A New, Simpler Linear-Time Dominators Algorithm

ADAM L. BUCHSBAUM, HAIM KAPLAN, ANNE ROGERS, and JEFFERY R. WESTBROOK
AT&T Labs

We present a new linear-time algorithm to find the immediate dominators of all vertices in a flowgraph. Our algorithm is simpler than previous linear-time algorithms: rather than employ complicated data structures, we combine the use of microtrees and memoization with new observations on a restricted class of path compressions. We have implemented our algorithm, and we report experimental results that show that the constant factors are low. Compared to the standard, slightly superlinear algorithm of Lengauer and Tarjan, which has much less overhead, our algorithm runs 10–20% slower on real flowgraphs of reasonable size and only a few percent slower on very large flowgraphs.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; E.1 [**Data Structures**]: Graphs and Networks; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Compilers, dominators, flowgraphs, microtrees, path compression

1. INTRODUCTION

We consider the problem of finding the immediate dominators of vertices in a flowgraph. A *flowgraph* is a directed graph $G = (V, A, r)$ with a distinguished start vertex $r = \mathit{root}(G) \in V$, such that there is a path from r to each vertex in V. Vertex w *dominates* vertex v if every path from r to v includes w; w is the *immediate dominator* (IDOM) *of* v, denoted $w = \mathit{idom}(v)$, if (1) w dominates v and (2) every other vertex x that dominates v also dominates w. Every vertex in a flowgraph has a unique immediate dominator [Aho and Ullman 1972; Lorry and Medlock 1969].

Finding immediate dominators in a flowgraph is an elegant problem in graph theory, with applications in global flow analysis and program optimization [Aho and Ullman 1972; Cytron et al. 1991; Ferrante et al. 1987; Lorry and Medlock 1969]. Lorry and Medlock [1969] introduced an $O(n^4)$-time algorithm, where $n = |V|$ and $m = |A|$, to find all the immediate dominators in a flowgraph. Successive improve-

Some of this material was presented at the *Thirtieth ACM Symposium on the Theory of Computing*, 1998.
Authors' address: AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932; {alb; hkl; amr; jeffw}@research.att.com.

ments to this time bound were achieved [Aho and Ullman 1972; Purdom and Moore 1972; Tarjan 1974], culminating in Lengauer and Tarjan's [1979] $O(m\alpha(m,n))$-time algorithm; α is the standard functional inverse of the Ackermann function and grows extremely slowly with m and n [Tarjan and van Leeuwen 1984]. Lengauer and Tarjan [1979] report experimental results showing that their algorithm outperforms all previous dominators algorithms for flowgraph sizes that appear in practice.

Reducing the asymptotic time complexity of finding dominators to $O(n+m)$ is an interesting theoretical exercise. Furthermore, various results in compiler theory rely on the existence of a linear-time dominators algorithm; Pingali and Bilardi [1997] give an example and further references. Harel [1985] claimed a linear-time dominators algorithm, but careful examination of his abstract reveals problems with his arguments. Alstrup et al. [1997] detail some of the problems with Harel's approach and offer a linear-time algorithm that employs powerful data structures based on bit manipulation to resolve these problems. While they achieve a linear-time dominators algorithm, their reliance on sophisticated data structures adds sufficient overhead to make any implementation impractical.

We present a new linear-time dominators algorithm, which is simpler than that of Alstrup et al. [1997]. Our algorithm requires no complicated data structures: we use only depth-first search, the fast union-find data structure [Tarjan and van Leeuwen 1984], topological sort, and memoization. We have implemented our algorithm, and we report experimental results, which show that, even with the extra overhead needed to achieve linear time, our constant factors are low. Ours is the first implementation of a linear-time dominators algorithm.

The rest of this article is organized as follows. Section 2 outlines Lengauer and Tarjan's approach. Section 3 gives a broad overview of our algorithm and differentiates it from previous work. Section 4 presents our algorithm in detail, and Section 5 analyzes its running time. Section 6 presents our new path-compression result, on which the analysis relies. Section 7 describes our implementation, and Section 8 reports experimental results. We conclude in Section 9.

2. THE LENGAUER-TARJAN ALGORITHM

Here we outline the Lengauer and Tarjan (LT) approach [Lengauer and Tarjan 1979] at a high level, to provide some details needed by our algorithm. Appel [1998] provides a thorough description of the LT algorithm.

Let $G = (V, A, r)$ be an input flowgraph with n vertices and m arcs. Let D be a depth-first search (DFS) tree of G, rooted at r. We sometimes refer to a vertex x by its DFS number; in particular, $x < y$ means that x's DFS number is less than y's. Let $w \overset{*}{\to} v$ denote that w is an ancestor (not necessarily proper) of v in D; $w \overset{*}{\to} v$ can also denote the actual tree path. Similarly, $w \overset{+}{\to} v$ denotes that w is a proper ancestor of v in D and can represent the corresponding path. For any tree T, let $p_T(v)$ be the parent of v in T, and let $\mathrm{nca}_T(u,v)$ be the nearest common ancestor of u and v in T. We will drop the subscripts and write $p(v)$ and $\mathrm{nca}(u,v)$ when the context resolves any ambiguity.

Let $P = (u = x_0, x_1, \ldots, x_{k-1}, x_k = v)$ be a path in G. Lengauer and Tarjan [1979] define P to be a *semidominator path* (abbreviated SDOM *path*) if $x_i > v$, $1 \le i \le k-1$. An SDOM path from u to v thus avoids all tree vertices between u and

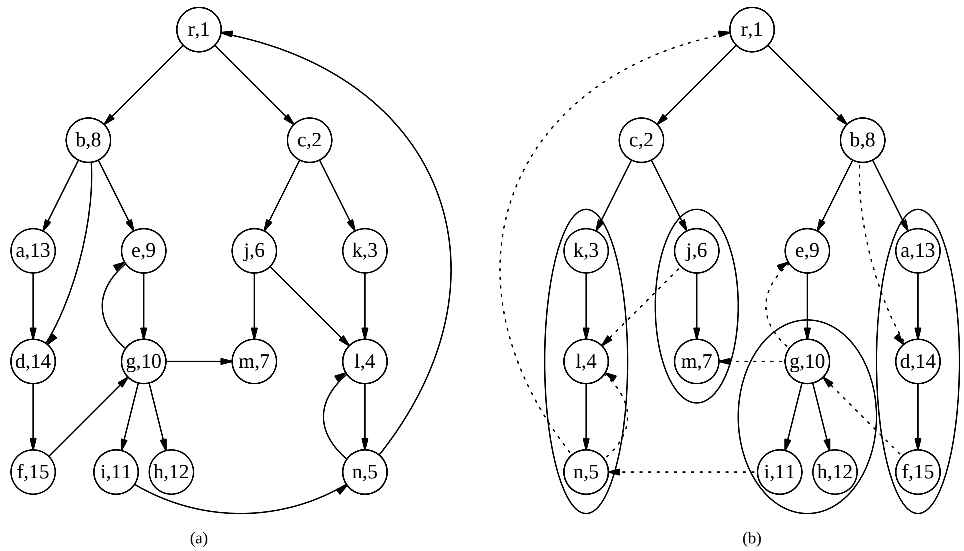

Fig. 1. (a) A flowgraph G with root r. Vertex labels are augmented with their DFS numbers. (b) A DFS tree D of G. Solid arcs are tree arcs; dotted arcs are nontree arcs. Breaking D into microtrees of size no more than 3 results in four nontrivial microtrees, rooted at k, j, g, and a; the vertices of each nontrivial microtree are encircled.

v. The *semidominator* (SEMI) *of vertex* v is

$$semi(v) = \min\{u \mid \text{there is an SDOM path from } u \text{ to } v\}.$$

For example, consider vertex g in DFS tree D in Figure 1(a). The DFS number of g is 10. Paths (e, g), (f, g), (d, f, g), (a, d, f, g), (b, d, f, g), and (b, a, d, f, g) are all the SDOM paths to g. Since b has the least DFS number over the initial vertices on these paths, $semi(g) = 8$.

To compute semidominators, Lengauer and Tarjan use an auxiliary *link-eval* data structure, which operates as follows. Let T be a tree with a real value associated with each vertex. We wish to maintain a forest F contained in the tree, subject to the following operations. (Initially F contains no arcs.)

—*link*(u): Add arc $(p_T(u), u)$ to F.

—*eval*(u): Let r be the root of the tree containing u in F. If $u = r$, return r. Otherwise, return any vertex $x \neq r$ of minimum value on the path $r \xrightarrow{*} u$.

Tarjan [1979a] shows how to implement link and eval using the standard disjoint set union data structure [Tarjan and van Leeuwen 1984]. Using linking by size and path compression, $n - 1$ links and m evals on an n-vertex tree T can be performed in $O(m\alpha(m, n) + n)$ time.

The LT algorithm traverses D in reverse DFS order, computing semidominators as follows. (Initially, $semi(v) \leftarrow v$ for all $v \in V$.)

For $v \in V$ in reverse DFS order **do**
    For $(w, v) \in A$ **do**

$u \leftarrow eval(w)$
$semi(v) \leftarrow \min\{semi(u), semi(v)\}$
done
$link(v)$
done

It then computes the immediate dominator for each vertex, using semidominators and the following facts, which we will also use to design our algorithm.

LEMMA 2.1 (LT LEM. 1). *If $v \leq w$ then any path from v to w in G must contain a common ancestor of v and w in D.*

LEMMA 2.2 (LT LEM. 4). *For any vertex $v \neq r$, $idom(v) \xrightarrow{*} semi(v)$.*

LEMMA 2.3 (LT LEM. 5). *Let vertices w, v satisfy $w \xrightarrow{*} v$. Then $w \xrightarrow{*} idom(v)$ or $idom(v) \xrightarrow{*} idom(w)$.*

LEMMA 2.4 (LT THM. 2). *Let $w \neq r$. Suppose every u for which $semi(w) \xrightarrow{+} u \xrightarrow{*} w$ satisfies $semi(u) \geq semi(w)$. Then $idom(w) = semi(w)$.*

LEMMA 2.5 (LT THM. 3). *Let $w \neq r$, and let u be a vertex for which $semi(u)$ is minimum among vertices u satisfying $semi(w) \xrightarrow{+} u \xrightarrow{*} w$. Then $semi(u) \leq semi(w)$ and $idom(u) = idom(w)$.*

3. OUTLINE OF A LINEAR-TIME ALGORITHM

The links and evals used by the LT algorithm make it run in $O(m\alpha(m,n))$ time. We can eliminate the $\alpha(m,n)$ term by exploiting the sensitivity of α to relative differences in m and n. In particular, when m is slightly superlinear in n, e.g., $m/n = \Omega(\log^{(O(1))} n)$, $\alpha(m,n)$ becomes a constant [Tarjan and van Leeuwen 1984].[1] Our dominators algorithm proceeds roughly as follows:

(1) Compute a DFS tree D of G, and partition D into regions. We discuss the partitioning in detail in Section 3.2. For now, it suffices to consider that D is partitioned into a collection of small, vertex-disjoint regions, called *microtrees*. We consider separately the microtrees at the bottom of D—those that contain the leaves of D—from the microtrees comprising the interior, D', of D.
(2) For each vertex, determine whether its IDOM is in its microtree and, if so, determine the actual IDOM.
(3) For each vertex v such that $idom(v)$ is not in v's microtree
   (a) compute $idom(v)$ by applying the LT algorithm only to vertices in D' or
   (b) find an ancestor u of v such that $idom(v) = idom(u)$ and $idom(u)$ can be computed by applying the LT algorithm only to vertices in D'.

Partitioning D into microtrees serves two purposes. First, the subgraph induced by the microtree roots will achieve the ratio m/n necessary to reduce $\alpha(m,n)$ to a constant. Second, when the microtrees are small enough, the number of distinct microtrees will be small compared to $n + m$. We can thus perform simple computations on each microtree in $O(n + m)$ time overall, using precomputed tables or memoization to eliminate redundant computations.

[1] $\log^{(i)} n$ is the iterated log function: $\log^{(0)} n = n$, and, for $i > 0$, $\log^{(i)} n = \log\log^{(i-1)} n$.

3.1 Comparison to Previous Approaches

We contrast our use of these facts to previous approaches. Harel [1985] and Alstrup et al. [1997] apply the LT algorithm to all of D, using the microtree partitioning to speed links and evals. Harel [1985] divides the entire tree D into microtrees, all of which can contain more than one vertex, and performs links and evals as described by Lengauer and Tarjan [1979] on the tree D' induced by the microtree roots. Alstrup et al. [1997] simplify Harel's approach [Harel 1985] by restricting nonsingleton microtrees to the bottom of D, leaving an upper subtree, D', of singleton microtrees as we do. They then perform links and evals on D' using two novel data structures, as well as the Gabow-Tarjan linear-time disjoint set union result [Gabow and Tarjan 1985] and transformations to D'. Both algorithms use precomputed tables to process evals on the internal microtree vertices. This approach requires information regarding which vertices outside microtrees might dominate vertices inside microtrees, to derive efficient encodings needed by the table lookup technique. Harel [1985] presents a method to restrict the set of such outside dominator candidates. Alstrup et al. [1997] demonstrate deficiencies in Harel's arguments and correct these problems, using Fredman and Willard's Q-heaps [Fredman and Willard 1994] to manage the microtrees.

We apply the LT algorithm just to the upper portion, D', of D. We combine our partitioning scheme with a new path compression result to show that the LT algorithm runs in linear time on D'. Instead of processing links and evals on internal microtree vertices, we determine, using any simple dominators algorithm, whether the dominators of such vertices are internal to their microtrees, and if so we compute them directly, using memoization to eliminate redundant computation. We process those vertices with dominators outside their microtrees without performing evals on internal microtree vertices. Our approach obviates the need to determine outside dominator candidates for internal microtree vertices, eliminating the additional complexity Alstrup et al. require to manage this information.

We can thus summarize the key differences in the various approaches as follows. Harel [1985] and Alstrup et al. [1997] partition D into microtrees and apply the standard LT algorithm to all of D, using precomputed tables to speed the computation of the link-eval data structure in the microtrees. We also partition D into microtrees, but we apply the LT algorithm, with the link-eval data structure unchanged, only to one, big region of D and use memoization to speed the computation of dominators in the microtrees. In other words, Harel [1985] and Alstrup et al. [1997] take a purely data structures approach, leaving the LT algorithm unchanged but employing sophisticated new data structures to improve its running time. We modify the LT algorithm so that, although it becomes slightly more complicated, simple and standard data structures suffice to implement it.

A minor difference in the two approaches regards the use of tables. Harel [1985] and Alstrup et al. [1997] precompute the answers to all possible queries on microtrees and then use table lookup to answer the queries during the actual dominators computation. We build the corresponding table incrementally using memoization, computing only the entries actually needed by the given instance. The two approaches have identical asymptotic time complexities, but memoization tends to outperform a priori tabulation in practice, because the former does not compute answers to queries that will never be needed.

3.2 Microtrees

Consider the following procedure that marks certain vertices in D. The parameter g is given, and initially all vertices are unmarked.

> **For** x in D in reverse DFS order **do**
> $S(x) \leftarrow 1 + \sum_{y \text{ child of } x} S(y)$
>   **If** $S(x) > g$ **then**
>   Mark all children of x
>   **endif**
> **done**
> Mark *root*(D)

For any vertex v, let *nma*(v) be the nearest marked ancestor (not necessarily proper) of v. The NMA function partitions the vertices of D into *microtrees* as follows. Let v be a marked vertex; $T_v = \{x \mid nma(x) = v\}$ is the microtree containing all vertices x such that v is the nearest marked ancestor of x. We say *root*$(T_v) = v$ is the *root of microtree* T_v. For any vertex x, *micro*(x) is the microtree containing x. See Figure 1.

For any v, if v has more than g descendents, all children of v are marked. Therefore, each microtree has size at most g. We call a microtree *nontrivial* if it contains a leaf of D. Only nontrivial microtrees can contain more than one vertex; these are the subtrees we process using memoization. The remaining microtrees, which we call *trivial*, are each composed of singleton, internal vertices of D; these vertices comprise the upper subtree, D', of D. Additionally, all the children of a vertex that forms a trivial microtree are themselves microtree roots. Call a vertex v that forms a trivial microtree *special* if each child of v is the root of a nontrivial microtree. (In Figure 1(b), c and e are special vertices.) If we were to remove the nontrivial microtrees from D, these special vertices would be the leaves of the resulting tree. Since each special vertex has more than g descendents, and the descendents of any two special vertices form disjoint sets, there are $O(n/g)$ special vertices.

We note that Alstrup et al. [1997] define microtrees only where they include leaves of D (our nontrivial microtrees), whereas our definition makes every vertex a member of some microtree. We could adopt the Alstrup et al. [1997] definition, but defining a microtree for each vertex allows more uniformity in our discussion, particularly in the statements and proofs of our lemmas and theorems.

Gabow and Tarjan [1985] pioneered the use of microtrees to produce a linear-time disjoint set union algorithm for the special case when the unions are known in advance. In that work, microtrees are combined into *microsets*, and precomputed tables are generated for the microsets. Dixon and Tarjan [1997] introduce the idea of processing microtrees only at the bottom of a tree.

3.3 Path Definitions

Let $P = (u = x_0, x_1, \ldots, x_{k-1}, x_k = v)$ be a path in G. We define P to be an *external dominator path* (abbreviated XDOM *path*) if P is an SDOM path and $x_0, \ldots, x_{k-1} \notin micro(v)$. An external dominator path is simply a semidominator path that resides wholly outside the microtree of the target vertex (until it hits the target vertex). The *external dominator of vertex v* is

$$xdom(v) = \min \{\{v\} \cup \{u \mid \text{there is an XDOM path from } u \text{ to } v\}\}.$$

In particular, for any vertex v that forms a singleton microtree, $xdom(v) = semi(v)$.

We define P to be a *pushed external dominator path* (abbreviated PXDOM *path*) if $x_i \geq root(micro(v))$, $1 \leq i \leq k-1$. Since nontrivial microtrees occur only at the bottom of D, a PXDOM path to v cannot exit and reenter $micro(v)$: to do so would require traversing a back arc to a proper ancestor of $root(micro(v))$. Therefore, a PXDOM path to v is (a) an XDOM path to some vertex $x \in micro(v)$ catenated with (b) an x-to-v path inside $micro(v)$. Either (a) or (b) may be the null path. The *pushed external dominator of vertex v* is

$$pxdom(v) = \min\{u \mid \text{there is a PXDOM path from } u \text{ to } v\}.$$

Note that $pxdom(v) \notin micro(v)$: since the arc $(p_D(root(micro(v))), root(micro(v)))$ catenated with the tree path $root(micro(v)) \xrightarrow{*} v$ forms a PXDOM path to v, we have that $pxdom(v) \leq p_D(root(micro(v)))$.

For example, consider vertices l and h in DFS tree D in Figure 1(b). The DFS number of l is 4. The path $P = (r, b, e, g, i, n, l)$ is an SDOM path from r to l, and so $semi(l) = 1$. Since $n \in micro(l)$, P is not an XDOM path. Path (c, j, l) is an XDOM path from c to l, and no XDOM path exists from r to l, so $xdom(l) = 2$. P is a PXDOM path, however: (r, b, e, g, i, n) is an XDOM path from r to $n \in micro(l)$, and (n, l) is a path internal to $micro(l)$. Thus, $pxdom(l) = 1 = semi(l)$. Continuing, the DFS number of h is 12. The only SDOM path to h is (g, h), and so $semi(h) = 10$. Path (b, d, f, g, h) is a PXDOM path to h, however, so $pxdom(h) = 8 \neq semi(h)$. In general, for any vertex, its SEMI, XDOM, and PXDOM values need not match.

We use the following lemmas. Note the similarity of Lemma 3.2 to Lemma 2.2.

LEMMA 3.1. *For any vertex v that forms a singleton microtree, $pxdom(v) = semi(v)$.*

PROOF. Let $u = pxdom(v)$, and let $P = (u = x_0, x_1, \ldots, x_{k-1}, x_k = v)$ be a PXDOM path from u to v. If v forms a singleton microtree, then $root(micro(v)) = v$, and so, by definition of PXDOM, $x_i \geq v$ for $1 \leq i < k$. Without loss of generality, however, since u is the minimum vertex from which there is a PXDOM path to v, we can assume that $x_i \neq v$ for $1 \leq i < k$. Therefore P is a semidominator path, so $semi(v) \leq u$. Any semidominator path, however, is a PXDOM path, so in fact $semi(v) = u$. □

LEMMA 3.2. *$idom(v) \notin micro(v) \Longrightarrow idom(v) \xrightarrow{*} pxdom(v)$.*

PROOF. Let $u = pxdom(v)$. As observed above, $u \notin micro(v)$. By definition of PXDOM, there is a path from u to v that avoids all vertices (other than u) on the tree path $u \xrightarrow{*} p_D(root(micro(v)))$. Therefore, if $idom(v) \notin micro(v)$, $idom(v)$ cannot lie on that tree path. □

In the next section, we give the details of our algorithm.

4. DETAILS OF OUR LINEAR-TIME ALGORITHM

At a high level, we can abstract our algorithm as follows:

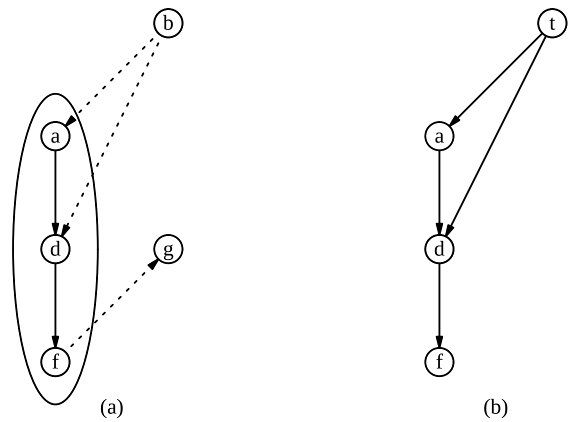

Fig. 2. (a) The microtree T, consisting of vertices a, d, and f from Figure 1(b), as well as incident arcs external to T. (b) The induced graph $aug(T)$.

(1) Using memoization to reduce running time, determine for each vertex v if $idom(v) \in micro(v)$ and, if so, the actual value $idom(v)$.
(2) Use the LT algorithm to compute IDOMS for all v such that $idom(v) \notin micro(v)$.

The remainder of this section provides the details behind our approach. For clarity, we describe as separate phases the resolution of the $idom(v) \in micro(v)$ question, the computation of $pxdom(v)$, and the overall algorithm to compute $idom(v)$. We discuss in Section 7 how to unite these phases into one traversal of D.

4.1 Computing Internal Dominators

We begin by showing how to determine whether $idom(v) \in micro(v)$ and, if it is, how to find the actual value $idom(v)$. For vertex v that comprises a singleton microtree, our decision is trivial: $idom(v) \notin micro(v)$. For a nonsingleton microtree T, we define the following augmented graph. Let $G(T)$ be the subgraph of G induced by vertices of T. Let $aug(T)$ be the graph $G(T)$ plus the following:

(1) A vertex t, which we call the *root of* $aug(T)$, or $root(aug(T))$.
(2) An arc (t, v) for each $v \in T$ such that there exists an arc $(u, v) \in A$ for some $u \notin T$. We call these *blue arcs*.

Note that there is a blue arc $(t, root(T))$. Vertex t represents the contraction of $G \setminus T$, ignoring all arcs that exit T. See Figure 2. We use the augmented graphs to capture the intuition that removing arcs that exit a microtree[2] does not change the dominator relationship.

We define the *internal immediate dominator* (IIDOM) *of vertex* x, $iidom(x)$, to be the immediate dominator of x in $aug(micro(x))$. We show that if $iidom(x) \in micro(x)$ (i.e., $iidom(x) \neq t$) then $idom(x) = iidom(x)$, and, conversely, that if $iidom(x) \notin micro(x)$ (i.e., $iidom(x) = t$) then $idom(x) \notin micro(x)$. Computing IIDOMS using memoization on $aug(micro(v))$ thus yields a fast procedure to deter-

[2]Arc (u, v) *exits* $micro(u)$ if $v \notin micro(u)$.

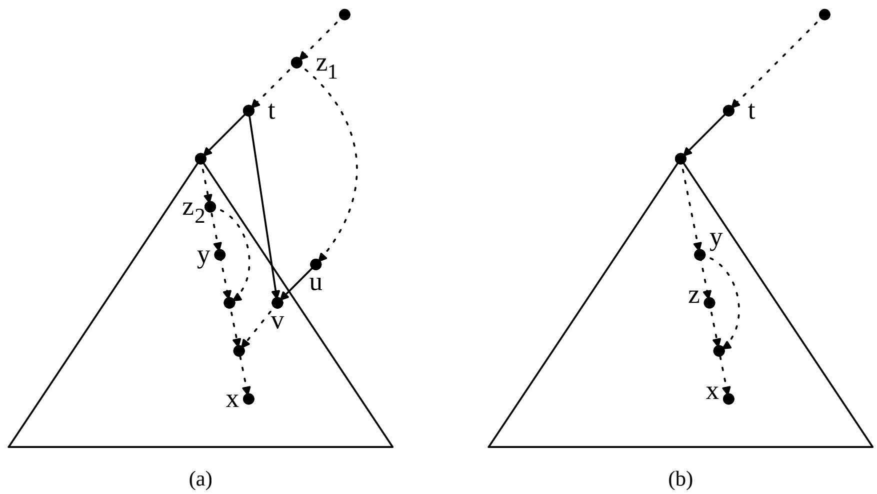

Fig. 3. $aug(micro(x))$, plus incident external arcs/paths from G. Solid lines are arcs; dotted lines are paths. (a) The case $y = iidom(x)$, $z = idom(x)$, $y \neq t$, and $z < y$. If $z \notin micro(x)$, e.g., $z = z_1$ in the figure, then there is a path from t to x in $aug(T)$, using blue arc (t, v), that avoids y. If $z \in micro(x)$, e.g., $z = z_2$ in the figure, then there is a path internal to $micro(T)$ from z to x that avoids y. Either case contradicts the assumption that $y = iidom(x)$. (b) Similar case, but $y < z$. There is a path P in $aug(T)$ from y to x, avoiding z. P contains no blue arcs, so it is a path in G, contradicting that $z = idom(x)$.

mine whether or not $idom(v) \in micro(v)$ for any v. We give the details of the memoization procedure below.

Lemma 4.1. $iidom(x) \neq root(aug(micro(x))) \Longrightarrow idom(x) = iidom(x)$.

Proof. Let $T = micro(x)$ and $t = root(aug(T))$. Let $y = iidom(x)$ and $z = idom(x)$ such that $y \neq t$ and $y \neq z$. If $z < y$, then in the full graph G, there exists a path P from z to x that avoids y. We use P to demonstrate a path P' in $aug(T)$ from some $z' \in \{t, z\}$ to x that avoids y, contradicting the assumption that $y = iidom(x)$. Let (u, v) be the last arc on P such that $u \notin aug(T)$. If there is no such arc then $P' = P$ yields an immediate contradiction. Otherwise, arc (u, v) induces blue arc $(t, v) \in aug(T)$. This arc together with a subpath of P from v to x provides path P'. See Figure 3(a).

On the other hand, if $y < z$, then there is a path P in $aug(T)$ from y to x that avoids z. By hypothesis, $y \neq t$, so P contains no blue arcs. (There are no arcs into t, and so $t \notin P$.) Therefore, P is also a path in G, contradicting that $z = idom(x)$. See Figure 3(b). $\square$

Lemma 4.2. $iidom(x) = root(aug(micro(x))) \Longrightarrow idom(x) \notin micro(x)$.

Proof. Let $T = micro(x)$ and $t = root(aug(T))$. Suppose $idom(x) = z \in micro(x)$ but $iidom(x) = t$. Then there is a path P in $aug(T)$ from t to x that avoids z. If P contains no blue arcs, then it is a path in the original graph, contradicting the claim that $z = idom(x)$. If P contains blue arc (t, v) for some v, then in G there is an arc (u, v) for some $u \notin T$. The tree path $root(G) \xrightarrow{*} u$ catenated with the arc (u, v) and the subpath in P from v to x gives a path in G to x that avoids

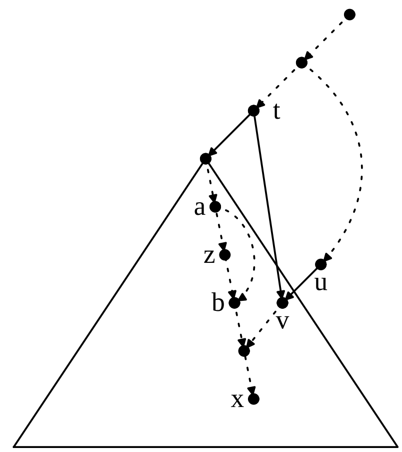

Fig. 4. $aug(micro(x))$, plus incident external arcs/paths from G. Solid lines are arcs; dotted lines are paths. Case in which $idom(x) = z \in T$ but $iidom(x) = t$. There is a path P in $aug(T)$ from t to x, avoiding z. If P contains no blue arcs (follows the path from a to b around z), this contradicts that $z = idom(x)$, for P exists in G. If P contains blue arc (t, v), then there is a path in G to x, using the arc (u, v), where $u \notin T$. Again, P avoids z, contradicting $z = idom(x)$.

z, again giving a contradiction. See Figure 4. $\square$

We memoize the computation of $iidom(v)$ as follows. The first time we compute the internal immediate dominators for some augmented graph $aug(T)$, we store the results in a table, I, indexed by graph $aug(T)$ and vertex v. We encode $aug(T)$ by a bit string corresponding to its adjacency matrix represented in row-major order. To compute this bit string, we traverse $aug(T)$ in DFS order, assigning DFS value one to the root of $aug(T)$ and using the DFS values as vertex identifiers; we refer to this as the *canonical encoding of* $aug(T)$.

If a subsequent microtree T' has an augmented graph that is isomorphic to $aug(T)$, their canonical encodings will be identical, so we can simply look up the IIDOM values for $aug(T')$ in table I. This obviates having to recompute the IIDOMS for $aug(T')$: we simply map the IIDOM values stored in table I, which are relative to the canonical encoding of $aug(T')$, to the current instantiation of $aug(T')$. (Vertex x in $aug(T')$ corresponds to vertex $x - root(aug(T')) + 1$ in the canonical encoding of $aug(T')$.)

4.2 Computing Pushed External Dominators

We now prove that the following procedure labels vertices with their PXDOMS. As we will show, this process allows us to avoid performing links and evals within nontrivial microtrees.

Initially, $label(v) = v$ for all $v \in D$, and we use a link-eval data structure with $label(v)$ as the value for vertex v. As we will see, by Theorem 4.4, $label(v) = pxdom(v)$ when v becomes linked. The link-eval values are thus PXDOMS.

Let $EN(v) = \{x \mid (x, v) \in A, x \notin micro(v)\}$, the external neighbors of v, be the vertices outside $micro(v)$ with arcs to v. The procedure processes the microtrees T

in reverse DFS order.

(1) For $v \in T$:
   (a) $B = \{\mathit{label}(x) \mid x \in \mathit{EN}(v)\}$;
   (b) $C = \{\mathit{label}(\mathit{eval}(p_D(\mathit{root}(\mathit{micro}(x))))) \mid x \in \mathit{EN}(v), x \not\xrightarrow{*} v\}$;
   (c) $\mathit{label}(v) \leftarrow \min(\{v\} \cup B \cup C)$.

   Lemma 4.3 proves that this labels v with $\mathit{xdom}(v)$.

(2) For $v \in T$, let $Y(v)$ be the set of all vertices in T from which there is a path to v consisting only of arcs in $G(T)$. Set $\mathit{label}(v) \leftarrow \min_{y \in Y(v)}\{\mathit{label}(y)\}$. We call this *pushing* to v. (Pushing can be done by computing the strongly connected components of $G(T)$ and processing them in topological order.) Theorem 4.4 proves that pushing labels v with $\mathit{pxdom}(v)$.

(3) If T is a trivial microtree, then $\mathit{link}(v)$.

Due to the pushing in Step (2), PXDOM values are nonincreasing along paths from the microtree root. This allows us to perform evals only on parents of microtree roots: the PXDOM pushing effectively substitutes for the evals on vertices inside the microtrees.

To prove that the above procedure correctly labels vertices in a microtree T, we assume by induction that the procedure has already labeled by their PXDOMS all vertices in all trees preceding T in reverse DFS order. The base case is vacuously true.

LEMMA 4.3. *After Step (1), $\mathit{label}(x) = \mathit{xdom}(x)$ for $x \in T$.*

PROOF. Let $w = \mathit{xdom}(x)$. We show that (1) $\mathit{label}(x) \le w$ and (2) $\mathit{label}(x) \ge w$.

(1) Consider the XDOM path P from w to x. Let $y \notin \mathit{micro}(x)$ be the last vertex on P before x. Let z be the least vertex excluding w that P touches on the tree path $w \xrightarrow{*} y$; $z \ge \mathrm{nca}(y, x)$, or else P is not an XDOM path. The prefix P' of P from w to z is a semidominator path. Otherwise, there exists some $u \ne w$ on P' such that $u < z$; by Lemma 2.1, P' contains a common ancestor of u and z, contradicting the assertion that z is the least vertex in P on the tree path $w \xrightarrow{*} y$. Therefore, $\mathit{pxdom}(z) \le \mathit{semi}(z) \le w$. By induction, $\mathit{label}(z) \le w$. If $z \in \mathit{micro}(y)$, then $\mathit{label}(z)$ got pushed to y, and thus $\mathit{label}(y) \le w$. (Note that $\mathit{label}(y) \in B$ in Step (1).) If $z \notin \mathit{micro}(y)$, then C in Step (1) contains some value no greater than $\mathit{label}(z)$, due to the previous links via Step (3). In either case the label considered for x via the (y, x) arc is no greater than $\mathit{label}(z) \le w$. See Figure 5.

(2) Consider arc (y, x) such that $y \notin \mathit{micro}(x)$. Let $\mathit{label}(y) = w'$; by induction, $w' = \mathit{pxdom}(y)$, so there is a PXDOM path P from w' to y. P catenated with the arc (y, x) is an XDOM path. Similarly, for $z = \mathit{eval}(p_D(\mathit{root}(\mathit{micro}(y))))$, there is a PXDOM path P from $w' = \mathit{label}(z)$ to z. P catenated with the tree path $z \xrightarrow{*} y$ and arc (y, x) forms an XDOM path from w' to x. In either case, $w \le w'$. Figure 5 again demonstrates the potential paths. □

THEOREM 4.4. *After Step (2), $\mathit{label}(x) = \mathit{pxdom}(x)$ for $x \in T$.*

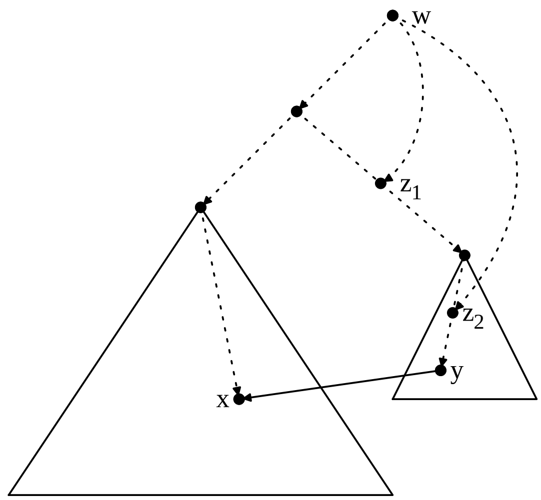

Fig. 5. Microtrees containing x and y, with incident external paths. Solid lines are arcs; dotted lines are paths. $pxdom(x) = w$. There is an SDOM path from w to some $z > \text{nca}(y, x)$; thus $pxdom(z) \leq w$. If $z \in micro(y)$ ($z = z_2$ in the figure), then $label(y) \leq w$. If $z \notin micro(y)$ ($z = z_1$ in the figure), then $label(eval(root(micro(y)))) \leq w$.

Proof. We argue analogously to the proof of Lemma 4.3. Let $w = pxdom(x)$; we show that (1) w is considered as a label for x via an internal pushing path and (2) for any w' so considered, there is a valid PXDOM path from w' to x.

(1) Consider the PXDOM path P from w to x. Let v be the first vertex on P inside T; $xdom(v) = w$. By Lemma 4.3, $label(v) = w$ after Step (1). During Step (2), w is pushed to x via the path from v to x.

(2) Consider any w' pushed to x. w' is an XDOM or PXDOM for some vertex $y \in T$. Since $y \in T \Longrightarrow y \geq root(T)$, there is a valid PXDOM path from w' to x. □

4.3 Computing Dominators

Using the information we computed in Sections 4.1 and 4.2, we now give an algorithm to compute immediate dominators. The algorithm proceeds like the LT algorithm; in fact, on the subtree of D induced by the trivial microtrees, it is exactly the LT algorithm. The algorithm relies on the following two lemmas:

Lemma 4.5. *For any v, there exists a $w \in micro(v)$ such that*

(1) $w \xrightarrow{*} v$;

(2) $pxdom(v) = pxdom(w)$:

(3) $pxdom(w) = semi(w)$;

(4) $iidom(w) = root(aug(micro(w)))$.

Proof. The proof proceeds as follows. We first find an appropriate vertex w on the tree path $root(micro(v)) \xrightarrow{*} v$. We show that $semi(w) = pxdom(v)$ and then

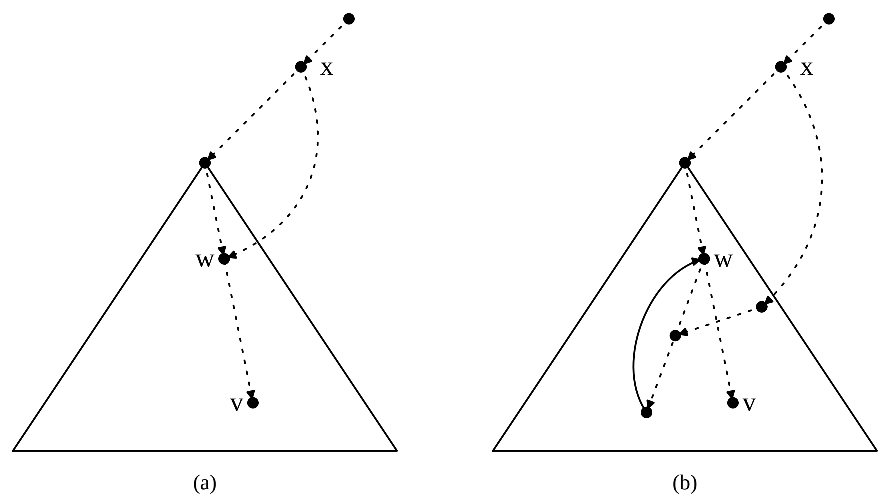

Fig. 6. The graph induced by $micro(v)$, plus incident external arcs/paths from G. Solid lines are arcs; dotted lines are paths. $pxdom(v) = x$; let P be the PXDOM path from x to v. w is the least vertex in P on the path from $root(micro(v))$ to v. (a) The prefix P' of P from x to w includes only vertices that are greater than v (except w). (b) P' includes descendents of w that are less than v and so must take a back arc to w. In either case, P' is an SDOM path from x to w, since w is the least vertex in P on the path from $root(micro(v))$ to v.

argue that $pxdom(w) = pxdom(v)$. This resolves postulates (1)–(3). Finally, we prove that $idom(w) \notin micro(x)$, which implies postulate (4).

Let $x = pxdom(v)$, and consider the PXDOM path P from x to v. Let w be the least vertex in P on the tree path $root(micro(v)) \xrightarrow{*} v$. We argue that the prefix P' of P from x to w is a semidominator path. If not, then there is some vertex $y \neq x$ on P' such that $y < w$. Since $w \leq v$, it must be that $y \in micro(v)$; otherwise, y violates the PXDOM path definition, since we only allow a $y < v$ on P if $y \in micro(v)$. By Lemma 2.1, the subpath of P' from y to w contains a common ancestor z of y and w. Since $y < w$, it must be that $z < w$. As with y, it must also be that $z \in micro(v)$, or else z violates the PXDOM path definition. This implies that z is on the tree path $root(micro(v)) \xrightarrow{*} v$, contradicting the assertion that w is the least such vertex on P. Therefore $semi(w) \leq x$. See Figure 6.

Now we argue that $semi(w) \geq x$. If not, there is a semidominator path P from some $y < x$ to w. P catenated with the tree path from w to v, however, forms a PXDOM path from y to v, contradicting the assumption that $pxdom(v) = x$.

Similarly, we argue that $pxdom(w) = x$. Since any semidominator path is also a PXDOM path, $pxdom(w) \leq x$. If there is a PXDOM path P from some $y < x$ to w, however, P catenated with the tree path from w to v is a PXDOM path that contradicts the assumption that $pxdom(v) = x$. Thus we have shown that $pxdom(v) = pxdom(w) = x$, and $pxdom(w) = semi(w)$.

By definition of PXDOM, $pxdom(w) < root(micro(w))$. Therefore, $pxdom(w) = semi(w)$ implies that $semi(w) \notin micro(w)$. By Lemma 2.2, therefore, $idom(w) \notin micro(w)$, and thus by Lemma 4.1, $iidom(w) = root(aug(micro(w)))$. □

LEMMA 4.6. *Let w, v be vertices in a microtree T such that*

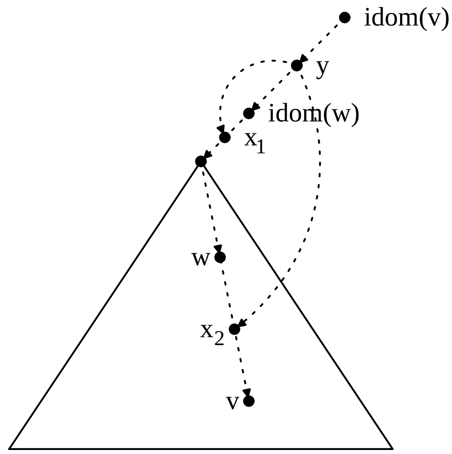

Fig. 7. The graph induced by $micro(v)$, plus incident external arcs/paths from G. Dotted lines are paths. If $idom(v) < idom(w)$, then there is an SDOM path from some $y < idom(w)$ to some $x > idom(w)$ such that there is a tree path from x to v. If x lies on the tree path from $idom(w)$ to w ($x = x_1$ in the figure), however, this contradicts the definition of $idom(w)$, and if x lies on the tree path from w to v ($x = x_2$ in the figure), this contradicts that $pxdom(v) = pxdom(w)$.

(1) $w \xrightarrow{*} v$,

(2) $pxdom(w) = pxdom(v)$, *and*

(3) $iidom(v) = iidom(w) = root(aug(T))$.

Then $idom(v) = idom(w)$.

Proof. Condition (3) and Lemma 4.2 imply that $idom(v), idom(w) \notin T$. In particular, $idom(v) < w$, so Lemma 2.3 implies that $idom(v) \leq idom(w)$. If $idom(v) < idom(w)$, then there is a path P from $idom(v)$ to v that avoids $idom(w)$. P must contain a semidominator subpath P' from some $y < idom(w)$ to some $x > idom(w)$ such that $x \xrightarrow{*} v$. x cannot lie on tree path $idom(w) \xrightarrow{+} w$, for this would contradict the definition of $idom(w)$. x cannot lie on tree path $w \xrightarrow{*} v$, for this would imply $pxdom(v) \leq y < pxdom(w)$. (By Lemma 3.2, $idom(w) \leq pxdom(w)$.) So no such P' can exist. See Figure 7. □

Lemmas 4.5 and 4.6 imply the following, which is formalized in the proof of Theorem 4.7. Consider a path in a microtree, from root to leaf. The vertices on the path are partitioned by PXDOM, with PXDOM values monotonically nonincreasing. Each vertex w at the top of a partition is such that $pxdom(w) = semi(w)$; furthermore, $idom(w) \notin micro(w)$. For another vertex v in the same partition as w, either $idom(v)$ is actually in the partition, or else $idom(v) = idom(w)$, outside the microtree. See Figure 8. That $pxdom(w) = semi(w)$ implies that our algorithm devolves into the LT algorithm on the upper subtree, D', of D consisting of trivial microtrees.

We can now compute immediate dominators by Algorithm `IDOM`, given in Figure 9. For each $v \in D$, `IDOM` either computes $idom(v)$ or determines a proper ancestor

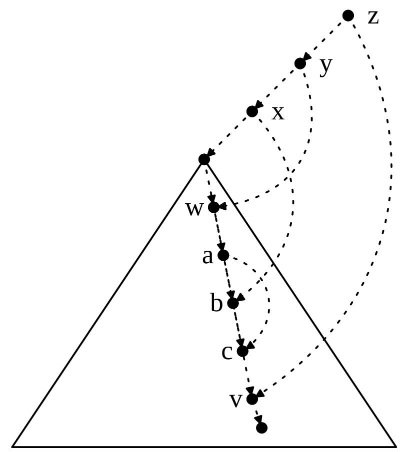

Fig. 8. A microtree with incident external paths. Dotted lines are paths. $pxdom(w) = y$, and $pxdom(v) = z$. All vertices on the tree path from w to $p(v)$ have PXDOM y; the path from x to b, a prefix of which is an XDOM path, does not affect the PXDOM values on the w-$p(v)$ part of the partition. The vertices in the partition need not share IDOMS, however. In this picture, $idom(c) = idom(b) = idom(w) = y$, but $idom(a) = p(a)$.

u of v such that $idom(v) = idom(u)$. We omit description of the straightforward postprocessing phase that resolves the latter identities. IDOM uses a second link-eval data structure, with $pxdom(v)$ as the value for vertex v; at the beginning of IDOM, no links have been done.

THEOREM 4.7. *Algorithm* IDOM *correctly assigns immediate dominators.*

PROOF. Lemma 4.1 shows that assigning $idom(v)$ to be $iidom(v)$ if $iidom(v) \in micro(v)$ is correct. Assume then that $iidom(v) \notin micro(v)$, and thus $idom(v) \notin micro(v)$ by Lemma 4.2.

Consider the processing of vertex v in bucket(u). Assume first that $pxdom(v) = semi(v) = u$. Let u' be the child of u on tree path $u \xrightarrow{+} v$. We claim that z is the vertex on tree path $u' \xrightarrow{*} v$ with minimum SEMI and that $pxdom(z) = semi(z)$. Assuming that this claim is true, if $pxdom(z) = u$, then by Lemma 2.4 $idom(v) = semi(v) = u$, and if $pxdom(z) < u$ then by Lemma 2.5 $idom(v) = idom(z)$.

Observe that for any $w \in micro(v)$ such that $w \xrightarrow{*} v$, $semi(v) = pxdom(v) \leq pxdom(w) \leq semi(w)$. Thus, if $u = p_D(root(micro(v)))$, then $u' = root(micro(v))$, $z = v$, and the claim holds. On the other hand, if $u \xrightarrow{+} p_D(root(micro(v)))$, then $z = eval(p_D(root(micro(v))))$ is the vertex on the tree path $P = u' \xrightarrow{*} p_D(root(micro(v)))$ of minimum PXDOM. The claim holds, since (1) $pxdom(u') \leq u = pxdom(v)$ and (2) $pxdom(y) = semi(y)$ for all $y \in P$ (by Lemma 3.1).

Consider the remaining case, when $pxdom(v) \neq semi(v)$. Lemma 4.5 shows that there exists a $w \in micro(v)$ such that $w \xrightarrow{*} v$, $iidom(w) = root(aug(micro(v)))$, $pxdom(w) = pxdom(v)$, and $pxdom(w) = semi(w)$. By hypothesis, $iidom(v) = root(aug(micro(v)))$, and so $idom(v) = idom(w)$ by Lemma 4.6. Since $pxdom(v) = pxdom(w)$ and $micro(v) = micro(w)$, v and w are both placed in the same bucket

```
Algorithm IDOM
    For v ∈ D in reverse DFS order do
        Process(v)
    done
    For u ∈ D such that {u} is a trivial microtree, in reverse DFS order do
        Process(bucket(u))
        link(u)
    done
Process(v)
    If iidom(v) ∈ micro(v) then
        idom(v) ← iidom(v)
    else
        add v to bucket(pxdom(v))
    endif
Process(bucket(u))
    For v ∈ bucket(u) do
        If u = p_D(root(micro(v))) then
            z ← v
        else
            z ← eval(p_D(root(micro(v))))
        endif
        If pxdom(z) = u then
            idom(v) ← u
        else
            idom(v) = idom(z)
        endif
    done
```

Fig. 9. Algorithm IDOM.

by IDOM. Therefore, IDOM does compute the same value for $idom(v)$ as for $idom(w)$, and by the previous argument, it computes the correct value for $idom(w)$. □

5. ANALYSIS

Here we analyze the running time of our algorithm. It should be clear that the generation of the initial DFS tree D and the division of D into microtrees can be performed in linear time, by the discussion in Section 3.2.

5.1 Computation of IIDOMS

Recall the memoized computation of IIDOMS described in Section 4.1. So that all the IIDOM computations run in linear time overall, the augmented graphs must be small enough so that (1) a unique description of each possible graph $aug(T)$ can be computed in $O(|aug(T)|)$ time and (2) all the immediate dominators for all possible augmented graphs are computable in linear time. (After computing immediate dominators for an augmented graph, future table lookups take constant time each.)

We require a description of $aug(T)$ to fit in one computer word, which we assume holds $\log n$ bits. Recall each microtree has no more than g vertices, for some parameter g. Thus, each augmented graph has no more than $g+1$ vertices. Without affecting the time bounds (we can use $g-1$ in place of g), we can assume that any $aug(T)$ has no more than g vertices. Therefore, $aug(T)$ has no more than g^2 arcs and can be uniquely described by a string of at most g^2 bits. To fit in one computer word,

$$g^2 \leq \log n.$$

We can traverse $aug(T)$ and compute its bitstring identifier in $O(|aug(T)|)$ time, assuming that we can (1) initialize a computer word to 0, and (2) set a bit in a computer word, both in $O(1)$ time. This further assumes that vertices in T are numbered from 1 to $|T|$, where $|T|$ is the number of vertices in T. As part of the DFS of G, we can assign secondary DFS numbers to each v, relative to $root(micro(v))$, satisfying this labeling constraint. The total time to generate bitstring identifiers is thus

$$O\left(\sum_{\text{microtree } T} |aug(T)|\right). \tag{1}$$

Since each vertex (respectively, arc) in G can be attributed to one vertex (respectively, arc) in exactly one augmented graph, and there is one extra root vertex for each augmented graph, Expression (1) $= O(n + m)$.

When first encountering a particular $aug(T)$, we can use any naive dominators algorithm to compute its immediate dominators in $\text{poly}(g)$ time. Then we can store the values for $iidom(v)$, for each $v \in aug(T)$, in table I in time $O(|aug(T)|)$. In the worst case, we would have to memoize all the IIDOM values for all possible distinct graphs on g or fewer vertices. There are about 2^{g^2} such graphs, so the total time is $O(2^{g^2}\text{poly}(g))$, inducing the constraint

$$2^{g^2}\text{poly}(g) \leq n.$$

A simple analysis shows that if $g = O(\log^{1/3} n)$, then using memoization, we can compute all needed IIDOM values in $O(n + m)$ time.

5.2 Computation of PXDOMS

Step (1) in the computation, the initial labeling of a vertex v, processes each vertex and arc in G once throughout the labelings of all vertices v. Additionally, Step (1) performs at most one eval operation, on a trivial microtree root, per arc in G.

Step (2) can be implemented by computing the strongly connected components (SCCs) of the subgraph of G induced by the microtree T, initially assigning each vertex in each SCC the minimum label among all the vertices in the SCC, and then pushing the labels through the SCCs in topological order. Computing SCCs can be done in linear time [Tarjan 1972], as can the topological processing of the SCCs.

Step (3) links $root(T)$ once for each trivial microtree T.

Thus, the time to compute the PXDOMS, summed over all the microtrees, is $O(m + n)$ plus the time to perform at most $n - 1$ link and m eval operations. We analyze the link-eval time in Section 6.

5.3 Computation of IDOMS

We implement the bucket associated with each vertex by a linked list. For each $v \in D$, Process(v) takes constant time to look up $iidom(v)$ and either assign $idom(v)$ or place v into $bucket(pxdom(v))$.

To process a vertex v in $bucket(pxdom(v))$ requires constant time plus the time to perform eval on $p_D(root(micro(v)))$. Each vertex appears in at most one bucket, so processing the buckets takes time $O(n)$ plus the time to do at most n evals on trivial microtree roots. (Since $pxdom(v) \notin micro(v)$, only trivial microtree roots have buckets.)

Again, we perform $link(v)$ only on trivial microtree roots, so the total time taken by `IDOM` is $O(m+n)$ plus the link-eval time.

5.4 Summary

By the above analysis, the total time required to compute immediate dominators in a flowgraph G with n vertices and m arcs is $O(m+n)$ plus the time to perform the links and evals on D. We next prove that since we do links and evals only on trivial microtree roots, the total link-eval time is $O(m+n)$ for an appropriate choice of the parameter g.

6. DISJOINT SET UNION WITH BOTTOM-UP LINKING

Recall that link and eval are based on disjoint set union, yielding the $\alpha(m,n)$ term in the LT time bound. Here we show that restricting the tree to which we apply links and evals to have few leaves results in the corresponding set union operations requiring only linear time.

Let U be a set of n vertices, initially partitioned into singleton sets. The sets are subject to the standard disjoint set union operations.

union(A,B,C). A, B, and C are the names of sets; the operation unites sets A and B and names the result C.

find(u). Returns the name of the set containing u.

It is well known [Tarjan and van Leeuwen 1984] that $n-1$ unions intermixed with m finds can be performed in $O(m\alpha(m,n)+n)$ time. The sets are represented by trees in a forest. A union operation links the root of one tree to the root of another. Operation *find*(u) traces the path from u to the root of the tree containing u. By linking the smaller tree as a child of the root of the larger tree during a union and compressing the path from u to the root of the tree containing u during *find*(u), the above time bound is achieved.

We show that given sufficient restrictions on the order of the unions, we can improve the above time bound. We know of no previous result based on this type of restriction. Previously, Gabow and Tarjan [1985] used a priori knowledge of the unordered set of unions to implement the union and find operations in $O(m+n)$ time. We do not require advance knowledge of the unions themselves, only that their order be constrained. Other results on improved bounds for path compression [Buchsbaum et al. 1995; Loebl and Nešetřil 1997; Lucas 1990] generally restrict the order in which finds, not unions, are performed.

Of the n vertices, designate l to be *special* and the remainder $n-l$ to be *ordinary.* The following theorem shows that by requiring the unions to "favor" a small set of vertices, the time bound becomes linear.

THEOREM 6.1. *Consider n vertices such that l are special and the remaining $n-l$ are ordinary. Let σ be a sequence of $n-1$ unions and m finds such that each* *union involves at least one set that contains at least one special vertex. Then the operations can be performed in* $O(m\alpha(m,l)+n)$ *time.*

Proof. The restriction on the unions ensures that at all times while the sequence is being processed, each set either contains at least one special vertex or is a singleton set containing an ordinary vertex. This observation can be proved by an induction on the number of unions.

The following algorithm can be used to maintain the sets. A standard union-find data structure is created containing all the special vertices as singleton sets. Recall that such a data structure consists of a forest of rooted trees built on the vertices, one tree per set. The root of a tree contains the name of the set. There is also an array, indexed by name, that maps a set name to the root of the corresponding tree. We will call this smaller data structure U' and denote unions and finds on it by *union′* and *find′*.

The ordinary vertices are kept separate. Each ordinary vertex contains a pointer that is initially null. The operations are performed as follows.

union(x, y, z). If each of x and y names a set that contains at least one special vertex, perform *union′*(x, y, z). Suppose one of x and y, say y, is a singleton set containing an ordinary vertex. Set the pointer of the ordinary vertex to point to the root of set x. Relabel that root z.

find(x). If x is a special vertex, execute *find′*(x). If x is ordinary and has a null pointer, return x. (It is in a singleton set.) If x is ordinary with a nonnull pointer to special vertex y, return *find′*(y).

The intuition is simple: unless an ordinary vertex x forms a singleton set, it can be equated to a special vertex y such that *find*$(x) =$ *find′*(y).

Each operation involves $O(1)$ steps plus, possibly, an operation on a union-find data structure U' containing l vertices. Let k be the total number of operations done on U'. Then the total running time is $O(k\alpha(k,l)+m+n)$, which is $O(m\alpha(m,l)+n)$ for $k = O(m)$. □

It is convenient to implement the above algorithm completely within the framework of a single standard union-find forest data structure, using path compression and union by size, as follows. Initially all special vertices are given weight one, and all ordinary vertices are given weight zero. Recall that the size of a vertex is the sum of the weights of its descendents, including itself.

To see that this implementation is essentially equivalent to that described in Theorem 6.1, observe the following points. First, by induction on the number of operations, an ordinary vertex is always a leaf in the union-find forest. The union-by-size rule ensures that whenever a singleton ordinary set is united with a set containing special elements, the ordinary vertex is made a child of the root of the other set. The standard find operation is done by following parent pointers to the root and then resetting all vertices on the path to point to the root. Hence any leaf vertex, and in particular any ordinary vertex, remains a leaf in the forest.

Each ordinary vertex is thus either a singleton root or contains a pointer to a special vertex, as in the proof of Theorem 6.1. Furthermore, since the ordinary vertices have weight zero they do not affect the size decisions made when uniting sets containing special vertices. A find on an ordinary vertex is equivalent to a find on its parent, which is a special vertex, just as in the proof of Theorem 6.1. The only difference is that the pointer in the ordinary vertex is possibly changed to point to a different special vertex, the root. This only adds $O(1)$ to the running time.

6.1 Bottom-Up linking

Let a sequence of unions on U be described by a rooted, undirected *union tree*, T, each vertex of which corresponds to an element of U. The edges in T are labeled zero or one; initially, they are all labeled zero. Vertices connected by a path in T of edges labeled one are in the same set. Labeling an edge $\{v, p(v)\}$ one corresponds to uniting the sets containing v and $p(v)$. The union sequence has the *bottom-up linking property* if no edge $\{v, p(v)\}$ is labeled one until all edges in the subtree rooted at v are labeled one.

COROLLARY 6.2. *Let T be a union tree with l leaves and the bottom-up linking property. Then $n-1$ unions and m finds can be performed in $O(m\alpha(m,l)+n)$ time.*

PROOF. Let the leaves of T be classed as special and all internal vertices classed as ordinary. When the union indicated by edge $\{x, p(x)\}$ occurs, all descendants of x, and in particular at least one leaf, are in the same set as x. Therefore the union sequence has the property in the hypothesis of Theorem 6.1. □

Alstrup et al. [1997] prove a variant of Corollary 6.2, with the $m\alpha(m,l)$ term replaced by $(l \log l + m)$, which suffices for their purposes. They derive the weaker result by processing long paths of unary vertices in T outside the standard set union data structure. We apply the standard set union data structure directly to T; we need only weight the leaves of T one and the internal vertices of T zero.

6.2 Application to Dominators

Recall the definition of special vertices from Section 3.2: a vertex is *special* if all of its children are roots of nontrivial microtrees.

THEOREM 6.3. *The $\Theta(n)$ links and $\Theta(m)$ evals performed during the computation of* PXDOMS *and by the algorithm* `IDOM` *require $O(n+m)$ time.*

PROOF. Consider the subtree T of D induced by the trivial microtree roots. All the links and evals are performed on vertices of T. The special vertices of D are precisely the leaves of T. We can view T as the union tree induced by the links. The links are performed bottom-up, due to the reverse DFS processing order.

As shown in Section 3.2, there are $O(n/g)$ special vertices in D and thus $O(n/g)$ leaves of T. We choose $g = \log^{1/3} n$, which suffices to compute IIDOMS in linear time. By Corollary 6.2, the link-eval time is thus $O(m\alpha(m, n/\log^{1/3} n) + n)$. The theorem follows, since $m \geq n$ and $\alpha(m,l) = O(1)$ if $m/l = \Omega(\log^{(O(1))} n)$. □

Our algorithm is completely general: it runs in linear time for any input flowgraph G. Corollary 6.2, however, implies that, implementing union-find as described above, the standard LT algorithm [Lengauer and Tarjan 1979] actually runs in linear time for all classes of graphs in which the corresponding DFS trees D have the following property: the number l of leaves of D is sufficiently sublinear in m so that $\alpha(m,l) = O(1)$.

7. IMPLEMENTATION

This section describes our implementation, which differs somewhat from our earlier description of the algorithm for efficiency reasons. The input is a flowgraph in adjacency list format, i.e., each vertex v is associated with a list of its successors. Figure 10 presents the top-level routines, which initialize the computation, perform a depth-first search and partition the DFS tree into microtrees, and compute dominators. The initialization code creates and initializes the memoization tables.

The partitioning code assigns DFS numbers, initializes the vertices and stores them in an array, *vertices*, in DFS order, computes the size of the subtree rooted at each vertex, and identifies microtrees using the subtree sizes. Each vertex is marked *Plain*, *MTRoot*, or *TrivMTRoot*, depending on whether it is a nonroot vertex in a microtree, the root of a nontrivial microtree, or the root of a trivial microtree. Also, each vertex is assigned a weight to be used by the link-eval computation: special vertices (recall that a vertex is special if all of its children are roots of nontrivial microtrees) have weight one, and ordinary vertices have weight zero. (See Lengauer and Tarjan [1979] for the implementation of link and eval.) Finally, we initialize an array, *pmtroot*, to contain $parent(v)$ for each v. This array will eventually store $parent(root(micro(v)))$ for each v. By initializing it to the vertex parent, we only have to update it for vertices in nontrivial microtrees, which we will do in ProcessMT below.

The code to compute dominators given the partitioned DFS tree differs from our earlier presentation in two ways. First, we combine the processing of vertices and buckets into a single pass, to eliminate a pass over the vertex set, as do Lengauer and Tarjan [1979]. Second, we separate the code for processing trivial microtrees from the code for processing nontrivial microtrees, which allows us to specialize the algorithm to each situation, resulting in simpler and more efficient code. These changes, which are simple rearrangements of the code, do not alter the time complexity of the algorithm.

Computedom calls ProcessV, to handle trivial microtrees, and ProcessMT, to handle nontrivial microtrees. ProcessV, shown in Figure 11, computes the XDOM and PXDOM of v, stores v in the appropriate bucket, links v to its parent, and then processes the bucket of v's parent. This code exhibits both of our changes. First, we follow the LT approach to combining the processing of vertices and buckets: we link v to p, its parent, and then process p's bucket. Immediately following the processing of v, only vertices from the subtree rooted at v are in p's bucket. Adding the link from v to p completes the path from any such vertex to p, which allows us to process the bucket. Second, we exploit that $idom(v)$ is guaranteed to be outside v's microtree, thereby eliminating a conditional expression.

ProcessMT (Figure 12) performs similar steps but is more complex, because it processes an entire microtree at once. The first step is to find the microtree's root. Since the vertices in a microtree have contiguous DFS numbers, we can find the root by searching backward from v in the vertices array for the first vertex that is marked as a nontrivial microtree root. Once we have the microtree root, we update $pmtroot(v)$ appropriately for each v in the microtree. Then we (1) compute the XDOM of each vertex in the microtree and an encoding for the augmented graph that corresponds to the microtree, (2) compute IIDOMS, (3) compute PXDOMS, and

```
Idom(Vertex root)
    Initialize computation
    Partition(root)
    status(root) ← TrivMTRoot
    Computedom(root)


Partition(Vertex v)
    Assign DFS number to v
    Mark v as visited

    samedom(v) ← NULL
    bucket(v) ← NULL
    link(v) ← NULL
    weight(v) ← 0
    label(v) ← dfsnum(v)
    status(v) ← Plain
    size(v) ← 1
    vertices[dfsnum(v)] ← v
    pmtroot(v) ← parent(v)

    For s ∈ successors(v) do
        If s has not been visited then
            parent(s) ← v
            Partition(s)
            size(v) ← size(v) + size(s)
        endif
        Add v to predecessors(s)
    done
    If size(v) > g then
        If all v's children are Plain then
            weight(v) ← 1
        Mark the Plain children of v in DFS tree with MTRoot
        status(v) ← TrivMTRoot
    endif


Computedom(Vertex root)
    For v ∈ D in reverse DFS order do
        If status(v) = TrivMTRoot then
            ProcessV(v)
        elseif v has not been processed
            ProcessMT(v)
        endif
    done

    For v ∈ D \ {root} in DFS order do
        If samedom(v) ≠ NULL then
            idom(v) ← idom(samedom(v))
        endif
    done
```

Fig. 10. Pseudocode for computing dominators.

```
ProcessV(Vertex v)
    label(v) ← dfsnum(v)
    For p ∈ predecessors(v) do
        If label(p) < label(v) then
            label(v) ← label(p)
        endif
        If dfsnum(p) > dfsnum(v) then
            evalnode ← Eval(pmtroot(p))
            If label(evalnode) < label(v) then
                label(v) ← label(evalnode)
            endif
        endif
    done

    Add v to bucket(vertices[label(v)])

    Link(v);

    For w ∈ bucket(parent(v)) do
        z ← Eval(pmtroot(w))
        If label(z) = dfsnum(parent(v)) then
            idom(w) ← parent(v)
        else
            samedom(w) ← z
        endif
        delete w from bucket(parent(v))
    done
```

Fig. 11. Pseudocode for processing trivial microtrees.

(4) process the bucket of the parent of the microtree root.

We compute XDOMS and the microtree encoding together, because both computations examine predecessor arcs. The microtree encoding is simple: two bits for each pair of microtree vertices, plus one bit for each blue arc. During this computation, we also identify a special class of microtrees: a microtree is *isolated* if the only target of a blue arc is the microtree root. We will use this information to speed the computation of PXDOMS.

The IIDOM computation uses memoization to maintain the linear time bound. To increase its effectiveness, we remove unnecessary bits and eliminate unnecessary information from the microtree encoding used to index the memoization tables. First, we remove the bits for self-loops. Second, we exploit that a blue arc to v implies that $\mathit{iidom}(v) \notin \mathit{micro}(v)$ and that none of the information about v's internal arcs is useful. In particular, since we know that the root of the microtree is always the target of some blue arc, we eliminate from the encoding the bits for arcs into the root. These changes reduce the size of the IIDOM encoding from $g^2 + g$ to $g^2 - g$ bits (from 12 bits to six, for $g = 3$). In addition to reducing the size of the encoding, we can reduce the number of populated slots in the memoization table, using the same observation. If there is a blue arc into a nonroot vertex, w, we zero the remaining bits for arcs into it, because they are irrelevant. We do not

```
ProcessMT(Vertex v)
    Find mtroot in vertices starting from v
    MT[ ] ← vertices[dfsnum(mtroot), ... , dfsnum(mtroot) + size(mtroot) − 1]
    initialize encoding
    isolated ← true

    For v ∈ MT do
        pmtroot(v) ← parent(mtroot)
        label(v) ← dfsnum(v)
        For p ∈ predecessors(v) do
            If p ∈ MT then
                Include (p, v) in encoding
            else
                Include blue arc to v in encoding
                If v ≠ mtroot then
                    isolated ← false
                endif
                If label(p) < label(v) then
                    label(v) ← label(p)
                endif
                If dfsnum(p) > dfsnum(v) then
                    evalnode ← Eval(pmtroot(p))
                    If label(evalnode) < label(v) then
                        label(v) ← label(evalnode)
                    endif
                endif
            endif
        done
    done

    iidomencoding ← reduced encoding
    If iidommemo[iidomencoding] is not defined then
            iidommemo[iidomencoding] ← Computeiidom(encoding)
    endif
    iidoms ← iidommemo[iidomencoding]

    If (isolated) then
        IsolatedPush(MT, iidoms)
    else
        Push(MT, encoding, iidoms)
    endif

    For w ∈ bucket(parent(mtroot)) do
        idom(w) ← parent(mtroot)
        delete w from bucket(parent(mtroot))
    done
```

Fig. 12. Pseudocode for processing nontrivial microtrees.

```
IsolatedPush(microtree MT, int iidoms[])
    mtroot ← MT[0]
    for v ∈ MT, v ≠ mtroot do
        idom(v) ← dfsnum(mtroot) + iidoms[secdfsnum(v)]
        label(v) ← label(mtroot)
    done
    Add mtroot to bucket(label(mtroot))
```

Fig. 13. Pseudocode for pushing in isolated microtrees.

remove these bits, because we want a fixed-length encoding. (The bits are extra only when there is a blue arc into w.) Once we compute the reduced encoding, we look it up in the memoization table to determine if futher computation is necessary to determine the IIDOMS. We use the $O(n^2)$-time bit-vector algorithm [Aho et al. 1986] augmented to exploit the blue arcs, when necessary.

The IIDOMS are expressed in terms of a DFS numbering of the augmented graph. We translate the augmented graph vertex into the corresponding vertex in the current microtree by adding its secondary DFS number to the primary DFS number of the root of the microtree.

We have implemented two forms of Push. The first, shown in Figure 13, is a simplified form that can be used for isolated microtrees. The absence of blue arcs into nonroot vertices implies that (1) the XDOM of the microtree's root is the PXDOM of all vertices in the microtree and (2) the immediate dominators of nonroot vertices will be local to the microtree (that is, $idom(v) = iidom(v)$ for all $v \neq root(micro(v))$). The root vertex has a nonlocal IDOM, so we simply add it to the bucket of its PXDOM.

The second, shown in Figure 14, handles the general case. First, we compute strongly connected components (SCCs) using memoization. In this case, the memoization is used only for efficiency. As with the IIDOM calculation, we use a reduced encoding for SCCs. The SCC encoding, which uses $g^2 - g$ bits, does not include self-loops or blue arcs, since neither affects the computation. We compute SCCs using the linear-time two-pass algorithm from Cormen et al. [1991]. Given the SCCs (either from the memoization table or by computing them), we process them in topological order: we find the minimum of the XDOMS of the vertices within the SCC and the incoming PXDOMS, and then assign this value to each vertex as its PXDOM. Given v's PXDOM and IIDOM, we either assign $idom(v)$ directly, if $iidom(v) \in MT$, or we put v into the appropriate bucket.

After pushing, we finish by processing the bucket of pmt, the parent of the microtree's root. Any vertex in the bucket must have pmt as its immediate dominator. By definition, the PXDOM of a vertex, v, in the microtree, is the minimum PXDOM along the path from pmt to v. As a result, we can skip the eval on any vertex in pmt's bucket and assign pmt as the immediate dominator directly.

8. RESULTS

This section describes our experimental results. It would be interesting to compare our algorithm (BKRW) with that of Alstrup et al. (AHLT) [1997], to judge relative constant factors, but AHLT relies on the atomic heaps of Fredman and Willard.

```
Push(microtree MT, int encoding, int iidoms[])
    mtroot ← MT[0]
    sccencoding ← reduced encoding
    If sccmemo[sccencoding] is not defined then
        sccmemo[sccencoding] ← Computescc(mtroot, encoding)
    endif

    For SCC ∈ sccmemo[sccencoding] in topological order do
        m ← min({label(v) | v ∈ SCC} ∪
                {label(u) | (u, v) ∈ A, u ∈ MT, u ∉ SCC, v ∈ SCC})
        For v ∈ SCC do
            label(v) ← m
            If iidoms[secdfsnum(v)] ∉ MT then
                Add v to bucket(label(v))
            else
                idom(v) ← dfsnum(mtroot) + iidoms[secdfsnum(v)]
            endif
        done
    done
```

Fig. 14. Pseudocode for pushing in the general case.

Atomic heaps, in turn, are composed of Q-heaps, which can store only $\log^{1/4} n$ elements given $O(n)$ preprocessing time. The atomic heap construction requires Q-heaps that store $12 \cdot \log^{1/5} n$ elements. For atomic heaps, and thus the AHLT algorithm, to run in linear time, therefore, n must exceed $2^{12^{20}}$ [Fredman and Willard 1994]. (Alternatively, one can consider AHLT to run in linear time, but with an impractically high additive constant term.) Alstrup et al. [1997] provide variants of their algorithm that do not use atomic heaps, but none of these runs in linear time. Ours is thus the only implementable linear-time algorithm, and we therefore compare our implementation of BKRW with an implementation of the LT algorithm derived from their paper [Lengauer and Tarjan 1979].

We performed two sets of experiments. The first set used flowgraphs collected from the SPEC 95 benchmark suite [SPEC 1995], using the CFG library from the Machine SUIF compiler [Holloway and Young 1997] from Harvard.[3] (Six files from the integer suite could not be compiled by Machine SUIF v. 1.1.2 and are omitted from the data.) The second set used some large graphs collected from our Lab.

We performed our experiments on one processor of an eight-processor SGI Origin 2000 with 2048MB of memory. Each processing node has an R10000 processor with 32KB data and instruction caches and a 4MB unified secondary cache. Both implementations were compiled with the Mongoose C compiler version 7.0.

We report aggregate numbers for the SPEC test set, because it contains a large number of flowgraphs. Table I reports the sizes of the flowgraphs, averaged by benchmark. Table II contains average running times for LT and for BKRW with microtree sizes of two and three. Figure 15 displays a scatter plot in which each

[3]Machine SUIF is an extension of the SUIF compiler [Amarasinghe et al. 1995] from Stanford. We used Machine SUIF version 1.1.2.

Table I. Graph Sizes, Averaged Over the Flowgraphs in Each Benchmark, for the SPEC 95 Flowgraphs

Benchmark	Number of Flowgraphs	Average Vertices	Average Arcs
CINT95 Suite			
130.li	357	9	12
129.compress	24	12	16
132.ijpeg	524	14	20
147.vortex	923	23	34
124.m88ksim	256	26	38
099.go	372	36	52
126.gcc	2013	47	69
134.perl	215	65	97
CFP95 Suite			
145.fpppp	37	19	26
102.swim	7	26	34
107.mgrid	13	27	35
103.su2cor	37	32	42
104.hydro2d	43	35	46
146.wave5	110	37	50
125.turb3d	24	52	71
110.applu	17	62	82
101.tomcatv	1	143	192

Table II. Running Times on the SPEC 95 Flowgraphs, Averaged Over the Flowgraphs in Each Benchmark. The Numbers in Parentheses Measure the Difference Between the Two Algorithms, as Computed by the Following Formula: $\frac{BKRW-LT}{LT} \cdot 100.0$. Positive Numbers Indicate That LT is Better; Negative Numbers Indicate That BKRW is Better.

Benchmark	LT	BKRW $g = 2$		BKRW $g = 3$	
CINT95 Suite					
130.li	20.01 us	33.91 us	(69.49%)	36.99 us	(84.90%)
129.compress	22.61 us	37.62 us	(66.42%)	43.84 us	(93.92%)
132.ijpeg	25.46 us	40.43 us	(58.78%)	45.86 us	(80.11%)
147.vortex	36.70 us	53.59 us	(46.02%)	61.29 us	(67.00%)
124.m88ksim	39.33 us	56.61 us	(43.93%)	63.73 us	(62.04%)
099.go	50.39 us	69.87 us	(38.66%)	79.37 us	(57.51%)
126.gcc	66.56 us	87.87 us	(32.01%)	95.61 us	(43.63%)
134.perl	89.54 us	112.23 us	(25.34%)	121.13 us	(35.28%)
CFP95 Suite					
145.fpppp	32.75 us	46.63 us	(42.37%)	49.33 us	(50.61%)
102.swim	38.53 us	53.14 us	(37.93%)	59.90 us	(55.47%)
107.mgrid	38.36 us	53.72 us	(40.02%)	60.32 us	(57.23%)
103.su2cor	46.74 us	62.06 us	(32.78%)	67.28 us	(43.95%)
104.hydro2d	49.99 us	66.71 us	(33.45%)	72.82 us	(45.68%)
146.wave5	51.71 us	68.45 us	(32.38%)	74.46 us	(44.00%)
125.turb3d	73.66 us	103.56 us	(40.60%)	110.36 us	(49.82%)
110.applu	78.21 us	99.01 us	(26.60%)	106.72 us	(36.46%)
101.tomcatv	174.60 us	210.20 us	(20.39%)	215.20 us	(23.25%)

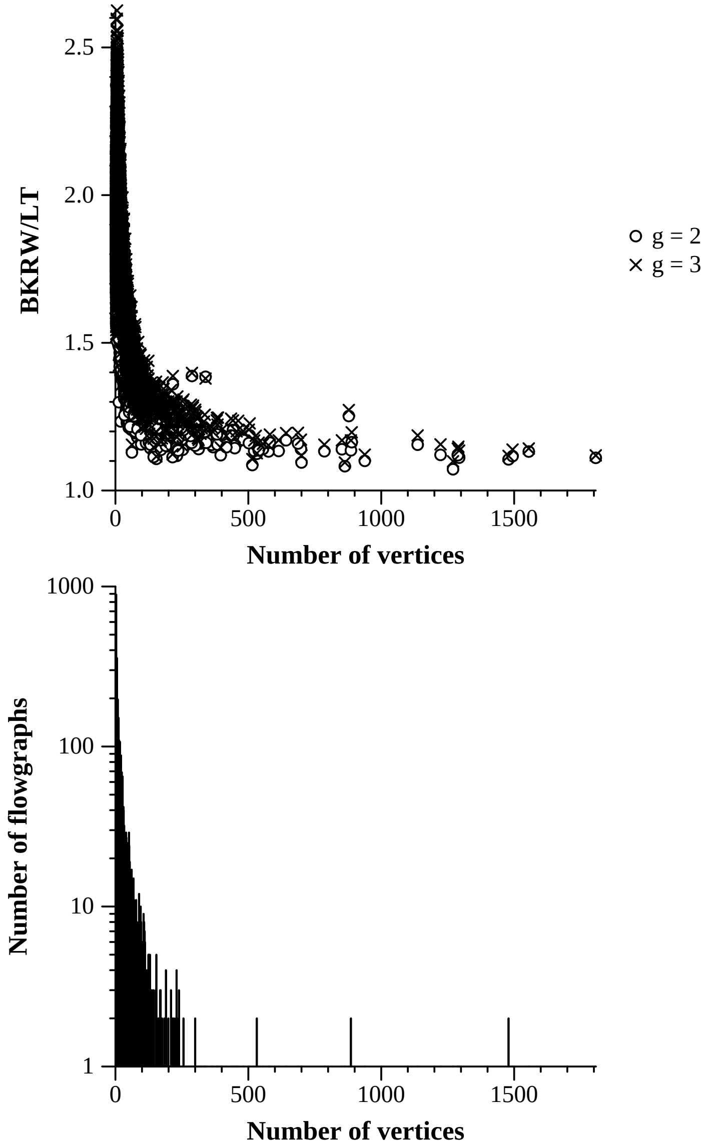

Fig. 15. Relative differences in running times of BKRW and LT (for $g = 2$ and $g = 3$). There is a point in the top plot for each flowgraph generated from the SPEC 95 benchmarks. The bottom plot displays the number of flowgraphs for each respective number of vertices. (Note that the y-axis on the bottom plot represents a logarithmic scale.)

point represents the running time of BKRW (with microtrees of size two or three) relative to LT on a single flowgraph. The plot shows that the overhead of BKRW is larger than that of LT on small graphs, but that the difference tails off to about 10% quickly. For this figure, we combined the data from the integer and floating-point suites; separating the two, as in Table II, would yield two similar plots.

Table III lists our large test graphs, which come from a variety of sources, along

Table III. Graph Sizes for the Large Test Graphs

Graph	Vertices	Arcs
ATIS	4950	515080
PW	330762	823330
NAB	406555	939984
Phone	1827332	2471847
AB1		
1024	2047	3582
2048	4095	7166
4096	8191	14334
8192	16383	28670
16384	32767	57342
32768	65535	114686
2097152	4194303	7340030
AB2		
1024	2559	5630
2048	5119	11262
4096	10239	22526
8192	20479	45054
16384	40959	90110
32768	81919	180222
2097152	5242879	11534334

with their sizes. The ATIS, NAB, and PW graphs are derived from weighted finite-state automata used in automatic speech recognition [Pereira and Riley 1997; Pereira et al. 1994] by removing weights, labels, and multiple arcs. The Phone graph represents telephone calling patterns. The augmented binary graphs (AB1 and AB2) were generated synthetically by building a binary tree of a given size (shown in the table as the graph's label) and then replacing each leaf by a subgraph. See Figure 16. The AB1 graphs use the subgraph shown in Figure 16(b), and the AB2 graphs use the subgraph shown in Figure 16(c). These graphs were designed to distinguish BKRW from LT. The subgraphs will be treated as isolated microtrees in BKRW, which means that all the nonroot vertices in a microtree will have dominators within the microtree and that the back and cross arcs will be handled cheaply (without evals) by BKRW. In particular, calls to eval related to these arcs will be avoided by BKRW, and as a result, no links in the link-eval forest will be compressed by BKRW.

We observed that, as expected, BKRW performs fewer links and evals than does LT. Running time is a more telling metric, however, and we present the running times for our experiments in Table IV. For the speech and Phone graphs, the overhead of processing the microtrees, which includes initializing the memoization tables, computing IIDOMS, computing microtree encodings, and pushing, outweighs the savings on calls to link and eval. BKRW does outperform LT on the larger augmented binary graphs. This is to be expected, since BKRW has substantially fewer calls to eval and compresses zero links for these graphs. In addition, the overhead of processing the microtrees is low, because they are all isolated. Note that the improvement of BKRW over LT decreases as the graphs get larger. The benefit gained by our algorithm is small relative to the cost due to paging, which increases as the graphs get larger.

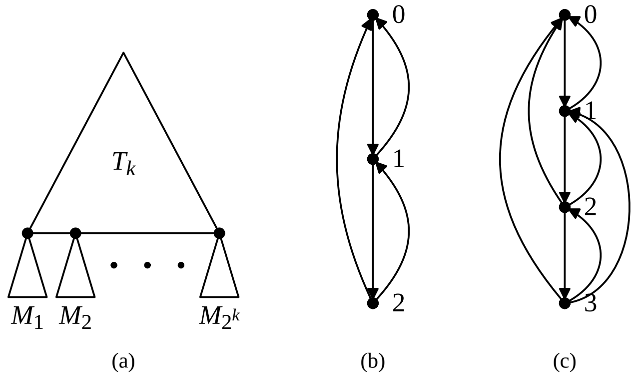

Fig. 16. (a) T_k is a k-depth binary tree. Augmented binary graph AB1 (respectively, AB2) is generated by replacing each leaf M_i with the subgraph shown in (b) (respectively, (c)).

Table IV. Running Times on the Large Test Graphs. The Numbers in Parentheses Measure the Difference Between the Two Algorithms, as Computed by the Following Formula: $\frac{BKRW-LT}{LT} \cdot 100.0$. Positive Numbers Indicate That LT is Better; Negative Numbers Indicate That BKRW is Better.

Graph	LT	BKRW			
		$g = 3$		$g = 4$	
ATIS	384.50 ms	423.38 ms	(10.11%)	427.25 ms	(11.12%)
PW	2739.00 ms	2836.25 ms	(3.55%)	2844.75 ms	(3.86%)
NAB	3127.04 ms	3195.98 ms	(2.20%)	3189.15 ms	(1.99%)
Phone	8313.62 ms	8594.75 ms	(3.38%)	8616.38 ms	(3.64%)
AB1					
1024	2.00 ms	2.00 ms	(0.00%)	2.50 ms	(25.00%)
2048	5.00 ms	5.00 ms	(0.00%)	5.00 ms	(0.00%)
4096	11.00 ms	10.00 ms	(-9.09%)	12.00 ms	(9.09%)
8192	24.38 ms	22.00 ms	(-9.74%)	23.00 ms	(-5.64%)
16384	52.62 ms	48.38 ms	(-8.08%)	48.38 ms	(-8.08%)
32768	127.38 ms	117.50 ms	(-7.75%)	117.88 ms	(-7.46%)
2097152	20188.00 ms	19499.00 ms	(-3.41%)	19498.38 ms	(-3.42%)
AB2					
1024	4.00 ms	3.12 ms	(-21.88%)	4.00 ms	(0.00%)
2048	8.00 ms	7.00 ms	(-12.50%)	7.00 ms	(-12.50%)
4096	16.00 ms	15.62 ms	(-2.34%)	16.00 ms	(0.00%)
8192	36.00 ms	34.25 ms	(-4.86%)	32.75 ms	(-9.03%)
16384	78.25 ms	74.25 ms	(-5.11%)	70.38 ms	(-10.06%)
32768	190.00 ms	182.12 ms	(-4.14%)	175.38 ms	(-7.70%)
2097152	51920.62 ms	51461.25 ms	(-0.88%)	51029.88 ms	(-1.72%)

Given the overhead that BKRW pays for computing microtree encodings and pushing and that α is very small, BKRW is surprisingly competitive, even for small flowgraphs, but these experiments suggest that LT is the algorithm of choice for most current practical applications. LT is simpler than BKRW and performs better on most graphs. BKRW performs better only on graphs that have a high percentage of isolated microtrees.

9. CONCLUSION

We have presented a new linear-time dominators algorithm that is simpler than previous such algorithms. We have implemented our algorithm, and experimental results show that the constant factors are low.

Rather than decompose an entire graph into microtrees, as in Harel's approach to dominators, our path-compression result allows microtree processing to be restricted to the "bottom" of a tree traversal of the graph. We have applied this technique [Buchsbaum et al. 1998] to simplify previous linear-time algorithms for least common ancestors, minimum spanning tree (MST) verification, and randomized MST construction. We also show [Buchsbaum et al. 1998] how to apply our techniques on pointer machines [Tarjan 1979b], which allows them to be implemented in pure functional languages.

ACKNOWLEDGMENTS

We thank Bob Tarjan, Mikkel Thorup, and Phong Vo for helpful discussions, Glenn Holloway for his help with Machine SUIF, and James Abello for providing the phone graph.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

AHO, A. V. AND ULLMAN, J. D. 1972. *The Theory of Parsing, Translation, and Compiling.* Vol. 2, *Compiling.* Prentice-Hall, Englewood Cliffs, N.J.

ALSTRUP, S., HAREL, D., LAURIDSEN, P. W., AND THORUP, M. 1997. Dominators in linear time. Manuscript available at `ftp://ftp.diku.dk/pub/diku/users/stephen/dom.ps`.

AMARASINGHE, S. P., ANDERSON, J. M., LAM, M. S., AND TSENG, C. W. 1995. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, Philadelphia, Pa.

APPEL, A. W. 1998. *Modern Compiler Implementation in C.* Cambridge University Press, Cambridge, U.K.

BUCHSBAUM, A. L., KAPLAN, H., ROGERS, A., AND WESTBROOK, J. R. 1998. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proceedings of the 30th ACM Symposium on Theory of Computing.* ACM, New York, 279–88.

BUCHSBAUM, A. L., SUNDAR, R., AND TARJAN, R. E. 1995. Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues. *SIAM J. Comput. 24,* 6, 1190–1206.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1991. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Mass.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4, 451–90.

DIXON, B. AND TARJAN, R. E. 1997. Optimal parallel verification of minimum spanning trees in logarithmic time. *Algorithmica 17*, 11–8.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The program dependency graph and its uses in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3, 319–49.

FREDMAN, M. L. AND WILLARD, D. E. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci. 48*, 533–51.

GABOW, H. N. AND TARJAN, R. E. 1985. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci. 30,* 2, 209–21.

HAREL, D. 1985. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing.* ACM, New York, 185–94.

HOLLOWAY, G. AND YOUNG, C. 1997. The flow analysis and transformation libraries of Machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop.* Stanford University, Stanford, Calif.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1,* 1, 121–41.

LOEBL, M. AND NEŠETŘIL, J. 1997. Linearity and unprovability of set union problem strategies I. Linearity of strong postorder. *J. Alg. 23*, 207–20.

LORRY, E. S. AND MEDLOCK, V. W. 1969. Object code optimization. *Commun. ACM 12,* 1, 13–22.

LUCAS, J. M. 1990. Postorder disjoint set union is linear. *SIAM J. Comput. 19,* 5, 868–82.

PEREIRA, F. AND RILEY, M. 1997. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing.* MIT Press, Cambridge, Mass.

PEREIRA, F., RILEY, M., AND SPROAT, R. 1994. Weighted rational transductions and their application to human language processing. In *Proceedings of the ARPA Human Language Technology Conference.* 249–54.

PINGALI, K. AND BILARDI, G. 1997. Optimal control dependence computation and the Roman chariots problem. *ACM Trans. Program. Lang. Syst. 19,* 3, 462–555.

PURDOM, P. W. AND MOORE, E. F. 1972. Algorithm 430: Immediate predominators in a directed graph. *Commun. ACM 15,* 8, 777–8.

SPEC. 1995. *The SPEC95 Benchmark Suite.* The Standard Performance Evaluation Corp., Manassas, Va. `http://www.specbench.org`.

TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1,* 2, 146–59.

TARJAN, R. E. 1974. Finding dominators in directed graphs. *SIAM J. Comput. 3,* 1, 62–89.

TARJAN, R. E. 1979a. Applications of path compression on balanced trees. *J. ACM 26,* 4, 690–715.

TARJAN, R. E. 1979b. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci. 18,* 2, 110–27.

TARJAN, R. E. AND VAN LEEUWEN, J. 1984. Worst-case analysis of set union algorithms. *J. ACM 31,* 2, 245–81.

Received January 1998; accepted June 1998