A LINEAR-TIME ALGORITHM FOR A SPECIAL CASE OF DISJOINT SET UNION

by

Harold N. Gabow*
University of Colorado at Boulder
Boulder, Colorado

and

Robert Endre Tarjan
Bell Laboratories
Murray Hill, New Jersey

ABSTRACT

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 a number of other problems, including two-processor scheduling, the off-line min problem, 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 a 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 on nonbipartite graphs.

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

1. INTRODUCTION

The disjoint set union problem occurs frequently in the design of combinatorial algorithms [AHU 1974, pp. 124-145, HS]. 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. The name of the new set is the name of the old set containing x. This operation 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 unites and finds.

This problem has many applications and has been widely investigated (see [T1975]; also [DR], [KS], [T1979b]). The fastest known algorithm for the disjoint set union problem runs in $O(m\alpha(m+n, n)+n)$ time and $O(n)$ space, where α is a functional inverse of Ackermann's function [T1975, TV1982]. There are in fact a number of such fast algorithms, all minor variants of each other [TV1982]. We call these algorithms *α-algorithms*. The α-algorithms run on a pointer machine [T1979b] 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 [T1979b]. 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 [AHU1974, 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 results may appear to be of only theoretic interest, experiments with an implementation of a restricted case of our algorithm indicate that in practice it is competitive with α-algorithms and often outperforms them.

We solve the following problem, called *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 on any node v except 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 finds 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 uniform cost measure and $\log n$ [1] word length [AHU1974, pp. 12-19].

We develop our algorithm in Section 2 of the paper. In Section 3 we sketch an extension of 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. Section 4 lists eleven applications.

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 subtree of T with root r; thus S induces a forest consisting of subtrees with a common parent in T. As a special case we allow r to be *null*; in this case S induces a subtree of T whose root is 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, according to a preorder for the induced forest (the microset is a forest by (c)). With each vertex v, we store *micro(v)*, the number of the microset containing v, and *number(v)*, the preorder 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 two-dimensional array, since the range of values of j depends on the value of i; rather, it is a collection of one-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 $1 \le j < b$ 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 $b \le w$ 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 $1 \le j < b$. (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. (To do this we precompute the powers of two, 2^j, $0 \le j < b$. Then Step 2 can be implemented by a simple sequence of arithmetic operations.)

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*. We initialize the macrosets by executing *makemacroset(v)* for every microset root v, thus making each such root into a singleton macroset.

There are several ways to implement the operations on macrosets. One is to use any α-algorithm. This will be the most desirable choice in Section 3, for the incremental version of the algorithm. Here it suffices to use a simpler algorithm, which merely relabels the smaller set in a union [AHU, pp. 124-129]. The time for m operations

[1] Throughout this paper log denotes logarithm to the base two.

on a universe of size n is $O(m + n \log n)$.

We define *find* as follows. (Our program notation is essentially Dijkstra's guarded command language [D 1976] augmented with procedures; we use a vertical bar "|" in place of Dijkstra's box " ".)

```
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(microfind(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 performed. 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 $\Omega(\log n)$ 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 the *macrounite* and *macrofind* operations. There are $m+O(n/b)$ of these, executed on a universe of size $O(n/b)$. Hence the time is $O(m+O(n/b) + O(n/b) \log O(n/b))$, which is $O(m+n)$ if b is $\Omega(\log n)$. ∎

If an α-algorithm is used for the macroset operations a similar estimate shows the time is linear. Actually an α-algorithm allows b to assume values *much* smaller than $\Omega(\log n)$. For instance in Section 3, b will be $\Omega(\log \log n)$ and the linear time bound still holds [T 1975].

Initializing 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 $root(i)$. The topology of the forest is represented in a table $forest(i, *)$, where $forest(i,j)$ is the number of children of $node(i,j)$. Recall that the forest is numbered in preorder. Hence it is uniquely determined by $forest(i, *)$, and in fact it can be constructed from $forest(i, *)$ in linear ($O(b)$) time.

We use the following *encoding scheme* to represent $forest(i, *)$ by a bit vector: An entry $forest(i,j) = c$ is encoded as 10^c [2], and these entries are concatenated together in order of increasing i. The resulting bit vector has length less than twice the number of nodes in the forest, i.e., at most $2b-3$ bits. So if we choose b so that $2b-3 < w$ we can fit each *forest* table into a single computer word. Hence we can treat a *forest* table as an integer on which we can do arithmetic in $O(1)$ time. In particular given such an integer we can construct $forest(i, *)$, and hence the forest itself, in $O(b)$ time. Conversely given the forest we can construct the corresponding bit vector in $O(b)$ time.

To facilitate *microfind* operations we construct a three-dimensional table $answer(f, a, j)$. The indices f, a and j range over $[0..2^{2b-3}-1]$, $[0..2^{b-1}-1]$, and $[1..b-1]$, respectively.[3] We interpret f as a forest table, a as a mark table, and j as a node number. We define $answer(f, a, j)$ to be $k > 0$ if f is a possible forest table and in the forest for f, node k is the nearest ancestor of node j with $a(k) = 0$; $answer(f, a, j)$ is 0 if f is not a possible forest table or if it is but no node k exists.

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

```
1.  function microfind(v);
2.      local i, j, k;
3.      i := micro(v); j := number(v); k :=
        answer(forest(i, *), mark(i, *), j);
4.      return if  k = 0 → root(i)  |
                   k > 0 → node(i, k)
5.  end microfind;
```

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

To construct the answer table, we iterate over all possible pairs of values f in $[0..2^{2b-3}-1]$ and a in $[0..2^{b-1}-1]$. For each pair f, a, we can compute $answer(f,a,j)$ for all j in the range $[1..b-1]$ in $O(b)$ time, as follows. We interpret f according to the encoding scheme for forests. If f does not represent a forest the entries in *answer* are 0. Otherwise we construct the forest for f. We interpret a as a mark table for f. Then we compute $answer(f, a, j)$ for all j by traversing the forest in preorder, always remembering the most previously reached node k with $a(k) = 0$. Details are left to the reader.

If we choose b so that $b2^{3b-4} = O(n)$, we can construct the entire answer table in $O(n)$ time. Note that this construction is part of the initialization and only occurs once. This choice of b also implies that the answer table uses $O(n)$ space.

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).

---

[2] 0^c is a vector of c zeroes.

[3] $|j..k|$ denotes the set of integers i such that $j \leq i \leq k$.

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 $d(v) < \frac{b+1}{2}$ 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 $d(v) < \frac{b+1}{2}$, process the next vertex 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 u in the microset consecutively from one in preorder, defining *micro*(u) and *number*(u) for each such u. Build *node*$(i,*)$, *mark*$(i,*)$, and *forest*$(i,*)$ (the last two encoded as bit vectors). 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 $d(w) < \frac{b+1}{2}$. Hence in Step 3 $d(v) < b+1$, and every microset formed contains fewer than b nodes. (The last microset contains fewer than $\frac{b+1}{2}$ nodes.) Thus the microsets have property (a). (See the beginning of this section for the definition of properties (a), (b), and (c)). Every microset except the last contains at least $\frac{b-1}{2}$ nodes. Thus the total number of microsets is at most $\frac{2n}{b-1} + 1$, 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, $b = \Omega(\log n)$ for the time bound of Lemma 2 to apply, $b2^{2b-4} = O(n)$ to construct the answer table in $O(n)$ time and space, and $2b-3 < w$, where w is the word length, to fit each forest table and mark table into a single word of storage. Assuming $w = \log n$, the choice $b = \left\lfloor \frac{1}{3}\log\left(\frac{n}{\log n}\right) \right\rfloor$ is satisfactory. (As noted after the proof of Lemma 2, much smaller values of b suffice when an α-algorithm is used. 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. ▪

A special case that deserves mention is when the union tree T is a path. This case has many applications (see Section 4) and is somewhat simpler than the general case. Each microset can be taken as a path of $b-1$ nodes. (The last microset can be padded out with dummy nodes.) This eliminates the need for the forest encoding scheme, and the answer table becomes two-dimensional instead of three. In addition the microset initialization is simplified since there is no need for a depth-first search of T.

In practice some computers allow the answer table to be eliminated entirely: When the microset is a path the answer table serves to locate the first zero bit beyond a given bit position in a mark table. Some computers can do this in one or two machine instructions. For instance in the CDC Cyber family the floating point Normalize instruction executes in *constant* time [Th]. If we reverse the roles of zero and one in the mark table we can extract the *answer* information from a mark table in constant time. Hence there is no need for the answer table or the preprocessing associated with it.

The algorithm for path union trees was implemented in the C programming language and run on a VAX 11/780. (A two-dimensional answer table was used.) The algorithm was compared to the usual α-algorithm based on weighted union and path compression [T1975]. Data was generated both randomly and in ways simulating set union in the applications of Section 4. The static tree algorithm was faster in many experiments. For instance on random data with n ranging from 200 to 1000, the time for the static tree algorithm was .6 that of the α-algorithm when there was one *find* per *unite*, and .7 when there were two *finds* per *unite* (the common cases). The static tree algorithm required less data space (eg., 1160 words versus 3000 words for n = 1000). More details are in [Hav]. It is premature to draw conclusions from this limited experience, but these results certainly do not rule out the possibility of our algorithm being useful in practice.

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 similar to the algorithm in Section 2, with two main differences. First the forest encoding scheme for microsets cannot be used, since a *grow* operation changes preorder numbers. Instead we represent the topology of a microset by a parent table. The parent table can be stored in one computer word if we choose b so that $(b-1)\lceil \log b \rceil < w$. This gives a slight increase in the size of the answer table for a given b. However choosing b as $\Theta\left(\frac{\log n}{\log \log n}\right)$ (or even smaller) and using an α-algorithm for macrosets allows the linear time bound to be maintained.

The second difference is in the construction of microsets, which change over time. The algorithm for *grow* adds a node to a microset. When an addition causes a microset to have b nodes, it is split into $O(1)$ microsets. The splitting operation is similar to the microset construction in Section 2. Details can be found in [GT].

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. ▪

4. APPLICATIONS

We conclude by listing eleven applications of our algorithms. (The list is intended to be illustrative, not inclusive.) For each problem except one, we obtain a linear-time algorithm (improving the previously best almost-linear-time algorithm).

The first five applications use static tree set union in the special case where 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 [G1982] 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) *p-processor scheduling algorithms*. There are two related applications. The first is computing a schedule from a priority list. The input is a collection of unit-time tasks with a partial order, a priority list giving a total order of the tasks, and a number of processors $p \le n$. 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 [S] 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 unit-time tasks with a partial order that is an interval dag, and a number of processors $p \le n$. The object is to schedule the tasks to minimize the last completion time. Papadimitriou and Yannakakis show that the priority list of an optimum schedule can be found on $O(m+n)$ time [PY, G1981]. Using the above algorithm for priority lists, their method runs in $O(m+n)$ time, improved from $O(m+n\alpha(n,n))$.

(3) *The off-line min problem* [AHU1974, pp. 139-141]. The object is to maintain a set of integers in the range $[1..n]$ 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 cardinality matching on a convex bipartite graph. The algorithm of Lipski and Preparata [LP] runs in $O(n)$ time, improved from $O(n\alpha(n,n))$, using static tree set union. Here n is the number of vertices.

In the second problem, the input is a collection of unit-time tasks, each having an integer release time and deadline, and a number of processors $p \le n$. The object is to schedule each task between its release time and deadline. Frederickson [F] gives an algorithm that uses the off-line min problem. Using the algorithm of application (3) the run time is $O(n)$, improved from $O(n\alpha(n,n))$. (The space is $O(D+n)$, where D is the largest deadline.)

(5) *VLSI channel routing*. The input is a set of n two-terminal nets. The output is a wire layout on a channel of least possible width. The algorithm of Preparata and Lipski [PL1982] runs in $O(n)$ time, improved from $O(n\alpha(n,n))$.

The next four applications use static tree set union in the general case.

(6) *Nearest common ancestors*. Aho, Hopcroft, and Ullman [AHU1976, T1979a] give an $O(m+n\alpha(m+n,n))$-time, $O(n)$-space algorithm to compute the nearest common ancestors of m pairs of nodes in an n-node tree off-line. Static tree set union improves this method to $O(m+n)$ time. Harel and Tarjan [H, HT1982] 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.

(7) *Flow graph reducibility*. Static tree set union improves the method of Tarjan [T1974] 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 [T1976] for this problem from $O(m\alpha(m,n))$ to $O(m)$ time.

(9) *Separators for chordal graphs*. Given a chordal graph the object is to find a good separator (i.e., one with $O(\sqrt{m})$ vertices). Gilbert and Rose [GR] present an $O(n+m\alpha(m,n))$-time algorithm. With a slight change their algorithm can use incremental tree set union. The result is an $O(n+m)$-time algorithm.

The next application uses incremental tree set union.

(10) *Matching on nonbipartite graphs*. The algorithm of Gabow [G1976] 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 [MV] runs in $O(\sqrt{n}m)$ 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 [MV p. 21]. However the proof is complicated (over fifty pages long [M]). Using incremental tree set union gives the $O(\sqrt{n}m)$ time bound directly. Both matching algorithms use $O(m)$ space.

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

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

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.

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

In their paper on disjoint set union [HU], Hopcroft and Ullman describe an $O((m+n)\log^* n)$-time algorithm, 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 algorithm, we can solve this problem in $O(m+n)$ time. The method is as follows.

First note that we can solve the set-splitting problem in $O(1)$ time per find plus $O(n \log n)$ time for all the splits, by the "relabel-the-smaller-half" method: With each integer i we store the name of the set containing it; when splitting a set, we rename the half containing fewer elements (as in Section 2, and [AHU, pp. 124-129].)

To obtain an $O(m+n)$ time bound for set splitting we combine this method with the table look-up method of Section 2. We partition the set $[1..n]$ into microsets that are intervals of $b-1$ consecutive integers. Each microset has a root in the next microset. The n/b roots are placed in a universe of macrosets, that is processed by the relabel-the-smaller-half method. The algorithms for *split* and *find* are similar to those of Section 2. One change is that the *split* operations update the macroset universe (as contrasted with Section 2 where finds update the macroset universe). Choosing $b = \left\lfloor \log\left(\frac{n}{\log n}\right) \right\rfloor$ gives

a linear algorithm. (Details of a similar method for a different problem can be found in [HT].)

In conclusion we note that there are important applications of set merging that our algorithm does not handle (e.g., checking the equivalence of two DFA's [AHU p. 143-5], computing dominators in a flow graph [LT] and related problems [T1979a]). We have not been able to extend our algorithm to the general problem. Nonetheless the special case we treat appears to be significant, both in theory and applications.

REFERENCES

[AHU1974] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[AHU1976] A.V. Aho, J.E. Hopcroft, J.D. Ullman, "On finding lowest common ancestors in trees," *SIAM J. Comp. 5* (1976), pp. 115-132.

[D1976] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DR] J. Doyle and R.L. Rivest, "Linear expected time of a simple union-find algorithm," *Inf. Proc. Letters 5*, 1976, pp. 146-148.

[F] G.N. Frederickson, "Scheduling unit-time tasks with integer release times and deadlines," Tech. Rept. CS-81-27, Dept. of Computer Sci., Penn. State Univ., University Park, PA, 1982.

[G1976] H.N. Gabow, "An efficient implementation of Edmonds' algorithm for maximum matching on graphs," *J. ACM 23* (1976) pp. 221-234.

[G1981] H.N. Gabow, "A linear-time recognition algorithm for interval dags," *Inf. Proc. Letters 12* (1981), pp. 20-22.

[G1982] H.N. Gabow, "An almost-linear algorithm for two-processor scheduling," *J. ACM*, 29, 3 (1982), pp. 766-780.

[GR] J.R. Gilbert and D.J. Rose, "A separator theorem for chordal graphs," Tech. Rept. TR 82-523, Dept. of Comp. Sci., Cornell Univ., Ithaca, New York, 1982.

[GT] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," Bell Laboratories Report, July 1982.

[H] D. Harel, "A linear time algorithm for the least common ancestors problem," *Proc. 21st Annual Symp. on Found. Comp. Sci.* (1980), pp. 308-319.

[Hav] B. Havens, "Experiments on an asymptotically optimum, special purpose set merging algorithm," M.S. Thesis, Dept. of Computer Sci., Univ. of Colorado, Boulder, CO, 1983.

[HS] E. Horowitz, and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

[HT1982] D. Harel, R.E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.*, submitted.

[HU1973] J.E. Hopcroft and J.D. Ullman, "Set merging algorithms," *SIAM J. Comput.* 2, 4, 1973, pp. 294-303.

[KS] D.E. Knuth and A. Schonhage, "The expected linearity of a simple equivalence algorithm," *Theoretical Comp. Sci.* 6 (1978), pp. 281-315.

[LP] W. Lipski, Jr. and F.P. Preparata, "Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems," *Acta Informatica* 15 (1981), pp. 329-346.

[LT] T. Lengauer and R.E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. on Prog. Lang. and Systems* 1, 1, 1979, pp. 121-141.

[M] S. Micali, private communication, May 1982.

[MV] S. Micali, and V.V. Vazirani, "An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs," *Proc. 21st Annual Symp. on Found. of Comp. Sci.* (1980), pp. 17-27.

[PY] C.H. Papadimitriou and M. Yannakakis, "Scheduling interval-ordered tasks," *SIAM J. Comput. 8, 3, 1979, pp. 405-409.*

[PL] F.P. Preparata and W. Lipski Jr., "Three layers are enough," *Proc. 23rd Annual Symp. on Foundations of Comp. Sci.*, 1982, pp. 350-357. Also personal communication, F.P. Preparata.

[S] R. Sethi, "Scheduling graphs on two processors," *SIAM J. Comp.* 5 (1976), pp. 73-82.

[SR] R.E. Stearns and D.J. Rosenkrantz, "Table machine simulation," *Proc. 10th Annual Symp. on Switching and Automata Theory*, 1969, pp. 118-128.

[T1974] R.E. Tarjan, "Testing flow graph reducibility," *J. Comp. Sys. Sci* 9 (1974), pp. 355-365.

[T1975] R.E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM 22* (1975), pp. 215-225.

[T1976] R.E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica* 6 (1976), pp. 171-185.

[T1979a] R.E. Tarjan, "Applications of path compression on balanced trees," *J. ACM 26*, 4 (1979), pp. 690-715.

[T1979b] R.E. Tarjan, "A class of algorithms which require non-linear time to maintain disjoint sets," *J. Comp. Sys. Sci.* 18 (1979), pp. 110-127.

[Th] J.E. Thornton, *Design of a computer: The Control Data 6600*, Scott, Foresman and Co., Glenview, Illinois, 1970.

[TV1982] R.E. Tarjan, J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. ACM*, submitted.