SIAM J. COMPUT.
Vol. 38, No. 4, pp. 1533–1573

LINEAR-TIME ALGORITHMS FOR DOMINATORS AND OTHER PATH-EVALUATION PROBLEMS*

ADAM L. BUCHSBAUM†, LOUKAS GEORGIADIS‡, HAIM KAPLAN§, ANNE ROGERS¶, ROBERT E. TARJAN‖, AND JEFFERY R. WESTBROOK**

Abstract. We present linear-time algorithms for the classic problem of finding dominators in a flowgraph, and for several other problems whose solutions require evaluating a function defined on paths in a tree. Although all these problems had linear-time solutions previously, our algorithms are simpler, in some cases substantially. Our improvements come from three new ideas: a refined analysis of path compression that gives a linear bound if the compressions favor certain nodes; replacement of random-access table look-up by a radix sort; and a more careful partitioning of a tree into easily managed parts. In addition to finding dominators, our algorithms find nearest common ancestors off-line, verify and construct minimum spanning trees, do interval analysis of a flowgraph, and build the component tree of a weighted tree. Our algorithms do not require the power of a random-access machine; they run in linear time on a pointer machine. The genesis of our work was the discovery of a subtle error in the analysis of a previous allegedly linear-time algorithm for finding dominators. That algorithm was an attempt to simplify a more complicated algorithm, which itself was intended to correct errors in a yet earlier algorithm. Our work provides a systematic study of the subtleties in the dominators problem, the techniques needed to solve it in linear time, and the range of application of the resulting methods. We have tried to make our techniques as simple and as general as possible and to understand exactly how earlier approaches to the dominators problem were either incorrect or overly complicated.

Key words. dominators, flowgraphs, pointer machine, random-access machine, set union, path compression, nearest common ancestors, minimum spanning trees, interval analysis, component tree, data structures, analysis of algorithms

AMS subject classifications. 05C85, 68N20, 68P05, 68Q05, 68Q25, 68W05

DOI. 10.1137/070693217

1. Introduction.

The story of this paper begins with the publication in 1979 [43] of an almost-linear-time algorithm for the problem of finding immediate domina-

*Received by the editors May 30, 2007; accepted for publication (in revised form) July 18, 2008; published electronically November 12, 2008. This work is partially covered by the extended abstracts *Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators*, in Proceedings of the 30th ACM Symposium on Theory of Computing, 1998, pp. 279–288, and *Finding dominators revisited*, in Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms, 2004, pp. 862–871. The work of the second and fifth authors was partially done at Princeton University and partially supported by the National Science Foundation under the Aladdin Project, grant CCR-0122581. The work of the fifth author was also partially supported by NSF grants CCF-0830676 and CCF-0832797. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.
http://www.siam.org/journals/sicomp/38-4/69321.html

†Madison, NJ (alb@adambuchsbaum.com). This author's was work done while a member of AT&T Labs.

‡HP Labs, 1501 Page Mill Rd, Palo Alto, CA 94304. Current address: Informatics and Telecommunications Engineering Department, University of Western Macedonia, Kozani, Greece (lgeorg@uowm.gr).

§School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel (haimk@math.tau.ac.il). This author's work was partially done while a member of AT&T Labs.

¶Department of Computer Science, University of Chicago, 1100 E. 58th Street, Chicago, IL 60637 (amr@cs.uchicago.edu). This author's work was partially done while a member of AT&T Labs.

‖Department of Computer Science, Princeton University, Princeton NJ 08544, and HP Labs, 1501 Page Mill Rd., Palo Alto, CA 94304 (ret@cs.princeton.edu).

**Los Angeles, CA (jwestbrook@acm.org). This author's work was partially done while a member of AT&T Labs.

tors in a flowgraph. Computing dominators is a fundamental problem in the theory of directed graphs and is a crucial first step in global optimizations of computer code. Although versions of the 1979 dominators algorithm are very fast in practice [34] and are widely used, the question remained of whether dominators could be computed in linear time, in either the random-access-machine (RAM) model or the less powerful pointer-machine model. Harel [37] claimed to have a linear-time RAM algorithm, but Alstrup et al. [10] found problems in his work. They developed a correct but very complicated algorithm that uses powerful bit-manipulation techniques. Buchsbaum et al. [16] claimed to present a simpler linear-time algorithm, but Georgiadis and Tarjan [32] found a flaw in their analysis, which, when corrected, results in a nonlinear-time bound. They also presented a way to repair and modify the algorithm so that it does indeed run in linear time, on a pointer machine. Buchsbaum et al. [16, Corrig.] gave a different fix for the RAM model.

Our paper addresses the question of exactly what techniques are needed to compute dominators in linear time and explores the range of application of these techniques. We clarify and extend the conference papers [15, 32] to provide not only a linear-time algorithm for finding dominators but linear-time algorithms for a variety of related problems as well. We avoid the use of bit-manipulation techniques, so that all our algorithms can run on a pointer machine. We describe the techniques needed (in different combinations) to obtain these results as well as the key difficulties in the dominators problem.

2. Overview. We study six problems—off-line computation of nearest common ancestors (NCAs), verification and construction of minimum spanning trees (MSTs), interval analysis of flowgraphs, finding dominators in flowgraphs, and building the component tree of a weighted tree—that directly or indirectly require the evaluation of a function defined on paths in a tree. Each of these problems has a linear-time algorithm on a RAM. Some of these algorithms are quite complicated, and the fastest pointer-machine algorithms are slower by an inverse-Ackermann-function factor.[1] (See Table 2.1.)

TABLE 2.1

Time bounds, where n is the number of vertices, m is either the number of edges/arcs for graph problems or the number of NCA queries for the NCA problem, and $\alpha(m, n)$ is the standard functional inverse of the Ackermann function.

Problem	Previous pointer-machine bound		Previous RAM bound	
Off-line NCAs	$O(m\alpha(m,n)+n)$	[3]	$O(n+m)$	[38, 52]
MST verification	$O(m\alpha(m,n)+n)$	[58]	$O(n+m)$	[22, 40]
MST construction	$O(m\alpha(m,n)+n)$	[18]	$O(n+m)$	[26, 39]
Interval analysis	$O(m\alpha(m,n)+n)$	[57]	$O(n+m)$	[29, 57]
Dominators	$O(m\alpha(m,n)+n)$	[43]	$O(n+m)$	[10, 16]
Component trees	$O(m\alpha(m,n)+n)$		$O(n+m)$	[62]

A pointer machine [59] allows binary comparisons and arithmetic operations on data, dereferencing of pointers, and equality tests on pointers. It does not permit pointer arithmetic or tests on pointers other than testing for equality and is thus less powerful than the RAM model [2]. Pointer machines are powerful enough to simulate

[1] We use Tarjan's definition [56]. Let $A(i, j)$ for $i, j \geq 1$ be defined by $A(1, j) = 2^j$ for $j \geq 1$; $A(i, 1) = A(i-1, 2)$ for $i \geq 2$; and $A(i, j) = A(i-1, A(i, j-1))$ for $i, j \geq 2$. Then $\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) > \log n\}$.

functional programming languages like LISP and ML. Pointer machine algorithms can be simpler than RAM algorithms in that they avoid the use of table look-up and more complicated bit-manipulation techniques. Thus pointer machine algorithms may have a conceptual, if not a practical, advantage over RAM algorithms.

We develop linear-time algorithms for all six problems that are simpler than the previous algorithms, in some cases substantially. We extract the commonalities among the problems and develop a set of techniques that in various combinations give our efficient algorithms. We do not use table look-up or any bit-manipulation techniques, so all our algorithms can run on a pointer machine.

Our improvements come mainly from three new ideas. The first is a refined analysis of path compression. Path compression is a well-known technique first used to speed up the standard disjoint-set-union (DSU) data structure [56] and later extended to speed up the evaluation of functions defined on paths in trees [58]. Our applications use either the DSU structure or path evaluation for the function *minimum* or *maximum*, or both. We show that, under a certain restriction on the compressions, which is satisfied by our applications, compression takes constant rather than inverse-Ackermann amortized time.

The second new idea is to replace the table-based methods of the RAM algorithms with radix sorting. Each of the RAM algorithms precomputes answers to small subproblems, stores the answers in a table, and looks up the answers by random access. If the size of the subproblems is small enough, the total size of all distinct subproblems and the total time to solve them are linear (or even sublinear) in the size of the original problem. Our alternative approach is to construct an encoding of each subproblem, group isomorphic subproblems together using a radix sort, solve one instance of each group of isomorphic subproblems, and transfer its solution to the isomorphic subproblems.

The third new idea is to change the partitioning strategy. In order to reduce the original problem to a collection of small subproblems, the RAM algorithms partition a tree corresponding to the original problem into small subtrees. For some of the problems, partitioning the entire tree into subtrees produces serious technical complications, especially for computing dominators. Instead, for all but one of the problems we partition only the bottom part of the tree into small subtrees. For NCAs and MSTs, this technique together with our refined analysis of path compression suffices to yield a linear-time algorithm. For interval analysis and finding dominators, we also partition the remainder of the tree into a set of maximal disjoint paths. Only one of our applications, building a component tree, relies on the original idea of partitioning the entire tree into small subtrees.

The remainder of our paper proceeds as follows. Section 3 formally defines the problems we consider and reviews previous work. Section 4 discusses DSU and computing path minima on trees, and gives a refined analysis of path compression. Section 5 discusses the use of radix sorting to solve a graph problem for a collection of many small instances. Sections 6 through 10 discuss our applications: NCAs, MSTs, flowgraph interval analysis, finding dominators, and building a component tree, respectively. Section 11 contains concluding remarks. Our paper is a significantly revised and improved combination of two conference papers [15, 32], including new results in sections 8 and 10.

3. Problem definitions and previous work. Throughout this paper we denote the base-two logarithm by log. We assume $n \geq 2$ throughout.

3.1. Nearest common ancestors.

PROBLEM 3.1 (off-line nearest common ancestors). *Given an n-node tree T rooted at node r and a set P of m node pairs, find, for each pair $\{v, w\}$ in P, the nearest common ancestor of v and w in T, denoted by $nca(v, w)$.*

The fastest previous pointer-machine algorithm is that of Aho, Hopcroft, and Ullman (AHU) [3], which runs in $O(n + m\alpha(m + n, n))$ time. The AHU algorithm uses a DSU data structure; it runs in $O(n + m)$ time on a RAM if this structure is implemented using the DSU algorithm of Gabow and Tarjan [29] for the special case in which the set of unions is known in advance. The first linear-time RAM algorithm was actually given by Harel and Tarjan [38]. Other linear-time RAM algorithms were given by Schieber and Vishkin [52], Bender and Farach-Colton [12], and Alstrup et al. [9].

There are several variants of the NCAs problem of increasing difficulty. For each but the last, there is a nonconstant-factor gap between the running time of the fastest RAM and pointer-machine algorithms.

- *Static on-line.* T is given a priori but P is given on-line: each NCA query must be answered before the next one is known.
- *Linking roots.* T is given dynamically. Specifically, T is initially a forest of singleton nodes. Interspersed with the on-line NCA queries are on-line $link(v, w)$ operations, each of which is given a pair of distinct roots v and w in the current forest and connects them by making v the parent of w.
- *Adding leaves.* Like linking roots, only v is any node other than w and w is a singleton.
- *General linking.* Like linking roots, only v can be any node that is not a descendant of w.
- *Linking and cutting.* Like general linking, but with additional interspersed $cut(v)$ operations, each of which is given a nonroot node and makes it a root by disconnecting it from its parent.

Harel and Tarjan [38] showed that the static on-line problem (and thus each more general variant) takes $\Omega(\log \log n)$ time on a pointer machine for each query, in the worst case. Alstrup and Thorup [7] gave a matching $O(n + m \log \log n)$-time pointer-machine algorithm for general linking, which is also optimal for the static on-line, linking roots, and adding leaves variants. Earlier, Tsakalidis and van Leeuwen [63] gave such an algorithm for the static on-line variant, and a modified version of van Leeuwen's earlier algorithm [64] has the same bound for linking roots. The fastest known pointer-machine algorithm for linking and cutting is the $O(n + m \log n)$-time algorithm of Sleator and Tarjan [53]; Harel and Tarjan [38] conjectured that this is asymptotically optimal, which in the cell-probe model follows from a result of Pătraşcu and Demaine [47]. On a RAM, the fastest known algorithms take $\Theta(n + m)$ time for the static on-line [38, 52] and adding leaves [27] variants, $O(n + m\alpha(m + n, n))$ time for linking roots [38] and general linking [27], and $O(n + m \log n)$ time for linking and cutting [53]. All these algorithms use $O(n + m)$ space. For a more thorough survey of previous work see Alstrup et al. [9].

3.2. Verification and construction of MSTs.

PROBLEM 3.2 (MST construction). *Given an undirected, connected graph $G = (V, E)$ whose edges have real-valued weights, find a spanning tree of minimum total edge weight (an MST) of G.*

PROBLEM 3.3 (MST verification). *Given an undirected, connected graph $G = (V, E)$ whose edges have real-valued weights and a spanning tree T of G, determine whether T is an MST of G.*

In both problems, we denote by n and m the numbers of vertices and edges, respectively. Since G is connected and $n \geq 2$, $m \geq n-1$ implies $n = O(m)$.

MST construction has perhaps the longest and richest history of any network optimization problem; Graham and Hell [35] and Chazelle [18] provide excellent surveys. A sequence of faster and faster algorithms culminated in the randomized linear-time algorithm of Karger, Klein, and Tarjan [39]. This algorithm requires a RAM, but only for a computation equivalent to MST verification. It is also *comparison-based*: the only operations it does on edge weights are binary comparisons. Previously, Fredman and Willard [26] developed a linear-time RAM algorithm that is not comparison-based. Subsequently, Chazelle [18] developed a deterministic, comparison-based $O(m\alpha(m,n))$-time pointer-machine algorithm, and Pettie and Ramachandran [49] developed a deterministic, comparison-based pointer-machine algorithm that runs in minimum time to within a constant factor. Getting an asymptotically tight bound on the running time of this algorithm remains an open problem.

Although it remains open whether there is a comparison-based, deterministic linear-time MST construction algorithm, even for a RAM, such algorithms do exist for MST verification. Tarjan [58] gave a comparison-based, deterministic $O(m\alpha(m,n))$-time pointer-machine algorithm for verification. Komlós [41] showed how to do MST verification in $O(m)$ comparisons, without providing an efficient way to determine which comparisons to do. Dixon, Rauch, and Tarjan [22] combined Tarjan's algorithm, Komlós's bound, and the tree partitioning technique of Gabow and Tarjan [29] to produce a comparison-based, deterministic linear-time RAM algorithm. King later gave a simplified algorithm [40].

3.3. Interval analysis of flowgraphs. A *flowgraph* $G = (V, E, r)$ is a directed graph with a distinguished *root vertex* r such that every vertex is reachable from r. A *depth-first spanning tree* D of G is a spanning tree rooted at r defined by some depth-first search (DFS) of G, with the vertices numbered from 1 to n in preorder with respect to the DFS (the order in which the search first visits them). We identify vertices by their preorder number. We denote by n and m the number of vertices and arcs of G, respectively.

PROBLEM 3.4 (interval analysis). *Given a flowgraph G and a depth-first spanning tree D of G, compute, for each vertex v, its* head $h(v)$*, defined by*

$$h(v) = \max\{u : u \textit{ is a proper ancestor of } v \textit{ in } D \textit{ and there is a path from } v \textit{ to } u \textit{ in } G \textit{ containing only descendants of } u\},$$
$$\textit{or null if this set is empty.}$$

The heads define a forest called the *interval forest* H, in which the parent of a vertex is its head. If v is any vertex, the descendants of v in H induce a strongly connected subgraph of G, which is called an *interval*; these intervals impose a hierarchy on the loop structure of G. Interval analysis has been used in global flow analysis of computer programs [4], in testing flowgraph reducibility [60], and in the construction of two maximally edge-disjoint spanning trees of a flowgraph [57]. Tarjan [57] gave an $O(m\alpha(m,n))$-time pointer-machine algorithm for interval analysis using DSU. The Gabow–Tarjan DSU algorithm [29] reduces the running time of this algorithm to $O(m)$ on a RAM.

3.4. Finding dominators. Let $G = (V, E, r)$ be a flowgraph. We denote by n and m the number of vertices and arcs of G, respectively. Vertex v *dominates* vertex

w if every path from r to w contains v, and v is the *immediate dominator* of w if every vertex that dominates w also dominates v. The dominators define a tree rooted at r, the *dominator tree* T, such that v dominates w if and only if v is an ancestor of w in T: for any vertex $v \neq r$, the immediate dominator of v is its parent in T.

PROBLEM 3.5 (finding dominators). *Given a flowgraph $G = (V, E, r)$, compute the immediate dominator of every vertex other than r.*

Finding dominators in flowgraphs is an elegant problem in graph theory with fundamental applications in global flow analysis and program optimization [1, 19, 24, 45] and additional applications in VLSI design [11], theoretical biology [5, 6], and constraint programming [51]. Lengauer and Tarjan [43] gave a practical $O(m\alpha(m, n))$-time pointer-machine algorithm, capping a sequence of previous improvements [1, 45, 50, 55]. Harel [37] claimed a linear-time RAM algorithm, but Alstrup et al. [10] found problems with some of his arguments and developed a corrected algorithm, which uses powerful bit-manipulation-based data structures. Buchsbaum et al. [16] proposed a simpler algorithm, but Georgiadis and Tarjan [32] gave a counterexample to their linear-time analysis and presented a way to repair and modify the algorithm so that it runs in linear time on a pointer machine; Buchsbaum et al. [16, Corrig.] gave a different resolution that results in a linear-time algorithm for a RAM.

3.5. Building a component tree. Let T be a tree and let L be a list of the edges of T. The *Kruskal tree* of T with respect to L is a tree representing the connected components formed by deleting the edges of T and adding them back one at a time in the order of their occurrence in L. Specifically, K contains $2n-1$ nodes. Its leaves are the nodes of T. Each internal node is a component formed by adding an edge (v, w) back to T; its children are the two components that combine to form it.

PROBLEM 3.6 (component-tree construction). *Given an n-node tree T and a list L of its edges, build the corresponding Kruskal tree.*

Compressed component trees (formed by adding edges a group at a time rather than one at a time) have been used in shortest-path algorithms [48, 62]. It is straightforward to build a component tree or a compressed component tree in $O(n\alpha(n, n))$ time on a pointer machine using DSU. The Gabow–Tarjan DSU algorithm [29] improves this algorithm to $O(n)$ time on a RAM, as described by Thorup [62].

4. Path compression on balanced trees.

4.1. DSU via path compression and balanced unions. The *disjoint set union* (DSU) problem calls for the maintenance of a dynamic partition of a universe U, initially consisting of singleton sets. Each set has a unique *designated element*; the designated element of a singleton set is its only element. Two operations are allowed:

- *unite*(v, w): Form the union of the sets whose designated elements are v and w, with v being the designated element of the new set.
- *find*(v): Return the designated element of the set containing element v.

There are alternative, equivalent formulations of the DSU problem. In one [59, 56], each set is accessed by a label, rather than by a designated element. In another [61], sets have labels but can be accessed by *any* element. In yet another [61], each set is accessed by a *canonical element*, which in the case of a *unite*(v, w) operation can be freely chosen by the implementation to be either v or w. Our formulation more closely matches our uses. We denote by n the total number of elements and by m the total number of finds.

The standard solution to the DSU problem [56, 61] represents the sets by rooted trees in a forest. Each tree represents a set, whose elements are the nodes of the tree.

Each node has a pointer to its parent and a bit indicating whether it is a root; the root points to the designated element of the set. To provide constant-time access to the root from the designated node, the latter is either the root itself or a child of the root. With this representation, to perform *unite*(v, w), find the roots of the trees containing v and w, link them together by making one root the parent of the other, and make v a child of the new root if it is not that root or a child of that root already. To perform *find*(v), follow parent pointers until reaching a root, reach the designated element of the set in at most one more step, and return this element. A unite operation takes $O(1)$ time. A find takes time proportional to the number of nodes on the find path. A sequence of intermixed unite and find operations thus takes $O(n + s)$ time, where s is the total number of nodes on find paths.

One way to reduce s is to use *path compression*: after a find, make the root the parent of every other node on the find path. Another way to reduce s is to do *balanced unions*. There are two well-known balanced-union rules. In the first, *union-by-size*, each root stores the number of its descendants. To perform *unite*(v, w), make the root of the larger tree the parent of the root of the smaller, making either the parent of the other in case of a tie. In the second, *union-by-rank*, each root has a nonnegative integer *rank*, initially zero. To perform *unite*(v, w), make the root of higher rank the parent of the root of lower rank; in case of a tie, make either root the parent of the other and add one to the rank of the remaining root. Both of these union rules produce *balanced* trees. More specifically, let F be the forest built by doing all the unite operations and none of the finds. We call F the *reference forest*. F is *balanced* or, more precisely, *c-balanced* if for a constant $c > 1$ the number of nodes of height h in F is $O(n/c^h)$ for every h. Both union-by-size and union-by-rank produce 2-balanced forests. Furthermore, since only roots must maintain sizes or ranks, these fields obviate the need for separate bits to indicate which nodes are roots.

For any sequence of unions and finds such that the unions build a balanced forest and the finds use path compression, the total running time is $O(n+m\alpha(m+n, n))$: the analysis of path compression by Tarjan and van Leeuwen [61] applies if the reference forest is balanced. We seek a linear time bound, which we can obtain for sequences of finds that are suitably restricted. Before obtaining this bound, we discuss a more general use of path compression and balanced union: to find minima on paths in dynamic trees.

4.2. Finding minima on paths. The *dynamic path-minimum problem* calls for the maintenance of a forest of rooted trees, each initially a one-node tree, whose arcs, which are directed from parent to child, have real values. The trees are subject to three operations:

- *link*(v, w, x): Nodes v and w are the roots of different trees in F, and x is a real number. Make v the parent of w by adding arc (v, w) to F, with value x.
- *findroot*(v): Return the root of the tree in F containing the node v.
- *eval*(v): Return the minimum value of an arc on the path to v from the root of the tree containing it.

We shall denote by n the total number of nodes and by m the total number of *findroot* and *eval* operations. Variants of this problem include omitting the *findroot* operation, replacing minimum by maximum, and requiring the *eval* operation to return an arc of minimum value rather than just the minimum value. The two solutions to be described are easily modified to handle these variants. We call a data structure that solves the dynamic path-minimum problem a *link-eval structure*.

Tarjan [58] considered this problem and developed two data structures to solve it: a simple one [58, sec. 2], which uses path compression on the forest defined by the links, and a sophisticated one [58, sec. 5], which uses path compression on a balanced forest related to the one defined by the links. Tarjan's simple link-eval structure uses a compressed version of F, represented by parent pointers, with the nodes rather than the arcs storing values. Each root has value infinity. Perform *link*(v, w, x) by making v the parent of w and giving w the value x. Perform *findroot*(v) by following parent pointers from v to the root of the tree containing it, compressing this path, and returning the root. Perform *eval*(v) by following parent pointers from v to the root of the tree containing it, compressing this path, and returning the value of v. To compress a path $v_0, v_1, \ldots, v_k$ with v_i the parent of v_{i+1} for $0 \leq i < k$, repeat the following step for each i from 2 through k: replace the parent of v_i by v_0, and replace the value of v_i by the value of v_{i-1} if the latter is smaller. Compression preserves the results of *findroot* and *eval* operations while making tree paths shorter.

If the final forest F is balanced, then this simple link-eval structure takes $O(n + m\alpha(m + n, n))$ time to perform a sequence of operations [58]: the effect of a compression on the structure of a tree is the same whether the compression is due to a *findroot* or an *eval*. In our MST application the final forest is actually balanced. Our application to finding dominators requires Tarjan's sophisticated link-eval structure.

4.3. Delayed linking with balancing. Tarjan's sophisticated structure delays the effect of some of the links so that they can be done in a way that makes the resulting forest balanced. Since our analysis requires some knowledge of the inner workings of this structure, we describe it here. We streamline the structure slightly, and we add to it the ability to do *findroot* operations, which were not supported by the original. We also describe (in section 4.4) a variant that uses linking-by-rank; the original uses linking-by-size.

We represent the forest F defined by the link operations by a *shadow forest* R. Each tree in F corresponds to a tree in R with the same vertices and the same root. Each tree T in R is partitioned into one or more subtrees $S_0, S_1, \ldots, S_k$, such that the root of S_i is the parent of the root of S_{i+1} for $0 \leq i < k$, and the root of S_0 is the root of T. We call the roots of the subtrees $S_0, S_1, \ldots, S_k$ (including the root of S_0) *subroots*. We represent R by a set of parent pointers that are defined for nodes that are not subroots and, for each subroot, a pointer to its child that is a subroot, if any. (Each subroot has a null parent pointer; the deepest subroot has a null child pointer.) Since parents are needed only for nodes that are not subroots and child pointers are required only for subroots, we can use a single pointer per node to store both kinds of pointers, if we mark each node to indicate whether it is a subroot. We shall use *shp*(v) to denote the parent of v in its subtree and *shc*(v) to denote the child of v that is a subroot, if there is one; *shp*$(v) =$ *null* if v is a subroot, and *shc*$(v) =$ *null* if v is a subroot without a child that is a subroot.

With each node v we store a value $b(v)$. We manipulate the trees of R and the node values to preserve two related invariants:

(i) *eval*$(v) = \min\{b(u) : u$ is an ancestor in R of v in the same subtree$\}$;

(ii) $b($*shc*$(v)) \leq b(v)$ if *shc*$(v) \neq$ *null*.

To help keep evaluation paths short, we use both path compression and a variant of union-by-size. We denote by *size*(v) the number of descendants of v in R and by *subsize*(v) the number of descendants of v in the same subtree as v. For convenience, we let *size*(*null*) $= 0$. Then *subsize*$(v) =$ *size*(v) if v is not a subroot, and *subsize*$(v) =$

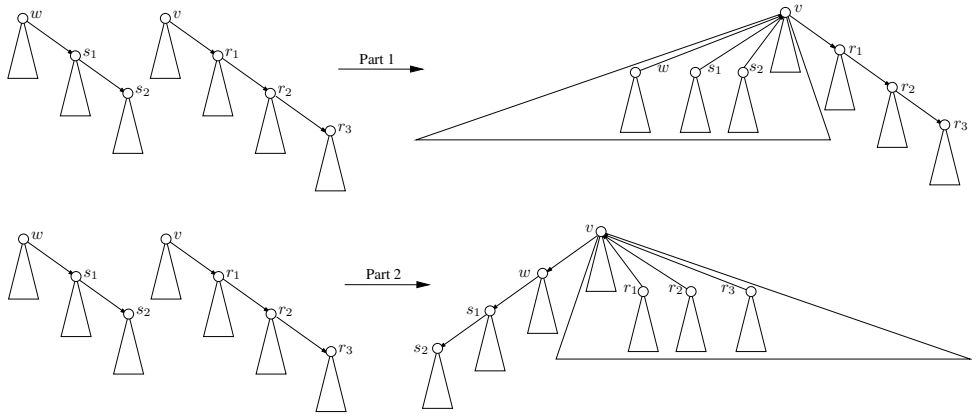

FIG. 4.1. *Linking by size: Part* 1, $size(v) \geq size(w)$, *and Part* 2, $size(v) < size(w)$.

$size(v) - size(shc(v))$ if v is a subroot. We maintain sizes but only for subroots, which allows us to compute the subsize of a subroot in constant time.

To initialize the structure, make each node v a singleton tree ($shp(v) = shc(v) = null$), with $b(v) = \infty$ and $size(v) = 1$. To perform $eval(v)$, return $b(v)$ if $shp(v) = null$; otherwise, compress the path to v from the subroot of the subtree containing it (exactly as in the simple link-eval structure of section 4.2), and then return $\min\{b(v), b(shp(v))\}$. Perform $link(v, w, x)$ as follows. First, set $b(w)$ (previously infinity) equal to x. Next, if $size(v) \geq size(w)$, perform Part 1 below; otherwise, perform Part 2 below and, if necessary, Part 3. (See Figures 4.1 and 4.2.)

Part 1 ($size(v) \geq size(w)$). Combine the subtree rooted at v with all the subtrees in the tree rooted at w by setting $shp(u) = v$ and $b(u) = \min\{b(u), x\}$ for each subroot u of a subtree in the tree rooted at w. Find such subroots by following shc pointers from w. (In Figure 4.1 (Part 1), the successive values of u are w, s_1, s_2.) This step effects a compression to v from the deepest subroot descendant of w. The updates to the b-values maintain (i) and (ii).

Part 2 ($size(v) < size(w)$). Combine all the subtrees in the tree rooted at v by setting $shp(u) = v$ for each subroot $u \neq v$ of a subtree in the tree rooted at v. (In Figure 4.1 (Part 2), the successive values of u are r_1, r_2, r_3.) This step effects a compression to v from the deepest subroot descendant of v. Then set $shc(v) = w$. This may cause violations of invariants (i) and (ii).

Part 3. In order to restore (i) and (ii) after Part 2, repeat the following step until it no longer applies. Let $s_0 = shc(v)$ and $s_1 = shc(s_0)$. (In the first iteration, $s_0 = w$.) If $s_1 \neq null$ and $x < b(s_1)$, compare the subsizes of s_0 and s_1. If the former is not smaller, combine the subtrees with subroots s_0 and s_1, making s_0 the new subroot, by simultaneously setting $shp(s_1) = s_0$ and $shc(s_0) = shc(s_1)$. If the former is smaller, combine the subtrees with subroots s_0 and s_1, making s_1 the new subroot, by simultaneously setting $shp(s_0) = s_1$, $shc(v) = s_1$, $b(s_1) = x$, and $size(s_1) = size(s_0)$. Once this step no longer applies, (i) and (ii) are restored.

Complete the linking by setting $size(v) = size(v) + size(w)$. We call this linking method *linking-by-size*.

The method must keep track of which nodes are subroots. Nodes that are not subroots can be marked as such by, e.g., setting their sizes to zero, since sizes are maintained only for subroots. We have omitted this updating from Parts 1, 2, and 3.

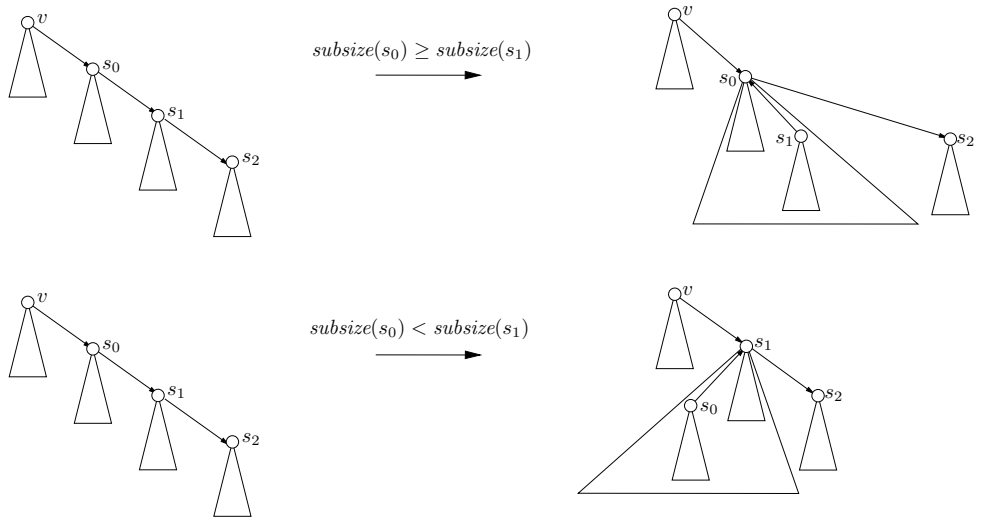

FIG. 4.2. *Linking by size: Part* 3.

This version of the data structure differs from the original [58] only in the placement of Part 3 of the link operation. In the original, Part 3 is done before Parts 1 and 2 to restore (i) and (ii), whether or not $size(v) \geq size(w)$. Delaying Part 3 allows it to be avoided entirely if $size(v) \geq size(w)$; in this case Part 1 alone suffices to restore (i) and (ii).

This structure does not support *findroot* (because an *eval* operation reaches only a subroot, not a root), but we can easily extend it to do so. To each subroot that is not a root, we add a pointer to its deepest subroot descendant; to each deepest subroot, we add a pointer to the root of its tree. Then a root is reachable from any subroot descendant in at most two steps. To perform *findroot*(v), compress the path to v from the subroot of its subtree; then follow at most two pointers to reach a root, and return this root. Operation *findroot* has the same asymptotic complexity as *eval*. The running time of a link operation increases by at most a constant factor because of the extra pointer updates needed.

In the sophisticated link-eval structure, path compression is performed on the subtrees, not on the trees. The next lemma implies that these subtrees are balanced.

LEMMA 4.1. *Consider a shadow forest built using linking-by-size. If u is a tree node such that $shp(u)$ and $shp(shp(u))$ are both nonnull, then $subsize(shp(shp(u))) \geq 2 \cdot subsize(u)$.*

Proof. A node u can be assigned a parent $shp(u)$ in Part 1, 2, or 3 of a link operation. If this occurs in Part 3, $subsize(shp(u)) \geq 2 \cdot subsize(u)$ after u gets its parent. Once this happens, $subsize(u)$ stays the same and $subsize(shp(u))$ can only increase. Thus when $shp(u)$ gets a parent, $subsize(shp(shp(u))) \geq subsize(shp(u)) \geq 2 \cdot subsize(u)$, and this inequality persists. Regardless of when u gets a parent $shp(u)$, if $shp(u)$ gets its parent in Part 3, then $subsize(shp(shp(u))) \geq 2 \cdot subsize(shp(u)) \geq 2 \cdot subsize(u)$ when this happens, and this inequality persists. Suppose then that both u and $shp(u)$ get their parents in Part 1 or 2. When u gets its parent, $size(shp(u)) \geq 2 \cdot subsize(u)$. Subsequently, $size(shp(u))$ cannot decrease until $shp(u)$ gets its parent, at which time $subsize(shp(shp(u))) \geq size(shp(u)) \geq 2 \cdot subsize(u)$. This inequality persists. ∎

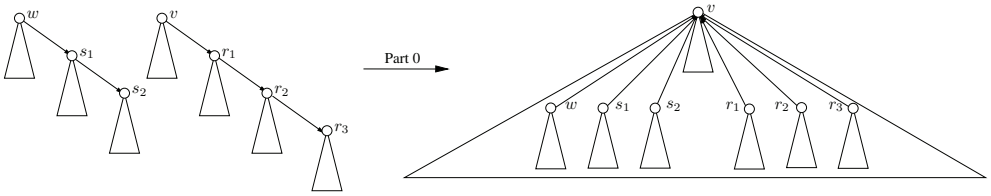

FIG. 4.3. *Linking by rank: Part* 0, *maxrank*(v) = *maxrank*(w).

COROLLARY 4.2. *The subtrees in any shadow forest built using linking-by-size are* $\sqrt{2}$*-balanced.*

4.4. Linking by rank. An alternative to using linking-by-size in the sophisticated link-eval structure is to use linking-by-rank. In place of a size, every node has a nonnegative integer *rank*, initially zero. The ranks satisfy the following invariant:

(iii) *rank*(*shp*(v)) > *rank*(v).

We explicitly maintain ranks only for subroots. If v is a virtual tree root (i.e., in F), we denote by *maxrank*(v) the maximum rank of a subroot descendant. With each virtual tree root v, we store *maxrank*(v) (in addition to *rank*(v)).

Perform *link*(v, w, x) as follows. First, set $b(w) = x$. Then compare *maxrank*(v) to *maxrank*(w). We split the rest of the operation into the following parts.

Part 0. If *maxrank*(v) = *maxrank*(w), set *rank*(v) = *maxrank*(v)+1, *maxrank*(v) = *maxrank*$(v) + 1$, and combine all the subtrees in the trees rooted at v and w into a single subtree rooted at v by setting *shp*$(u) = v$ for each subroot $u \neq v$, setting *shc*(v) = *null*, and setting $b(u) = \min\{b(u), b(w)\}$ if u was a descendant of w. (See Figure 4.3.)

Part 1. If *maxrank*(v) > *maxrank*(w), set *rank*$(v) = \max\{$*rank*(v), *maxrank*$(w) + 1\}$, and combine the subtree rooted at v with all the subtrees in the tree rooted at w by setting *shp*$(u) = v$ and $b(u) = \min\{b(u), b(w)\}$ for each subroot descendant u of w.

Part 2. If *maxrank*(v) < *maxrank*(w), combine all the subtrees in the tree rooted at v into a single subtree, unless *shc*(v) = *null*, by setting *rank*(v) = *maxrank*$(v) + 1$, *maxrank*(v)=*maxrank*(w), and, for each subroot $u \neq v$, *shp*$(u) = v$. Then set *shc*(v) = w. This may cause violations of invariants (i) and (ii).

Part 3. To restore (i) and (ii) after Part 2, repeat the following step until it no longer applies. Let s_0 = *shc*(v) and s_1 = *shc*(s_0). If $s_1 \neq$ *null* and $x < b(s_1)$, compare *rank*(s_0) to *rank*(s_1), and if *rank*(s_0) = *rank*(s_1), simultaneously set *shp*$(s_1) = s_0$, *shc*(s_0) = *shc*(s_1), *rank*(s_0) = *rank*$(s_0) + 1$, and *maxrank*(v) = $\max\{$*maxrank*(v), *rank*$(s_0) + 1\}$; if *rank*(s_0) > *rank*(s_1), simultaneously set *shp*(s_1) = s_0 and *shc*(s_0) = *shc*(s_1); if *rank*(s_0) < *rank*(s_1), simultaneously set *shp*$(s_0) = s_1$, *shc*$(v) = s_1$, and $b(s_1) = x$.

Parts 1, 2, and 3 of linking-by-rank correspond to Parts 1, 2, and 3 of linking-by-size; Part 0 handles the case of equal *maxranks*, in which all subtrees of both trees are combined. (We could add a corresponding Part 0 to linking-by-size, but this is unnecessary.) As does linking-by-size, linking-by-rank produces balanced forests, as we now show. For a node u, let *subsize*(u) be the number of descendants of u in its subtree.

LEMMA 4.3. *In any shadow forest built using linking-by-rank, any node* u *has* *subsize*$(u) \geq 2^{(rank(u)-1)/2}$.

Proof. To obtain this result we actually need to prove something stronger. Suppose we perform a sequence of link-by-rank operations. We track the states of nodes,

their ranks, and their subsizes as the links take place. Each node is in one of two states: *normal* or *special*. The following invariants will hold:

(a) a *normal node* u has $subsize(u) \geq 2^{rank(u)/2}$;

(b) a *special node* u has $subsize(u) \geq 2^{(rank(u)-1)/2}$;

(c) a *special root* u has a normal subroot descendant of rank at least $rank(u)$.

Initially all nodes are normal; since all initial ranks are zero, (a), (b), and (c) hold initially. We need to determine the effect of each part of an operation $link(v, w, x)$.

If $maxrank(v) = maxrank(w)$, we make v normal after the link; all other nodes retain their states. This preserves (a), (b), and (c); the only question is whether v satisfies (a), since it gains one in rank and can change from special to normal. Before the link, both the tree rooted at v and the tree rooted at w have a subroot of rank $maxrank(v)$. Since each of these nodes has subsize at least $2^{(maxrank(v)-1)/2}$ before the link by (a) and (b), after the link $subsize(v) \geq 2 \cdot 2^{(rank(v)-2)/2} = 2^{rank(v)/2}$. Hence (a) holds for v after the link.

If $maxrank(v) > maxrank(w)$ and $rank(v)$ does not change as a result of the link, all nodes retain their states. The link preserves (a), (b), and (c), because no node increases in rank. If $rank(v)$ does change because of the link (becoming one greater than the old value of $maxrank(w)$), we make v special. Node v now satisfies (b), because before the link w had a normal subroot descendant u of rank $maxrank(w)$, and $subsize(u) \geq 2^{maxrank(w)/2}$ by (a); hence, after the link, $subsize(v) \geq 2^{(rank(v)-1)/2}$. Node v satisfies (c), because before the link it had a normal subroot descendant z of rank $maxrank(v) \geq maxrank(w) + 1$, which it retains after the link.

The last case is $maxrank(v) < maxrank(w)$. In this case we look at the effects of Part 2 and Part 3 separately. If Part 2 does anything, we make v special. Node v satisfies (b), because before the link it had a normal subroot descendant of rank $maxrank(v)$, which satisfied (a); hence, after the link, $subsize(v) \geq 2^{(rank(v)-1)/2}$. Node v satisfies (c) after the link, because before the link w had a normal subroot descendant of rank $maxrank(w) \geq maxrank(v) + 1$ by (a), which becomes a normal subroot descendant of v.

Finally, we must account for the effect of Part 3. Each combination of subtrees done by Part 3 preserves (a), (b), and (c), except possibly for those that combine two subtrees with subroots, say y and z, of equal rank. In this case the rank of the surviving subroot increases by one; and if the ranks of y and z previously equaled $maxrank(v)$, $maxrank(v)$ increases by one. To preserve the invariants in this case, we make the surviving root, say y, normal. Now y satisfies (a), because before the subtrees rooted at y and z were combined, both y and z have subsize at least $2^{(rank(y)-1)/2}$; after the subtrees are combined, $subsize(y) \geq 2 \cdot 2^{(rank(y)-2)/2} = 2^{rank(y)/2}$. Because y satisfies (a), v satisfies (c).

Thus linking preserves the invariants. By induction, they remain true throughout any sequence of links. The lemma follows from (a) and (b). □

COROLLARY 4.4. *The subtrees in any shadow forest built using linking-by-rank are* $\sqrt{2}$*-balanced.*

THEOREM 4.5. *A sequence of operations performed using the sophisticated link-eval structure with either linking-by-size or linking-by-rank takes* $O(n)$ *time for the links and* $O(n + m\alpha(m + n, n))$ *time for the findroot and eval operations.*

Proof. The time for a link is $O(k + 1)$, where k is the decrease in the number of subtrees caused by the link. Thus the total time for all the links is $O(n)$. The total length of compressed paths, and hence the total time for *findroot* and *eval* operations, is $O(n+m\alpha(m+n, n))$ by the Tarjan–van Leeuwen analysis of path compression [61], applying Corollary 4.2 (for linking-by-size) or Corollary 4.4 (for linking-by-rank). □

4.5. Refined analysis of path compression. In order to use path compression on balanced trees as a tool for building linear-time algorithms, we need to show that the total time becomes linear if the compressions are suitably restricted. In order to capture both DSU and link-eval applications, we abstract the situation as follows. An intermixed sequence of the following two kinds of operations is performed on a rooted forest, initially consisting of n single-node trees:

- *assign*(u, v). Given two distinct roots u and v, make u the parent of v.
- *compress*(u). Compress the path to u from the root of the tree containing it by making the root the parent of every other node on the path.

LEMMA 4.6. *Suppose ℓ nodes are* marked *and the remaining $n - \ell$* unmarked. *Suppose the assignments build a balanced forest, and that each node has its parent change at most k times before it is in a tree containing a marked node. If there are m compress operations, then the total number of nodes on compression paths is $O(kn + m\alpha(m + \ell, \ell))$.*

Proof. Let F be the balanced forest built by the entire sequence of assignments, ignoring the compressions; let $c > 1$ be such that F is c-balanced; and let $h(v)$ be the height of a node v in F. Let

$$a = \lceil \log_c (n/\ell) + \log_c (1/(c-1)) + 1 \rceil.$$

Classify each node v into one of three types: *low*, if v has no marked descendant in F; *middle*, if v has a marked descendant in F and $h(v) < a$; and *high* otherwise.

A compression path from a tree root to one of its descendants consists of zero or more high nodes followed by zero or more middle nodes followed by zero or more low nodes. Every node on the path except the first two (totaling at most $2m$ over all compressions) has its parent change to one of greater height as a result of the compression.

Consider a compression path containing only low nodes. Since the root is low, the tree in which the compression takes place contains no marked nodes. All but two nodes on the path change parent but remain in a tree with no marked nodes. The number of times this can happen to a particular node is at most k by the hypothesis of the lemma, totaling at most kn over all compressions.

Consider a compression path containing at least one middle or high node. Every low node on the path except one has its parent change from low to middle or high as a result of the compression. Thus the total number of low nodes on such paths is at most $n + m$. Every middle node on the path whose parent changes obtains a parent of greater height. This can happen to a middle node at most a times before its parent is high. At most one middle node on a compression path has a high parent, totaling at most m over all compression paths. Each middle node has a marked node as a descendant; each marked node has at most $a + 1$ middle nodes as ancestors (at most one per height less than a). The total number of middle nodes is thus at most $\ell(a + 1)$. Combining estimates, we find that the total number of middle nodes on compression paths is at most $\ell \cdot a \cdot (a + 1) + m$. Since $\ell \le n$ and a is $O(\log (n/\ell))$, the first term is $O(n)$, implying that the total number of middle nodes on compression paths is $O(n) + m$.

Finally, we need to count the number of high nodes on compression paths. Since F is c-balanced, the total number of high nodes is at most

$$\sum_{i \ge a} \frac{n}{c^i} \le \frac{n}{c^a} \cdot \frac{c}{c-1} = \frac{n}{c^{a-1}(c-1)} \le \ell.$$

Let the *rank* of a node v be $h(v) - a$. Then every high node has nonnegative rank, and the number of high nodes of rank $i \geq 0$ is at most ℓ/c^i. The analysis of Tarjan and van Leeuwen [61, Lem. 6] applied to the high nodes bounds the number of high nodes on compression paths by $O(\ell + m\alpha(m + \ell, \ell))$. Combining all our estimates gives the lemma. □

Lemma 4.6 gives a bound of $O(n+m)$ if, for example, $\ell = O(n/\log\log n)$, by the properties of the inverse-Ackermann function [56]. In our applications $\ell = n/\log^{1/3} n$, which is sufficiently small to give an $O(n + m)$ bound.

We conclude this section by reviewing some previous results on DSU and refined analysis of the DSU structure. The linear-time RAM DSU algorithm of Gabow and Tarjan [29] assumes a priori knowledge of the unordered set of unions. An earlier version of our work [15] contained a result much weaker than Lemma 4.6, restricted to disjoint set union, which required changing the implementation of unite based on the marked nodes. Alstrup et al. [10] also proved a weaker version of Lemma 4.6 in which the $m\alpha(m + \ell, \ell)$ term is replaced by $\ell \log \ell + m$, which sufficed for their purpose. They derived this result for a hybrid algorithm that handles long paths of unary nodes outside the standard DSU structure. Dillencourt, Samet, and Tamminen [20] gave a linear-time result assuming the *stable tree property*: essentially, once a find is performed on any element in a set X, all subsequent finds on elements currently in X must be performed before X can be united with another set. Fiorio and Gustedt [25] exploited the specific order of unions in an image-processing application. Gustedt [36] generalized the previous two works to consider structures imposed on sets of allowable unions by various classes of graphs. This work is orthogonal to that of Gabow and Tarjan [29]. Other improved bounds for path compression [14, 44, 46] restrict the order in which finds are performed, in ways different from our restriction.

5. Topological graph computations. Consider a computation that takes as input a graph G whose vertices and edges (or arcs) have $O(1)$-bit labels and produces some output information (possibly none) associated with the graph itself and with each vertex and edge (or arc). We call such a computation a *topological graph computation*, because it is based only on the graph structure and the $O(1)$-bit labels, in contrast, for example, to a problem in which graph vertices and edges (or arcs) have associated real values. In general the output of a topological graph computation can be arbitrarily complex, even exponential in size, and can contain pointers to elements of the input graph. Our MST verification algorithm will exploit this flexibility; in all our other applications, the size of the output is linear in the size of the input.

Suppose we need to perform a topological graph computation on not just one input graph but on an entire collection of graphs. If the input instances are small and there are many of them, then many of them will be isomorphic. By doing the computation once for each nonisomorphic instance (a *canonical instance*) and copying these solutions to the duplicate instances, we can amortize away the cost of actually doing the computations on the canonical instances; most of the time is spent identifying the isomorphic instances and transferring the solutions from the canonical instances to the duplicate ones. The total time spent is then linear in the total size of all the instances.

Gabow and Tarjan [29] used this idea to solve a special case of DSU in which the unordered set of unions is given in advance; Dixon, Rauch, and Tarjan [21] applied the technique to MST verification and other problems. These applications use table look-up and require a RAM. Here we describe how to accomplish the same thing on a pointer machine. Our approach is as follows. Encode each instance as a list of

pointers. Use a radix sort to sort these lists. Identify the first instance in each group of identically encoded instances as the canonical instance. Solve the problem for each canonical instance. Map the solutions back to the duplicate instances. The details follow.

Let $\mathcal{G}$ be the set of input instances, each of which contains at most g vertices. Let N be the total number of vertices and edges (or arcs) in all the instances. Let k be the maximum number of bits associated with each vertex and edge (or arc) of an instance. Construct a singly linked master list whose nodes, in order, represent the integers from zero through $\max\{g, 2^k + 1\}$ and are so numbered. For each instance G, perform a DFS, numbering the vertices in preorder and adding to each vertex a pointer into the master list corresponding to its preorder number; the preorder numbering allows us to maintain a global pointer into the master list to facilitate this assignment of pointers to vertices. Represent the label of each vertex and edge (or arc) by a pointer into the master list, using a pointer to the zero node to encode the lack of a label. Construct a list L of triples corresponding to the vertices of G, one triple per vertex, consisting of a pointer to the vertex, and its number and label, both represented as pointers into the master list. Construct a list Q of quadruples corresponding to the edges (or arcs) of the graph, one quadruple per edge (or arc), consisting of a pointer to the edge (or arc), and the numbers of its endpoints and its label, represented as pointers into the master list. (For an undirected graph, order the numbers of the edge endpoints in increasing order.) Encode the instance by a list whose first entry is a pair consisting of a pointer to the instance and the number of its vertices, represented as a pointer into the master list, catenated with lists L and Q.

Constructing encodings for all the instances takes $O(N)$ time. Recall that the elements of the encodings are pointers to the master list. Attach a bucket to each element of the master list. Use a radix sort for variable length lists [2], following the encoding pointers to reach the buckets, to arrange the encodings into groups that are identical except for the first components of each list element (pair, triple, or quadruple): instances whose encodings are in the same group are isomorphic. This also takes $O(N)$ time.

Now perform the topological graph computation on any one instance of each group (the canonical instance for that group). Finally, for each duplicate instance, traverse its encoding and the encoding of the corresponding canonical instance concurrently, transferring the solution from the canonical instance to the duplicate instance. The exact form this transfer takes depends upon the form of the output to the topological graph computation. One way to do the transfer is to traverse the encodings of the canonical instance and the duplicate instance in parallel, constructing pointers between corresponding vertices and edges (or arcs) of the two instances. Then visit each vertex and edge (or arc) of the canonical instance, copying the output to the duplicate instance but replacing each pointer to a vertex or edge (or arc) by a pointer to the corresponding vertex or edge (or arc) in the duplicate instance. If the output has size linear in the input, this takes $O(N)$ time. Summarizing, we have the following theorem.

THEOREM 5.1. *If the output of a topological graph computation has size linear in the input size, the computation can be done on a collection of instances of total size N in $O(N)$ time on a pointer machine, plus the time to do the computation on one instance of each group of isomorphic instances.*

This method extends to allow the vertices and edges (or arcs) of the instances to be labeled with integers in the range $[1, g]$ if these labels are represented by pointers to the nodes of a precomputed master list. We shall need this extension in our

applications to finding dominators and computing component trees (sections 9 and 10, respectively). In another of our applications, MST verification, the output of the topological graph computation has exponential size: it is a comparison tree, whose nodes indicate comparisons between the weights of two edges. In this case, we do not construct a new copy of the comparison tree for each duplicate instance. Instead, when we are ready to run the comparison tree for a duplicate instance, we construct pointers from the edges of the canonical instance to the corresponding edges of the duplicate instance and run the comparison tree constructed for the canonical instance, but comparing weights of the corresponding edges in the duplicate instance. The total time is $O(N)$ plus the time to build the comparison trees for the canonical instances plus the time to run the comparison trees for all the instances.

It remains to bound the time required to do the topological graph computation on the canonical instances. The number of canonical instances is $g^{O(g^2)}$. In all but one of our applications, the time to do a topological graph computation on an instance of size g or smaller is $O(g^2)$; for MST verification, it is $g^{O(g^2)}$. Thus the following theorem suffices for us.

THEOREM 5.2. *If a topological graph computation takes $g^{O(g^2)}$ time on a graph with g or fewer vertices, and if $g = \log^{1/3} N$, then the total time on a pointer machine to do the topological graph computation on a collection of graphs of total size N, each having at most g vertices, is $O(N)$.*

Proof. The proof is immediate from Theorem 5.1, since the total time to do the topological graph computation on the canonical instances is $g^{O(g^2)}g^{O(g^2)} = g^{O(g^2)} = O(N)$. □

The ability to recover the answers from the topological graph computations on the instances in $\mathcal{G}$ is subtle but crucial. Alstrup, Secher, and Spork [8] show how to compute connectivity queries on a tree T undergoing edge deletions in linear time. They partition T into bottom-level microtrees (discussed in the next section) and compute, for each vertex v in a microtree, a bit-string that encodes the vertices on the path from v to the root of its microtree. They show how to answer connectivity queries using a constant number of bitwise operations on these bit-strings and applying the Even and Shiloach decremental connectivity algorithm [23] to the upper part of T.

The Alstrup, Secher, and Spork algorithm [8] runs on a pointer machine: since the connectivity queries return yes/no answers, they do not need index tables to recover the answers. In contrast, while their method can be extended to solve the off-line NCAs problem in linear time on a RAM, and even to simplify the Gabow–Tarjan linear-time DSU result [29], both of these extensions require index tables to map the results of the bitwise operations back to vertices in T.

The idea of using pointers to buckets in lieu of indexing an array was described in general by Cai and Paige [17] in the context of multisequence discrimination. Their technique leaves the efficient identification of buckets with specific elements as an application-dependent problem. They solve this problem for several applications, including discriminating trees and DAGs, but their solutions exploit structures specific to their applications and do not extend to general graphs.

6. Nearest common ancestors. We now have the tools to solve our first application, the off-line nearest common ancestors (NCAs) problem: given a rooted n-node tree T and a set P of m queries, each of which is a pair $\{v, w\}$ of nodes in T, compute $nca(v, w)$ for each query $\{v, w\}$. Aho, Hopcroft, and Ullman's algorithm [3] for this problem, as presented by Tarjan [58], solves it using DSU. The algorithm traverses T bottom-up, building a shadow copy as a DSU forest. It maintains, for each subtree

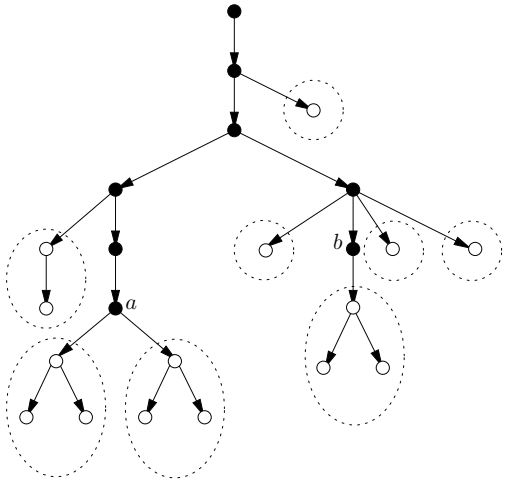

FIG. 6.1. *Partitioning of a tree T with $g = 3$; fringe nodes are open, and core nodes are filled; bottom-level microtrees are encircled. Nodes a and b are the leaves of the core.*

built so far, the set of its nodes, with the root of the subtree as the designated element. Initially, each node is in a singleton set. Each node v also has a set $P(v)$ of queries $\{v, w\}$; each query is in two such lists, one for v and one for w. The algorithm is as follows.

Visit the nodes of T in a postorder [54]. (Any postorder will do.) When visiting a node v, for every pair $\{v, w\}$ in $P(v)$ such that w has already been visited, return *find*(w) as the answer to the query *nca*(v, w). Finish the visit to v by performing *unite*$(p(v), v)$ if v is not the root of T, where $p(v)$ is the parent of v in T.

The correctness of this algorithm follows from basic properties of postorder. The DSU operations dominate the running time, which is $O(n+m\alpha(m+n, n))$ if the standard DSU structure presented in section 4 is used. In this algorithm, the unordered set of unions is known in advance, since it is given by the input tree T. Thus the use of the Gabow–Tarjan [29] linear-time RAM DSU algorithm results in a linear-time RAM algorithm for NCAs. Knowing the set of unions in advance, however, is not sufficient to solve the DSU problem in linear time on a pointer machine [46]. We exploit a different property of the unions: they occur in a bottom-up order.

We partition T into a set of small bottom-level trees, called *microtrees*, and T', the rest of T. For any node v, let $T(v)$ be the subtree of T induced by the descendants of v, and let $|T(v)|$ be the number of nodes in $T(v)$. Let $g \geq 1$ be a fixed parameter to be chosen later. We define $T(v)$ to be a *microtree* if $|T(v)| \leq g$ but $|T(p(v))| > g$. For a node x in $T(v)$, *micro*$(x) = T(v)$ is the *microtree of* x and *root*(*micro*(x)) is the *root of its microtree*. Let T' be the subtree of T induced by the vertices in T that are not in microtrees. Each leaf in T' has at least g descendants in T, and the descendants of two different leaves of T' form disjoint sets, so T' has at most n/g leaves. We call the microtrees the *fringe* of T and T' the *core* of T. See Figure 6.1. It is straightforward to partition T into its microtrees and core in linear time by visiting the nodes in postorder and computing their numbers of descendants.

We call a query $\{v, w\}$ *small* if v and w are in the same microtree, and *big* otherwise. We can partition the queries into big and small and assign each small query to the microtree containing it in linear time. We answer all the big queries by

using the AHU algorithm. We answer all the small queries by doing a topological graph computation on the set of graphs defined by each microtree and its associated queries. By choosing g appropriately, we get a linear-time bound for both parts of the computation.

Specifically, choose $g = \log^{1/3} n$. Answer all the big queries by running the AHU algorithm, restricted to the big queries. To bound the running time, apply Lemma 4.6 to the tree built by the parent assignments done by the unite operations. Mark every leaf of T'. Each find occurs in a set containing at least one marked node. Therefore, setting $k = 1$, to count the initial parent assignment for each node, satisfies the hypothesis of the lemma. Since the number of marked nodes is at most $n/g = n/\log^{1/3} n$, the lemma implies an $O(n + m)$ bound on the time to answer all the big queries.

Answer all the small queries by constructing, for each microtree, a graph containing the microtree edges and, for each query with both nodes in the microtree, an edge denoted as a query edge by a bit. Then do a topological graph computation on these graphs to answer the small queries, using the method of section 5. With $g = \log^{1/3} n$, this takes $O(n + m)$ time. Thus we obtain the following theorem.

THEOREM 6.1. *The off-line NCAs problem can be solved in $O(n + m)$ time on a pointer machine.*

7. Minimum spanning trees.

7.1. Verification. Our next applications, minimum spanning tree (MST) verification and construction, combine topological graph processing with use of the simple link-eval structure of section 4.2. Let T be a spanning tree of a connected, undirected graph G whose edges have real-valued weights. For any edge $\{v, w\}$, let $c(v, w)$ be the weight of $\{v, w\}$. We denote the set of nontree edges by P. For any pair (v, w) of vertices, we denote by $T(v, w)$ the unique path from v to w in T. The tree T is minimum if and only if, for every edge $\{v, w\}$ in P, $c(v, w) \geq c(x, y)$ for every edge $\{x, y\}$ on $T(v, w)$. Thus to verify that T is minimum it suffices to compute $\max\{c(x, y) : \{x, y\} \text{ on } T(v, w)\}$ for every edge $\{v, w\}$ in P. We assume henceforth that T is rooted at a fixed but arbitrary vertex and that each vertex v has a set $P(v)$ of the pairs $\{v, w\}$ in P.

Tarjan's $O(m\alpha(m, n))$-time MST verification algorithm [58] is like the AHU NCA algorithm, except that it uses a link-eval structure (with max instead of min) in place of a DSU structure to compute the needed path maxima. The algorithm builds the link-eval forest during a bottom-up traversal of T. As part of the process of computing path maxima, the algorithm computes $u = \mathit{nca}(v, w)$ for each pair $\{v, w\}$ in P and stores $\{v, w\}$ in a set $Q(u)$. Initially each node of T is in a single-node tree of the link-eval structure, and $Q(u)$ is empty for each node u. The algorithm follows.

Visit the nodes of T in a postorder. (Any postorder will do.) When visiting a vertex v, for every pair $\{v, w\}$ in $P(v)$ such that w has already been visited, add $\{v, w\}$ to $Q(\mathit{findroot}(w))$. For every pair $\{x, y\}$ in $Q(v)$, return $\max\{\mathit{eval}(x), \mathit{eval}(y)\}$ as the answer to the query $\{x, y\}$. Finish the visit to v by performing $\mathit{link}(p(v), v, c(p(v), v))$ unless v is the root of T.

When the algorithm answers a query $\{x, y\}$ while visiting a vertex v, $v = \mathit{nca}(x, y)$, and $\mathit{eval}(x)$ and $\mathit{eval}(y)$ are the maximum costs of the arcs on $T(v, x)$ and $T(v, y)$, respectively. In Tarjan's original presentation, the NCA calculations are separate from the path evaluations, but combining them gives a more coherent algorithm. Ignoring the arc costs and *eval* operations, the link-eval structure functions exactly like the DSU structure in the AHU NCA algorithm.

If the sophisticated link-eval structure of section 4.3 or section 4.4 is used, this algorithm runs in $O(m\alpha(m,n))$ time. Unfortunately, these structures delay the effect of the links, so parent assignments do not necessarily occur in a bottom-up order, and we cannot immediately apply the approach of section 6 to reduce the running time to linear. This problem was pointed out by Georgiadis and Tarjan [32]. Instead, we use a result of King [40] to transform the original tree into an $O(n)$-node balanced tree on which to compute path maxima. Then we can use the simple link-eval structure of section 4.2 in combination with the approach of section 6 to obtain a linear-time algorithm.

7.2. The Borůvka tree. A *Borůvka step* [13] applied to a weighted, undirected graph G is as follows: Select a least-weight edge incident to each vertex, and contract to a single vertex each connected component formed by the selected edges. Repeating this step until only a single vertex remains produces an MST defined by the original edges corresponding to the edges selected in all the steps if all edge weights are distinct, which we can assume without loss of generality.

This algorithm can be enhanced to produce the *Borůvka tree* B, whose nodes are the connected components that exist during the Borůvka steps, with each node having as children those components from which it is formed during a Borůvka step. If component C is the parent of component D, the weight of arc (C,D) is the weight of the edge selected for the vertex corresponding to D by the Borůvka step in which D is contracted into C. The leaves of B are the vertices of G, each of which is originally a single-vertex component. Each Borůvka step reduces the number of vertices by at least a factor of two; hence, B is 2-balanced. Also, B contains at most $2n-1$ nodes. In general the enhanced Borůvka algorithm runs in $O(m \log n)$ time on a pointer machine. On a tree, however, it runs in $O(n)$ time, because each contracted graph is a tree, and a tree has $O(n)$ edges. We apply the enhanced Borůvka algorithm to the tree T that is to be verified, thereby constructing the Borůvka tree B of T. In addition to being balanced, B has the following key property [40]: for any pair of vertices $\{v,w\}$, $\max\{c(x,y) : (x,y) \text{ on } T(v,w)\} = \max\{c(x,y) : (x,y) \text{ on } B(v,w)\}$. Thus we can compute path maxima on B instead of on T without affecting the answers to the queries.

7.3. Comparison trees for computing path maxima. Now we can apply the approach of section 6. Let $g = \log^{1/3} n$. Partition B into microtrees and a core B' as in section 6. Partition the pairs in P into *big pairs*, those with ends in different microtrees, and *small pairs*, those with ends in the same microtree. Compute path maxima for all the big pairs by running Tarjan's algorithm on B, restricted to the big pairs and using the simple link-eval structure of section 4.2.

To bound the running time of this computation, we apply Lemma 4.6 to B. Mark every leaf of B'. Each *findroot* and *eval* occurs in a subtree of B containing a marked node, so setting $k = 1$ satisfies the hypothesis of the lemma. Since the number of marked nodes is at most $2n/g = 2n/\log^{1/3} n$, the lemma implies an $O(m)$ bound on the time to compute path maxima for all the big pairs.

We would like to compute path maxima for all the small pairs by applying the method of section 5. To this end, construct for each microtree a graph containing the microtree edges and, for each pair with both ends in the microtree, an edge designated as a query edge by a bit. Now a new difficulty arises: since the edge costs are arbitrary real numbers, computing path maxima is not a topological graph computation; we cannot encode the edge costs in $O(1)$ bits, or even in $O(\log g)$ bits.

We overcome this difficulty by following the approach of Dixon, Rauch, and Tar-

jan [22]: Do a topological graph computation that builds, for each distinct marked graph, a comparison tree, whose nodes designate binary comparisons between costs of unmarked edges of the graph (tree edges), such that the output nodes of the comparison tree designate, for each marked edge (query pair), which of the unmarked edges on the path between the ends of the edge has maximum cost. Having built all the comparison trees, run the appropriate comparison tree for each microtree and its associated pairs, using the actual costs of the microtree arcs to determine the outcomes of the comparisons.

With $g = \log^{1/3} n$, the time for this computation is $O(m)$, plus the time to build comparison trees for the topologically distinct instances, plus the time to run the comparison trees for the actual instances. Komlós [41] proved that the path maxima needed for MST verification can be determined in a number of binary comparisons of tree edge costs that is linear in the number of graph edges, which implies for each instance the existence of a comparison tree that has depth linear in the number of edges. Dixon, Rauch, and Tarjan [22] observed that the comparison tree implied by Komlós' result can be built in a time per comparison-tree node that is quadratic in the number of graph vertices. If we use their method to build the comparison trees during the topological graph computation, then $g = \log^{1/3} n$ implies by the results of section 5 that the total time to build the comparison trees is $O(m)$. The total time to run them is linear in the total size of all the actual instances, which is also $O(m)$. Thus we obtain the following theorem.

THEOREM 7.1. *Computing all the path maxima needed for MST verification, and doing the verification itself, takes $O(m)$ time on a pointer machine.*

7.4. Construction of MSTs. The randomized linear-time MST construction algorithm of Karger, Klein, and Tarjan [39] runs on a pointer machine except for the part that computes the path maxima needed for MST verification. Using the algorithm of section 7.3, this part can be done (deterministically) in linear time on a pointer machine, resulting in a randomized linear-time pointer-machine algorithm for constructing an MST.

7.5. Remarks. It is instructive to compare our MST verification algorithm to those of Dixon, Rauch, and Tarjan [22] and of King [40]. Our use of King's Borůvka tree construction as an intermediate step allows us to use only bottom-level microtrees, whereas Dixon et al. partition the original tree entirely into microtrees, with an extra *macrotree* to represent the connections between them. It also allows us to use the simple link-eval structure instead of the sophisticated one. Lemma 4.6 allows us to break big queries into only two parts (having an NCA in common); Dixon et al. break each big query into as many as six parts. King explicitly implements Komlós' comparison algorithm for the Borůvka tree, but her algorithm is heavily table-driven and requires a RAM. She also must compute NCAs separately.

There is an alternative, though more complicated, way to verify an MST in linear time on a pointer machine. This method replaces the use of the Borůvka tree by a partition of the original tree into bottom-level microtrees and a set of maximal paths that partition the core. The method does NCA computations on trees derived from the maximal paths, and it uses a sophisticated link-eval structure instead of the simple one. We discuss this method in more detail in section 9.7. Though the use of the Borůvka tree gives us a simpler algorithm for MST verification, there is no corresponding concept for either of our remaining applications, and we must rely on the alternative of partitioning the core into maximal paths.

8. Interval analysis. We turn now to two problems on flowgraphs. The first is *interval analysis*. Let $G = (V, A, r)$ be a flowgraph, and let D be a given DFS tree rooted at r. Identify vertices by their preorder number with respect to the DFS: $v < w$ means that v was visited before w. *Reverse preorder* of the vertices is decreasing order by (preorder) vertex number. For each vertex v, the *head* of v is

$$h(v) = \max\{u \neq v : \text{there is a path from } v \text{ to } u \text{ containing only descendants of } u\};$$

$h(v) = \mathit{null}$ if this set is empty. The heads define a forest H called the *interval forest*: $h(v)$ is the parent of v in H. Each subtree $H(v)$ of H induces a strongly connected subgraph of G, containing only vertices in $D(v)$ (the descendants of v in D). See Figure 8.1. Tarjan [57] proposed an algorithm that uses an NCA computation, incremental backward search, and a DSU data structure to compute H in $O(m\alpha(m, n))$ time on a pointer machine. We shall add microtrees, a maximal path partition of the core, and a stack to Tarjan's algorithm, thereby improving its running time to $O(m)$ on a pointer machine.

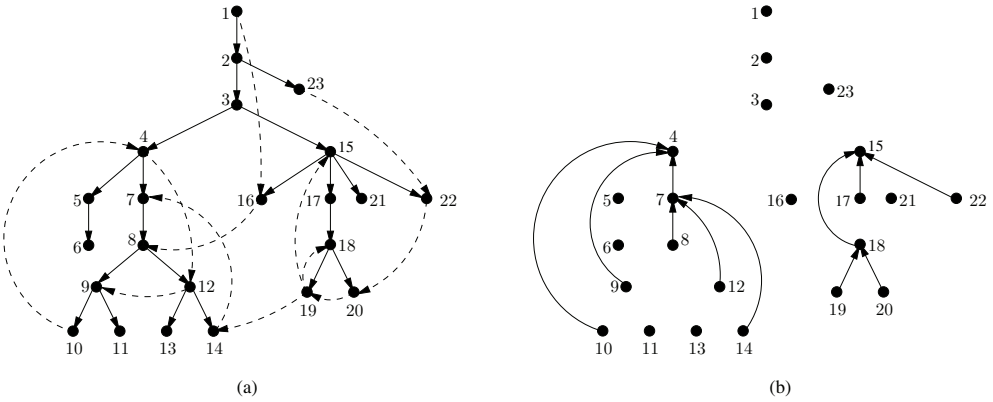

FIG. 8.1. (a) *A DFS tree D of the input flowgraph G; nontree arcs are dashed.* (b) *The interval forest H of G with respect to D; arrows are parent pointers.*

Tarjan's algorithm proceeds as follows: Delete all the arcs from the graph. For each vertex u, form a set of all deleted arcs (x, y) such that $\mathit{nca}(x, y) = u$. Process the vertices in any bottom-up order. (Tarjan uses reverse preorder, but any bottom-up order will do.) To process a vertex u, add back to the graph arcs corresponding to all the deleted arcs (x, y) with $\mathit{nca}(x, y) = u$. Then examine each arc (v, u) entering u. If $v \neq u$, set $h(v) = u$, and contract v into u; for all arcs having v as an end, replace v by u. This may create multiple arcs and loops, which poses no difficulty for the algorithm. Continue until all arcs into u have been examined, including those formed by contraction. When adding arcs back to the graph, the arc corresponding to an original arc is the one formed by doing end replacements corresponding to all the contractions done so far.

To keep track of contractions, Tarjan's algorithm uses a DSU structure whose elements are the graph vertices. The algorithm also uses a reverse adjacency set $R(u)$, initially empty, for each vertex u. A more detailed description of the algorithm is as follows. To process u, for each arc (x, y) such that $\mathit{nca}(x, y) = u$, add x to $R(\mathit{find}(y))$. (The replacement for x is done later.) Then, while $R(u)$ is nonempty, delete a vertex x from $R(u)$; let $v \leftarrow \mathit{find}(x)$; if $v \neq u$, set $h(v) \leftarrow u$, set $R(u) \leftarrow R(u) \cup R(v)$, and do $\mathit{unite}(u, v)$.

With the sets $R(u)$ represented as singly linked circular lists (so that set union takes constant time), the running time of this algorithm on a pointer machine is linear except for the NCA computations and the DSU operations, which take $O(m\alpha(m,n))$ time in Tarjan's original implementation. We shall reduce the running time to linear by using microtrees to eliminate redundant computation and by reordering the unites into a bottom-up order.

As in section 6, partition D into a set of bottom-level microtrees (the fringe), each with fewer than $g = \log^{1/3} n$ vertices, and D', the remainder of D (the core). Use a topological graph computation to compute $h(v)$ for every vertex v such that $h(v)$ is in the fringe. The definition of heads implies that for any such vertex v, $h(v)$ and v are in the same microtree, and furthermore that the only information needed to compute heads in the fringe is, for each microtree, the subgraph induced by its vertices, with nontree edges marked by a bit. With $g = \log^{1/3} n$, this computation takes $O(m)$ time by Theorem 5.2.

It remains to compute heads for vertices whose heads are in the core. Our approach is to run Tarjan's algorithm starting from the state it would have reached after processing the fringe. This amounts to contracting all the strong components in the fringe and then running the algorithm. This approach does not quite work as stated, because the DSU operations are not restricted enough for Lemma 4.6 to apply. To overcome this difficulty, we partition the core into maximal paths. Then we run Tarjan's algorithm path-by-path, keeping track of contractions with a hybrid structure consisting of a DSU structure that maintains contractions outside the path being processed and a stack that maintains contractions inside the path being processed. The latter structure functions in the same way as the one Gabow used in his algorithm [28] for finding strong components. Now we give the complete description of our algorithm.

Partition the vertices in D' into a set of maximal paths by choosing, for each nonleaf vertex v in D', a child $c(v)$ in D'. (Any child will do.) The arcs $(v, c(v))$ form a set of paths that partition the vertices in D'. For such a path P, we denote the smallest and largest vertices on P by $top(P)$ and $bottom(P)$, respectively; $bottom(P)$ is a leaf of D'. Since D' has at most n/g leaves, the number of paths is at most n/g. Partitioning D' into paths takes $O(n)$ time.

After constructing a maximal path partition of the core, initialize a DSU structure containing every vertex (fringe and core) as a singleton set. Visit the fringe vertices in bottom-up order, and, for each fringe vertex v with $h(v)$ also in the fringe, perform $unite(h(v), v)$; for such a vertex, $h(v)$ has already been computed. Initialize $R(u) \leftarrow \emptyset$ for every vertex u. For every arc (x, y) with x and y in the same microtree, add x to $R(find(y))$. For every remaining arc (x, y), compute $u = nca(x, y)$ and add (x, y) to the set of arcs associated with u. These NCA computations take $O(m)$ time using the algorithm of section 6. Indeed, every NCA query is big, so the AHU algorithm answers them in linear time. This completes the initialization.

Now process each path P in the path partition, in bottom-up order with respect to $top(P)$. To process a path P, initialize an empty stack S. Process each vertex u of P in bottom-up order. To process u, for each arc (x, y) such that $nca(x, y) = u$, add x to $R(find(y))$ unless $u = x$ and $p(y) \neq x$. (Heads do not depend on arcs (x, y) such that x is an ancestor of y but not its parent.) Then, while $R(u)$ is nonempty, delete a vertex x from $R(u)$. Let $v \leftarrow find(x)$. If v is not on P, set $h(v) \leftarrow u$, set $R(u) \leftarrow R(u) \cup R(v)$, and do $unite(u, v)$. If, on the other hand, v is on P, $v \neq u$, and v is no less than the top vertex on S, pop from S each vertex w less than or equal to v, set $h(w) \leftarrow u$, and set $R(u) \leftarrow R(u) \cup R(w)$. Once $R(u)$ is empty, push u onto

S. After processing all vertices on P, visit each vertex u on P again, in bottom-up order, and if $h(u)$ is now defined, perform *unite*$(h(u), u)$. See Figure 8.2

This algorithm delays the unites for vertices on a path until the entire path is processed, using the stack to keep track of the corresponding contractions. Specifically, the algorithm maintains the following invariant: if vertex u on path P is currently being processed and x is any original vertex, then the vertex into which x has been contracted is $v =$ *find*(x) if v is not on P, or the largest vertex on S less than or equal to v if v is on P and S is nonempty, or u otherwise. It is straightforward to verify this invariant by induction on time; the correctness of this implementation of Tarjan's algorithm follows.

THEOREM 8.1. *The interval analysis algorithm runs in $O(m)$ time on a pointer machine.*

Proof. The running time is linear except for the find operations: each vertex gets added to S once and has its head set at most once. To bound the time for the find operations, we apply Lemma 4.6 to the tree built by the parent assignments done by the unite operations. Mark the tops of all paths. Since there are at most n/g paths, there are at most $n/g = n/\log^{1/3} n$ marked vertices. We claim that $k = 5$ satisfies the hypothesis of the lemma. We need a property of the interval forest H: if $h(v) = u$, then every vertex $w \neq u$ on the path in D from u to v is a descendant of u in H. This holds because there is a path containing only vertices in $D(u)$ from w to v (via D) to u.

Consider any vertex v. We bound the number of times the parent of v in the DSU structure can change before v is in a set with a marked vertex. One parent change can occur during the initialization. After initialization, the parent of v can change once by a find before v is in a set with a designated vertex on the current path P. The parent of v can change only once more by a find while P is the current path, since its set does not change again until P is no longer the current path. Once P is not the current path, the next parent change of v caused by a find results in v being in a set with a designated vertex on the new current path Q. The parent of v can change only once more while Q is the current path, after which it is in the same set as *top*(P) (by the property above) and thus is in a set with a marked vertex. Therefore, the parent of v can change at most five times before it is in a set with a marked vertex, so the claim is true.

With $k = 5$ and $\ell \leq n/\log^{1/3} n$, Lemma 4.6 gives a bound of $O(m)$ on the time for the *find* operations. □

Interval analysis is an important component of program flow analysis [4]. It also has other applications, including testing flow graph reducibility [60], finding a pair of arc-disjoint spanning trees in a directed graph [57], and verifying a dominator tree [33]. Our interval analysis algorithm gives $O(m)$-time algorithms on a pointer machine for these applications as well.

In the next section we shall need a compressed version of the interval forest H' that is defined with respect to the fringe-core partition: the parent $h'(v)$ of a vertex v is its nearest core ancestor in H if it has one, *null* otherwise. We can easily compute H' from H in linear time, but if we only want H' and not H, we can avoid the topological graph computation on the microtrees: First, find the strong components of the graphs induced by the vertex sets of the microtrees. For each such component, find its smallest vertex u, and perform *unite*(u, v) for every other vertex v in the component. Then run the algorithm above for the core. This computes $h(v) = h'(v)$ for every vertex v with head in the core. Complete the computation by setting $h'(v) = h'(u)$ for each vertex $v \neq u$ in a fringe strong component with smallest vertex u.

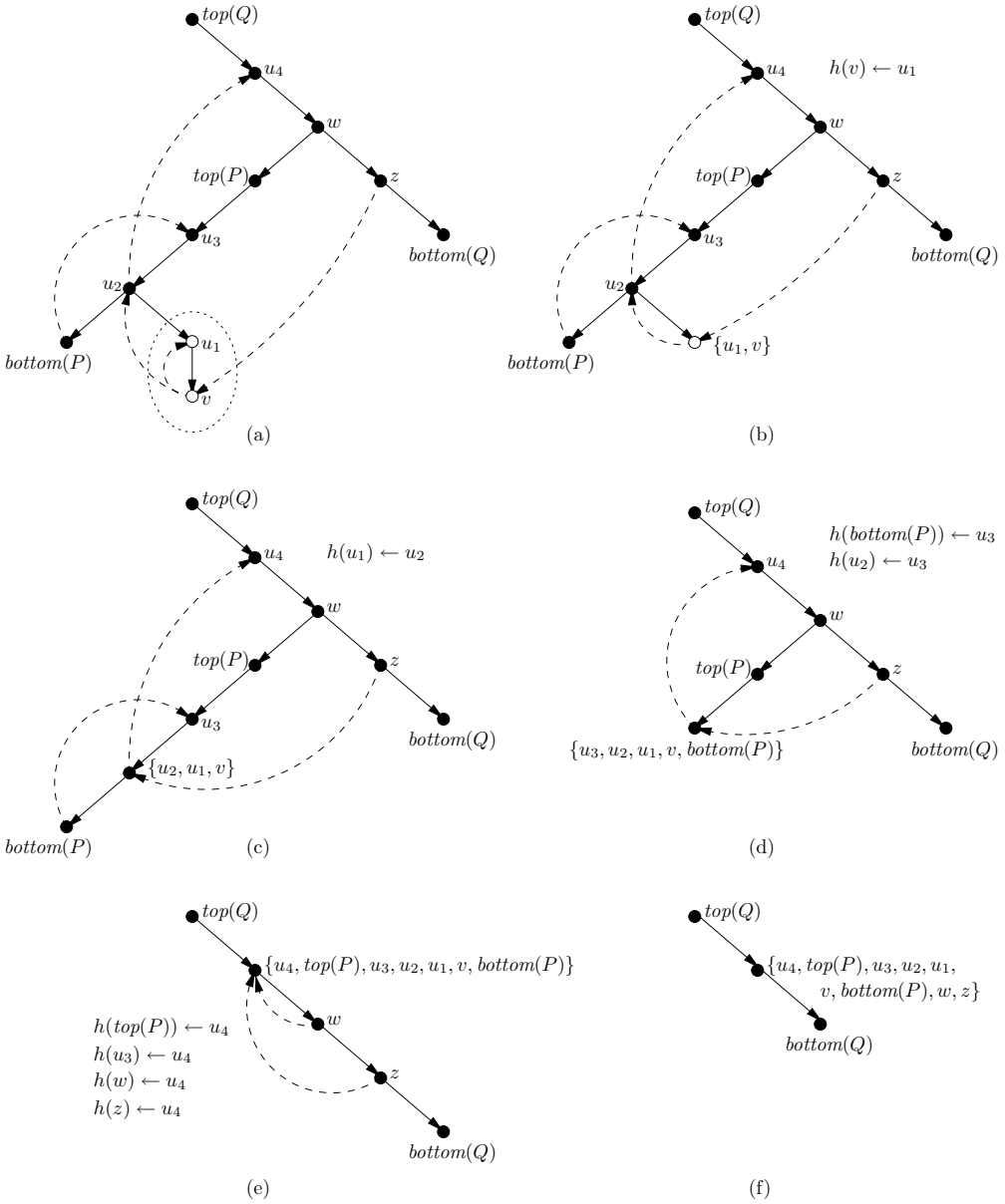

FIG. 8.2. *Idealized execution of the algorithm on the graph in* (a)*, with circled microtree. Arcs depict the effects of contractions: whenever* $x \in R(y)$*,* $(find(x), find(y))$ *is an arc in the corresponding graph. The first vertex in each labeled set is the corresponding original vertex in* (a). (a)→(b) *During preprocessing,* $h(v) \leftarrow u_1$*, and* v *is inserted into the set of* u_1*.* (b)→(c) *When processing* u_2*,* $h(u_1) \leftarrow u_2$ *via the arc* (v, u_2)*.* (c)→(d) *When processing* u_3*, the stack* S *is (top-down)* $(u_2, bottom(P))$*. Hence, when processing the arc* $(bottom(P), u_3)$*,* S *is popped so that* $h(u_2) \leftarrow u_3$ *and* $h(bottom(P)) \leftarrow u_3$*.* (d) *shows the state after doing the unite*$(\cdot)$*'s for path* P*.* (d)→(e) *When processing* u_4*,* S *is* $(w, z, bottom(Q))$*. Arc* (u_2, u_4) *sets* $h(u_3) \leftarrow u_4$ *and adds* $top(P)$ *and* z *to* $R(u_4)$*. Processing* $top(P)$ *causes* $h(top(P)) \leftarrow u_4$*, and processing* z *pops the stack so that* $h(w) \leftarrow u_4$ *and* $h(z) \leftarrow u_4$*.* (f) *After processing path* Q*.*

9. Dominators. Our second flowgraph problem is finding immediate dominators. Let $G = (V, A, r)$ be a flowgraph. We denote the immediate dominator of any vertex v by $idom(v)$. Let D be an arbitrary but fixed DFS tree rooted at r. As in section 8, we identify vertices by their preorder number with respect to the DFS; reverse preorder is decreasing order by vertex number. We use the notation $v \stackrel{*}{\to} w$ to denote that v is an ancestor of w in D, and $v \stackrel{+}{\to} w$ to denote that v is a proper ancestor of w in D. Sometimes we use the same notation to denote the respective paths in D from v to w. We denote by $p(v)$ the parent of v in D. We shall need the following basic property of DFS.

LEMMA 9.1 (see [54]). *Any path from a vertex v to a vertex $w > v$ contains a common ancestor of v and w.*

We shall describe an algorithm to compute immediate dominators in $O(m)$ time on a pointer machine. This is our most complicated application: it uses all the ideas and algorithms we have developed so far. Our algorithm is a reengineering of the algorithms presented by Buchsbaum et al. [16] and Georgiadis and Tarjan [31, 32]. As we proceed with the description, we shall point out the relationships between concepts we introduce here and the corresponding ideas in those previous works.

9.1. Semidominators, relative dominators, tags, and extended tags. Lengauer and Tarjan (LT) [43] devised a three-pass, $O(m\alpha(m, n))$-time algorithm to compute immediate dominators. We shall improve their algorithm by speeding up the first two steps. Central to the LT algorithm is the concept of *semidominators*. A path $x_0, x_1, \ldots, x_k$ in G is a *high path* if $x_i > x_k$ for $i < k$. As a degenerate case, a single vertex is a high path. A high path avoids all proper ancestors of its last vertex. The *semidominator* of a vertex w is

$$sdom(w) = \min(\{w\} \cup \{u : \text{for some } (u, v) \text{ in } A \text{ there is a high path from } v \text{ to } w\}).$$

The *relative dominator* of a vertex w is

$$rdom(w) = \operatorname{argmin}\{sdom(u) : sdom(w) \stackrel{+}{\to} u \stackrel{*}{\to} w\}.$$

With this definition, relative dominators are not unique, but for any vertex any relative dominator will do.

The LT algorithm operates as follows:

Step 1. Compute semidominators.

Step 2. Compute relative dominators from semidominators.

Step 3. Compute immediate dominators from relative dominators.

Step 3 relies on the following lemma.

LEMMA 9.2 (see [43, Cor. 1]). *For any vertex $v \neq r$, if $sdom(rdom(v)) = sdom(v)$, then $idom(v) = sdom(v)$; otherwise, $idom(v) = idom(rdom(v))$.*

Using Lemma 9.2, the LT algorithm performs Step 3 in a straightforward top-down pass over D that takes $O(n)$ time on a pointer machine.

The LT algorithm performs Steps 1 and 2 in a single pass that visits the vertices of D in reverse preorder and uses a link-eval data structure to compute semidominators and relative dominators. We shall present separate algorithms for Steps 1 and 2, although these steps can be partially combined, as we discuss in section 9.7.

Step 2 is almost identical to MST verification. Indeed, suppose we assign a cost $sdom(v)$ to each tree arc $(p(v), v)$ and apply the MST verification algorithm to the tree D (ignoring arc directions) with query set $Q = \{\{sdom(v), v\} : v \neq r\}$, with the modification that the answer to a query is an arc of minimum cost on the query path

rather than the cost of such an arc. Then for $v \neq r$, $rdom(v)$ is the vertex u such that $(p(u), u)$ is the answer to the query $(sdom(v), v)$. Modifying the link-eval structure to replace maximum by minimum and to return arcs (or, better, vertices) rather than costs is straightforward. The algorithm of section 7 thus performs Step 2 in $O(n)$ time on a pointer machine. (The number of queries is $O(n)$.)

It remains to implement Step 1, the computation of semidominators. Lengauer and Tarjan reduce this computation, also, to a problem of finding minima on tree paths, using the following lemma.

LEMMA 9.3 (see [43, Thm. 4]). *For any vertex w,*

$$sdom(w) = \min\big(\{w\} \cup \{nca(u,w) : (u,w) \in A\} \cup \{sdom(v) : \exists (u,w) \in A, nca(u,w) \stackrel{+}{\to} v \stackrel{*}{\to} u\}\big).$$

The lemma gives a recurrence for $sdom(w)$ in terms of $sdom(v)$ for $v > w$. The LT algorithm performs Step 1 by visiting the vertices in reverse preorder and using a link-eval structure to perform the computations needed to evaluate the recurrence.

Even though Step 1 is now reduced to computing minima on tree paths, we cannot use the MST verification algorithm directly for this purpose, because that algorithm answers the queries in an order incompatible with the requirements of the recurrence. Instead we develop an alternative strategy. For convenience we restate the problem, which allows us to simplify slightly the recurrence in Lemma 9.3. Suppose each vertex w has an integer *tag* $t(w)$ in the range $[1, n]$. The *extended tag* of a vertex w is defined to be

$$et(w) = \min\{t(v) : \text{there is a high path from } v \text{ to } w\}.$$

LEMMA 9.4. *If $t(w) = \min(\{w\} \cup \{v : (v,w) \in A\})$ for every vertex, then $sdom(w) = et(w)$ for every vertex.*

Proof. The proof is immediate from the definitions of semidominators and extended tags. □

We can easily compute the tag specified in Lemma 9.4 for every vertex in $O(m)$ time. Thus the problem of computing semidominators becomes that of computing extended tags.

Lemma 9.3 extends to give the following recurrence for extended tags.

LEMMA 9.5. *For any vertex w,*

$$et(w) = \min(\{t(w)\} \cup \{et(v) : \exists (u,w) \in A,\ nca(u,w) \stackrel{+}{\to} v \stackrel{*}{\to} u\}).$$

Proof. The proof is analogous to that of Lemma 9.3. Let x be the right side of the equation in the statement of the lemma. First we prove $et(w) \leq x$. If $x = t(w)$, $et(w) \leq x$ is immediate from the definition of $et(w)$. Suppose $x = et(v)$ for v such that $nca(u,w) \stackrel{+}{\to} v \stackrel{*}{\to} u$ and (u,w) in A. By the definition of $et(v)$, $et(v) = t(z)$ for some vertex z such that there is a high path from z to v. Extending this path by the tree path from v to u followed by the arc (u,w) gives a high path from z to w. Hence $et(w) \leq et(v) = x$.

Next we prove $x \leq et(w)$. Let z be a vertex such that $et(w) = t(z)$ and there is a high path from z to w (by the definition of the extended tags). If $z = w$, then $x \leq et(w)$ from the definition of x. If not, let (u,w) be the last arc on the high path from z to w. Let v be the first vertex along the high path such that $nca(u,w) \stackrel{+}{\to} v \stackrel{*}{\to} u$. Such a v exists since u is a candidate ($nca(u,w) \leq w < u$). We claim that the part of

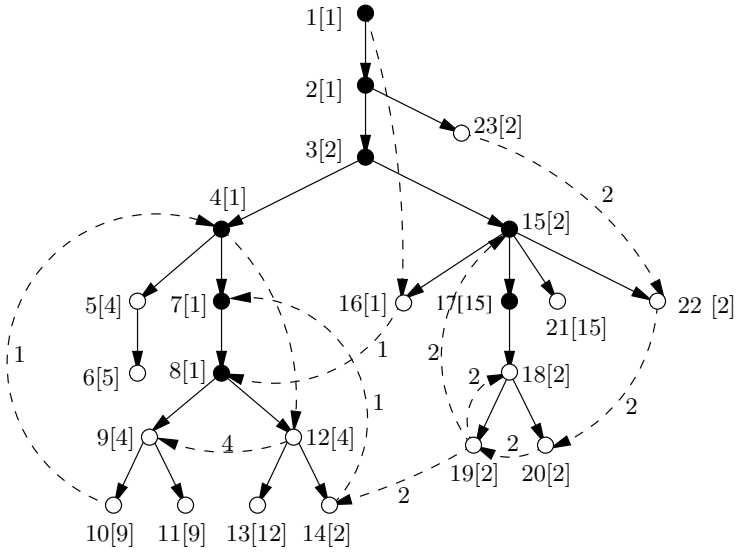

FIG. 9.1. *Extended tags and arc tags. The number inside each bracket is the extended tag of the corresponding vertex. The number on each arc is its tag; the arc tag of a tree or forward arc is infinite and not shown in the figure.*

the high path from z to v is itself a high path. Suppose to the contrary that this part contains a vertex less than v, and let y be the last such vertex. Then y must be an ancestor of v by Lemma 9.1, and since y is on a high path for w, $nca(u,w) \stackrel{+}{\rightarrow} y \stackrel{+}{\rightarrow} v$. This contradicts the choice of v. It follows that $et(v) \leq t(z)$; that is, $x \leq et(w)$. □

We introduce one more definition that simplifies some of our formulas and discussion. For an arc (u,w), the *arc tag* of (u,w) is

$$at(u,w) = \min\{et(v) : nca(u,w) \stackrel{+}{\rightarrow} v \stackrel{*}{\rightarrow} u\}$$

if this minimum is over a nonempty set, and infinity otherwise (when $nca(u,w) = u$). An example is shown in Figure 9.1. Using arc tags, the recurrence in Lemma 9.5 becomes

$$et(w) = \min(\{t(w)\} \cup \{at(u,w) : (u,w) \in A\}). \tag{9.1}$$

9.2. The interval forest. We could use (9.1) to compute extended tags just as the LT algorithm uses Lemma 9.3 to compute semidominators, but we seek a faster method. Note that there are two kinds of arcs (u,w) that must be handled: those such that u and w are unrelated (*cross arcs*), and those such that $w \stackrel{+}{\rightarrow} u$ (*back arcs*). (Arcs such that $u \stackrel{+}{\rightarrow} w$ do not contribute to the recurrence.) We apply different techniques to the cross arcs and the back arcs, which allows us to tease apart the intertwined computations implied by (9.1) and reorder them to apply our techniques.

To handle the back arcs, we use the interval forest discussed in section 8. Recall the following definitions. For each vertex w, the head $h(w)$ of w is the maximum vertex $u \neq w$ such that there is a path from w to u containing only descendants of u, if this maximum is over a nonempty set, and *null* otherwise. Lemma 9.1 implies that the constraint on u in the definition of $h(w)$ is equivalent to $u \stackrel{+}{\rightarrow} w$ and there is a high path from w to u. The heads define a forest H called the *interval forest*: $h(w)$

is the parent of w in H. The following lemma allows us to compute extended tags by computing arc tags only for the cross arcs and propagating minima up the interval forest.

LEMMA 9.6. *For any vertex w,*

$$et(w) = \min(\{t(v) : v \in H(w)\} \cup \{at(u,v) : (u,v) \in A,\ v \in H(w),\ u \notin D(w)\}).$$

Proof. Let x be the right side of the equation in the statement of the lemma. First we prove $et(w) \le x$. Let v be in $H(w)$. Since there is a high path from v to w, $et(w) \le t(v)$. Let (u,v) be in A such that v is in $H(w)$ but u is not in $D(w)$. Let y be a vertex of minimum $et(y)$ such that $nca(u,v) \stackrel{+}{\to} y \stackrel{*}{\to} u$, and let z be a vertex of minimum $t(z)$ such that there is a high path from z to y. Then there is a high path from z to y to u to v to w, which implies $et(w) \le t(z) = et(y) = at(u,v)$. We conclude that $et(w) \le x$.

Next we prove $x \le et(w)$. Let z be a vertex such that $et(w) = t(z)$ and there is a high path from z to w. If z is in $H(w)$, then $x \le t(z) = et(w)$. Suppose z is not in $H(w)$. Let (u,v) be the first arc along the high path from z to w such that v is in $H(w)$. Then u cannot be in $D(w)$, or it would be in $H(w)$, contradicting the choice of (u,v). Thus $nca(u,v) \stackrel{+}{\to} u$. Let y be the first vertex along the high path such that $nca(u,v) \stackrel{+}{\to} y \stackrel{*}{\to} u$. By Lemma 9.1, the part of the high path from z to y is itself a high path. Thus $x \le at(u,v) \le et(y) \le t(z) = et(w)$. ∎

COROLLARY 9.7. *For any vertex w,*

$$et(w) = \min(\{t(w)\} \cup \{et(v) : h(v) = w\} \cup \{at(v,w) : (v,w) \textit{ is a cross arc}\}).$$

Corollary 9.7 gives an alternative recursion for computing extended tags by processing the vertices in reverse preorder. Lemma 9.6 also allows us to compute extended tags for all the vertices on a tree path, given only arc tags for arcs starting to the right of the path.

9.3. Microtrees and left paths. As in section 6, we partition D into a set of bottom-level microtrees (the fringe), each containing fewer than $g = \log^{1/3} n$ vertices, and D' (the core), the remainder of D. We call a cross arc *small* if both its ends are in the same microtree, and *big* otherwise. We also partition D' into maximal paths as in section 8, but a particular set of maximal paths. Specifically, we partition D' into *left paths*, as follows: An arc $(p(v), v)$ of D' is a *left arc* if v is the smallest child of $p(v)$ in D'. A *left path* is a maximal sequence of left arcs. We can partition D into microtrees and left paths in $O(m)$ time during the DFS that defines D. If P is a left path, as in section 8 we denote by $top(P)$ and $bottom(P)$ the smallest and largest vertices on P, respectively. The importance of left paths is twofold. First, there are at most n/g of them. Second, if $(p(v), v)$ is a left arc, any child of $p(v)$ smaller than v must be in the fringe, not the core. That is, left paths have only microtrees descending on their left. Left paths serve in place of the *lines* of Georgiadis and Tarjan [31, 32]; left paths are catenations of those lines.

Our hypothetical plan for computing extended tags in linear time is to use a topological graph computation to handle the microtrees and a link-eval structure to compute arc tags for the big cross arcs. This plan does not quite work: computing extended tags is unlike the previous problems we have considered in that there is an interaction between the fringe and the core. In particular, we need at least some information about the small cross arcs in order to compute extended tags in the core, and information about the big cross arcs to compute extended tags in the fringe. For

the former computation we do not, however, need to compute arc tags for the small cross arcs: the recurrence in Lemma 9.6 expresses the extended tags of vertices in the core in terms only of tags of vertices and arc tags of big cross arcs. To handle the limited interaction between fringe and core, we use a two-pass strategy. During the first pass, we compute arc tags of big cross arcs and extended tags in the core while computing limited information in the fringe. In the second pass, we use the information computed in the first pass in a topological graph computation to compute extended tags in the fringe.

The information we need in the fringe is a set of values defined as follows. For a vertex w in a microtree $D(s)$, the *microtag* of w is

$$\begin{aligned} mt(w) = \min\big(&\{t(v) : \text{there is a path from } v \text{ to } w \text{ in } D(s)\} \cup \\ &\{at(u,v) : (u,v) \text{ is a cross arc}, v \in D(s), u \notin D(s), \\ &\qquad \text{and there is a path in } D(s) \text{ from } v \text{ to } w\}\big). \end{aligned}$$

Our microtags correspond to the *pushed external dominators* of Buchsbaum et al. [16] (also used by Georgiadis and Tarjan [31, 32]). The next lemma shows that when computing the arc tags of big cross arcs, we can use microtags in place of extended tags for fringe vertices; that is, we shall use microtag values in the link-eval structure when linking fringe vertices.

LEMMA 9.8. *Let w be a vertex in a microtree $D(s)$. Then*

$$\min\{et(v) : s \stackrel{*}{\to} v \stackrel{*}{\to} w\} = \min\{mt(v) : s \stackrel{*}{\to} v \stackrel{*}{\to} w\}.$$

Proof. Let x and y be the values of the left and right sides of the equation in the statement of the lemma, respectively. First we prove that $x \geq y$. Let v be a vertex such that $x = et(v)$ and $s \stackrel{*}{\to} v \stackrel{*}{\to} w$. Let z be a vertex such that $t(z) = et(v)$ and there is a high path from z to v. If z is in $D(s)$, then this high path is in $D(s)$, which implies that $x = t(z) \geq mt(v) \geq y$. Suppose on the other hand that z is not in $D(s)$. Let (p,q) be the last arc along the high path such that p is not in $D(s)$, and let z' be the first vertex along the high path such that $nca(p,q) \stackrel{+}{\to} z' \stackrel{*}{\to} p$. Note that (p,q) must be a cross arc, since p is not in $D(s)$ and is on a high path to v in $D(s)$. See Figure 9.2. As in the proof of Lemma 9.5, the part of the high path from z to z' is itself a high path, which implies $x = t(z) \geq et(z') \geq at(p,q) \geq mt(v) \geq y$.

Next we prove that $x \leq y$. Let v be a vertex such that $y = mt(v)$ and $s \stackrel{*}{\to} v \stackrel{*}{\to} w$. Suppose $mt(v) = t(z)$ for some z in $D(s)$ from which there is a path to v in $D(s)$. Let u be the first vertex on this path that is an ancestor of w. Then the path from z to u is a high path by Lemma 9.1 and the choice of u. Thus $x \leq et(u) \leq t(z) = y$. Suppose on the other hand that $mt(v) = at(p,q)$ for an arc (p,q) such that q but not p is in $D(s)$ and there is a path from q to v. Let u be the first vertex on this path that is an ancestor of w. By Lemma 9.1, the part of the path from q to u is a high path. Let z be a vertex such that $t(z) = at(p,q)$ and there is a high path from z to a vertex z' such that $nca(p,q) \stackrel{+}{\to} z' \stackrel{*}{\to} p$. See Figure 9.2. This path, together with the path $z' \stackrel{*}{\to} p$, the arc (p,q), and the high path from q to u, is a high path. Thus $x \leq et(u) \leq t(z) = at(p,q) = mt(v) = y$. ∎

To help compute extended tags during the first pass, we use a compressed interval forest H' in place of the interval forest H. Recall that in H', the parent $h'(v)$ of a vertex v is the nearest ancestor of v in H that is a core vertex. Forests H and H' are identical on the core; each subtree of H consisting of fringe vertices with a core root is compressed in H' to the root with all the fringe vertices as children. The use

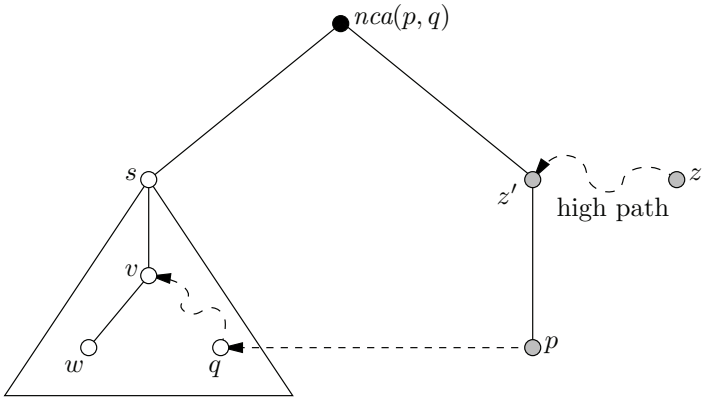

FIG. 9.2. *Proof of Lemma* 9.8. *Dashed curves represent graph paths. Solid edges represent tree paths. Each gray vertex can be in the core or in the fringe.*

of H' in place of H is an optimization only: we can build either H or H' in linear time using the algorithm of section 8, but, as noted in section 8, building H' instead of H avoids the use of topological graph computations on the microtrees and thus is simpler. The algorithm of section 8 builds H' by partitioning D into microtrees and maximal paths. We can use the set of left paths as the maximal paths, avoiding the need for two different partitions.

To compute extended tags in the core, we use the following corollary of Lemma 9.6.

COROLLARY 9.9. *If w is a core vertex*

$$\begin{aligned} et(w) = \min\big(&\{t(v) : v = w \text{ or } v \text{ is fringe with } h'(v) = w\} \cup \\ &\{et(v) : v \text{ is core with } h'(v) = w\} \cup \\ &\{at(u,v) : (u,v) \text{ is a big cross arc such that} \\ &\qquad v = w \text{ or } v \text{ is fringe with } h'(v) = w\}\big). \end{aligned}$$

The algorithm of Georgiadis and Tarjan [32] for computing dominators does not use H' explicitly, but it does do an incremental backward search using a stack to maintain strongly connected parts of lines, in effect doing a just-in-time computation of (part of) H'. Making this computation separate, as we have done, breaks the overall algorithm into smaller, easier-to-understand parts, which could be combined if desired.

9.4. Computation of arc tags. The heart of the algorithm is the computation of arc tags. We split each such computation into two parts, either of which can be void: a *top part*, which computes a minimum of extended tags over part or all of a left path, and a *bottom part*, which computes a minimum of extended tags of core vertices and microtags of fringe vertices using a sophisticated link-eval structure. Specifically, let (u, v) be a big cross arc. Let P be the left path containing $nca(u, v)$, and let Q be the intersection of P and the path $nca(u, v) \stackrel{+}{\rightarrow} u$. We denote the last vertex on Q by $mid(u, v)$. Note that Q can be nonempty (contain arcs) only if v is a fringe vertex. See Figure 9.3.

For a given left path P, we compute minima of extended tags for all such nonempty paths Q at the same time. We do not need to know any of these minima until all

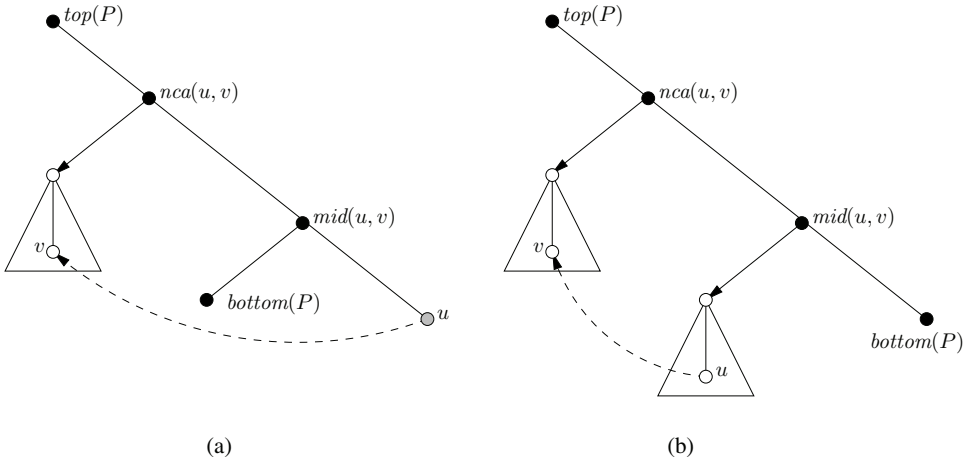

FIG. 9.3. *Examples of nonempty* $nca(u,v) \xrightarrow{+} mid(u,v)$ *paths.* (a) *Case* $u > bottom(P)$. (b) *Case* $u < bottom(P)$.

the extended tags for vertices on P have been computed. This allows us to compute the minima for such paths Q in arbitrary order. One way to compute these minima is to use the MST verification algorithm, as suggested above for doing Step 2 of the LT algorithm. In this application, however, the tree being verified is actually a path, and we can use an algorithm that is at least conceptually simpler, if not asymptotically faster. The problem we need to solve is that of computing minima for given subsequences of a sequence of numbers. This is the *range minimum query* (RMQ) problem [30]. This problem has a linear-time reduction [30] to an NCA problem on a Cartesian tree [65]. We can thus compute minima for paths Q by constructing the Cartesian tree and applying our NCA algorithm. Either method allows us to compute the top parts of arc tags in $O(m)$ time on a pointer machine.

To compute the bottom parts of arc tags, we use a sophisticated link-eval structure. We delay the links for arcs on a left path until the top of the left path is reached, and for arcs in a microtree until its root is reached. This allows us to establish a linear-time bound for all the link-eval operations using Lemma 4.6.

9.5. The first pass. We now have all the pieces necessary to describe the first pass of our algorithm for computing extended tags. Before the first pass, build the compressed interval forest H', compute $nca(u,v)$ for each big cross arc (u,v), and construct, for each core vertex w, the set of big cross arcs (u,v) with $nca(u,v) = w$. This takes $O(m)$ time on a pointer machine using the method of section 8: the NCAs are computed as part of the algorithm that builds H'. Each vertex v has a *computed tag* $ct(v)$ that is initialized to $t(v)$ and that decreases as the first pass proceeds, until $ct(v) = mt(v)$ if v is fringe, or $ct(v) = et(v)$ if v is core. Each fringe vertex v also has an associated set of cross arcs, initially empty. For each fringe vertex v, if v has a parent in H' and $ct(h'(v)) > ct(v)$, replace $ct(h'(v))$ by $ct(v)$. Finally, initialize a sophisticated link-eval data structure with no edges and each vertex of G as a node.

The first pass visits each microtree once and each left path twice. The visits are in reverse preorder with respect to the roots of the microtrees and the top and bottom vertices of the left paths; the first visit to a left path corresponds to its bottom (largest) vertex, and the second visit to its top (smallest) vertex. Conceptually, envision a

reverse preorder traversal of D, with actions taken as described below whenever a microtree root or bottom or top vertex of a left path is visited.

When visiting a microtree $D(s)$, it will be true that, for each vertex v in $D(s)$,

$$et(v) \leq ct(v) \leq \min(\{t(v)\} \cup \{at(u,v) : (u,v) \in A,\ u \notin D(s)\}). \tag{9.2}$$

Compute microtags for all vertices in $D(s)$ by finding the strong components of the subgraph induced by the vertices in $D(s)$ and processing the strong components in topological order. To process a component, compute a microtag for the component, equal to the minimum of the $ct(\cdot)$ values for all vertices in the component and the microtags for all preceding components (those with an arc leading to the component). Then set $ct(v)$ for every vertex in the component equal to the computed microtag. The assigned value of $ct(v)$ must be $mt(v)$, assuming (9.2) holds. The time required for this computation is linear in the size of the subgraph induced by $D(s)$ [54]. Having computed microtags for $D(s)$, perform $link(p(v), v, ct(v))$ for every vertex in $D(s)$, in bottom-up order. Finally, for each cross arc (u, v) in the set of cross arcs of a vertex u in $D(s)$, set $ct(v) \leftarrow \min\{ct(v), eval(u)\}$, and then set $ct(h'(v)) \leftarrow \min\{ct(h'(v)), ct(v)\}$ if v has a parent in H'. Such computations happen here only for arcs (u, v) such that u is in a microtree hanging on the left of some left path. It will become clear later that, for such an arc, the top part of the evaluation of $at(u, v)$ gets done first, when the left path is processed. The $eval(u)$ operation does the bottom part of the evaluation, finishing the job. We describe below when these arcs are entered in the set associated with u.

When visiting a left path P for the first time, begin by visiting the vertices w of P in bottom-up order and setting $ct(h'(w)) \leftarrow \min\{ct(h'(w)), ct(w)\}$ if w has a parent in H'. Once these updates are completed, $ct(w) = et(w)$ for every vertex w on P. Then collect all the arcs (u, v) in the sets associated with the vertices on P, i.e., the arcs (u, v) such that $nca(u, v) \in P$. For each such arc (u, v), set $mid(u, v) \leftarrow p(root(micro(u)))$ if $u < bottom(P)$, and $mid(u, v) \leftarrow findroot(u)$ otherwise. The *findroot* operation in the latter case is an operation on the link-eval structure. Having computed all the *mid* values for all the cross arcs, evaluate the top parts of their arc tags, using either of the methods discussed in section 9.4. For each such arc (u, v) with computed arc tag top part x, do the following. If $u > bottom(v)$ (see Figure 9.3(a)), set $x \leftarrow \min\{x, eval(u)\}$; otherwise (see Figure 9.3(b)), add (u, v) to the set of cross arcs of u. In the former case, the $eval(u)$ operation computes the bottom part of the arc tag; in the latter case, the computation of the bottom part is done when the microtree containing u (which hangs to the left of P) is visited. In either case, set $ct(v) \leftarrow \min\{ct(v), x\}$, and then set $ct(h'(v)) \leftarrow \min\{ct(h'(v)), ct(v)\}$ if v is a fringe vertex with a parent in H'.

When visiting a left path P for the second time, perform $link(p(w), w, ct(w))$ for each vertex on P in bottom-up order, unless P is the last path, in which case the first pass is done.

Based on the results of the previous sections, it is straightforward (but tedious) to prove that this algorithm correctly computes extended tags. Note that the algorithm eagerly pushes $ct(\cdot)$ values up H', rather than lazily pulling them; the latter would require computing sets of children for H', whereas the former can be done using just parent pointers.

LEMMA 9.10. *The first pass takes $O(m)$ time on a pointer machine.*

Proof. The running time of all parts of the algorithm is linear based on previous results, except for the *findroot* and *eval* operations. To bound the time for these, we

apply Lemma 4.6 to the shadow subtrees built by the link operations. These subtrees are $\sqrt{2}$-balanced by Corollary 4.2 for linking-by-size and Corollary 4.4 for linking-by-rank. Mark the parents (in D) of the tops of all the left paths. This marks at most $n/g = n/\log^{1/3} n$ vertices. We claim that $k = 5$ satisfies the hypothesis of the lemma.

We need to use details of the link implementation, for which we refer the reader to section 4.3 for linking-by-size and section 4.4 for linking-by-rank. The links occur in batches with no intermixed *findroot* or *eval* operations, one batch per microtree and one batch per left path. Let v be any vertex. We count the number of times the subroot of the shadow subtree containing v can change, as the result of a batch of links, before v is in a subtree containing a marked node. Let $v_0 = v, v_1, v_2, \ldots$ be the successive roots of the shadow trees containing v. The subroot of the shadow subtree containing v can change only as the result of a batch of links that include the current v_i as one of the vertices being linked. Suppose v is fringe. The first batch of links to include v_0 is the one for $\mathit{micro}(v)$. This batch of links makes $p(\mathit{root}(\mathit{micro}(v)))$ the root of the tree containing v; that is, $v_1 = p(\mathit{root}(\mathit{micro}(v)))$. The next batches of links that include v_1 are those for other microtrees whose roots are children of v_1 in D. Such a batch does not change the root of the tree containing v but can change the subroot of the subtree containing v, making it equal to v_1. Once such links are done, the only remaining batch of links that includes v_1 is the one for the left path P_1 containing v_1. This batch makes $v_2 = p(\mathit{top}(P_1))$, which means that the shadow tree containing v (but not necessarily the shadow subtree containing v) has a marked vertex. The next batches of links that include v_2 are those for microtrees whose roots are children of v_2 in D. Such a batch cannot change the root of the tree containing v, but it can change the subroot of the subtree containing v, making it equal to v_2, which is marked. Otherwise, the next (and last) batch of links that includes v_2 is the one for the left path P_2 containing v_2. This batch makes $v_3 = p(\mathit{top}(P_2))$.

Now v is either in the subtree rooted at v_3, and hence in a subtree with a marked vertex, or it is a shadow descendant of v_2, which is no longer the root of the shadow tree containing v. No subsequent link can change the root of the subtree containing v without putting v and v_2, which is marked, in the same subtree. Tracing through the analysis above, we see that the subroot of the shadow subtree containing a fringe vertex v can change at most four times before v is in a subtree with a marked vertex. If v is a core vertex, the last part of the same analysis applies: the first batch of links that can change either the root of the tree containing v or the subroot of the subtree containing v is the one for the left path containing v; the subroot of the subtree containing v can change at most twice before v is in a subtree with a marked vertex. The shadow parent of vertex v can change at most once before the root of the shadow subtree containing v changes. Thus the shadow parent of v can change at most five times before v is in a shadow subtree with a marked vertex. This verifies the claim. With $k = 5$ and $\ell \le n/\log^{1/3} n$, Lemma 4.6 gives a bound of $O(m)$ on the time for the *findroot* and *eval* operations. □

9.6. The second pass. Having computed extended tags for all core vertices, we compute extended tags for all fringe vertices by using a topological graph computation on the microtrees. In the first pass, just before a microtree $D(s)$ is processed, each vertex v in $D(s)$ has $ct(v) = \min(\{t(v)\} \cup \{at(u, v) : (u, v) \in A,\ u \notin D(s)\})$. It follows that if we compute extended tags within the subgraph induced by the vertices of $D(s)$, using these $ct(\cdot)$ values as the initial tags, we will obtain the correct extended tags for the vertices in $D(s)$ with respect to the original tags in the entire graph. The $ct(\cdot)$ values are in the range $[1, n]$, but we can map them to the range $[1, g]$ by sorting all the

$ct(\cdot)$ values using a pointer-based radix sort, extracting a sorted list of $ct(\cdot)$ values for each subproblem, and mapping each such sorted list to $[1, g]$. To do this on a pointer machine, we need to maintain a singly linked master list of length n, whose nodes correspond to the integers 1 through n, and store with each integer a pointer to its corresponding position in the master list, and we need to track such pointers through the entire running of the algorithm. We assume that each input tag is given along with a corresponding pointer into the master list. For the special case of computing semidominators, we construct the master list and the corresponding pointers as we perform the DFS and number the vertices. The only manipulations of vertex numbers are comparisons, so it is easy to track these pointers through the entire computation.

Once the tags are mapped to $[1, g]$, the computation of extended tags on the microtrees is a topological graph computation, which we perform using the method described in section 6. With the choice $g = \log^{1/3} n$, the second pass requires $O(m)$ time on a pointer machine.

Combining all the parts of the algorithm, we obtain the following theorem.

THEOREM 9.11. *Finding immediate dominators takes $O(m)$ time on a pointer machine.*

9.7. An alternative method for Step 2. We conclude our discussion of dominators by sketching an alternative method for performing Step 2 (computing relative dominators) that does some of the work in the second pass of Step 1 and then uses a simplification of the algorithm for the first pass of Step 1 to do the rest.

For a microtree $D(s)$, the $ct(\cdot)$ values of its vertices just before $D(s)$ is processed provide enough information not only to compute the semidominators of each of its vertices but also to compute the relative dominator of each vertex v such that $sdom(v)$ is in $D(s)$. This we can do as part of the topological graph computation that forms the second pass of Step 1. The remaining part of Step 2 is to compute $rdom(v) = \text{argmin}\{sdom(u) : sdom(v) \stackrel{+}{\rightarrow} u \stackrel{*}{\rightarrow} v\}$ for each vertex v with $sdom(v)$ in the core. We can do this by running a simplified version of the first pass of Step 1. We modify the link-eval structure so that an *eval* returns a vertex of minimum value, rather than the value itself. We compute the relative dominators in the same way that pass 1 of Step 1 computes the arc tags of big cross arcs, but without using the interval tree H' and without using NCAs. We begin by storing each pair $(sdom(v), v)$ with $sdom(v)$. Then we perform $link(p(v), v, sdom(v))$ for every fringe vertex v, in reverse preorder. Finally, we process each left path P, in reverse preorder with respect to $bottom(P)$. To process a left path P, we collect all the pairs (u, v) stored with its vertices. For each such pair, we set $mid(u, v) \leftarrow findroot(v)$. We evaluate each top part from u to $mid(u, v)$ using an NCA computation on a derived Cartesian tree as discussed in section 9.4, modified to return a candidate relative dominator $rd(u, v)$ for each pair. For each pair we set $rdom(v) \leftarrow \text{argmin}\{sdom(eval(v)), sdom(rd(u, v))\}$. Finally, we perform $link(p(v), v, sdom(v))$ for every vertex on P in reverse preorder, unless P is the last path, in which case we are done. This method for doing Step 2 takes $O(n)$ time.

This approach also leads to an alternative algorithm for MST verification, as mentioned in section 7.5, which avoids the use of the Borůvka tree as an intermediate step, replacing it with NCA computations on Cartesian trees derived from the paths of a partition of the core of the original tree T into maximal paths. We must still do verification within microtrees, but these are microtrees of the original tree rather than of the Borůvka tree.

9.8. Remarks. From the definition of microtags we have that for any w in a microtree $D(s)$, $mt(w) \leq mt(v)$ for any $s \stackrel{*}{\rightarrow} v \stackrel{*}{\rightarrow} w$. This inequality implies that the

eval function need only operate on the core tree. The algorithms of Buchsbaum et al. [16] and Georgiadis and Tarjan [31, 32] rely on this fact but also require a hybrid link-eval structure for the evaluation of path minima on the core. Lemma 4.6 allows us to use a standard (simpler) link-eval structure that can include the fringe, which also yields a more uniform treatment of the core and fringe vertices.

Our dominators algorithm uses the linear-time offline NCA algorithm for two subproblems: interval analysis and range minimum queries. Georgiadis [31] observed that a refined partition of the core tree into unary paths of size $O(g)$ enables us to use trivial algorithms to compute NCAs; topological graph computations are still required, but they are performed on Cartesian trees corresponding to each unary path.

10. Component trees. Our final application is a tree problem, unusual in that it seems to require partitioning all of the given tree, rather than just the bottom part, into microtrees.

10.1. Kruskal trees. The Borůvka tree discussed in section 7 represents the connected components that are formed as Borůvka's MST algorithm is run. We can define the analogous concept for other MST algorithms. For example, the *Kruskal tree* is the tree whose nodes are the connected components formed as Kruskal's MST algorithm [42] is run. Kruskal's algorithm starts with all vertices in singleton components and examines the edges in increasing order by weight, adding an edge to the MST being built, and combining the two corresponding components when the edge has ends in two different components. The Kruskal tree K is binary, with one node per component, whose children are the components combined to form the given component. Each leaf of K is a vertex of the original graph; each nonleaf node is a nonsingleton component. See Figure 10.1.

Even if the given graph is a tree, constructing the Kruskal tree is equivalent to sorting the edges by weight, because the Kruskal tree for a star (a tree of diameter two) contains enough information to sort the edges. If we are given the edges in order by weight, however, the problem of constructing the Kruskal tree becomes more interesting. We shall develop an $O(n)$-time, pointer-machine algorithm to build the Kruskal tree K of a tree T, given a list of the edges of T in order by weight.

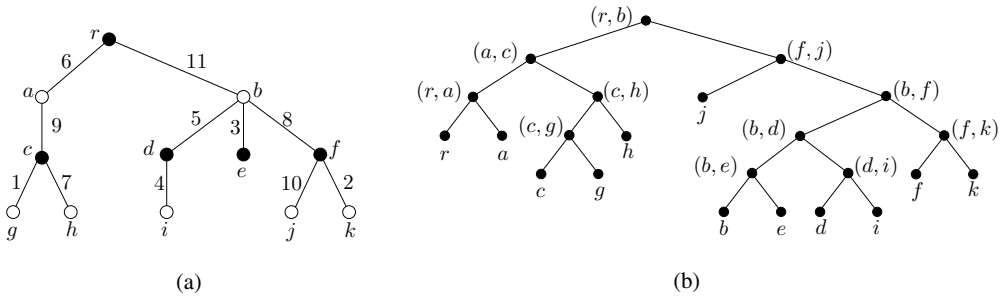

FIG. 10.1. (a) *The input weighted tree T; the filled nodes are subtree roots when T is partitioned with $g = 3$.* (b) *The Kruskal tree K of T. Leaves correspond to the nodes of T; internal nodes correspond to edges of T.*

10.2. Bottom-up construction of a Kruskal tree. It is straightforward to build K bottom-up using a DSU structure whose nodes are the nodes of T and whose sets are the node sets of the current components. As the algorithm proceeds, each designated node of a set stores the node of K corresponding to the set. Root T at an

arbitrary vertex; let $p(v)$ denote the parent of v in the rooted tree. Initialize a DSU structure with each node in a singleton set, storing itself (a leaf of K). Process the edges (now arcs) in the given order. To process an arc $(p(v), v)$, let $u = \mathit{find}(p(v))$. Add a new node x to K, whose two children are the nodes stored at u and v. Store x at u, and perform $\mathit{unite}(u, v)$. (For example, in Figure 10.1, the node corresponding to (f, j) is stored at b.)

This algorithm runs in $O(n\alpha(n, n))$ time on a pointer machine; only the finds take nonlinear time. Although it builds K bottom-up, it does not process T bottom-up but in the given arc order. As in sections 7–9, we thus cannot directly apply the method of section 6 to reduce the running time to linear. On the other hand, if we generalize the DSU structure to allow *unite* operations to have arbitrary nodes, rather than just designated nodes, as parameters, and we replace each $\mathit{unite}(u, v)$ operation in the algorithm by $\mathit{unite}(p(v), v)$, then the (unordered) set of unions is known in advance, because the unions correspond to the arcs of T. As Thorup [62] observed in the context of solving an equivalent problem (see section 10.4), this means that the algorithm runs in linear time on a RAM if the linear-time DSU algorithm of Gabow and Tarjan [29] is used.

Not only are the unions not bottom-up on T, but also there is no obvious way to transform the problem into one on a balanced tree as in section 7. Instead, we partition all of T into microtrees and do a topological graph computation to precompute the answers to finds within the microtrees. Once these answers are known, running the algorithm to build K takes $O(n)$ time. Number the arcs of T from 1 through $n - 1$ in the given order. For any nonroot vertex v, let $\mathit{num}(v)$ be the number of $(p(v), v)$; let $\mathit{num}(v) = \infty$ if v is the root. For any nonroot vertex v, let $f(v)$ be the node returned by $\mathit{find}(p(v))$ in the algorithm that builds K. (For example, in Figure 10.1, $f(j) = b$.) Then $f(v)$ is the nearest ancestor u of v that has $\mathit{num}(u) > \mathit{num}(v)$. We will precompute $f(v)$ if v and $f(v)$ are in the same microtree.

10.3. Linear-time construction. Let $g = n/\log^{1/3} n$. Partition all of T into microtrees, each of size at most g, using the method of Dixon, Rauch, and Tarjan [22], slightly modified. Visit the nodes of T in a bottom-up order, computing, for each node v, a size $s(v)$ and possibly marking v as a subtree root. The value of $s(v)$ is the number of descendants w of v such that no node on the path from v to w is marked. When visiting v, set $s(v) \leftarrow 1 + \sum\{s(w) : w \text{ is a child of } v\}$. If $s(v) > g$, mark every child of v and set $s(v)$ to 1. Every marked node v determines a microtree whose nodes are the descendants w of v such that v is the only marked node on the path from v to w. The construction guarantees that every microtree contains at most g nodes. It also guarantees that there are at most n/g parents of marked nodes, since, for each such parent, the set of microtrees rooted at its children contains at least g nodes. Partitioning T into microtrees takes $O(n)$ time.

To precompute the answers to finds in the microtrees, begin by initializing $f(v) \leftarrow \mathit{null}$ for every nonroot node v. Then use a pointer-based radix sort to renumber the nodes in each microtree consecutively from 1 up to at most g in an order consistent with their original numbers (given by *num*). This does not affect the answers to the finds for any vertex whose answer is in the same microtree. To do the pointer-based radix sort, build a master list of nodes representing the numbers 1 through n, and use pointers to these nodes in lieu of the actual numbers. For each microtree, build a similar master list of nodes representing the numbers 1 through the number of nodes in the microtree, and use pointers to these nodes in lieu of numbers. Now the problem of answering the finds within microtrees is actually a topological graph computation

as defined in section 5, and with $g = n/\log^{1/3} n$ it can be done in $O(n)$ time by Theorem 5.2. This computation gives a nonnull value $f(v)$ for every vertex v such that v and $f(v)$ are in the same microtree.

Having precomputed the answers to some of the finds, we run the algorithm that builds K, but using the precomputed answers. Specifically, to process an arc $(p(v), v)$, let $u = f(v)$ if $f(v) \neq null$, $u = find(p(v))$ otherwise. Then proceed as in section 10.2.

THEOREM 10.1. *Suppose that the edges of a weighted tree T are given in order by weight. Then the Kruskal tree of T can be built in $O(n)$ time on a pointer machine.*

Proof. The algorithm runs on a pointer machine; the running time is $O(n)$ except for the time to do the finds. We bound the time for the finds by applying Lemma 4.6 to the tree built by the parent assignments done by the unite operations. Mark every parent of a microtree root. This marks at most n/g nodes. If an operation $find(p(v))$ is actually done, because its answer is not precomputed, $f(v)$ and v are in different microtrees. The union operations are such that if x and y are in the same set and x is an ancestor of y, every vertex on the tree path from x to y is also in the same set. Thus when $find(p(v))$ is done, $f(v)$, $p(v)$, and $p(root(micro(v)))$ are all in the same set. Since $p(root(micro(v)))$ is marked, this find occurs in a set with a marked node. We conclude that Lemma 4.6 applies with $k = 1$, giving an $O(n)$ time bound for the finds that are not precomputed. □

We do not know whether there is a way to build K in linear time using only bottom-level microtrees. If there is, it is likely to be considerably more complicated than the algorithm we have proposed.

10.4. Compressed Kruskal trees. We can generalize the Kruskal tree to allow equal-weight edges: when adding edges, we add all edges of the same weight at the same time and add a node to the Kruskal tree for every new component so formed, whose children are the components connected together to form it. The resulting component tree is not necessarily binary. Thorup [62] and Pettie and Ramachandran [48] have used such a compressed Kruskal tree in shortest path algorithms. Given a tree and a partition of its edges into equal-weight groups, ordered by weight, we can construct the generalized Kruskal tree in linear time on a pointer machine as follows: Break ties in weight arbitrarily. Build the Kruskal tree, labeling each component node with the group of the edge that formed it. Contract into a single node each connected set of nodes labeled with the same group. The last step is easy to do in $O(n)$ time.

11. Concluding remarks. We have presented linear-time pointer-machine algorithms for six tree and graph problems, all of which have in common the need to evaluate a function defined on paths in a tree. Linear time is optimal and matches the previous bound for RAM algorithms for these problems; our algorithms improve previous pointer-machine algorithms by an inverse-Ackermann-function factor. Our improvements rely mainly on three new ideas: refined analysis of path compression when the compressions favor certain nodes; radix sorting to group isomorphic small subproblems; and careful partitioning of the tree corresponding to the original problem into a collection of microtrees and maximal paths, as appropriate to the particular application.

Our algorithms are simpler than the previous linear-time algorithms. Indeed, our approach provides the first linear-time dominators algorithm that could feasibly be implemented at all: the linear-time algorithm of Alstrup et al. [10] requires Q-heaps [26], implying an impossibly-large constant factor. Buchsbaum et al. implemented their original dominators algorithm [16], of which our algorithm is an improvement,

and presented experimental results demonstrating low constant factors, though the simpler LT algorithm was faster. Georgiadis, Tarjan, and Werneck [34] report more recent experiments with algorithms for finding dominators, with results that vary depending on input size and complexity.

Our methods are sufficiently simple and general that we expect them to have additional applications, which remain to be discovered.

Note Added in Proof. We recently discovered how to avoid the need for NCAs in Tarjan's algorithm for interval analysis described in section 8. The resulting simplified algorithm builds the sets $R(u)$ and finds heads during a single DFS of the flowgraph. It maintains the same DSU structure as in section 8. Initially $R(u)$ is empty for every vertex u. The algorithm is as follows. When the DFS retreats along an arc (x, y), add x to $R(\mathit{find}(y))$. When the DFS visits a vertex u in postorder, while $R(u)$ is nonempty delete a vertex x from $R(u)$, let $v \leftarrow \mathit{find}(x)$, and if $v \neq u$, set $h(v) \leftarrow u$, set $R(u) \leftarrow R(u) \cup R(v)$, and do $\mathit{unite}(u, v)$. This is the computation done by the original algorithm after it does insertions into R-sets for arcs (x, y) such that $\mathit{nca}(x, y) = u$. If the original algorithm processes the vertices in postorder with respect to the DFS, then the R-sets of the descendants of a vertex u just after the insertions for arcs (x, y) such that $\mathit{nca}(x, y) = u$ are exactly the same as they are in the simplified algorithm just before it processes u; the simplified algorithm has done these insertions earlier. It follows that the simplified algorithm is correct.

This idea extends to the linear-time algorithm for interval analysis in section 8, which can be simplified further by making the maximal path partition of the core rightmost instead of arbitrary. Then the computation of heads for vertices whose heads are in the core can be done in a single DFS that also generates the path partition. Here are the details of this computation. Do set unions to combine the vertices in each strong component of the fringe into a single set. Redo the DFS that generated the tree D used to define the fringe and the core, but do the following. Initialize the current path P, the stack S, and all sets $R(u)$ to be empty. When retreating along an arc (x, y), if $\mathit{find}(y)$ is not on P add x to $R(\mathit{find}(y))$; if (x, y) is a tree arc and y is in the core add y to P and push y onto S. (If $\mathit{find}(y)$ is on P when retreating along (x, y), (x, y) is a forward arc; as noted in section 8, heads do not depend on such arcs.) When visiting a core vertex u in postorder, while $R(u)$ is nonempty do the following. Delete a vertex x from $R(u)$. Let $v \leftarrow \mathit{find}(x)$. If v is not on P, set $h(v) \leftarrow u$, set $R(u) \leftarrow R(u) \cup R(v)$, and do $\mathit{unite}(u, v)$. If v is on P and v is no less than the top vertex on S, pop from S each vertex w less than or equal to v, set $h(w) \leftarrow u$, and set $R(u) \leftarrow R(u) \cup R(w)$. When advancing along a tree arc (x, y), do $\mathit{unite}(h(v), v)$ for each vertex on P but not on S, and then empty P and S. It is straightforward to show that this algorithm is correct. An analysis like that in section 8 shows that the algorithm runs in linear time.

Acknowledgment. We thank Stephen Alstrup and Amos Fiat for some pointers to previous work.

REFERENCES

[1] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol.* II: *Compiling*, Prentice–Hall, Englewood Cliffs, NJ, 1972.

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

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *On finding lowest common ancestors in trees*, SIAM J. Comput., 5 (1976), pp. 115–132.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison–Wesley, Reading, MA, 1986.
[5] S. Allesina and A. Bodini, *Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions*, J. Theoret. Biol., 230 (2004), pp. 351–358.
[6] S. Allesina, A. Bodini, and C. Bondavalli, *Secondary extinctions in ecological networks: Bottlenecks unveiled*, Ecological Modelling, 194 (2006), pp. 150–161.
[7] S. Alstrup and M. Thorup, *Optimal algorithms for finding nearest common ancestors in dynamic trees*, J. Algorithms, 35 (2000), pp. 169–188.
[8] S. Alstrup, J. P. Secher, and M. Spork, *Optimal on-line decremental connectivity in trees*, Inform. Process. Lett., 64 (1997), pp. 161–164.
[9] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe, *Nearest common ancestors: A survey and a new distributed algorithm*, in Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architecture, 2002, pp. 258–264.
[10] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, *Dominators in linear time*, SIAM J. Comput., 28 (1999), pp. 2117–2132.
[11] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana, *Fault equivalence identification using redundancy information and static and dynamic extraction*, in Proceedings of the 19th IEEE VLSI Test Symposium, 2001, pp. 124–130.
[12] M. A. Bender and M. Farach-Colton, *The LCA problem revisited*, in Proceedings of the 4th Latin American Symposium on Theoretical Informatics, Lecture Notes in Comput. Sci. 1776, Springer-Verlag, New York, 2000, pp. 88–94.
[13] O. Borůvka, *O jistém problému minimálním*, Práce Moravské Přírodovědecké Společnosti v Brně (Acta Societ. Science. Natur. Moravicae), 3 (1926), pp. 37–58.
[14] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan, *Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues*, SIAM J. Comput., 24 (1995), pp. 1190–1206.
[15] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook, *Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators*, in Proceedings of the 30th ACM Symposium on Theory of Computing, 1998, pp. 279–288.
[16] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook, *A new, simpler linear-time dominators algorithm*, ACM Trans. Programming Languages and Systems, 20 (1998), pp. 1265–1296; *Corrigendum*, 27 (2005), pp. 383–387.
[17] J. Cai and R. Paige, *Using multiset discrimination to solve language processing problems without hashing*, Theoret. Comput. Sci., 145 (1995), pp. 189–228.
[18] B. Chazelle, *A minimum spanning tree algorithm with inverse-Ackermann type complexity*, J. ACM, 47 (2000), pp. 1028–1047.
[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Programming Languages and Systems, 13 (1991), pp. 451–490.
[20] M. B. Dillencourt, H. Samet, and M. Tamminen, *A general approach to connected-component labeling for arbitrary image representations*, J. ACM, 39 (1992), pp. 253–280.
[21] B. Dixon, M. Rauch, and R. E. Tarjan, *Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time*, Tech. report CS-TR-289-90, Department of Computer Science, Princeton University, Princeton, NJ, 1990.
[22] B. Dixon, M. Rauch, and R. E. Tarjan, *Verification and sensitivity analysis of minimum spanning trees in linear time*, SIAM J. Comput., 21 (1992), pp. 1184–1192.
[23] S. Even and Y. Shiloach, *An on-line edge deletion problem*, J. ACM, 28 (1981), pp. 1–4.
[24] J. Ferrante, K. Ottenstein, and J. Warren, *The program dependency graph and its uses in optimization*, ACM Trans. Programming Languages and Systems, 9 (1987), pp. 319–349.
[25] C. Fiorio and J. Gustedt, *Two linear time union-find strategies for image processing*, Theoret. Comput. Sci., 154 (1996), pp. 165–181.
[26] M. L. Fredman and D. E. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Comput. System Sci., 48 (1994), pp. 533–551.
[27] H. N. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*, in Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1990, pp. 434–443.
[28] H. N. Gabow, *Path-based depth-first search for strong and biconnected components*, Inform. Process. Lett., 74 (2000), pp. 107–114.
[29] H. N. Gabow and R. E. Tarjan, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209–221.
[30] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, *Scaling and related techniques for geometry problems*, in Proceedings of the 16th ACM Symposium on Theory of Computing, 1984, pp. 135–143.

[31] L. Georgiadis, *Linear-Time Algorithms for Dominators and Related Problems*, Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, NJ, 2005.
[32] L. Georgiadis and R. E. Tarjan, *Finding dominators revisited*, in Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2004, pp. 862–871.
[33] L. Georgiadis and R. E. Tarjan, *Dominator tree verification and vertex-disjoint paths*, in Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2005, pp. 433–442.
[34] L. Georgiadis, R. E. Tarjan, and R. F. Werneck, *Finding dominators in practice*, J. Graph Algorithms Appl., 10 (2006), pp. 69–94.
[35] R. L. Graham and P. Hell, *On the history of the minimum spanning tree problem*, Ann. History Comput., 7 (1985), pp. 43–57.
[36] J. Gustedt, *Efficient union-find for planar graphs and other sparse graph classes*, Theoret. Comput. Sci., 203 (1998), pp. 123–141.
[37] D. Harel, *A linear time algorithm for finding dominators in flow graphs and related problems*, in Proceedings of the 17th ACM Symposium on Theory of Computing, 1985, pp. 185–194.
[38] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.
[39] D. R. Karger, P. N. Klein, and R. E. Tarjan, *A randomized linear-time algorithm to find minimum spanning trees*, J. ACM, 42 (1995), pp. 321–328.
[40] V. King, *A simpler minimum spanning tree verification algorithm*, Algorithmica, 18 (1997), pp. 263–270.
[41] J. Komlós, *Linear verification for spanning trees*, Combinatorica, 5 (1985), pp. 57–65.
[42] J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 53–57.
[43] T. Lengauer and R. E. Tarjan, *A fast algorithm for finding dominators in a flowgraph*, ACM Trans. Programming Languages and Systems, 1 (1979), pp. 121–141.
[44] M. Loebl and J. Nešetřil, *Linearity and unprovability of set union problem strategies* I. *Linearity of strong postorder*, J. Algorithms, 23 (1997), pp. 207–220.
[45] E. S. Lorry and V. W. Medlock, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.
[46] J. M. Lucas, *Postorder disjoint set union is linear*, SIAM J. Comput., 19 (1990), pp. 868–882.
[47] M. Pătraşcu and E. D. Demaine, *Logarithmic lower bounds in the cell-probe model*, SIAM J. Comput., 35 (2006), pp. 932–963.
[48] S. Pettie and V. Ramachandran, *Computing shortest paths with comparisons and additions*, in Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2002, pp. 267–276.
[49] S. Pettie and V. Ramachandran, *An optimal minimum spanning tree algorithm*, J. ACM, 49 (2002), pp. 16–34.
[50] P. W. Purdom and E. F. Moore, *Algorithm* 430*: Immediate predominators in a directed graph*, Comm. ACM, 15 (1972), pp. 777–778.
[51] L. Quesada, P. Van Roy, Y. Deville, and R. Collet, *Using dominators for solving constrained path problems*, in Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages, Lecture Notes in Comput. Sci. 3819, Springer-Verlag, New York, 2006, pp. 73–87.
[52] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.
[53] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.
[54] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.
[55] R. E. Tarjan, *Finding dominators in directed graphs*, SIAM J. Comput., 3 (1974), pp. 62–89.
[56] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. ACM, 22 (1975), pp. 215–225.
[57] R. E. Tarjan, *Edge-disjoint spanning trees and depth-first search*, Acta Inform., 6 (1976), pp. 171–185.
[58] R. E. Tarjan, *Applications of path compression on balanced trees*, J. ACM, 26 (1979), pp. 690–715.
[59] R. E. Tarjan, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.
[60] R. E. Tarjan, *Testing flow graph reducibility*, J. Comput. System Sci., 9 (1994), pp. 355–365.
[61] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, J. ACM, 31 (1984), pp. 245–281.

[62] M. Thorup, *Undirected single-source shortest paths with positive integer weights in linear time*, J. ACM, 46 (1999), pp. 362–394.
[63] A. K. Tsakalides and J. van Leeuwen, *An Optimal Pointer Machine Algorithm for Finding Nearest Common Ancestors*, Tech. report RUU-CS-88-17, Department of Computer Science, University of Utrecht, Utrecht, Germany, 1988.
[64] J. van Leeuwen, *Finding Lowest Common Ancestors in Less Than Logarithmic Time*, unpublished report, 1976.
[65] J. Vuillemin, *A unifying look at data structures*, Comm. ACM, 23 (1980), pp. 229–239.