* **31** (1984), 245–281.
32. D. HAREL, A linear algorithm for finding dominators in flow graphs and related problems, *in* "Proc. 17th Annual ACM Sympos. on Theory of Computing," 1985, to appear.