SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

ON FINDING LOWEST COMMON ANCESTORS IN TREES*

A. V. AHO†, J. E. HOPCROFT‡ AND J. D. ULLMAN¶

Abstract. Trees in an n-node forest are merged according to instructions in a given sequence, while other instructions in the sequence ask for the lowest common ancestor of pairs of nodes. We show that any sequence of $O(n)$ such instructions can be processed "on-line" in $O(n \log n)$ steps on a random access computer.

If we can accept our answer "off-line", that is, no answers need to be produced until the entire sequence of instructions has been seen, then we may perform the task in $O(n\alpha(n))$ steps, where $\alpha(n)$ is the very slowly growing inverse Ackermann function defined in [14].

A third algorithm solves a problem of intermediate complexity. We require the answers on-line, but we assume that all tree merging instructions precede the information requests. This algorithm requires $O(n \log \log n)$ time.

We apply the first on-line algorithm to a problem in code optimization, that of computing immediate dominators in a reducible flow graph. We show how this computation can be performed in $O(n \log n)$ steps.

Key words. algorithms, computational complexity, graphs, trees, first common ancestor, code optimization, dominators

1. Introduction. Suppose that we are running the following genealogy service. During the course of a day, we receive new information concerning the ancestry relationships among a fixed set of men. (E.g., "B is a son of A.") We also receive requests asking for the closest common male ancestor of pairs of men. (E.g., "Who is the most recent common male parent of C and D?") Our problem is to process each new request in turn using the most current information.

We can abstract our problem as follows. We have n nodes in a finite set of trees (see [1] for definitions), hereafter called a *forest*. We receive a sequence of instructions to execute. The instructions are of two types:

1. The instruction LINK(u, v) makes node u a son of node v. We assume that at the time this instruction is received, nodes u and v are on different trees and that u is a root. Thus, after executing this instruction, the nodes will remain a forest.

2. The instruction LCA(u, v) prints the lowest common ancestor of nodes u and v.

Example 1. Suppose that we initially have a forest consisting of eight isolated

* Received by the editors May 3, 1973.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Department of Computer Science, Cornell University, Ithaca, New York 14850. The work of this author was supported by the Office of Naval Research under Grant N00014-67-A-0077-0021.

¶ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540. The work of this author was supported in part by the National Science Foundation under Grant GJ-1052.

nodes $u_1, u_2, \cdots, u_8$ and we receive the following sequence of instructions.

$$\begin{aligned}&\text{LINK}(u_1, u_2)\\&\text{LINK}(u_3, u_4)\\&\text{LINK}(u_5, u_6)\\&\text{LINK}(u_7, u_8)\\&\text{LINK}(u_2, u_4)\\&\text{LINK}(u_6, u_8)\\&\text{LCA}(u_5, u_7)\\&\text{LINK}(u_4, u_6)\\&\text{LCA}(u_2, u_3).\end{aligned}$$

When the instruction $\text{LCA}(u_5, u_7)$ is received, the forest is as shown in Fig. 1(a). Thus u_8, the lowest common ancestor of u_5 and u_7, is printed.

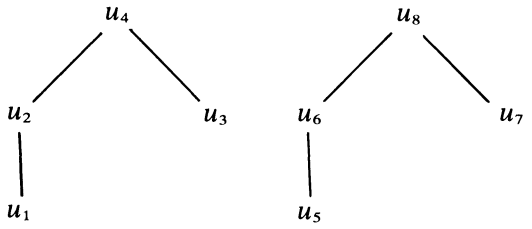


FIG. 1(a). *Tree structures after* $\text{LCA}(u_5, u_7)$ *instruction*

Fig. 1(b) shows the forest when $\text{LCA}(u_2, u_3)$ is executed. This instruction causes u_4 to be printed.

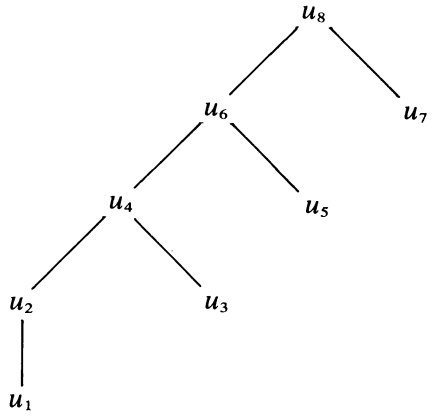


FIG. 1(b). *Tree structure after* $\text{LCA}(u_2, u_3)$ *instruction*

In this paper we shall consider the problem of executing a sequence of $O(n)$ LINK and LCA instructions on a forest with n nodes. We shall hereafter refer to this sequence as σ. If we execute $O(n)$ LINK instructions, trees with paths of length $n - 1$ can develop. Consequently, if we execute the LCA instructions in the obvious way, we could spend $O(n)$ time on each LCA instruction, or $O(n^2)$ time in total when there are $O(n)$ LCA instructions.

We shall first give an on-line algorithm that requires $O(n \log n)$ steps to execute σ. We shall also provide an asymptotically faster off-line algorithm and an algorithm of intermediate complexity which solves an intermediate problem. We then apply the on-line algorithm to compute the immediate dominators of an n-node reducible program flow graph in $O(n \log n)$ steps.

2. A useful data structure for forests. In our on-line algorithm, we define two forests with information attached to the various nodes. These two forests are used together to find the least common ancestors. To distinguish these forests from the actual forest that would be constructed by the LINK instructions, we shall refer to the latter forest as the *implied forest*.

The first defined forest, which we call the *A-forest*, has the same structure as the implied forest. For the A-forest we shall maintain an array ANCESTOR[u, i], where u is a node and i an integer such that $0 \leqq i < \log n$.[1] At all times, ANCESTOR[u, i] will be either the (2^i)th ancestor of node u in the implied forest or will be undefined. ANCESTOR[u, i] could be undefined even though u has a (2^i)th ancestor, since we shall not compute ancestor information until needed. Maintenance of ancestor information will be discussed in § 4.

The second forest, called the *D-forest*, has nodes grouped into the same trees as the implied forest, but the internal structure of corresponding trees will in general be different. The sole purpose of the D-forest is to keep track of the depth of nodes in the implied forest.

In what follows, we shall refer to a node in the A- and D-forests merely by its name in the implied forest. We trust no confusion will result. It should be borne in mind, however, that "the depth of u" always refers to the depth of u in the implied forest or, equivalently, to the depth of u in the A-forest.

3. Maintaining the D-forest. Our first algorithm uses the D-forest to compute the depth of nodes in the implied forest. We attach an integer WEIGHT[u] to each node u in the D-forest. To find the depth of a node, we find the representative of the node in the D-forest and trace the path from this node to its root, summing the weights of the nodes along this path. Then, except for the root, we make each node along this path be a son of the root, updating the weights of the nodes appropriately.

ALGORITHM 1.

procedure DEPTH(u):
begin
1. Find the path from node u to its root in the D-forest. Suppose $u_k, u_{k-1}, \cdots, u_0$ is that path, where u_0 is the root and u_k is u.
2. $sum \leftarrow$ WEIGHT[u_0] + WEIGHT[u_1] + $\cdots$ + WEIGHT[u_k].
3. Make each of $u_2, u_3, \cdots, u_k$ a son of u_0 in the D-forest.
4. **for** $i = 2$ **until** k **do**
   WEIGHT[u_i] $\leftarrow$ WEIGHT[u_i] + WEIGHT[u_{i-1}];
5. **return** *sum*

end

[1] All logarithms in this paper are to the base 2.

The root of each tree in the D-forest also has an associated COUNT, giving the number of nodes in the tree. Initially each node in the D-forest is in a tree by itself, having a COUNT of 1 and a WEIGHT of 0.

We now give an algorithm to merge the two corresponding trees in the D-forest when a LINK(u, v) instruction is executed.

ALGORITHM 2 (Merging trees in the D-forest).

1. Find the roots x and y of the trees holding u and v, respectively, by executing steps 1, 3 and 4 of Algorithm 1.

2. Compute DEPTH(v) using Algorithm 1.

3. If COUNT[x] $\leqq$ COUNT[y], then make x a son of y in the D-forest and do the following:

COUNT[y] $\leftarrow$ COUNT[y] + COUNT[x];

WEIGHT[x] $\leftarrow$ WEIGHT[x] + DEPTH[v] + 1 $-$ WEIGHT[y].

4. Otherwise, if COUNT[x] $>$ COUNT[y], make y a son of x in the D-forest and counts appropriately.

COUNT[x] $\leftarrow$ COUNT[y] + COUNT[x];

WEIGHT[x] $\leftarrow$ WEIGHT[x] + DEPTH[v] + 1;

WEIGHT[y] $\leftarrow$ WEIGHT[y] $-$ WEIGHT[x].

In steps 3 and 4 we merge the smaller tree into the larger, adjusting the weights and counts appropriately.

LEMMA 1. *Suppose Algorithm* 2 *is used every time trees must be merged by a* LINK *instruction and Algorithm* 1 *is used every time we wish to compute* DEPTH[u]. *Then*

(a) *each value* DEPTH[u] *found in Algorithm* 1 *is correct, and*

(b) *if* $O(n)$ *tree merges and* $O(n)$ *depth computations are done, the total time spent in Algorithms* 1 *and* 2 *is* $O(n\alpha(n))$.[2]

Proof. Observe that Algorithm 1 does not change the sum of the weights along the path from any node to its root. In particular, step 4 of Algorithm 1 adds the sum of the weights of $u_1, u_2, \cdots, u_{i-1}$ to u_i for each node u_i moved in step 3. This adjustment corrects for the fact that the path from u_i to u_0 no longer passes through $u_1, u_2, \cdots, u_{i-1}$.

Algorithm 2 adds the value DEPTH(v) + 1 to paths from nodes in the tree containing u and does not change other paths. Since u is the root (in the A-forest) of its tree, we may conclude (a). For part (b) observe that the number of steps taken by Algorithm 1 is proportional to that of the set merging algorithm of [2]; [14] shows this algorithm takes $O(n\alpha(n))$ time. □

It is important that the weights in the D-forest do not grow too large since we are assuming that arithmetic on integers can be accomplished in one step. Should numbers grow larger than say $O(n)$, we would have to consider the cost of multiple-precision arithmetic. The following bound, however, justifies our ignoring the cost of arithmetic.

LEMMA 2. *No weight in the D-forest exceeds n in magnitude.*

Proof. Suppose that u is a node in the D-forest and $u, v_1, v_2, \cdots, v_k$ is the

[2] $\alpha(n)$ is the inverse Ackermann function defined in [14].

path from u to its root. Then the difference in the depth of nodes u and v_1 is easily seen to be WEIGHT[u]. Since no depth exceeds n, the difference of two depths cannot exceed n. Also, the depth of a node represented by a root in the D-forest is never greater than n. Thus $\lceil \text{WEIGHT}[u] \rceil \leqq n$ for all u. □

4. Computing ancestor information. Maintaining the structure of the A-forest is easy, since a LINK(u, v) instruction can be executed by attaching an additional pointer to node u, i.e., setting ANCESTOR[u, 0] to v. What is difficult is the maintenance of the ancestor information. We shall define a recursive routine INSTALL(u, i) which inserts the (2^i)th ancestor of u into the ANCESTOR array. This routine will be called at various times when ancestor information is needed to execute an LCA instruction. It is written under the assumption that ANCESTOR[u, 0] $\neq$ *undefined* for any u to which the routine INSTALL will be applied.

```
procedure INSTALL(u, i):
begin
    if ANCESTOR[u, i − 1] = undefined then INSTALL(u, i − 1);
    if ANCESTOR[ANCESTOR[u, i − 1], i − 1] = undefined then
        INSTALL(ANCESTOR[u, i − 1], i − 1);
    ANCESTOR[u, i] ← ANCESTOR[ANCESTOR[u, i − 1], i − 1]
end
```

Given the assumption that ANCESTOR[v, 0] is correctly defined for all v and that u has a (2^i)th ancestor, a straightforward induction on $i \geqq 1$ shows that INSTALL(u, i) correctly computes ANCESTOR[u, i].

We shall also define a procedure FIND(u, v, i, d) which takes as arguments two distinct nodes u and v of equal depth d such that $2^{i-1} \leqq d$ and such that the (2^i)th ancestors of u and v are the same or neither exists. The result of FIND(u, v, i, d) is the lowest common ancestor of u and v; FIND works by repeatedly halving the range in which the length of the path from u to the lowest common ancestor of u and v is known to lie.

```
procedure FIND(u, v, i, d):
if i = 0 then return ANCESTOR[u, 0];
else
    begin
        if ANCESTOR[u, i − 1] = undefined then INSTALL(u, i − 1);
        if ANCESTOR[v, i − 1] = undefined then INSTALL(v, i − 1);
        if ANCESTOR[u, i − 1] = ANCESTOR[v, i − 1] then return
            FIND(u, v, i − 1, d)
        else
            begin
                j ← min (i − 1, ⌊log (d − 2^(i−1))⌋);
                return FIND(ANCESTOR[u, i − 1],
                    ANCESTOR[v, i − 1], j, d − 2^(i−1))
            end
    end
```

It is straightforward to show that FIND works correctly given that $2^{i-1} \leqq d$. The selection of j on the next to last line of the procedure insures that $2^{j-1} \leqq d - 2^{i-1}$.

We can now give an algorithm to compute the lowest common ancestor of an arbitrary pair of nodes.

ALGORITHM 3 (Lowest common ancestor of u and v).

1. Use Algorithm 1 to compute DEPTH(u) and DEPTH(v). Assume without loss of generality that DEPTH(u) $\geqq$ DEPTH(v).

2. Find the ancestor a of u having the same depth as v by the following procedure:

```
begin
    a ← u;
    d ← DEPTH(u) − DEPTH(v);
    while d ≠ 0 do
        begin
            j ← ⌊log d⌋;
            if ANCESTOR[a, j] = undefined then INSTALL (a, j);
            a ← ANCESTOR[a, j];
            d ← d − 2^j
        end;
    return a
end
```

3. If $a = v$, then a is the lowest common ancestor of u and v. Otherwise, execute FIND(a, v, i, d), where d = DEPTH(v) and $i = \lfloor \log d \rfloor$.

5. An on-line algorithm. We now utilize Algorithms 1, 2 and 3 to execute the sequence σ on-line, that is, providing the answer to the ith instruction in σ before the $(i + 1)$st instruction is read.

ALGORITHM 4 (On-line execution of σ).

1. Initialize the D-forest with all nodes in separate trees, having counts of 1 and weights of 0.

2. Initialize the A-forest with all nodes in separate trees and with ANCESTOR[u, i] = *undefined* for all u and i.

3. Execute an LCA(u, v) instruction by applying Algorithm 3 to u and v. Print the resulting lowest common ancestor.

4. Execute a LINK(u, v) instruction as follows:

(a) Set ANCESTOR[$u, 0$] = v.

(b) Use Algorithm 2 to merge the trees in the D-forest holding u and v.

THEOREM 1. *If Algorithm 4 is applied to execute σ, the execution of the algorithm requires at most $O(n \log n)$ steps of a random access computer.*[3]

Proof. Algorithm 4 results in $O(n)$ calls of Algorithm 2. There are also $O(n)$ calls to Algorithm 3, which result in $O(n)$ calls to Algorithm 1. By Lemma 1, all calls to Algorithms 1 and 2 are handled in $O(n\alpha(n))$ steps.

Exclusive of calls to FIND and INSTALL, Algorithm 3 clearly requires

[3] See [1] for a discussion of the formal "RAM" model.

$O(\log n)$ steps per call, for a total of $O(n \log n)$ steps. Since FIND calls itself with the third argument decreased by at least 1 each time, at most $O(n \log n)$ calls to FIND may be made. Since each call of FIND requires constant time exclusive of calls to itself or to INSTALL, the total cost of FIND exclusive of calls to INSTALL is $O(n \log n)$.

Since INSTALL(u, i) is called only if ANCESTOR$[u, i]$ = *undefined*, and ANCESTOR$[u, i]$ will be defined after this call to INSTALL(u, i), we see that no more than $O(n \log n)$ calls of INSTALL can occur. Since each call of INSTALL requires constant time exclusive of calls to itself, the cost of INSTALL is $O(n \log n)$.

Thus Algorithm 4 requires at most $O(n \log n)$ steps on any sequence of $O(n)$ LINK and LCA instructions. □

One might argue that the "obvious" method of executing σ has an expected time of $O(n \log n)$, since a random sequence of LINK instructions might produce paths of length $O(\log n)$, rather than $O(n)$. In this case, however, it is easy to bound the expected (not worst-case) time taken by Algorithm 4 at $O(n \log \log n)$. In fact, if the expected path length in trees is $f(n)$, then our algorithm will run in $O(\max \{n \log f(n), n\alpha(n)\})$ steps. In § 7 we shall see that in the case where all LINK instructions precede all LCA instructions, $O(n \log \log n)$ is an upper bound on the running time of a modified algorithm, as well as its expected time.

6. An off-line algorithm. Algorithm 4 produces an answer to the ith instruction in σ before the $(i + 1)$st is read. If we are willing to wait until all of σ has been seen before producing any answers, however, we can do better than $O(n \log n)$; an $O(n\alpha(n))$ algorithm exists.

To begin, we use the $O(n\alpha(n))$ set merging algorithm of [2] to check that no LCA(u, v) instruction in σ has u and v on different trees. Having thus assured ourselves that we have a legal sequence of instructions, we may build the forest required by the LINK instructions in σ. If there is more than one tree in the final forest at the end, we can make all roots be sons of a new node, so that exactly one tree T results. For each LCA(u, v) instruction in σ, the lowest common ancestor of u and v in T will be their lowest common ancestor in the forest built by the LINK instructions preceding that LCA instruction in σ.

We shall number the nodes of T so that if we visit them in preorder, we visit them in the order $1, 2, \cdots$. For example, the nodes in Fig. 1(b) would be numbered as shown in Fig. 2.

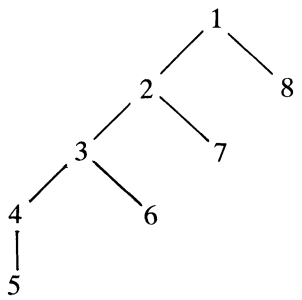


FIG. 2. *Preorder numbering*

The construction of T and the preorder numbering of the nodes can clearly be done in $O(n)$ steps.

Note that if a and b are preordered nodes, then $a < b$ if and only if

(i) a is an ancestor of b, or

(ii) a is to the left of b.

We shall now identify each LCA(u, v) instruction in σ with a distinct *object* X with which we shall associate the integer pair (i, j), such that $i < j$, and i and j are the numbers associated with nodes u and v. Let L and R be the projection functions such that $L(X) = i$ and $R(X) = j$.

We wish to generate the answers to the LCA instructions in σ. To do this, we shall first derive from the tree T a sequence τ of new instructions ENTER(X) and REMOVE(i), where X is an object and i an integer. We can think of these instructions as entering and removing objects from a "bin" which is initially empty.

1. The instruction ENTER(X) places object X in the bin.

2. The instruction REMOVE(i) removes from the bin all objects X such that $L(X) \geqq i$. In addition, for each object X removed, we set $A[X] = i$, where A is an array indexed by the objects. We shall see that i is the lowest common ancestor of the pair of nodes associated with the object X.

We shall subsequently show that the execution of the sequence τ can be simulated in $O(n\alpha(n))$ steps, off-line, by the $O(n\alpha(n))$ set merging algorithm of [2]. To begin, we show how the sequence τ is generated from the set of objects and the tree T.

ALGORITHM 5 (Generating τ).

1. For each node i, list those objects X for which $R(X) = i$.

2. Process each node of T in postorder. That is, node i is processed before node j if and only if i is to the left of j or a descendant of j. The nodes in postorder for the tree of Fig. 2 are:

$$5 \quad 4 \quad 6 \quad 3 \quad 7 \quad 2 \quad 8 \quad 1.$$

When at node i, do the following:

(a) Generate the instruction ENTER(X) for each X such that $R(X) = i$.

(b) Generate the instruction REMOVE(i).

We shall now prove an important property of the sequence τ of ENTER and REMOVE instructions generated by Algorithm 5.

LEMMA 3. *Object X is removed from the bin by the instruction* REMOVE(a) *in τ if and only if a is the lowest common ancestor of $L(X)$ and $R(X)$.*

Proof. If. Let X be an object such that a is the lowest common ancestor of $L(X)$ and $R(X)$. Object X will be placed in the bin by the ENTER(X) instruction generated when node $R(X)$ is processed. Because the nodes are processed in postorder, all nodes processed between $R(X)$ and a must either be ancestors of $R(X)$ and descendants of a or they must be descendants of a to the right of $R(X)$.

$L(X)$ is not the descendant of any node between $R(X)$ and a in the postorder. Thus $L(X) < u$ for all nodes u processed between $R(X)$ and a. Since $L(X) = a$ or $L(X)$ is a descendant of a, we must have $L(X) \geqq a$. Thus object X is removed from the bin by the REMOVE(a) instruction in τ.

Only if. Suppose object X is removed from the bin by the instruction REMOVE(a). Then $R(X)$ must precede a in the postorder, $L(X) \geqq a$, and for any node u between $R(X)$ and a in the postorder, $L(X) < u$.

Suppose $R(X)$ is not a descendant of a. Then $R(X)$ is to the left of a (since $R(X)$ precedes a in the postorder) and $R(X) < a$. But then $L(X)$ also must be less than a, a contradiction. Therefore $R(X)$ must be a descendant of a.

Let u be a descendant of a and an ancestor of $R(X)$. That is, u is a node between $R(X)$ and a in the postorder. $L(X)$ cannot be a descendant of u since $L(X) < u$. However, either $L(X) = a$ or $L(X)$ is to the left of $R(X)$. In addition, we know $L(X) \geqq a$. Thus $L(X)$ must be a descendant of a (or a itself). Hence, a is the lowest common ancestor of $L(X)$ and $R(X)$. □

We shall now give an algorithm that will simulate the execution of the sequence τ.

ALGORITHM 6 (Simulation of σ). We note that all instructions in σ are distinct, and that the last instruction in σ is REMOVE(1).

1. Suppose that there are k nodes in the tree T, so for all objects X, we have $1 \leqq L(X) < k$. (Note that $k \leqq n + 1$, where n is the number of nodes in the original forest.) For each i, $1 \leqq i \leqq k$, make a list $OBJ[i]$ of those objects X for which $L(X) = i$.

2. Create an "atom" e_X for each ENTER(X) instruction in τ and an atom r_i for each REMOVE(i) instruction in τ. Also create an initially empty set named S_i for each REMOVE(i) instruction in σ. Place e_X in S_i if $R(X) = i$. Place r_i in S_j if REMOVE(j) is the first REMOVE instruction in τ to follow REMOVE(i).

3. For $i = k, k - 1, \cdots, 1$ in turn do the following:

(a) For each X on $OBJ[i]$, find the set S_j of which e_X is currently a member. Then do (b) and (c).

(b) If $i \geqq j$, place X on list $REM[j]$, which will hold all objects that are removed from the bin when the instruction REMOVE(j) in τ is executed. Consider the next X in step 3(a).

(c) If $i < j$, merge set S_j with that set S_h such that r_j is in S_h. Call the new set S_h. Return to step (b) with j set to h.

4. Examine each REMOVE(i) instruction of τ in turn from the beginning. List those pairs (X, i) such that X is on $REM[i]$.

In step 1 of Algorithm 6 we create the list OBJ to sort the objects in terms of their first components. In step 2 we enter the atoms representing the objects into sets indexed by the second component of the object. We also include in set S_j the atom r_i corresponding to the instruction REMOVE(i) if node j follows node i in the postorder. In step 3, for each object X in set S_j, we locate via the r-atoms the first ancestor a of node j such that $L(X) \geqq a$. Node a is the lowest common ancestor of nodes $L(X)$ and $R(X)$.

The motivation behind Algorithm 6 is that each REMOVE(i) instruction in τ is presumed to remove from the bin all objects X such that the instruction ENTER(X) precedes REMOVE(i) in τ. If we are working on some X for which $L(X) > i$, however, then we will have already found all those objects which will be removed by the instruction REMOVE(i). We therefore "get rid of" the instruction REMOVE(i) by merging the set S_i with the set for the next remaining REMOVE instruction.[4] The atom r_i allows us to find the next REMOVE instruction, since r_i will always be in the set associated with that instruction.

[4] Note that the last instruction, REMOVE(1), can never disappear, so step 3(c) can always be carried out.

A formal proof that Algorithm 6 works correctly is quite similar to the proof regarding the "INSERT-EXTRACT" instructions in [2], and we omit it.

We now summarize the off-line LINK-LCA algorithm.

ALGORITHM 7 (Off-line simulation of the sequence σ of LINK and LCA instructions).

1. Test that when an LCA(u, v) instruction is encountered in σ, u and v are on the same tree, using the $O(n\alpha(n))$ set merging algorithm of [2].

2. Build the forest as dictated by the LINK instructions in σ. If necessary, add one root to make the final forest a tree T.

3. Number the nodes of T in preorder.

4. For each LCA(u, v) instruction, create an object $X = (i, j)$, where i and j are the preorder numbers of u and v.

5. Use Algorithm 5 to generate the sequence τ of ENTER and REMOVE instructions for T and the set of objects created in step 4.

6. Use Algorithm 6 to simulate the sequence τ.

7. Scan the output of Algorithm 6. For each (X, i) in the output, set $A[X] = i$.

8. Scan the LCA instructions in the original sequence σ. For each LCA(u, v) instruction, determine the corresponding object X; $A[X]$ is the lowest common ancestor of u and v.

THEOREM 2. *Algorithm 7 requires $O(n\alpha(n))$ steps on a random access computer.*

Proof. Step 1 can be done in $O(n\alpha(n))$ steps. Steps 2–4 are each easily seen to be $O(n)$, and the sequence of ENTER and REMOVE instructions generated is $O(n)$ in length.

Since k in Algorithm 6 is at most $n + 1$, steps 1 and 2 of Algorithm 6 can be done in $O(n)$ time. In step 2, $O(n)$ atoms (of the forms e_X and r_i) are created. As step 3(c) of Algorithm 6 can apply only $O(n)$ times, step 3 involves at most $O(n)$ operations of merging two sets or finding the set containing a given node. Thus step 3 can be performed in $O(n\alpha(n))$ steps if we use the $O(n\alpha(n))$ algorithm of [2] for the set merging and name finding operations.

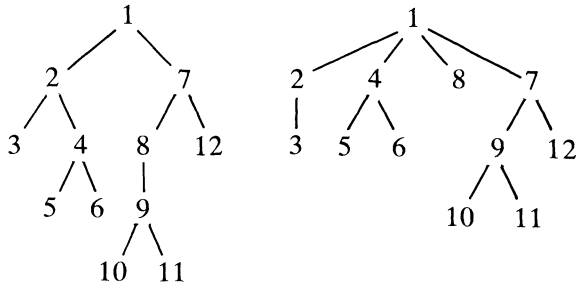
Finally, steps 7 and 8 are clearly $O(n)$. Thus the aggregate time required by Algorithm 7 is $O(n\alpha(n))$. □

7. An intermediate problem. Let us return to the on-line processing of σ, our original sequence of LINK and LCA instructions, but now assuming that all LINK instructions in σ precede all LCA instructions. In this case, we may build the implied forest first, and then process the LCA instructions without changing the forest. Before executing the LCA instructions, however, we shall modify the implied forest so that all paths in the forest are bounded by $O(\log n)$ in length. Then we can process each LCA instruction in at most $\log \log n$ steps.

Given a tree T in the forest, we shall construct from it a *virtual tree* V which has the same nodes as T but in which nodes have different fathers. The father of a node u in V is the lowest ancestor of u in T having at least twice as many descendants in T as does u. As a special case, if no such ancestor exists and u is not the root of T, we then make the root of T the father of u in V.

Example 2. A tree T is shown in Fig. 3(a). Its virtual tree is shown in Fig. 3(b). For example, node 9 has three descendants (we are assuming a node is a descendant

of itself). Node 8 has four, but node 7 has six, so node 7 becomes the father of node 9 in the virtual tree.



(a) *Tree* (b) *Virtual tree*

FIG. 3. *Tree and its virtual tree*

LEMMA 4. *Let T be a tree with n nodes and V its virtual tree. No path in V is longer than* $\log n$.

Proof. The ith node in a path beginning from a leaf has at least 2^{i-1} descendants in T, provided the ith node is not the root. If the path is longer than $\log n$, T would have more than n nodes, a contradiction. □

LEMMA 5. *Let T be a tree with n nodes and V its virtual tree. Let u and v be two nodes and let node a be their lowest common ancestor in T. Assume u, v and a are all distinct, and suppose v has at least as many descendants as u. Then f, the father of u in V, is a descendant of a in T (possibly a itself). Thus, in T, a is also the lowest common ancestor of f and v.*

Proof. Suppose not. Then node a would have fewer than twice as many descendants as u, a contradiction, since a has more descendants than u and v put together. □

An efficient off-line method for determining whether one of two nodes in a tree T is a descendant of the other is to preorder the nodes of T and to attach to each node i (that is, i is its number in preorder) the value HIGH[i] which is the highest numbered node that is a descendant of node i. Node i is a descendant of j if and only if $j \leqq i$ and HIGH[j] $\geqq$ HIGH[i]. It should be clear that HIGH can be computed for a tree with n nodes in $O(n)$ steps.

We also observe without proof that in $O(n)$ steps we can compute COUNT[u], the number of descendants of node u, for all nodes u. Furthermore, in $O(n)$ steps we can find for each node u, a son of u having the largest count.

We shall now outline an $O(n)$ algorithm to construct a virtual tree from a tree T with n nodes. The heart of the algorithm is a procedure BUILD(u) which finds fathers in the virtual tree for all nodes in the subtree T_u of T with root u. The result of BUILD is a queue Q of those nodes v in T_u such that $2 * \text{COUNT}[v] > \text{COUNT}[u]$.

A *queue* is a list of elements from which elements are removed from the front

and added to the rear; *front*(Q) is the first element of a queue Q.

```
procedure BUILD(u):
begin
    construct a list of nodes u_1, u_2, ..., u_k such that u_1 = u, u_k is a leaf, and
        u_{i+1} is a son of u_i with the largest count, for 1 ≦ i < k;
    Q ← u_k;
    for i = k − 1 step − 1 until 1 do
        begin
            while COUNT[u_i] ≧ 2 * COUNT[front (Q)] do
                begin
                    make u_i the father of front (Q) in the virtual tree;
                    delete front (Q) from Q
                end;
            add u_i to the rear of Q
        end
    for i = 1 until k − 1 do
        for each son v of u_i other than u_{i+1} do
            begin
                R ← BUILD(v);
                for each w on R do
                    make u_i the father of w in the virtual tree
            end;
    return Q
end
```

ALGORITHM 8 (Constructing the virtual tree).

1. Execute BUILD(u_0), where u_0 is the root of T.

2. For each node $v \neq u_0$ on the resulting queue, make u_0 the father of v in the virtual tree.

Example 3. After applying BUILD to node 7 of Fig. 3(a), Q contains nodes 8 and 7.

LEMMA 6. *Algorithm* 8 *requires* $O(n)$ *steps and correctly builds the virtual tree.*

Proof. For the linearity of the algorithm, it suffices to observe that BUILD takes time proportional to the number of nodes on the path $u_1, \cdots, u_k$ found in the first statement of BUILD, exclusive of recursive calls to itself. However, no node in T will be on the path created by two distinct calls of BUILD, except the first node of the path.

For the correctness of the algorithm, it is easy to determine that each node on the path $u_1, \cdots, u_k$ is given its correct father if that father is on T_u. Moreover, if v is a son of u_i and $v \neq u_{i+1}$, then COUNT[u_i] is no less than twice COUNT[v], so v's father in T_u is u_i. The same is clearly true of any descendant w of v remaining on the queue R when BUILD(v) is called. That is,

$$\text{COUNT}[u_i] \geqq 2 * \text{COUNT}[w] > \text{COUNT}[v]$$

We shall use the following strategy to simulate σ. After all LINK instructions in σ have been seen, we shall build the implied forest F and then from F a virtual forest V. Then, when we see an instruction LCA(u, v), we choose the one of u

and v having the smaller count, say u. We find the $\lfloor(\log n)/2\rfloor$th ancestor of u in V, say a. If $a = v$, LCA(u, v) is clearly v. If a is a proper ancestor of v in F, we repeat this procedure, finding the $\lfloor(\log n)/4\rfloor$th ancestor of u in V. If a is not an ancestor of v in V, we repeat the procedure, assuming the instruction was LCA(a, v) and beginning with the $\lfloor(\log n)/4\rfloor$th ancestor in V of the one of a and v having the smaller COUNT.

In $\log \log n$ steps we shall converge upon a node a in V which is an ancestor of one of u and v in V. Since we are effectively following paths in F from u and v toward a root, and since we always move from the current ancestor of u or v having the smaller count, Lemma 5 guarantees us that a is the lowest common ancestor of u and v in F.

To implement this strategy, we shall use a procedure LOCATE(u, v, i, j) which finds the lowest common ancestor of u and v on F, given that

(a) the (2^i)th ancestor of u on V either does not exist or is an ancestor of v in F, and

(b) the (2^j)th ancestor of v in V either does not exist or is an ancestor of u in F.

In what follows, we assume that COUNT and HIGH refer to the implied forest F and ANCESTOR$[p, i]$ to the virtual forest V. We assume that this information has already been computed.

```
(1)  procedure LOCATE(u, v, i, j):
(2)  without loss of generality assume COUNT[u] ≦ COUNT[v]
(3)  otherwise (u, v, i, j) ← (v, u, j, i) in[5]
(4)      if i = 0 then return ANCESTOR[u, 0]
         else
             begin
(5)              a ← ANCESTOR[u, i − 1];
(6)              if v = a then return a
(7)              else if a = undefined or (v > a and HIGH[v] ≦ HIGH[a])
(8)                  then return LOCATE(u, v, i − 1, j)
(9)                  else return LOCATE(a, v, i − 1, j)
             end
```

We now summarize the entire algorithm.

ALGORITHM 9 (On-line execution of σ, assuming all LINK instructions in σ precede all LCA instructions).

1. As the LINK instructions are read, build the implied forest F in the obvious way.

2. When the first LCA instruction is encountered, do the following steps.

(a) For each tree in F, build a virtual tree by Algorithm 8. Call the resulting virtual forest V.

(b) Use the procedure INSTALL of § 4 to compute ANCESTOR$[u, i]$ for all nodes u and $0 \leqq i \leqq \lfloor\log(1 + \log n)\rfloor$.

(c) Preorder the nodes of F and compute COUNT$[u]$ and HIGH$[u]$ for all nodes u.

[5] This statement is a compile-time macro. See [1].

3. Now process each LCA instruction in turn. To compute LCA(u, v), we check whether one of u and v is a descendent of the other using the HIGH information. If so, the response is obvious. If not, we execute LOCATE(u, v, k, k), where $k = \lfloor \log(1 + \log n) \rfloor$ and print the result.

THEOREM 3. *Algorithm* 9 *correctly simulates* σ, *assuming that if the instruction* LCA(u, v) *is encountered*, u *and* v *are on the same tree.* (This condition can be checked in $O(n\alpha(n))$ time, as in Algorithm 7.)

Proof. The crux of the proof is showing that LOCATE works correctly. To do this, we shall show by induction on the sum $i + j$ that LOCATE(u, v, i, j) produces the lowest common ancestor of u and v, given that:

(a) the (2^i)th (resp. (2^j)th) ancestor of u (resp. v) in V is undefined or an ancestor of v (resp. u) in F, and

(b) COUNT[u] $\leqq$ COUNT[v],

(c) $i \geqq 0$ and $j \geqq 0$.

Basis. $i = j = 0$. Let f be the father of u in V, and let a be the lowest common ancestor of u and v. By hypothesis, f is an ancestor of v, and hence of a. By Lemma 5, f is a descendant of a, so $f = a$. Since line (4) makes the result of LOCATE be f in the case $i = 0$, we have the basis.

Inductive step. If $i = 0$, the argument is the same as for the basis. If $i \neq 0$, let a be the (2^{i-1})st ancestor of u in V, and let b be the lowest common ancestor of u and v. If $a = v$, then $b = v$ and this relationship is reflected in line (6) of LOCATE.

If a is a proper ancestor of v in F, or is undefined, then by the inductive hypothesis, LOCATE(u, v, $i - 1$, j) invoked on line (8) correctly produces b. If a is not an ancestor of v in F, then the lowest common ancestor of the pairs (u, v) and (a, v) are the same. Moreover, in F the (2^i)th ancestor of u is the (2^{i-1})st ancestor of a. Since a must be a descendant of b, and the (2^j)th ancestor of v, if it is defined, is an ancestor of b, it follows that the (2^j)th ancestor of v is an ancestor of a if it is defined. Thus the inductive hypothesis tells us that the result of LOCATE (a, v, $i - 1$, j) invoked on line (9) produces the correct result.

THEOREM 4. *Algorithm* 9 *operates in* $O(n \log \log n)$ *time.*

Proof. Steps 1, 2(a) and 2(c) are $O(n)$. Step 2(b) is $O(n \log \log n)$. Step 3 requires $O(n \log \log n)$ time since a call to LOCATE(u, v, k, k) can result in at most $2k + 1$ additional calls to LOCATE.

8. Dominators and reducible graphs. We shall now apply Algorithm 4 to a problem in code optimization. This section presents the basic definitions.

A *flow graph* is a triple $G = (N, E, u_0)$ where N is a finite set of nodes, E is a subset of $N \times N$ (the set of directed edges), and u_0 in N is the *initial node*. There is a path from u_0 to every node.

If each node has no more than two successors, we call G a *program* flow graph.

We say that node d *dominates* another node u if every path from u_0 *to* u passes through d. That is, if $u_0, u_1, \cdots, u_i$ is a path with $u_i = u$, then there exists an integer j, $0 \leqq j < i$ such that $u_j = d$. We say d *immediately dominates* u if d dominates u and every other dominator of u also dominates d. There are several interesting properties of the dominator and immediate dominator relations. The

following lemma is taken from [3].

LEMMA 7. (a) *Every node except the initial node has an immediate dominator.*

(b) *We may construct a tree (called the* dominator tree) *in which u is a son of d if and only if d immediately dominates u. The ancestors of u in the tree are precisely the dominators of u.*

Information about the dominator relation is useful for certain compiler code optimizations, such as those involving the detection of "loops". See [3], [4] for elaboration. By Lemma 7(b), the dominator information can be stored in a tree constructed knowing only the immediate dominators. If, as in [3], only a small number of dominators—the lowest ancestors in the tree—are used, we may not even need to construct the complete dominator relation.

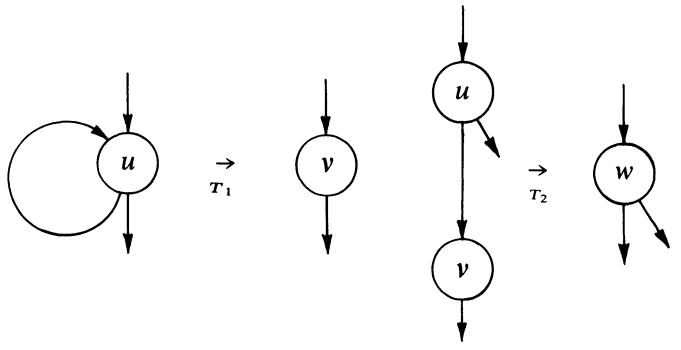
Algorithms to compute the dominator relation are given in [3] and [5]. Each requires $O(n^3)$ steps for program flow graphs, where n is the number of nodes in the graph. $O(n^2)$ algorithms for program flow graphs are found in [4] and [6], and these appear to be optimal, as it can take $O(n^2)$ time just to print the answer. To our knowledge, no one has developed a faster algorithm to compute only the immediate dominators. Here we do so for the important special case of reducible program flow graphs.

Reducible graphs were defined in [7]. They form a large class of graphs. For example, every rooted directed acyclic graph is reducible, and the flow graphs of gotoless programs are reducible. In fact, experiments have shown that the flow graphs of most programs written in practice are reducible. Moreover, any flow graph can be made reducible by judicious node splitting [8]. While this process could be expensive, those flow graphs which come "from nature" but which are not reducible readily yield to the node splitting technique. As a result, many code optimization algorithms such as

(i) eliminating common subexpressions [9], [10],
(ii) propagating constants [4],
(iii) eliminating useless definitions [4], and
(iv) finding active variables [11]

have been couched in terms of reducible flow graphs.

We shall give a definition of reducible flow graphs taken from [12]. This involves two transformations on directed graphs illustrated in Fig. 4 and defined



(a) *Transformation* T_1 (b) *Transformation* T_2

FIG. 4.

as follows:

T_1: Delete a loop.

T_2: Let node u be the lone predecessor of node v, where v is not the initial node. Merge u and v into a single node w. The predecessors of u become predecessors of w. The successors of u and v become successors of w. Note that w has a loop if there was formerly an edge to u from u or v. If u was the initial node, w becomes the new initial node. In this transformation we say v is *consumed*.

It is known that if T_1 and T_2 are applied to a given flow graph until no longer possible, a unique flow graph results. If this flow graph is a single node, we call the original graph *reducible*.

Example 4. Fig. 5 shows a sequence of reductions by T_1 and T_2. The initial node is u_0.

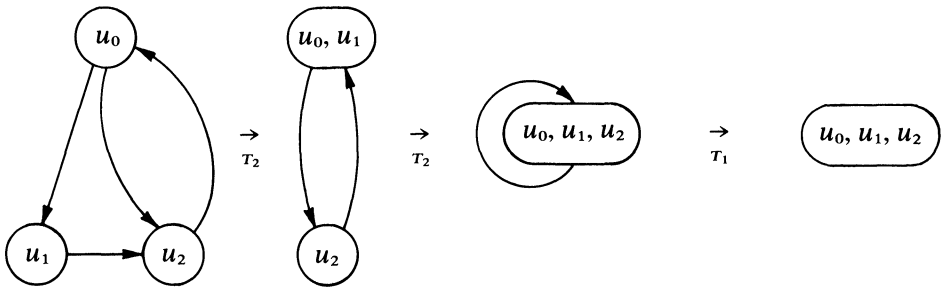


FIG. 5. *Reduction of flow graph.*

In the first step, T_2 is applied to u_1 with lone predecessor u_0. At the second step, T_2 is applied to u_2 with lone predecessor $\{u_0, u_1\}$. There is a loop introduced since $\{u_0, u_1\}$ is a successor of u_2.

A *region* R is a subset of the nodes of a flow graph such that there is a node h in R, called the *header*, having the property that every node in $R - \{h\}$ has all of its predecessors in R. Thus, the header dominates every other node in the region.

Example 5. Any node by itself is a region. In the previous example, $\{u_0, u_1\}$ is a region with header u_0, but $\{u_1, u_2\}$ is not since both u_1 and u_2 have a predecessor, u_0, outside the set.

Following [10], we may observe that as we reduce a flow graph by applying T_1 and T_2, each node of each successive graph *represents* a region in the following sense.

1. Initially, each node represents itself.
2. If we apply T_1 to a node, it continues to represent the same region as before.
3. If we apply T_2 when node u is the lone predecessor of node v, the resulting node represents the union of the regions represented by u and v. The header of the region represented by u is the header of the new region.

The following lemma is a restatement of the definitions of "dominator tree" and "region."

LEMMA 8. (a) *If u is a node in region R and u is not the header of R, then the immediate dominator of u is a member of R.*

(b) *If T_2 is applied to nodes u and v, with u the lone predecessor of v, and u and v represent regions R_u and R_v, respectively, then the immediate dominator of the header h of R_u is the lowest common ancestor (on the dominator tree of R_v) of the predecessors of h in the original graph.*

9. Dominators of a reducible flow graph. As a consequence of Lemma 8, the following algorithm may be used to construct the dominator tree for a reducible flow graph. The algorithm generates a sequence of LINK and LCA instructions for each reduction by T_2. These instructions will place u_0, the header of the region consumed by T_2, in its proper place on the dominator tree. Thus, if the flow graph is reducible, every node except the initial node will be the header of a region consumed by T_2, and its immediate dominator will be known. The details of the algorithm are as follows.

ALGORITHM 10 (Construction of dominator tree).

1. List the predecessors of each node of the original graph.

2. Use the algorithm of [13] to reduce the graph. Keep track of each region represented by the nodes of the "current" graph. Each time a node is consumed by T_2, create the sequence of instructions

$$\begin{aligned} &\mathrm{LCA}(u_1, u_2)\\ &\mathrm{LCA}(v_1, u_3)\\ &\quad\vdots\\ &\mathrm{LCA}(v_{k-2}, u_k)\\ &\mathrm{LINK}(u_0, v_{k-1}) \end{aligned}$$

and simulate them by Algorithm 4. Here u_0 is the header of the consumed region, $u_1, \cdots, u_k$ are all its predecessors, v_1 is the lowest common ancestor of u_1 and u_2, and v_i is the lowest common ancestor of v_{i-1} and u_{i+1} for $2 \leqq i < k$. If $k = 1$, the sequence is just $\mathrm{LINK}(u_0, u_1)$. (Note that neither Algorithm 7 nor 9 is sufficient. Direct on-line simulation is required.)

3. The desired dominator tree is the final A-forest (which must be a tree, since only one root, the initial node, remains).

THEOREM 5. *Algorithm* 10 *correctly constructs the dominator tree and requires $O(e \log e)$ steps on a flow graph with e edges.*

Proof. The correctness of the algorithm follows immediately from Lemma 8. By Lemma 8(a), the nodes of each region may be formed into a dominator tree with the header as root. By Lemma 8(b), when T_2 is applied, the dominator tree for the new region is created by making the header h of the consumed region a son of the lowest common ancestor of the predecessors of h in the original graph.

Step 1 requires $O(e)$ time. The reduction of the graph may be accomplished in $O(e\alpha(e))$ time by [13], and the sequence of instructions generated clearly has length $O(e)$. Thus step 2 requires $O(e \log e)$ time by Theorem 1. □

COROLLARY. *Algorithm* 10 *requires* $O(n \log n)$ *steps on an n-node program flow graph.*

Proof. An n-node program flow graph has no more than $2n$ edges. □

COROLLARY. *Algorithm* 10 *requires* $O(e \log e)$ *steps on a rooted directed acyclic graph with e edges.*

Proof. A rooted directed acyclic graph is reducible. □

After this paper was written, Tarjan [15] developed an $O(\max(n \log n, e))$ dominator algorithm for general graphs.

10. Conclusions. We have defined a problem that involves merging nodes into trees while retaining the ability to determine the lowest common ancestor of any two nodes. We have offered an $O(n \log n)$ algorithm to solve the problem on-line. We have shown how this algorithm provides a fast way of computing the dominator tree of a reducible flow graph. If an off-line solution is sufficient, the LINK-LCA problem can be solved in $O(n\alpha(n))$ steps. An on-line solution in the case where all LINK instructions precede all LCA instructions can be achieved in $O(n \log \log n)$ steps.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass, 1974.

[2] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, this Journal, 2 (1973), pp. 294–303.

[3] E. S. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[4] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling. Vol.* 2: *Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[5] M. SCHAEFER, *A Mathematical Theory of Global Flow Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[6] P. W. PURDOM AND E. F. MOORE, *Algorithm* 430: *Immediate predominators in a directed graph*, Comm. ACM, 15 (1972), pp. 777–778.

[7] F. E. ALLEN, *Control flow analysis*, SIGPLAN Not., 5 (1970), pp. 1–19.

[8] J. COCKE AND R. E. MILLER, *Some techniques for optimizing computer programs*, Proc. 2nd Internat. Conf. on System Sciences, Honolulu, Hawaii, 1969.

[9] J. COCKE, *Global common subexpression elimination*, SIGPLAN Not., 5 (1970), pp. 20–24.

[10] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, Acta Informatica, 2 (1973), pp. 191–213.

[11] K. KENNEDY, *A global flow analysis algorithm*, Internat. J. Comput. Math., 3 (1971), pp. 5–16.

[12] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

[13] R. E. TARJAN, *Testing flow graph reducibility*, J. Comput. System Sci., 9 (1974), pp. 355–365.

[14] ———, *On the efficiency of a good but not linear disjoint set union algorithm*, J. Assoc. Comput. Mach., 22 (1975). pp. 215–225.

[15] ———, *Finding dominators in directed graphs*, this Journal, 3 (1974), pp. 62–89.