SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

FAST ALGORITHMS FOR FINDING NEAREST COMMON ANCESTORS*

DOV HAREL† AND ROBERT ENDRE TARJAN‡

Abstract. We consider the following problem: Given a collection of rooted trees, answer on-line queries of the form, "What is the nearest common ancester of vertices x and y?" We show that any pointer machine that solves this problem requires $\Omega(\log \log n)$ time per query in the worst case, where n is the total number of vertices in the trees. On the other hand, we present an algorithm for a random access machine with uniform cost measure (and a bound of $O(\log n)$ on the number of bits per word) that requires $O(1)$ time per query and $O(n)$ preprocessing time, assuming that the collection of trees is static. For a version of the problem in which the trees can change between queries, we obtain an almost-linear-time (and linear-space) algorithm.

Key words. graph algorithm, nearest common ancestor, tree, inverse Ackermann's function, random access machine, computational complexity

1. Introduction.

Aho, Hopcroft, and Ullman [2] consider the following problem: Given a collection of rooted trees,[1] answer queries of the form, "What is the nearest common ancestor of vertices x and y?" (We shall denote this vertex by *nca* (x, y).) There are actually many versions of this problem, depending upon whether the queries are all specificed in advance and how much the trees change during the course of the queries. We shall consider the five versions of the problem listed below, given in order from least dynamic (easiest) to most dynamic (hardest).

Problem 1 (off-line). The collection of trees is static and the entire sequence of queries is specified in advance.

Problem 2 (static trees). The collection of trees is static but the queries are given on-line. That is, each query must be answered before the next one is known.

Problem 3 (linking roots). The queries are given on-line. Interspersed with the queries are on-line commands of the form *link* (x, y) where x and y are tree roots. The effect of a command *link* (x, y) is to combine the trees containing x and y by making x the parent of y.

Problem 4 (linking). The queries are on-line. Interspersed with the queries are on-line commands *link* (x, y) such that y, but not necessarily x, is a tree root.

Problem 5 (linking and cutting). The queries are on-line. Interspersed with the queries are on-line commands of two types: *link* (x, y), where y but not necessarily x is a tree root, and *cut* (x), where x is not a root. The effect of a command *cut* (x) is to cut the edge connecting x and its parent, splitting the tree containing x into two trees: one containing all descendants of x and another containing all nondescendants of x.

In our discussion, we shall use n to denote the total number of vertices in the trees and m to denote the total number of operations (queries, links, and cuts). We shall distinguish between two machine models: *pointer machines* [8], [12] and *random access machines* [1]. In a pointer machine, memory consists of a collection of *nodes* (sometimes called *records*). Each node consists of a fixed number of *fields*. The fields have associated *types*, such as *pointer*, *integer*, *real*. A field of a given type contains a value of that type; for instance, a pointer field contains a pointer to a node. In a

* Received by the editors May 24, 1982, and in final form July 21, 1983.

† University of New Hampshire, Durham, New Hampshire 03824. Present address: Intermetrics, Inc., Cambridge, Massachusetts 02138.

‡ Bell Laboriatories, Murray Hill, New Jersey 07974.

[1] See the appendix for the tree terminology used in this paper.

random access machine the memory is an array of words, each of which holds an integer expressed in binary.

For measuring time on a random access machine, we use the *uniform cost measure*, in which each operation on a word or pair of words (such as an addition, comparison or branch) requires $O(1)$ time. We place an upper bound of $O(\log n)$ on the number of bits a word can hold, thereby precluding fast algorithms that obtain the effect of parallelism by manipulating very large integers. The main difference between our machine models is that address arithmetic is possible on random access machines but not on pointer machines.

Let us review what is known about Problems 1–5. In their seminal paper, Aho, Hopcroft, and Ullman consider Problems 1, 2 and 4. They describe an $O(n+m\alpha(m+n, n))$-time algorithm running on a pointer machine for Problem 1 (off-line). Here α is the functional inverse of Ackermann's function defined by Tarjan [10], [13]. Their algorithm requires $O(n)$ storage. For Problem 2 (static trees) they propose a random access machine algorithm requiring $O(n \log\log n)$ preprocessing time, $O(n \log\log n)$ space, and $O(\log\log n)$ time per query. This algorithm uses their algorithm for Problem 4 (linking), which also runs on a random-access machine and requires $O((m+n)\log n)$ time and $O(n \log n)$ space.

Several more recent papers improve and extend the results of Aho, Hopcroft, and Ullman. Van Leeuwen [14] considers Problem 3 (linking roots). He gives an $O(n+m \log\log n)$-time algorithm that can be modified to run on a pointer machine in $O(n)$ space. Maier [7] addresses Problem 5 (linking and cutting). Although his algorithm is not very time-efficient, his results do improve the space efficiency of Aho, Hopcroft, and Ullman's algorithm for Problem 4 (linking). Sleator and Tarjan [9] use their data structure for dynamic trees to solve Problem 5 in $O(n+m \log n)$ time and $O(n)$ space on a pointer machine.

Table 1 summarizes the known results. For Problem 1, Aho, Hopcroft, and Ullman's $O(n+m\alpha(m+n, n))$-time algorithm is the fastest known. For Problems 2 and 3, van Leeuwen's $O(n+m \log\log n)$-time algorithm is fastest. For Problems 4 and 5, Sleator and Tarjan's $O(n+m \log n)$-time algorithm is best. All these algorithms use $O(n)$ space and run on pointer machines.

TABLE 1
Best pointer machine algorithms for finding nearest common ancestors

Problem	Algorithm	Time	Space
1. Off-line	Aho, Hopcroft, and Ullman [2]; see also Tarjan [11]	$O(n+m\alpha(m+n, n))$	$O(n)$
2. Static	modified van Leeuwen [14]	$O(n+m \log\log n)$	$O(n)$
3. Linking roots	modified van Leeuwen [14]	$O(n+m\log\log n)$	$O(n)$
4. Linking	Sleater and Tarjan [9]	$O(n+m \log n)$	$O(n)$
5. Linking and cutting	Sleator and Tarjan [9]	$O(n+m \log n)$	$O(n)$

In this paper, our goal is to study the effect of the machine model on the nearest common ancester problem. Our results are three. In § 2 we show that any pointer machine requires $\Omega(\log\log n)$ time per query to solve Problem 2 (static trees). This means that van Leeuwen's algorithm is optimum to within constant factors for Problems 2 and 3. In §§ 3–5 we develop an algorithm for Problem 2 that runs on a random access machine and uses $O(n)$ preprocessing time, $O(1)$ time per query, and $O(n)$ space. This algorithm is also optimum to within a constant factor. Harel's paper [5]

TABLE 2
Random access machine algorithms for finding nearest common ancestors

Problem	Time	Space
1. Off-line	$O(n+m)$	$O(n)$
2. Static	$O(n+m)$	$O(n)$
3. Linking roots	$O(n+m\alpha(m+n, n))$	$O(n)$

gives a preliminary version of the results in §§ 2–5. In §§ 6 and 7 we extend our algorithm to Problem 3, for which we obtain an $O(n+m\alpha(m+n, n))$-time, $O(n)$-space algorithm. Our results thus explicate the difference in power between pointer machines and random-access machines. Table 2 summarizes our upper bounds.

2. A lower bound for pointer machines. In this and the next three sections we shall restrict our attention to the nearest common ancester problem on static trees (Problem 2). Without loss of generality we can assume that there is only one tree. If not, we create a new (dummy) vertex r and make it the parent of the roots of all the actual trees. The nearest common ancester *nca* (x, y) of two vertices x and y in the new tree is the same as the nearest common ancester of x and y in the collection of old trees; if *nca* $(x, y) = r$ in the new tree then x and y are in different old trees.

Let us consider pointer machine solutions to the nearest common ancester problem on a static tree. We make the following assumptions about the way such a machine solves the problem. We assume that the tree is represented by a list structure (which may change during the course of the computation), with each tree vertex represented by a single node. The structure may contain additional nodes not representing any tree vertex. Each node contains a fixed number of pointers, independent of n; without loss of generality we may take this number to be two. As input for a query, the algorithm is given pointers to the nodes corresponding to two tree vertices x and y. To answer the query, the algorithm must return a pointer to the node corresponding to the tree vertex *nca* (x, y). We assume that the algorithm remembers nothing between queries. (Any fixed amount of global memory can be encoded into the list structure.)

THEOREM 1. *Let T be a complete binary tree with n leaves. Any pointer machine requires $\Omega(\log \log n)$ time to answer any nca query in the worst case, independent of the representation of the tree.*

Proof. Let us fix our attention on the time just before a query. The key point is that, from any node in the data structure, at most $2^{j+1}-1$ nodes are accessible in j steps or less. Let k be such that any possible *nca* query can be answered in k steps or less. For each leaf x of T let A_x denote the set of nodes representing tree vertices that are accessible from x in k steps or less. Let w be a nonleaf vertex of T and let u and v be its two children. We claim that either w belongs to A_x for every leaf x that is a descendant of u, or w belongs to A_y for every leaf y that is a descendant of v. Otherwise, for some descendant x of u and some descendant y of v, w would be accessible from neither x nor y in k steps, and an *nca* query on x and y would be unanswerable in k steps.

We conclude that w belongs to A_x for at least half the leaves x that are descendants of w. If w has height $i \geqq 1$, then w has 2^i leaf descendants, and thus w occurs in at least 2^{i-1} sets A_x. Since if $1 \leqq i \leqq h = \lg n^2$ there are 2^{h-i} vertices of height i, we see

[2] We use $\lg n$ to denote $\log_2 n$.

that vertices of height i contribute $2^{h-i}2^{i-1}=2^{h-1}=n/2$ occurrences to the collection of sets A_x. Summing over all heights i from 1 to h, we find that

$$\sum_{x\in L} |A_x| \geqq \frac{n}{2}\lg n,$$

where L is the set of leaves of T. Since for any leaf x we have $|A_x|<2^{k+1}$, we get

$$n2^{k+1} > \frac{n}{2}\lg n,$$

which means

$$k > \lg\lg n - 2. \qquad \square$$

Theorem 1 implies that van Leeuwen's algorithm for static trees runs in optimum time, to within a constant factor. It also provides a nontrivial lower bound for pointer machines on a natural problem. (For a similar result see [12].) We conjecture that Problem 5 (linking and cutting) requires $\Omega(\log n)$ time per command in the worst case on a pointer machine, and leave the proof (or disproof) of this conjecture as an open problem.

3. Overview of a fast algorithm for static trees.

If we allow algorithms on random-access machines, we can beat the lower bound in Theorem 1. In this and the next two sections, we shall develop an algorithm that runs on a random-access machine in $O(n)$ space, using $O(n)$ preprocessing time and $O(1)$ time per query. These bounds are best possible; there are n^{n-1} distinct rooted trees with n labeled vertices [6], implying that $\Omega(n \log n)$ bits, or $\Omega(n)$ words, are necessary to store a tree of n vertices in a random-access machine. Just reading in such a tree requires $\Omega(n)$ time, and answering a query requires $\Omega(1)$ time.

We begin by observing that on complete binary trees the nearest common ancestor and related problems can be solved in $O(1)$ time by direct calculation. Let T be a complete binary tree whose vertices are numbered from 1 to n in symmetric order. (See Fig. 1.) We use *sym* (v) to denote the number of a vertex v and *sym*$^{-1}$ (i) to

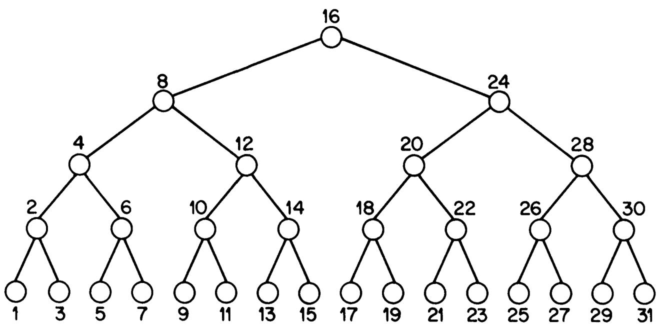

FIG. 1. *Symmetric-order numbering of a complete binary tree.*

denote the vertex whose number is i. For any height h, the vertices of height h are numbered $2^h, 3\cdot 2^h, 5\cdot 2^h, \cdots$ from left to right. It is easy to verify the following facts about the numbering:

LEMMA 1. *The height of a vertex v is the largest integer h such that 2^h divides* $sym(v)$. *We shall denote this height by* $h(v)$.

LEMMA 2. *The descendants of vertex v are those vertices with numbers in the range* $[sym(v)-2^{h(v)}+1,\ sym(v)+2^{h(v)}-1]$.

LEMMA 3. *If v is a vertex and h is a height such that $h \geqq h(v)$, then the ancestor of v of height h has number* $2^{h+1}\lfloor \mathrm{sym}(v)/2^{h+1}\rfloor+2^h$.

LEMMA 4. *If v and w are two unrelated vertices, the height of the nearest common ancestor of v and w is* $\lfloor \lg(sym(v)\oplus sym(w))\rfloor$, *where $i\oplus j$ is the integer whose binary representation is the bitwise exclusive or of the binary representations of i and j.*

As examples of Lemma 3, vertex number 22 has vertex number $20=8\lfloor 22/8\rfloor+4$ as its ancestor of height 2 and vertex number $24=16\lfloor 22/16\rfloor+8$ as its ancestor of height 3. (See Fig. 1.) As an example of Lemma 4, the nearest common ancestor of vertices number 20 and number 27 is number 24, of height 3; $\lfloor \lg(20\oplus 27)\rfloor = \lfloor \lg(10100_2)\oplus 11011_2\rfloor = \lfloor \lg(1111_2)\rfloor = 3$.

Lemmas 2 and 4 allow us to solve the following problem in $O(1)$ time, given that we know the number and height of each vertex and the depth d of the tree:

The nca depth problem. Given vertices v and w, compute the depth of the nearest common ancestor of v and w.

Algorithm to solve the nca depth problem. If v is an ancestor of w (that is, if $sym(w)\in[sym(v)-2^{h(v)}+1,\ sym(v)+2^{h(v)}-1]$), return $d-h(v)$. If w is an ancestor of v, return $d-h(w)$. If v and w are unrelated, return $d-\lfloor \lg(sym(v)\oplus sym(w))\rfloor$.

Lemmas 1 and 3 provide a way to solve the following problem in $O(1)$ time, if we know the number of each vertex, the vertex corresponding to each number, and the depth d of the tree:

The depth problem. Given a vertex v of depth d_1 and a depth $d_2 \leqq d_1$, compute the ancestor of v whose depth is d_2.

Algorithm to solve the depth problem.
Let $h=d-d_2$. Return $sym^{-1}(2^{h+1}\lfloor sym(v)/2^{h+1}\rfloor+2^h)$.

These algorithms combine to give an $O(1)$-time algorithm for the nearest common ancestor problem:

Algorithm to compute nca(v, w).
Step 1. Compute d_0, the depth of *nca* (v, w).
Step 2. Compute and return the depth-d_0 ancestor of v.

The $O(1)$ time bounds for these three algorithms depend on the ability to perform multiplication, division, powers of two, base two discrete logarithm, and exclusive or in $O(1)$ time. If these operations are not part of the machine's repertoire, we can in $O(n)$ time construct operation tables that allow these operations to be carried out in $O(1)$ time using (possibly repeated) table look-up.

Let us turn now to the nearest common ancestor problem on an arbitrary tree T. Our plan is to convert the *nca* problem on T into an *nca* problem on a subtree of a moderately-sized complete binary tree; then we can use the method above. The transformation proceeds by a sequence of steps, which involve solving depth and *nca* depth problems on two auxiliary trees: a *compressed tree C* and a *balanced binary tree B. C* has the same vertex set as T; B contains all vertices in T and possibly some auxiliary vertices. To facilitate the solution of depth problems on B C, both of these trees are divided into *plies.* We construct B and C in a preprocessing step requiring $O(n)$ time. We compute nearest common ancestors as follows:

Algorithm to compute $nca_T(v, w)$.
Step 1. Compute $nca_C(v, w)$ as follows:
Step 1a. Compute $nca_B(v, w)$ using algorithms for the *nca* depth problem and the depth problem on B.
Step 1b. Given $nca_B(v, w)$, look up $nca_C(v, w)$.
Step 2. Look up the depth in C of $nca_C(v, w)$. Using an algorithm for the depth problem on C, compute $nca_T(v, w)$.

Each step of the *nca* computation takes O(1) time. Sections 4 and 5, which discuss the compressed tree C and the balanced binary tree B respectively, give the details of this method.

4. The compressed tree. Let T be an arbitrary n-vertex tree with root r. We define a compressed tree C, representing T, as follows. For each vertex v in T, let *size* (v) be the number of descendants of v (including v itself) in T. Define an edge $v \to p_T(v)$ to be *light* if $2 \cdot size_T(v) \leqq size_T(p_T(v))$ and *heavy* otherwise. Since the size of a vertex is one greater than the sum of the sizes of its children, at most one heavy edge enters each vertex. Thus the heavy edges partition the vertices of T into a collection of *heavy paths.* (A vertex with no entering or exiting heavy edge is a single-vertex heavy path.) (See Fig. 2.)

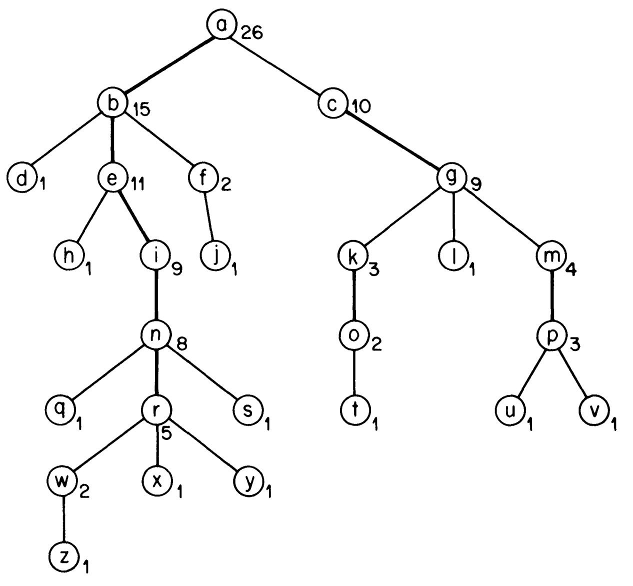

FIG. 2. *Heavy and light edges in a tree. Numbers are vertex sizes.*

The *apex* of a heavy path is the vertex on the path of smallest depth. For any vertex v we denote by *apex* (v) the apex of the heavy path containing v, and by *hp size* (v) the number of descendants of v on the same heavy path as v. The *compressed tree* C is defined by the set of edges

$$\{v \to apex\,(p_T(v)) \mid v \text{ is a vertex of } T \text{ other than } r\}.$$

(See Fig. 3.) The compressed tree was used by Tarjan [10] to compute functions defined on paths in trees; Aho, Hopcroft, and Ullman used a closely related idea in their *nca* algorithm for static trees.

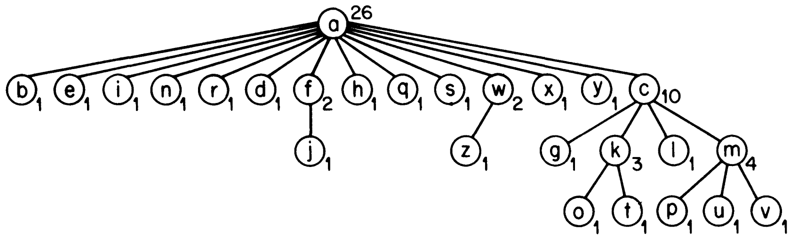

FIG. 3. *The compressed tree corresponding to the tree of Fig. 2. Numbers are vertex sizes.*

In $O(n)$ time, we can compute the following information for each vertex v: $p_T(v)$, $p_C(v)$, *apex* (v), *hp size* (v), $d_C(v)$ (the depth of v in C), and $size_C$ (v) (the size of v in C). Note that we can use *apex* to test in $O(1)$ time whether two vertices are on the same heavy path in T. To compute $nca_T(v, w)$, we first compute $nca_C(v, w)$ using the balanced tree B. (We describe how to do this in the next section.) Then we proceed as follows:

Step 2 *of algorithm to compute* nca_T (v, w):

Step 2a. Let $u = nca_C$ (v, w). (Either $u = v = w$ or u is the apex of the heavy path containing $nca_T(v, w)$). If $u = v$ or $u = w$, return u as nca_T (v, w). Otherwise look up $d_C(u)$.

Step 2b. Compute the ancestor v' of v in C whose depth is $d_C(u)+1$. If *apex* $(v') = u$ (u and v' are on the same heavy path) let $v'' = v'$. Otherwise let $v'' = p_T(v')$.

Step 2c. Compute the ancestor w' of w in C whose depth is $d_C(u)+1$. If *apex* $(w') = u$ (u and w' are on the same heavy path) let $w'' = w'$. Otherwise let $w'' = p_T(w')$.

Step 2d. Return as $nca_T(v, w)$ whichever of v'' and w'' has the larger value of *hp size.*

As an example of Step 2, consider nca_T (n, o), where T is the tree of Fig. 2. We have $nca_C(n, o) = a$ (see Fig. 3), $n' = n$, $n'' = n$, $o' = c$, $o'' = a$, and nca_T $(n, o) = a$. We omit the easy proof that Step 2 is correct; crucial to the proof is the observation that $v''(w'')$ is the nearest ancestor of v (respectively w) on the heavy path containing u. Step 2 requires $O(1)$ time plus the solution of at most two depth problems on C. To solve depth problems on C, we must do some additional preprocessing. We need some simple facts about the structure of C, which we state without proof. (See [11].)

LEMMA 5. *If v is an apex, $size_C$ $(v) = size_T$ (v); if v is not an apex, $size_C(v) = 1$.*

LEMMA 6. *Every edge $v \rightarrow p_C(v)$ of C satisfies $2 \cdot size_C$ $(v) \leqq size_C$ $(p_C(v))$.*

LEMMA 7. *C has depth at most* $\lfloor \lg n \rfloor$.

We divide C into three plies as follows. Define the *rank* of a vertex v to be *rank* $(v) = \lfloor \lg (size_C\ (v)) \rfloor$. Ply three consists of all vertices with rank $\lfloor \lg^{(2)} n \rfloor$[3] or greater; ply two consists of all vertices with rank between $\lfloor \lg^{(3)} n \rfloor$ and $\lfloor \lg^{(2)} \rfloor - 1$ (inclusive); ply one consists of all vertices with rank less than $\lfloor \lg^{(3)} n \rfloor$. Fig. 4 schematically illustrates this definition.

LEMMA 8. *For any rank i, the number of vertices of rank i is at most $n/2^i$.*

Proof. By Lemma 6, any two vertices having the same rank are unrelated in C and hence have disjoint sets of descendants in C. Any vertex v of rank i has $size_C$ $(v) \geqq 2^i$. Thus there can be at most $n/2^i$ vertices of rank i. □

[3] For any nonnegative integer i and function f, we define $f^{(i)}(x)$ by $f^{(0)}(x) = x$, $f^{(i+1)}(x) = f(f^{(i)}(x))$.

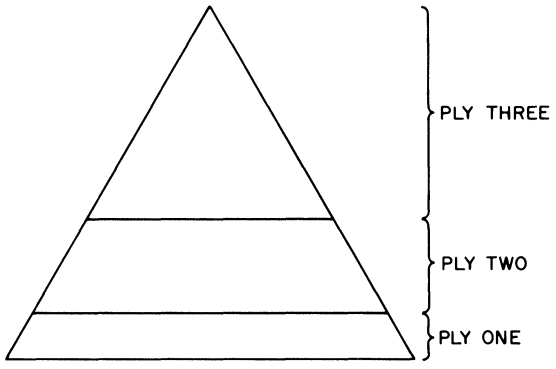

FIG. 4. *Plies of a compressed tree.*

LEMMA 9. *Ply three contains at most $O(n/\log n)$ vertices. Ply two contains at most $O(n/\log^{(2)} n)$ vertices. Each connected component of ply one is a subtree of C containing at most $\log^{(2)} n$ vertices.*

Proof. By Lemma 8, the total number of vertices with rank k or greater is at most $\sum_{i=k}^{\infty} n/2^i = n/2^{k-1}$. This implies that ply three contains at most $4n/\lg n$ and ply two at most $4n/\lg^{(2)} n$ vertices. Each connected component of ply one is a subtree of C whose root has rank at most $\lfloor \lg^{(3)} n \rfloor - 1$ and thus contains at most $2^{\lfloor \lg^{(3)} n \rfloor} \leqq \lg^{(2)} n$ vertices. □

With each vertex v in ply i ($i \in \{1, 2, 3\}$), we store the vertex $a(v)$ such that $a(v)$ is the shallowest ancestor of v in C whose ply is i. We also represent each ply so that it is possible to carry out the following computation in $O(1)$ time: given a vertex v in ply i and a depth $d \in [d_C(a(v)), d_C(v))]$, compute the ancestor of v whose depth is d. (We shall describe this representation below.) Then we can solve the depth problem on C as follows.

Algorithm to compute the ancestor of vertex v in C whose depth is d. Repeat the following step until a vertex is returned:

General step. If $d \in [d_C(a(v)), d_C(v)]$ compute and return the ancestor of v whose depth is d. Otherwise, replace v by $p_C(a(v))$.

This method requires $O(1)$ time. (The ply of v increases by at least one with each iteration of the general step.)

The representation of plies two and three is very simple: with each vertex v of ply i ($\in \{2, 3\}$), we store an array of all ply-i ancestors of v, indexed by depth. Then we can solve a depth problem within ply two or three in $O(1)$ time by table look-up. Note that the size of all arrays for ply three is $O((n/\log n) \log n) = O(n)$ by Lemma 9, since any vertex has $O(\log n)$ ancestors in C. Similarly the size of all arrays for ply two is $O((n/\log^{(2)} n)\log^{(2)} n = O(n)$ by Lemma 9, since any vertex has $O(\log^{(2)} n)$ ply-two ancestors in C. It is straightforward to divide the vertices into plies, compute $a(v)$ for every vertex, and construct all the arrays for plies two and three in $O(n)$ time.

Ply one has very small connected components but contains most of the vertices in C, and we represent it differently than plies two and three. Let v be any vertex in ply one. We denote by $D_{a(v)}$ the subtree of C rooted at $a(v)$; this is the connected component of ply one containing v. $D_{a(v)}$ contains at most $\lg^{(2)} n$ vertices and has depth at most $\lg^{(3)} n$. Let d be the depth of $D_{a(v)}$. We embed the vertices of $D_{a(v)}$ in a complete binary tree $E_{a(v)}$ of depth $d\lceil \lg^{(3)} n \rceil$ so that a vertex of depth i in $D_{a(v)}$

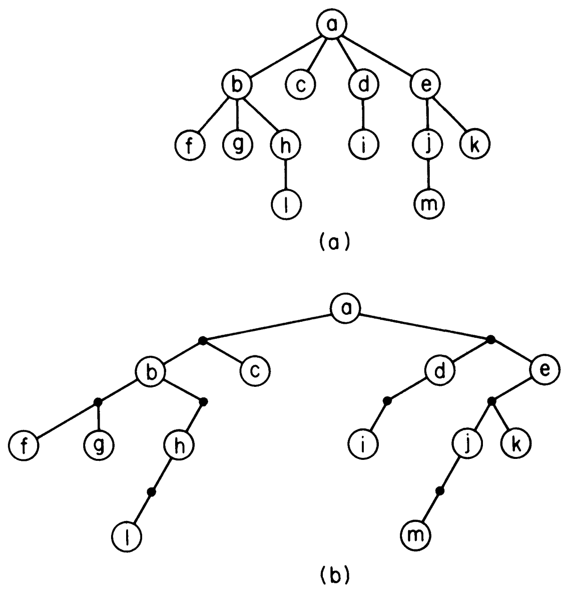

FIG. 5. *Embedding a multiway tree into a complete binary tree.* (a) *Multiway tree.* (b) *Part of complete binary tree containing the multiway tree.*

has depth $i\lfloor \lg^{(3)} n\rceil$ in $E_{a(v)}$, and so that if v is a vertex of depth i in $D_{a(v)}$ and $0 \leqq i' \leqq i$, the ancestor of v in $D_{a(v)}$ whose depth is i' is also the ancestor of v in $E_{a(v)}$ whose depth is $i'\lceil \lg^{(3)} n\rceil$. (See Fig. 5.) Then we can solve depth problems in ply one using the method of § 3. The algorithm is as follows:

Algorithm to compute the ancestor of v in C whose depth is d (v is in ply one and $d \geqq d_C(a(v))$).

Let $h = (d_c(a(v)) - d)\lceil \lg^{(3)} n\rceil$. Return

$$sym_{a(v)}^{-1}\,(2^{h+1}\lfloor sym\,(v)/2^{h+1}\rfloor + 2^h).$$

The notation in this algorithm requires a little explanation. If v is any vertex in ply one, *sym* (v) is the symmetric-order number of v corresponding to its position in $E_{a(v)}$, and $sym_{a(v)}^{-1}\,(i)$ is the vertex in $E_{a(v)}$ with number i, if this vertex is in $D_{a(v)}$; otherwise $sym_{a(v)}^{-1}\,(i)$ is undefined. This method solves depth problems within ply one in $O(1)$ time, if *sym* and $sym_{a(v)}^{-1}$ are precomputed.

The difficulty with this method is that the trees $E_{a(v)}$ are too big to construct explicitly, so we must represent them implicitly. To represent the trees $E_{a(v)}$, we store two numbers, *sym* (v) and *pre* (v), with each vertex v in ply one, and two arrays, $pre_{a(v)}^{-1}$ and $inverse_{a(v)}$, with each vertex $a(v)$ in ply one. The values of *pre*, pre^{-1}, *inverse* are defined as follows: We number the vertices of each tree $D_{a(v)}$ in preorder from 1 to $|D_{a(v)}|$. If v is a vertex in $D_{a(v)}$, *pre* (v) is the number of v and $pre_{a(v)}^{-1}\,(pre\,(v)) = v$. Note that the total size of the arrays $pre_{a(v)}^{-1}$ for all vertices $a(v)$ in ply one is $O(n)$. For $i \in [1, |E_{a(v)}|]$, $inverse_{a(v)}(i)$ is the pre-order number of the vertex in $D_{a(v)}$ whose symmetric-order number in $E_{a(v)}$ is i; $inverse_{a(v)}\,(i) = 0$ if the vertex in $E_{a(v)}$ whose symmetric-order number is i is not in $D_{a(v)}$. Note that $inverse_{a(v)}$ contains $|D_{a(v)}|$ nonzero entries and needs $O(|E_{a(v)}| \lg |D_{a(v)}|) = 2^{O((\lg^{(3)} n)^2)} \lg^{(3)} n = O(\log n)$ bits for its storage. Thus the total number of nonzero entries in all the $inverse_{a(v)}$ arrays is $O(n)$, and each $inverse_{a(v)}$ array fits into one word of storage.

Given *sym* (v) for each vertex v in ply one and $pre^{-1}_{a(v)}$ and $inverse_{a(v)}$ for each vertex $a(v)$ in ply one, we can solve depth problems within ply one in $O(1)$ time, since $sym^{-1}_{a(v)}(i) = pre^{-1}_{a(v)}(inverse_{a(v)}(i))$ if i is the symmetric-order number in $E_{a(v)}$ of a vertex in $D_{a(v)}$. It is easy to construct the $pre^{-1}_{a(v)}$ arrays in $O(n)$ time. We can construct the *sym* numbers and the $inverse_{a(v)}$ arrays by traversing each ply-one tree in preorder as follows: Let $D_{a(v)}$ be a ply one tree of height h, to be embedded in the complete binary tree $E_{a(v)}$. We initialize $inverse_{a(v)}$ to be zero. Then we execute $traverse(a(v), h\lceil \lg^{(3)} n\rceil, 1)$, where *traverse* is defined as follows:

```
Procedure traverse(v, h, i):
  [$v$ is a vertex in $D_{a(v)}$, $h$ is the height of $v$ in $E_{a(v)}$, and $i$ is the symmetric-order
    number of the smallest-numbered descendant of $v$ in $E_{a(v)}$]
  Step1. Let $j = 2^h\lceil i/2^h\rceil$ ($j$ is the symmetric-order number of $v$ in $E_{a(v)}$). Assign
    $sym(v) = j$ and $inverse_{a(v)}(j) = pre(v)$.
  Step 2. Initialize $k = i$. For each child $w$ of $v$, carry out the following steps:
  Step 2a. Execute $traverse(w, h - \lceil \lg^{(3)} n\rceil, k)$.
  Step 2b. Replace $k$ by $k + 2^{h-\lceil \lg^{(3)} n\rceil+1}$.
```

It is routine to verify that this method is correct and requires $O(n)$ time to preprocess all the ply-one trees. This completes our description of the algorithm for solving depth problems on C and with it our description of the role of C in the nearest common ancestor algorithm. As noted by Hal Gabow (private communication), it is possible to simplify the solution of depth problems in ply one by using the table look-up technique of Gabow and Tarjan [4]. This requires additional preprocessing, but the additional time is only $O(n)$.

5. The balanced binary tree.

To complete our solution of the *nca* problem, we need a way to solve the *nca* problem on the compressed tree C. For this purpose we represent C by a second auxiliary tree B called the *balanced binary tree.* B contains all the vertices of C, called *green vertices*, and some additional vertices, called *red vertices.* For each family in C consisting of a parent v and a set of children W, B contains a subtree whose root is v, whose leaves are the vertices in W, and whose remaining vertices are red. Each vertex v contains a pointer to its nearest green ancestor *green* (v). Nearest common ancestors in B and C are related by the equation

$$nca_C(v, w) = green(nca_B(v, w))$$

if v and w are vertices in C. Thus if we can solve the *nca* problem on B, we can solve the *nca* problem on C in $O(1)$ additional time.

Thus it remains for us to describe how to construct B and how to solve the *nca* problem on B. Consider a family in C consisting of a parent v and a set of children W such that $|W| \geq 3$. To binarize the family, we execute the recursive procedure *binarize* (v, W), where *binarize* is defined as follows:

```
Procedure binarize (v, W):
  Step 1. Let $W = \{w_1, \cdots, w_k\}$ and $s = \sum_{i=1}^k size_C(w_i)$.Let $j$ be the minimum index
    such that $\sum_{i=1}^j size_C(w_i) \geq s/2$. If $j = k$, replace $j$ by $k-1$.
  Step 2. If $j = 1$, attach $w_1$ as the left child of $v$. Otherwise, let $x_1$ be a new red
    vertex. Attach $x_1$ as the left child of $v$ and execute binarize $(x_1, W_1)$, where
    $W_1 = \{w_1, \cdots, w_j\}$.
  Step 3. If $j = k-1$, attach $w_k$ as the right child of $v$. Otherwise, let $x_2$ be a new
    red vertex. Attach $x_2$ as the right child of $v$ and execute binarize $(x_2, W_2)$,
    where $W_2 = \{w_{j+1}, \cdots, w_k\}$.
```

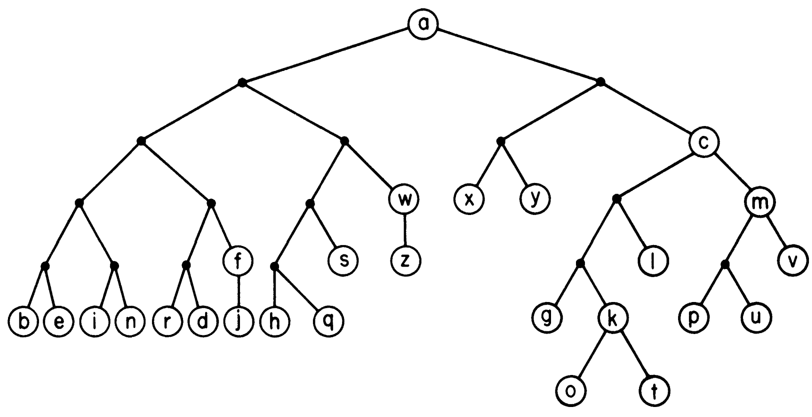

FIG. 6. *Balanced binary tree corresponding to the compressed tree of Fig. 3. Dots are red vertices.*

(See Fig. 6.) This method can be implemented to run in $O(|W|)$ time [3]. The idea is to construct an array of size k, whose jth position contains $\sum_{i=1}^{j} size_C\ (w_i)$. Then one can find the appropriate index j in $O(\log(\min\{j, k-j\}))$ time by using a form of binary search simultaneously from both ends of the array. The same array can be used for all the recursive subproblems. The running time $t(k)$ of the method is given by the recurrence

$$t(k) = \max_{0<j<k} \{t(j) + t(k-j) + O(\log(\min\{j, k-j\}) + 1)\},$$

which has the solution $t(k) = O(k)$.

To construct B, we binarize each family of C using the method above. Since B contains at most $2n-1$ vertices, the total time to construct B is $O(n)$.

Let us define $size_B\ (v)$ for a vertex v in B to be the number of green descendants of v in B. The following lemmas show that the tree B has properties similar to those of C:

LEMMA 10. *If v is a green vertex $size_B\ (v) = size_C\ (v)$.*

Proof. Immediate. □

LEMMA 11. *If v is a vertex such that $p_B^4(v)$ is defined, then $2\ size_B\ (v) \leqq size_B\ (p_B^4(v))$.*

Proof. If the path in B from v to $p_B^4(v)$ contains two green vertices, the lemma is immediate from Lemmas 6 and 10. Otherwise, the path from v to $p_B^4(v)$ contains three consecutive vertices $w \to p_B(w) \to p_B^2(w)$ such that w and $p_B(w)$ are red. If $2 \cdot size_B\ (w) > size_B\ (p_B^2(w))$, the operation of *binarize* guarantees that w is the left child of $p_B(w)$ and $p_B(w)$ is the left child of $p_B^2(w)$. But the right children of $p_B(w)$ and $p_B^2(w)$ together have less than half the size in C of $p_B^2(w)$, which contradicts the operation of *binarize* on $p_B^2(w)$. □

LEMMA 12. *B has depth $O(\log n)$.*

Proof. Immediate from Lemma 11. □

We solve the *nca* problem on B by solving the *nca* depth problem and the depth problem on B. To solve the *nca* depth problem on B we embed B in a complete binary tree B' and use direct calculation as described in § 3; all we need to know for each vertex in B is its symmetric-order number and height (as a vertex in B'). To number the vertices of B we execute the recursive procedure *number*$(r, h, 2^h)$, where

r is the root of B, h is the height of B, and *number* is defined as follows:

```
Procedure number $(v, h, i)$: [$v$ is a vertex in $B$, $h$ is the height of $v$ in $B'$, and $i$ is the symmetric-order number of $v$ in $B'$]
Step 1. Assign number $i$ and height $h$ to $v$.
Step 2. If $v$ has a left child $w_1$, execute number $(w_1, h-1, i-2^{h-1})$.
Step 3. If $v$ has a right child $w_2$, execute number $(w_2, h-1, i+2^{h-1})$.
```

This computation takes $O(n)$ time. Once the vertices of B are numbered, we can solve the *nca* depth problem on B in $O(1)$ time.

We solve the depth problem on B in the same way we solved it on C. For each vertex v in B, we define $rank_B(v) = \lfloor \lg(size_B(v)) \rfloor$. We divide B into three plies as follows: Ply three consists of all vertices with rank $\lfloor \lg^{(2)} n \rfloor$ or greater, ply two consists of all vertices with rank between $\lfloor \lg^{(3)} n \rfloor$ and $\lfloor \lg^{(2)} n \rfloor - 1$ (inclusive), and ply one consists of the remaining vertices. Lemma 9 holds for the plies in B just as for the plies in C, except that the constants are larger by a factor of four because the weaker Lemma 11 holds for B in place of Lemma 6. We represent the plies of B exactly as we did the plies of C, and we solve the depth problem in the same way. We can simplify the representation of ply one a little because B is binary; in particular, we do not need to binarize the ply one trees. Filling in the details is routine and we leave it as an exercise.

Constructing the plies of B requires $O(n)$ preprocessing time and allows us to solve the depth problem on B in $O(1)$ time. Hence we can solve the *nca* problem on B, on C, and on T in $O(1)$ time. This completes our description of the *nca* algorithm for static trees.

6. A fast algorithm for the linking roots problem.

We now turn our attention to the linking roots problem. In this and the next section we shall develop an algorithm for this problem that runs in $O(n + m\alpha(m+n, n))$ time and $O(n)$ space on a random-access machine. We begin in this section by developing a simpler $O(n + m \log^* n)$-time algorithm.

We shall use $\mathcal{T}$ to denote the set of trees defined by the link operations. We maintain the following information for each vertex v: $p_{\mathcal{T}}(v)$, $size_{\mathcal{T}}(v)$, a list of the children of v in $\mathcal{T}$, the child w of v (if any) such that $w \to v$ is a heavy edge, *hp size* (v) (the number of descendants of v on the same heavy path as v), and *light* (v), defined to be true if $p_{\mathcal{T}}(v)$ is defined and $v \to p_{\mathcal{T}}(v)$ is light, and false otherwise. It is easy to update this information in $O(1)$ time per link. Note that a link can cause a formerly heavy edge to become light (but not vice versa) and can create either a heavy or a light edge. We also maintain a data structure [1], [11] that allows us to rapidly compute, for any vertex v, the root $r_{\mathcal{T}}(v)$ of the tree in $\mathcal{T}$ containing v. Updating this data structure during links and carrying out $O(m)$ root calculations requires $O(n + m\alpha(m+n, n))$ time.

Corresponding to $\mathcal{T}$ we maintain a forest $\mathcal{C}$ of compressed trees; $\mathcal{C}$ contains one or more trees representing each tree in $\mathcal{T}$. We build the trees in $\mathcal{C}$ family-by-family in delayed fashion. When a link creates a light edge $v \to p_{\mathcal{T}}(v)$ or causes a formerly heavy edge $v \to p_{\mathcal{T}}(v)$ to become light, we explore the heavy path σ whose apex is v, constructing the set $S = \{w \neq v \mid w \text{ is on } \sigma \text{ or } p_{\mathcal{T}}(w) \text{ is on } \sigma\}$. We then combine the compressed trees whose roots are in S into a single compressed tree; the root of this tree is v and the children of the root are the vertices in S. In general a tree T in $\mathcal{T}$ is represented by several compressed trees; if r is the root of T and σ is the heavy path whose apex is r then $\mathcal{C}$ contains a tree for every vertex v on σ whose single vertex is v, and a tree for every vertex w such that $p_{\mathcal{T}}(w)$ but not w is on σ whose vertices are all the descendants of w in T. (See Fig. 7.)

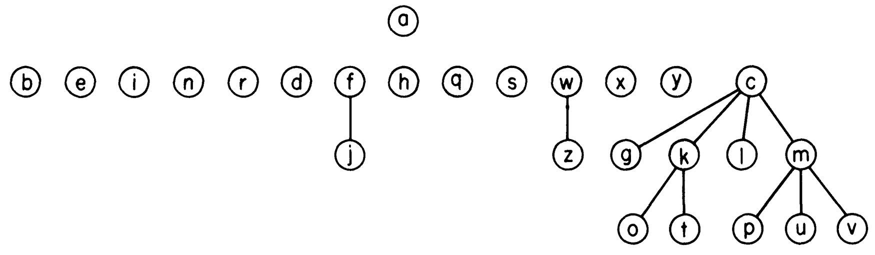

FIG. 7. *Set of compressed trees corresponding to the tree in Fig. 2, for the linking roots problem (see Fig. 3).*

We maintain certain information about the compressed trees. With each vertex v we store $size_{\mathscr{C}}(v)$. With each vertex v such that $p_{\mathscr{C}}(v)$ is defined, we store $p_{\mathscr{C}}(v)$ and *apex* (v). Updating this information during links requires $O(n)$ total time. We also maintain a data structure [1], [11] that allows us to rapidly compute two pieces of information about any vertex v: $d_{\mathscr{C}}(v)$ (the depth of v in the forest $\mathscr{C}$) and $r_{\mathscr{C}}(v)$ (the root of the tree in $\mathscr{C}$ containing v). Maintaining this data structure during links and carrying out $O(m)$ root and depth computations requires $O(n+m\alpha(m+n, n))$ time. We compute nearest common ancestors in $\mathscr{T}$ using the following variant of the method in §§ 3 and 4:

Algorithm to compute $nca_{\mathscr{T}}(v, w)$.

Step 1. Compute $r_{\mathscr{T}}(v)$ and $r_{\mathscr{T}}(w)$. If $r_{\mathscr{T}}(v) \neq r_{\mathscr{T}}(w)$, return a message that v and w are in different trees.

Step 2. Compute $r_{\mathscr{C}}(v)$ and $r_{\mathscr{C}}(w)$. If $r_{\mathscr{C}}(v)=r_{\mathscr{C}}(w)$, go to Step 3. Otherwise (v and w are in different compressed trees), let $v'=r_{\mathscr{C}}(v)$ if *light* $(r_{\mathscr{C}}(v))$ is false, $v'=p_{\mathscr{T}}(r_{\mathscr{C}}(v))$ if *light* $(r_{\mathscr{C}}(v))$ is true. Let $w'=r_{\mathscr{C}}(w)$ if *light* $(r_{\mathscr{C}}(w))$ is false, $w'=p_{\mathscr{T}}(r_{\mathscr{C}}(v))$ if *light* $(r_{\mathscr{C}}(w))$ is true. Return as $nca_{\mathscr{T}}(v, w)$ whichever of v' and w' has the larger value of *hp size.*

Step 3 (v and w are in the same compressed tree). Compute $u=nca_{\mathscr{C}}(v, w)$. If $u=v$ or $u=w$, return u as $nca_{\mathscr{T}}(v, w)$. Otherwise, compute $d_{\mathscr{C}}(u)$.

Step 4. Compute the ancestor v' of v in $\mathscr{C}$ whose depth is $d_{\mathscr{C}}(u)+1$. If *apex* $(v')=u$, let $v''=v'$. Otherwise let $v''=p_{\mathscr{T}}(v')$.

Step 5. Compute the ancestor w' of w in $\mathscr{C}$ whose depth is $d_{\mathscr{C}}(u)+1$. If *apex* $(w')=u$, let $w''=w'$. Otherwise let $w''=p_{\mathscr{T}}(w')$.

Step 6. Return as $nca_{\mathscr{T}}(v, w)$ whichever of v'' and w'' has the larger value of *hp size.*

This method requires $O(n+m\alpha(m+n, n))$ time plus time to solve m *nca* problems on $\mathscr{C}$ and $2m$ depth problems on $\mathscr{C}$. We shall postpone a discussion of how to solve depth problems on $\mathscr{C}$, as this is the hardest part of the algorithm. To solve *nca* problems on $\mathscr{C}$, we use the method of § 5. That is, we represent the forest $\mathscr{C}$ of compressed trees by a forest $\mathscr{B}$ of balanced binary trees. If C is a tree in $\mathscr{C}$, $\mathscr{B}$ contains a tree B whose green vertices are the same as those of C. B is constructed exactly as in § 5; each time we create a new family in $\mathscr{C}$ we combine the corresponding trees in $\mathscr{B}$, using procedure *binarize* if the set of trees has size three or greater. For each vertex v, we maintain $size_{\mathscr{B}}(v)$. Constructing $\mathscr{B}$ and updating the size information during links takes $O(n)$ total time.

As in § 5, we solve the *nca* problem on $\mathscr{B}$ by solving the *nca* depth problem and the depth problem on $\mathscr{B}$. To solve the *nca* depth problem on $\mathscr{B}$ we embed $\mathscr{B}$ in a complete binary tree $\mathscr{B}'$ and use direct calculation as described in § 3; all we need to know for each vertex in $\mathcal{B}$ is its symmetric-order number and height (as a vertex in $\mathcal{B}'$). These numbers require updating as $\mathcal{B}$ is built. Let u be a vertex in $\mathcal{B}$ with left child v and right child w. When edges $v \rightarrow u$ and $w \rightarrow u$ are added to $\mathcal{B}$, we must update the numbers of the descendants of u as follows:

Let $h_1 = height_{\mathcal{B}'}(v)$, $h_2 = height_{\mathcal{B}'}(w)$, and $h = \max\{h_1, h_2\}$. Redefine

$$height_{\mathcal{B}'}(x) = \begin{cases} height_{\mathcal{B}'}(x) - h_1 + h & \text{if } x \text{ is a descendant of } v, \\ h+1 & \text{if } x = u, \\ height_{\mathcal{B}'}(x) - h_2 + h & \text{if } x \text{ is a descendant of } w; \end{cases}$$

$$sym_{\mathcal{B}'}(x) = \begin{cases} 2^{h-h_1}\, sym_{\mathcal{B}'}(x) & \text{if } x \text{ is a descendant of } v, \\ 2^{h+1} & \text{if } x = u, \\ 2^{h-h_2}\, sym_{\mathcal{B}'}(x) + 2^{h+1} & \text{if } x \text{ is a descendant of } w. \end{cases}$$

We can use the data structure of [11, § 3] to maintain the numbers implicitly. The total time to maintain the data structure during links and compute $O(m)$ numbers is $O(n + m\alpha(m+n, n))$. Given the numbers, solving an *nca* depth problem on $\mathcal{B}$ requires $O(1)$ time.

We come now to the last and hardest part of the method: solving depth problems on $\mathcal{B}$ and $\mathcal{C}$. Since we use the same method for both forests, we shall discuss only $\mathcal{C}$. To solve the depth problem on $\mathcal{C}$, we divide $\mathcal{C}$ into $O(\log^* n)$ plies. As in § 4, define $rank(v) = \lfloor \lg(size_{\mathcal{C}}(v)) \rfloor$. Define the function $f(i)$ by the recursion

$$f(1) = 8;$$
$$f(i) = 2^{f(i-1)/2} \text{ for any integer } i \geqq 2.$$

Define the ply i of vertex v to be the minimum value of i such that $f(i) > rank(v)$. It is easy to compute the ply of a vertex at the moment its size is determined; this takes $O(1)$ time per vertex for a total of $O(n)$ time. (We precompute a table that for any integer $x \in [0, \lfloor \lg n \rfloor]$ gives the minimum i such that $f(i) > x$.) The total number of plies is $O(\log^* n)$.

With each vertex v, we store not only its ply but several other values: the vertex $a(v)$ that is the shallowest ancestor of v in the same ply as v, a list of the children of v that are in the same ply as v, and an array, indexed by depth, of all the ancestors of v in the same ply as v. Each time we add a new family to $\mathcal{C}$, we update this information by performing a search from the root r of the new tree, using the lists of children to reach all descendants of r in the same ply as r. The total amount of time spent updating this information, and the total size of the ancestor arrays, is bounded by a constant times the following sum:

$$8n + \sum_{i=1}^{\infty} \frac{f(i+1)n}{2^{f(i)}} = 8n + \sum_{i=1}^{\infty} \frac{n}{2^{f(i)/2}} = O(n).$$

To solve depth problems on $\mathcal{C}$, we use the same method as in § 4.

Algorithm to compute the ancestor of vertex v in $\mathcal{C}$ whose depth is d. Repeat the following step until a vertex is returned:

General step. If $d \in [d_{\mathcal{C}}(a(v)), d_{\mathcal{C}}(v)]$, look up in v's ancestor array the ancestor whose depth is d. Otherwise, replace v by $p_{\mathcal{C}}(a(v))$.

This method requires $O(1)$ time per execution of the general step, or $O(\log^* n)$ time per depth problem. (Note that if we store with each vertex v the current size of v's ancestor array, we can compute $d_{\mathcal{C}}(a(v))$ and $d_{\mathcal{C}}(p_{\mathcal{C}}(a(v)))$ from $d_{\mathcal{C}}(v)$ in $O(1)$ time.)

Using this method on both $\mathscr{B}$ and $\mathscr{C}$, we obtain a total time bound of $O(n+m\log^* n)$ for the linking roots problem; solving the depth problem is the most time-consuming part of the algorithm.

7. A faster algorithm for the linking roots problem. In order to obtain a faster algorithm for the linking roots problem, we must improve our method for solving depth problems on $\mathscr{B}$ and $\mathscr{C}$. Our method in § 6 was based on the representation of plies used in § 4 for plies two and three. By including the method used in § 4 for ply one and defining the plies more dynamically, we obtain an $O(n+m\alpha(m+n, n))$-time method. As in § 6, we shall discuss the depth problem only for $\mathscr{C}$; we use the same method for $\mathscr{B}$. We shall assume that n, the number of vertices, and m, the number of *nca* queries, are known in advance. Working knowledge of the analysis of the disjoint set union algorithm [10], [13] will help the reader in what follows.

We divide $\mathscr{C}$ into three super-plies exactly as in § 4: super-ply one consists of vertices with rank less than $\lg^{(3)} n$, super-ply two consists of vertices with rank between $\lg^{(3)} n$ and $\lg^{(2)} n-1$ (inclusive), and super-ply three consists of vertices with rank $\lg^{(2)} n$ or greater. We maintain super-plies two and three exactly as we maintained the plies in § 6. The results of § 4 imply that updating super-plies two and three requires $O(n)$ total time.

We divide the trees of super-ply one into a number of subtrees whose definition is based on Ackermann's function. For integers $i, j \geqq 0$ define $A(i, j)$ by

$A(0, j) = j$ for $j \geqq 0$;
$A(1, j) = 2^j$ for $j \geqq 0$;
$A(i, 0) = A(i-1, 1)$ for $i \geqq 2$ (also valid for $i=1$);
$A(i, j) = A(i-1, A(i, j-1))$ for $i \geqq 2, j \geqq 1$.

Let $\alpha(m+n, n) = \min\{i \geqq 1 | A(i, \lfloor (m+n)/n \rfloor) > \lg n\}$.

Let v be a vertex that is in super-ply one and such that $p_{\mathscr{C}}(v)$ exists. We define the *level* of v to be the maximum value of $i \in [0, \alpha(m+n, n)]$ such that, for some j, *rank* $(v) < A(i, j) \leqq$ *rank* $(p_{\mathscr{C}}(v))$; we define the *position* of v to be the minimum such j. We can compute the level and position of a vertex when its parent is defined in $O(1)$ time by using a small precomputed table.

We maintain super-ply one as a collection of subtrees. Each vertex in super-ply one for which $p_{\mathscr{C}}(v)$ is undefined is in a single-vertex tree by itself. When a vertex v in ply one of positive rank has its parent defined, we build a new subtree S_v with root v. S_v includes all descendants of v with ranks in the range $[x, A(i, j)-1]$, where i is the level of v, j is the position of v, and x is defined as follows: if there is an integer $A(i', j')$ such that $i < i' \leqq \alpha(m+n, n)$, $j' \geqq 1$, and $A(i', j') < A(i, j)$ then x is the maximum such integer; if there is no such integer let x be the maximum integer less than the rank of v of the form $A(i', 0)$ for some $0 \leqq i' \leqq i$. We can compute x in $O(1)$ time by table look-up. The definitions of S_v and Ackermann's function guarantee that the vertex set of the new subtree S_v is the union of the vertex sets of one or more older subtrees; thus each vertex is only in one subtree at a time.

This partitioning into subtrees has the following effect. Let r be a shallowest vertex in super-ply one and let v be a leaf descendant of r. The subtrees partition the path from v to r as follows. Let l be the maximum level of any vertex along the path from v to r and let x be the last vertex along this path of level l. The subtrees partition the path from v to x into segments $(v=v_1=x_1)$, $(v_2, \cdots, x_2), \cdots, (v_l, \cdots, x_l)$, $(v_{l+1}, \cdots, x), (y_{l-1}, \cdots, w_{l-1}), (y_{l-2}, \cdots, w_{l-2}), \cdots, (y_0, w_0), (r)$, where x_i for $i \in [1, l]$ is the first vertex on the path of level i preceding all vertices of higher level and w_i for $i \in [0, l]$ is the last vertex on the path of level i succeeding all vertices of higher level. Some of the v_i's and w_i's may be undefined; the corresponding segments of the path are empty. It is also possible that $(v_l, \cdots, x_l) = (v_{l+1}, \cdots, x)$. When $p_{\mathscr{C}}(r)$ becomes defined, subtrees are combined to form a segment (y_i, r), where i is the level of r; if $i = l$, the new segment is $(v_{l+1}, \cdots, r)$, and if $i > l$, the new segment is $(v_l, \cdots, r)$.

To facilitate the construction of the subtrees, we store with each vertex in super-ply one a list of its children, in order by rank. Then we can determine the vertex set of S_v in $O(|S_v|)$ time by searching from v. If we represent each list of children as a list of buckets, each bucket containing the children of a given rank, then inserting a new child requires time proportional to its rank, and Lemma 8 implies that the total time for constructing these lists of children is $O(\sum_{i=0}^{\infty} (i+1)n/2^i) = O(n)$.

We represent the subtrees of super-ply one exactly as we represented the trees of ply one in § 4. We also store with each vertex in such a subtree a pointer to the root of the subtree. If S_v is one of these subtrees, constructing the representation of S_v requires $O(|S_v|)$ time.

We solve depth problems in $\mathscr{C}$ exactly as we solved them in § 6. The following argument shows that this method solves a single depth problem in $O(\alpha(m+n, n))$ time. Let v be a vertex, d a depth such that $d \leqq d_{\mathscr{C}}(v)$, and w the ancestor of v whose depth is d. With each iteration of the general step, v is replaced by an ancestor that is either in a new subtree of super-ply one or in a new super-ply, until eventually w is reached. Each execution of the general step requires $O(1)$ time. Thus it suffices to bound the number of subtrees in super-ply one encountered during this process. Let r be the shallowest ancestor of (the original) v in super-ply one, let l be the maximum level of any vertex along the path from v to r, and let x be the last vertex along this path of level l. Before x is reached, every iteration of the general step (except possibly the one that reaches x) causes the level of the current vertex to increase by at least one; after x is reached but before r is reached, every iteration of the general step causes the level of the current vertex to decrease by at least one. Thus there are $O(\alpha(m+n, n))$ iterations of the general step.

It remains for us to bound the total time spent constructing subtrees in super-ply one. For each vertex v, this time is $O(1)$ for each subtree in which v is placed. Our analysis is almost identical to the analysis of the disjoint set union algorithm [10], [13]. For $i \in [0, \alpha(m+n, n)+1]$, we define a multiple partition on ranks. The blocks of the level i partition (see Fig. 8) are given by

$$\textit{block}\ (i, j) = [A(i, j), A(i, j+1)-1] \text{ for } i \in [0, \alpha(m+n, n)], j \geqq 0;$$
$$\textit{block}\ (i, 0) = [0, \lfloor \lg^{(3)} n \rfloor] \text{ for } i = \alpha(m+n, n)+1.$$

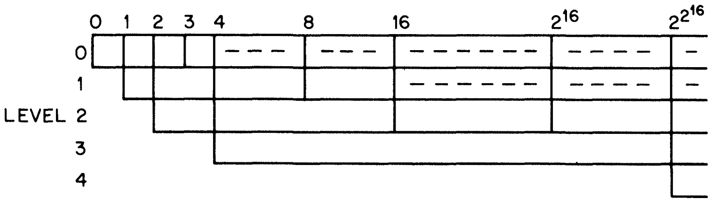

FIG. 8. *Multiple partition for analysis of subtree construction. Some block boundaries in levels zero and one are omitted.*

Note. For $i \in [0, \alpha(m+n, n)]$, the level i partition does not include ranks smaller than $A(i, 0)$. Thus not all ranks are in every partition. □

We define n_{ij} to be number of vertices with rank in *block* (i, j) and b_{ij} to be the number of level $i-1$ blocks whose intersection with *block* (i, j) is nonempty ($i \geqq 1$). As the algorithm proceeds, we define the *effective level* of a vertex v in a nontrivial subtree of super-ply one to be the minimum value of i such that *rank* (v) and *rank* $(p_{\mathscr{C}}(r(v)))$ are in the same block of the level i partition, where $r(v)$ is the root of the subtree containing v. As v is placed in larger-and-larger subtrees, its effective level increases from a minimum of one up to a maximum of $\alpha(m+n, n)+1$.

Consider a vertex v in *block* (i, j). We would like to bound the number of different subtrees that can contain v while v is at effective level i. First consider the case $i \leqq \alpha(m+n, n)$. When v is the first at level i, *rank* (v) and *rank* $(p_{\mathscr{C}}(r(v)))$ are in different level $i-1$ blocks. Subsequently, each time v is placed in a new subtree, *rank* $(p_{\mathscr{C}}(r(v)))$ moves to a new level $i-1$ block. Thus the number of subtrees that can contain v while v is at effective level i is at most $b_{ij}-1$. Second consider the case $i=\alpha(m+n, n)+1$. If $v \in$ *block* $(i', 0)$ for some $i' < \alpha(m+n, n)$, then v is only in one subtree while at effective level i. If $v \in$ *block* (i, j') for some $j' \geqq 0$, v while at effective level i is in at most as many subtrees as there are level-$\alpha(m+n, n)$ blocks, namely $\lfloor (m+n)/n \rfloor - 1$.

By Lemma 8, the number of vertices in *block*(i, j) is at most $2n/2^{A(i,j)}$. The following sum gives an upper bound, to within a constant factor, on the time spend building subtrees in super-ply one:

$$\sum_{i=1}^{\alpha(m+n,n)} \sum_{j=0}^{\infty} b_{ij} 2n/2^{A(i,j)} + n\lfloor (m+n)/n \rfloor .$$

Now we must estimate b_{ij}. For $i=1$, $j \geqq 0$, $A(i, j+1)-A(i, j)=2^j=A(i, j)$, which means $b_{ij} < A(i, j)$. For $i \geqq 2$, $j \geqq 0$, $A(i, j+1)=A(i-1, A(i, j))$, which means $b_{ij} < A(i, j)$ in this case also. Thus the time spent building subtrees in super-ply one is at most a constant times the following estimate:

$$\sum_{i=1}^{\alpha(m+n,n)} \sum_{j=0}^{\infty} 2nA(i, j)/2^{A(i,j)} + O(m+n) \leqq \sum_{i=1}^{\alpha(m+n,n)} O(nA(i, 0)/2^{A(i,0)}) + O(m+n)$$
$$= O(m+n).$$

Thus the entire algorithm solves the linking roots problem in $O(n+m\alpha(m+n, n))$ time.

This algorithm has the disadvantage that both m and n must be known ahead of time. We leave to the reader the easy exercise of modifying the algorithm to avoid this. (The idea is to re-estimate m and n each time the actual value of either grows by a factor of two.)

Appendix. Tree terminology. Throughout this paper we consider only rooted trees. A rooted tree T consists of a *vertex set* V, a *root* $r \in V$, and a mapping $p(v)$: $V-\{r\} \rightarrow V$ such that, for every vertex v, there is an integer $i \geqq 0$ such that $p^i(v)=r$, where $p^i(v)$ is defined inductively by $p^0(v)=0$, $p^{i+1}(v)=p(p^i(v))$. For $v \in V-\{r\}$, $p(v)$ is the *parent* of v; v is a *child* of $p(v)$. The *edges* of T are the ordered vertex pairs $v \rightarrow p_T(v)$ for $v \in V-\{r\}$. A *leaf* is a vertex with no children. If $p^i(v)=w$ for some $i \geqq 0$, v is a *descendant* of w and w is an *ancestor* of v. Two vertices are *unrelated* if neither is an ancestor of the other. A *path* in T is a sequence of vertices $v, p(v)$, $p^2(v), \cdots, p^k(v)$; the *length* of this path is k. The *depth* of a vertex v is the length of the path from v to r; the *height* of v is the length of the longest path from a leaf to v. The depth (and height) of T is the length of the longest path in T. We use $|T|$ to denote the number of vertices in T. The *nearest common ancestor* $nca(x,y)$ of two vertices x and y is the vertex of greatest depth that is an ancestor of both x and y.

A *forest* $\mathcal{T}$ is a collection of vertex-disjoint trees. (We use capital italic letters to denote trees and capital script letters to denote forests.) When discussing parameters associated with several trees or forests, we use the names of the trees or forests as subscripts to distinguish the parameters. For example, $p_T(v)$ is the parent of v in T.

Acknowledgment. The first author's research was done partly while he was a graduate student at the University of California, Irvine. He would like to thank his thesis advisor, George Lueker, for his patience, encouragement, and numerous conversations regarding preliminary versions of the results presented here. Both authors thank the referees for their exceptionally thorough job in catching and correcting a number of minor errors in the original manuscript.

REFERENCES

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

[2] ——, *On finding lowest common ancestors in trees*, this Journal, 5 (1976), pp. 115–132.

[3] M. L. FREDMAN, *Two applications of a probabilistic search technique: sorting $X+Y$ and building balanced search trees*, Proc. Seventh ACM Symposiun on Theory of Computing, 1975, pp. 240–244.

[4] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comp. Sys. Sci., to appear; also Proc. Fifteenth ACM Symposium on Theory of Computing, 1983, pp. 246–251.

[5] D. HAREL, *A linear time time algorithm for the lowest common ancestors problem*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 308–319.

[6] D. E. KNUTH, *The Art of Computer Programming, Volume* I: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[7] D. MAIER, *An efficient method for storing ancestor information in trees*, this Journal, 8 (1979), pp. 599–618.

[8] A. SCHÖNHAGE, *Storage modification machines*, this Journal, 9 (1980), pp. 490–508.

[9] D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

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

[11] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

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

[13] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., to appear.

[14] J. VAN LEEUWEN, *Finding lowest common ancestors in less than logarithmic time*, unpublished report, 1976.