ON FINDING LOWEST COMMON ANCESTORS IN TREES

by

A. V. Aho
Bell Laboratories
Murray Hill, New Jersey 07974

J. E. Hopcroft*
Cornell University
Ithaca, New York 14850

J. D. Ullman**
Princeton University
Princeton, New Jersey 08540

ABSTRACT

Trees in an n node forest are to be 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) 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 seen, then we may perform the task in O(n G(n)) steps, where G(n) is the number of times we must apply $\log_2$ to n to obtain a number less than or equal to zero.

A third algorithm solves a problem of intermediate complexity. We require the answers on line, but we suppose 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.

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 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 if it exists, and the word "unrelated" otherwise.

Example 1: Suppose that we initially have a forest consisting of eight isolated nodes $u_1, u_2, \dots, u_8$ and we receive the following sequence of instructions.

* Work supported by ONR grant N00014-67-A-0077-0021

** Work partially supported by NSF grant GJ-1052.

link(u_1,u_2)
link(u_3,u_4)
link(u_5,u_6)
link(u_7,u_8)
link(u_2,u_4)
link(u_6,u_8)
lca (u_5,u_7)
link(u_4,u_6)
link(u_2,u_3)

When the instruction *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.

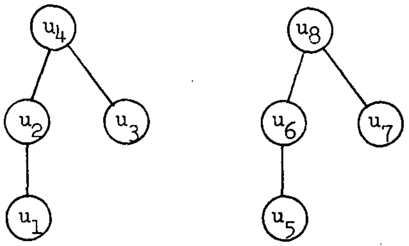


Fig. 1(a) Tree Structures After *lca*(u_5,u_7) Instruction

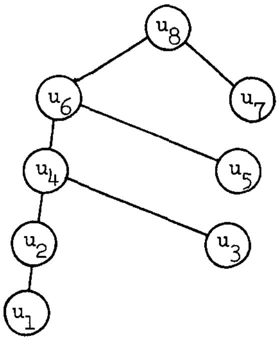


Fig. 1(b) Tree Structures After *lca*(u_2,u_3) Instruction

Fig. 1(b) shows the forest when *lca*(u_2,u_3) is executed. This instruction causes u_4 to be printed.

In this paper we shall consider the problem of executing a sequence α of *link* and *lca* instructions on a forest with n nodes. 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 given an on-line algorithm that requires O(n log n) steps to execute α. 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. AN ON-LINE ALGORITHM

In this section we shall present an O(n log n) algorithm that will execute the sequence α, providing an answer to each *lca* instruction before the next instruction is read. We begin by describing the data structure used to *store* the forest. We then show the operations used to simulate the effect of each *link* and *lca* instruction on the forest.

2.1 A Useful Data Structure for Forests

Let us call the forest which the *link* instructions manipulate the *implied forest*. In our on-line algorithm we shall represent the information in the implied forest in *the computer in* two forests, called the A-forest and the D-forest.

The first defined forest, 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 $0 \leq i < \log n$. All algorithms in this paper *are to the* base 2. 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. However, *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 Section 2.3.

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 depth information 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.

2.2 Maintaining the D-Forest

The following procedure uses the D-forest to compute the depth of nodes in the implied forest. We have attached 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. We then make each node along this path, except the root, be a son of the root, updating the weights of the nodes appropriately.

```
procedure depth(u):
    (1) Find the path from node u to its root in the D-forest. Suppose $u_k, u_{k-1},\dots,u_1,u_0$ is that path,
        where $u_0$ is the root and $u_k$ is u.
    (2) depth(u) <-- $\sum_{i=0}^{k}$ weight($u_i$)
    (3) for i = 2 to k do
            {make $u_i$ a son of $u_0$ in the D-forest;
            weight($u_i$) <-- weight($u_i$) + weight($u_{i-1}$)}
```

The root of each tree in the D-forest also has an associated count, giving the number of nodes in the tree. Before any instructions in α are executed, each node in the D-forest is in a tree by itself, having a count of 1 and a weight of 0. When a link(u,v) instruction is to be executed, the following procedure is invoked.

```
procedure merge(u,v):
    (1) In the D-forest find the roots x and y of the trees holding u and v, respectively, by executing
        depth(u) and depth(v).
    (2) If count(x) $\leq$ count(y), then make x a son of y in the D-forest and do the following:
            count(y) <-- count(y) + count(x);
            weight(x) <-- weight(x) + depth(v) + 1 - weight(y)
        Otherwise, if count(x) > count(y), make y a son of x in the D-forest and do the following:
            count(x) <-- count(y) + count(x);
            weight(x) <-- weight(x) + depth(v) + 1;
            weight(y) <-- weight(y) - weight(x)
```

In step (2) we merge the smaller tree into the larger, adjusting the weights and count appropriately.

Define G(n) to be the smallest number of times the log function must be applied to n to yield zero or less. For example, G(9) = 4, since

$$\log \log \log(9) > 0$$

but

$$\log \log \log(9) \leq 0.$$

Note that $G(n) \leq 6$ for $n \leq 2^{65536}$.

Lemma 1: Suppose the procedure merge is used every time a link instruction is to be executed. Then

(a) each value depth(u) is correct, and

(b) if O(n) tree mergers and O(n) depth computations are done, the total time spent is O(n G(n)).

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. However, the following bound suffices to justify our ignoring the cost of arithmetic.

Lemma 2: No weight in the D-forest exceeds n in magnitude.

2.3 Computing Ancestor Information

It is straightforward to maintain the structure of the A-forest, since a link(u,v) instruction can be executed by setting ancestor(u,0) to v. What is difficult is the maintenance of the ancestor information. We shall define a recursive routine getancestor(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. We note that whenever getancestor(u,i) is called, ancestor(u,0) $\neq$ undefined.

```
procedure getancestor(u,i):

  if ancestor(u,i-1) = undefined then;

     ancestor(u,i) <-- getancestor(getancestor(u,i-1),i-1);

  resultis ancestor(u,i)
```

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 $i \geq 1$ shows that getancestor(u,i) correctly returns 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 $2^{i-1} \leq d$ 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 resultis  ancestor(u,0);

  else if getancestor(u,i-1) = getancestor(v,i-1) then

          resultis find(u,v,i-1,d)

       else

          resultis find(getancestor(u,i-1),getancestor(v,i-1),

                        min(i-1,⌊log(d - $2^{i-1}$)⌋), d - $2^{i-1}$)
```

It is straightforward to show that find works correctly given that $2^{i-1} \leq d$. The selection of the third argument on the last line of the procedure insures that $2^{j-1} \leq d - 2^{i-1}$ where $j = \min(i-1, \log \lfloor (d-2^{i-1}) \rfloor)$.

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

procedure lowest(u,v);

(1) Compute depth(u) and depth(v). If u and v are on different trees, resultis "unrelated." Otherwise, assume without loss of generality that depth(u) $\geq$ depth(v).

(2) Find the ancestor a of u having the same depth as v by the following procedure

```
          a <-- u;

          d <-- depth(u) - depth(v);

          while d > 0 do

             {j <-- ⌊log d⌋;

              a <-- ancestor(a,j);

              d <-- d - $2^j$}*

          resultis a
```

* The control of this loop can also be implemented by treating d as a binary number, examining its least significant bit and right shifting. This avoids having to evaluate the logarithm.

```
(3)  if a = v resultis a else resultis find(a,v, $\lfloor \log d \rfloor$,depth(v))
```

2.4 The On-line Algorithm

Let us write α as $\alpha_1\alpha_2\ldots\alpha_m$ where each α_i represents a link or lca instruction and m is O(n). We now provide an algorithm to execute α on-line, that is, providing the answer to α_i before α_{i+1} is read.

Algorithm 1; On-line execution of α.

```
(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)  for i = 1 to m do
       if $\alpha_i$ = lca(u,v) resultis lowest(u,v);
       if $\alpha_i$ = link(u,v) then
        {ancestor(u,0) <--v ;
         merge(u,v)}
```

Theorem 1: If Algorithm 1 is applied to execute α, the execution of the algorithm requires at most O(n log n) steps of a random access computer.

One might argue that the "obvious" method of executing α 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 1 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 G(n)}) steps. In Section 4 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.

3. AN OFF-LINE ALGORITHM

Algorithm 1 produces an answer to the i^{th} instruction in α before the i + 1^{st} is read. However, if we are willing to wait until all of α has been seen before producing any answers, we can do better than O(n log n);an O(n G(n)) algorithm exists.

To begin, we use the O(n G(n)) set merging algorithm of [2] to find all lca(u,v) instructions in α with u and v on different trees. We answer all such lca instructions with the message "unrelated" and remove these instructions from α. We then build the forest created by the link instructions in α. 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 now in α, 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 α.

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

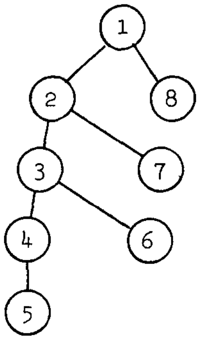


Fig. 2. Preorder Numbering

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 ancestor of b or (ii) a is to the left of b.

We shall now identify each lca(u,v) instruction in α with a distinct object X with which we shall associate the integers (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 α. To do this we shall first derive from the tree T a sequence β 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 $L(X) \ge i$. In addition, for each object removed, we set $LCA(X) = i$, where LCA is a table indexed by the objects. We shall later 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 β can be simulated in O(n G(n)) steps off-line by the O(n G(n)) set merging algorithm of [2]. To begin, we show how the sequence β is generated from the set of objects and the tree T.

Algorithm 2; Generating β.

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

(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 4 6 3 7 2 8 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 state an important property of the sequence β of enter and remove instructions generated by Algorithm 2.

Lemma 3: Object X is removed from the bin by the instruction remove(a) in β if and only if 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 β.

Algorithm 3; Simulation of β. We note that all instructions in β are distinct, and that the last instruction in β is remove(1).

(1) Suppose that there are k nodes in the tree T, so for all objects X, we have $1 \le L(X) < k$. (Note that $k \le n + 1$, where n is the number of nodes in the original forest.) For each i, $1 \le i \le k$, 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 β and an atom r_i for each remove(i) instruction in β. Also create an initially empty set named S_i for each remove(i) instruction in β. 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 β to follow remove(i).

(3) For $i = k, k\text{-}1, \ldots, 1$ 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.

(b) If $i \ge j$, place X on list REM(j), which will hold all objects that are removed from the bin when the instruction remove(j) in β is executed. Return.

(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 β in turn from the beginning. List those pairs (X,i) such that X is on REM(i).

In step (1) of Algorithm 3 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_i, we locate via the r-atoms the first ancestor a of node j such that $L(X) \geq a$. Node a is the lowest common ancestor of nodes L(X) and R(X).

The motivation behind Algorithm 3 is that each *remove*(i) instruction in β is presumed to *remove* from the bin all objects X such that the instruction *enter*(X) precedes *remove*(i) in β. However, if we are working on some X for which $L(X) > i$, then we will have already fround 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.* (Note that the last instruction, *remove*(i), can never disappear, so step 3(c) can always be carried out.) 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.

A formal proof that Algorithm 3 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 4: Off-line simulation of the sequence α of *link* and *lca* instructions.

(1) Using the O(n G(n)) set merging algorithm in [2], remove from α all *lca* instructions whose arguments are on different trees. For these instructions give the answer "unrelated."

(2) Build the forest as dictated by the *link* instructions in α. If necessary, add one root to make the final forest a tree T.

(3) Number the nodes of T in preorder.

(4) For each remaining *lca*(u,v) instruction in α, create an object X = (i,j) where i and j are the preorder numbers of u and v.

(5) Use Algorithm 2 to generate the sequence β of *enter* and *remove* instructions for T and the set of objects created in step (4).

(6) Use Algorithm 3 to simulate the sequence β.

(7) Scan the output of Algorithm 3. For each (X,i) in the output, set LCA(X) = i.

(8) For each *lca*(u,v) instruction currently in α, determine the corresponding object X; LCA(X) is the lowest common ancestor of u and v.

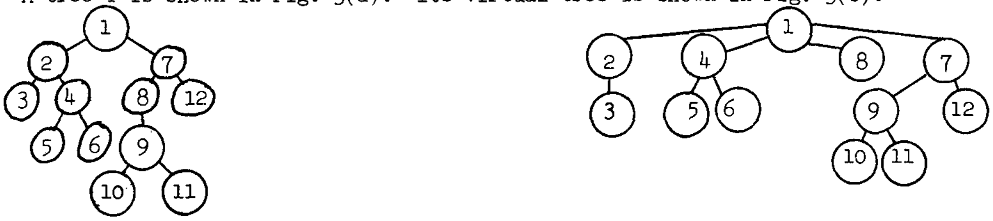
Theorem 2: Algorithm 4 requires O(n G(n)) steps on a random access computer.

4. AN INTERMEDIATE PROBLEM

Let us return to the on-line processing of α, our original sequence of *link* and *lca* instructions, but now assuming that all *link* instructions in α precede all *lca* instructions. In this case we may build the implied forest first, and then process the *lca* instructions without changing the forest. However, before executing the *lca* instructions we shall convert the implied forest into a tree T by making the roots direct descendants of a new root labeled "unrelated." We then modify this tree 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 the tree T, 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 as 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).



(a) Tree

(b) Virtual Tree

Fig. 3 Tree and its Virtual Tree

* Note that the last instruction, *remove*(1), can never disappear so step 3(c) can always be carried out.

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.

Lemma 4: Let T be a tree with n nodes, and V its virtual tree. No path in V is longer than log n.

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.

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 attach to each node i (that is, i is its preorder number) the value high(i) which is the highest numbered node that is a descendant of node i. It can be shown that i is a descendant of j if and only if $j \leq i$ and $high(j) \geq high(i)$. 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 * count(v) > 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):

   construct a list of nodes $u_1, u_2, \ldots, u_k$ 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 $1 \leq i < k$;

   Q <-- $u_k$;

   for i = k-1, k-2, ..., 1 do

      {while count($u_i$) $\geq$ 2 * count(front(Q)) do

         {make $u_i$ the father of front(Q) in the virtual tree;

          delete front(Q) from Q};

          add $u_i$ to the rear of Q};

   resultis Q;

   for i = 1, 2, ..., k-1 do

          for each son v of $u_i$ other than $u_{i+1}$ do

             {R <-- build(v);

             for each w on R do make $u_i$ the father of w in the virtual tree}
```

Algorithm 5: 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 1 of Fig. 3(a) Q contains nodes 6 and 1.

Lemma 6: Algorithm 5 requires O(n) steps and correctly builds the virtual tree.

We shall use the following strategy to simulate α. After all link instructions in α have been seen, we shall build the implied forest F. We make F into a tree T by making the roots of F direct descendants of a new root labeled "unrelated." We then construct from T a virtual tree 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 $\lfloor \log n \rfloor /2$ 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 T, we repeat this procedure, finding the $\lfloor \log n \rfloor /4$ 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 $\lfloor \log n \rfloor /4$ 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 T from u and v toward the 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 T.

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 T, given that

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

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

In what follows, we assume that count and high refer to the implied tree T and ancestor(p,i) to its virtual tree V. We assume that this information has already been computed.

```
procedure locate(u,v,i,j):
    if count(u) > count(v) then u,v,i,j <-- v,u,j,i;
    comment We assume that u now has the smaller count;
    if i = 0 then resultis ancestor(u,0)
    else {a <-- ancestor(u,i-1);
         if v = a then resultis a
         else if (a = undefined) or (v > a and high(v) < high(a))
              then resultis locate(u,v,i-1,j)
              else resultis locate(a,v,i-1,j)}
```

We now summarize the entire algorithm.

Algorithm 6: On-line execution of α, assuming all link instructions in α 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) Make F into a tree T by making the roots in F direct descendants of a new root labeled "unrelated."

(b) Using Algorithm 5, from T construct a virtual tree V.

(c) Use the procedure getancestor of Section 2.3 to compute ancestor(u,i) for all nodes u and $0 \leq i \leq \lfloor \log(1 + \lfloor \log n \rfloor) \rfloor$.

(d) Preorder the nodes of T and compute count(u) and high(u) for all nodes u.

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

Theorem 3: Algorithm 6 correctly simulates α.

5. DOMINATORS AND REDUCIBLE GRAPHS

We shall now apply Algorithm 1 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 nonempty 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 node u if every path from u_0 to u passes through d. That is, if $u_0, u_1, \ldots, u_i$ is a path with $u_i = u$, then there exists an integer j, $0 \leq j < i$ 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 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 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]. The flow graphs of goto-less 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

(1) eliminating common subexpressions [9,10],

(2) propagating constants [4],

(3) eliminating useless definitions [4], and

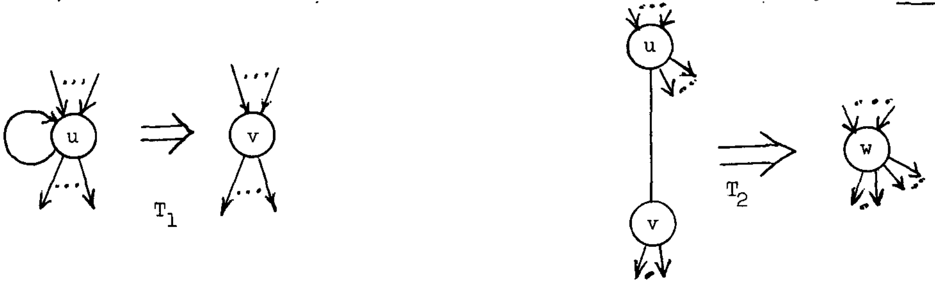
(4) finding active variables [11]

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 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.



(a) Transformation T_1

(b) Transformation T_2

Fig. 4.

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. We should observe that every rooted directed acyclic graph is reducible.

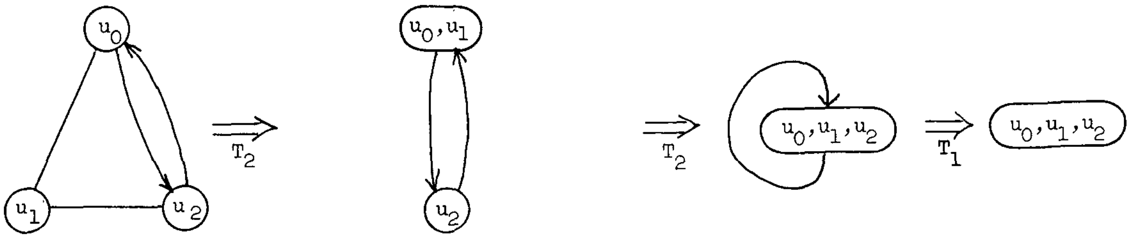


Fig. 5. Reduction of a Flow Graph

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 sucessor of u_2. The loop is then removed by T_1.

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.

6. 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 7: 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

$$\begin{array}{l} \underline{\mathrm{lca}}(u_1,u_2) \\ \underline{\mathrm{lca}}(v_1,u_3) \\ \quad\cdot \\ \quad\cdot \\ \quad\cdot \\ \underline{\mathrm{lca}}(v_{k-2},u_k) \\ \underline{\mathrm{link}}(u_0,v_{k-1}) \end{array}$$

and simulate them by Algorithm 1. (Note that neither Algorithm 4 or 6 is sufficient. Direct on-line simulation is required.) Here u_0 is the header of the consumed region, $u_1,\ldots,u_k$ 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 $2 \leq i < k$. If $k = 1$, the sequence is just $\underline{\mathrm{link}}(u_0,u_1)$.

(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 7 correctly constructs the dominator tree and requires O(E log E) steps on a reducible flow graph with E edges.

Corollary: Algorithm 7 works for and requires O(E log E) steps on a rooted directed acyclic graph with E edges.

7. 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 G(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.

A number of open questions remain.

(1) Are the algorithms given here the best possible for their respective problems?

(2) Can the techniques in Algorithm 7 be extended to arbitrary (irreducible) flow graphs?

(3) Can the *link-lca* problem be used effectively as a subproblem to help provide faster algorithms for problems other than the computation of dominators?

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.

[2] J. E. Hopcroft and J. D. Ullman, "Set Merging Algorithms," Submitted to SIAM J. Computing.

[3] E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Comm. ACM*, 12:1 (January 1969), 13-22.

[4] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling*, Volume II: *Compiling*, Prentice-Hall, Englewood Cliffs, N.J., January 1973.

[5] M. Schaefer, *A Mathematical Theory of Global Flow Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[6] P. W. Purdom and E. F. Moore, "Algorithm 430: Immediate Predominators in a Directed Graph," *Comm. ACM*, 15:8 (August 1972), 777-778.

[7] F. E. Allen, "Control Flow Analysis," *SIGPLAN Notices*, 5:7 (July 1970), 1-19.

[8] J. Cocke and R. E. Miller, "Some Analysis Techniques for Optimizing Computer Programs," *Proc. Second International Conference on System Sciences*, Honolulu, Hawaii, 1969.

[9] J. Cocke, "Global Common Subexpression Elimination," *SIGPLAN Notices*, 5:7 (July 1970), 20-24.

[10] J. D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions," Technical Report TR-106, Department of Electrical Engineering, Computer Sciences Laboratory, March, 1972. Also in *Proc. IEEE 13th Annual Symposium on Switching and Automata Theory*, October, 1972,

[11] K. Kennedy, "A Global Flow Analysis Algorithm," International J. Computer Mathematics, 3:1 (December 1971), 5-16.

[12] M. S. Hecht and J. D. Ullman, "Flow Graph Reducibility," SIAM J. Computing, 1:2 (June 1972), 188-202.

[13] R. E. Tarjan, "Testing Flow Graph Reducibility," Department of Computer Science, Cornell University, Ithaca, N.Y.