Finding Dominators via Disjoint Set Union

Wojciech Fraczak[1] Loukas Georgiadis[2] Andrew Miller[3]
Robert E. Tarjan[4]

October 9, 2013

Abstract

The problem of finding dominators in a directed graph has many important applications, notably in global optimization of computer code. Although linear and near-linear-time algorithms exist, they use sophisticated data structures. We develop an algorithm for finding dominators that uses only a "static tree" disjoint set data structure in addition to simple lists and maps. The algorithm runs in near-linear or linear time, depending on the implementation of the disjoint set data structure. We give several versions of the algorithm, including one that computes loop nesting information (needed in many kinds of global code optimization) and that can be made self-certifying, so that the correctness of the computed dominators is very easy to verify.

1 Introduction

A *flow graph* $G = (V, A, s)$ is a directed graph with vertex set V, arc set A, and a distinguished *start vertex* s such that every vertex is reachable from s. A vertex u *dominates* another vertex v in a flow graph G if every path from s to v contains u. The dominator relation is reflexive and transitive. Furthermore, the graph of the transitive reduction of this relation is a tree rooted at s, called the *dominator tree* D: every vertex $v \neq s$ has an *immediate dominator* $d(v) \neq v$, the parent of v in D, such that all dominators of v other than v also dominate $d(v)$. Thus D, which we represent by its parent function d, succinctly represents the dominator relation. Our goal is to find the dominator tree of a given flow graph G. See Figure 1.

Dominators have a variety of important applications, notably in optimizing compilers [2, 10] but also in many other areas [3, 6, 17, 18, 25, 27, 29, 30]. Thus it is not surprising that considerable effort has gone into devising fast algorithms for finding dominators. In 1979,

[1]Université du Québec en Outaouais, Gatineau, Québec, Canada.

[2]Department of Computer Science & Engineering, University of Ioannina, Greece. E-mail: loukas@cs.uoi.gr.

[3]Benbria Corporation, Ottawa, Ontario, Canada.

[4]Department of Computer Science, Princeton University, Princeton, NJ, and Microsoft Research Silicon Valley. E-mail: ret@cs.princeton.edu. Research at Princeton University partially supported by NSF grant CCF-0832797.

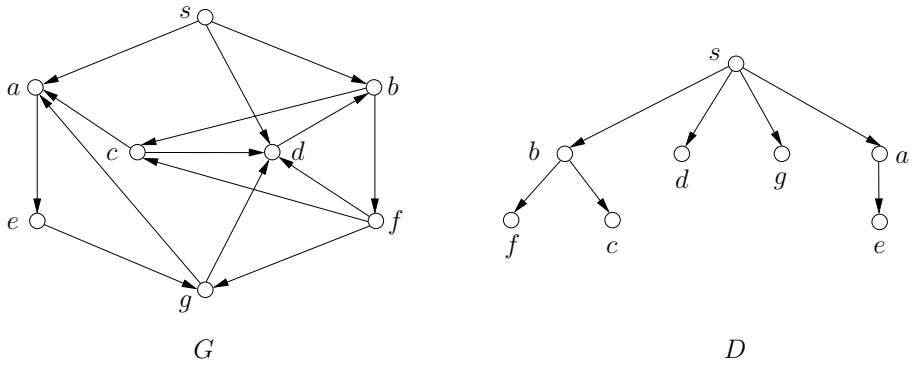

Figure 1: A flow graph and its dominator tree.

Lengauer and Tarjan [28] presented an algorithm, the *LT algorithm*, that is fast both in theory and in practice [19, 24] and that has been widely used: Lengauer and Tarjan included pseudo-code for the complete algorithm in their paper. The algorithm has one conceptually complicated part, a data structure for computing minima on paths in a tree [38]. The running time of the LT algorithm depends on the implementation of this data structure: with a simple implementation, the LT algorithm runs in $O(m \log n)$ time on an n-vertex, m-arc graph; with a sophisticated implementation, it runs in $O(m\alpha(n, m/n))$ time. Here α is a functional inverse of Ackermann's function defined as follows: For natural numbers k and j, let $A(k, j)$ be defined recursively by $A(0, j) = j + 1$, $A(k, 0) = A(k - 1, 1)$ if $k > 0$, and $A(k, j) = A(k - 1, A(k, j - 1))$ if $j, k > 0$. Let $\alpha(n, d) = \min\{k > 0 \mid A(k, \lfloor d \rfloor) > n\}$. Function α grows extremely slowly; it is constant for all practical purposes. Later work produced more-complicated but truly linear-time variants of the LT algorithm [4, 7, 8, 20].

In practice, the simple version of the LT algorithm performs at least as well as the sophisticated version, in spite of the smaller worst-case time bound of the former [19, 24]. The complexity of the underlying data structure has led researchers to seek both simpler fast algorithms [9, 15, 24, 32] and ways to certify the correctness of the output of a dominator-finding algorithm [21, 22, 23, 41]. Notable results related to our work are the following.

Ramalingam and Reps [33] gave an algorithm for finding dominators in an acyclic graph by computing nearest common ancestors (nca's) in a tree that grows by leaf additions. For this version of the nca problem, Gabow [12] gave an $O(m)$-time RAM algorithm, and Alstrup and Thorup [5] gave an $O(m \log \log n)$-time pointer machine algorithm. These algorithms give implementations of the Ramalingam-Reps algorithm that run in $O(m)$ time on a RAM and $O(m \log \log n)$ time on a pointer machine, respectively. Later, Ramalingam [32] gave a reduction of the problem of finding dominators in a general graph to the same problem in an acyclic graph. Although he did not mention it, his reduction is an extension of the algorithm of Tarjan [37] for finding a loop nesting forest. In addition to simple lists and maps, his reduction uses a data structure for the "static tree" version of the disjoint set union problem [16, 36]. There are simple, well-known disjoint-set data structures with an inverse-Ackermann-function amortized time bound per operation [36, 40]. Use of any of these in Ramalingam's reduction results in an $O(m\alpha(n, m/n))$ running time. Furthermore static tree disjoint set union has an $O(m)$-time RAM algorithm [16]. Use of this in Ramalingam's reduction gives an $O(m)$-time RAM implementation. Combining Ramalingam's

reduction with the Ramalingam-Reps algorithm for acyclic graphs gives an algorithm for finding dominators that runs in $O(m)$ time on a RAM or $O(m \log\log n)$ time on a pointer machine, depending on the implementation.

Georgiadis and Tarjan [22, 23] developed methods for making a dominator-finding algorithm *self-certifying*, by adding the computation of a *low-high order*. One of their methods requires only simple data structures and a loop nesting forest, which can be computed by Tarjan's algorithm [37].

Recently, Gabow [15] has developed a dominator-finding algorithm that uses only simple data structures and a data structure for static tree set union, thereby eliminating the need for finding path minima in a tree or computing nca's in an incremental tree. His algorithm is based on his previous work on the minimal-edge poset [13, 14].

Our work builds on these results. We develop a dominator-finding algorithm that, like Gabow's, uses only simple data structures and a data structure for static tree set union. Our algorithm does different computations than Gabow's algorithm, and it differs from his in several other ways. We compare our algorithm with his in Section 5.

In addition to this introduction, our paper contains four sections. In Section 2 we develop a dominator-finding algorithm for the special case of acyclic graphs. As part of its initialization, the algorithm finds a spanning tree rooted at s and computes nearest common ancestors (nca's) in this tree. Any spanning tree will do, but the algorithm becomes simpler if the spanning tree is depth-first and its vertices are processed in reverse preorder. This eliminates the need to compute nca's. Section 3 extends the algorithm of Section 2 to general graphs. The extension requires the spanning tree to be depth-first, and it requires an nca computation. Section 4 describes a variant of the algorithm of Section 3 that runs Tarjan's algorithm for finding a loop-nesting forest as part of the computation. This eliminates the need to compute nca's, and it allows the algorithm to be easily extended to make it self-certifying. Section 5 contains final remarks, including a comparison of our algorithm with Gabow's. We develop our algorithm in as general a way as possible. This leaves several design decisions up to the implementer, such as whether to keep the several passes of the algorithm separate or to combine them.

Our paper is a completely rewritten and extended version of a conference paper [11] by the first and third authors. The algorithm in that paper has a gap (discussed in Section 3) that was corrected by the second and fourth authors.

2 Finding Dominators in an Acyclic Graph

In the remainder of our paper, $G = (V, A, s)$ is a flow graph with n vertices and m arcs, D is the dominator tree of G, and d is the parent function of D. Tree D has root s and vertex set V, but it is not necessarily a spanning tree of G, since its arcs need not be in A. To simplify time bounds we assume $n > 1$, which implies $m > 0$ since all vertices are reachable from s. We assume that there are no arcs into s; such arcs do not change D. We assume that the original graph contains no multiple arcs (two or more arcs (v, w)) and no loop arc (an arc (v, v)). Such arcs can be created by the contractions done by the algorithm, but they do not affect the dominators. A graph is *acyclic* if it contains no cycle of more than one vertex; thus a loop arc is not a cycle. In an abuse of notation, we denote an arc by the

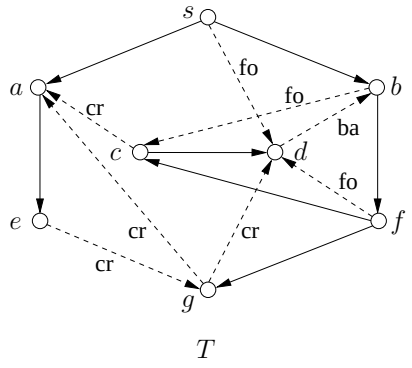

Figure 2: A spanning tree of the flow graph in Figure 1 shown with solid arcs; non-tree arcs are shown dashed and labeled by type: fo = forward arc, cr = cross arc, and ba = back arc.

ordered pair of its end vertices: even though there may be several arcs with the same ends, they are all equivalent. To make graph search efficient, we assume that each vertex v has a set of its outgoing arcs (v, w) and a set of its incoming arcs (u, v), or equivalently a set of the vertices w such that (v, w) is an arc and a set of the vertices u such that (u, v) is an arc. This allows examination of the arcs out of a vertex or into a vertex in time proportional to their number.

Let T be an arbitrary spanning tree of G rooted at s. For each vertex $v \neq s$, let $p(v)$ be the parent of v in T. Arc (v, w) of G is a *tree arc* if $v = p(w)$, a *forward arc* if v is a proper ancestor of $p(w)$ in T (by *proper* we mean $v \neq p(w)$), a *back arc* if v is a proper descendant of w in T, a *cross arc* if v and w are unrelated in T, and a *loop arc* if $v = w$. See Figure 2. Note that we allow multiple tree arcs into the same vertex w. Such arcs can be created by contractions. An alternative definition is to specify one such arc to be the tree arc into w and define the others to be forward arcs. Our algorithm does not distinguish between tree arcs and forward arcs, so either definition works, as does defining all tree arcs to be forward arcs: we keep track of the current spanning tree via its parent function, not its arcs. For any vertex v, the path in T from s to v avoids all vertices that are not ancestors of v in T, so $d(v)$ is a proper ancestor of v in T.

Our dominator-finding algorithm for acyclic graphs, Algorithm AD, begins by building an arbitrary spanning tree T rooted at s. It then repeatedly selects a vertex $v \neq s$ and contracts it into its parent in T, thereby deleting v from both G and T. Each such contraction preserves the dominators of the undeleted vertices. For each undeleted vertex v, the algorithm maintains a set $same(v)$ of vertices having the same immediate dominator as v. Just before contracting v into its parent, the algorithm either computes $d(w)$ for each vertex w in $same(v)$, including v, or identifies a non-deleted vertex x such that $d(v) = d(x)$ and adds all vertices in $same(v)$ to $same(x)$. Contractions continue until only s remains, by which time all immediate dominators have been computed.

We begin the development of Algorithm AD by discussing contractions and establishing sufficient conditions for a contraction to

(a) provide a way to compute the immediate dominator of the deleted vertex from those of the undeleted vertices, and

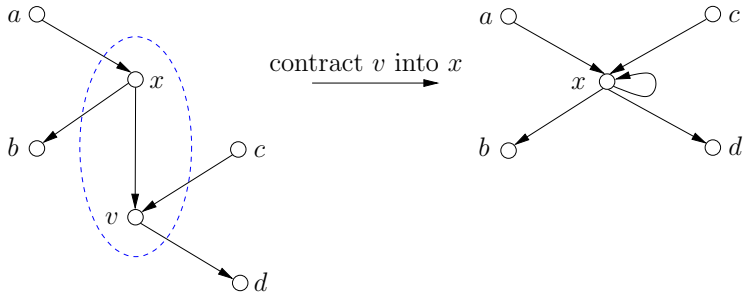

Figure 3: Vertex contraction.

(b) preserve the dominators of the undeleted vertices.

A *contraction* of vertex v *into* vertex $x \neq v$ replaces each arc end equal to v by x. That is, it replaces each arc (u, v) with $u \neq v$ by (u, x), each arc (v, y) with $y \neq v$ by (x, y), and each loop arc (v, v) by (x, x). Any arc (x, v) or (v, x) becomes a new loop arc (x, x). See Figure 3. Each contraction done by our algorithm contracts a vertex $v \neq s$ into its parent $p(v)$ in the current spanning tree T. In addition to changing the graph, the contraction changes T into a new spanning tree T' rooted at s, whose arcs are those of T not into v. The parent function p' of T' is defined for $w \notin \{s, v\}$ and is $p'(w) = p(w)$ if $p(w) \neq v$, $p'(w) = p(v)$ if $p(w) = v$.

Lemma 2.1. *Suppose G is acyclic. Let $v \neq s$, and let G' be the graph formed from G by contracting v into $p(v)$. Graph G' contains one new loop arc $(p(v), p(v))$ for each arc $(p(v), v)$ in G, and no other new loop arcs.*

Proof. Suppose G is acyclic. Every arc added by the contraction corresponds to a path in G, so such additions cannot create any cycles. Each tree arc $(p(v), v)$ in G becomes a loop arc $(p(v), p(v))$ in G'. Since G is acyclic, it contains no arc $(v, p(v))$, so no such arc becomes a loop arc in G'. □

The following two lemmas provide properties (a) and (b) of a contraction, respectively. They hold for arbitrary flow graphs. Assign the vertices of T distinct integers from 1 to n in a bottom-up order (an order such that v is numbered less than $p(v)$ for all $v \neq s$) and identify vertices by number.

Lemma 2.2. *Let $v \neq s$ be a vertex with no entering cross arc or back arc, let (u, v) be an arc with u maximum, and suppose $d(p(v)) \geq u$. If $u = p(v)$, then $d(v) = u$; otherwise, $d(v) = d(p(v))$.*

Proof. If $u = p(v)$, then every arc into v is from u. (There may be more than one, if there are multiple arcs.) Thus u dominates v. Since $d(v)$ is an ancestor of $p(v)$ in T, $u = d(v)$. Suppose on the other hand that $u \neq p(v)$. Both $d(v)$ and $d(p(v))$ are ancestors of u in T. Suppose there is a path from s to $p(v)$ that avoids $d(v)$. Adding $(p(v), v)$ to this path produces a path from s to v that avoids $d(v)$, a contradiction. Thus $d(v)$ dominates $p(v)$. Since $d(v) \neq p(v)$, $d(v)$ dominates $d(p(v))$. Suppose there is a path from s to v that avoids $d(p(v))$. Let (x, v) be the last arc on this path. Then x is a descendant of u and an ancestor

of $p(v)$ in T. Replacing (x, v) by the path in T from x to $p(v)$ gives a path from s to $p(v)$ that avoids $d(p(v))$, a contradiction. Thus $d(p(v))$ dominates v. Since $d(p(v)) \neq v$, $d(p(v))$ dominates $d(v)$. We conclude that $d(v) = d(p(v))$. □

Lemma 2.3. *Let $v \neq s$ be a vertex with no entering cross arc or back arc, let (u, v) be an arc entering v with u maximum, and suppose $d(x) \geq u$ for every descendant x of u in T. Let G' be the graph formed from G by contracting v into $p(v)$. Then the dominator tree D' of G' is D with v and the arc into v deleted.*

Proof. Since $d(x) \geq u \geq p(v)$ for every descendant x of v in T, v dominates no vertices in G except itself. Thus v is a leaf in D. Let x and w be vertices in G'. Suppose x does not dominate w in G'. Then there is a simple path P from s to w in G' that avoids x, which is either a path in G or can be converted into a path in G that avoids x as follows. Replace any new arc from $p(v)$ by a pair of old arcs, one into v and one out of v. If x is not a proper descendant of u, replace any new arc $(y, p(v))$ by the path in T from y to $p(v)$: $y \neq x$, and y must be a descendant of u and a proper ancestor of $p(v)$. If x is a proper descendant of u, replace the part of P from s to $p(v)$ by a path in G from s to $p(v)$ that avoids x. Such a path must exist since $d(p(v)) \geq u$. Thus x does not dominate w in G.

Conversely, suppose x does not dominate w in G. Then there is a simple path P from s to w in G that avoids x. If v is not on P, P is a path in G'. If v is on P, P can be converted to a path from s to w in G' that avoids x as follows. Let T' be the spanning tree in G' formed from T by the contraction, and let (y, v) and (v, z) be the arcs into and out of v on P, respectively. If $x \neq p(v)$, replace (y, v) and (v, z) on P by $(y, p(v))$ and $(p(v), z)$. Suppose $x = p(v)$. Then $u \neq p(v)$, since $u = p(v)$ implies by Lemma 2.2 that $u = x$ dominates v in G, but P contains a path from s to v that avoids x. If z is not a descendant of u, in T, then z is not a descendant of u in T'. Replace the part of P from s to z by the path in T' from s to z. If z is a descendant of u in T, replace the part of P from s to z by a path in G from s to z that avoids x, which must exist since $d(z) \geq u > p(v) = x$. Thus x does not dominate w in G'.

We conclude that the dominators of any vertex $w \neq v$ are the same in G and G'. □

Algorithm AD uses Lemmas 2.1, 2.2, and 2.3 to find dominators. It chooses contractions by marking arcs. When the last arc (u, v) into a vertex v is marked, it contracts v into $p(v)$, thereby deleting v. Each new arc created by a contraction is marked if and only if the arc it replaces is marked. Figure 4 illustrates how Algorithm AD works.

Remark: During the main loop of Algorithm AD, just before contracting a vertex v into $p(v)$, we can store a *relative dominator* $rd(v) \leftarrow p(v)$ and an indicator bit $b(v)$ for v, where we set $b(v) \leftarrow$ **true** if $u = p(v)$, and $b(v) \leftarrow$ **false** otherwise. This eliminates the need to maintain the *same*-sets but results in a three pass algorithm. The third pass processes vertices in a top-down order (the reverse of a bottom-up order) and for each vertex v it assigns $d(v) \leftarrow rd(v)$ if $b(v) =$ **true**, and $d(v) \leftarrow d(rd(v))$ otherwise. The LT algorithm computes immediate dominators in the same way, but it uses a different, more static definition of relative dominators.

Algorithm AD: Find Dominators in an Acyclic Graph, Version 1

Initialization: Find a spanning tree T of the input graph G, let p be the parent function of T, number the vertices of T from 1 to n in a bottom-up order, and identify vertices by number. Unmark all arcs of G. Assign $same(u) \leftarrow \{u\}$ for each vertex u.

Main Loop: **for** $u = 1$ **until** n **do**
while some tree or forward arc (u, v) is unmarked **do**
{ mark (u, v);
if all arcs into v are marked **then**
{ **if** $u = p(v)$ **then for** $w \in same(v)$ **do** $d(w) \leftarrow u$
else $same(p(v)) \leftarrow same(p(v)) \cup same(v)$;
contract v into $p(v)$ } }

To prove that Algorithm AD is correct, our main task is to verify that the hypotheses of Lemmas 2.2 and 2.3 hold for each contraction, and that the main loop deletes all vertices except s. The bottom-up order of vertex processing guarantees these properties. The correctness of the dominators computation follows immediately from Lemma 2.2.

Lemma 2.4. *If the original graph is acyclic, then throughout the main loop the current graph is acyclic, and every loop arc is marked.*

Proof. Contractions preserve acyclicity by Lemma 2.1. By assumption the original graph contains no loop arcs. By Lemma 2.1, any new loop arc $(p(v), p(v))$ is created by a contraction of v into $p(v)$ and replaces a former arc $(p(v), v)$. For such a contraction to occur, $(p(v), v)$ must be marked. The new loop arc inherits the mark. □

Lemma 2.5. *Throughout the main loop, each marked arc (x, y) is a tree, forward, or loop arc such that $x \leq u$.*

Proof. Each arc (u, v) marked in the main loop satisfies the lemma when it is marked, and it continues to do so since vertices are processed in increasing order. All new arcs into $p(v)$ created by contracting v into $p(v)$ are tree, forward, or loop arcs that satisfy the lemma when added and continue to do so. If (v, y) is a tree, forward, or loop arc before v is contracted into $p(v)$, then so is new arc $(p(v), y)$, and $p(v) \leq u$, so if (v, y) is marked its replacement $(p(v), y)$ satisfies the lemma and continues to do so. □

The next lemma is the main part of the correctness proof.

Lemma 2.6. *For any u, during iteration u of the main loop, every undeleted vertex $v \neq s$ has $d(v) \geq u$. At the end of the iteration, every undeleted vertex $v \neq s$ has $d(v) > u$. Each v contracted into $p(v)$ during iteration u is such that u and v satisfy the hypotheses of Lemmas 2.2 and 2.3.*

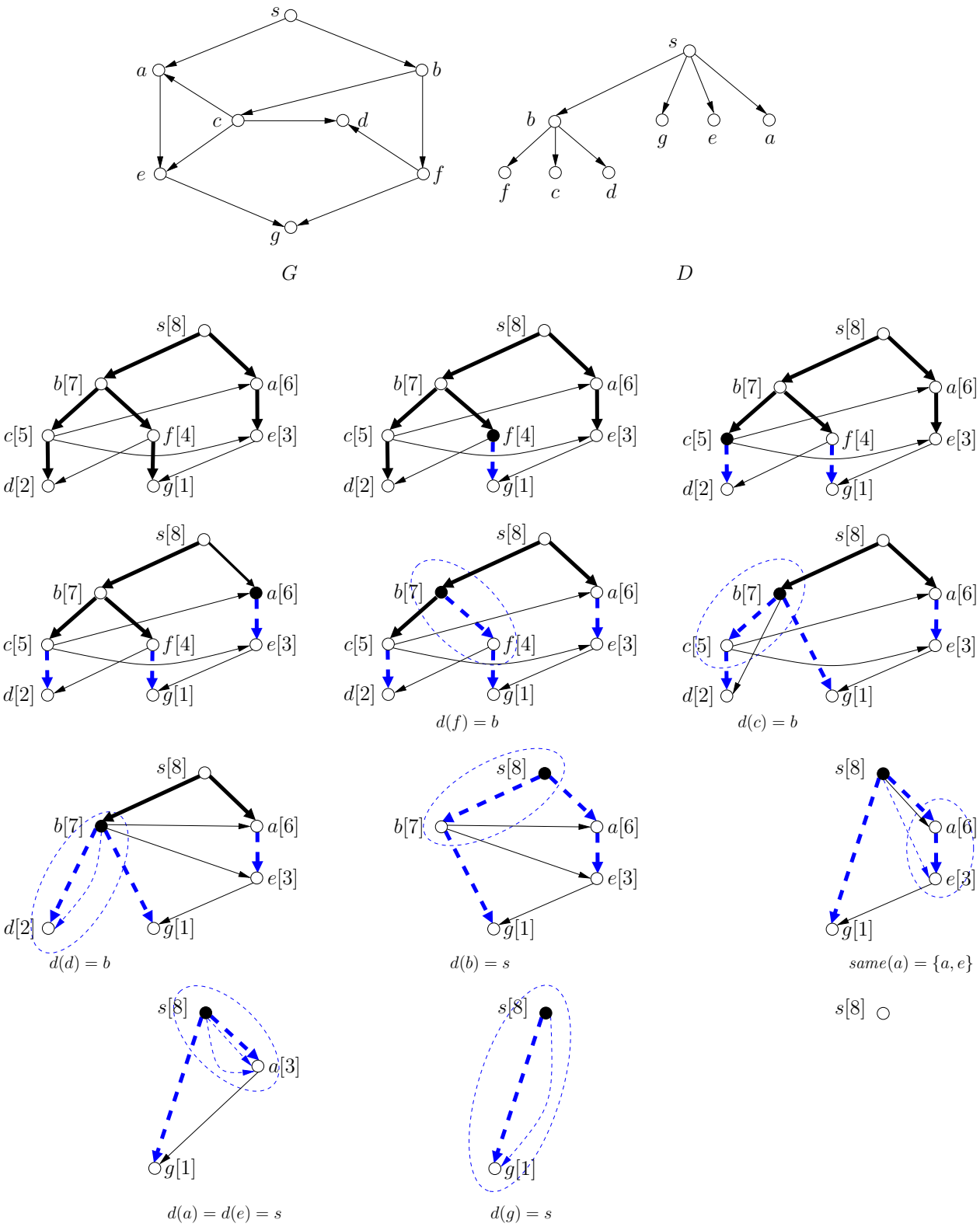

Figure 4: Execution of algorithm AD on an acyclic flow graph. Tree arcs are shown bold; the vertex u processed in the main loop is shown filled; marked arcs are shown dashed; loop arcs created by contractions are omitted.

Proof. Since $v \geq 1$ for all v, the first invariant holds at the beginning of the first iteration. If the second invariant holds at the end of an iteration, then the first invariant holds at the beginning of the next. Suppose the first invariant holds before a contraction of v into $p(v)$ in iteration u. By the first invariant, $d(x) \geq u$ for every descendant x of u in T. All arcs into v are marked, so by Lemma 2.5 they are all tree, forward, or loop arcs from vertices no less than u. The last one marked is (u, v), so all the hypotheses of Lemmas 2.2 and 2.3 hold. By Lemma 2.3, the contraction preserves the dominators of the undeleted vertices, so the first invariant holds after the contraction. By induction on the number of contractions, the first and third invariants hold during the entire iteration.

Consider a time during the iteration when there is a vertex v such that $d(v) = u$. Among such vertices, choose one, say v, such that there is no path from another such vertex to v. Such a v must exist since the current graph is acyclic by Lemma 2.4. Suppose v has an entering non-loop arc (x, v) with $x \neq u$. By the choice of v, $x = s$ or $d(x) > u$. In either case there is a path from s to v that avoids u, contradicting $d(v) = u$. We conclude that $u = p(v)$, every arc entering v is a tree arc or loop arc, and v will be deleted during iteration u once all incoming tree arcs are marked. Hence there can be no such undeleted v at the end of the iteration, making the second invariant true. □

Corollary 2.7. *At the end of the main loop, only s is undeleted.*

Proof. Once the second invariant of Lemma 2.6 holds for $u = n$, $s = n$ is the only undeleted vertex, since $d(x) \leq n$ if $x \neq s$. □

Theorem 2.8. *Algorithm AD is correct.*

Proof. The theorem is immediate from Lemmas 2.2, 2.3, 2.6, and Corollary 2.7. □

Now we develop an efficient implementation of Algorithm AD. The first step is to observe that keeping track of unmarked arcs suffices. Instead of marking arcs, we delete them. When a vertex v loses its last incoming arc, we contract it into $p(v)$. This contraction adds no new arcs into $p(v)$, and in particular creates no loops.

Theorem 2.9. *Algorithm AD remains correct if each contraction of a vertex v into its parent is preceded by deleting all arcs into v.*

Proof. Consider a run of the modified version of Algorithm AD. Rerun the original version of the algorithm, choosing the same spanning tree and the same vertex numbering, and making the same choices of arcs to mark as in the run of the modified algorithm. An induction on the number of arcs marked shows that this is always possible. The two runs compute the same immediate dominators. □

In algorithm AD with arcs deleted as described, the number of arcs never increases, and only the first end of an arc changes. Each new first end is a proper ancestor in the original spanning tree of the old first end. To keep track of the effect of contractions, we use a *disjoint set data structure* [36]. Such a data structure maintains a collection of disjoint sets, each with a name, under three operations:

make-set(x)**:** Create a new set $\{x\}$ with name x. Element x must be in no existing set.

find(x)**:** Return the name of the set containing element x.

unite(x, y)**:** Unite the sets containing elements x and y and give the new set the name of the old set containing x.

We maintain the original arc ends and the parent function p of the original spanning tree, and use the disjoint set data structure to map these to the current graph and the current tree. To initialize the sets, we perform *make-set*(v) for every vertex v. When a vertex v is contracted into its current parent, we perform *unite*$(p(v), v)$. If (x, y) is an original arc, the corresponding current arc is $(\mathit{find}(x), y)$. If p is the original parent function, the current parent of undeleted vertex $v \neq s$ is $\mathit{find}(p(v))$.

To determine when to do contractions, we maintain an integer $\mathit{total}(v)$ for each vertex $v \neq s$, equal to the number of undeleted arcs into v. When $\mathit{total}(v)$ reaches zero, we contract v into its current parent.

The last thing we need is a way to keep track of tree and forward arcs. In the main loop we add an arc to the graph only once it is guaranteed to become a tree or forward arc by the time it is a candidate for marking. For any two vertices x and y, let $\mathit{nca}(x, y)$ be their nearest common ancestor in the original spanning tree T. An original cross arc (x, y) cannot become a tree or forward arc until u in the current iteration of the main loop is at least $\mathit{nca}(x, y)$. An original tree or forward arc (x, y) has $\mathit{nca}(x, y) = x$. This means that the algorithm can only mark the current arc corresponding to (x, y) once $u \geq \mathit{nca}(x, y)$. We add (x, y) to the graph at the beginning of iteration u. Specifically, we maintain, for each undeleted vertex x, a bag (multiset) $\mathit{out}(x)$ of vertices v such that (x, v) is a current undeleted arc with $\mathit{nca}(x, v) \leq u$, where u is the current iteration of the main loop. This bag is initially empty. (We denote an empty bag by "[ ]".) At the beginning of iteration u of the main loop, for each original arc (x, v) such that $\mathit{nca}(x, v) = u$, we add v to $\mathit{out}(\mathit{find}(x))$. When contracting a vertex v into its current parent x, we replace $\mathit{out}(x)$ by $\mathit{out}(x) \cup \mathit{out}(v)$. This guarantees that during iteration u of the main loop, $\mathit{out}(u)$ will contain a copy of vertex v for each undeleted tree or forward arc (u, v), and no other vertices.

Combining these ideas produces Version 2 of Algorithm AD.

In the main loop, the *unite* operation and the union of bags implement the contraction of v into its current parent x. The first argument of the *unite* could be x, but by making it $p(v)$ we make the sequence of set operations an instance of static tree set union, in which the set of *unite* operations is known in advance but their order is determined on-line. The main loop does exactly one operation *unite*$(p(v), v)$ for each vertex $v \neq s$, but their order depends on the non-tree arcs.

Lemma 2.10. *Suppose G is acyclic. During the* ***while*** *loop in iteration u of the main loop, for each original arc (x, v) such that $\mathit{nca}(x, v) \leq u$ and whose corresponding current arc $(\mathit{find}(x), v)$ is undeleted, $\mathit{out}(\mathit{find}(x))$ contains one copy of v, and such vertices are the only vertices in out-bags.*

Proof. The proof is by induction on the number of steps in the main loop. The inner **for** loop in iteration 1 establishes the invariant for iteration 1. If the invariant holds at the end of iteration u, then the inner **for** loop in iteration $u + 1$ establishes it for iteration $u + 1$. Deletions of vertices from *out*-bags correspond to deletions of the corresponding arcs, so such

Algorithm AD: Find Dominators in an Acyclic Graph, Version 2

Initialization: Find a spanning tree T of the input graph G, let p be the parent function of T, number the vertices of T in a bottom-up order, and identify vertices by number. Let A be the arc set of G. Compute $\mathit{total}(u) \leftarrow |\{(x,u) \in A\}|$ and $\mathit{arcs}(u) \leftarrow \{(x,y) \in A \mid \mathit{nca}(x,y) = u\}$ for each vertex u.

Main Loop: **for** $u = 1$ **until** n **do**
{ $\mathit{out}(u) \leftarrow [\,]$;
$\mathit{make\text{-}set}(u)$;
$\mathit{same}(u) \leftarrow \{u\}$;
for $(x,y) \in \mathit{arcs}(u)$ **do** add y to $\mathit{out}(\mathit{find}(x))$;
while $\mathit{out}(u) \neq [\,]$ **do**
{ delete some v from $\mathit{out}(u)$; $\mathit{total}(v) \leftarrow \mathit{total}(v) - 1$;
if $\mathit{total}(v) = 0$ **then**
{ $x \leftarrow \mathit{find}(p(v))$;
if $u = x$ **then for** $w \in \mathit{same}(v)$ **do** $d(w) \leftarrow u$
else $\mathit{same}(x) \leftarrow \mathit{same}(x) \cup \mathit{same}(v)$;
$\mathit{unite}(p(v), v)$;
$\mathit{out}(x) \leftarrow \mathit{out}(x) \cup \mathit{out}(v)$ } } }

deletions preserve the invariant. The *unite* operations and bag unions done to implement contractions also preserve the invariant. □

Corollary 2.11. *During the* ***while*** *loop in iteration u of the main loop, for each undeleted tree or forward arc (u,v), there is a copy of v in $\mathit{out}(u)$, and such vertices are the only vertices in $\mathit{out}(u)$.*

Proof. If (u,v) is a current undeleted tree or forward arc, it corresponds to an original arc (x,v) such that x and v, and hence $\mathit{nca}(x,v)$, are descendants of u in the original spanning tree T. Since the vertex order is bottom-up, $\mathit{nca}(x,v) \leq u$. By Lemma 2.10, there is a copy of v in $\mathit{out}(u)$ corresponding to (x,v). Conversely, if v is in $\mathit{out}(u)$ during the **while** loop in iteration u, then by Lemma 2.10 there is an undeleted current arc (u,v) corresponding to an original arc (x,v) such that $\mathit{nca}(x,v) \leq u$. Since the vertex numbering is bottom-up, v must be a descendant of u. Thus (u,v) is a tree or forward arc. □

Theorem 2.12. *Version 2 of Algorithm AD is correct.*

Proof. The Theorem is immediate from Theorem 2.9 and Corollary 2.11. □

To implement version 2 of Algorithm AD, we represent each set $\mathit{same}(v)$ and each bag $\mathit{out}(v)$ by a singly-linked circular list. The circular linking allows unions to be done in $O(1)$ time. Since each vertex is in only one *same*-set, the lists representing these sets can be endogenous. The lists representing *out*-bags must be exogenous, since a vertex can be

in several bags, or even in the same bag several times. Alternatively, the *out*-bags can be represented by endogenous lists of the corresponding arc sets. For a discussion of endogenous and exogenous lists, see [39]. Each vertex and arc is examined $O(1)$ times. Not counting the nearest common ancestor computations in the initialization, the running time of the algorithm is $O(m)$ plus the time for $n-1$ *unite* operations and at most $m+n$ *find* operations. If *unite* and *find* are implemented using compressed trees with appropriate heuristics [36], the total time for the *unite* and *find* operations is $O(m\alpha(n, m/n))$. Furthermore the set of *unite* operations is known in advance, although their sequence is not. This makes the set operations an instance of the static tree disjoint set union problem, which is solvable in $O(m)$ time on a RAM [16]. The computation of nearest common ancestors can also be done by solving an instance of the static tree disjoint set union problem [1, 16]. We conclude that the overall running time of Version 2 of Algorithm AD is $O(m\alpha(n, m/n))$, or $O(m)$ on a RAM, depending on the implementation. The only data structure needed other than simple lists and maps is one to maintain disjoint sets.

Our final refinement of Algorithm AD, Version 3, eliminates the need to compute nearest common ancestors. We accomplish this by choosing the spanning tree and vertex order carefully. Specifically, we choose a *depth-first spanning tree* and a corresponding *reverse preorder* [34]. Such a tree and order have the property that every tree or forward arc (v, w) is has $v > w$ and every cross or back arc (v, w) has $v < w$ [34]. We insert vertices into *out* bags as follows. At the beginning of iteration u of the main loop, for every arc (x, u), we insert u into $out(find(x))$. If (x, u) is a tree or forward arc, x has not yet been processed in main loop, so $find(x) = x$, and when x is processed later, u will be in $out(x)$ as desired. If (x, u) is a cross arc, x has already been processed in the main loop. Vertex u will remain in $out(find(x))$ (which changes as $find(x)$ changes) until $nca(x, u)$ is processed, at which time Version 2 of Algorithm AD would add u to $out(find(x))$. Thus, even though the new version adds vertices to *out* bags sooner than Version 2, these early additions do not change the candidates for arc deletions, making Version 3 correct.

Lemma 2.13. *Suppose G is acyclic. During the* ***while*** *loop in iteration u of the main loop, for each original arc (x, v) such that $v \leq u$ and whose corresponding current arc $(find(x), v)$ is undeleted, $out(find(x))$ contains one copy of v, and such vertices are the only vertices in out bags.*

Proof. The proof is analogous to the proof of Lemma 2.10. □

Corollary 2.14. *During the* ***while*** *loop in iteration u of the main loop, for each undeleted tree or forward arc (u, v), there is a copy of v in $out(u)$, and such vertices are the only vertices in $out(u)$.*

Proof. If (u, v) is a current undeleted tree or forward arc, it replaces an original arc (x, v) such that x and v are descendants of u in the original spanning tree T. Since the vertex order is bottom-up, $v \leq u$. By Lemma 2.13, there is a copy of v in $out(u)$ corresponding to (x, v). Conversely, if v is in $out(u)$ during the **while** loop in iteration u, then by Lemma 2.13 there is an undeleted current arc (u, v) replacing an original arc (x, v) such that $v \leq u$. Since (u, v) is the current arc replacing (x, v), u is an ancestor of x in T. Since the vertex numbering is reverse preorder, every ancestor of x in T that is not an ancestor of v in T has

Algorithm AD: Find Dominators in an Acyclic Graph, Version 3

Initialization: Do a depth-first search of G to generate a depth-first spanning tree T of G with parent function p and to number the vertices in reverse preorder with respect to the search. Identify vertices by number. Let A be the arc set of G.

Main Loop: **for** $u = 1$ **until** n **do**
{ $total(u) \leftarrow 0$;
$out(u) \leftarrow [\,]$;
make-set(u);
$same(u) \leftarrow \{v\}$;
for $(x, u) \in A$ **do**
{ $total(u) \leftarrow total(u) + 1$; add u to $out(find(x))$ }
while $out(u) \neq [\,]$ **do**
{ delete some v from $out(u)$; $total(v) \leftarrow total(v) - 1$;
if $total(v) = 0$ **then**
{ $x \leftarrow find(p(v))$;
if $u = x$ **then for** $w \in same(v)$ **do** $d(w) \leftarrow u$
else $same(x) \leftarrow same(x) \cup same(v)$;
$unite(p(v), v)$;
$out(x) \leftarrow out(x) \cup out(v)$ } } }

number less than v. Since u is an ancestor of x and $u \geq v$, u is an ancestor of v. Thus (u, v) is a tree or forward arc. □

Theorem 2.15. *Version 3 of Algorithm AD is correct.*

Proof. The Theorem is immediate from Theorem 2.9 and Corollary 2.14. □

If we use a different vertex order in the main loop, namely postorder, then we can fold the main loop into the depth-first search that builds the spanning tree. The result is a one-pass algorithm to find dominators. Unfortunately this method must compute nca's to determine when to add vertices to *out*-bags, so it uses two disjoint set data structures concurrently, one to keep track of contractions and the other to compute nca's. We discuss this approach more fully in Section 3, since it can be used for general graphs as well.

3 Finding Dominators in a General Graph

As discussed in Section 1, Ramalingam [32] gave a reduction of the dominator-finding problem on a general graph to the same problem on an acyclic graph. His reduction uses simple data structures and static-tree disjoint set union, so it has the same asymptotic time bound as Algorithm AD. We give a streamlined version of his reduction in Section 4. By combining this reduction or his original reduction with Algorithm AD, we obtain an algorithm that

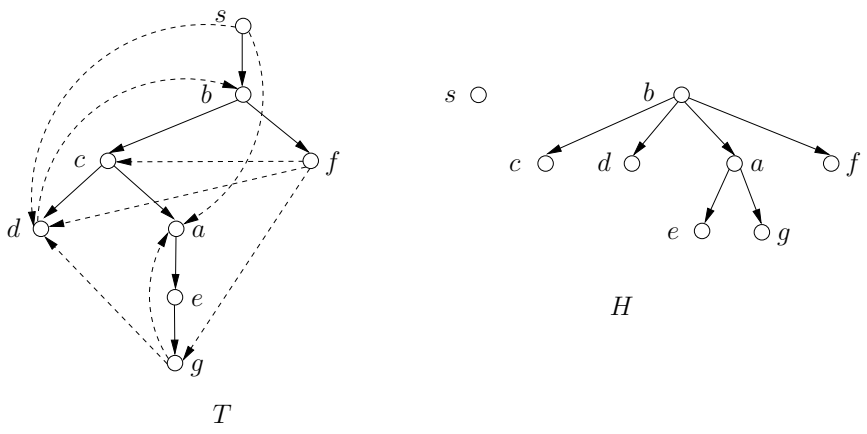

Figure 5: A depth-first spanning tree of the flow graph in Figure 1 (tree are shown solid, non-tree arcs are shown dashed), and the corresponding loop nesting forest.

finds dominators in an arbitrary graph in near-linear or linear time and uses only simple data structures and static-tree disjoint set union. Although this achieves our goal, we prefer an algorithm that is self-contained and as simple as possible. We develop such an algorithm in this section.

To explain the algorithm, we need some terminology about strongly connected subgraphs. Let T be an arbitrary spanning tree of G rooted at s, let p be the parent function of T, and suppose the vertices of T are numbered from 1 to n in a bottom-up order and identified by number. If u is any vertex, the *loop* of u, denoted by $loop(u)$, is the set of all descendants x of u in T such that there is a path from x to u containing only descendants of u in T. Vertex u is the *head* of the loop. The loop of u induces a strongly connected subgraph of G (every vertex is reachable from any other), and it is the unique maximal set of descendants of u that does so. If u and v are any two vertices, their loops are either disjoint or nested (one is contained in the other). The *loop nesting forest* H is the forest with parent function h such that $h(v)$ is the nearest proper ancestor u of v in T whose loop contains v if there is such a vertex, **null** otherwise. If u is any vertex, $loop(u)$ is the set of all descendants of u in H. An *entry* to $loop(u)$ is an arc (v, w) such that w is in $loop(u)$ but v is not; (v, w) is a head entry if $w = u$ and a non-head entry otherwise. An *exit* from $loop(u)$ is an arc from a vertex in $loop(u)$ to a vertex in $loop(h(u)) - loop(u)$. A loop has an exit if and only if it is contained in a larger loop. These definitions extend to an arbitrary spanning tree the corresponding definitions for a depth-first spanning tree [32, 37]. See Figure 5.

A loop is *reducible* if all its entries enter its head; that is, it has no non-head entries. The head of a reducible loop dominates all vertices in the loop. A flow graph is *reducible* [26, 35] if all its loops are reducible. If G is reducible, deletion of all its back arcs with respect to any spanning tree produces an acyclic graph with the same dominators as G. Thus Algorithm AD extends to find the dominators of any reducible graph.

To extend algorithm AD to general graphs, we need a way to delete vertices on cycles. For this purpose we use the *transform* operation, which adds certain arcs to the graph and then does a contraction. The operation $transform(u, v)$ requires that u be a proper ancestor of v in T and consists of the following two steps:

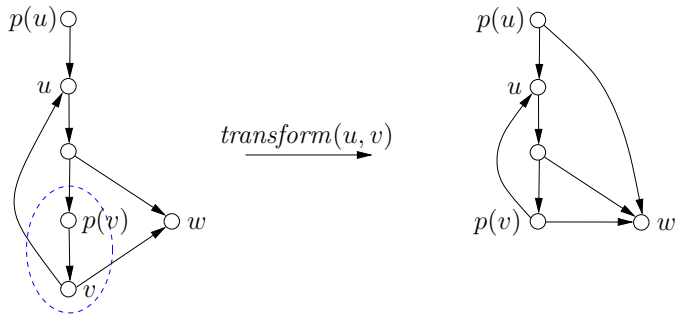

Figure 6: Transform operation.

Step 0: For each arc from v to a proper descendant $w \notin \{v, p(v)\}$ of u in T, add an arc from $p(u)$ to w.

Step 1: Contract v into $p(v)$.

See Figure 6. The effect of the transform is to replace G by a new graph G' and T by a new spanning T'.

The purpose of Step 0 is to preserve dominators. The arcs added in Step 0 play a crucial role in proving the correctness of our algorithm, but the algorithm itself does not actually keep track of such arcs. This makes the behavior of the algorithm a bit subtle.

The next two lemmas, analogous to Lemmas 2.2 and 2.3, justify the use of transforms.

Lemma 3.1. *Let u and v be distinct vertices such that $v \in loop(u)$ and $d(v) > u$. Then $d(v) = d(u)$.*

Proof. Neither u nor v dominates the other, since both have immediate dominator greater than u. Let $x \neq u$ be a vertex that does not dominate v. Then there is a path P from s to v that avoids x. If x is a proper descendant of u then x does not dominate u. Suppose x is not a descendant of u. Since $v \in loop(u)$, there is a path from v to u containing only descendants of u in T, and hence not containing x. Adding this path to P produces a path from s to u that avoids x. Thus x does not dominate u. Conversely, suppose x does not dominate u. Then there is a path P from s to u that avoids x. If x is not a descendant of u, adding to P the path in T from u to v produces a path from s to v that avoids x. If x is a descendant of u, there is a path from s to v that avoids x since $d(v) > u$. Thus x does not dominate v. It follows that u and v have the same proper dominators (dominators other than themselves). □

Lemma 3.2. *Let u and v be distinct vertices such that $v \in loop(u)$ and $d(x) > u$ for all descendants x of u in T. Let G' and T' be the graph and spanning tree formed from G and T, respectively, by doing transform(u, v). Then the dominator tree D' of G' is D with v and its incoming arc $(d(v), v)$ deleted.*

Proof. Since $d(x) > u \geq p(v)$ for every descendant x of v in T, v dominates no vertices in G except itself. Thus v is a leaf in D. Let x and w be vertices in G'. Suppose x does not dominate w in G'. Then there is a path P from s to w in G' that avoids x. Suppose P is not in G. Replace a new arc $(p(v), z)$ by old arcs $(p(v), v)$ and (v, z). If x is not a descendant of

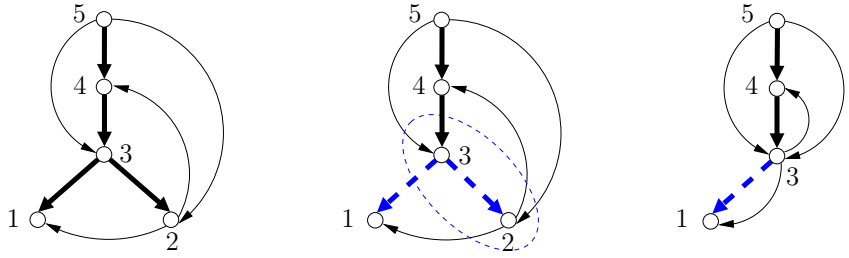

Figure 7: Counterexample to Lemma 3.2 if Step 0 is dropped from *transform*(u, v). Vertex 5 is the immediate dominator of all other vertices. Let $u = 4$ and $v = 2$. Contracting 2 into 3 without adding an arc into 1 results in 1 having immediate dominator 3 instead of 1.

u, replace a new arc $(p(u), z)$ by the path in T from $p(u)$ to v followed by old arc (v, z), and replace a new arc $(y, p(v))$ by old arc (y, v), followed by a path of descendants of u from v to u (which must exist since $v \in \mathit{loop}(u)$), followed by the path in T from u to $p(v)$. The result is a path in G from s to w that avoids x. Suppose x is a descendant of u. If P contains an arc $(p(u), z)$, replace the part of P from s to z by a path in G from s to v that avoids x, which must exist since $d(v) > u$, followed by old arc (v, z). If P contains a new arc $(y, p(v))$, replace the part of P from s to $p(v)$ by a path in G from s to $p(v)$ that avoids x, which must exist since $d(p(v)) > u$. The result is a path in G from s to w that avoids x. Thus x does not dominate w in G.

Conversely, suppose x does not dominate w in G. Then there is a simple path P from s to w in G that avoids x. If v is not on P, P is a path in G'. Suppose v is on P. Let (y, v) and (v, z) be the arcs into and out of v on P, respectively. If $x \neq p(v)$, replace (y, v) and (v, z) on P by $(y, p(v))$ and $(p(v), z)$. The result is a path in G' from s to w that avoids x. Suppose $x = p(v)$. If z is not a descendant of u, replace the part of P from s to z by the path in T' from s to z, which avoids x. If z is a descendant of u, replace the part of P from s to z by the path in T' from s to $p(u)$ followed by new arc $(p(u), z)$. The result is a path in G' from s to w that avoids x. Thus x does not dominate w in G'.

We conclude that the dominators of any vertex $w \neq v$ are the same in G and G'. □

Lemma 3.2 is false if Step 0 is dropped from *transform*(u, v), as the example in Figure 7 shows.

For transforms to suffice for deleting vertices on cycles, every cycle must be in a loop. Cycles outside of loops, and indeed without back arcs, can exist if the spanning tree T is arbitrary, but not if T is chosen carefully, specifically if T is a depth-first spanning tree. If so, every cycle contains a back arc [34], and more generally every cycle contains a vertex u that is a common ancestor of all other vertices on the cycle [34]. That is, all vertices on the cycle are in $\mathit{loop}(u)$.(See Figure 5.) What makes these statements true is that if vertices are numbered in postorder with respect to the depth-first search that generates the tree, then every arc (x, y) with x numbered less than y is a back arc [37].

Lemma 3.3. *If T is a depth-first spanning tree and T' is formed from T by contracting v into $p(v)$ or doing transform(u, v), then T' is also a depth-first spanning tree, with preorder on T' being preorder on T restricted to the vertices other than v, and postorder on T' being postorder on T restricted to the vertices other than v.*

Proof. The proof is straightforward. □

We choose T to be a depth-first spanning tree. To delete vertices on cycles, we add a **while** loop at the end of the main loop of Algorithm AD that repeatedly does transforms on pairs u, v such that (v, u) is a back arc. Just before doing such a transform it adds all vertices in $same(v)$ to $same(u)$. Once there are no such back arcs, any cycle containing u also contains a proper ancestor of u. The result is Algorithm GD, which finds dominators in a general graph. It marks arcs just like Version 1 of Algorithm AD. Each arc added in Step 0 of *transform* is unmarked, and each arc added by a contraction is marked if and only if the arc it replaces was marked.

Algorithm GD: Find Dominators in a General Graph, Version 1

Initialization: Find a depth-first spanning tree T of the input graph G, let p be the parent function of T, number the vertices of T from 1 to n in a bottom-up order, and identify vertices by number. Unmark all arcs of G. Assign $same(u) \leftarrow \{u\}$ for each vertex u.

Main Loop: **for** $u = 1$ **until** n **do**
while some tree or forward arc (u, v) is unmarked **do**
{ mark (u, v);
if all arcs into v are marked **then**
{ **if** $u = p(v)$ **then for** $w \in same(v)$ **do** $d(w) \leftarrow u$
else $same(p(v)) \leftarrow same(p(v)) \cup same(v)$;
contract v into $p(v)$ } }
while back or loop arc (v, u) exists **do**
if $v = u$ **then** mark (v, u)
else { $same(u) \leftarrow same(u) \cup same(v)$; $transform(u, v)$ }

Figure 8 illustrates how Algorithm GD algorithm works. The correctness proof of Algorithm GD is analogous to that of Algorithm AD.

Lemma 3.4. *If $transform(u, v)$ is done during iteration u of the main loop and $p(v) \neq u$, then $transform(u, p(v))$ is also done.*

Proof. The operation $transform(u, v)$ adds a back arc $(p(v), u)$. The existence of such an arc will trigger $transform(u, p(v))$ later in iteration u. □

Corollary 3.5. *If v is deleted during the second **while** loop of iteration u, then so are all ancestors of v that are proper descendants of u in T.*

Proof. The corollary follows from Lemma 3.4 by induction on the number of ancestors of v that are proper descendants of u. □

Lemma 3.6. *During the main loop of Algorithm GD, except in the middle of the second **while** loop, no vertex $x < u$ has an entering back arc.*

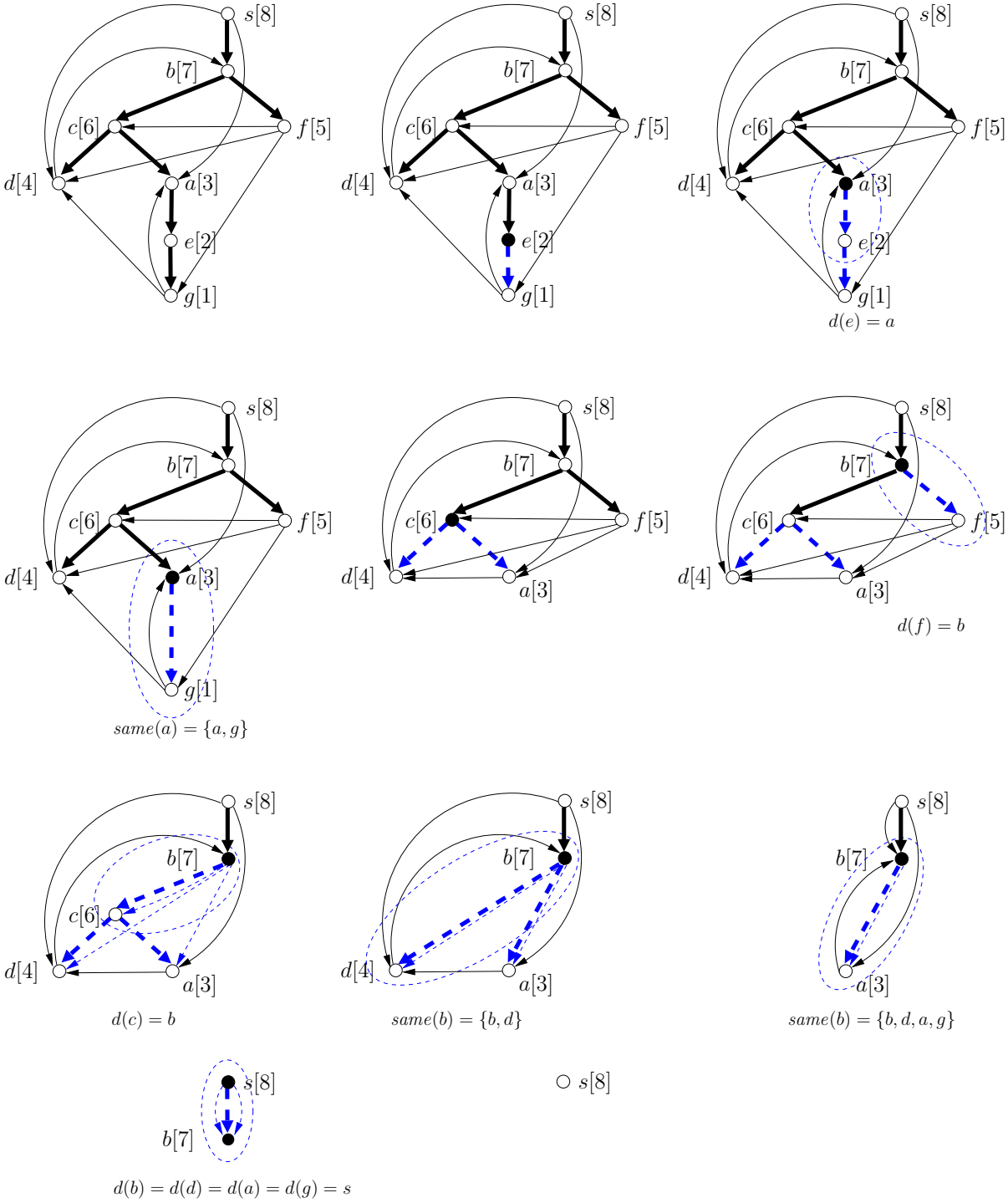

Figure 8: Execution of algorithm GD on the flow graph in Figure 1, using the depth-first spanning tree in Figure 5. Tree arcs are shown bold; the vertex u processed in the main loop is shown filled; marked arcs are shown dashed; loop arcs created by contractions are omitted.

Proof. The lemma is true before the first iteration of the main loop. Suppose the lemma is true at the beginning of iteration u. A contraction of v into $p(v)$ done during the first **while** loop of iteration u can only create a new back arc by replacing an arc (v, z) by $(p(v), z)$. If $(p(v), z)$ is a back arc, so is (v, z). Since the lemma was true before the contraction, $z \geq u$, which makes the lemma true after the contraction. By induction on the number of contractions done in the first **while** loop, the lemma is true at the end of the loop. Consider the transforms done during the second **while** loop of iteration u. Step 0 of a transform cannot add any back arcs. If *transform*(u, v) adds a back arc $(y, p(v))$, Corollary 3.5 implies that this arc will have been replaced by a loop or back arc (y', u) by the end of the **while** loop. The **while** loop cannot end while a back arc into u exists, so all such arcs become loop arcs by the end of the **while** loop. The other possibility is that *transform*(u, v) adds a back arc $(p(v), z)$. If z is a descendant of u, Corollary 3.5 implies that this back arc is replaced by a loop arc (u, u) by the end of the **while** loop. If z is not a descendant of u, $z > u$. We conclude that any back arcs into vertices less than u that are added during the second **while** loop become loop arcs before the end of the **while** loop, and any back arcs into u existing before the second **while** loop or added during the second **while** loop also become loop arcs before the end of the loop. Thus the lemma holds at the end of the **while** loop, and at the beginning of the next iteration of the main loop. By induction on the number of iterations, the lemma holds throughout the main loop. □

Lemma 3.7. *Throughout the main loop, each loop arc (x, x) has $x \leq u$. During the first **while** loop, all loop arcs (x, x) with $x < u$ are marked. At the end of the second **while** loop, all loop arcs are marked.*

Proof. By assumption, the initial graph has no loop arcs. Each new loop arc is added by a contraction of a vertex v into its parent $p(v)$. Suppose the lemma holds before the addition. The new loop arc is $(p(v), p(v))$. It replaces an arc (v, v), $(p(v), v)$, or $(v, p(v))$. Furthermore $p(v) \leq u$. Thus the first part of the lemma holds after the addition. Suppose the new loop is added during the first **while** loop. If it replaces an arc (v, v) or $(p(v), v)$, it is marked when added. If it replaces $(v, p(v))$, Lemma 3.6 implies that $p(v) = u$. Thus the second part of the lemma holds after the addition.

Suppose the new loop arc is added during the second **while** loop. Corollary 3.5 implies that either the new loop arc is (u, u) or it will be replaced by (u, u) before the second **while** loop ends. It follows that the second **while** loop cannot end while there is an unmarked loop (x, x) with $x < u$. It also cannot end while there is an unmarked loop (u, u). □

Lemma 3.8. *Throughout the main loop, each marked arc (x, y) is a tree, forward, or loop arc such that $x \leq u$.*

Proof. Each arc newly marked by the algorithm satisfies the lemma, and contractions and transforms preserve it. □

Lemma 3.9. *During any iteration of the **for** loop of the main loop, every undeleted vertex $v \neq s$ has $d(v) \geq u$; during the second **while** loop of the iteration, every undeleted vertex $v \neq s$ has $d(v) > u$. Each contraction done during the first **while** loop is such that u and v satisfy the hypotheses of Lemmas 2.2 and 2.3. Each operation transform(u, v) is such that u and v satisfy the hypotheses of Lemmas 3.1 and 3.2.*

Proof. By the proof of Lemma 2.6, (i) the first invariant holds at the beginning of the first iteration; (ii) if the first invariant holds at the beginning of the first iteration, it continues to hold throughout the first **while** loop; and (iii) by Lemma 3.7, if the first invariant holds before a contraction done during the first **while** loop, u and v satisfy the hypotheses of Lemmas 2.2 and 2.3. The proof of Lemma 2.6 also implies that if the first invariant holds at the end of the first **while** loop, then so does the second, since the proof that there is no undeleted vertex v such that $d(v) = u$ only requires that there are no cycles among the proper descendants of u, which follows from Lemma 3.6. If the second invariant holds before an operation *transform*(u, v), then u and v satisfy the hypotheses of Lemmas 3.1 and 3.2, and by Lemma 3.2 the second invariant holds after the transform. The lemma follows by induction. □

Corollary 3.10. *At the end of the main loop, only s is undeleted.*

Proof. Once the second invariant of Lemma 3.9 holds for $u = n$, s is the only undeleted vertex. □

Theorem 3.11. *Algorithm GD is correct.*

Proof. The theorem is immediate from Lemmas 3.1, 3.2, 3.9, and Corollary 3.10. □

We make Algorithm GD efficient in the same way as Algorithm AD. First we observe that the algorithm remains correct with Step 0 of *transform* deleted, so that it becomes merely a contraction. Furthermore the algorithm remains correct if all marked arcs are deleted. Arcs that are marked when added have no effect on the computation. A new arc $(p(u), w)$ added by *transform*(u, v) is unmarked when added, but if Algorithm GD marks each such arc at the beginning of iteration $p(u)$ of the main loop, the effect is the same as if the arc had never been added in the first place. Once an arc is marked, it has no further effect on the behavior of the algorithm.

Theorem 3.12. *Algorithm GD with each transform replaced by a contraction and with all marked arcs deleted is correct.*

Proof. Consider a run of Algorithm GD with transforms replaced by contractions and arcs deleted when they are marked. Rerun the original version of Algorithm GD, choosing the same spanning tree and the same vertex numbering, preferentially marking arcs added by Step 0 of *transform*, and otherwise making the same choices of arcs to mark as in the run with the transforms replaced and marked arcs deleted. An induction on the number of arcs marked shows that this is always possible. The two runs compute the same immediate dominators. □

As in Algorithm AD, we use a disjoint set data structure to keep track of contractions. We maintain the original arc ends and use the disjoint set structure to map these to the current ends: if (x, y) is an original arc, the corresponding current arc is $(\mathit{find}(x), \mathit{find}(y))$. In the main loop we add arcs to the graph at the same time as in Algorithm AD. Specifically, if (x, y) is an arc such that $\mathit{nca}(x, y) = u$, we add (x, y) to the graph at the beginning of iteration u. Arc (x, y) cannot become a tree or forward arc until x is replaced by an ancestor of u, which cannot happen until iteration u. Similarly, arc (x, y) cannot become a back or

loop arc until y is replaced by an ancestor of u. We need to keep track of arcs both into and out of vertices, so we maintain two bags of vertices for each undeleted vertex v: $in(v)$, which contains a copy of vertex x for each original arc (x, y) such that $v = \mathit{find}(y)$ and (x, y) has been added to the graph, and $out(v)$, which contains a copy of vertex y for each original arc (x, y) such that $v = \mathit{find}(x)$ and (x, y) has been added to the graph. As arcs are marked, the corresponding entries are deleted from *in*-bags and *out*-bags, but some of these deletions are done lazily, as we discuss below. If y is in $out(u)$ during iteration u, then there is a corresponding tree, forward, or loop arc $(u, \mathit{find}(y))$; if x is in $in(u)$ during iteration u, then there is a corresponding back or loop arc $(\mathit{find}(x), u)$.

Because of the lazy deletions, we need a slightly more complicated way of deciding when to do contractions in the first **while** loop than in Algorithm AD. In addition to maintaining a count $total(v)$ of the total number of unmarked arcs into vertex v, we maintain a count $added(v)$ of the number of arcs into v that have been added to the graph during the main loop but are not yet marked. When a tree or forward arc (u, v) is marked in the first **while** loop, we decrement both $total(v)$ and $added(v)$. To avoid the possibility of marking loop arcs twice (once in each **while** loop), we mark loop arcs only in the second **while** loop, and we do the marking implicitly at the end of this **while** loop. At the end of iteration u, all added arcs into u are loop arcs. Some of these may have been marked earlier, when they were tree or forward arcs. The value of $added(u)$ is the number of such arcs that have not yet been marked. To mark these implicitly, we assign $total(u) \leftarrow total(u) - added(u)$ and $added(u) \leftarrow 0$. This avoids the need for synchronized deletions of corresponding vertices from *in*-bags and *out*-bags, allowing some of these deletions to be done late or not at all.

The resulting implementation is Version 2 of Algorithm GD. The body of the loop "**while** $v \neq u$ **do**" inside the second **while** loop of the main loop adds the vertices in $same(v)$ to $same(u)$, contracts v into its parent x, and replaces v by x. The effect of this loop is to contract the entire tree path from u to the original v into u. Instead of waiting to delete a back arc (v, u) until it becomes a loop arc, the algorithm deletes it as soon as it is found but does contractions as if the arc were undeleted until it becomes a loop arc. Whereas a contraction in the first **while** loop can change only the first ends of arcs and does not add any new arcs into a vertex (since v has no incoming arcs when contracted), a contraction in the second **while** loop can change either end of an arc, and can add new arcs into x. Thus not only $in(x)$ but also $out(x)$, $total(x)$, and $added(x)$ must be updated.

Lemma 3.13. *During either **while** loop in iteration u of the main loop, for each original arc (x, y) such that $nca(x, y) \leq u$ and whose corresponding arc is unmarked, y is in $out(\mathit{find}(x))$ and x is in $in(\mathit{find}(y))$.*

Proof. The proof is straightforward by induction on the number of steps done by the main loop. □

Lemma 3.14. *Let (x, y) be an original arc that is a back arc or becomes a back arc. Let u be the iteration of the main loop during which (x, y) is added if it is originally a back arc or during which it becomes a back arc. During the second **while** loop of iteration u, vertex x will be deleted from $in(u)$ and (x, y) will become a loop arc (u, u). Vertex y will never be deleted from an out-set.*

Algorithm GD: Find Dominators in a General Graph, Version 2

Initialization: Find a depth-first spanning tree of the input graph, let p be its parent function, number its vertices in a bottom-up order, and identify vertices by number. Compute $total(u) \leftarrow |\{(x,u) \in A\}|$ and $arcs(u) \leftarrow \{(x,y) \in A \mid nca(x,y) = u\}$ for each vertex u.

Main Loop: **for** $u = 1$ **until** n **do**

```
{   out(u) ← [ ]; in(u) ← [ ];
    make-set(u); added(u) ← 0;
    same(u) ← {u};
    for (x, y) ∈ arcs(u) do
    {   add y to out(find(x)); add x to in(find(y));
        added(find(y)) ← added(find(y)) + 1   }
    while out(u) ≠ [ ] do
    {   delete some y from out(u); v ← find(y);
        if v ≠ u then
        {   total(v) ← total(v) − 1; added(v) ← added(v) − 1   }
        if total(v) = 0 then
        {   x ← find(p(v));
            if u = x then for w ∈ same(v) do d(w) ← u
            else same(x) ← same(x) ∪ same(v);
            unite(p(v), v); out(x) ← out(x) ∪ out(v)   }   }
    while in(u) ≠ [ ] do
    {   delete some z from in(u); v ← find(z);
        while v ≠ u do
        {   same(u) ← same(u) ∪ same(v);
            x ← find(p(v)); unite(p(v), v);
            in(x) ← in(x) ∪ in(v);
            out(x) ← out(x) ∪ out(v);
            total(x) ← total(x) + total(v);
            added(x) ← added(x) + added(v);
            v ← x   }   }
    total(u) ← total(u) − added(u); added(u) ← 0   }
```

Proof. The proof is straightforward by induction on the number of steps taken by the main loop. □

Lemma 3.15. *Let (x, y) be an original arc that is a tree or forward arc or becomes a tree or forward arc. If x is deleted from $in(u)$ during the second* ***while*** *loop of iteration u of the main loop, (x, y) has become a loop arc; that is, $find(x) = u$.*

Proof. The proof is straightforward by induction on the number of steps taken by the main loop. □

Lemma 3.16. *The main loop correctly maintains total and added.*

Proof. Mark an original arc (z, y) when y is deleted from some $out(u)$ such that $find(y) \neq u$ or at the end of the second **while** loop during which z is deleted from some $in(u)$, whichever comes first. When the former happens, (z, y) is a tree or forward arc. With this definition, the main loop maintains the invariant that, for each vertex v, $total(v)$ is the number of unmarked arcs into v and $added(v)$ is the number of unmarked arcs into v so far added by the inner **for** loop, as can be proved by induction on the number of steps taken by the main loop. Also, the proof of Lemma 3.7 shows that after each iteration of the main loop, all loop arcs are marked by this definition. □

Theorem 3.17. *Version 2 of Algorithm GD is correct.*

Proof. The theorem is immediate from Theorem 3.12 and Lemmas 3.13, 3.14, 3.15, and 3.16. □

Version 2 of Algorithm GD has a running time of $O(m\alpha(n, m/n))$, or $O(m)$ on a RAM, depending on the disjoint set implementation. The only data structure needed other than simple lists and maps is one to maintain disjoint sets.

Unlike Algorithm AD, Algorithm GD cannot avoid the need to compute nearest common ancestors to determine when to add vertices to *in*-bags and *out*-bags. If the vertex order is postorder, nca's are not needed for filling *in*-bags; if the order is reverse preorder, nca's are not needed for filling *out*-bags; but no order suffices for both. The crucial difficulty is that one cannot tell a priori whether a cross arc will become a back arc or a tree or forward arc. The algorithm presented in the conference version of our paper [11] does not compute nca's, and consequently is incorrect on general graphs, although it is correct for acyclic graphs.

Another way to keep track of added arcs that is perhaps more intuitive than the one used in Version 2 of Algorithm GD is to maintain the added arcs themselves instead of their ends. Then $in(v)$ and $out(v)$ become sets of arcs rather than bags of vertices. When marking an arc, we delete it from both sets containing it. We can avoid the need to implement these sets as doubly-linked lists (needed to support arbitrary deletion) by marking arcs as deleted. This approach avoids the need to maintain *added* counts as well as *total* counts. But the approach we have chosen has the advantage that it allows us to separate the acyclic part of the algorithm from the cyclic part. We shall exploit this ability in Section 4.

Like Algorithm AD, Algorithm GD can be modified to build the spanning tree and compute dominators in a single depth-first search. We conclude this section by presenting Version 3 of Algorithm GD, which does this. The algorithm uses two disjoint set data

structures concurrently: one to keep track of contractions, the other to compute nca's. We use the prefix "*n*-" to indicate the operations of the latter. To compute nca's, the algorithm uses the following streamlined version [38] of the nca algorithm of Hopcroft and Ullman [1]: During the depth-first search, when the algorithm first visits a vertex u (the preorder visit to u), it does *n-makeset*(u). When retreating along a tree arc $(p(u), u)$, it does *unite*$(p(u), u)$. This maintains the invariant that if u is the current vertex of the search and v is any visited vertex, *n-find*(v) is the nearest common ancestor of u and v in the depth-first spanning tree. Thus when the search retreats along an arc (u, v), $nca(u, v) =$ *n-find*(v).

To run the depth-first search, the algorithm maintains a bit *visited*(u) for each vertex u such that *visited*(u) is **true** if and only if u has been visited in preorder. When visiting a vertex u in postorder, the algorithm executes iteration u of the main loop.

Theorem 3.18. *Version 3 of Algorithm GD is correct.*

Proof. The Theorem follows from Theorem 3.17 and the correctness of the nca computation. □

4 Loops and Dominators

Many applications of dominators to global flow analysis also require a loop nesting forest H (defined with respect to some depth-first search tree T). We can find such an H as part of the computation of dominators. Indeed, the part of Algorithm GD that does transforms is almost the same as Tarjan's algorithm for finding H, and if we eliminate the other contractions done by Algorithm GD, we obtain a version of Tarjan's algorithm. We can turn this idea around: we find H using Tarjan's algorithm, and then find dominators by running Algorithm GD modified to choose transforms using H. This gives us Algorithm HD, which computes H as well as D. The algorithm consists of three steps: initialization, the H loop, which computes H, and the D loop, which computes D. The D loop is the main loop of Algorithm GD, modified to use H to select transforms and to process vertices in reverse preorder. The use of preorder eliminates the need to compute nca's; the use of H eliminates the need to use *in*-bags. The H loop is a depth-first search that builds a depth-first spanning tree T with parent function p, constructs a list *revpre* of the vertices in reverse preorder with respect to the search for use in the D loop, builds the loop nesting forest H with respect to T, and finds an exit *exit*(v) from each loop *loop*(v) that is contained in a larger loop. To build H, it processes each vertex u in postorder, doing a backward search from u among its descendants in T. Any bottom-up order will do, but using postorder avoids the need to compute nca's to determine which arcs to traverse in backward searches [7]. The vertices reached by the backward search are exactly those in *loop*(u). To avoid revisiting vertices that are in nested loops, the algorithm contracts each loop into its head once it is found. This is Tarjan's loop-finding algorithm [37] streamlined to eliminate the need for nca's [7]. The computation of H is essentially the same as Algorithm GD with the contractions not in transforms and the computation of D omitted. The H and D loops can use the same disjoint set data structure; the D loop reinitializes the sets to be singletons.

From H, a set of loop exits, and D, one can compute a low-high order of the vertices, which suffices to easily verify that D is correct [22]. Computing a low-high order takes an

Algorithm GD: Find Dominators in a General Graph, Version 3

```
for u ∈ V do visited(u) ← false;
dfs(s);

procedure dfs(u):
{   previsit(u);
    for (u, v) ∈ A do
    {   if visited(v) = false then { dfs(v); p(v) ← u; n-unite(p(v), v) }
        total(v) ← total(v) + 1; add (u, v) to arcs(n-find(v))   }
    postvisit(u)  }

procedure previsit(u):
{   visited(u) ← true; total(u) ← 0; arcs(u) ← [ ]; n-makeset(u)   }

procedure postvisit(u):
{   out(u) ← [ ]; in(u) ← [ ];
    make-set(u); added(u) ← 0;
    same(u) ← {u};
    for (x, y) ∈ arcs(u) do
    {  add y to out(find(x)); add x to in(find(y));
       added(find(y)) ← added(find(y)) + 1  }
    while out(u) ≠ [ ] do
    {  delete some y from out(u); v ← find(y);
       if v ≠ u then { total(v) ← total(v) − 1; added(v) ← added(v) − 1 }
       if total(v) = 0 then
       {  x ← find(p(v));
          if u = x then for w ∈ same(v) do d(w) ← v
          else same(x) ← same(x) ∪ same(v);
          unite(p(v), v); out(x) ← out(x) ∪ out(v)  }  }
    while in(u) ≠ [ ] do
    {  delete some z from in(u); v ← find(z);
       while v ≠ u then
       {  same(u) ← same(u) ∪ same(v)
          x ← find(p(v)); unite(p(v), v);
          in(x) ← in(x) ∪ in(v);
          out(x) ← out(x) ∪ out(v);
          total(x) ← total(x) + total(v);
          added(x) ← added(x) + added(v);
          v ← x  }  }
    total(u) ← total(u) − added(u); added(u) ← 0  }
```

extra $O(m)$-time pass, which we omit; [22] gives the details of this computation. If a low-high order is not needed, loop exits need not be computed, and *in*-bags of vertices (as in Algorithm GD) can be used in place of arc sets (*in-arcs*(v)) to guide the backward searches. The H loop can be extended to test whether the graph is reducible and to identify the reducible loops [31, 37].

Theorem 4.1. *Algorithm HD is correct.*

Proof. The theorem is immediate from the proof of Theorem 3.17 and the correctness of the H loop. □

The running time of Algorithm HD is $O(m\alpha(n, m/n))$, or $O(m)$ on a RAM, depending on the disjoint set implementation. The D loop can be folded into the H loop, but only at the cost of doing an nca computation to fill the *in*-bags. The result is an algorithm that consists of initialization and a single depth-first search, but that uses three disjoint set data structures concurrently, one to find nca's, one to find loops, and one to find dominators. Keeping the H and D loops separate seems likely to gives a more efficient implementation, but this is a question to be resolved by experiments.

Ramalingam's reduction uses a loop nesting forest in a different way: to transform the graph by deleting certain arcs and adding other arcs and vertices to make the graph acyclic while preserving dominators. A streamlined version of his transformation that avoids adding vertices is the following. Let G be a flow graph, let T be a depth-first spanning tree of G with parent function p, and let H be the loop nesting forest of G with respect to T. Form G' from G by (i) deleting all back arcs; (ii) for each u such that *loop*(u) and *loop*$(h(u))$ have a common non-head entry (v, w), add arc $(p(h(u)), u)$; and (ii) for each non-head entry (v, w) (into some loop), add arc (v, u), where *loop*(u) is the largest loop with entry (v, w). Since all loops containing w are nested, u is an ancestor in T of every u' such that (v, w) enters *loop*(u'), which implies that (v, w) is a non-head entry of *loop*(u). Every arc added by (ii) is a forward arc, and every arc added by (iii) is a forward or cross arc, so G' is acyclic, since it contains no back arcs. The number of arcs added by (ii) is at most $n - 2$ and by (iii) at most m, so G' has $O(m)$ arcs.

Theorem 4.2. *Graphs G and G' have the same dominators.*

Proof. Suppose x does not dominate y in G. Then there is a simple path P from s to y in G that avoids x. This path is in G' unless it contains a back arc. If it does, let (z, u') be the last back arc on P. Since P is simple, (z, u') is preceded on P by a non-head entry (v, w) to *loop*(u'). Let *loop*(u) be the largest loop entered by (v, w). Then u is an ancestor of u' in T. Also $x \neq u'$. We modify P to produce a path in G' from s to y that avoids x.

There are three cases. If x is not a proper ancestor of u' in T, then replacing the part of P from s to u' by the path in T from s to u' gives the desired path. If x is a proper ancestor of u in T, then replacing the part of P from v to u' by the new arc (v, u) followed by the path in T from u to u' gives the desired path. The third and last case is x a descendant of u and a proper ancestor of u' in T. Let u'' be the ancestor of u' such that x is a proper ancestor of u'', arc (v, w) enters *loop*(u''), and among all such u'', *loop*(u'') is largest. Vertex u'' is well-defined since u' is a candidate. Also $u'' \neq u$, so $h(u'')$ is defined. Replacing the part of P from s to u' by the path in T from s to $p(h(u''))$ followed by the new arc $(p(h(u'')), u'')$

Algorithm HD: Find Loops and Dominators in a General Graph

Initialization: **for** u in V **do** *visited*$(u) \leftarrow$ **false**; *revpre* $= [\,]$; $H \leftarrow \{\,\}$;
H **loop:** *dfs*(s);
D **loop:** **for** $u \in$ *revpre* **do**
{ *out*$(u) \leftarrow [\,]$; *make-set*(u); *added*$(u) \leftarrow 0$; *same*$(u) \leftarrow \{u\}$;
for $(x, u) \in A$ **do**
{ add u to *out*(*find*(x)); *added*(*find*(u)) $\leftarrow$ *added*(*find*(u)) $+ 1$ }
while *out*$(u) \neq [\,]$ **do**
{ delete some y from *out*(u); $v \leftarrow$ *find*(y);
if $v \neq u$ **then** { *total*$(v) \leftarrow$ *total*$(v) - 1$; *added*$(v) \leftarrow$ *added*$(v) - 1$ }
if *total*$(v) = 0$ **then**
{ $x \leftarrow$ *find*$(p(v))$;
if $u = x$ **then for** $w \in$ *same*(v) do $d(w) \leftarrow u$
else *same*$(x) \leftarrow$ *same*$(x) \cup$ *same*(v);
unite$(p(v), v)$; *out*$(x) \leftarrow$ *out*$(x) \cup$ *out*(v) } }
for $(u, z) \in H$ **do**
{ $v \leftarrow$ *find*(z);
if $v \neq u$ **then**
{ *same*$(u) \leftarrow$ *same*$(u) \cup$ *same*(v);
$x \leftarrow$ *find*$(p(v))$; *unite*$(p(v), v)$;
out$(x) \leftarrow$ *out*$(x) \cup$ *out*(v);
total$(x) \leftarrow$ *total*$(x) +$ *total*(v);
added$(x) \leftarrow$ *added*$(x) +$ *added*(v) } }
total$(u) \leftarrow$ *total*$(u) -$ *added*(u); *added*$(u) \leftarrow 0$ }

procedure *dfs*(u):
{ *previsit*(u);
for $(u, v) \in A$ **do**
{ **if** *visited*$(v) =$ **false then** { *dfs*(v); $p(v) \leftarrow u$ }
total$(v) \leftarrow$ *total*$(v) + 1$; add (u, v) to *in-arcs*(*find*(v)) }
postvisit(u) }

procedure *previsit*(u):
{ *visited*$(u) \leftarrow$ **true**; *total*$(u) \leftarrow 0$; *in-arcs*$(u) \leftarrow [\,]$;
make-set(u); add u to the front of *revpre* }

procedure *postvisit*(u):
{ **while** *in-arcs*$(u) \neq [\,]$ **do**
{ delete some (z, y) from *in-arcs*(u);
$v \leftarrow$ *find*(z);
while $v \neq u$ **do**
{ add (u, v) to H; *exit*$(v) \leftarrow (z, y)$;
$x \leftarrow$ *find*$(p(v))$; *unite*$(p(v), v)$;
in$(x) \leftarrow$ *in*$(x) \cup$ *in*(v);
$v \leftarrow x$ } } }

followed by the path in T from u'' to u' gives the desired path. We conclude that x does not dominate y in G'.

Conversely, suppose x does not dominate y in G'. Let P be a simple path in G' from s to y that avoids x. For each new arc on P, if any, we replace part or all of P containing the new arc by a path in G that avoids x. After doing all such replacements, the result is a path in G from s to y that avoids x.

Suppose P contains a new arc $(p(h(u)), u)$ such that $loop(u)$ and $loop(h(u))$ have a common non-head entry (v, w). If x is not a descendant of $p(h(u))$ and an ancestor of u, then we replace $(p(h(u)), u)$ by the path in T from $p(h(u))$ to u. If x is a descendant of $h(u)$ and a proper ancestor of u, we replace the part of P from s to u by the path in T from s to v, followed by (v, w), followed by a path in $loop(u)$ from w to u. Since v is not in $loop(h(u))$, v is not a descendant of $h(u)$, but x is, so the path in T from s to v avoids x. Since x is a proper ancestor of u, $x \notin loop(u)$, so the path in $loop(u)$ from v to u also avoids x.

Suppose P contains a new arc (v, u) such that $loop(u)$ is the largest loop with non-head entry (v, w). If x is not an ancestor of u, we replace the part of P from s to u by the path in T from s to u. If x is an ancestor of u it must be a proper ancestor of u. In this case we replace the arc (v, u) by (v, w) followed by a path in $loop(u')$ from v to u'. We conclude that x does not dominate y in G. □

One can compute G' be augmenting the computation of H appropriately. This takes only simple data structures and $O(m)$ additional time [32]. Applying Algorithm AD to G' produces the dominator tree of G. Our version of Ramalingam's transformation adds only arcs that are added by Version 1 of Algorithm GD. As we saw in Section 3, the dynamics of that algorithm allow it to keep track only of arcs formed by contractions, not the extra arcs added to preserve dominators.

5 Remarks

As mentioned in the introduction, Gabow [15] has also developed an algorithm for finding dominators that uses only simple data structures and static-tree disjoint set union. The heart of his algorithm is the computation of the minimal-edge poset [13, 14]. This computation operates on a transformed graph. The transformation increases the graph size by a constant factor, which increase its running time and storage space. Our algorithm operates entirely on the original graph. Gabow's presentation relies on his work on the minimal-edge poset; ours is self-contained. His algorithm uses an nca computation on a static tree. It also uses an algorithm for finding strongly connected components, augmented to contract components as they are found. Our algorithm does not require an algorithm for strongly connected components, and the variant presented in Section 4 does not need to compute nca's.

We believe that the techniques of Buchsbaum et al. [7] can be applied to our algorithm to make it run on a pointer machine in $O(m)$ time, but the details are not straightforward, so we leave this as an open problem. Our algorithm is simple enough that it may be competitive in practice with the LT algorithm, and we plan to do experiments to find out whether this is true and to determine the best possible practical implementation.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM Journal on Computing*, 5(1):115–32, 1976.

[2] A. V. Aho and J. D. Ullman. *Principles of Compilers Design*. Addison-Wesley, 1977.

[3] S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.

[4] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

[5] S. Alstrup and M. Thorup. Optimal algorithms for finding nearest common ancestors in dynamic trees. *Journal of Algorithms*, 35:169–88, 2000.

[6] 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*, March 2001.

[7] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

[8] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum in 27(3):383-7, 2005.

[9] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:110, 2001.

[10] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[11] W. Fraczak and A. Miller. Disjoint set forest digraph representation for an efficient dominator tree construction. In S. Arumugam and W.F. Smyth, editors, *Combinatorial Algorithms*, volume 7643 of *Lecture Notes in Computer Science*, pages 46–59. Springer Berlin Heidelberg, 2012.

[12] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 434–43, 1990.

[13] H. N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. 32th IEEE Symp. on Foundations of Computer Science*, pages 812–821, 1991. Full version: CU-CS-545-91, Dept. of Computer Science, University of Colrado at Boulder, 1991.

[14] H. N. Gabow. The minimal-set poset for edge connectivity. Unpublished manuscript, 2013.

[15] H. N. Gabow. A poset approach to dominator computation. Unpublished manuscript 2010, revised unpublished manuscript, 2013.

[16] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.

[17] L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

[18] L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

[19] L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan. Dominator certification and independent spanning trees: An experimental study. In *Proc. 12th Int'l. Symp. on Experimental Algorithms*, pages 284–295, 2013.

[20] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 862–871, 2004.

[21] L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proc. 16th ACM-SIAM Symp. on Discrete Algorithms*, pages 433–442, 2005.

[22] L. Georgiadis and R. E. Tarjan. Dominator tree certification and independent spanning trees. *CoRR*, abs/1210.8303, 2012.

[23] L. Georgiadis and R. E. Tarjan. Dominators, directed bipolar orders, and independent spanning trees. In *Proc. 39th Int'l. Coll. on Automata, Languages, and Programming*, pages 375–386, 2012.

[24] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

[25] M. Gomez-Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *29th International Conference on Machine Learning (ICML)*, 2012.

[26] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[27] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447(0):74–84, 2012.

[28] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

[29] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 115–124, 2010.

[30] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

[31] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, 1999.

[32] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002.

[33] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. 21th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 287–296, 1994.

[34] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–59, 1972.

[35] R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974.

[36] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[37] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

[38] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[39] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[40] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[41] J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012.